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

Los *asistentes de demostración* son programas que simplifican la escritura de demostraciones matemáticas, permitiendo la colaboración masiva o la verificación formal de programas. Existen muchos asistentes como Mizar, Coq e Isabelle, basados en distintas teorías. Un criterio deseable que pueden cumplir es el de De Bruijn: a partir de demostraciones en un lenguaje de alto nivel se pueden extraer demostraciones en un lenguaje núcleo fácilmente verificable. Esto elimina la necesidad de tener que confiar en la implementación del asistente, ya que las demostraciones pueden ser verificadas por un programa independiente.

En esta tesis se presenta PPA, un asistente de demostración para lógica clásica de primer orden. Cumple con el criterio de De Bruijn, generando demostraciones en el sistema lógico de *deducción natural* a partir de programas escritos en un lenguaje de alto nivel, cuyo objetivo es ser similar a cómo serían en lenguaje natural. Tiene un mecanismo principal de demostración, el `by`, que por debajo cuenta con un *solver* heurístico para lógica de primer orden que facilita la escritura de demostraciones. Está inspirado en el mecanismo análogo en Mizar.

Algunos asistentes implementan la *extracción de testigos de existencial*. Dada una demostración de $\exists x.p(x)$, se extrae un *testigo* t tal que cumpla $p(t)$. Es sencillo hacerlo sobre lógica intuicionista por su naturaleza constructiva, pero un desafío sobre lógica clásica, que no lo es. Para ello hay dos grandes categorías: directas (mediante técnicas semánticas como realizabilidad clásica [Miq11]) o indirectas (mediante traducciones a otra lógica, como intuicionista).

El aporte principal del trabajo es una implementación práctica de la extracción de testigos. PPA la implementa de forma indirecta usando la traducción de Friedman, que permite convertir demostraciones clásicas de fórmulas Π_2 de la forma $\forall y_0 \ldots \forall y_n.\exists x.\varphi$, con φ sin cuantificadores, a demostraciones intuicionistas. Se describe cómo una vez traducidas pueden ser normalizadas usando reglas de reducción bien conocidas, que se corresponden con las reglas de reducción del cálculo lambda vistas desde el isomorfismo Curry-Howard. Finalmente, de una demostración normalizada se podrá extraer un testigo. Identificamos algunos detalles en la implementación práctica de la traducción, que limitan las fórmulas para las cuales se puede usar (más allá de la limitación teórica de Π_2) y también limitan las axiomatizaciones de las teorías que se pueden hacer.

Palabras claves: asistente de demostración, lógica de primer orden, deducción natural, lógica clásica, lógica intuicionista, extracción de testigos, traducción de Friedman.

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

Proof assistants are programs that simplify the writing of mathematical proofs, enabling large-scale collaboration or the formal verification of programs. There are many proof assistants such as Mizar, Coq, and Isabelle, based on different theories. A desirable property they may have is described by the De Bruijn criterion: from proofs written in a high level language, one can extract proofs in a core, easily verifiable language. This eliminates the need to trust their implementation, as the extracted proofs can be verified by an independent program.

This thesis presents *PPA*, a proof assistant for classical first-order logic. It fulfills De Bruijn's criterion by generating proofs in the natural deduction proof system from high-level proofs, designed to resemble natural language as closely as possible. Its main proof mechanism, `by`, features an underlying heuristic solver for first-order logic, inspired by Mizar's analogous mechanism.

Some proof assistants implement *existential witness extraction*. Given a proof of $\exists x.p(x)$, they extract a *witness* t such that $p(x)$ holds. While this is straightforward in intuitionistic logic due to its constructive nature, it is challenging in classical logic, which lacks constructiveness. There are two main approaches: direct (using semantic techniques such as classical realizability [Miq11]) or indirect (via translations to another logic, such as intuitionistic).

The main contribution of this work is a practical implementation of witness extraction. PPA achieves this indirectly by using Friedman's translation, which allows converting classical proofs of Π_2 formulas of the form $\forall y_0 \dots \forall y_n.\exists x.\varphi$, with the formula φ having no quantifiers, into intuitionistic ones. We describes how, once translated, these proofs can be normalized using well-known reduction rules, which correspond to the reduction rules of lambda calculus viewed through the Curry-Howard isomorphism. Finally, a normalized proof enables the extraction of a witness. We identify some practical implementation details of the translation, which limit the formulas for which it can be used (beyond the theoretical Π_2 restriction) and also constrain the axiomatizations of theories that can be made.

Keywords: proof assistant, first-order logic, natural deduction, classical logic, intuitionistic logic, witness extraction, Friedman's translation.

AGRADECIMIENTOS

A mi director, Pablo, por toda su ayuda y paciencia a lo largo de estos años y por traerme un tema de tesis que era justo lo que buscaba. Gracias a Vero y Miguel por brindar su tiempo para ser jurados en esta tesis.

A mi viejo, por inculcarme la importancia de tener una formación universitaria.

A mi vieja, por acompañarme durante toda la cursada, bancando mis largas tardes de estudio y esperándome con la comida los días que volvía a las 10 de la noche.

A mis amigos y compañeros de cursada que me acompañaron toda la carrera: Elias, Gasti, las Guadas, Giampa, Nacho, Tropi, Vladi, Ale, Bruno, Gabi, Juli, Octo, Schuster y Delgado. Sin ustedes, no hubiera llegado hasta acá.

A Jess, por acompañar los momentos estresantes del último tiempo, y enseñarme el valor de relajar y no estar siempre desbordado con cosas.

A mis docentes, de la secundaria Chamo que me convenció de estudiar esta hermosa carrera, y los de la carrera por inspirarme a seguir adelante. Gracias a mis alumnos y alumnas por permitirme dar al menos un poco de lo que recibí, y gracias a la Universidad pública por hacerlo todo posible.

Índice general

1. INTRODUCCIÓN

Los asistentes de demostración son herramientas que facilitan la escritura y chequeo de demostraciones en una computadora. Se pueden usar para formalizar teoremas y realizar verificación formal de programas, entre otros. A diferencia de escribir demostraciones y chequearlas manualmente, el uso de asistentes permite la colaboración a gran escala: no es necesario confiar ni revisar a detalle las demostraciones que hace el resto del equipo, alcanza con que el asistente las considere válidas. También facilitan la generación de demostraciones con técnicas de inteligencia articial, que muchas veces usan razonamientos lógicos erróneos difíciles de detectar a mano. Pero se puede delegar su verificación al asistente.

Trabajan con distintas *teorías*. Por ejemplo, el asistente Mizar [Miz] con lógica de primer orden, Coq [Coq] con teoría de tipos (cálculo de construcciones o CoC) y Agda [Agd] también (teoría unificada de tipos dependientes, basada en teoría de tipos de Martin-Löf).

Una propiedad deseable cumplida por muchos asistentes es el **criterio de De Bruijn** [BW05]. En principio, para estar seguros de que una demostración es correcta sería necesario confiar en la implementación del asistente, que puede ser muy compleja. Un asistente se dice que cumple con el criterio si construye una demostración en un formato elemental, sencillo, que pueda ser chequeada por un programa independiente, escrito por cualquiera que desconfíe de la implementación original del asistente.

En este trabajo diseñamos e implementamos un asistente de demostración *PPA* (*Pani's proof assistant*). Trabaja sobre teorías de lógica clásica de primer orden. Cumple con el criterio de De Bruijn porque *certifica* las demostraciones en el lenguaje de alto nivel PPA, generando demostraciones de bajo nivel en el sistema de deducción natural de Gentzen [Gen35]. Su sintaxis busca ser lo más parecida posible al lenguaje natural, inspirada en el *mathematical vernacular* de Freek Wiedijk [Wie]. En él, Wiedijk compara y combina la sintaxis de Mizar e Isar [Wen99] generando un lenguaje núcleo más simple. La sintaxis de PPA también se puede entender como una notación de deducción natural en el estilo de Fitch [PH24], en donde las demostraciones son esencialmente listas de fórmulas, con las que aparecen más adelante siendo consecuencia de las que aparecieron antes.

No es un demostrador automático de teoremas, pero su mecanismo de demostración principal, el **by**, incluye un pequeño demostrador heurístico para lógica de primer orden y completo para proposicional, que simplifica la escritura de demostraciones. Está inspirado en el mecanismo análogo de Mizar [Wie02].

Algunos asistentes de demostración implementan la *extracción de testigos de existenciales*. Dada una demostración de $\exists x.p(x)$, encuentran un *testigo* t tal que cumpla $p(t)$. Para ello, es deseable que las demostraciones sean **constructivas**: una demostración de $\exists x.p(x)$ nos debe decir cómo encontrar a un objeto t que cumpla con $p(t)$. PPA usa lógica clásica, que no siempre es constructiva por sus principios de razonamiento clásicos. Por ejemplo, con el principio del tercero excluido o LEM (*law of excluded middle*), siempre vale $A \vee \neg A$, y podemos realizar una demostración que al usarlo no explicite cual de las dos es la que vale. Los asistentes como Coq aseguran el constructivismo mediante el uso de **lógica intuicionista**, que siempre es constructiva por su naturaleza.

En general se pueden encontrar dos grandes categorías de extracción de testigos para lógica clásica. Las **directas**, mediante semánticas operacionales de cálculos λ clásicos, como *realizabilidad clásica* [Miq11], y las **indirectas**, que traducen las demostraciones a otra lógica, como la intuicionista.

En este trabajo, la extracción se hace de forma *indirecta* en dos pasos. Primero hacemos uso de la **traducción de Friedman** [Sel92], que permite traducir una demostración clásica a una intuicionista para cierta clase de fórmulas: las Π_2, de la forma $\forall x_1 \ldots \forall x_n.\exists y.\varphi$ donde

φ es una fórmula sin cuantificadores. Luego, *normalizamos* la demostración intuicionista, y de su forma normal extraemos el testigo del existencial. Hasta donde sabemos, no hay asistentes de demostración para lógica clásica que implementen la extracción de testigos basándose en estas técnicas. Este es el aporte principal del trabajo.

1.1. Lógica de primer orden

A continuación se presentan definiciones preliminares de lógica de primer orden. Suponemos dados:

- Un conjunto infinito numerable de **variables** $\{x, y, z, \dots\}$.
- Un conjunto infinito numerable de **símbolos de función** $\{f, g, h, \dots\}$.
- Un conjunto infinito numerable de **símbolos de predicado** $\{p, q, r, \dots\}$.

Def. 1 (Términos)**.** Los términos están dados por la gramática

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

Def. 2 (Fórmulas)**.** Las fórmulas están dadas por la gramática

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

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

Notación. Usamos

- $a, b, c, \dots$, $A, B, C, \dots$ y $\varphi, \psi, \dots$ para referirnos a fórmulas.
- $t, u, \dots$ para referirnos a términos

Def. 3 (Variables libres y ligadas)**.** Las variables pueden ocurrir libres o ligadas. Los cuantificadores son los que las ligan. El conjunto de variables libres de un término o fórmula X se nota $fv(X)$ y se define recursivamente de la siguiente manera.

- 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

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

1.2. Arquitectura de ppa

A lo largo del trabajo, usaremos la sigla "PPA" para referirnos a dos cosas separadas: PPA el lenguaje para escribir demostraciones, y `ppa` la herramienta que implementa el lenguaje y la extracción de testigos, implementada en **Haskell**. Funcionalmente tiene la siguiente arquitectura representada gráficamente en la Figura 1.1 (Representación gráfica de la arquitectura funcional de `ppa`).

- El usuario escribe una demostración en alto nivel en el lenguaje PPA.
- Puede *chequear* la demostración de alto nivel, que primero se **certifica** generando una demostración en deducción natural de bajo nivel (el "certificado"), para que luego un módulo independiente chequee su correctitud. Si el usuario escribe una demostración inválida, debería fallar el certificador y reportar un error al usuario. De todas formas, el certificado es verificado por el chequeador de deducción natural como mecanismo de *fallback*.
- Si es una demostración de un existencial, el usuario puede optar por *extraer un testigo*: primero se **traduce** la demostración de clásica a intuicionista, y luego se reduce hacia su formal normal de la cual se puede tomar el testigo.

Describimos los tipos de las principales funciones que implementan las distintas etapas de la herramienta.

- El tipo `type Result a = Either String a` es la mónada principal que se usa para devolver resultados o errores.
- `certify :: Program -> Result Context`. Implementada por el módulo `PPA.Certifier`. Donde `Program` es el tipo de los programas escritos en PPA (listas de axiomas y teoremas) y `Context` el tipo de los certificados (lista de axiomas y teoremas junto con su demostración en deducción natural).
- `check :: Env -> Proof -> Form -> CheckResult`. Implementada por el módulo `ND.Checker`. Chequea una demostración de una fórmula asumiendo un entorno o contexto de demostración. El tipo `Env` representa los contextos en deducción natural

(asociación de etiquetas a fórmulas), `Proof` las demostraciones en deducción natural, `Form` las fórmulas de primer orden y `CheckResult` es una sofisticación de `Result` específica para el chequeo.

- `translateFriedman :: Proof -> FormPi02 -> (Proof, R)`. Implementada por el módulo `Extractor.Translator.Proof`. Traduce una demostración clásica de una fórmula Π_2 generando una demostración intuicionista, usando la traducción de Friedman parametrizada por una fórmula `R`. Esta parametrización se explicará en detalle más adelante, en el Capítulo 5. El tipo `FormPi02` es un refinamiento del de las fórmulas, `Form`, que solo representa las de la clase Π_2.
- `reduce :: Proof -> Proof`. Reduce una demostración a su forma normal. Implementada en el módulo `Extractor.Reducer`.

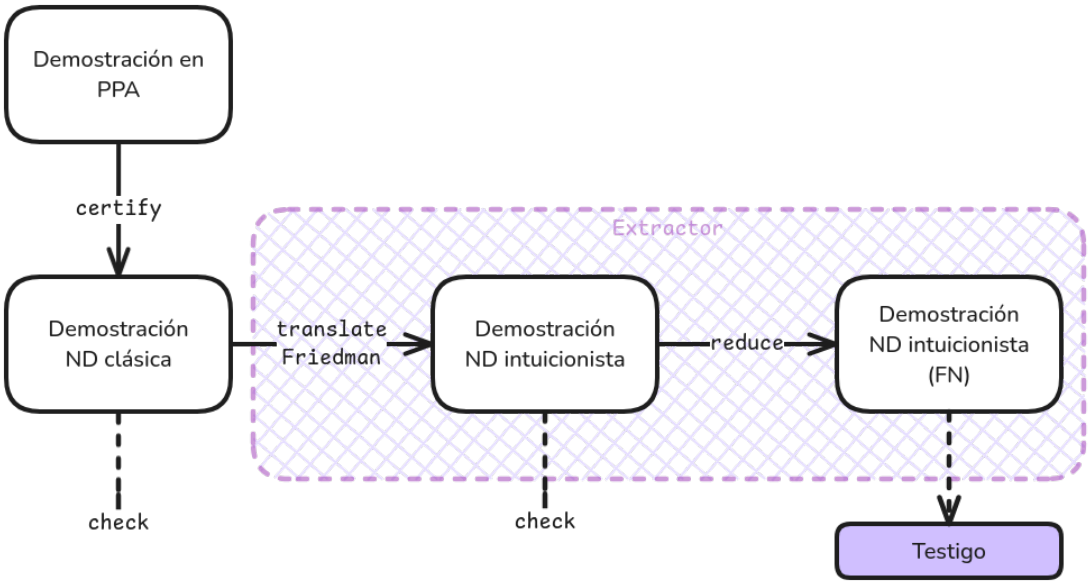

Fig. 1.1: Representación gráfica de la arquitectura funcional de `ppa`

Cada caja corresponde a las versiones de una demostración durante el flujo del programa, y las flechas están etiquetadas por la función que realiza la transformación. Primero es representada en el lenguaje de alto nivel PPA, transformada mediante `certify` al sistema de deducción natural (ND) clásico, traducida a intuicionista por `translateFriedman` y finalmente reducida a su forma normal (FN) por `reduce`, a partir de la cual se puede extraer un testigo. Además, se representa el mecanismo de *fallback* que verifica la correctitud de cada demostración de deducción natural con `check`, que permite encontrar problemas en las transformaciones.

1.3. Estructura del escrito

El trabajo se divide en 5 capítulos principales además de la introducción y la conclusión. Comenzamos por el Capítulo 2 (Deducción natural) en el que se presenta de forma completa el bien conocido sistema de deducción natural que se usa para los certificados. Se introducen las reglas de inferencia junto con sus intuiciones, el concepto de *reglas admisibles*, demostraciones de ejemplo y algunos algoritmos implementados: chequeo (función `check`), alfa equivalencia de fórmulas y sustitución sin capturas.

En el Capítulo 3 (El lenguaje PPA) introducimos el lenguaje desde el punto de vista de un usuario: cómo están compuestos los programas, cómo usar el pequeño demostrador (**by**) para facilitar la escritura de demostraciones, y una descripción exhaustiva de todos los comandos soportados. En Capítulo 4 (El certificador de PPA) se muestra la implementación interna del *certificador* de PPA, cómo genera demostraciones en deducción natural a partir de programas (la función `certify`). De forma central se explica el funcionamiento del **by**, que es el corazón del certificador.

En el Capítulo 5 (Extracción de testigos de existenciales) se muestra el proceso completo de extracción de testigos, comenzando por la traducción de Friedman y luego la normalización de demostraciones intuicionistas (`translateFriedman` y `reduce`).

Finalmente, en el Capítulo 6 (La herramienta `ppa`) se detallan los pasos para instalar y usar la herramienta `ppa`, y se mencionan algunos detalles de implementación como la arquitectura del programa a nivel módulos y dependencias entre ellos, el compilador y el modelado de deducción natural.

2. DEDUCCIÓN NATURAL

Comencemos por los fundamentos: queremos armar un programa que permita escribir teoremas y demostraciones, pero, ¿cómo se representa una demostración en la computadora? Veamos un ejemplo. Supongamos que tenemos la siguiente *teoría* sobre exámenes en la facultad, que iremos 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. A continuación mostramos una posible demostración en lenguaje natural.

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

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

Como se puede ver, es poco precisa. No es claro cómo se podría representar rigurosamente. Existen áreas de la lógica que estudian las demostraciones como objetos formales, como la de *teoría de pruebas*. En ellas se describen los *sistemas de inferencia* o *deductivos*: sistemas lógicos formales que permiten demostrar sentencias. Nos son útiles pues pueden ser modelados como un tipo abstracto de datos, por lo que son representables en la computadora. ¿Cómo podrá ser formalizada la demostración del Ejemplo 1 en uno?

2.1. El sistema de deducción natural

Los sistemas de inferencia 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 usamos 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]. Tiene dos tipos de *reglas de inferencia* para cada conectivo ($\wedge$, $\vee$, $\exists$, ...) y cuantificador ($\forall$, $\exists$), que nos permite razonar de dos formas distintas.

- **Regla de introducción**: ¿Cómo demuestro una fórmula formada por este conectivo?
- **Regla de eliminación**: ¿Cómo uso una fórmula formada por este conectivo para demostrar otra?

Por ejemplo, la regla I$\wedge$ nos va a permitir *introducir* una conjunción, demostrarla. E$\vee$ *eliminar* una disyunción, usarla para demostrar otra fórmula. Primero vemos algunas definiciones preliminares, luego las reglas de inferencia, un ejemplo de una demostración, y finalmente explicamos cada regla en detalle.

Def. 4 (Contexto de demostración)**.** Definimos,

- Los **contextos de demostración** Γ como un conjunto de fórmulas, compuesto por las hipótesis que se asumen a lo largo de una demostración.
- Notamos la unión como
$$\Gamma, \varphi = \Gamma \cup \{\varphi\}$$
- Para algunas reglas es necesario conocer las variables libres de un contexto, que se definen de la forma esperable:
$$fv(\Gamma) = \bigcup_{\varphi \in \Gamma} fv(\varphi)$$

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 clásica de primer orden

Def. 6 (Relación de derivabilidad)**.** Las reglas de inferencia de la Figura 2.1 definen la relación de derivabilidad $\vdash$, que nos permite escribir *juicios* $\Gamma \vdash A$. Intuitivamente interpretarse como "A es una consecuencia de las suposiciones de Γ". Dicho de forma más precisa, el juicio será cierto si en una cantidad finita de pasos podemos, a partir de las fórmulas de Γ, los axiomas y las reglas de inferencia, concluir A. En ese caso decimos que A es *derivable* a partir de Γ.

Cuando el contexto es omitido, $\vdash A$ se usa como abreviación de $\emptyset \vdash A$.

Ejemplo 2. Demostración de Ejemplo 1 en deducción natural. Lo modelamos para un solo alumno y materia, sin cuantificadores. 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. Nombramos el *contexto* de hipótesis como Γ.
- 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, vamos de derecha a izquierda.
- 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

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

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

- E⊥: a partir de ⊥, algo que es falso, vamos a poder deducir cualquier fórmula.
- I⊤: ⊤ 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∧: 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∨: 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→: para demostrar una implicación, asumimos el antecedente (agregándolo a las hipótesis) y en base a eso se demuestra el consecuente.
- E→: también conocida como *modus ponens*. A partir de una implicación, si podemos demostrar su antecedente, entonces vale su consecuente.
- I¬: para demostrar una negación, lo hacemos por el absurdo: asumimos que vale la fórmula y llegamos a una contradicción.
- E¬: 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 Γ 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. Usamos

- Las siguientes funciones 0-arias: 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 al modelar teorías de primer orden.

