Universidad ORT Uruguay
Facultad de Ingeniería

Diseño de Asistente Para Derivaciones Esquemáticas del Sistema de Deducción Natural de Lógica de Primer Orden

Entregado como requisito para la obtención del título de Master en Ingeniería por Investigación

Jorge Pais – 143280

Tutor: Álvaro Tasistro

2013

Declaración de Autoría

Yo, Jorge Pais, declaro que el trabajo que se presenta en esa obra es de mi propia mano.
Puedo asegurar que:
- La obra fue producida en su totalidad mientras se realizaba la tesis para la obtención del título de Master en Ingeniería
- Cuando se consultó el trabajo publicado por otros, se lo ha atribuido con claridad;
- Cuando se citaron obras de otros, se indicó en las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, se acusa recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, se explica claramente qué fue contribuido por otros, y qué fue contribuido por mí;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Jorge Pais
22/07/2013

Abstract: Se presenta el diseño completo, y correspondiente prototipo, de un asistente para el cálculo de Deducción Natural para Lógica de Orden uno y medio, basado en consideraciones didácticas. La lógica de orden uno y medio es la de los esquemas de fórmulas y de derivaciones que usualmente se consideran en las presentaciones en cursos y libros de texto de la Lógica de Primer Orden. En particular, el asistente resultante permite derivar y re-usar correctamente meta-teoremas, incluyendo la expresión y verificación de sus condiciones laterales. En base a técnicas de sintaxis nominal se definen conceptos del meta-nivel como parte de la sintaxis del sistema, lo cual hace posible lidiar con metavariables sobre fórmulas y términos, alfa-equivalencia, substitución con evitación de captura y restricciones de aparición libre de variables en fórmulas de forma explícita en las derivaciones.

Palabras Clave: Software educativo, Enseñanza de lógica, Pruebas formales

Índice

1 Introducción 6
1.1 Deducción Natural en la Universidad 6
1.2 Uso de asistentes computarizados 7
1.3 Funcionalidades deseadas 9
1.4 Estudio de sistemas disponibles 11
1.4.1 Criterios de comparación 11
1.4.2 Relevamiento 12
1.4.3 Discusión 22
2 Lógica de Orden 1½ 23
2.1 Términos Nominales 24
2.2 Frescura 27
2.3 Deducción Natural 28
2.4 Resultados Meta-Teóricos 30
2.4.1 Consistencia 30
2.4.2 Derivabilidad de cut 30
2.4.3 Instanciación 30
2.5 Matching de Fórmulas en Sintaxis Nominal 31
3 Descripción del Sistema 35
3.1 Representación de fórmulas lógicas 35
3.2 Representación de términos nominales 35
3.3 Pruebas y Derivaciones 36
3.4 Reglas de Inferencia 37
3.4.1 Reglas Backward 38
3.4.2 Reglas Forward 40
3.5 Automatización de Demostración de Condiciones Laterales 42
3.6 Aplicación de Lemas 43
4 Ejemplos de uso 45
5 Conclusiones 49
6 Bibliografía 50
7 Anexo I 52
7.1 Algebra Nominal y Derivabilidad 52
7.1.1 La Teoría CORE 52
7.1.2 Decidibilidad en SUB 54
7.1.3 Algoritmo de Derivación Automática de Condiciones Laterales 55
8 Anexo II 57
8.1 Manual de usuario 57

1 Introducción

Un objetivo deseable en la educación de la Ingeniería de Software es que los estudiantes puedan argumentar por qué sus programas funcionarán bajo toda posible circunstancia. La tarea tiene aún más valor cuando se toma en cuenta que la disciplina en sí misma está aún muy atada a prácticas en las que la correspondencia entre los programas y sus especificaciones es una, a veces muy arriesgada, conjetura. Como consecuencia, la construcción de software se transforma en un proceso de prueba y error y nunca se llega a la confianza total. Es importante entonces que los estudiantes de Ingeniería de Software tengan la oportunidad de aprender y practicar la capacidad de realizar demostraciones y en particular demostraciones de corrección de software.

La demostración es la actividad matemática consistente en alcanzar cierto conocimiento deductivamente, i.e. partiendo de postulados, supuestos o principios evidentes y realizando sucesivas inferencias, cada una de las cuales extrae una conclusión de premisas previas. En la aplicación de esta práctica a la programación, tenemos en la base la semántica de los lenguajes de programación, que nos permite entender el código de los programas y por lo tanto saber exactamente qué es lo que cada programa computa. Esto permite afirmar deductivamente que las computaciones realizadas por un programa satisfacen ciertas propiedades. Entre estas propiedades están las relaciones de entrada-salida o patrones de comportamiento que constituyen una formulación precisa de la llamada especificación funcional de un programa o sistema.

Dentro de la Ingeniería de Software se define una disciplina llamada Métodos Formales como aquella que consiste en el uso de lenguajes y sistemas formales y herramientas relacionadas para expresar especificaciones y llevar a cabo demostraciones de corrección de los programas. Estas demostraciones, como mínimo, son verificadas automáticamente para asegurar que son correctas. Además, las herramientas automáticas pueden ofrecer facilidades para ayudar en el desarrollo de las demostraciones. Los lenguajes y sistemas de demostración empleados en estas actividades son resultados de la investigación en Lógica (Formal).

Esto agrega otra dimensión al significado de la Lógica en la educación en Ingeniería de Software. Por un lado promueve, en virtud de su naturaleza teórica, la reflexión en la actividad natural y espontánea de razonar, proveyendo de fundamentos, seguridad y herramientas intelectuales. Pero además, por su rol en los fundamentos de los métodos formales para la construcción de software, se convierte en un marco de trabajo dentro del cual son formuladas y comprendidas herramientas y métodos computarizados relevantes para la práctica profesional.

1.1 Deducción Natural en la Universidad

En nuestra universidad se dicta un curso de Lógica Formal para los programas de Ingeniería en Sistemas y Licenciatura en Sistemas de Información, basado en las premisas anteriores. El curso comienza con un estudio de la inducción y la recursión, abarcando la definición inductiva de conjuntos y sus correspondientes principios de prueba por inducción y de definición de funciones por recursión. Esto permite establecer las bases para el tratamiento de lenguajes entre los cuales se encuentran los lenguajes y sistemas de la Lógica.

Luego están los contenidos clásicos de un curso de Lógica Matemática, i.e. Lógica Proposicional y de Primer Orden, cada una con su correspondiente sintaxis, semántica y sistema de demostraciones. Esto último introduce al curso la idea de demostración formal, realizada con un sistema de reglas. Se hace hincapié en la similitud entre un sistema formal de demostraciones y un lenguaje de programación, especialmente en cuanto a las restricciones que se imponen en las formas concretas de expresiones permitidas.

Principalmente por priorizar la usabilidad frente al análisis matemático, se emplea el Cálculo de Deducción Natural [1]. Este sistema fue diseñado con el objetivo de imitar la práctica común en demostraciones matemáticas informales (en lenguaje natural) permitiendo reglas que hacen uso de supuestos adicionales temporales. Por ejemplo, para probar una fórmula que establece una implicación $A \rightarrow B$, es posible usar una regla que nos permite asumir A para poder probar B a partir de esa hipótesis. Específicamente, la regla establece que si se tiene una prueba de B que depende de suponer A, entonces se puede inferir que $A \rightarrow B$, obteniendo entonces una prueba que ya no depende del supuesto mencionado. Se dice que el supuesto se descarga o se *cancela*.

Las reglas de inferencia están organizadas (con alguna excepción) alrededor de las constantes lógicas (conectivas y cuantificadores) y son de una de dos clases en cada caso. Una regla puede ser:

- de Introducción, estableciendo un modo en que una fórmula que contiene a la constante lógica en cuestión como operador principal puede ser derivada en forma directa.
- de Eliminación, indicando cómo una tal fórmula puede ser usada para derivar nuevas consecuencias.

Las reglas tienen en general varias premisas pero siempre una única conclusión, por lo que las pruebas formales se representan naturalmente como árboles. Estos árboles se diagraman con la raíz en la parte baja representando a la conclusión del teorema y con los supuestos iniciales en las hojas en la parte de arriba. El uso de reglas dentro del árbol sigue un patrón característico: al leer el árbol desde arriba, primero se aplican reglas de eliminación para obtener información a partir de las premisas, en una fase conocida como *análisis*. En algún punto durante la derivación, uno comienza a *sintetizar*, formando nuevas conclusiones de la información obtenida al aplicar reglas de introducción. Para una presentación completa del sistema, referirse a [2].

1.2 Uso de asistentes computarizados

Es natural, luego de lo expuesto anteriormente, que los estudiantes de nuestro curso tengan a su disposición un asistente computarizado para realizar demostraciones formales, de la misma forma que usan una implementación de un lenguaje de programación para aprender a programar.