- **Axioma 1**: si un alumno reprueba el final de una materia, entonces recursa

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

- **Axioma 2**: 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 $\Gamma_0 \vdash \forall a.\ \forall m.\ F(a, \text{final}(m)) \to R(a, m)$

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

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

PPA genera demostraciones en deducción natural, pero no usa exactamente el sistema descrito en la Figura 2.1, sino que tuvimos que hacer algunos ajustes, que describimos a continuación.

2.3.1. Hipótesis etiquetadas

En la Def. 4 (Contexto de demostración) presentamos a los contextos Γ como *conjuntos* de fórmulas. Pero en realidad, para proveer mayor claridad y precisión en las demostraciones, vamos a querer que las hipótesis estén nombradas. Entonces cada regla que introduce una hipótesis tendrá que darle nombre, y cada regla que la use, tiene que explicitar qué nombre tiene.

- Los contextos Γ se redefinen como conjuntos de *pares* $h : A$ de etiquetas y fórmulas. Se asume que no hay etiquetas repetidas.
- Las reglas que hacen uso de hipótesis, lo hacen nombrándolas.

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

$$\dfrac{\Gamma \vdash A \vee B \qquad \Gamma, \boldsymbol{h_1} : A \vdash C \qquad \Gamma, \boldsymbol{h_2} : B \vdash C}{\Gamma \vdash C}\,\text{E}\vee_{(\boldsymbol{h_1}, \boldsymbol{h_2})}$$

$$\dfrac{\Gamma \vdash \exists x.A \qquad \Gamma, \boldsymbol{h} : A \vdash B \qquad x \notin fv(\Gamma, B)}{\Gamma \vdash B}\,\text{E}\exists_{\boldsymbol{h}}$$

Notación. Como abuso de notación, nos vamos a permitir la libertad de usar ambas versiones a lo largo del documento. En casos en donde no sea relevante usar etiquetas, no las usaremos, y en los casos en donde sí, lo haremos explícitamente.

Obs. Las reglas están decoradas con el nombre de la hipótesis porque esto hace que según cual se use, el uso de la regla y por lo tanto la demostración sea diferente. Por ejemplo, si demostramos $A \to A \to A$, con etiquetas hay dos demostraciones distintas (dependiendo de si se usa la primera o segunda hipótesis para demostrar la tercera) y sin etiquetas hay únicamente una (como el contexto es un conjunto, la primera y segunda hipótesis se combinan).

Además, el hecho de agregar etiquetas hace que los árboles de derivación se correspondan exactamente con los términos de una extensión del cálculo lambda cuyo sistema de tipos es la lógica clásica de primer orden.

2.3.2. Variables libres en contexto

Las reglas I$\forall$ y E$\exists$ validan que la variable del cuantificador no esté libre en el contexto. Cuando sí aparece, el algoritmo de chequeo debe reportar un error. Para corregirlo, el usuario debería re-escribir la demostración cambiando los nombres de las variables evitando conflictos.

Esto resultó inconveniente para la generación de demostraciones, pues se daban conflictos en casos inofensivos: si tenemos una demostración de un teorema que usa las variables x, y, y lo citamos en otro que también las usa, al chequearlo se generaría un conflicto: cuando estamos chequeando la segunda demostración se incluyen las variables al contexto, y cuando recursivamente chequeamos la primera en donde se cita, aparece. En este caso sería accidental, no estamos queriendo razonar de forma inválida.

Para evitarlo manteniendo las reglas como están, habría que renombrar con cuidado las variables de forma automática. En su lugar, las reformulamos para que no suceda: en lugar de validar que la variable no esté, directamente se borran del contexto todas las hipótesis que contengan libre esa variable en la sub-demostración. Cada sub-demostración tiene un *contexto limpio*. Esto permite que si el conflicto era accidental, la demostración siga chequeando (porque no se usaban las hipótesis que generaban conflicto). Pero si se usaban, seguiría fallando pero con un error diferente (hipótesis inexistente).

Definimos

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

Luego, las reglas se re-definen como

$$\frac{\boldsymbol{\Gamma \ominus x} \vdash A}{\Gamma \vdash \forall x.A}\ \mathrm{I}\forall$$

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

2.4. Reglas admisibles

Antes mencionamos *modus tollens* como regla de inferencia, pero no aparece en las reglas de 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 se pueden demostrar para cualquier fórmula, así luego podemos usarlas como *macros*.

Ejemplo 4 (*Modus tollens*)**.** Se puede demostrar como regla admisible.

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

2.5. Algoritmos

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

2.5.1. Chequeador

El algoritmo de chequeo de una demostración en deducción natural consiste en recorrer recursivamente el árbol de demostración, asegurando que todas las inferencias sean válidas y manteniendo un contexto Γ en el camino. Se chequea que cada regla se use con el conectivo que le corresponde (no un I$\wedge$ para un $\vee$) y que cumpla con las condiciones impuestas.

El módulo que se encarga de implementarlo es el `ND.Checker`, con su función principal `check :: Env -> Proof -> Form`, donde `Env` es el contexto Γ, `Proof` es la demostración en deducción natural y `Form` es la fórmula que demuestra.

2.5.2. Alfa 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 **alfa-equivalentes**: su estructura es la misma, pero tienen nombres diferentes para variables *ligadas* (no libres)

Def. 7 (Alfa equivalencia)**.** Se define la relación $\overset{\alpha}{=}$ como la que permite renombrar variables ligadas evitando capturas. Es la congruencia más chica que cumple con

$$(\forall x.A) \overset{\alpha}{=} (\forall y.A') \text{ si } A\{x := z\} \overset{\alpha}{=} A'\{y := z\} \text{ con } z \notin fv(A) \cup fv(A')$$
$$(\exists x.A) \overset{\alpha}{=} (\exists y.A') \text{ si } A\{x := z\} \overset{\alpha}{=} A'\{y := z\} \text{ con } z \notin fv(A) \cup fv(A')$$

Para implementarlo, un algoritmo naíf podría ser de tiempo cuadrático en peor caso: 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 de tiempo cuasilineal en la estructura de la fórmula en peor caso. 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 alfa 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,

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

Pero queremos evitar **captura de variables**. Por ejemplo, sean p un predicado unario y la siguiente sustitución

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

Si se efectúa sin más, estaríamos involuntariamente "capturando" a y: originalmente estaba libre, pero luego de la sustitución estaría ligada. 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. 8 (Sustitución sin capturas)**.** Se define por inducción estructural.

- 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

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

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 alfa igualdad, esto se puede implementar de forma naíf cuadrática en peor caso pero lo hicimos cuasilineal. 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.

Def. 9 (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

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

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

3. EL LENGUAJE PPA

El lenguaje PPA (*Pani's Proof Assistant*) se construye sobre 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 PPA, implementado por `ppa`, con un enfoque de referencia del lenguaje. Los detalles teóricos de cómo está implementado se abordan en el Capítulo 4, y los de implementación y su instalación en el Capítulo 6. Para introducirlo veamos un ejemplo: en la Figura 3.1 representamos el mismo de alumnos del Ejemplo 3 pero con un poco más de sofisticación.

Vayamos parte por parte. La primera 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 pueden agregar a modo informativo como un comentario. Los axiomas son fórmulas que siempre son consideradas como válidas. Definimos,

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

```
/* Teoría de alumnos y exámenes

Predicados
   - reprueba(A, P): El alumno A reprueba el parcial P
   - recursa(A, M): El alumno A recursa la materia M

Funciones
   - parcial(M): El parcial de una materia
   - recu(P): El recuperatorio de un parcial
*/

axiom reprueba_recu_parcial_recursa: forall A. forall M.
   (reprueba(A, parcial(M)) & reprueba(A, recu(parcial(M))))
      -> recursa(A, M)

axiom rinde_recu_reprobo_parcial: forall A. forall P.
   rinde(A, recu(P)) -> reprueba(A, P)

axiom reprobo_rinde: forall A. forall P.
   reprueba(A, P) -> rinde(A, P)

axiom falta_reprueba: forall A. forall P.
   falta(A, P) -> reprueba(A, P)
```

En base a eso demostramos dos teoremas. El primero, **theorem** `reprueba_recu_recursa`, nos permite concluir que un alumno recursa solo a partir de que reprueba el recuperatorio. 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  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.

```
24 theorem reprueba_recu_recursa:
25    forall A. forall M.
26       reprueba(A, recu(parcial(M))) -> recursa(A, M)
27 proof
28    let A
29    let M
30    suppose reprueba_recu: reprueba(A, recu(parcial(M)))
31
32    claim reprueba_p: reprueba(A, parcial(M))
33    proof
34       have rinde_recu: rinde(A, recu(parcial(M)))
35          by reprueba_recu, reprobo_rinde
36
37       hence reprueba(A, parcial(M))
38          by rinde_recu_reprobo_parcial
39    end
40
41    hence recursa(A, M)
42       by reprueba_recu,
43          reprueba_p,
44          reprueba_recu_parcial_recursa
45 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 correctos, y luego de ejecutarlos todos, la tesis se reduce por completo.

- **`let`** permite demostrar un **`forall`**, asignando un nombre general a la variable, 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`** inserta una sub-demostración auxiliar, cuya fórmula se agrega como hipótesis. No reduce la tesis.
- **`have`** agrega una hipótesis auxiliar, 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 **`axiom`** `falta_reprueba` podemos demostrar que si un alumno falta a un recuperatorio, recursa la materia.

```
46 theorem falta_recu_recursa:
47    forall A. forall M.
48       falta(A, recu(parcial(M))) -> recursa(A, M)
49 proof
50    let A
51    let M
52
53    suppose falta_recu: falta(A, recu(parcial(M)))
54
55    have reprueba_recu: reprueba(A, recu(parcial(M)))
56       by falta_recu, falta_reprueba
57
58    hence recursa(A, M) by reprueba_recu_recursa
59 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. Su sintaxis busca ser lo más parecida posible al lenguaje natural, inspirada en Mizar y el *mathematical vernacular* de Freek Wiedijk [Wie]. También se puede entender como una notación de deducción natural en el estilo de Fitch [PH24], en donde las demostraciones son esencialmente listas de fórmulas, con las que aparecen más adelante siendo consecuencia de las que aparecieron antes. El estilo de Fitch y el de Gentzen son equivalentes, describen la misma relación de derivabilidad $\Gamma \vdash \varphi$. 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. En ella pueden citar todas las hipótesis definidas previamente.

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

3.1.1. Identificadores

Los identificadores se dividen en tres tipos:

- **Variables** (`<var>`). Son una secuencia de uno o más símbolos, que pueden ser alfanuméricos o `"-"`, `"_"`. Deben comenzar por `"_"` o una letra mayúscula, y pueden estar seguidas de cero o más apóstrofes `"'"`.

  También pueden ser descritas por la siguiente expresión regular del estilo PCRE (*Perl Compatible Regular Expressions*):

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

- **Identificadores** (`<id>`). Son una secuencia de uno o más simbolos, que pueden ser alfanuméricos o `"_"`, `"-"`, `"?"`, `"!"`, `"#"`, `"$"`, `"%"`, `"*"`, `"+"`, `"<"`, `">"`, `"="`, `"?"`, `"@"`, `"^"`. Pueden estar seguidas de cero o más apóstrofes `"'"`.

  También pueden ser descritas por la siguiente expresión regular:

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

- **Nombres** (`<name>`): pueden ser identificadores, o *strings* arbitrarios encerrados por comillas dobles (`"..."`), que son descritos por la siguiente expresión regular

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

3.1.2. Comentarios

Se pueden escribir 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** `~<form>`
- **Cuantificadores**
  - **`exists`** `<var> . <form>`
  - **`forall`** `<var> . <form>`
- **`true`**, **`false`**
- **Paréntesis**: `(<form>)`

3.2. Demostraciones

Las demostraciones consisten de 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 - el 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 cuantificadores universales de a lo sumo una hipótesis, no intenta con más de una). Exploramos las limitaciones en Subsección 4.6.6 (Alcance y limitaciones).

En general, `<form> by <justification>` se interpreta como que `<form>` 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 el *solver* heurístico puede demostrar que 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 del contexto para su uso posterior. 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 guión medio (-).

Comando	Alternativo	¿Reduce la tesis?
thus	**hence**	Sí
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 en **thus** o **have**, la fórmula debe ser demostrable por el *solver* heurístico sin partir de ninguna hipótesis, lo cual en particular vale para todas las tautologías proposicionales. Por ejemplo,

```
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 manual, detallista y de bajo nivel de escribir demostraciones en deducción natural resuelve automáticamente con el uso **by**.

Regla	Comando
I$\exists$	**take**
E$\exists$	**consider**
I$\forall$	**let**
E$\forall$	**by**
I$\vee_1$	**by**
I$\vee_2$	**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**
I$\top$	**by**
E$\bot$	**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 $\neg A \equiv A \rightarrow \bot$, se puede usar para introducir negaciones, 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$\vee$)

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

```
theorem "cases":
   (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 las 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)`.

Además, se puede omitir el **by** para razonar mediante LEM, donde los casos son φ y $\neg\varphi$.

- **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.

  También es posible usar una fórmula α-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, por ejemplo `let Y`.

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
```

O también para razonar por el absurdo mediante la eliminación de la doble negación,

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

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

4. EL CERTIFICADOR DE PPA

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

4.1. Certificados

Los programas de PPA se **certifican**, generando un certificado: una demostración en deducción natural. ¿Por qué? El lenguaje es complejo, la implementación no es trivial. Si se programa una demostración, para confiar en que es correcta hay que confiar también en la implementación de PPA. Pero si generara una demostración de bajo nivel, que use las reglas de un sistema lógico simple y conocido, entonces cualquiera que desconfíe podría fácilmente escribir un chequeador independiente, o usar uno confiable. Decimos que un asistente de demostración que cuenta con esta propiedad cumple con el **criterio de de Bruijn**. `ppa` lo cumple al generar demostraciones en deducción natural.

Criterio (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 `PPA.Certifier`. Si bien toda demostración que genere debería ser correcta, para atajar posibles errores siempre se chequean con el `ND.Checker`, el pequeño módulo que implementa el chequeo de los certificados.

4.2. Contextos

Durante el proceso de certificación, en realidad no se genera una sola demostración, sino que al poder haber más de un teorema en un programa de PPA, el certificado es un **contexto** compuesto por una lista ordenada de **hipótesis** de dos tipos:

- **Teoremas**: son fórmulas con demostraciones de deducción natural asociadas.
- **Axiomas**: son fórmulas que se asumen válidas (pueden ser usadas para modelar una teoría)

Se puede ver un ejemplo en la Figura 4.1. También se puede ver en el tipo de la función principal del módulo: `certify :: Program -> Result Context`. Como para un programa generamos muchas demostraciones, debemos extender el chequeo a contextos: Cada demostración será válida en el *prefijo estricto del contexto* que la contiene. Es decir, a la hora de chequear la demostración de un teorema, se deben asumir como ciertas todas las hipótesis que fueron definidas antes que él. En el ejemplo, cuando chequeamos la demostración de `t1`, debemos asumir como válidos los axiomas `ax1, ax2, ax3`.

4.2.1. Contexto local

No solo los axiomas y teoremas declarados en el programa se agregan al contexto. Cada demostración de un teorema tendrá además un *contexto local* que extiende al anterior, solo válido durante el alcance de su demostración (se omiten en el certificado).

```
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
```

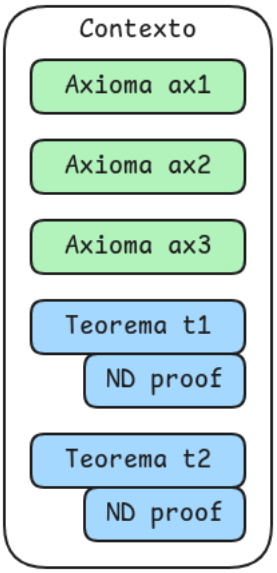

Fig. 4.1: Contexto resultante de certificar de un programa

En él, las afirmaciones auxiliares que no afectan la tesis como **have**, **claim** y **consider**, etc. se agregan como teoremas. Por lo tanto, cuando se citen, se pueden copiar sus demostraciones tomándolas del contexto local. 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 generada de deducción natural. Se puede ver un ejemplo en la Figura 4.2 (Contexto local).

4.3. Certificado de demostraciones

Ya vimos cómo las hipótesis se agregan al contexto. Pero ¿cómo generamos una demostración de deducción natural a partir de una demostración de un teorema de PPA? Para cada comando introducido en el Capítulo 3, deberemos *certificarlo* generando una demostración, y el resto del programa debería demostrar sus premisas. En la Figura 4.3 se puede ver un ejemplo, en donde **take** se certifica como I∃, y la demostración de su premisa está dada por el certificado del resto del programa, **thus** `p(v)` **by** `ax`, que se certifica como Ax.

```
1 axiom ax: p(v)
2 theorem t: exists X . p(X)
3 proof
4    take X := v
5    thus p(v) by ax
6 end
```

$$\dfrac{\dfrac{}{p(v) \vdash p(v)}\,\mathrm{Ax}}{p(v) \vdash \exists x.p(X)}\,\mathrm{I}\exists$$

Fig. 4.3: Ejemplo de certificado generado para un programa

En el resto del capítulo presentamos cómo cada comando de PPA puede ser certificado.

```
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
```

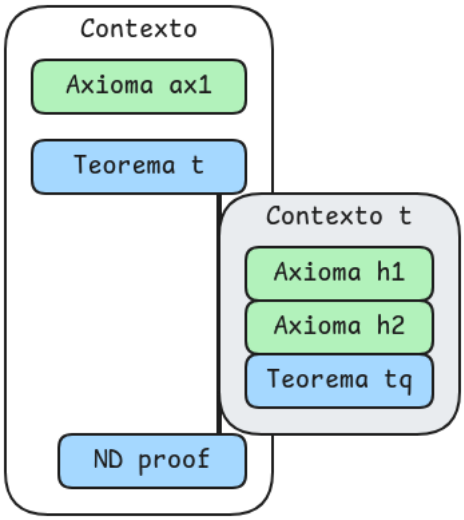

Fig. 4.2: Contexto local

- Comenzamos por ver cómo se certifican los comandos que corresponden a reglas de inferencia de forma directa, que ayuda a entender el funcionamiento de `ppa`: **`take`** (I∃), **`consider`** (E∃), **`let`** (I∀), **`cases`** (E∨) y **`suppose`** (I→, I¬). También los comandos adicionales: **`equivalently`** y **`claim`**.
- Luego, describimos el *solver* que se usa por debajo del **`by`**, que es central al funcionamiento de todo el certificador. Aquí veremos que la demostración generada para 4.3 no es exactamente como se presenta.
- Concluimos por cómo podemos usarlo para facilitar la descarga de conjunciones en un orden arbitrario.

4.4. Comandos correspondientes a reglas de inferencia

Como se puede ver en Tabla 3.1 (Reglas de inferencia y comandos), muchos de los comandos se corresponden directamente con reglas de inferencia, por lo que su traducción es directa.

- **`take`** (I∃)

  Si la tesis es **`exists`** `X . a`, el comando **`take`** `X := t` la reduce a `a` con `X` reemplazado por `t` y la certifica como

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

  insertando la demostración certificada del resto en en $\Gamma \vdash A\{x := t\}$

- **`consider`** (E∃)

  El comando **`consider`** `X` **`st`** `h: a` **`by`** `h1, ... hn` se certifica como

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

- $\Gamma \vdash \exists x.A$ se demuestra mediante el by.
- Se agrega la hipótesis `h: a` al contexto como axioma, se continúa la certificación y se inserta la demostración resultante en $\Gamma, A \vdash B$.

- **`let`** (I$\forall$)

  Si la tesis es **`forall`** `X. a`, el comando **`let`** `X` reduce la tesis a `a` y continúa la certificación, insertando el resultado en la demostración de $\Gamma \vdash A$

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

- **`cases`** (E$\vee$)

  El comando

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

  se certifica como varios E$\vee$ anidados, en donde el primer $\vee$ se certifica mediante by. Cada rama certifica la sub-demostración agregando la hipótesis del caso al contexto como axioma.

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

- **`suppose`** (I$\rightarrow$, I$\neg$)

  Si la tesis es `a -> b`, el comando **`suppose`** `h: a` reduce la tesis a `b` y agrega al contexto la hipótesis `h: a` como axioma. La certifica como

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

  insertando el resto de la demostración certificada en su sub-demostración.

  Si la tesis es `~a`, es análogo pero interpretándolo como `a -> false`.

4.5. Comandos adicionales

- **`equivalently`**

  Si la tesis es a, el comando **`equivalently`** `a'` usa el mismo solver que el by para demostrar $a' \rightarrow a$ y reduce la tesis a a'.

- **`claim`**

  La certificación del comando

```
claim h: f
proof
   ...
end
```

Consiste en certificar la sub-demostración y agregar la hipótesis `h: f` como teorema al contexto.

4.6. Implementación del by

El **by** es el mecanismo principal de demostración en PPA, y el corazón del `PPA.Certifier`. Muchas funcionalidades están implementadas a su alrededor. Genera **automáticamente** una demostración de que una fórmula es consecuencia lógica de una lista de hipótesis. Es un *solver heurístico* para lógica de primer orden. Esto es aceptable, puesto que la validez de lógica de primer orden es indecidible (por el teorema de Church [Par]), y el objetivo del trabajo no fue dar un solver innovador, sino alguno que se pueda certificar. Está basado en la implementación del mecanismo análogo en el `Checker` de Mizar [Wie02].

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

- Queremos demostrar la implicación de las hipótesis a la fórmula.

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

  A esta fórmula la llamamos **tesis**.

- **Razonamos por el absurdo**: asumiendo la negación de la tesis buscamos encontrar 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 negación de la tesis a forma normal disyuntiva (**DNF**), una disyunción de cláusulas, cada una de las cuales es una conjunción de literales (fórmulas atómicas afirmadas o negadas, y fórmulas iniciadas por 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 fórmulas de cada cláusula y a_j^i es la j-ésima fórmula de la i-ésima cláusula.

- Buscamos una **contradicción** refutando cada cláusula individualmente. Una cláusula $a_1 \wedge \ldots \wedge a_n$ será refutable si cumple una de las siguientes condiciones.

  - 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, sino que el desafío principal fue **generar la demostración en deducción natural**. Veamos un ejemplo sin generar la demostración, y sin eliminar existenciales.

Ejemplo 5 (Ejemplo sin cuantificadores)**.** Tenemos el siguiente programa

```
1 axiom ax1: a -> b
2 axiom ax2: a
3 theorem t: b
4 proof
5    thus b by ax1, ax2
6 end
```

Para certificar `thus b by ax1, ax2` hay que generar una demostración para la implicación $\big((a \to b) \land a\big) \to b$.

1. Negamos la fórmula
$$\neg[\big((a \to b) \land a\big) \to b]$$
2. La convertimos a DNF

$$\begin{aligned}
&\neg[\big((a \to b) \land a\big) \to b] \\
&\equiv \neg[\neg\big((a \to b) \land a\big) \lor b] && (A \to B \equiv \neg A \lor B) \\
&\equiv \neg\neg\big((a \to b) \land a\big) \land \neg b && (\neg(A \lor B) \equiv \neg A \land \neg B) \\
&\equiv \big((a \to b) \land a\big) \land \neg b && (\neg\neg A \equiv A) \\
&\equiv (\neg a \lor b) \land a \land \neg b && (A \to B \equiv \neg A \lor B) \\
&\equiv (\neg a \lor b) \land a \land \neg b && ((A \lor B) \land C \equiv (A \land C) \lor (B \land C)) \\
&\equiv (\neg a \land a \land \neg b) \lor (b \land a \land \neg b)
\end{aligned}$$

3. Buscamos una contradicción refutando cada cláusula
   - En $(\neg a \land a \land \neg b)$ tenemos $\neg a$ y a.
   - En $(b \land a \land \neg b)$ tenemos b y $\neg b$.

4.6.1. Certificado del by

En el resto de la sección, entramos en detalle de cómo generar una demostración para cada paso del by.

1. Subsección 4.6.2 (Razonamiento por el absurdo): primero necesitamos una forma de razonar por el absurdo, que nos permita deducir la fórmula original asumiendo su negación y demostrando $\bot$. Esto lo hacemos mediante dos reglas admisibles: la eliminación de doble negación (E$\neg\neg$), equivalente a LEM, y cut que nos permite juntar demostraciones.

2. Subsección 4.6.3 (Conversión a DNF): luego vemos cómo demostrar de forma automática que una fórmula es equivalente a su versión en DNF, que se realiza a partir de un sistema de reescritura.

3. Subsección 4.6.4 (Contradicciones): una vez que la fórmula está en DNF, tenemos que demostrar a partir de ella una contradicción.

4. Subsección 4.6.5 (Eliminación de cuantificadores universales): si no se encuentra una contradicción de forma simple, se procede a eliminar $\forall$ consecutivos y reiniciar el proceso pero con unificación de fórmulas en lugar de igualdad, para así instanciar las variables cuantificadas universalmente.

5. Subsección 4.6.6 (Alcance y limitaciones): finalmente evaluamos el alcance y limitaciones del *solver* implementado, que resulta completo para proposicional y heurístico para primer orden.

4.6.2. Razonamiento por el absurdo

Queremos asumir que no vale la fórmula original, es decir $\neg(B_1 \wedge \ldots \wedge B_n \to A)$, y llegar a una contradicción. Pero en la demostración que estamos generando, tenemos que demostrar $(B_1 \wedge \ldots \wedge B_n \to A)$. ¿Cómo se puede razonar por el absurdo?

De la misma forma que en la Sección 2.4 se introduce *modus tollens* como una regla admisible, para razonar por el absurdo vamos a usar la **eliminación de la doble negación**. Es un principio de razonamiento clásico que es equivalente a LEM.

Teorema 1 (Eliminación de la doble negación)**.** Sea A una fórmula cualquiera. Vale $\neg\neg A \vdash A$, y lo notamos como regla admisible

$$\dfrac{}{\neg\neg A \vdash A}\ \mathrm{E}\neg\neg$$

Demostración. En deducción natural,

$$\dfrac{\mathrm{LEM}\ \dfrac{}{\neg\neg A \vdash A \vee \neg A} \quad \dfrac{}{\neg\neg A, A \vdash A}\ \mathrm{Ax} \qquad \mathrm{Ax}\ \dfrac{}{\neg\neg A, \neg A \vdash \neg\neg A} \quad \dfrac{}{\neg\neg A, \neg A \vdash \neg A}\ \mathrm{Ax}}{\dfrac{}{\neg\neg A, \neg A \vdash A}\ \mathrm{E}\neg}\ \mathrm{E}\vee$$

□

¿Cómo lo usamos? Introducimos otra regla admisible: **cut**, que nos permite "pegar" demostraciones entre sí. Si estamos queriendo demostrar A, y queremos reducir el problema a B que sí podemos probar, esta regla nos permite hacerlo.

Teorema 2 (Cut)**.** La siguiente regla de inferencia es admisible

$$\dfrac{\Gamma, B \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A}\ \mathrm{cut}$$

Demostración. La regla cut se puede ver como un *macro* que por atrás genera la siguiente demostración

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

□

Ejemplo 6. Cut nos permite continuar la demostración por otra fórmula a partir de la cual podamos demostrar la original. Sean $\Pi_{B\to A}$ una demostración de $B \vdash A$ y Π_B una demostración de B (la continuación). Podemos usar cut de la siguiente manera.

$$\dfrac{\begin{array}{cc}\Pi_{B\to A} & \Pi_B \\ \Gamma, B \vdash A & \Gamma \vdash B\end{array}}{\Gamma \vdash A}\ \text{cut}$$

Se certifica como

$$\dfrac{\dfrac{\begin{array}{c}\Pi_{B\to A} \\ \Gamma, B \vdash A\end{array}}{\Gamma \vdash B \to A}\ \text{I}\to \qquad \begin{array}{c}\Pi_B \\ \Gamma \vdash B\end{array}}{\Gamma \vdash A}\ \text{E}\to$$

Notación. En los casos en donde queramos continuar la demostración por la nueva y la demostración de la implicación sea omitida (por ejemplo por ser una regla admisible) lo notaremos de una forma más sucinta:

$$\dfrac{\begin{array}{c}\Pi_B \\ \Gamma \vdash B\end{array}}{\Gamma \vdash A}\ \text{cut, } \Pi_{B\to A}$$

Lema 1 (Razonamiento por el absurdo)**.** Imaginemos que queremos demostrar A por el absurdo. Podemos usar cut y la eliminación de la doble negación para continuar la demostración por $\neg\neg A$. Al introducirla, debemos demostrar el juicio $\Gamma, \neg A \vdash \bot$: asumiendo que no es cierta la fórmula, deducimos una contradicción.

$$\dfrac{\dfrac{}{\Gamma, \neg\neg A \vdash A}\ \text{E}\neg\neg \qquad \dfrac{\begin{array}{c}\vdots \\ \Gamma, \neg A \vdash \bot\end{array}}{\Gamma \vdash \neg\neg A}\ \text{I}\neg}{\Gamma \vdash A}\ \text{cut}$$

Que también se puede notar de la siguiente forma

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ \Gamma, \neg A \vdash \bot\end{array}}{\Gamma \vdash \neg\neg A}\ \text{I}\neg}{\Gamma \vdash A}\ \text{cut, E}\neg\neg$$

Obs. A E$\neg\neg$ la formulamos como $\neg\neg A \vdash A$ y la usamos con **cut**, pero otra alternativa equivalente, levemente más tediosa para generar demostraciones, hubiera sido demostrarla como $\neg\neg A \to A$ y usarla con E$\to$ directamente.

4.6.3. Conversión a DNF

Tenemos que generar una demostración de que la negación de la tesis genera una contradicción, pero lo queremos hacer a partir de la tesis en DNF. ¿Cómo la convertimos?. Más aún, ¿Cómo generamos una demostración para la conversión?. Este es un resultado estándar, que adaptamos mínimamente para su uso en el contexto de `ppa`.

Def. 10 (DNF)**.** Una fórmula está en **forma normal disyuntiva** o DNF (*disjunctive normal form*) si es una disyunción de conjunciones de literales. Llamamos **cláusulas** a las conjunciones que la componen. Un literal será un predicado, una negación de un predicado o una fórmula cualquiera que comienza con un cuantificador. Ejemplos:

Sean a, b, c predicados 0-arios. Luego,

- $a \wedge b$ está en DNF.
- $(a \wedge c) \vee (a \wedge b)$ también.
- $(a \to b) \vee (a \wedge b)$ no lo está, porque $(a \to b)$ no es una conjunción de literales.
- $\big(\forall x.(a \to b)\big) \vee (a \wedge b)$ si.

Teorema 3 (Conversión a DNF)**.** Para toda fórmula A, existe A_{DNF} su conversión a DNF y son equivalentes: vale $\vdash A \leftrightarrow A_{\text{DNF}}$, es decir $\vdash (A \to A_{\text{DNF}}) \wedge (A_{\text{DNF}} \to A)$

Demostración. La demostración se basa en el algoritmo que presentamos a continuación en la Sección 4.6.3. □

Obs. 1. Continuamos la demostración por la refutación de la fórmula en DNF mediante el uso de cut.

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ \Gamma, A \vdash A_{\text{DNF}} & \Gamma, A, A_{\text{DNF}} \vdash \bot\end{array}}{\Gamma, A \vdash \bot}\ \text{cut}$$

Para convertir una fórmula cualquiera a DNF, vamos a implementar una traducción *small-step* mediante el siguiente sistema de reescritura bien conocido.

$\neg\neg a \rightsquigarrow a$	eliminación de $\neg\neg$
$\neg\bot \rightsquigarrow \top$	
$\neg\top \rightsquigarrow \bot$	
$a \to b \rightsquigarrow \neg a \vee b$	definición de implicación
$\neg(a \vee b) \rightsquigarrow \neg a \wedge \neg b$	distributiva de $\neg$ sobre $\wedge$
$\neg(a \wedge b) \rightsquigarrow \neg a \vee \neg b$	distributiva de $\neg$ sobre $\vee$
$(a \vee b) \wedge c \rightsquigarrow (a \wedge c) \vee (b \wedge c)$	distributiva de $\wedge$ sobre $\vee$ (der)
$c \wedge (a \vee b) \rightsquigarrow (c \wedge a) \vee (c \wedge b)$	distributiva de $\wedge$ sobre $\vee$ (izq)
$a \vee (b \vee c) \rightsquigarrow (a \vee b) \vee c$	asociatividad de $\vee$
$a \wedge (b \wedge c) \rightsquigarrow (a \wedge b) \wedge c$	asociatividad de $\wedge$

Fig. 4.4: Sistema de reescritura para conversión a DNF de forma sintáctica

Pero no podemos hacerlo meramente de forma sintáctica, sino que tenemos *generar una demostración* para cada equivalencia. Cada una será una regla admisible.

Congruencias

Hay pasos de la traducción a DNF en donde tenemos que reemplazar una sub-fórmula por una equivalente. Por ejemplo para reescribir $a \vee \neg(b \vee c) \rightsquigarrow a \vee (\neg b \wedge \neg c)$ reescribimos la sub-fórmula $\neg(b \vee c) \rightsquigarrow \neg b \wedge \neg c$. Esto de forma sintáctica sería trivial, basta con hacerlo recursivamente. Pero para demostrarlo hay que usar la la *congruencia* de los conectivos, que también hay que demostrar.

$$A \vdash A' \Rightarrow A \wedge B \vdash A' \wedge B$$
$$A \vdash A' \Rightarrow A \vee B \vdash A' \vee B$$
$$A' \vdash A \Rightarrow \neg A \vdash \neg A'$$

No hay regla de congruencia para $\rightarrow$ pues se convierte en un $\vee$. Es sumamente importante observar que $\neg$ es *contravariante*, para demostrar $\neg A \vdash \neg A'$ no necesitamos una demostración de $A \vdash A'$, sino de $A' \vdash A$. Esto quiere decir que para todas las equivalencias, incluso las congruencias, no nos alcanza con demostrarlas en un solo sentido, ya que si se usan dentro de un $\neg$, necesitaremos el otro. Vamos a necesitar demostrar la ida y la vuelta: para $\neg(a \vee b) \rightsquigarrow \neg a \wedge \neg b$ necesitamos $\neg(a \vee b) \vdash \neg a \wedge \neg b$ y $\neg a \wedge \neg b \vdash \neg(a \vee b)$. Lo notamos como

$$\neg(a \vee b) \dashv\vdash \neg a \wedge \neg b$$

Algoritmo

Finalmente, el algoritmo para generar la demostración de la conversión de una fórmula en DNF se implementa en dos partes. Por un lado, contamos con una conversión *small-step* que hace un paso de reescritura. Con él, podemos implementar la conversión como su *clausura reflexiva transitiva*: aplicarla 0 o más veces hasta que ya no cambie. Los pasos pueden ser o bien ser un paso de reescritura, o muchos recursivos de congruencia para reescribir una sub-fórmula anidada. En cada uno se usa una de las demostraciones de la Figura 4.5. En total son 26 demostraciones. Nos ahorramos los detalles porque son sencillas y bien conocidas, por ejemplo leyes de De Morgan.

Pasos base

$$\begin{aligned}
\neg\neg a &\dashv\vdash a \\
\neg\bot &\dashv\vdash \top \\
\neg\top &\dashv\vdash \bot \\
a \to b &\dashv\vdash \neg a \vee b \\
\neg(a \vee b) &\dashv\vdash \neg a \wedge \neg b \\
\neg(a \wedge b) &\dashv\vdash \neg a \vee \neg b \\
(a \vee b) \wedge c &\dashv\vdash (a \wedge c) \vee (b \wedge c) \\
c \wedge (a \vee b) &\dashv\vdash (c \wedge a) \vee (c \wedge b) \\
a \vee (b \vee c) &\dashv\vdash (a \vee b) \vee c \\
a \wedge (b \wedge c) &\dashv\vdash (a \wedge b) \wedge c
\end{aligned}$$

Pasos recursivos de congruencia (con $A \dashv\vdash A'$)

$$\begin{aligned}
A \wedge B &\dashv\vdash A' \wedge B \\
A \vee B &\dashv\vdash A' \vee B \\
\neg A &\dashv\vdash \neg A'
\end{aligned}$$

Fig. 4.5: Reglas de conversión a DNF

4.6.4. Contradicciones

Tenemos la fórmula traducida a DNF. Debemos demostrar una contradicción a partir de ella. Si tenemos las cláusulas $C_1 \vee C_2$, podemos demostrar $C_1 \vee C_2 \vdash \bot$ usando E$\vee$ y deducir una contradicción asumiendo cada una. Esto se generaliza a N cláusulas mediante el uso de E$\vee$ anidados. Por lo tanto, para refutar la disyunción de las cláusulas alcanza con refutar cada una de ellas. Para refutar una cláusula en particular, el método usa tres técnicas distintas:

- Contienen fórmulas opuestas: $A \wedge \neg A$ (con E$\neg$)
- Contienen $\bot$
- Eliminando cuantificadores universales consecutivos (Subsección 4.6.5)

Supongamos que queremos refutar la cláusula $\neg a \wedge \ldots \wedge a \wedge \ldots \wedge a_n$. Tenemos que usar E$\neg$ y demostrar a partir de ella a y $\neg a$. Pero cómo E$\wedge_1$ y E$\wedge_2$ son reglas binarias, hay que usarlas de forma anidada para llegar a cada fórmula. Este anidamiento va a depender de cómo esté asociada la conjunción. Hacerlo a mano cada vez sería muy laborioso. Para simplificarlo demostraremos otra regla admisible: la proyección de un elemento E$\wedge_\varphi$.

Lema (Regla admisible E$\wedge_\varphi$)**.** Sea φ alguna fórmula de la conjunción $\varphi_1 \wedge \ldots \wedge \varphi_n$. Notamos por E$\wedge_\varphi$ a la proyección *de la fórmula*, sin importar en qué posición de la conjunción está.

$$\dfrac{\Gamma \vdash \varphi_1 \wedge \ldots \wedge \varphi_i \wedge \ldots \wedge \varphi_n \qquad n \in \mathbb{N}}{\Gamma \vdash \varphi_i}\, \mathrm{E}\wedge_{\varphi_i}$$

Demostración. Para generar la demostración correspondiente usando $\mathrm{E}\wedge_1$ y $\mathrm{E}\wedge_2$, basta con identificar el camino hacia φ_i, y luego caminar recursivamente el $\wedge$ usando $\mathrm{E}\wedge_1$ si el camino continúa por la izquierda y $\mathrm{E}\wedge_2$ si sigue por la derecha. □

Ejemplo 7 (Contradicción)**.** Veamos un ejemplo de las primeras dos formas de refutar cláusulas. Los cuantificadores universales los veremos en la siguiente sección.

$$\dfrac{\mathrm{Ax}\,\dfrac{}{\Gamma \vdash (\neg a \wedge a \wedge \neg b) \vee (b \wedge a \wedge \bot)} \qquad \Pi_L \qquad \dfrac{\dfrac{}{\Gamma_1 \vdash b \wedge a \wedge \bot}\,\mathrm{Ax}}{\Gamma, b \wedge a \wedge \bot \vdash \bot}\,\mathrm{E}\wedge_\bot}{\Gamma = (\neg a \wedge a \wedge \neg b) \vee (b \wedge a \wedge \bot) \vdash \bot}\,\mathrm{E}\vee$$

donde

$$\Pi_L = \dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \neg a \wedge a \wedge \neg b}\,\mathrm{Ax}}{\Gamma_1 \vdash \neg a}\,\mathrm{E}\wedge_{\neg a} \qquad \dfrac{\dfrac{}{\Gamma_1 \vdash \neg a \wedge a \wedge \neg b}\,\mathrm{Ax}}{\Gamma_1 \vdash a}\,\mathrm{E}\wedge_a}{\Gamma_1 = \Gamma, b \wedge a \wedge \bot \vdash \bot}\,\mathrm{E}\neg$$

4.6.5. Eliminación de cuantificadores universales

Hasta ahora logramos razonar por el absurdo, convertir la fórmula a DNF, y encontrar una contradicción siempre que no haya que eliminar cuantificadores universales. Pero es usual que en una teoría de primer orden, los axiomas los usen y sea necesario eliminarlos para poder certificar un **by**. Al hacerlo, vamos a reemplazar las ocurrencias de su variable por *meta-variables*: aquellas que pueden ser unificadas.

Def. 11 (Términos con meta-variables)**.** Extendemos la gramática de los términos (Def. 1) para que un término pueda ser una meta-variable $(\mathrm{u}, \mathrm{v}, \ldots)$

$$t ::= \ldots \mid \mathrm{u}.$$

Def. 12 (Sustitución)**.** Una sustitución es una función que asigna un término a cada metavariable. Se pueden aplicar a términos y fórmulas de la manera usual.

Def. 13 (Unificación)**.** Sean A, B dos fórmulas. Decimos que *unifican* y lo notamos $A \doteq B$ si existe una sustitución θ de meta-variables tal que $A\theta = B\theta$. Veamos algunos ejemplos. Sean u, v meta-variables.

- $p(\mathrm{u}) \doteq p(a)$ con $\{\mathrm{u} := a\}$
- $p(\mathrm{u}) \not\doteq q(a)$
- $p(\mathrm{u}) \wedge q(b) \doteq p(a) \wedge q(\mathrm{v})$ con $\{\mathrm{u} := a, \mathrm{v} := b\}$
- $p(\mathrm{u}) \to q(b) \not\doteq p(a) \wedge q(\mathrm{v})$

Obs. El problema de unificación que nos encontramos aquí es el problema estándar de unificación de primer orden. El algoritmo que usa `ppa` es una variante del algoritmo de Martelli-Montanari [MM82], que encuentra un unificador más general.

Ejemplo 8 (Ejemplo de by con cuantificadores)**.** Tenemos el siguiente programa