La construcción de demostraciones en papel tiende a ser tediosa, no hay linealidad y a los estudiantes les cuesta adaptarse. Deben borrar varios pasos para volver atrás en lo que estaban haciendo e intentar por otro camino; escriben en distintas partes de la hoja para hacer anotaciones o sub-pruebas. Factores estéticos hacen difícil visualizar la prueba completa. Además, los estudiantes se ven menos motivados a trabajar en papel que en una computadora, especialmente en etapas tempranas de su educación profesional, que es cuando toman el curso de lógica. Esperan poder trabajar frecuentemente en computadoras, lo cual ciertamente

es uno de los principales motivos que los impulsa a elegir una carrera de este tipo.
Finalmente, pero no menos importante, el hecho de que un asistente computarizado chequee que las pruebas construidas son correctas, provee seguridad y motivación a los estudiantes para que trabajen por sus propios medios, minimizando la dependencia de los docentes.
Este último punto es de extrema importancia ya que se requiere de una extensa práctica para que el estudiante progrese en el proceso de aprendizaje de construcción de pruebas formales, particularmente con el sistema de deducción natural. Usualmente comienzan en una etapa en la que aplican reglas de inferencia cual si fuesen piezas de un puzle sin tener idea real del estado y dirección de la prueba. Esperan que el mero encastre de las reglas de inferencia derive finalmente en una prueba completa. Ven a las pruebas como unas simples transformaciones formales sin aprehender realmente su significado lógico. Es la práctica, acompañada inicialmente por una guía adecuada, la que finalmente provee de sentido y disciplina a las manipulaciones formales.

En virtud de esto, desde hace un tiempo el grupo de Computación Teórica y la Cátedra de Teoría de la Computación se han embarcado en el desarrollo de una herramienta para asistir en la práctica de la demostración formal utilizando el sistema de Deducción Natural. La primera versión ANDY 0 [3], [4] (por sus siglas en inglés *Assistant for Natural Deduction*) se desarrolló en el año 2011 y se logró evaluar su utilidad en la práctica, adoptándolo como herramienta en el curso de Lógica. Esa versión estaba acotada para la Lógica Proposicional, pero implementaba características especiales que no se encontraban en otras herramientas similares relevadas y se adaptaba a requerimientos concretos importantes para nuestro curso.

Ahora nos embarcamos en la tarea de extender las funcionalidades deseadas para la Lógica de Primer Orden. El cambio no es menor; se debió rescribir el motor de inferencias en el que estaba basado el sistema original y adaptarlo para el manejo de nuevos elementos de sintaxis. Hay una característica fundamental de la versión anterior que plantea un gran desafío a la hora de implementarlo para Lógica de Primer Orden: el manejo de metavariables, definidas como variables que representan fórmulas del lenguaje (a diferencia de las variables simples que representan elementos del dominio de interpretación). Este elemento extra en la sintaxis tiene un gran impacto en la complejidad del sistema. Se aplicarán conceptos de sintaxis nominal para lidiar con problemas de ligadura y captura de variables e instanciación de lemas.

Se propondrá una implementación inicial del sistema ANDY 1½ embebida en el lenguaje Haskell. Se aspira a que en una próxima versión se pueda dotar al sistema de una interfaz gráfica similar a la de la versión anterior.

1.3 Funcionalidades deseadas

1.3.1.1 Representación arborescente de las pruebas

Las pruebas deberán representarse en forma de árboles, coincidiendo con la forma en que se presentan en el curso de Lógica. Este punto resulta de suma importancia ya que existen asistentes de demostración similares pero que ofrecen representaciones basadas en diagramas de Fitch (diagramas de caja) que básicamente consiste en la construcción de rectángulos embebidos que representan sub-pruebas. Más información sobre la representación de Fitch puede encontrarse en [5]

Se prefiere esta representación porque surge de forma natural de la composición de reglas de inferencia, i.e. con las premisas encima de una barra horizontal que representa al acto de inferir la conclusión que aparece por debajo.
Esta representación es un factor diferencial comparado a otros programas similares.

1.3.1.2 Uso de Metavariables e Instanciación de Lemas

Las formulas lógicas manejadas por el sistema podrán contener metavariables que representen tanto fórmulas como términos.
El uso de metavariables permitirá usar pruebas completas como justificación de inferencias realizadas en otras pruebas.

Un lema se define como una prueba completa. En general, interesan esquemas de prueba que representan infinitas familias de teoremas concretos y pueden ser expresados al emplear metavariables, por ejemplo:

$$\forall x. (\alpha \rightarrow \beta), \forall x. \alpha \vdash \forall x. \beta$$

Esto es válido para cualquier fórmula α y β, sea lo que sean, por lo que estamos frente a una familia infinita de teorema concretos. Representa por ejemplo a:

$$\forall x. (P(x, y) \rightarrow \bot), \forall x. P(x, y) \vdash \forall x. \bot$$
$$\forall x. (Q(x) \rightarrow P(f(y), a)), \forall x. Q(x) \vdash \forall x. P(f(y), a)$$

Los lemas se usarán durante la construcción de otras pruebas como si fuesen nuevas reglas de inferencia (de hecho son reglas derivadas). Se requiere que el sistema soporte la funcionalidad generando una instancia apropiada del lema a ser usado, basado en el estado actual de la prueba en el punto donde se aplique. Se espera que el estudiante evalúe a priori si la instanciación del lema es posible en el contexto actual, pero sin la necesidad de dar una correspondencia explícita entre metavariables y fórmulas.

1.3.1.3 Asistente, no un demostrador automático

La herramienta debe asistir al estudiante en la construcción de pruebas, no completarlas por él. Este concepto se manifiesta de las siguientes formas:

- Mientras la prueba esté incompleta, el sistema debe indicar qué sub-pruebas deben ser completadas.
- Debe haber un mecanismo *backwards* de construcción de pruebas, esto es, que permita seleccionar una conclusión que falta probar y una regla de inferencia y deduzca un conjunto apropiado de premisas que lleven a la conclusión deseada a través de la regla. Estas premisas se convierten en la nueva conclusión a probar.
- El sistema valida localmente la aplicación de cada regla de inferencia.
- El sistema no valida que el camino o la estrategia general elegida para construir la prueba sea válido.
- Debe haber un mecanismo *forward* para la construcción de pruebas, i.e. partiendo desde los supuestos hacia las conclusiones.
- Debe haber una manera de insertar pruebas completas en el lugar de cualquier sub-prueba pendiente.

La posibilidad de trabajar de forma *backward* y *forward* refleja el hecho de que en el sistema de Deducción Natural es natural trabajar de la siguiente manera:

- Al principio se intenta ir en sentido *backward* desde la conclusión hacia nuevas conclusiones que deban ser probadas, generalmente seleccionando la regla de introducción correspondiente. Se repite el procedimiento hasta que no puedan aplicarse reglas de este tipo o éstas lleven a conclusiones que no sean apropiadas.
- En ese punto, se empiezan a combinar las hipótesis en forma *forward,* desde las premisas conocidas hacia nuevas conclusiones inferidas, intentando obtener los resultados pendientes del paso anterior. Esto se logra aplicando reglas de eliminación de las conectivas.

Por lo tanto, el árbol es por lo general construido primero desde la raíz hacia arriba hasta un punto correspondiente a la finalización de la etapa de síntesis. Luego, uno procede desde las hojas hacia abajo, construyendo la parte analítica de la prueba. Esta forma de proceder parece opuesta a la que sugiere la estructura analítica-sintética de las pruebas, pero hay que tener en cuenta que la forma en la que la prueba se presenta es en general diferente a la forma en la que se concibe. Además, la estrategia anterior no es la única apropiada, sino que en general es una bastante útil, especialmente cuando se tiene poca experiencia en derivaciones.
Esta estrategia deberá poderse ejecutar naturalmente en el asistente.

1.4 Estudio de sistemas disponibles

El problema de la construcción de sistemas que faciliten el aprendizaje del cálculo de deducción natural en cursos introductorios de lógica ya ha sido estudiado en el ambiente académico. Hay varios programas que implementan soluciones diferentes, cada uno con un conjunto particular de funciones. Se describirá y caracterizará a algunos de ellos, que en nuestra opinión, resumen el estado del arte:

- PANDORA [6]
- ADN [7]
- PANDA [8]
- ProofBuilder [9]
- DC Proof [10]
- JAPE [11]
- ANDY [3,4]

1.4.1 Criterios de comparación

Además de una breve descripción del sistema, se define una serie de puntos de comparación basados en los que aparecen en [12], donde se comparan sistemas de deducción natural desde el punto de vista de la interfaz gráfica. A estos se agregan puntos sobre el manejo de meta teoremas que son de interés para la comparación con nuestro sistema.