```
1 axiom ax1: forall X . p(X) -> q(X)
2 axiom ax2: p(a)
3 theorem t: q(a)
4 proof
5    thus q(a) by ax1, ax2
6 end
```

Para certificar `thus q(a) by ax1, ax2` hay que generar una demostración para la implicación $\Big(\big(\forall x.(p(x) \to q(x))\big) \land p(a)\Big) \to q(a)$.

1. Negamos la fórmula
$$\neg \left[\Big(\big(\forall x.(p(x) \to q(x))\big) \land p(a)\Big) \to q(a)\right].$$

2. La convertimos a DNF
$$\begin{aligned}
&\neg \left[\Big(\big(\forall x.(p(x) \to q(x))\big) \land p(a)\Big) \to q(a)\right] \\
&\equiv \neg \left[\neg\Big(\big(\forall x.(p(x) \to q(x))\big) \land p(a)\Big) \lor q(a)\right] && (A \to B \equiv \neg A \lor B) \\
&\equiv \neg\neg\Big(\big(\forall x.(p(x) \to q(x))\big) \land p(a)\Big) \land \neg q(a) && (\neg(A \lor B) \equiv \neg A \land \neg B) \\
&\equiv \big(\forall x.(p(x) \to q(x))\big) \land p(a) \land \neg q(a) && (\neg\neg A \equiv A)
\end{aligned}$$
como a los ojos de DNF un $\forall$ es opaco, a pesar de que dentro tenga una implicación, la fórmula ya está en forma normal.

3. Buscamos una contradicción refutando cada cláusula. No hay forma encontrando literales opuestos o $\bot$, por ej. la cláusula $p(a)$ no es refutable.

4. Probamos eliminando $\forall x.(p(x) \to q(x))$. Reemplazamos x por una meta-variable fresca u.
$$(p(\mathrm{u}) \to q(\mathrm{u})) \land p(a) \land \neg q(a)$$
¡No está en DNF! Hay que volver a convertir.

5. Convertimos a DNF
$$\begin{aligned}
&(p(\mathrm{u}) \to q(\mathrm{u})) \land p(a) \land \neg q(a) \\
&\equiv (\neg p(\mathrm{u}) \lor q(\mathrm{u})) \land p(a) \land \neg q(a) && (A \to B \equiv \neg A \lor B) \\
&\equiv ((\neg p(\mathrm{u}) \land p(a)) \lor (q(\mathrm{u}) \land p(a))) \land \neg q(a) && ((A \lor B) \land C \equiv (A \land C) \lor (A \land C)) \\
&\equiv (\neg p(\mathrm{u}) \land p(a) \land \neg q(a)) \lor && ((A \lor B) \land C \equiv (A \land C) \lor (A \land C)) \\
&\quad (q(\mathrm{u}) \land p(a) \land \neg q(a))
\end{aligned}$$

6. Buscamos una contradicción refutando cada cláusula. Esta vez, por diseño, no podemos volver a eliminar un cuantificador universal. Los literales opuestos tienen que *unificar* en lugar de ser iguales. Las sustituciones tienen que ser compatibles entre todas las cláusulas (no pueden asignar valores diferentes a la misma meta-variable)

   - $\neg p(\mathrm{u}) \land p(a) \land \neg q(a)$ tenemos $p(\mathrm{u}) \doteq p(a)$ con $\{\mathrm{u} := a\}$
   - $q(\mathrm{u}) \land p(a) \land \neg q(a)$ tenemos $q(\mathrm{u}) \doteq q(a)$ con $\{\mathrm{u} := a\}$

Algoritmo

Si la cláusula no puede ser refutada por contener $\bot$ ni literales opuestos, procedemos a ejecutar la eliminación de cuantificadores universales:

1. Para cada fórmula, si comienza con $\forall$ (ej. $\forall x_0.\forall x_1 \ldots \forall x_n.p(x_0, \ldots, x_n)$) se busca una refutación sin generar la demostración.
   - Para cada combinación de prefijos de $\forall$ consecutivos, se intenta eliminarlos reemplazando sus variables por meta-variables frescas, re-convirtiendo a DNF y buscando una refutación (unificando con α-equivalencia). Por ejemplo, se probaría con las fórmulas:
     - $\forall x_1 \forall x_2 \ldots \forall x_n.p(u_0, x_1, x_2, \ldots, x_n)$
     - $\forall x_2 \ldots \forall x_n.p(u_0, u_1, x_2, \ldots, x_n)$
     - $\ldots$
     - $p(u_0, \ldots, u_n)$
   - Nos quedamos con el primero que logra una refutación, así solo eliminamos los cuantificadores necesarios y no todos. Esto nos da como resultado una sustitución que asigna a cada meta-variable u_i un término t_i.
2. A partir de la sustitución, sabemos cuales $\forall$ hay que eliminar. Para cada uno, se usa E$\forall$ instanciando cada variable por el término asignado a su meta-variable correspondiente en la sustitución. Con ello podemos demostrar a partir de la fórmula original, la fórmula con los $\forall$ eliminados y las variables instanciadas en los términos que logran la refutación.
3. A partir de la fórmula instanciada se genera la refutación de la forma previa.

Ejemplo 9 (Eliminación consecutiva minimal)**.** Veamos dos ejemplos que muestran cómo sólo se eliminan los $\forall$ necesarios para lograr la refutación.

- $(\forall x.\forall y.p(x) \vee q(y)) \wedge \neg p(a) \wedge \neg q(b)$

  Se eliminarían ambos $\forall$, reemplazando tanto x como y por meta-variables frescas u_0 y u_1, quedando así $(p(u_0) \vee q(u_1)) \wedge \neg p(a) \wedge \neg q(b)$ que se podrá refutar.

- $(\forall x.\forall y.p(x) \wedge q(y)) \wedge \neg(\forall z.p(a) \wedge q(z))$

  Es necesario eliminar solamente $\forall x$. Y la unificación será capaz de tener en cuenta la α-equivalencia, así puede unificar $\forall y.p(u_0) \wedge q(y) \doteq \forall z.p(a) \wedge q(z)$.

Ejemplo 10 (Eliminación de una sola fórmula)**.** Se eliminan los $\forall$ consecutivos de una sola fórmula de la cláusula. Por ejemplo, en la siguiente o bien eliminamos $\forall x.p(x)$ o $\forall y.q(y)$ pero no ambos.

$$(\forall x.p(x)) \wedge (\forall y.q(y)) \wedge \neg p(a) \wedge \neg q(b)$$

Compatibilidad de sustituciones

Como cada cláusula se refuta por separado, hay que asegurar que las sustituciones de cada una sean compatibles entre sí. Es decir, que no asignen términos diferentes a las mismas meta-variables. Debemos probar con todas las sustituciones posibles, porque puede haber algunas combinaciones incompatibles y otras no. Por ejemplo, en las siguientes cláusulas hay más de una sustitución candidata para cada una.

- $f(u_0) \land \neg f(a) \land \neg f(b)$ tenemos $\{u_0 := a\}$, $\{u_0 := b\}$
- $f(u_0) \land \neg f(b)$ solo $\{u_0 := b\}$

Pero si en la primera nos quedamos con $\{u_0 := a\}$, no podríamos refutar la segunda. Por lo tanto la correcta es $\{u_0 := b\}$.

4.6.6. Alcance y limitaciones

El solver implementado es **completo para lógica proposicional**, pero heurístico para lógica de primer orden. Para la demostración de completitud, usaremos la noción de **semántica** de lógica de primer orden que aún no definimos. En síntesis,

- Una *valuación* v es una función que asigna un valor de verdad a cada variable proposicional.
- Sean φ una fórmula proposicional y v una valuación. Notamos $v \vDash \varphi$ si la valuación hace verdadera a la fórmula.
- Si todas las valuaciones hacen verdaderas a φ, lo notamos como $\vDash \varphi$.

Teorema 4. El solver es **completo** para lógica proposicional. Sea φ una fórmula proposicional, sin ocurrencias de cuantificadores. Si es universalmente válida, $\vdash \varphi$, entonces el solver encuentra una demostración.

Demostración. Este es un resultado bien conocido, con adaptaciones mínimas para el marco puntual que estamos usando.

Lo vamos a demostrar semánticamente. Sea φ una fórmula proposicional cualquiera, supongamos que es válida. Queremos argumentar que el solver va a poder generar una demostración para ello. Sabemos que es válida si y solo si su negación es insatisfactible

$$\vdash \varphi \iff \vDash \varphi \iff \nvDash \neg\varphi,$$

por lo que es correcto buscar una refutación de la negación. Además, siempre vamos a poder convertir $\neg\varphi$ a DNF (Teorema 3), por lo que tenemos

$$\nvDash \neg\varphi \iff \nvDash (a_1 \land \ldots \land a_n) \lor \ldots \lor (b_1 \land \ldots \land b_m).$$

Para que sea insatisfactible, por definición de la semántica de $\lor$ todas las cláusulas tienen que serlo. Para que una claúsula lo sea, no tiene que haber una valuación que la haga verdadera. Supongamos que existe una valuación v que satisface $a_1 \land \ldots \land a_n$. Los valores que asigna a $a_1, \ldots, a_n$ están unívocamente determinados: al ser una conjunción, tiene que asignar a cada literal a_i un 1 si es positivo o un 0 si es negativo. Luego, será una

contradicción si y solo si aparece la misma variable y su opuesta, o si aparece $\bot$. Como esto es exactamente lo que busca el solver, concluimos que si es refutable, siempre va a poder refutarla. Por lo tanto es completo para lógica proposicional.

□

Obs. El solver es **incompleto** para lógica de primer orden. El siguiente es un contra ejemplo, que no encuentra la refutación porque para ello debería eliminar ambos $\forall$, pero elimina a lo sumo los de una fórmula.

```
1 axiom ax1: forall X . p(X) -> q(X)
2 axiom ax2: forall X . p(X)
3 theorem t: q(a)
4 proof
5    thus q(a) by ax1, ax2
6 end
```

Pero esto no es lo único que le falta para ser completo. También se podría dar un contra ejemplo más burdo que requiera un *SAT solver*.

4.6.7. Azúcar sintáctico

La última parte de la certificación del **by** es su azúcar sintáctico. Antes mencionamos que se puede usar **hence** en lugar de **thus** referenciando automáticamente a la hipótesis anterior, y análogamente lo mismo para **have** y **then**. Esto se implementa desde el *parser*. Son solamente una construcción sintáctica que se genera directamente como **have** o **thus**, y se agrega la hipótesis anterior (-) a cada uno. Esto simplifica al certificador, permitiendo que solo opere con ellos dos.

4.7. Descarga de conjunciones

Veamos la certificación de la *descarga de conjunciones*: si la tesis es una conjunción `a_1 & ... & a_n`, debemos poder *descargar cualquier subconjunto* de ella con el **thus**. Por ejemplo ¿cómo podemos certificar el siguiente programa?

```
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
```

Veamos en particular el primer comando de la demostración. La tesis es una conjunción que está asociada de una forma particular, y se quiere demostrar $a \wedge e$ que es parte de la tesis. Aprovecharemos que las conjunciones se pueden representar como conjuntos para simplificar el proceso.

Prop (Conjunciones como conjuntos)**.** La conjunción es un conectivo asociativo, conmutativo e idempotente si se lo considera módulo equivalencia lógica. Es decir,

$$\begin{aligned} A \wedge (B \wedge C) &\iff (A \wedge B) \wedge C \\ A \wedge B &\iff B \wedge A \\ A \wedge A &\iff A \end{aligned}$$

Esto nos permite representar a una fórmula construida como un árbol binario de conjunciones a través del conjunto de sus hojas.

1. Representamos la tesis y lo que se busca demostrar como conjuntos

$$\begin{aligned} (a \wedge b) \wedge ((c \wedge d) \wedge e) &\rightsquigarrow \{a, b, c, d, e\} \\ (a \wedge e) &\rightsquigarrow \{a, e\} \end{aligned}$$

2. Si está contenido, se separa lo demostrado y el resto

$$\{a, b, c, d, e\} \rightsquigarrow \{b, c, d\}\{a, e\}$$

3. Se construye una *nueva tesis* reordenando la fórmula de forma tal que sea sencillo hacer I$\wedge$ (poniendo a la izquierda lo demostrado aislado de lo demás)

$$(a \wedge e) \wedge (b \wedge c \wedge d)$$

   De esa forma, al usar la regla

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

   la demostración de $a \wedge e$ se hace de la forma usual con by, y se reduce la tesis a $b \wedge c \wedge d$ continuando la certificación por allí, insertando la demostración resultante en $\Gamma \vdash B$.

4. Para demostrar la equivalencia entre la tesis vieja y su versión re-ordenada se usa el mismo solver que el by. Al ser completo para proposicional, también puede resolver equivalencias por asociatividad, conmutatividad e idempotencia.

Finalmente, queda certificado como

$$\dfrac{\text{solver}\ \dfrac{}{\Gamma \vdash (a \wedge e) \wedge (b \wedge c \wedge d) \to (a \wedge b) \wedge ((c \wedge d) \wedge e)} \qquad \dfrac{\dfrac{}{\Gamma \vdash a \wedge e}\,\text{by} \qquad \begin{array}{c}\vdots\\ \Gamma \vdash b \wedge c \wedge d\end{array}}{\Gamma \vdash (a \wedge e) \wedge (b \wedge c \wedge d)}\,\mathrm{I}\wedge}{\Gamma \vdash (a \wedge b) \wedge ((c \wedge d) \wedge e)}\,\mathrm{E}\to$$

5. EXTRACCIÓN DE TESTIGOS DE EXISTENCIALES

En los capítulos anteriores vimos como el lenguaje PPA puede ser usado para escribir demostraciones de alto nivel, que se certifican generando demostraciones de bajo nivel en el sistema lógico de deducción natural. Ahora vamos a introducir una nueva funcionalidad: la **extracción de testigos**.

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

Fig. 5.1: Extracción simple

```
axiom ax: forall Y . p(Y, v)
theorem t:
   forall X. exists V . p(X, V)
proof
   let X
   take V := v
   thus p(X, v) by ax
end
```

Fig. 5.2: Extracción con instanciación

```
axiom ax1: q(m)
axiom ax2: forall X. q(X) -> p(X)

theorem t1: exists X. q(X)
proof
   take X := m
   thus q(m) by ax1
end

theorem t2: exists X. p(X)
proof
   consider Y st h: q(Y) by t1
   take X := Y
   hence p(Y) by ax2
end
```

Fig. 5.3: Extracción indirecta

Por ejemplo, en el programa Figura 5.1 (Extracción simple) la extracción nos permitirá encontrar un término t que sea *testigo* de $\exists x.p(x)$: que cumpla $p(t)$. En este caso es fácil encontrarlo a ojo sobre la demostración de PPA, sería `v`. Pero puede haber casos en donde no sea tan trivial, como en el programa Figura 5.3 (Extracción indirecta), en donde en el **`theorem`** `t2` se instancia la variable en un término de forma indirecta. Incluso casos como Figura 5.4 (Extracción via razonamiento por el absurdo) en donde al razonar por el absurdo con la negación de un universal, la instanciación de la variable en un término no es con **`take`** sino la unificación del **`by`**.

También vamos a querer poder extraer en casos donde haya cuantificadores universales, como en Figura 5.2 (Extracción con instanciación). Aquí, ya que vale para todo `Y`, querremos poder instanciarlo en un término cualquiera. Por ejemplo `p(v, f(v))`.

Buscamos un mecanismo general, que nos permita a partir de cualquier demostración una fórmula de la forma $\forall x_0 \ldots \forall x_n \exists y.\varphi(x_0, \ldots, x_n, y)$ extraer un testigo u tal que, para $t_0, \ldots, t_n$ cuales quiera, valga $\varphi(t_0, \ldots, t_n, u)$. Vamos a hacerlo a partir de los certificados de deducción natural.

5.1. La lógica clásica no es constructiva

El objetivo es extraer el testigo de las demostraciones generadas por el certificador, pero estas son en lógica clásica, que tiene el gran problema de que en general **no es constructiva**. ¿Qué quiere decir? Puede suceder que una demostración de $\exists x.p(x)$ no diga explícitamente quien es x, y por lo tanto no podamos extraer un testigo. Esto sucede porque vale el *principio del tercero excluido* o LEM

```
axiom juanEsBajo: bajo(juan)

theorem noTodoElMundoEsAlto: ~forall X. ~bajo(X)
proof
   suppose todosSonAltos: forall X. ~bajo(X)
   thus false by juanEsBajo, todosSonAltos
end

theorem hayAlguienBajo : exists X. bajo(X)
proof
   equivalently ~~exists X. bajo(X)
   suppose h1: ~exists X. bajo(X)

   claim h2: forall X. ~bajo(X)
   proof
      let X
      suppose h3: bajo(X)
      claim h4: exists Y. bajo(Y)
      proof
         take Y := X
         hence bajo(X) by h3
      end
      hence false by h1
   end

   hence false by noTodoElMundoEsAlto
end
```

Fig. 5.4: Extracción via razonamiento por el absurdo

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 5. Existen dos números irracionales a y b tales que a^b es racional.

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 $a = b = \sqrt{2}$ que por hipótesis es racional.

- 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.

En ambos casos podemos encontrar a, b tales que a^b es racional. □

La prueba no nos da forma de saber cuales son a y b. Es por eso que en general, en lógica clásica, tener una demostración de un teorema que afirma la existencia de un objeto que cumpla cierta propiedad, no necesariamente nos da una forma de encontrarlo. De esta demostración nunca podríamos extraer un testigo. Pero se podría haber demostrado de otra manera que sí sea constructiva, y omita el uso de LEM. En particular tomando $a = \sqrt{2}$ y $b = 2\log_2 3$ (ver [Bau]) ¿Es este el caso con todas las demostraciones?

Ejemplo 11 (Fórmula sin demostración constructiva)**.** Si consideramos el siguiente esquema

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

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

Veamos un ejemplo concreto. Digamos que $\texttt{HALT}(p, x)$ representa la afirmación de que el programa p termina cuando se le provee la entrada x. Luego, no podremos demostrar la siguiente formula de forma constructiva:

$$\forall p.\forall x.\exists y.\big((y = 1 \wedge \mathrm{HALT}(p, x)) \vee (y = 0 \wedge \neg \mathrm{HALT}(p, x))\big)$$

5.2. Lógica intuicionista

Como alternativa a la lógica clásica existe existe la lógica **intuicionista**, que se puede definir como la lógica clásica sin LEM[1]. Al no valer ese principio, las demostraciones siempre son constructivas. Esto permite por un lado tener interpretaciones computacionales y además que la noción de *forma normal* de una demostración nos ayude a extraer testigos. Existen métodos bien conocidos para reducir pruebas hacia su forma normal con un proceso análogo a una reducción de cálculo λ. Vemos esto más en detalle en la Sección 5.5 (Normalización (o reducción)).

Esto permite usar como estrategia de extracción la siguiente: normalizar la demostración y obtener el testigo de la forma normal. En ella, se esperaría que toda demostración de un $\exists$ sea mediante I$\exists$, explicitando el testigo.

5.3. Estrategia de extracción de testigos

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 métodos que permiten *embeber* la lógica clásica en la intuicionista. Uno de ellos es la **traducción de Friedman** que se aborda en la siguiente sección (5.4). La estrategia general entonces es la siguiente (esquematizada en la arquitectura de `ppa` que vimos en la introducción: Figura 1.1), dado un programa en PPA como por ejemplo de Figura 5.1:

1. La certificamos generando una demostración clásica en deducción natural, usando el `PPA.Certifier`. Nos da un contexto con una demostración por teorema.

   Limitación: No vamos a poder axiomatizar cualquier teoría. Los axiomas deben ser *F-fórmulas* (ver Sección 5.6).

[1] Al no tener LEM, tampoco valen principios de razonamiento clásicos equivalentes, como la eliminación de la doble negación.

2. Queremos extraer un testigo de un teorema, que puede citar a los anteriores. Generamos una única demostración haciendo *inline* de las demostraciones de otros teoremas citados, así cuando se reduce, se reduce la demostración completa y no una parte.

3. Usamos la traducción de Friedman para obtener una demostración intuicionista de la misma fórmula.

   Restricción: La fórmula a demostrar debe ser de la forma $\forall y_0 \dots \forall y_n . \exists x . \varphi$. con φ *conjuntiva* (ver Def. 18).

4. Instanciamos las variables de los $\forall$ en términos proporcionados por el usuario, quedando una fórmula de la forma $\exists x . A$.

5. Normalizamos la demostración.

   Limitación: No vamos a poder llevar cualquier demostración a una forma normal útil, dado que algunos "desvíos" no están contemplados (ver Subsección 5.5.3).

6. Al ser una demostración normalizada de un $\exists$, debe comenzar con I$\exists$, que especifica el término que hace cierta la fórmula. Este es precisamente el testigo que estábamos buscando.

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

En las siguientes secciones vemos en detalle la traducción de Friedman (Sección 5.4) y la normalización (o reducción) de demostraciones (Sección 5.5). Al final vemos un detalle técnico de la traducción, necesario para permitir la reducción, que limita la forma de los axiomas (Sección 5.6).

5.4. Traducción de Friedman

Existen muchos métodos que permiten embeber la lógica clásica en la intuicionista. Un mecanismo general es la traducción de **doble negación**, que tiene distintas variaciones. Una es la *Gödel-Gentzen*. Ver, por ejemplo, [AF98] para una explicación de las traducciones de doble negación.

Def. 14 (Traducción *Gödel-Gentzen*)**.** Dada una fórmula A se asocia con otra A^{N}. La traducción se define por inducción estructural.

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

Def. 15 (Traducción de contextos)**.** Se extiende a contextos de la forma esperable

$$\Gamma^{\mathrm{N}} = \{A^{\mathrm{N}} \mid A \in \Gamma\}.$$

Notación. Notamos,

- $\vdash_C$ para expresar que un juicio es derivable en lógica clásica, y $\vdash_I$ para intuicionista.
- $\Pi \rhd \Gamma \vdash A$ para expresar que Π es una demostración de $\Gamma \vdash A$. Equivalente a $\dfrac{\Pi}{\Gamma \vdash A}$

Teorema 6. Si tenemos $\Gamma \vdash_C A$, luego $\Gamma^{\mathrm{N}} \vdash_I A^{\mathrm{N}}$.

Dada una demostración en lógica clásica, podemos obtener una en lógica intuicionista de su traducción. Pero esto no es exactamente lo que queremos, pues si quisiéramos extraer un testigo de una demostración de la fórmula $\exists x.p(x)$, al traducirla nos quedaría $(\exists x.p(x))^{\mathrm{N}} = \neg\forall x.\neg\neg\neg p(x)$. Si bien su demostración será intuicionista (y por lo tanto constructiva), como no es de un $\exists$ al normalizarla no podremos hacer la extracción.

5.4.1. El truco de Friedman

La idea de Friedman, explicada claramente en [Miq11], es generalizar la traducción Gödel-Gentzen reemplazando la negación intuicionista $\neg A \equiv A \to \bot$ por una *negación relativa* $\neg_R A \equiv A \to R$ parametrizada por una fórmula arbitraria R. Esto nos va a permitir, con una elección inteligente de R, traducir una demostración clásica de una fórmula a una intuicionista, y usarla para generar una demostración de **la fórmula original**. Esto nos permite reducirla y hacer la extracción. No va a ser posible para cualquier fórmula, sino las de una clase particular que definimos más adelante (de la forma $\forall y_1 \ldots \forall y_m.\exists x.\varphi$).

Def. 16 (Traducción de doble negación relativizada)**.**

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

Teorema 7. Si $\Gamma \vdash_C A$, luego $\Gamma^{\neg\neg} \vdash_I A^{\neg\neg}$

Demostración. Dada una demostración en deducción natural clásica $\Gamma \vdash_C A$, podemos traducirla recursivamente extendiendo la traducción de fórmulas a reglas de inferencia, así generando una demostración de $\Gamma^{\neg\neg} \vdash_I A^{\neg\neg}$. Este proceso está descrito en detalle en Subsección 5.4.4 (Traducción de demostraciones) □

Vamos a enunciar diferentes versiones de la traducción de Friedman en orden de sofisticación, según para qué clase de fórmulas funcionan. No solo ayuda a entenderla, sino que también fue el mismo enfoque con el que las implementamos. Cada una incluye a la anterior, por lo que en `ppa` solo quedó implementada la última. Todas emplean la misma estrategia. Usando el Teorema 7, se traduce la demostración clásica a una intuicionista para la fórmula traducida, que se usa para probar la fórmula original.

Def. 17. Por analogía con la jerarquía aritmética, definimos las clases Π_n y Σ_n. Se definen por inducción en n.

- Si φ es una fórmula sin cuantificadores, está en Π_0 y Σ_0.
- Sean las clasificaciones Π_n y Σ_n. Definimos para $n+1$.
  - Si φ es una formula de la forma $\exists x_1 \dots \exists x_k.\psi$ donde ψ es Π_n, entonces φ es asignada la clasificación Σ_{n+1}.
  - Si φ es una formula de la forma $\forall x_1 \dots \forall x_k.\psi$ donde ψ es Σ_n, entonces φ es asignada la clasificación Π_{n+1}.

Una fórmula de Σ_n es equivalente a una que comienza con cuantificadores existenciales y alterna $n-1$ veces entre series de universales y existenciales. Mientras que una Π_n es análoga pero comenzando con universales. Las dos relevantes son:

- Σ_1: fórmulas de la forma $\exists x_1 \dots \exists x_k.\varphi$.
- Π_2: fórmulas de la forma $\forall y_1 \dots \forall y_m.\exists x_1 \dots \exists x_k.\varphi$

5.4.2. Versiones de la traducción

Podremos probar una fórmula en lógica intuicionista a partir de una demostración clásica solo si es de la clase Π_2 con dos salvedades: debe tener un solo $\exists$, por lo que es de la forma $\forall y_1 \dots \forall y_m.\exists x.\varphi$, y φ debe ser *conjuntiva*, noción que explicaremos más adelante. En lugar de abordar directamente el caso general, vamos a abordar dos versiones más simples primero para que se entienda mejor el proceso.

1. **Fórmulas Σ_1 atómicas** (Teorema 8 usando Lema 2):

   $$\exists x.A(x) \text{ con } A \text{ atómica.}$$

2. **Fórmulas Π_2 atómicas** (Teorema 9):

   $$\forall y_1 \dots \forall y_n.\exists x.A(x, y_1, \dots, y_n) \text{ con } A \text{ atómica.}$$

3. **Fórmulas Π_2 no atómicas** (Teorema 10 usando Lema 5):

   $$\forall y_1 \dots \forall y_n.\exists x.\varphi(x, y_1, \dots, y_n) \text{ con } \varphi \textit{ conjuntiva.}$$

   Por ejemplo, podría ser $p(x) \wedge q(y_1, \dots, y_n)$. Pero no podrá ser cualquier fórmula, por ej. no $\neg\neg p(x)$. En el Lema 5 damos una caracterización.

Lema 2 (Eliminación de triple negación relativa)**.** $\neg_R\neg_R\neg_R A \iff \neg_R A$ y lo demostramos como dos reglas admisibles, una para cada lado

$$\frac{}{\neg_R\neg_R\neg_R A \vdash_I \neg_R A}\,\mathrm{E}\neg_R\neg_R\neg_R \qquad\qquad \frac{}{\neg_R A \vdash_I \neg_R\neg_R\neg_R A}\,\mathrm{I}\neg_R\neg_R\neg_R$$

Demostración. Primero $\mathrm{I}\neg_R\neg_R\neg_R$

$$\dfrac{\dfrac{}{\neg_R A, \neg_R\neg_R A \vdash_I \neg_R\neg_R A}\,\mathrm{Ax} \quad \dfrac{}{\neg_R A, \neg_R\neg_R A \vdash_I \neg_R A}\,\mathrm{Ax}}{\dfrac{\neg_R A, \neg_R\neg_R A \vdash_I R}{\neg_R A \vdash_I \neg_R\neg_R\neg_R A}\,\mathrm{I}\to}\,\mathrm{E}\to$$

Ahora $\mathrm{E}\neg_R\neg_R\neg_R$

$$\dfrac{\dfrac{}{\neg_R\neg_R\neg_R A, A \vdash_I \neg_R\neg_R\neg_R A}\,\mathrm{Ax} \quad \dfrac{\dfrac{}{\Gamma \vdash_I \neg_R A}\,\mathrm{Ax} \quad \dfrac{}{\Gamma \vdash_I A}\,\mathrm{Ax}}{\dfrac{\Gamma = \neg_R\neg_R\neg_R A, A, \neg_R A \vdash_I R}{\neg_R\neg_R\neg_R A, A \vdash_I \neg_R\neg_R A}\,\mathrm{I}\to}\,\mathrm{E}\to}{\dfrac{\neg_R\neg_R\neg_R A, A \vdash_I R}{\neg_R\neg_R\neg_R A \vdash_I \neg_R A}\,\mathrm{I}\to}\,\mathrm{E}\to$$

□

Teorema 8 (Traducción de Friedman para fórmulas Σ_1)**.** Sea Π una demostración clásica de $\exists x.A$, y A una fórmula atómica. Si tenemos

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

podemos generar una demostración intuicionista de *la misma fórmula*

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

Demostración. Aplicando la traducción, tenemos que

$$\begin{array}{c} \left(\Pi \rhd \Gamma \vdash_C \exists x.A\right)^{\neg\neg} \\ \Updownarrow \\ \Pi^{\neg\neg} \rhd \Gamma^{\neg\neg} \vdash_I \neg_R\forall x.\neg_R\neg_R\neg_R A \end{array}$$

luego, tomando $R = \exists x.A$ la fórmula que buscamos probar,

$$\begin{array}{lll} \Pi^{\neg\neg} \rhd \Gamma^{\neg\neg} \vdash_I \neg_R\forall x.\neg_R\neg_R\neg_R A & & \\ \iff \Gamma^{\neg\neg} \vdash_I \neg_R\forall x.\neg_R A & & \text{(Lema 2)} \\ = \Gamma^{\neg\neg} \vdash_I (\forall x.(A \to R)) \to R & & (\neg_R A = A \to R) \\ = \Gamma^{\neg\neg} \vdash_I (\forall x.(A \to \exists x.A)) \to \exists x.A & & (R = \exists x A) \\ \Rightarrow \Gamma^{\neg\neg} \vdash_I \exists x.A & & \text{(Obs. 2)} \end{array}$$

En deducción natural,

$$\dfrac{\Pi^{\neg\neg} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma^{\neg\neg}, A \vdash_I A}\,\mathrm{Ax}}{\Gamma^{\neg\neg}, A \vdash_I R = \exists x A}\,\mathrm{I}\exists}{\Gamma^{\neg\neg} \vdash_I \neg_R A}\,\mathrm{I}\to}{\Gamma^{\neg\neg} \vdash_I \neg_R\neg_R\neg_R A}\,\mathrm{cut},\ \mathrm{I}\neg_R\neg_R\neg_R}{\Gamma^{\neg\neg} \vdash_I \forall x\neg_R A^{\neg\neg}}\,\mathrm{I}\forall}{\Gamma^{\neg\neg} \vdash_I \neg_R\forall x\neg_R A^{\neg\neg} \qquad \Gamma^{\neg\neg} \vdash_I \exists x.A}\,\mathrm{E}\to$$

□

Obs. En la demostración del Teorema 8 y las de esta sección, luego de aplicar la traducción de Friedman, todas las demostraciones generadas deben ser intuicionistas para que sigan siendo constructivas. No podemos usar LEM.

Obs. 2. $\vdash_I \forall x(A \to \exists x A)$. Trivialmente, para cualquier x si vale A entonces va a existir un x tal que valga A.

Teorema 9 (Traducción de Friedman para fórmulas Π_2 atómicas)**.** Sea Π una demostración clásica de $\forall y_1 \dots \forall y_n.\exists x.A(x, y_1, \dots, y_n)$ y A una fórmula atómica

Si tenemos

$$\Gamma \vdash_C \forall y_1 \dots \forall y_n.\exists x.A(x, y_1, \dots, y_n),$$

podemos generar una demostración intuicionista de la misma fórmula

$$\Gamma^{\neg\neg} \vdash_I \forall y_1 \dots \forall y_n.\exists x.A(x, y_1, \dots, y_n).$$

Demostración. Lo demostramos en deducción natural para un solo $\forall$. Para una cantidad arbitraria es análoga y fácilmente generalizable a partir de esta. La estrategia consiste en primero introducir el $\forall$ reemplazando su variable por una fresca, para evitar conflictos con las variables usadas en la demostración original. Luego se procede a demostrar el $\exists$ de forma análoga al Teorema 8, usando la demostración original traducida pero tomando R como el $\exists$ con la variable ligada por el $\forall$ reemplazada por la fresca en lugar de la fórmula original.

Tomando $R = \exists x.A(x, y_0)$ y aplicando la traducción, tenemos que

$$\begin{array}{c} \left(\Pi \rhd \Gamma \vdash_C \forall y \exists x.A(x,y)\right)^{\neg\neg} \\ \Updownarrow \\ \Pi^{\neg\neg} \rhd \Gamma^{\neg\neg} \vdash_I \forall y \neg_R \forall x.\neg_R \neg_R \neg_R A(x,y) \end{array}$$

Luego

$$\dfrac{\dfrac{\Pi^{\neg\neg}}{\Gamma^{\neg\neg} \vdash_I \forall y \neg_R \forall x.\neg_R A(x,y_0)^{\neg\neg}}\,\mathrm{E}\forall \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma^{\neg\neg}, A(x,y_0) \vdash_I A(x,y_0)}\,\mathrm{Ax}}{\Gamma^{\neg\neg}, A(x,y_0) \vdash_I R = \exists x.A(x,y_0)}\,\mathrm{I}\exists}{\Gamma^{\neg\neg} \vdash_I \neg_R A(x,y_0)}\,\mathrm{I}\to}{\Gamma^{\neg\neg} \vdash_I \neg_R \neg_R \neg_R A(x,y_0)}\,\mathrm{cut},\ \mathrm{I}\neg_R\neg_R\neg_R}{\Gamma^{\neg\neg} \vdash_I \forall x.\neg_R A(x,y_0)^{\neg\neg}}\,\mathrm{I}\forall}{\dfrac{\Gamma^{\neg\neg} \vdash_I \exists x.A(x,y_0)}{\Gamma^{\neg\neg} \vdash_I \forall y \exists x.A(x,y)}\,\mathrm{I}\forall}\,\mathrm{E}\to$$

□

Corolario 1 (Instanciación de $\forall$)**.** El Teorema 9 nos permite realizar la instanciación de las variables del $\forall$ por cualquier término usando E$\forall$. De esa forma, la demostración final es sobre un $\exists$, requerido para poder hacer la extracción sobre la demostración normalizada. Por ejemplo,

$$\dfrac{\dfrac{(9)}{\Gamma^{\neg\neg} \vdash_I \forall y \exists x.A(x,y)}}{\Gamma^{\neg\neg} \vdash_I \exists x.A(x,t)}\,\mathrm{E}\forall$$

5.4.3. Traducción de Friedman para formulas no atómicas

Hasta ahora usamos la traducción de Friedman para traducir las demostraciones de una fórmula de clásica a intuicionista, manteniendo esa fórmula, siempre y cuando su sub-fórmula Σ_0 sea **atómica**. Pero queremos generalizarlo a fórmulas como $\forall y \exists x.\varphi(x,y)$ donde φ no sea atómica, por ejemplo $A(x) \wedge B(y)$. Para ello, la única diferencia es en el uso de cut. Para fórmulas atómicas, tenemos

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma^{\neg\neg} \vdash_I \neg_R A\end{array}}{\Gamma^{\neg\neg} \vdash_I \neg_R A^{\neg\neg} = \neg_R\neg_R\neg_R A^{\neg\neg}}\ \text{cut, I}\neg_R\neg_R\neg_R$$

En donde aprovechamos que la traducción de fórmulas atómicas es $\neg_R A^{\neg\neg} = \neg_R\neg_R\neg_R A$ y podemos llevarla a $\neg_R A$ mediante la eliminación de la triple negación ($\text{I}\neg_R\neg_R\neg_R$). Pero esto no es así para fórmulas no atómicas. Enunciamos el Lema 5, el cual podemos usar para demostrar la traducción en el Teorema 10. No podremos enunciarlo para todas las fórmulas, solo las de una forma en particular, que llamaremos *conjuntivas*.

Def. 18 (Fórmulas conjuntivas)**.** Decimos que una fórmula A es *conjuntiva* si y solo si está generada por la siguiente gramática (conjunciones y fórmulas atómicas)

$$A ::= \bot \mid \top \mid p(t_1, \ldots, t_n) \mid A \wedge A,$$

donde t_i son términos.

Lema 3 (Congruencia de $\neg_R\neg_R$)**.** Si $A \vdash_I A'$, luego $\neg\neg A \vdash_I \neg\neg A'$.

Lema 4 (Distributividad del $\neg_R$ sobre $\wedge$)**.** $\neg_R A \vee \neg_R B \vdash_I \neg_R(A \wedge B)$.

Lema 5 (Introducción de $\neg_R$)**.** Si A es *conjuntiva*, entonces vale $\neg_R A \vdash_I \neg_R A^{\neg\neg}$ y lo notamos con la regla admisible $\text{I}(\neg_R \cdot \neg\neg)$.

Demostración. La demostración es por inducción estructural en la fórmula.

- $\bot, \top$ son triviales. Predicados con $\text{I}\neg_R\neg_R\neg_R$.
- $\wedge$ es cierta para sub-fórmulas que cumplan con la hipótesis inductiva. Tiene algunos trucos.

Veamos esquemáticamente la demostración del $\wedge$.

- Debemos probar $\neg_R(A \wedge B) \vdash_I \neg_R(A \wedge B)^{\neg\neg}$
- Intuitivamente, queremos llevarlo a $\neg_R(A \wedge B) \vdash_I \neg_R A^{\neg\neg} \vee \neg_R B^{\neg\neg}$. En lógica clásica esta demostración requiere el uso de LEM, en particular $\text{E}\neg\neg$ para razonar por el absurdo, que no vale para lógica intuicionista.
- Pero podemos usar un truco para razonar por el absurdo.
  - Tenemos que $\neg_R(A \wedge B)^{\neg\neg}$ es siempre equivalente a $\neg_R\neg_R(\neg_R(A \wedge B)^{\neg\neg})$ por eliminación de triple negación.

- Podemos usar dos lemas auxiliares: $\neg_R A^{\neg\neg} \vee \neg_R B^{\neg\neg} \vdash_I \neg_R(A \wedge B)^{\neg\neg}$ (4) y la congruencia de la doble negación, que al ser doble es covariante: $A \vdash_I A' \Rightarrow \neg_R\neg_R A \vdash_I \neg_R\neg_R A'$ (3) para demostrar

$$\neg_R\neg_R(\neg_R A^{\neg\neg} \vee \neg_R B^{\neg\neg}) \vdash_I \neg_R\neg_R(\neg_R(A \wedge B)^{\neg\neg})$$

- Esto nos permite llevar $\neg_R(A \wedge B) \vdash_I \neg_R(A \wedge B)^{\neg\neg}$ a

$$\neg_R(A \wedge B) \vdash_I \neg_R\neg_R(\neg_R A^{\neg\neg} \vee \neg_R B^{\neg\neg})$$

que se puede demostrar por el absurdo de forma análoga a la demostración clásica bien conocida.

En deducción natural,

$$\dfrac{\dfrac{\dfrac{}{\neg_R A^{\neg\neg} \vee \neg_R B^{\neg\neg} \vdash_I \neg_R(A \wedge B)^{\neg\neg}}\,(4)}{\neg_R\neg_R(\neg_R A^{\neg\neg} \vee \neg_R B^{\neg\neg}) \vdash_I \neg_R\neg_R\neg_R(A \wedge B)^{\neg\neg}}\,(3) \quad \dfrac{\begin{array}{c}\vdots\\ \textit{(dem por el absurdo)}\end{array}}{\Gamma, \neg_R(\neg_R A^{\neg\neg} \vee \neg_R B^{\neg\neg}) \vdash_I R \quad \Gamma \vdash_I \neg_R\neg_R(\neg_R A^{\neg\neg} \vee \neg_R B^{\neg\neg})}\,\mathrm{I}{\to}}{\Gamma \vdash_I \neg_R\neg_R\neg_R(A \wedge B)^{\neg\neg}}\,\mathrm{cut}}{\Gamma = \neg_R(A \wedge B) \vdash_I \neg_R(A \wedge B)^{\neg\neg}}\,\mathrm{cut},\ \mathrm{E}\neg_R\neg_R\neg_R$$

□

Obs. El Lema 5 no podría valer para todas las fórmulas, ya que en ese caso podríamos usar siempre la traducción de Friedman para traducir cualquier demostración, y sabemos que no es posible (porque la lógica clásica no es equivalente a la intuicionista). No obstante, la clasificación de *conjuntivas* que deja fuera a $\neg, \vee, \to, \exists$ y $\forall$ es excesivamente restrictiva, debería ser posible extenderla.

Teorema 10 (Traducción de Friedman para fórmulas Σ_1 en general)**.** Sea Π una demostración clásica de $\forall y_1 \ldots \forall y_n.\exists x.\varphi(x, y_1, \ldots, y_n)$, y φ una fórmula *conjuntiva*. Si tenemos

$$\Gamma \vdash_C \forall y_1 \ldots \forall y_n.\exists x.\varphi(x, y_1, \ldots, y_n),$$

podemos generar una demostración intuicionista de la misma fórmula

$$\Gamma^{\neg\neg} \vdash_I \forall y_1 \ldots \forall y_n.\exists x.\varphi(x, y_1, \ldots, y_n).$$

Demostración. Al igual que el Teorema 9 lo demostramos para un solo $\forall$. La demostración es análoga con la diferencia del uso del Lema 5: $\mathrm{I}(\neg_R \cdot \neg\neg)$. En lugar de tener

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma^{\neg\neg} \vdash_I \neg_R A\end{array}}{\Gamma^{\neg\neg} \vdash_I \neg_R A^{\neg\neg} = \neg_R\neg_R\neg_R A^{\neg\neg}}\,\mathrm{cut},\ \mathrm{I}\neg_R\neg_R\neg_R$$

tenemos

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma^{\neg\neg} \vdash_I \neg_R\varphi\end{array}}{\Gamma^{\neg\neg} \vdash_I \neg_R\varphi^{\neg\neg}}\ \text{cut, I}(\neg_R \cdot \neg\neg)$$

□

Corolario 2 (Traducción de Friedman con instanciación de ∀)**.** De la misma forma que el Corolario 1, podemos usar el Teorema 10 para tener la **versión final de la traducción**, que deja las demostraciones listas para la extracción de testigos.

Sea Π una demostración clásica de $\forall y_1 \dots \forall y_n.\exists x.\varphi(x, y_1, \dots, y_n)$. Si tenemos

$$\Gamma \vdash_C \forall y_1 \dots \forall y_n.\exists x.\varphi(x, y_1, \dots, y_n),$$

podemos generar una demostración intuicionista de la misma fórmula, instanciando las variables cuantificadas universalmente por términos $t_1, \dots, t_n$ cualesquiera (a elección del usuario)

$$\Gamma^{\neg\neg} \vdash_I \exists x.\varphi(x, t_1, \dots, t_n).$$

5.4.4. Traducción de demostraciones

Ya vimos como podemos usar la traducción de Friedman para, dada una traducción de la demostración clásica original, usarla para demostrar la misma fórmula de forma intuicionista. Pero aún no ahondamos en un detalle importante. En el Teorema 7 se introduce la necesidad de extender la traducción de doble negación relativizada de fórmulas a demostraciones, para poder realizar la traducción de una demostración clásica a intuicionista. En esta sección lo vemos más en detalle.

La conversión se efectúa por inducción estructural en la demostración. Para cada regla de inferencia que demuestra A, se genera una demostración a partir de ella para demostrar $A^{\neg\neg}$. La estrategia para hacerlo es similar para todas: recursivamente convertir las subdemostraciones, y usarlas para generar la nueva. Pero hay algunas un poco rebuscadas.

- I∧ (Lema 6), E∧$_1$, E∧$_2$, I→, E→, I∨$_1$, I∨$_2$, I∀, E∀, I¬, E¬, I⊤, Ax son todas similares entre sí, por lo que solo mostramos una.
- I∃ (Lema 7) es una regla simple pero más interesante que las anteriores, por la traducción de ∃.
- LEM (Lema 8) es sumamente interesante, ya que se encuentra en el corazón de la traducción: ¿cómo traducimos el principio de razonamiento clásico que lo separa de la lógica intuicionista?
- E⊥ (Lema 9) se prueba como lema por inducción estructural en la fórmula a demostrar.
- E∨ (Lema 11) y E∃ son análogos y requieren un truco: usar la eliminación de la doble negación. Si bien al ser un principio de razonamiento clásico no vale para lógica intuicionista (por ser equivalente a LEM), lo que si vale es la eliminación de la doble negación relativizada: E$\neg_R\neg_R$ (Lema 10).

Lema 6 (Traducción de I∧)**.** Dada una aparición de la regla I∧,

$$\dfrac{\begin{array}{cc}\Pi_A & \Pi_B \\ \Gamma \vdash_I A & \Gamma \vdash_I B\end{array}}{\Gamma \vdash_I A \wedge B}\,\mathrm{I}\wedge$$

es posible traducirla generando una demostración de $(A \wedge B)^{\neg\neg} = A^{\neg\neg} \wedge B^{\neg\neg}$.

Demostración. Por hipótesis inductiva, tenemos que

$$\Pi_A^{\neg\neg} \rhd \Gamma^{\neg\neg} \vdash_I A^{\neg\neg}$$
$$\Pi_B^{\neg\neg} \rhd \Gamma^{\neg\neg} \vdash_I B^{\neg\neg}$$

Luego, podemos generar una demostración de $A^{\neg\neg} \wedge B^{\neg\neg}$

$$\dfrac{\begin{array}{cc}\Pi_A^{\neg\neg} & \Pi_B^{\neg\neg} \\ \Gamma^{\neg\neg} \vdash_I A^{\neg\neg} & \Gamma^{\neg\neg} \vdash_I B^{\neg\neg}\end{array}}{\Gamma^{\neg\neg} \vdash_I A^{\neg\neg} \wedge B^{\neg\neg}}\,\mathrm{I}\wedge$$

□

Lema 7 (Traducción de I∃)**.** Dada una aparición de la regla I∃,

$$\dfrac{\begin{array}{c}\Pi \\ \Gamma \vdash A\{x := t\}\end{array}}{\Gamma \vdash \exists x.A}\,\mathrm{I}\exists$$

es posible traducirla generando una demostración de $\exists x.A^{\neg\neg} = \neg_R \forall x.\neg_R A^{\neg\neg}$.

Demostración. Por hipótesis inductiva, tenemos que

$$\Pi^{\neg\neg} \rhd \Gamma^{\neg\neg} \vdash_I (A\{x := t\})^{\neg\neg}$$

Luego, podemos generar una demostración de $\neg_R \forall x.\neg_R A^{\neg\neg}$

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash_I \neg_R \forall x A^{\neg\neg}}\,\mathrm{Ax}}{\Gamma_1 \vdash_I \neg_R (A\{x := t\})^{\neg\neg}}\,\mathrm{E}\forall \qquad \begin{array}{c}\Pi^{\neg\neg} \\ \Gamma_1 \vdash_I (A\{x := t\})^{\neg\neg}\end{array}}{\dfrac{\Gamma_1 = \Gamma^{\neg\neg}, \forall x.\neg_R A^{\neg\neg} \vdash_I R}{\Gamma^{\neg\neg} \vdash_I \neg_R \forall x.\neg_R A^{\neg\neg}}\,\mathrm{I}\rightarrow}\,\mathrm{E}\rightarrow$$

□

Lema 8 (Traducción de LEM)**.** Dada una aparición de la regla LEM,

$$\dfrac{}{\Gamma \vdash A \vee \neg A}\,\mathrm{LEM}$$

es posible traducirla generando una demostración de

$$(A \vee \neg A)^{\neg\neg} = \neg_R(\neg_R A^{\neg\neg} \wedge \neg_R \neg_R A^{\neg\neg}).$$

Demostración. Como no hay HI, podemos demostrarlo para cualquier contexto Γ.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash_I \neg_R A^{\neg\neg} \wedge \neg_R \neg_R A^{\neg\neg}}\,\mathrm{Ax}}{\Gamma_1 \vdash_I \neg_R \neg_R A^{\neg\neg}}\,\mathrm{E}\wedge_2 \qquad \dfrac{\dfrac{}{\Gamma_1 \vdash_I \neg_R A^{\neg\neg} \wedge \neg_R \neg_R A^{\neg\neg}}\,\mathrm{Ax}}{\Gamma_1 \vdash_I \neg_R A^{\neg\neg}}\,\mathrm{E}\wedge_1}{\dfrac{\Gamma_1 = \Gamma, \neg_R A^{\neg\neg} \wedge \neg_R \neg_R A^{\neg\neg} \vdash_I R}{\Gamma \vdash_I \neg_R(\neg_R A^{\neg\neg} \wedge \neg_R \neg_R A^{\neg\neg})}\,\mathrm{I}\rightarrow}\,\mathrm{E}\rightarrow$$

□

Lema 9 (Traducción de E⊥)**.** Dada una aparición de la regla E⊥,

$$\dfrac{\begin{array}{c}\Pi_\bot \\ \Gamma \vdash \bot\end{array}}{\Gamma \vdash \varphi}\,\mathrm{E}\bot$$

es posible generar una demostración de $\varphi^{\neg\neg}$ a partir de $\bot^{\neg\neg} = R$.

Demostración. Por hipótesis inductiva (de inducción estructural sobre la demostración), tenemos que

$$\begin{array}{c}(\Pi_\bot \rhd \Gamma \vdash_C \bot)^{\neg\neg} \\ \Updownarrow \\ \Pi_\bot^{\neg\neg} \rhd \Gamma^{\neg\neg} \vdash_I \bot^{\neg\neg} \\ \Updownarrow \\ \Pi_R \rhd \Gamma^{\neg\neg} \vdash_I R\end{array}$$

Pero no es posible demostrar de forma directa $\varphi^{\neg\neg}$ a partir de R. Lo hacemos por inducción estructural en φ. Todos los casos son parecidos. Por ejemplo, veamos un caso inductivo y uno base.

- Dada una conjunción $A \wedge B$, su traducción es $A^{\neg\neg} \wedge B^{\neg\neg}$. Por HI (de inducción estructural sobre la fórmula) a partir de Π_R podemos probar $A^{\neg\neg}$ al igual que $B^{\neg\neg}$. Luego,

$$\dfrac{\begin{array}{cc}\text{(HI)} & \text{(HI)} \\ \Gamma^{\neg\neg} \vdash_I B^{\neg\neg} & \Gamma^{\neg\neg} \vdash_I A^{\neg\neg}\end{array}}{\Gamma^{\neg\neg} \vdash_I A^{\neg\neg} \wedge B^{\neg\neg}}\,\mathrm{I}\wedge$$

- Dada una disyunción $A \vee B$, su traducción es $\neg_R(\neg_R A^{\neg\neg} \wedge \neg_R B^{\neg\neg})$. Luego, podemos demostrarlo sin usar la HI.

$$\dfrac{\begin{array}{c}\Pi_R \\ \Gamma^{\neg\neg}, \neg_R A^{\neg\neg} \wedge \neg_R B^{\neg\neg} \vdash_I R\end{array}}{\Gamma^{\neg\neg} \vdash_I \neg_R(\neg_R A^{\neg\neg} \wedge \neg_R B^{\neg\neg})}\,\mathrm{I}\to$$

El resto de los casos son análogos. □

Lema 10 (Eliminación de doble negación relativizada)**.** Para toda fórmula A, vale

$$\neg_R \neg_R A^{\neg\neg} \vdash_I A^{\neg\neg}.$$

Y se define como regla admisible $\mathrm{E}\neg_R\neg_R$.

Demostración. Se demuestra por inducción estructural en la estructura de A (sin traducir).

- $\bot, \top$ son triviales.

- Predicados, $\neg, \exists, \vee$ son todas iguales: como su traducción comienza por $\neg_R$, su doble negación es una triple negación, y se puede demostrar con el Lema 2 (Eliminación de triple negación relativa).

- $\wedge, \to, \forall$ mantienen su forma luego de la traducción, y su demostración usa la misma técnica. Veamos por ejemplo el caso de la conjunción $A \wedge B$. Por HI tenemos que

$$\neg_R\neg_R A^{\neg\neg} \vdash_I A^{\neg\neg}$$
$$\neg_R\neg_R B^{\neg\neg} \vdash_I B^{\neg\neg}$$

Luego, introducimos la conjunción y usamos la hipótesis inductiva en cada una para razonar por el absurdo. Esto es análogo a como usamos E$\neg\neg$ para razonar por el absurdo en lógica clásica en el Lema 1 (Razonamiento por el absurdo).

$$\frac{\begin{array}{cc}\Pi_L & \Pi_R \\ \neg_R\neg_R(A \wedge B)^{\neg\neg} \vdash_I A^{\neg\neg} & \neg_R\neg_R(A \wedge B)^{\neg\neg} \vdash_I B^{\neg\neg}\end{array}}{\neg_R\neg_R(A \wedge B)^{\neg\neg} \vdash_I A^{\neg\neg} \wedge B^{\neg\neg}}$$

donde Π_L y Π_R son simétricas, y

$$\Pi_L = \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_I \neg_R\neg_R(A \wedge B)^{\neg\neg}}\,\text{Ax} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash_I \neg_R A^{\neg\neg}}\,\text{Ax} \quad \dfrac{\dfrac{}{\Gamma_1 \vdash_I A^{\neg\neg} \wedge B^{\neg\neg}}\,\text{Ax}}{\Gamma_1 \vdash_I A^{\neg\neg}}\,\text{E}\wedge_1}{\Gamma_1 = \Gamma, (A \wedge B)^{\neg\neg} \vdash_I R}\,\text{E}\to}{\Gamma \vdash_I \neg_R(A \wedge B)^{\neg\neg}}\,\text{I}\to}{\Gamma = \neg_R\neg_R(A \wedge B)^{\neg\neg}, \neg_R A^{\neg\neg} \vdash_I R}\,\text{E}\to}{\neg_R\neg_R(A \wedge B)^{\neg\neg} \vdash_I \neg_R\neg_R A^{\neg\neg}}\,\text{I}\to}{\neg_R\neg_R(A \wedge B)^{\neg\neg} \vdash_I A^{\neg\neg}}\,\text{cut, (HI)}$$

□

Lema 11 (Traducción de E$\vee$)**.** Dada una aparición de la regla E$\vee$,

$$\frac{\begin{array}{ccc}\Pi_\vee & \Pi_L & \Pi_R \\ \Gamma \vdash A \vee B & \Gamma, A \vdash C & \Gamma, B \vdash C\end{array}}{\Gamma \vdash C}\,\text{E}\vee$$

podemos usarla para demostrar $C^{\neg\neg}$ a partir de las sub-demostraciones traducidas.

Demostración. Por HI, tenemos

$$\Pi_\vee^{\neg\neg} \rhd \Gamma^{\neg\neg} \vdash_I (A \vee B)^{\neg\neg} = \neg_R(\neg_R A^{\neg\neg} \wedge \neg_R B^{\neg\neg})$$
$$\Pi_L^{\neg\neg} \rhd (\Gamma, A)^{\neg\neg} \vdash_I C^{\neg\neg}$$
$$\Pi_R^{\neg\neg} \rhd (\Gamma, B)^{\neg\neg} \vdash_I C^{\neg\neg}$$

Luego, podemos generar una demostración de $C^{\neg\neg}$ por el absurdo usando E$\neg_R\neg_R$ (Lema 10).

$$\dfrac{\dfrac{\dfrac{\Pi_\vee^{\neg\neg}}{\Gamma^{\neg\neg}, \neg_R C^{\neg\neg} \vdash_I \neg_R(\neg_R A^{\neg\neg} \wedge \neg_R B^{\neg\neg})} \quad \dfrac{\Sigma}{\Gamma^{\neg\neg}, \neg_R C^{\neg\neg} \vdash_I \neg_R A^{\neg\neg} \wedge \neg_R B^{\neg\neg}}\,\mathrm{E}\!\to}{\dfrac{\Gamma^{\neg\neg}, \neg_R C^{\neg\neg} \vdash_I R}{\Gamma^{\neg\neg} \vdash_I \neg_R \neg_R C^{\neg\neg}}\,\mathrm{I}\!\to}}{\Gamma^{\neg\neg} \vdash_I C^{\neg\neg}}\,\mathrm{cut},\ \mathrm{E}\neg_R\neg_R$$

Donde

$$\Sigma = \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash_I \neg_R C^{\neg\neg}}\,\mathrm{Ax} \quad \dfrac{\Pi_L^{\neg\neg}}{\Gamma_1 \vdash_I C^{\neg\neg}}}{\Gamma_1 = \Gamma^{\neg\neg}, \neg_R C^{\neg\neg}, A^{\neg\neg} \vdash_I R}\,\mathrm{E}\!\to}{\Gamma^{\neg\neg}, \neg_R C^{\neg\neg} \vdash_I \neg_R A^{\neg\neg}}\,\mathrm{I}\neg \quad \begin{array}{c}\vdots \\ \textit{(simétrico)} \\ \Gamma^{\neg\neg}, \neg_R C^{\neg\neg} \vdash_I \neg_R B^{\neg\neg}\end{array}}{\Gamma^{\neg\neg}, \neg_R C^{\neg\neg} \vdash_I \neg_R A^{\neg\neg} \wedge \neg_R B^{\neg\neg}}\,\mathrm{I}\wedge$$

□

5.5. Normalización (o reducción)

Repasemos dónde estamos parados. Queremos extraer un testigo de la demostración de un existencial (i.e. extraer de una demostración de $\exists x.p(x)$ a un t tal que $p(t)$). Partimos de una demostración escrita en el lenguaje de alto nivel PPA, que se certifica a una demostración en deducción natural *clásica* (que tiene el problema para la reducción que no es constructiva). Mediante la traducción de Friedman, para cierto tipo de fórmulas (Π_2), podemos convertir la demostración a *intuicionista*. Ahora, queremos extraer el testigo a partir de ella. Eso lo podemos lograr **normalizando** la demostración. ¿Cómo funciona?

Intuitivamente, la estrategia que empleamos consiste en evitar *desvíos superfluos* en una demostración, sucesivamente simplificándolos hasta que queda una sin desvíos, que estará en **forma normal**. Por ejemplo, en la siguiente demostración se prueba $A \to A$ usando $(A \to A) \wedge (B \to B)$ y demostrándo ambos por separado. Pero se puede demostrar de forma más directa, usando la demostración de $A \to A$ y eliminado el desvío por el $\wedge$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\mathrm{Ax}}{\vdash A \to A}\,\mathrm{I}\!\to \quad \dfrac{\dfrac{}{B \vdash B}\,\mathrm{Ax}}{\vdash B \to B}\,\mathrm{I}\!\to}{\vdash (A \to A) \wedge (B \to B)}\,\mathrm{I}\wedge}{\vdash A \to A}\,\mathrm{E}\wedge_1 \quad \rightsquigarrow \quad \dfrac{\dfrac{}{A \vdash A}\,\mathrm{Ax}}{\vdash A \to A}\,\mathrm{I}\!\to$$

Fig. 5.5: Ejemplo de desvío y su normalización

Todos los desvíos que normalizamos se ven de esta forma: una **eliminación** demostrada inmediatamente por su **introducción** correspondiente. Por ejemplo, $\mathrm{E}\wedge_1$ demostrada por $\mathrm{I}\wedge$.

Este proceso es análogo a las reducciones de cálculo λ. Y no solo análogo, sino que existe un *isomorfismo* entre demostraciones en deducción natural y términos de cálculo λ: El isomorfismo Curry-Howard [SU10]. Con él, la normalización de demostraciones isomorfa a las reducciones en cálculo λ, se corresponde a su semántica operacional. Por ejemplo, el isomorfismo nos permite pensar en una conjunción como el tipo de las tuplas, y las eliminaciones como proyecciones. Luego, podemos ver cómo la regla de normalización de la conjunción es isomorfa a la regla de reducción de las proyecciones.

$$\pi_1(\langle M_1, M_2\rangle) \rightsquigarrow M_1$$
$$\pi_2(\langle M_1, M_2\rangle) \rightsquigarrow M_2$$

$$\dfrac{\dfrac{\begin{matrix}\Pi_1 \\ \Gamma \vdash A_1\end{matrix} \quad \begin{matrix}\Pi_2 \\ \Gamma \vdash A_2\end{matrix}}{\Gamma \vdash A_1 \wedge A_2}\,\mathrm{I}\wedge}{\Gamma \vdash A_i}\,\mathrm{E}\wedge_i \quad \rightsquigarrow \quad \begin{matrix}\Pi_i \\ \Gamma \vdash A_i\end{matrix}$$

Fig. 5.6: Relación entre conjunciones y tuplas

De acá en adelante presentamos las reducciones directamente en deducción natural, dado que es lo implementado en `ppa`. Usamos de forma intercambiable *reducción* y *normalización*, pues en el fondo, son lo mismo.

Obs. Es interesante notar que en las reducciones mencionadas en esta sección, no se explicita ni $\vdash_I$ ni $\vdash_C$ a diferencia de la traducción de Friedman. Esto es porque la normalización se puede ejecutar en ambos casos, con la diferencia de que si lo hacemos para una demostración de lógica clásica que use LEM, se "traba" cuando llega ahí y su forma normal no es muy útil. El proceso de normalización no podrá simplificar una demostración que conste de un E$\vee$ que demuestra el $\vee$ con LEM.

5.5.1. Sustituciones

No todas las reglas de reducción son igual de sencillas que $\wedge$. Veamos el caso de la implicación. Una eliminación seguida de una introducción tiene la siguiente forma.

$$\dfrac{\dfrac{\begin{matrix}\Pi_B \\ \Gamma, h : A \vdash B\end{matrix}}{\Gamma \vdash A \to B}\,\mathrm{I}\to_h \quad \begin{matrix}\Pi_A \\ \Gamma \vdash A\end{matrix}}{\Gamma \vdash B}\,\mathrm{E}\to$$

Para eliminar el desvío, uno podría estar tentado a hacer directamente $\Pi_B \rhd \Gamma \vdash B$ pero **¡no sería correcto**! La demostración Π_B requiere la hipótesis $h : A$, que no necesariamente está en Γ, es agregada por $\mathrm{I}\to_h$. Lo correcto sería usar Π_B, pero reemplazando todas las ocurrencias de la hipótesis h por la demostración Π_A. Es necesaria una noción de sustitución de *hipótesis* por demostraciones.

Def. 19 (Hipótesis libres)**.** Una hipótesis ocurre libre en una demostración si se cita sin ser definida. Las reglas que etiquetan y agregan hipótesis al contexto son las que las ligan (i.e. las mencionadas en Subsección 2.3.1 (Hipótesis etiquetadas): Ax_h, $\mathrm{I}\to_h$, $\mathrm{I}\neg_h$, $\mathrm{E}\vee_h$, $\mathrm{E}\exists_h$)

Def. 20 (Hipótesis citadas por una demostración)**.** Definimos el conjunto de hipótesis citadas por una demostración hyps(Π) como todas las hipótesis ligadas y libres que aparecen en ella.

Def. 21 (Sustitución de hipótesis)**.** Notamos como $\Pi\{h := \Sigma\}$ a la sustitución sin capturas de todas las ocurrencias libres de una hipótesis h por una demostración Σ en otra

demostración Π. Una *captura* ocurriría si en alguna sub-demostración de Π se liga una hipótesis $h_2 \neq h$ tal que $h_2 \in \text{hyps}(\Sigma)$ (se cite en Σ).

Para evitar la captura, al igual que en la sustitución de variables, no podemos renombrar en Σ (porque la hipótesis está libre), sino que en la sub-demostración de Π que liga h_2 introduciendo el conflicto la renombramos por una fresca h_2'.

Además, para la reducción del $\exists$ y $\forall$, necesitaremos extender la sustitución usual de variables por términos (Def. 8) a demostraciones. La implementación es análoga, evitando capturas y haciéndolo en una pasada de forma lineal.

Def. 22 (Sustitución de variables en demostraciones)**.** Notamos como $\Pi\{x := t\}$ a la sustitución sin capturas de todas las ocurrencias libres de una variable x por un término t en toda una demostración Π. Incluyendo los contextos.

5.5.2. Algoritmo de reducción

A partir de todas las reglas de reducción (Figura 5.7) podemos implementar el algoritmo. Para acotar el alcance del trabajo, implementamos un mecanismo sencillo e ingenuo, que tiene algunas limitaciones que exploraremos en la siguiente Subsección 5.5.3.

Como las reglas son *pasos* que acercan la demostración sucesivamente a su versión normalizada, podemos implementarlo de forma análoga a la conversión a DNF (Sección 4.6.3) como la clausura reflexiva transitiva de la reducción en un paso: aplicarla 0 o más veces hasta que esté en forma normal. Al igual que DNF, también necesitaremos reglas de congruencia: si no hay una eliminación de una introducción para simplificar, se trata de reducir recursivamente las sub-demostraciones.

En una primera implementación las sub-demostraciones las reducíamos en un paso, de izquierda a derecha. Por ejemplo, si teníamos una introducción de una conjunción,

$$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\, \text{I}\wedge$$

reducíamos de a un paso a la vez $A \rightsquigarrow A_1 \rightsquigarrow A_2 \rightsquigarrow \ldots \rightsquigarrow A^*$ hasta llegar a A^* irreducible y recién ahí aplicamos mismo para B, paso por paso. Esto resultó excesivamente lento, dado que las demostraciones a reducir son muy grandes, porque son generadas automáticamente. Además, si la sub-demostración que había que reducir estaba muy anidada, había que recorrerla toda cada vez para efectuar cada paso.

$$\dfrac{\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\, \text{I}\wedge}{\vdots}$$
$$\Pi$$

Las sustituciones también eran más costosas, porque tenían que recorrer demostraciones de una profundidad mayor, y hay que recorrer el árbol entero para ver las hipótesis citadas y efectuar la sustitución en si.

Para solucionar este problema, implementamos una reducción en muchos pasos. En la literatura se pueden encontrar muchas *estrategias de reducción* alternativas, que se clasifican en si reducen en un paso o muchos para cada iteración de la clausura. Dentro de las de muchos, la que implementamos fue **Gross-Knuth**, que reduce en muchos pasos todos los sub-términos posibles al mismo tiempo. Aplicado a demostraciones, quiere decir que en un único paso de reducción, reduciríamos $A \wedge B \rightsquigarrow A^* \wedge B^*$.

$$\dfrac{\dfrac{\begin{matrix}\Pi_1\\ \Gamma \vdash A_1\end{matrix} \quad \begin{matrix}\Pi_2\\ \Gamma \vdash A_2\end{matrix}}{\Gamma \vdash A_1 \wedge A_2}\,\mathrm{I}\wedge}{\Gamma \vdash A_i}\,\mathrm{E}\wedge_i \quad \rightsquigarrow \quad \begin{matrix}\Pi_i\\ \Gamma \vdash A_i\end{matrix}$$

$$\dfrac{\dfrac{\begin{matrix}\Pi_B\\ \Gamma, h : A \vdash B\end{matrix}}{\Gamma \vdash A \to B}\,\mathrm{I}\to_h \quad \begin{matrix}\Pi_A\\ \Gamma \vdash A\end{matrix}}{\Gamma \vdash B}\,\mathrm{E}\to \quad \rightsquigarrow \quad \left(\begin{matrix}\Pi_B\\ \Gamma, h : A \vdash B\end{matrix}\right)\{h := \Pi_A\}$$

$$\dfrac{\dfrac{\begin{matrix}\Pi_{A_i}\\ \Gamma \vdash A_i\end{matrix}}{\Gamma \vdash A_1 \vee A_2}\,\mathrm{I}\vee_i \quad \begin{matrix}\Pi_1\\ \Gamma, h_1 : A_1 \vdash B\end{matrix} \quad \begin{matrix}\Pi_2\\ \Gamma, h_2 : A_2 \vdash B\end{matrix}}{\Gamma \vdash B}\,\mathrm{E}\vee$$
$$\rightsquigarrow \quad \left(\begin{matrix}\Pi_i\\ \Gamma, h_i : A_i \vdash B\end{matrix}\right)\{h_i := \Pi_{A_i}\}$$

$$\dfrac{\dfrac{\begin{matrix}\Pi_\bot\\ \Gamma, h : A \vdash \bot\end{matrix}}{\Gamma \vdash \neg A}\,\mathrm{I}\neg \quad \begin{matrix}\Pi_A\\ \Gamma \vdash A\end{matrix}}{\Gamma \vdash \bot}\,\mathrm{E}\neg \quad \rightsquigarrow \quad \left(\begin{matrix}\Pi_\bot\\ \Gamma, h : A \vdash \bot\end{matrix}\right)\{h := \Pi_A\}$$

$$\dfrac{\dfrac{\begin{matrix}\Pi\\ \Gamma \vdash A\end{matrix}}{\Gamma \vdash \forall.xA}\,\mathrm{I}\forall}{\Gamma \vdash A\{x := t\}}\,\mathrm{E}\forall \quad \rightsquigarrow \quad \left(\begin{matrix}\Pi\\ \Gamma \vdash A\end{matrix}\right)\{x := t\}$$

$$\dfrac{\dfrac{\begin{matrix}\Pi_A\\ \Gamma \vdash A\{x := t\}\end{matrix}}{\Gamma \vdash \exists x.A}\,\mathrm{I}\exists \quad \begin{matrix}\Pi_B\\ \Gamma, h : A \vdash B\end{matrix}}{\Gamma \vdash B} \quad \rightsquigarrow \quad \left(\left(\begin{matrix}\Pi_B\\ \Gamma, h : A \vdash B\end{matrix}\right)\{x := t\}\right)\{h := \Pi_A\}$$

Fig. 5.7: Reglas de reducción

5.5.3. Limitaciones

El mecanismo sencillo de reducción que implementamos tiene dos desventajas: no contempla todas las reglas de reducción posible, y es ineficiente. Comencemos por la primera. Como solamente reduce eliminaciones de introducciones de la misma regla, no contempla las "reducciones permutativas", que mezclan introducciones y eliminaciones de conectivos distintos. Si estos patrones ocurren, no nos permitirán llegar a una forma normal útil. Entonces no siempre podremos extraer un testigo. Por ejemplo, en la Figura 5.8 (Ejemplo de demostración con forma normal que no sirve para la extracción de testigos), se puede observar una demostración sencilla que usa **cases**. Pero al certificarla, traducirla y reducirla, no queda en una forma normal útil: aparece dos veces I∃, cuando se podría mover hacia la raíz del árbol y evitar repetirlo en cada rama del E∨. Esto hace que al intentar extraer un testigo, el programa reporte una falla. Para soportar estos casos sería necesario sofisticar la reducción agregando reducciones permutativas, que quedó fuera del alcance del trabajo.

Por otro lado, también es ineficiente: en cada paso reinicia la búsqueda de todos los focos de evaluación. Se podría mejorar usando una máquina abstracta que implemente reducción a forma normal, inspirada por ejemplo en la máquina de Crégut para reducción *call-by-name* fuerte [Cré07] o la máquina de Biernacka para reducción *call-by-need* fuerte [BC19].

5.6. Manteniendo el contexto

Describimos todo el proceso de cómo a partir de una demostración en PPA, se certifica, se traduce mediante Friedman y se reduce para extraer testigos de un existencial. Pero falta exponer un detalle fundamental. Cuando estábamos probando la integración de todas las partes, nos dimos cuenta de un problema. `ppa` implementa el resultado del Corolario 2 (Traducción de Friedman con instanciación de ∀), que nos permite llevar las demostraciones a esta forma

$$\mathbf{\Gamma}^{\neg\neg} \vdash_I \exists x.\varphi(x, t_1, \ldots, t_n).$$

Pero si tenemos por ejemplo el programa de la Figura 5.1 (Extracción simple),