1. **Características de implementación:** Detalles sobre el lenguaje de programación usado y portabilidad del sistema.
2. **Fecha de creación / última versión:** Indica la fecha de creación del programa, la versión actual y el estado del proyecto (discontinuado, en uso, etc.)
3. **Versión de deducción natural:** Sistema lógico (clásico, minimal, intuicionista, predicados, etc.), reglas de inferencia disponibles y su comportamiento, restricciones del sistema.
4. **Visualización de derivaciones:** La forma en que las derivaciones se muestran al usuario (como árboles, como secuencias). Además, visualización de premisas, goals, aplicación de las reglas, fórmulas y numeración de pasos.
5. **Uso de metavariables y esquemas de derivación:** funcionalidad del sistema que permite generar esquemas de derivación mediante el uso de metavariables.
6. **Gestión de derivación:** Posibilidad de guardar derivaciones y usarlas luego como lemas.
7. **Tipos de razonamiento:** Backward, forward, ambos.
8. **Ayuda:** Dependiendo de la implementación, el sistema puede ofrecer ciertos mecanismos de ayuda para finalizar la prueba. Esto resulta sumamente importante desde un punto de vista pedagógico. Mucha ayuda puede derivar en que el estudiante aplique aleatoriamente varias reglas hasta que la prueba se complete. Establecemos cuatro mecanismos de ayuda:
    a. Ayuda global: Independiente de la prueba en curso, basado en tutoriales. Describe las reglas disponibles y puede contener ejemplos de cómo aplicarlas.
    b. Ayuda táctica: Le da al estudiante una posible táctica para resolver el problema dependiendo del estado actual de la prueba. Por lo general consiste de un único paso deductivo.

c. <u>Ayuda estratégica:</u> Extensión de la ayuda táctica. Da un plan de prueba que consiste de varios pasos deductivos.

d. <u>Debugging:</u> Indica los errores que surgen en la prueba, por ejemplo cuando una regla se aplica incorrectamente.

9. **Validación de la derivación:** Mecanismo por el cual el sistema valida que la derivación es correcta. Puede ser paso a paso, evitando pasos que llevarían a una prueba errónea o imposible de completar, o una validación final cuando la prueba está completada.

1.4.2 Relevamiento

1.4.2.1 Pandora

PANDORA (por las siglas en inglés para *Proof Assistant for Natural Deduction using Organised Rectangular Areas*) fue desarrollado en el *Department of Computing* del *Imperial College London* con el objetivo de apoyar el aprendizaje de Deducción Natural para Lógica de Primer Orden. Su primera versión es del año 1996 y fue escrito en TCL/TK y Prolog. Luego en 1999 fue reescrito en Java y se mantiene así hasta el día de hoy.
Se basa en la construcción de derivaciones en deducción natural usando la notación de Fitch o de cajas. Permite que los usuarios razonen de manera forward y backward, con reglas de introducción y eliminación para las conectivas comunes y los cuantificadores existencial y universal. Además implementa algunas reglas avanzadas como la eliminación de la doble negación, prueba por contradicción, ley del tercero excluido y otras.

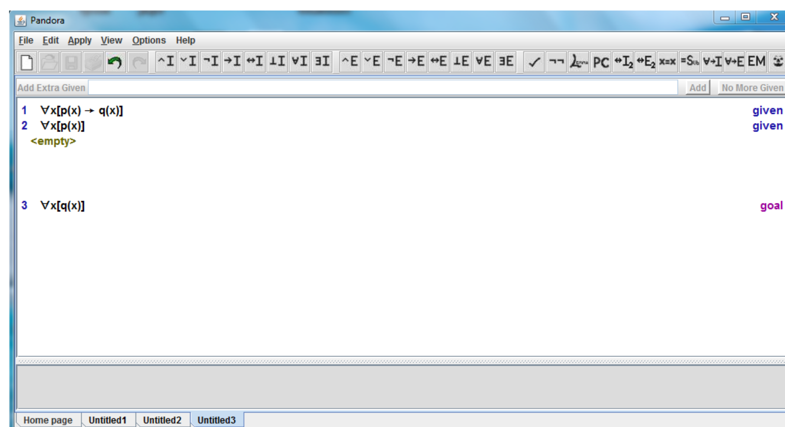

Figura 1 - Inicio de una derivación en PANDORA

Durante la construcción de la derivación se muestra un panel de ayuda que muestra los errores y pasos para aplicar las reglas correctamente. Además tiene un completo manual de usuario y un tutorial con varios ejercicios de ejemplo con distintos niveles de dificultad.
Más información sobre el sistema en [13]

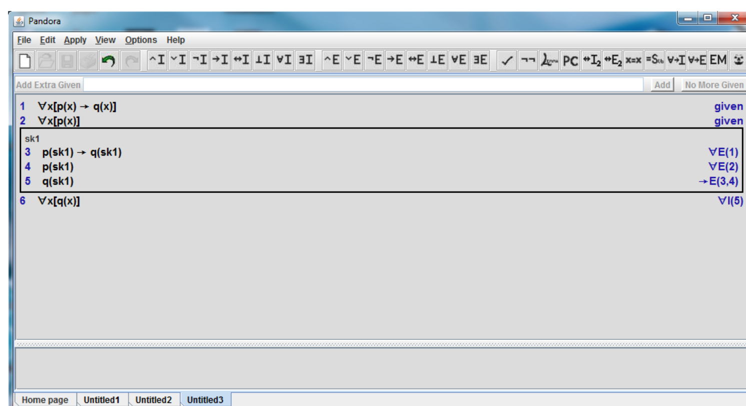

Figura 2 - Derivación completada en PANDORA

Criterio	Descripción
Características de implementación	Aplicación desarrollada en Java. Puede usarse como un applet. Altamente portable.
Fecha de creación / última versión	La primera versión es de 1996. El último registro de actividad es del 2004 cuando se agregó el tutorial.
Versión de Deducción Natural	Deducción Natural para lógica de primer orden con las reglas clásicas para introducción/eliminación de conectivas y cuantificadores. Agrega reglas avanzadas para la igualdad, ley del tercero excluido, prueba por el absurdo.
Visualización de derivaciones	Las derivaciones se muestran secuencialmente como diagramas de Fitch. Los lugares donde se necesita completar la derivación se simbolizan con una etiqueta <empty>. Las obligaciones de prueba se indican con la etiqueta <goal>. Utiliza el concepto de *signature* de la derivación, como el conjunto de predicados, variables y constantes que aparecen en ella. Hay una *signature* global que afecta a toda la prueba y una *signature* local para cada rectángulo en la prueba (por lo general tiene más elementos que la global). El *signature* global se crea automáticamente por el programa y no puede editarse; sin embargo, las *signature* locales pueden modificarse en cualquier momento, por ejemplo para crear una variable fresca que pueda usarse para aplicar la regla de eliminación del cuantificador universal.
Uso de metavariables y esquemas de derivación	No permite usar metavariables, y por lo tanto tampoco generar esquemas de derivación. Se usa una regla de lema, pero no en el sentido que quisiéramos. En este sistema, la regla de lema permite extraer una subprueba como una prueba separada y construir su derivación en un nuevo panel. Su utilidad parece más de organización visual y estética de la prueba.
Gestión de derivaciones	Permite guardar derivaciones completes e incompletas como archivos .pan. No permite reutilizar derivaciones previas como parte de derivaciones en curso. Puede simularse el uso de lemas mediante una regla llamada *Trust Me* que permite marcar un goal como probado sin necesidad de dar la derivación explícita. Aun así, no es el mecanismo de lemas que se desea.
Tipos de razonamiento	Permite razonamiento forward y backward, determinado por la forma en la que se aplica la regla. Para aplicar una regla en su variante forward, por lo general, se selecciona una línea con la etiqueta <empty> y luego una regla en el panel de opciones, mientras que para aplicarla en su variante backward se selecciona una línea <goal> y después una de las reglas.

Ayuda	El sistema contiene una extensa guía y un tutorial, accesible desde el menú de ayuda de la aplicación. Tiene explicaciones detalladas de cómo usar cada regla en su variante forward y backward, además de convenciones, algunas pistas genéricas de cómo construir las derivaciones y detalles sobre la sintaxis. No hay ayuda estratégica o táctica. Muestra errores cuando las reglas se intentan aplicar incorrectamente.
Validación de la derivación	Se valida la prueba luego de que está finalizada. La unica restricción a la aplicación de las reglas está dada por el sistema de ayuda.

Tabla 1- Descripción de PANDORA

1.4.2.2 ADN

El sistema ADN [14](Asistente para la Deducción Natural) es una herramienta didáctica creada por el departamento de Ciencia de la Computación e Inteligencia Artificial de la Universidad de Alicante. El objetivo de la herramienta es ayudar a los estudiantes a escribir fórmulas bien formadas y hacer deducciones correctamente. No es un sistema que construya derivaciones automáticamente, sino que asiste y guía al estudiante durante el proceso.
Tiene un asistente animado al que se le pueden pedir consejos durante la construcción y muestra las posibles reglas a aplicar según el estado de la derivación.
El mecanismo de aplicación de las reglas no es muy intuitivo y tampoco hay suficiente información en la ayuda que indique como hacerlo correctamente.

Criterio	Descripción
Características de implementación	Desarrollado en Java, puede usarse como applet con los navegadores actuales. Muy portable.
Fecha de creación / última versión	Creado en el 1999. No hay registros de actividad en el sistema ni en publicaciones después del 2004. Última versión 2.1
Versión de Deducción Natural	Deducción Natural para lógica de Primer Orden sin metavaribles. Implementa reglas de inferencia para los cuantificadores y negación, conjunción, disyunción e implicación.
Visualización de derivaciones	La derivación se muestra mediante diagramas de Fitch, con cada paso numerado a la izquierda y con una descripción de la regla aplicada a la derecha.
Uso de metavariables y esquemas de derivación	No se hace uso de metavariables.
Gestión de derivaciones	No hay forma de guardar las pruebas completadas.
Tipos de razonamiento	Razonamiento forward.
Ayuda	El asistente provee un mecanismo de ayuda táctica, dando consejos sobre las reglas que pueden aplicarse en un momento dado, basándose en información sobre el objetivo de la derivación, sobre las fórmulas introducidas y sobre aspectos generales de la construcción de derivaciones.
Validación de la derivación	Valida la correctitud de la derivación cuando termina la construcción. No permite dar pasos incorrectos.