```
1 axiom ax: p(v)
2 theorem t: exists X . p(X)
3 proof
4    take X := v
5    thus p(v) by ax
6 end
```

Al certificarlo y traducirlo, la demostración resultante es válida **con el contexto traducido**. Este está conformado por los axiomas, entonces la demostración traducida es de

$$\neg_R\neg_R p(v) \vdash_I \exists x.p(x)$$

que al reducirlo, queda

$$\dfrac{\dfrac{}{\neg_R\neg_R p(v) \vdash_I \neg_R\neg_R p(v)}\,\mathrm{Ax} \qquad \dfrac{\dfrac{\dfrac{}{\neg_R\neg_R p(v), p(v) \vdash_I p(v)}\,\mathrm{Ax}}{\neg_R\neg_R p(v), p(v) \vdash_I R = \exists x.p(x)}\,\mathrm{I}\exists}{\neg_R\neg_R p(v) \vdash_I \neg_R p(v)}\,\mathrm{I}\to}{\neg_R\neg_R p(v) \vdash_I \exists x.p(x)}\,\mathrm{E}\to$$

¡Nunca llega a la forma normal deseada! Conceptualmente, para que exista la forma normal debería poder comenzar con I∃ y demostrar de alguna forma $\neg_R\neg_R p(v) \vdash_I p(v)$, que no es posible. Otra forma más práctica de verlo es que con las reglas que definimos no se puede reducir, porque se elimina una implicación que se demuestra mediante Ax de un axioma en vez de I→, por lo que se traba.

Esto es análogo al problema que tendríamos si intentamos normalizar la demostración de un teorema que cita a otros, sin insertar las demostraciones de ellos. Cuando llega a la cita como axioma, se queda trabado.

La solución que implementamos fue **mantener los axiomas originales**, y demostrar la introducción de la traducción (Lema 12, regla $\mathrm{I}^{\neg\neg}$): $p(v) \vdash_I p(v)^{\neg\neg}$. Una vez generada la demostración de Friedman para la traducción se efectúa un paso más: por cada axioma, reemplazar cada lugar en donde se cita (que pretende encontrarlo traducido) por la demostración de la traducción *a partir del axioma original*.

$$\dfrac{\dfrac{}{p(v) \vdash_I \neg_R \neg_R p(v)}\,\mathrm{I}^{\neg\neg} \qquad \dfrac{\dfrac{\dfrac{}{p(v) \vdash_I p(v)}\,\mathrm{Ax}}{p(v) \vdash_I R = \exists x.p(x)}\,\mathrm{I}\exists}{p(v) \vdash_I \neg_R p(v)}\,\mathrm{I}\to}{p(v) \vdash_I \exists x.p(x)}\,\mathrm{E}\to$$

De esa forma, como los axiomas se mantienen, existe la forma normal que buscamos y nuestra reducción llega a ella para este caso.

$$\dfrac{\dfrac{}{p(v) \vdash_I p(v)}\,\mathrm{Ax}}{p(v) \vdash_I \exists x.p(x)}\,\mathrm{I}\exists$$

Esto no valdrá para todas las fórmulas, lo que restringirá las axiomatizaciones. **Solo podrán ser axiomas las que llamaremos *F-fórmulas*.**

Def. 23 (F-fórmulas)**.** Decimos que B es una *F-fórmula* si está generada por la siguiente gramática:

$$\begin{aligned} B ::= {} & p(t_1, \ldots, t_n) \mid \bot \mid \top \\ & \mid B \wedge B \mid B \vee B \\ & \mid \forall x.B \mid \exists x.B \\ & \mid A \to B \\ & \mid \neg A \end{aligned}$$

Donde A son fórmulas *conjuntivas* (Def. 18).

Lema 12 (Introducción de la traducción $\neg\neg$)**.** Si B es una *F-fórmula*, vale $B \vdash_I B^{\neg\neg}$.

Demostración. Se demuestra por inducción estructural en la fórmula (sin traducir).

- Los casos de predicados, $\top, \bot, \vee, \wedge, \forall, \exists$ salen directo.
- $\neg$ y $\to$ son las rebuscadas. Al ser contra variantes, no se pueden demostrar de forma directa porque no alcanza con la HI, sino que es necesario $A^{\neg\neg} \vdash_I A$ que en general no vale. Pero sí es posible reducirlo a $\neg_R A \vdash_I \neg_R A^{\neg\neg}$ (Lema 5, $\mathrm{I}(\neg_R \cdot \neg\neg)$).

  La negación es la más sencilla

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg A, A \vdash_I \neg A}\,\mathrm{Ax} \qquad \dfrac{}{\neg A, A \vdash_I A}\,\mathrm{Ax}}{\neg A, A \vdash_I R}\,\mathrm{E}\bot}{\neg A \vdash_I \neg_R A}\,\mathrm{I}\to}{\neg A \vdash_I \neg_R A^{\neg\neg}}\,\mathrm{cut},\ \mathrm{I}(\neg_R \cdot \neg\neg)$$

  La implicación no tanto. Veamos la estrategia.

- Debemos probar $A \to B \vdash_I A^{\neg\neg} \to B^{\neg\neg}$.
- Lo vamos a reducir a demostrar los contrarrecíprocos con cut

$$\neg_R B \to \neg_R A \vdash_I \neg_R B^{\neg\neg} \to \neg_R A^{\neg\neg}$$

  Para ello necesitamos los siguientes lemas, sencillos de demostrar.

- $A \to B \vdash_I \neg_R B \to \neg_R A$
- $\neg_R B^{\neg\neg} \to \neg_R A^{\neg\neg} \vdash_I A^{\neg\neg} \to B^{\neg\neg}$ (requiere E$\neg_R\neg_R$)

- Reducimos a $\neg_R A \vdash_I \neg_R A^{\neg\neg}$ para el antecedente y usamos la HI para el consecuente ($B \vdash_I B^{\neg\neg}$) y concluimos.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash_I \neg_R B \to \neg_R A}\,\text{Ax} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma_1, B \vdash_I \neg_R B^{\neg\neg}}\,\text{Ax} \qquad \Gamma_1, B \vdash_I B^{\neg\neg}}{\Gamma_1, B \vdash_I R}\,\text{E}\to}{\Gamma_1 \vdash_I \neg_R B}\,\text{I}\to}{\Gamma_1 \vdash_I \neg_R A}\,\text{E}\to \quad \text{(HI)}}{\Gamma_1 = \neg_R B \to \neg_R A, \neg_R B^{\neg\neg} \vdash_I \neg_R A^{\neg\neg}}\,\text{cut, I}(\neg_R \cdot \neg\neg)}{\neg_R B \to \neg_R A \vdash_I \neg_R B^{\neg\neg} \to \neg_R A^{\neg\neg}}\,\text{I}\to$$

□

Obs (Completitud)**.** Vimos que el Lema 12 (Introducción de la traducción $\neg\neg$), $A \vdash_I A^{\neg\neg}$, se reduce al Lema 5 (Introducción de $\neg_R$), que cumplen $\neg_R A \vdash_I \neg_R A^{\neg\neg}$. Las *F-fórmulas* se ven restringidas por las *conjuntivas*.

Por otro lado, en [Sel92], se exhibe el contra ejemplo $\neg\neg a$ para $A \vdash_I A^{\neg\neg}$ (en una versión distinta pero equivalente de la traducción de Friedman). En nuestro caso, como el Lema 12 se reduce a Lema 5, si el primero no vale en general (por el contra ejemplo) entonces el segundo tampoco podría. Esto es consistente con nuestros resultados, tampoco podemos demostrar $\neg\neg a \vdash_I (\neg\neg a)^{\neg\neg}$ pues no la consideramos como *F-fórmula*.

Esto resultó suficientemente permisivo para los axiomas que se suelen definir.

Obs. 3 (Similitud con fórmulas de Harrop)**.** Aparentemente, la noción de F-fórmulas a la que llegamos en Def. 23 se vincula con las fórmulas hereditarias de Harrop [GR97], definidas por la siguiente gramática:

$$\begin{aligned}
A &::= p(t_1, \ldots, t_n) \mid \bot \mid \top \\
G &::= A \\
&\quad \mid G \wedge G \mid G \vee G \\
&\quad \mid \forall x.G \mid \exists x.G \\
&\quad \mid H \to G \\
H &::= A \\
&\quad \mid H \wedge H \\
&\quad \mid \forall x.H \\
&\quad \mid G \to A
\end{aligned}$$

Donde A son fórmulas atómicas, G son *G-fórmulas* (similares a las F-fórmulas) y H son fórmulas *Harrop Hereditarias* (similares a las conjuntivas). Habría que estudiar este posible vínculo más en profundidad en el futuro.

Sería llamativo que sean iguales, pues las fórmulas *conjuntivas* que definimos no incluyen $\forall$ y las hereditarias sí. Por cómo son usadas en el Teorema 10, que incluyan $\forall$ implicaría que podríamos usar la traducción de Friedman para fórmulas Π_3 de la forma $\forall x.\exists y.\forall z.\varphi$ con φ sin cuantificadores.

5.7. Otros métodos de extracción

A lo largo de este capítulo presentamos nuestra estrategia para extracción de testigos de demostraciones de lógica clásica, que tienen el desafío de no siempre ser constructivas por la existencia del principio de razonamiento clásico LEM. Pero hay más formas de hacerlo. Vimos que las reglas de reducción en lógica intuicionista en realidad no son más que una semántica operacional del cálculo λ, por Curry-Howard. Muchas veces se usan isomorfismos como este, por ser más fácil pensar en semánticas operacionales que normalización de demostraciones. El problema con la lógica clásica es que no es sencillo dar esa interpretación computable, de una forma tal que permita hacer la extracción. Los métodos para hacerlo se dividen en dos categorías a grandes rasgos: directos e indirectos [Miq11].

Los **indirectos** traducen las demostraciones clásicas de un subconjunto de fórmulas a otras lógicas que se porten mejor, como la intuicionista. Aprovechan el hecho de que es constructiva y tiene una interpretación computable bien conocida. Se puede efectuar con diferentes tipos de traducciones negativas, como la traducción de Friedman, que es lo que hicimos en este trabajo.

Por otro lado, también se puede hacer de forma **directa**, dando una interpretación computable de la lógica clásica. A esto se le llama *realizabilidad clásica*, que consiste en dar una semántica para cálculos λ clásicos [Miq17].

(a) Programa de PPA

```
1  axiom ax_1: p(k) | q(k)
2  axiom ax_2: p(k) -> r(k)
3  axiom ax_3: q(k) -> r(k)
4
5  theorem t: exists Y . r(Y)
6  proof
7     take Y := k
8     cases by ax_1
9        case p(k)
10          hence r(k) by ax_2
11
12       case q(k)
13          hence r(k) by ax_3
14    end
15 end
```

(b) Demostración certificada, traducida y reducida (generada por `ppa`)

$$\dfrac{\dfrac{}{\Gamma \vdash_I p(k) \vee q(k)}\,\mathrm{Ax} \qquad \begin{matrix}\Pi_L & \Pi_R \\ \Gamma, p(k) \vdash_I \exists y.r(y) & \Gamma, q(k) \vdash_I \exists y.r(y)\end{matrix}}{\Gamma \vdash_I \exists y.r(y)}\,\mathrm{E}\vee$$

con Π_L definido de la siguiente forma, y Π_R simétrico.

$$\Pi_L = \dfrac{\dfrac{}{\Gamma, p(k) \vdash_I p(k) \to r(k)}\,\mathrm{Ax} \qquad \dfrac{}{\Gamma, p(k) \vdash_I p(k)}\,\mathrm{Ax}}{\dfrac{\Gamma, p(k) \vdash_I r(k)}{\Gamma, p(k) \vdash_I \exists y.r(y)}\,\mathrm{I}\exists}$$

(c) Demostración en forma normal (deseada)

$$\dfrac{\dfrac{}{\Gamma \vdash_I p(k) \vee q(k)}\,\mathrm{Ax} \qquad \begin{matrix}\Pi_L & \Pi_R \\ \Gamma, p(k) \vdash_I r(y) & \Gamma, q(k) \vdash_I r(k)\end{matrix}}{\dfrac{\Gamma \vdash_I r(k)}{\Gamma \vdash_I \exists y.r(y)}\,\mathrm{I}\exists}\,\mathrm{E}\vee$$

con Π_L y Π_R análogas al caso anterior, pero sin I$\exists$.

Fig. 5.8: Ejemplo de demostración con forma normal que no sirve para la extracción de testigos

6. LA HERRAMIENTA ppa

En el Capítulo 3 (El lenguaje PPA) vimos a PPA como un lenguaje, desde el punto de vista de un usuario. En Capítulo 4 (El certificador de PPA) abordamos el funcionamiento interno, que expandimos en Capítulo 5 (Extracción de testigos de existenciales) con extracción de testigos. En este capítulo presentamos la herramienta de línea de comandos `ppa`, que permite ejecutar programas del lenguaje. Está implementada en Haskell.

6.1. Instalación

Para instalar `ppa` a partir de los fuentes, seguir los siguientes pasos.

1. Instalar Haskell y Cabal.
2. Clonar el repositorio de `ppa` o descargarlo `https://github.com/mnPanic/tesis`.
3. Instalar la herramienta con cabal.

   ```
   tesis/ppa:~$ cabal install ppa
   ```

4. Verificar la correcta instalación.

   ```
   tesis/ppa:~$ ppa version
   ppa version 0.1.0.0
   ```

6.2. Interfaz y ejemplos

El ejecutable `ppa` cuenta con dos comandos: `check` (certificado y chequeo de programas) y `extract` (extracción de testigos). En el repositorio de los fuentes, el directorio `ppa/doc/examples` contiene ejemplos de programas. La extensión por convención de programas de `ppa` es `.ppa`.

6.2.1. `check` - Chequeo de programas

Permite certificar y chequear programas.