Tabla 2 - Descripción de ADN

1.4.2.3 PANDA

PANDA (por las siglas en inglés para *Proof Assistant in Natural Deduction for All*) Es un asistente interactivo para la construcción de pruebas en Deducción Natural para estudiantes de grado [15]. Tiene una muy buena interfaz gráfica que permite combinar, borrar y modificar pruebas haciendo drag&drop de fórmulas y partes de derivaciones. Fue desarrollado en el IRIT (*Institut de Recherche en Informatique de Toulouse*) de la *Université de Toulouse*.

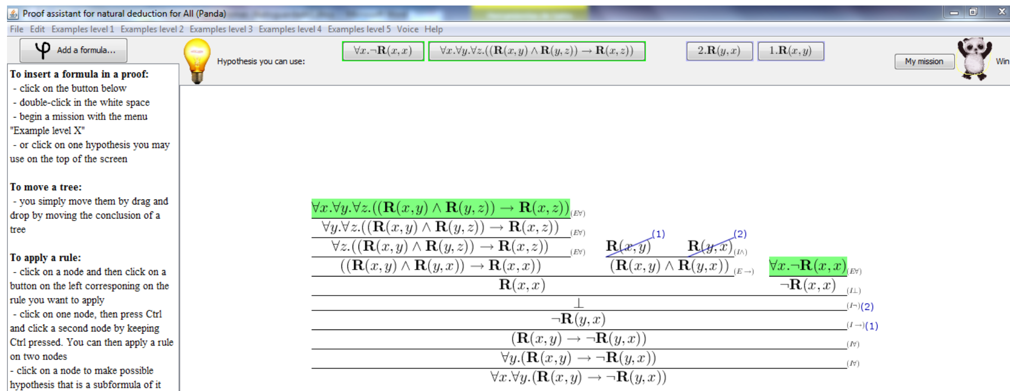

Figura 3 - Visualización de una derivación completa en PANDA

El sistema es muy intuitivo. La aplicación de reglas y la manipulación de fórmulas y partes de derivaciones es muy sencilla. No es muy claro el mecanismo para generar la prueba a partir de un juicio de la forma $\Gamma \vdash \alpha$

En algunos casos resuelve automáticamente situaciones que podrían ser deseables los resolviera el estudiante, por ejemplo la cancelación de hipótesis temporales.

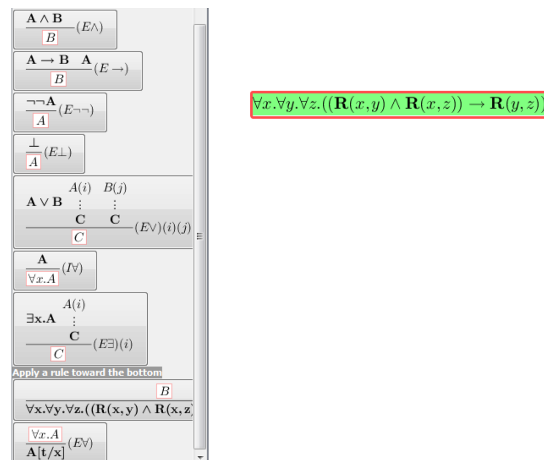

Figura 4- Al seleccionar una fórmula, se muestran las reglas aplicables

Criterio	Descripción
Características de implementación	Escrito en Java. Funciona desde cualquier navegador.
Fecha de creación / última versión	Creado en 2011. Hay proyectos que buscan extender funcionalidades.
Versión de Deducción	Lógica de primer orden con las conectivas clásicas y cuantificadores.

Natural	
Visualización de derivaciones	Las derivaciones se muestran como árboles al estilo Gentzen. Permite tener varios árboles de derivación en el panel de prueba que pueden arrastrarse y componerse unos con otros para lograr otras derivaciones.
Uso de metavariables y esquemas de derivación	No hace uso de metavariables.
Gestión de derivaciones	Permite guardar pruebas en formato LATEX para utilizar en documentos. También pueden guardarse en formato XML para volver a cargarse en el sistema.
Tipos de razonamiento	Permite una mezcla arbitraria de razonamiento backward y forward.
Ayuda	No hay un manual completo sobre cómo usar el programa, por lo que puede ser difícil de usar para usuarios primerizos. Sin embargo el proceso de construcción de prueba es muy intuitivo para quienes conocen la representación basada en árboles. Provee ayuda al momento de aplicar reglas de inferencia, solo muestra aquellas reglas que pueden aplicarse correctamente.
Validación de la derivación	En cada paso solo permite reglas válidas.

Tabla 3 - Descripción de PANDA

1.4.2.4 DC Proof

Desarrollado por Dan Christensen, no es un programa exclusivo para derivaciones en Deducción Natural, sino que se enfoca en pruebas matemáticas en general, implementando conceptos de teoría de conjuntos y pruebas por inducción.
Es algo complejo de utilizar para usuarios primerizos por la cantidad de conceptos matemáticos que tiene.

Criterio	Descripción
Características de implementación	No hay documentación indicando el lenguaje en el que está implementado. En la página de descarga aclara que es una aplicación para ambientes Windows y se necesitan emuladores para ejecutar en otros sistemas operativos.
Fecha de creación / última versión	No hay información sobre la fecha de creación, pero el sitio web muestra que la aplicación es constantemente actualizada.
Versión de Deducción Natural	No sigue la terminología clásica para las reglas de deducción natural, probablemente porque no es un sistema dedicado a esto sino a pruebas matemáticas en general. Pueden construirse derivaciones en lógica de primer orden.
Visualización de derivaciones	Los pasos se numeran secuencialmente, indicando la regla aplicada en cada paso y las líneas que involucra. Pueden comprimirse partes de la derivación para reducir el espacio ocupado.
Uso de metavariables y esquemas de derivación	No permite el uso de metavariables.
Gestión de derivaciones	Permite guardar derivaciones completas e incompletas, imprimirlas y exportarlas a archivos HTML.

Tipos de razonamiento	El razonamiento es fuertemente forward. Para comenzar una prueba, deben indicarse las premisas y la conclusión debe obtenerse mediante la aplicación sucesiva de las reglas y tácticas.
Ayuda	Tiene un manual con la explicación de cada táctica, un tutorial con ejemplos de uso y un manual de referencia sobre el uso del programa en general. No hay ayuda durante la construcción de la prueba.
Validación de la derivación	Ya que las derivaciones se construyen a partir de as premisas sin conocer a priori la concusión, la validación se hace paso a paso. El sistema asegura que la prueb es correcta, ya que todos los pasos son controlados.

Tabla 4 - Descripción de DC Proof

1.4.2.5 ProofBuilder

Programa desarrollado por Hugh McGuire de la Grand Valley State University [16]. Es un software pedagógico para la construcción de pruebas. Cubre la lógica proposicional y de predicados, algo de lógica temporal, algunas funciones matemáticas y combinatorias.

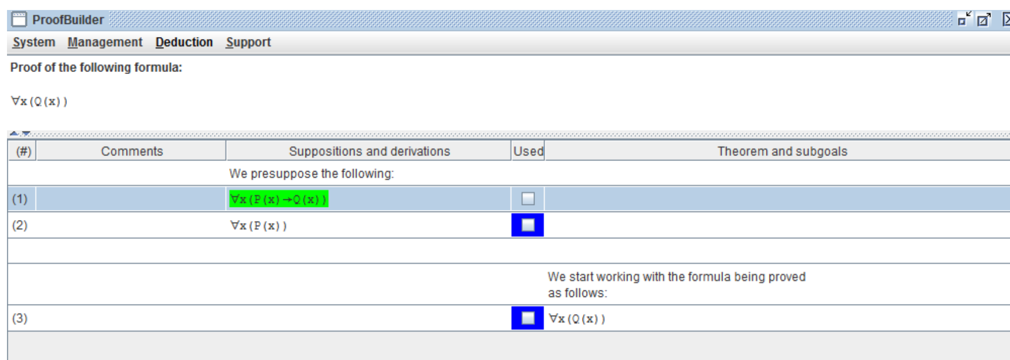

Figura 5 - Inicio de una derivación en ProofBuilder

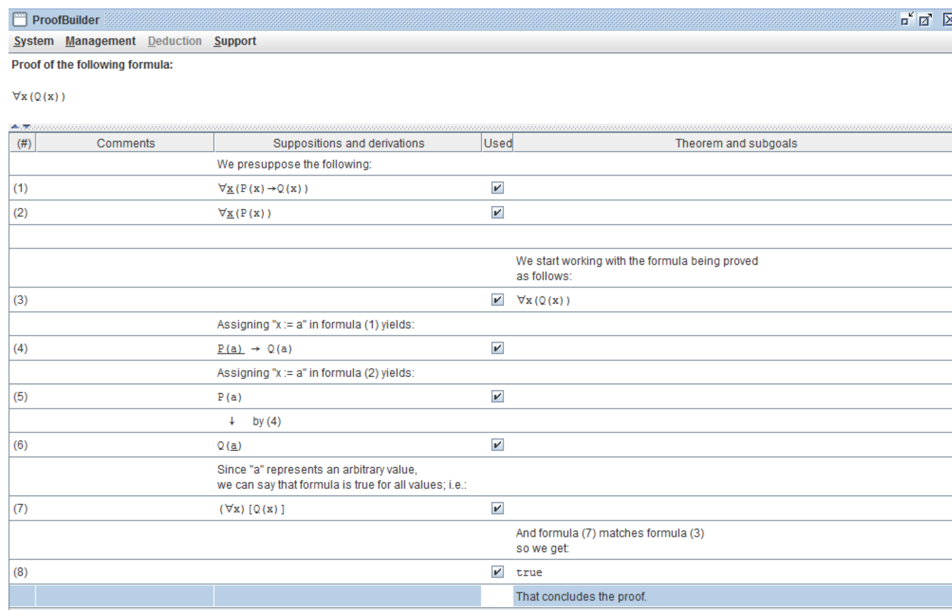

Figura 6 - Derivación completada en ProofBuilder

Criterio	Descripción
Características de implementación	Implementado en Java. Puede descargarse desde el sitio web.

Fecha de creación / última versión	Creado en el 2006. Actualizado periódicamente.
Versión de Deducción Natural	Permite construir derivaciones en lógica proposicional y de primer orden. Implementa reglas para todas las conectivas, pero no respeta la nomenclatura de introducción y eliminación conocida. Permite reescribir formulas mediante la aplicación de leyes lógicas.
Visualización de derivaciones	La visualización es secuencial en dos columnas. A la izquierda están los supuestos y las derivaciones obtenidas, mientras que a la derecha están las obligaciones de prueba. Además, en otra columna se indican las fórmulas usadas (i.e. a las que se les aplicó alguna táctica). Un detalle interesante es que a medida que se aplican reglas, el sistema agrega comentarios en lenguaje natural sobre el estado de la prueba. Al final se obtiene una descripción textual del proceso de derivación. Cuando se selecciona una formula, el sistema identifica con color a la conectiva principal y las reglas que pueden aplicarse.
Uso de metavariables y esquemas de derivación	No permite usar metavariabes.
Gestión de derivaciones	Las pruebas incompletas pueden guardarse para continuarlas después. Pueden además exportarse en format HTML.
Tipos de razonamiento	Pemrite razonamiento backward y forward.
Ayuda	No hay ayuda incluida en el programa, solo ayuda online en el sitio web. No hay ayuda táctica o estratégica. El sistema bloquea aquellas reglas que no son aplicables para la fórmula seleccionada, de forma que no puedan darse casos de error.
Validación de la derivación	La derivación se valida al final, aunque por el mecanismo de bloqueo de reglas mencionado, no pueden dares pasos incorrectos.

Tabla 5 - Descripción de ProofBuilder

1.4.2.6 JAPE

JAPE es un asistente de demostración desarrollado en la Universidad de Oxford [17, 18]. Permite trabajar con distintos sistemas lógicos, pudiendo llegar a definir nuevos sistemas y codificar sus reglas de inferencia.

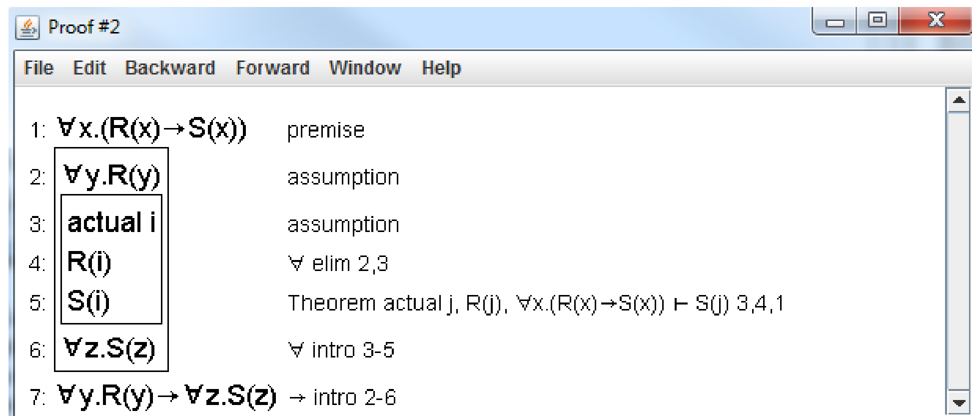

Figura 7 - Prueba completa en JAPE usando un lema demostrado previamente

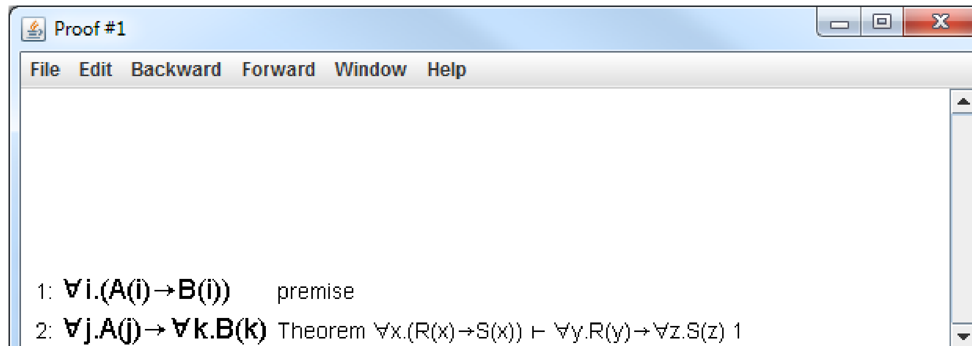

Figura 8 - Derivación en JAPE usando una instancia de un lema demostrado previamente.

Criterio	Descripción
Características de implementación	Aplicación escrita en Java. Se puede descargar desde el sitio web.
Fecha de creación / última versión	Actualizado continuamente.
Versión de Deducción Natural	Permite construir derivaciones en varias teorías (conjuntos, lógica de predicados, lógica proposicional, lógica de hoare, etc.). Lo más importante es que es possible definer nuevas teorías, reglas y propiedades, cargarlas en JAPE y construir derivaciones. Para el caso de Deducción Natural, se implementa para lógica proposicional y de primer orden. Esto incluye todas las reglas conocidas de introducción y eliminación. Clasifica las reglas en forward y backward, o sea no es sensible al contexto sino que en cada caso debe indicarse además de la regla, cuál de sus dos variantes se quiere aplicar.
Visualización de derivaciones	Las derivaciones se muestran como diagramas de Fitch. Cada paso está numerado e indica la regla que se aplicó. Mientras se construye la derivación, las partes incompletas se muestran con puntos suspensivos. La ausencia de estos, indica que la derivación está completa.
Uso de metavariables y esquemas de derivación	No permite el uso de metavariables. Sin embargo permite tomar derivaciones completas (lemas) y usarlas en una derivación en curso, instanciando los símbolos de predicado y variables que aparecen en el lema.

Gestión de derivaciones	Permite guarder pruebas completes e incompletas. Tiene un conjunto predefinido de ejemplos, que luego de ser probados pueden ser aplicados en nuevas derivaciones.
Tipos de razonamiento	Permite razonamiento forward y backward con alguna preferencia hacia el backward. Por ejemplo en el caso de la introducción del cuantificador universal, la refla solo puede aplicarse de forma backward.
Ayuda	No hay ayuda táctica o estratégica, la derivación se construye paso a paso y la ayuda solo aparece cuando una regla se palica incorrectamente. Hay un completo manual en linea sobre el uso de JAPE para deducción natural.
Validación de la derivación	El sistema no permite pasos equivocados. La validación se hace paso a paso.

Tabla 6 - Descripción de JAPE

1.4.2.7 ANDY

ANDY es la versión inicial del asistente para deducción natural desarrollado por el grupo de Computación Teórica de Universidad ORT. Se basa en la premisa de ser un asistente y no un demostrador automático, por lo que intenta automatizar lo menos posible los pasos de deducción natural a la vez que ayuda al estudiante a enfocarse en los conceptos claves de la construcción de derivaciones.
Si bien, a diferencia de los sistemas anteriores, este implementa solo la lógica proposicional, sus funcionalidades se consideran importantes para la futura versión.

Criterio	Descripción
Características de implementación	Aplicación de escritorio basado en el Framework Microsoft .NET 4.0. El núcleo del sistema está desarrollado en F# y la interfaz gráfica en C#.
Fecha de creación / última versión	Creado en 2010. En actualización.
Versión de Deducción Natural	Lógica proposicional con reglas para la conjunción, disyunción, implicación, negación, doble implicación, absurdo, más una regla para la reducción al absurdo.
Visualización de derivaciones	Representadas como árboles, donde la raíz es la conclusión y las hojas son las hipótesis del teorema. Cada nodo representa la aplicación de una regla de inferencia. Las hipótesis disponibles para un nodo dado se muestran en un panel lateral, discriminando con un código de colores si son hipótesis del teorema, supuestos o hipótesis temporales derivadas por la aplicación de alguna regla.
Uso de metavariables y esquemas de derivación	Permite usar metavariables que representan a cualquier fórmula de la lógica proposicional. Luego, las derivaciones que están compuestas por metavariables pueden reusarse en otras derivaciones mediante un mecanismo de instanciación que la transforma en una derivación concreta.
Gestión de derivaciones	Las pruebas, completas o incompletas, pueden ser exportadas como archivos XML. Las pruebas incompletas se pueden importar para continuar con la derivación, mientras que las pruebas completas pueden usarse como lemas. El sistema puede escanear un directorio donde hayan pruebas XML guardadas y obtener el juicio $\Gamma \vdash \alpha$ antes de importarlo a la derivación.