```
ppa check <in> <args>
```

Argumentos:

- El primer argumento posicional es el archivo que contiene el programa a certificar. Puede ser `-` para leer de la entrada estándar *stdin*.
- `-out`, `-o` (*opcional*): Path para el archivo al cual escribir el certificado (con el sufijo `_raw.nk`) o `-` para usar la salida estándar *stdout*.

Primero lee, certifica y chequea el programa reportando el resultado. En caso de haber proporcionado un archivo de output, escribe el certificado de deducción natural a `<path>_raw.nk`.

Ejemplos:

```
$ ppa check doc/examples/parientes.ppa
Checking... OK!
```

```
$ ppa check doc/examples/parientes.ppa --out out
Checking... OK!
Writing...
Wrote raw to out_raw.nk
```

6.2.2. `extract` - Extracción de testigos

Permite ejecutar la extracción de testigos.

```
ppa extract <in> <args>
```

Argumentos:

- El primer argumento posicional es el archivo de entrada que contiene el programa. Puede ser `-` para leer de la entrada estándar *stdin*.
- `-theorem`, `-t` (*obligatorio*): el nombre del teorema para el cual extraer el testigo
- `-terms`, `ts`: la lista de términos que indica cómo instanciar las variables cuantificadas universalmente. Debe contener la misma cantidad de términos que variables.
- `-out`, `-o` (*opcional*): Path para el archivo al cual escribir los certificados (con los sufijos `_raw.nk` y `.nj`) o `-` para usar la salida estándar *stdout*.

Primero lee, certifica y chequea el programa. Luego, lo traduce con la traducción de Friedman, normaliza la demostración e intenta extraer un testigo, reportando cual fue el testigo extraído y cómo queda el término final con el testigo y los términos proporcionados para instanciar las variables cuantificadas universalmente.

Por ejemplo, si tenemos el programa de Figura 5.2

```
$ ppa extract doc/listings/extract/forall.ppa --theorem t --terms x
Running program... OK!
Translating... OK!
Checking translated... OK!
Extracted witness: v
of formula: p(x, v)
```

Y especificando output,

```
$ ppa extract doc/listings/extract/forall.ppa --theorem t --terms x -o out
Running program... OK!
Translating... OK!
Writing...
Wrote raw to out_raw.nk
Wrote translated+reduced to out.nj
Checking translated... OK!
Extracted witness: v
of formula: p(x, v)
```

6.3. Detalles de implementación

En esta sección contamos algunos detalles relevantes sobre la implementación, que deberían facilitar la navegación de los archivos fuente. Comencemos por la arquitectura de los módulos.

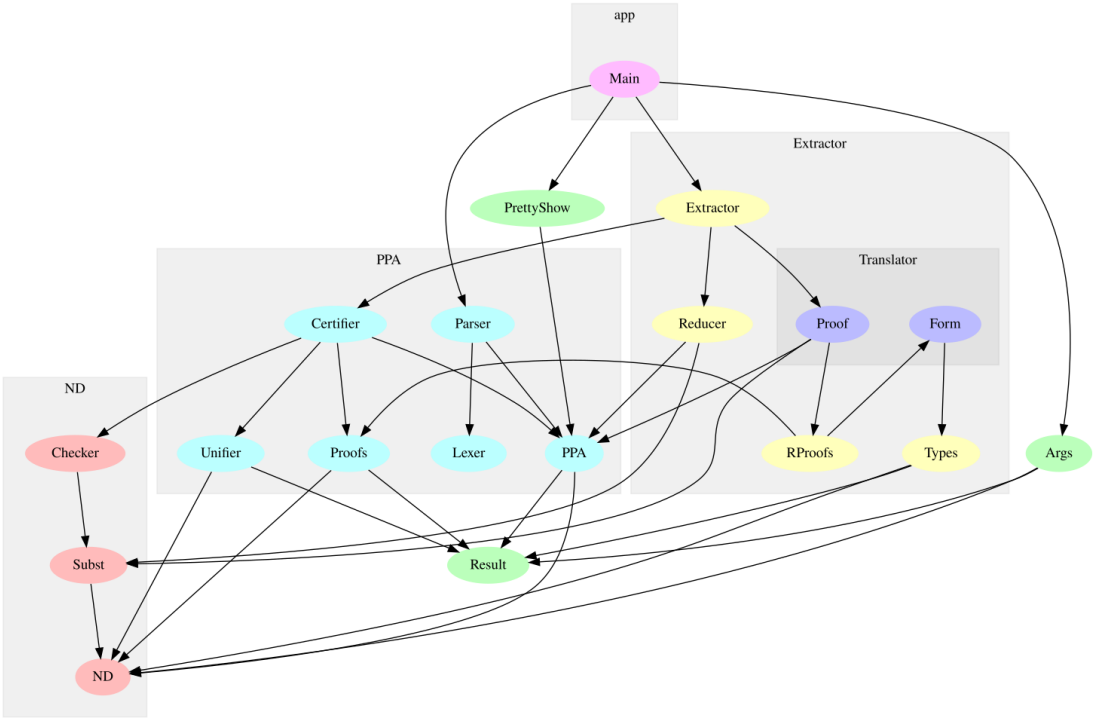

Fig. 6.1: Grafo de módulos del programa `ppa` generado con `graphmod`

Los más relevantes son,

- `ND`: Contiene el modelo central de deducción natural usado para los certificados. Fórmulas, términos y demostraciones. Tiene 3 sub módulos: `ND.Checker` (chequeador de demostraciones), `ND.Subst` (sustituciones sobre fórmulas y demostraciones) y `ND.ND` (el modelo en si). Ver Subsección 6.3.2 (Modelado de deducción natural).
- `PPA`: Contiene la implementación del lenguaje PPA.
  - `PPA.Parser` y `PPA.Lexer` son el lexer y el parser respectivamente. Ver Subsección 6.3.1 (Parser y Lexer).
  - `PPA.Certifier` implementa el certificador, que genera demostraciones de deducción natural a partir de programas de PPA. Usa `PPA.Unifier` para unificación (usado en eliminación de $\forall$) y `PPA.Proofs` para demostraciones de equivalencias en ND (usado en DNF).
- `Extractor`: Contiene la lógica de extracción de testigos. Tiene varios sub módulos
  - `Extractor.Translator` implementa la traducción de Friedman en sus dos submódulos, `Extractor.Translator.Proof` para demostraciones y `Extractor.Translator.Form` para fórmulas.

- `Extractor.Reducer` implementa la normalización.
- `Extractor.Extractor` combina todo lo anterior para hacer la extracción de testigos de principio a fin, certificando, traduciendo y reduciendo. Usa `Extractor.RProofs` que es análogo a `PPA.Proofs` e implementa los lemas necesarios sobre demostraciones intuicionistas con negación relativizada.

6.3.1. Parser y Lexer

Una parte esencial de `ppa` es el compilador, que permite tomar un programa desde un archivo de texto y llevarlo a una representación estructurada, como un un tipo abstracto de datos, para el uso en el resto del programa. Se implementa en dos etapas.

- La primera es el análisis léxico o *lexer*, que convierte el texto plano en una lista de *lexemas* o *tokens*. Se puede implementar con expresiones regulares.
- Una vez que el programa ya fue convertido en *lexemas*, es procesado por el parser, que interpreta las estructuras sintácticas del lenguaje. Puede generar una representación intermedia como un tipo abstracto de datos.

Para el parsing en general hay dos caminos bien conocidos. O bien hacerlo a mano con alguna biblioteca de *parser combinators* como parsec, o usar un *parser generator*: un programa que genere el código de un *parser* automáticamente a partir de una gramática en algún formato, como EBNF. Nosotros decidimos, por familiaridad con el proceso, usar un *parser generator*. Los más conocidos históricamente son Lex (para el lexer) y Yacc (para el parser, *yet another compiler compiler*). En Haskell existen dos paquetes análogos: Alex para el lexer y Happy para el parser. Estos generan un lexer a partir de expresiones regulares y un parser respectivamente a partir de una gramática en un formato similar a EBNF.

En la Figura 6.2 se puede ver un extracto del archivo de input de Alex y en Figura 6.3 uno del de Happy de `ppa`. La implementación del parser solo construye términos de programas (definidos en `PPA.PPA`) y fórmulas (`ND.ND`) a partir de los cuales el programa opera.

6.3.2. Modelado de deducción natural

La parte más interesante del modelado son las demostraciones de deducción natural, que se pueden ver en la Figura 6.6. Una demostración está compuesta por la aplicación recursiva de reglas de inferencia, y su modelo omite todos los detalles que se pueden inferir durante su chequeo. De esa forma las demostraciones son más fáciles de escribir y generar, al costo de ser un poco más costosas de leer por un humano, cosa que de todas formas no debería ser usual. Por ejemplo, la introducción de la implicación, I→, modelada por `PImpI` no especifica cuál es la implicación que se está introduciendo, dado que durante el chequeo debería ser la fórmula actual a demostrar. Tampoco se explicita cual es el contexto de demostración. Se genera de forma dinámica.

```
tokens :-
    $white+                         ;
    "//".*                          ; -- comments
    "/*"(.|(\r\n|\r|\n))*"*/"       ; -- block comments
    \.              { literal TokenDot }
    \,              { literal TokenComma }
    \&              { literal TokenAnd }
    \|              { literal TokenOr }
    true            { literal TokenTrue }
    false           { literal TokenFalse }
    \-\>            { literal TokenImp }
    \<\-\>          { literal TokenIff }
    \~              { literal TokenNot }
    exists          { literal TokenExists }
    forall          { literal TokenForall }
    \(              { literal TokenParenOpen }
    \)              { literal TokenParenClose }
    axiom           { literal TokenAxiom }
    theorem         { literal TokenTheorem }
    proof           { literal TokenProof }
    end             { literal TokenEnd }
    \;              { literal TokenSemicolon }
    \:              { literal TokenDoubleColon }
    suppose         { literal TokenSuppose }
    thus            { literal TokenThus }
    hence           { literal TokenHence }
    have            { literal TokenHave }
    then            { literal TokenThen }
    by              { literal TokenBy }
    equivalently    { literal TokenEquivalently }
    claim           { literal TokenClaim }
    cases           { literal TokenCases }
    case            { literal TokenCase }
    take            { literal TokenTake }
    \:\=            { literal TokenAssign }
    st              { literal TokenSuchThat }
    consider        { literal TokenConsider }
    let             { literal TokenLet }

    \"[^\"]*\"          { lex (TokenQuotedName . firstLast) }

    (\_|[A-Z])[a-zA-Z0-9\_\-]*(\')*                     { lex TokenVar }
    [a-zA-Z0-9\_\-\?!#\$\%\*\+\<\>\=\?\@\^]+(\')*       { lex TokenId }
```

Fig. 6.2: Extracto del lexer de `ppa`

```
%token
    -- Enumeración de tokens del lexer

Prog    : Declarations

Declarations : Declaration Declarations
               | Declaration

Declaration : Axiom
            | Theorem

Axiom : axiom Name ':' Form

Theorem : theorem Name ':' Form proof Proof end

Proof   : ProofStep Proof
        | {- empty -}

ProofStep : suppose Name ':' Form
            | thus Form OptionalBy
            | hence Form OptionalBy
            | have Name ':' Form OptionalBy
            | then Name ':' Form OptionalBy
            | equivalently Form
            | claim Name ':' Form proof Proof end
            | cases OptionalBy Cases end
            | take var ':=' Term
            | let var
            | consider var st Name ':' Form by Justification

Cases   : Case Cases
        | {- empty -}

Case    : case Form Proof
        | case Name ':' Form Proof

OptionalBy : by Justification
OptionalBy : {- empty -}

Justification : Name ',' Justification
               | Name

Name    : id
        | name
```

Fig. 6.3: Extracto del parser de `ppa` (programas)

```
-- Resolución automática de conflictos shift/reduce
%right exists forall dot
%right imp iff
%left and or
%nonassoc not
%%

Form    : id TermArgs
        | Form and Form
        | Form or Form
        | Form imp Form
        | Form iff Form
        | not Form
        | exists var dot Form
        | forall var dot Form
        | true
        | false
        | '(' Form ')'

Term    : var
        | id TermArgs

TermArgs : {- empty -}
           | '(' Terms ')'

Terms   : Term
        | Term ',' Terms
```

Fig. 6.4: Extracto del parser de `ppa` (lógica de primer orden)

```
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. 6.5: Modelado de fórmulas y términos de LPO

```
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. 6.6: Modelado de reglas de inferencia para demostraciones

7. CONCLUSIONES

En este trabajo presentamos el lenguaje PPA junto con los detalles de su implementación en `ppa`. Primero dimos una definición completa del sistema lógico de deducción natural, junto con ejemplos de demostraciones. Describimos cómo implementamos el algoritmo de chequeo, y por otro lado los de alfa equivalencia y sustitución sin capturas en tiempo cuasilineal en peor caso.

Dimos un manual de usuario para el lenguaje PPA, explicando como escribir demostraciones y cómo usar el mecanismo de demostración principal **`by`**. Dimos una demostración completa que muestra diferentes capacidades del lenguaje. También se listaron todos los comandos, ejemplos funcionales para cada uno, y su relación con las reglas de inferencia de deducción natural. Profundizamos en la implementación del certificador y cómo están implementadas cada una de las partes de la interfaz. Centralmente el *solver* usado por debajo del **`by`**.

Finalmente vimos una implementación posible de extracción de testigos existenciales. Mencionamos las limitaciones al tratar de hacerlo de forma directa sobre lógica clásica, las distintas versiones de la traducción de Friedman según el tipo de fórmula a demostrar, y las limitaciones que se presentaron (no se podrá asumir cualquier axioma, ni demostrar cualquier fórmula Π_2). Luego describimos cómo a partir del isomorfismo Curry-Howard pudimos implementar un mecanismo de normalización de demostraciones, que también presentó limitaciones (no todas las demostraciones podrán ser llevadas a su forma normal) y requirió cambios en la estrategia de reducción de una sencilla a una más sofisticada (Gross-Knuth) debido al tamaño de las demostraciones.

El principal aporte del trabajo es la implementación práctica de un método de extracción indirecto mediante el uso de la traducción de Friedman y la normalización usual de la lógica intuicionista.

7.1. Trabajo futuro

Surgieron varias líneas de trabajo futuro, que quedaron fuera del alcance de la tesis. Las listamos a continuación.

- **Modelar de forma nativa inducción e igualdad**: las teorías que se pueden axiomatizar están limitadas al no poder representar inducción de forma nativa (como predica sobre predicados, es lógica de segundo orden). Si bien sí se puede axiomatizar de forma *adhoc*, sería más amigable de estar como regla de inferencia y a lo largo de todo el programa. También se podría agregar de forma nativa la noción de *igualdad*.

- **Sofisticar *solver* del by**: en Subsección 4.6.6 (Alcance y limitaciones) se mencionan las limitaciones del *solver* usado por el **`by`**. Una funcionalidad que quedó afuera pero sería sencilla de agregar es que no solo se busque eliminar los $\forall$ consecutivos de *una* hipótesis, sino que el proceso sea recursivo: que exhaustivamente intente de eliminar los $\forall$ de todas las combinaciones de hipótesis posibles.

- **Mejorar a PPA como lenguaje de programación**: el lenguaje PPA no brinda soporte para tener un ecosistema. Se pueden agregar muchas funcionalidades que mejorarían la calidad de vida del desarrollador como permitir importar archivos o módulos e implementar una biblioteca estándar de teorías y teoremas.

- **Extender PPA con tipos**: el lenguaje no permite ni especificar ni chequear tipos. Se podría implementar una versión tipada, basada en lógica de primer orden *many sorted* (con varios géneros) que facilitaría la escritura de programas más sofisticados.

- **Refinar fórmulas *conjuntivas***: en la traducción de Friedman introducimos dos lemas centrales: $B \vdash_I B^{\neg\neg}$ Lema 12 (Introducción de la traducción $\neg\neg$), que vale para *F-fórmulas* y restringe las axiomatizaciones, y $\neg_R A \vdash_I \neg_R A^{\neg\neg}$ Lema 5 (Introducción de $\neg_R$), que vale para fórmulas *conjuntivas*. Como el primero se reduce al segundo, podríamos refinar las fórmulas *conjuntivas*, lo que permitiría usar más tipos de axiomas. También se podría profundizar en el vínculo con las fórmulas de Harrop mencionado en Obs. 3.

- **Extender traducción de Friedman a más de un existencial**: en su versión final, lo máximo que llega a convertir son fórmulas de la forma $\forall y_1 \dots \forall y_n . \exists x . \varphi(\dots)$. Se debería poder extender a fórmulas de la forma $\forall y_1 \dots \forall y_n . \exists x_1 \dots \exists x_m . \varphi(\dots)$.

- **Implementar versión completa de reducción de demostraciones**: en Subsección 5.5.3 (Limitaciones) vimos las limitaciones del mecanismo de reducción implementado. Es ineficiente, y no se puede llegar a una forma normal útil para todas las demostraciones, evitando la extracción de testigos en casos donde debería ser posible. Se podrían extender para incluir reducciones permutativas e implementar una máquina abstracta para reducción *call-by-need* o *call-by-name*.

- **Mejorar reporte de errores**: los errores reportados por la herramienta en general son muy rústicos, implementativos y de bajo nivel. Se podrían hacer más amigables y accionables, ayudando a resolver problemas sin saber cómo funciona internamente la herramienta.

BIBLIOGRAFíA

[AF98] Jeremy Avigad y Solomon Feferman. «Godel's Functional Interpretation». En: *Handbook of proof theory*. Ed. por Samuel R. Buss. Elsevier, 1998, págs. 337-405. URL: https://philpapers.org/rec/FEFOFD.

[Agd] Agda. *The Agda Wiki*. https://wiki.portal.chalmers.se/agda/pmwiki.php.

[Bau] Andrej Bauer. *Constructive gem: irrational to the power of irrational that is rational*. https://math.andrej.com/2009/12/28/constructive-gem-irrational-to-the-power-of-irrational-that-is-rational/.

[BC19] Małgorzata Biernacka y Witold Charatonik. «Deriving an Abstract Machine for Strong Call by Need». En: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. por Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 8:1-8:20. ISBN: 978-3-95977-107-8. DOI: 10.4230/LIPIcs.FSCD.2019.8. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2019.8.

[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.

[Coq] Coq. *The Coq Proof Assistant*. https://coq.inria.fr/.

[Cré07] Pierre Crégut. «Strongly reducing variants of the Krivine abstract machine». En: *High. Order Symb. Comput.* 20.3 (2007), págs. 209-230. DOI: 10.1007/S10990-007-9015-Z. URL: https://doi.org/10.1007/s10990-007-9015-z.

[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.

[GR97] Dov M Gabbay y J A Robinson, eds. *Handbook of logic in artificial intelligence and logic programming: Volume 5: Logic programming*. en. Handbook of Logic in Artificial Intelligence and Logic Programming. Oxford, England: Clarendon Press, sep. de 1997.

[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.

[Miq17] Alexandre Miquel. *An introduction to classical realizability*. https://ejcim2017.sciencesconf.org/data/pages/slides_realiz.pdf. 2017.

[Miz] Mizar. *Mizar Home Page*. https://mizar.uwb.edu.pl/.

[MM82] Alberto Martelli y Ugo Montanari. «An Efficient Unification Algorithm». En: *ACM Transactions on Programming Languages and Systems* 4.2 (abr. de 1982), págs. 258-282. ISSN: 1558-4593. DOI: 10.1145/357162.357169. URL: http://dx.doi.org/10.1145/357162.357169.

[Par] Rohit Parikh. «Church's theorem and the decision problem». En: *Routledge Encyclopedia of Philosophy*. Routledge. ISBN: 9780415250696. DOI: 10.4324/9780415249126-y003-1. URL: http://dx.doi.org/10.4324/9780415249126-Y003-1.

[PH24] Francis Jeffry Pelletier y Allen Hazen. «Natural Deduction Systems in Logic». En: *The Stanford Encyclopedia of Philosophy*. Ed. por Edward N. Zalta y Uri Nodelman. Spring 2024. Metaphysics Research Lab, Stanford University, 2024.

[Sel92] Peter Selinger. «Friedman's A-Translation». En: (1992). URL: https://www.mscs.dal.ca/~selinger/papers/papers/friedman.pdf.

[SU10] Morten Sørensen y Paweł Urzyczyn. «Lectures on the Curry-Howard Isomorphism». En: *Studies in Logic and the Foundations of Mathematics* 149 (oct. de 2010). DOI: 10.1016/S0049-237X(06)80005-4. URL: https://disi.unitn.it/~bernardi/RSISE11/Papers/curry-howard.pdf.

[Wen99] Markus Wenzel. «Isar — A Generic Interpretative Approach to Readable Formal Proof Documents». En: *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, 1999, págs. 167-183. ISBN: 9783540482567. DOI: 10.1007/3-540-48256-3_12. URL: http://dx.doi.org/10.1007/3-540-48256-3_12.

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

[Wie02] Freek Wiedijk. *Checker*. https://www.cs.ru.nl/~freek/mizar/by.pdf. 2002.