Tipos de razonamiento	Permite razonamiento backward y forward. Las reglas backward se aplican en las hojas de la prueba principal, mientras que las reglas forward se aplican en pruebas secundarias que pueden generarse en el panel de pruebas. La prueba principal y las secundarias pueden unirse en cualquier momento mediante un proceso de instanciación de fórmulas, transparente al estudiante.
Ayuda	Incluye una guía de ayuda que explica todas las funcionalidades y como aplicar cada regla. Además implementa ayuda tipo debugging, al señalar los errores cuando una regla está mal aplicada. No hay ayuda táctica ni estratégica.
Validación de la derivación	Validación paso a paso. El sistema impide que se apliquen reglas incorrectas en una fórmula.

Tabla 7 - Descripción de ANDY

1.4.3 Discusión

En la mayoría de los sistemas listados anteriormente hay poca información sobre cómo fue implementado internamente el mecanismo de prueba. La excepción es JAPE, que provee abundante información sobre estructuras internas, desarrollo de interfaces y experiencias y experimentos de uso.

Hay varias formas de nombrar a las reglas clásicas de Deducción Natural (introducción, eliminación), algunas de ellas pueden resultar confusas, en particular para principiantes. Nuestro sistema usará la nomenclatura clásica, porque es la más extendida y porque es la usada en nuestro curso de Lógica.

La combinación de razonamiento forward y backward se presenta en varios sistemas. Esta característica es la que más asemeja a la construcción de pruebas con lápiz y papel, por lo que la consideraremos de gran importancia.

JAPE parece ser el sistema con la interfaz gráfica más simple y el más fácil de entender. De todas formas, para nosotros es imperativo que el sistema muestre las pruebas como árboles, por las razones explicadas anteriormente. Esta característica para lógica de primer orden solo la presenta PANDA, aprovechando también facilidades de *Drag&Drop* para manipular derivaciones.

Una funcionalidad interesante de ProofBuilder es la de generar una descripción en lenguaje natural de la derivación. Esto, combinado con la representación arborescente puede ser una herramienta muy ponderosa para ayudar a entender cómo se va construyendo la prueba.

La versión anterior de nuestro sistema, ANDY, parece ser el que provee el mecanismo más general de manejo de lemas. Si bien JAPE permite utilizar derivaciones completas en nuevas pruebas, el mecanismo de instanciación solo permite reemplazo de símbolos, y no hay un manejo real de metavariables. Esto en nuestra opinión es una característica muy importante, ya que incorpora el principio de desarrollo modular que es relevante tanto en la práctica como en un punto de vista teórico y metodológico.

Finalmente, ninguno de los sistemas anteriores es capaz de manipular derivaciones con metavariables en la lógica de primer orden. Esta será la característica diferenciante de nuestro sistema frente a los existentes.

2 Lógica de Orden 1½

Consideremos las reglas de Eliminación del cuantificador universal $(E\forall)$ y la de Introducción del cuantificador universal $(I\forall)$ en el cálculo de Deducción Natural para la lógica de primer orden:

$$\begin{array}{c}\Gamma\\ \vdots\\ (I\forall)\,\dfrac{\alpha}{\forall x.\,\alpha}\ \big(x \notin fv(\Gamma)\big)\end{array} \qquad\qquad \begin{array}{c}\Gamma\\ \vdots\\ (E\forall)\,\dfrac{\forall x.\,\alpha}{\alpha[x \mapsto t]}\end{array}$$

En ellas α y β son metavariables y representan a cualquier fórmula de la lógica de primer orden $(\alpha, \beta \in Form_1)$; Γ es un conjunto de hipótesis (fórmulas); $fv(\Gamma)$ es el conjunto de variables libres de (las fórmulas que aparecen en) Γ y $\alpha[x \mapsto t]$ es la formula α pero donde todas las ocurrencias libres de la variable x se reemplazaron por el término t.

Considérese ahora juicios como el siguiente:

$$\alpha, \forall x.\,(\alpha \longrightarrow \beta) \vdash \forall x.\,\beta \quad con\ x \notin fv(\alpha)$$

Estrictamente, este juicio no se puede derivar formalmente en el cálculo de deducción natural bajo la sintaxis de primer orden, porque las metavariables, así como la condición lateral, no pueden representarse explícitamente en su sintaxis.

Podemos sin embargo, construir esquemas que representan a una familia de derivaciones válidas generadas por la instanciación de las metavariables en fórmulas concretas del lenguaje [Fig. 9]. En la [Fig. 10] se muestran dos derivaciones concretas según las fórmulas elegidas para instanciar α y β. Notar que la fórmula elegida para instanciar a α debe cumplir con la condición de que x no pertenezca a su conjunto de variables libres, de lo contrario la derivación resultante no será correcta.

$$\dfrac{\dfrac{\dfrac{\forall x.\,(\alpha \longrightarrow \beta)}{(\alpha \longrightarrow \beta)[x \mapsto x]}\,(E\forall)\,* \qquad \dfrac{}{\alpha}\,(Hip)}{\beta}\,(E\rightarrow)}{\forall x.\,\beta}\,(I\forall)\,**$$

$*\ Por\ definición\ de\ sustitución, (\alpha \longrightarrow \beta)[x \mapsto x] \equiv (\alpha \longrightarrow \beta)$

$**\ x \notin fv(\alpha)\ por\ enunciado\ del\ juicio\ y\ x \notin fv\big(\forall x.\,(\alpha \longrightarrow \beta)\big)\ por\ aparecer\ cuantificada$

Figura 9 - Derivación con justificación de condiciones laterales

Para un humano la verificación de las condiciones no sintácticas y el manejo de metavariables es relativamente simple aun con la falta de una formalización. Sin embargo para un asistente de demostración como el que queremos, esas restricciones deben estar explícitamente formalizadas.

$$\dfrac{\dfrac{\dfrac{\overline{\forall x.(\alpha \to \beta)}\,(Hip)}{\alpha \to \beta}\,(E\forall) \qquad \dfrac{}{\alpha}\,(Hip)}{\beta}\,(E\to)}{\forall x.\beta}\,(I\forall)$$

$\alpha \mapsto P(a,b)$
$\beta \mapsto \bot$

$$\dfrac{\dfrac{\dfrac{\overline{\forall x.(P(a,b) \to \bot)}\,(Hip)}{P(a,b) \to \bot}\,(E\forall) \qquad \overline{P(a,b)}\,(Hip)}{\bot}\,(E\to)}{\forall x.\bot}\,(I\forall)$$

$\alpha \mapsto P(y) \to Q(y)$
$\beta \mapsto \forall y.P(y)$

$$\dfrac{\dfrac{\dfrac{\overline{\forall x.\big((P(y) \to Q(y)) \to \forall y.P(y)\big)}\,(Hip)}{(P(y) \to Q(y)) \to \forall y.P(y)}\,(E\forall) \qquad \overline{P(y) \to Q(y)}\,(Hip)}{\forall y.P(y)}\,(E\to)}{\forall x.\forall y.P(y)}\,(I\forall)$$

Figura 10 - Ejemplo de instanciación de un esquema de derivación

Consideraremos entonces una extensión de la lógica de primer orden llamada Lógica de Orden 1½ en la cual se agregan símbolos para representar metavariables y se expresan propiedades y operaciones sintácticas como "$x \notin fv(\alpha)$" y la substitución "$\alpha[x \mapsto t]$"

La lógica de orden 1½ es definida en [19] junto con un sistema deductivo basado en el cálculo de secuentes. Nuestro objetivo será adaptar la definición de la lógica propuesta e implementar el cálculo de Deducción Natural. El sistema estará basado en el uso de términos nominales, ya que éstos ofrecen soporte intrínseco para metavariables y el concepto de ligadura de variables. Además adaptaremos algoritmos para proponer una solución a la derivación automática de restricciones laterales que buscarán eliminar carga al estudiante sobre el proceso de construcción de la prueba.

2.1 Términos Nominales

Se explican las ideas principales con respecto a los términos nominales. Para una presentación completa referirse a [20], [21], [22] y [23].

Los términos nominales proporcionan una estructura para lenguajes con ligaduras y metavariables. Proveen, por lo tanto:

1. Una categoría de nombres llamados *átomos* (posiblemente clasificados en tipos). Son los nombres que pueden estar ligados en expresiones del lenguaje. En nuestro caso será necesaria sólo una categoría de átomos $\mathbb{A}$, ya que los nombres que pueden ser ligados corresponden a variables de la lógica de primer orden, y éstos son de un único tipo, a saber, las que representan a los individuos. Como es usual, cada categoría de nombres se define como infinita numerable.

2. Un constructor de expresiones llamado abstracción, que sirve para representar las ligaduras. Tiene la forma *[a]t*, donde a es un átomo y t una expresión. Se hacen algunas consideraciones sobre las abstracciones:

a. Las abstracciones no deben ser interpretadas como funciones, sino que [a]t es simplemente una expresión donde a está ligada, i.e. es un nombre local.

b. Las abstracciones no se consideran expresiones bien formadas del lenguaje objeto, sino que son expresiones auxiliares subordinadas. Una abstracción debe ser pasada como argumento a un functor para formar parte de una expresión completa. Por ejemplo, una formula cuantificada (universalmente) será representada como una expresión de la forma $\forall([a]t)$ y que debe ser leída como la aplicación del cuantificador universal a una abstracción.

c. Se introduce un sistema muy simple de categorías de expresiones para asegurar la restricción sintáctica anterior. En este caso, hay dos categorías de expresiones "completas": $\mathbb{T}$ la categoría sintáctica de términos y $\mathbb{F}$ la de fórmulas. Además de éstas, hay categorías para las abstracciones. Alcanzaría con tener sólo abstracciones en la que los átomos se abstraen en fórmulas y cuya categoría se escribe $[\mathbb{A}]\mathbb{F}$. Sin embargo, en la lógica de orden uno y un medio también se introduce una categoría$[\mathbb{A}]\mathbb{T}$ de abstracciones sobre términos, que permite definir operadores con ligaduras a nivel de términos, como lo serán por ejemplo λ en abstracciones funcionales y Σ en las sumas. Luego se asocia una aridad a cada functor que estipula los tipos de sus argumentos y su resultado. Así, $\forall$ tiene una aridad $([\mathbb{A}]\mathbb{F})\mathbb{F}$, mientras que la de Σ es $(\mathbb{T}, \mathbb{T}, [\mathbb{A}]\mathbb{T})\mathbb{T}$.

3. Permutaciones finitas de átomos como mecanismo de renombre para implementar α-conversión. Una permutación finita π es una biyección en los átomos tal que para algún conjunto finito de átomos $\pi(a) \neq a$ y para todos los átomos fuera de ese conjunto $\pi(a) = a$.

   La permutación vacía es la identidad (*id*); π^{-1} es la permutación inversa de π y $\pi \circ \pi'$ a la composición de las permutaciones π y π' que se define como $(\pi \circ \pi')(a) = \pi(\pi'(a))$.
   Se define *(a b)*, una permutación particular llamada swap que intercambia a por b y viceversa y no afecta a ningún otro átomo.

4. Las metavariables se conocen como *incógnitas* y se representan por lo general con letras mayúsculas $X, Y, ...$ Cada una tiene una categoría sintáctica asociada, que en este caso, puede ser $\mathbb{T}$ o $\mathbb{F}$. Una incógnita representa a una expresión arbitraria del tipo correspondiente.
   Asociado a las incógnitas, hay una noción de substitución que llamaremos instanciación. La instanciación no evita la captura de variables (una incógnita representa realmente a una expresión arbitraria) y por lo tanto su acción se define naturalmente.

   Surge un problema cuando consideramos la acción de una permutación sobre una incógnita. No puede calcularse, pero se debe registrar para una futura instanciación de la incógnita. Por lo tanto las incógnitas aparecen por lo general con la forma $\pi \cdot X$. A esta expresión se la conoce como incógnita moderada y se dice que la permutación π está suspendida en *X*.

Haciendo esta salvedad, se puede definir la acción de permutación por recursión en los términos:

$$\#_2 \cdot \#_1 :: TN \to (\mathbb{A} \to \mathbb{A}) \to TN$$
$$\pi \cdot a \ = \pi(a)$$
$$\pi \cdot (\pi' \cdot X) = (\pi \circ \pi') \cdot X$$
$$\pi \cdot [a]t = [\pi(a)](\pi \cdot t)$$
$$\pi \cdot f(t_1, \dots, t_n) = f(\pi \cdot t_1, \dots, \pi \cdot t_n)$$

Una permutación se propaga por la estructura de los términos nominales hasta encontrar un átomo, donde aplica la función de permutación, o una metavariable moderada donde se compone con la permutación existente. *f* representa a cualquier constructor de términos y depende de la instanciación del framework.

Los términos nominales para la lógica de primer orden *t, u, v* se definen inductivamente:

$$(atom)\frac{}{a \in \mathbb{T}}(a \in \mathbb{A}) \qquad (perm)\frac{}{\pi.X \in \tau}(\tau \in \{\mathbb{F}, \mathbb{T}\})$$

$$(bottom)\frac{}{\bot \in \mathbb{F}} \qquad (Imp)\ \frac{\alpha \in \mathbb{F} \quad \beta \in \mathbb{F}}{(\alpha \to \beta) \in \mathbb{F}} \qquad (Abs)\ \frac{t \in \tau}{[a]t \in \tau}\ (a \in \mathbb{A}, \tau \in \{\mathbb{F}, \mathbb{T}\})$$

$$(\forall)\ \frac{[a]t \in [\forall]\mathbb{F}}{\forall t \in \mathbb{F}} \qquad (Sub)\ \frac{t \in [\mathbb{A}]\tau \quad u \in \mathbb{T}}{t[a \to u] \in \tau}\ (\tau \in \{\mathbb{F}, \mathbb{T}\})$$

$$(Fun)\frac{t_1 \in \tau_1, \dots, t_n \in \tau_n}{f(t_1, \dots, t_n) \in \tau}\ (\tau \in \{\mathbb{F}, \mathbb{T}\})$$

Figura 11 - Definición inductiva de términos nominales

Se asume una lista de functores cuyo resultado debe ser $\mathbb{T}$ (símbolos de función) o $\mathbb{F}$ (símbolos de predicado).
Las conectivas tienen el mismo significado tradicional que en la lógica de primer orden: $\bot$ representa al absurdo (bottom), $\to$ la implicación, $\forall$ la cuantificación universal. $t[a \mapsto u]$ representa la substitución que evita la captura de variables. Nótese que, por lo tanto, la substitución es explícita en este lenguaje. La razón de esto, y sus consecuencias, se explican más abajo.

2.2 Frescura

Una afirmación de frescura es una pareja $a\#t$ de un átomo a y un término t, y representa intuitivamente que $a \notin fv(t)$ y se lee como $a\ está\ fresca\ en\ t$.
Se define un contexto de frescura como $\Delta = \{a_1\#X_1, \dots, a_n\#X_n\}$, un conjunto de afirmaciones de frescura donde los términos son siempre metavariables.

Al usar condiciones de frescura y permutaciones, es posible definir la igualdad en cualquier sistema de términos nominales de forma tal que abarque la α-conversión. Por ejemplo, la igualdad en abstracciones puede ser especificada por la regla:

$$\frac{(a\ c) \cdot t = (b\ c) \cdot u}{[a]t = [b]u}\ c\#t, c\#u$$

Podemos definir derivaciones de afirmaciones de frescura como mecanismo de demostración de que un átomo ocurre fresco en un término. Dado un contexto de frescura Δ y una afirmación $a\#t$, se escribe $\Delta \vdash a\#t$ para indicar que existe una derivación de la afirmación usando los elementos de Δ como hipótesis y las reglas de inferencia en [Fig. 12].

$$\frac{}{a\#b}(\#ab) \qquad \frac{\pi^{-1}(a)\#X}{a\#\pi \cdot X}(\neq X) \qquad (\pi \neq Id)$$

$$\frac{}{a\#[a]t}(\#[\,]a) \qquad \frac{a\#t}{a\#[b]t}(\#[\,]b) \qquad \frac{a\#t_1 \cdots a\#t_n}{a\#f(t_1 \cdots, t_n)}(\#f)$$

Figura 12 - Reglas de inferencia para juicios de frescura

La regla (#f) aplica para todos los functores del lenguaje.
La derivación de frescura es decidible. El sistema implementa un algoritmo de derivación automática basado en estas reglas.

2.3 Deducción Natural

Sea $\Gamma = \{\beta_0, \dots, \beta_n\}\ con\ \beta_i \in \mathbb{F},\ \alpha \in \mathbb{F},\ \Delta = \{a_0\#X_0, \dots, a_n\#X_n\}$ un contexto de frescura Un juicio en deducción natural para la lógica de orden 1½ es una tripleta $\Gamma \vdash_\Delta \alpha$

Las reglas de derivación son similares a las de la lógica de primer orden y se presentan en [Fig. 5]. En esta tesis nos restringimos al fragmento con las constantes lógicas $\bot, \rightarrow$ y $\forall$. Este conjunto es suficientemente general para la lógica clásica (aunque en nuestro sistema omitimos, por simplicidad, un axioma de doble negación o demostración indirecta). Las siguientes son algunas características interesantes del sistema:

1) Algunas reglas incluyen lo que llamaremos condiciones laterales. Estas son condiciones que deben cumplirse para aplicar la regla de inferencia y se ven afectadas por el contexto de frescura. Las condiciones laterales son decidibles y su derivación automatizable. Hay dos tipos de condiciones laterales:
    a. Juicios de frescura: Se debe probar que un átomo está fresco en una fórmula a partir de un contexto de frescura. La derivación se realiza usando las reglas de inferencia en [Fig. 4]
    b. Juicios de equivalencia de fórmulas: Se prueba que dos términos son equivalentes construyendo una derivación basada en una teoría de igualdad con α-equivalencia y substitución con evitación de captura de variables llamada SUB (se define más adelante). Como ya se dijo antes, la substitución con control de captura es explícita en este lenguaje. Esto se debe a la presencia de metavariables (incógnitas) sobre las cuales la substitución no se puede efectuar. Por lo tanto, no es posible introducirla como una meta-operación. La teoría SUB debe ser introducida para proporcionar axiomas para la igualdad sintáctica, teniendo en cuenta el correcto comportamiento de la substitución además de la conversión α.
2) La regla (Fr) permite introducir nuevos átomos en el contexto de frescura de la derivación. En algunas situaciones se necesitará generar nuevas afirmaciones de frescura del tipo $a\#X$ para incluir átomos frescos y poder aplicar ciertas reglas. Notar que siempre se incluyen átomos que no ocurren sintácticamente en alguna parte de la derivación, basándose en que como el conjunto de átomos es infinito (contable) siempre podemos encontrar un átomo que no aparece en la derivación.
3) La regla $(I\forall)$ agrega una condición lateral del tipo (a) $\Delta\vdash a\#\Gamma$. Esta condición puede leerse como "a no ocurre libre en las hipótesis disponibles". Para que esta regla se pueda aplicar, se debe demostrar también la condición lateral.
4) La regla (reesc) permite intercambiar una fórmula por otra equivalente según la teoría SUB. Para que se pueda aplicar, se debe demostrar también que $\Delta\ \vdash_{SUB} t = u$

$$(\text{Hip})\frac{}{\{\alpha\} \cup \Gamma \vdash_\Delta \alpha} \quad (\text{E} \perp)\frac{\Gamma \vdash_\Delta \perp}{\Gamma \vdash_\Delta \alpha} \quad (\text{E} \rightarrow)\frac{\Gamma \vdash_\Delta \alpha \rightarrow \beta \quad \Gamma \vdash_\Delta \alpha}{\Gamma \vdash_\Delta \beta} \quad (\text{I} \rightarrow)\frac{\Gamma \cup \{\alpha\} \vdash_\Delta \beta}{\Gamma \vdash_\Delta \alpha \rightarrow \beta}$$

$$(\text{I}\forall)\frac{\Gamma \vdash_\Delta \alpha}{\Gamma \vdash_\Delta \forall x\alpha}(\Delta \vdash x\#\Gamma) \quad (\text{E}\forall)\frac{\Gamma \vdash_\Delta \forall x\alpha}{\Gamma \vdash_\Delta \alpha[x \mapsto t]} \quad (\text{reesc})\frac{\Gamma \vdash_\Delta \alpha'}{\Gamma \vdash_\Delta \alpha}(\Delta \vdash_{sub} \alpha = \alpha')$$

$$(Fr)\frac{\Gamma \vdash_{\Delta \cup \{a\#X_1, \ldots, a\#X_n\}} \alpha}{\Gamma \vdash_\Delta \alpha}(n \geq 1, a \notin \Gamma, a \notin \Delta)$$

Figura 13 - Reglas de inferencia para la lógica de orden 1½

En el siguiente ejemplo se da la derivación para $X, \forall a. (X \rightarrow Y) \vdash_{a\#X} \forall a. Y$ (el contexto queda implícito)

$$(I\forall)\frac{(E\rightarrow)\dfrac{(Hip)\dfrac{}{X} \qquad (Resc)\dfrac{(E\forall)\dfrac{(Hip)\dfrac{}{\forall a. (X \rightarrow Y)}}{(X \rightarrow Y)[a \mapsto a]}}{(X \rightarrow Y)}(a\#X \vdash X \rightarrow Y = (X \rightarrow Y)[a \mapsto a])}{Y}}{\forall a. Y}(a\#X \vdash a\#X, a\#X \vdash a\#\forall a. (X \rightarrow Y))$$

Al aplicar la regla *(Resc)* se genera como condición lateral la prueba en la teoría SUB de la igualdad $X \rightarrow Y = (X \rightarrow Y)[a \mapsto a]$ con hipótesis $a\#X$ (afirmación de frescura que está en el contexto).
Además, la regla de introducción del cuantificador universal genera como condición lateral 2 pruebas de frescura de a sobre las hipótesis disponibles.

2.4 Resultados Meta-Teóricos

Son de interés los siguientes resultados acerca del sistema descrito.

2.4.1 Consistencia

En [19], la lógica de orden 1½ se presenta bajo la forma de un cálculo de secuentes. Para este se da una prueba de la eliminación cut, y por lo tanto de su consistencia. Hemos probado que la presentación de Deducción Natural dada es equivalente a la original en el cálculo de secuentes. Esta prueba es estándar por inducción en las reglas, por lo que se omiten los detalles.

La consistencia de este sistema se sigue del hecho de que si $\bot$ fuese derivable en él, entonces sería derivable en el original, lo cual no puede pasar.

2.4.2 Derivabilidad de cut

Puede verse que la siguiente regla cut es derivada de nuestro sistema, usando la introducción y la eliminación de la implicación:

$$\frac{\Gamma, \gamma \vdash_{\Delta} \alpha \qquad \Theta \vdash_{\Delta'} \gamma}{\Gamma, \Theta \vdash_{\Delta \cup \Delta'} \alpha}$$

Que puede generalizarse por inducción a:

$$\frac{\gamma_1 \dots \gamma_n \vdash_{\Delta} \alpha \qquad \Gamma_1 \vdash_{\Delta_1} \gamma_1 \dots \Gamma_n \vdash_{\Delta_n} \gamma_n}{\bigcup_{i=1\dots n} \Gamma_i \vdash_{\Delta \cup (\bigcup_{i=1\dots n} \Delta_i)} \alpha}$$

Esta última regla a la que llamaremos cut generalizado (gcut) permitirá que se use un teorema (lema) como una regla de inferencia. En efecto, si tenemos un teorema como el expuesto como la primera premisa de la regla, entonces podemos obtener su conclusión en cualquier contexto en el cual podamos probar sus hipótesis, como se indica en el resto de las premisas del cut generalizado.

Como nuestros teoremas son esquemáticos, nos gustaría poder instanciarlos. Esto se logra al aplicar la regla siguiente:

2.4.3 Instanciación

Es admisible la siguiente regla:

$$\frac{\Gamma \vdash_{\Delta} \alpha}{\Gamma\sigma \vdash_{\Delta'} \alpha\sigma} \; \Delta' \vdash \Delta\sigma$$

Para cualquier substitución σ de metavariables y un contexto Δ'. La condición lateral significa que para cada afirmación de frescura $a\#t \in \Delta\sigma$ se debe probar que $\Delta' \vdash a\#t$

2.5 Matching de Fórmulas en Sintaxis Nominal

Se dijo que el sistema debe poder tomar un lema (derivación completada previamente) y utilizarlo como una regla de inferencia en futuras pruebas.

Supóngase que se está construyendo la derivación de un juicio dado. En un estado de la misma se tiene que probar $\Gamma \vdash_{\Delta} \alpha$ (derivar α a partir del conjunto de hipótesis Γ y el contexto de frescura Δ), y para esto se quiere usar un lema conocido $\Theta \vdash_{\Delta'} \gamma$. Esto es en efecto posible y justificado por la correctitud de las reglas del cut generalizado y de instanciación. Entonces, se quiere obtener α como la aplicación de una instancia adecuada del lema. Luego, las hipótesis de esta instancia deben ser derivadas a partir de las hipótesis disponibles, o sea Γ.

Notar que el lema podrá tener elementos sintácticos (variables, metavariables, símbolos de predicado, etc.) distintos (y disjuntos) de los que están presentes en la derivación. Por lo tanto, antes de aplicar el lema se debe transformar el conjunto de símbolos del lema aplicando una substitución que transforme sus metavariables en términos nominales de la prueba actual.

Ahora, la cuestión principal es como encontramos esa substitución (y por lo tanto la instancia) adecuada a aplicar al lema. Para esto se debe "emparejar" el resultado deseado α contra la conclusión (esquemática) del lema γ. El proceso, llamado *matching*, en general compara una expresión concreta contra un patrón (o expresión abstracta) que contiene ciertas variables, y encuentra, si existe, la manera más general de instanciar el patrón de forma que se vuelva *igual* a la expresión concreta. Lo que se debe establecer es el significado de lo que es *igual*, i.e. bajo qué relación de igualdad (sintáctica) debe llevarse a cabo el proceso de matching.
En principio se definen tres candidatos:

1. Simple, igualdad sintáctica.
2. Igualdad módulo conversión α.
3. Igualdad módulo la teoría SUB.

Para empezar, tener en cuenta que:

1. El matching con respecto a la igualdad simple puede implementarse fácilmente, se da un algoritmo más abajo.
2. El matching módulo conversión α está resuelto en [15].
3. El matching módulo SUB, hasta donde sabemos no ha sido resuelto.

A pesar de esto, argumentaremos en nuestro caso a favor del matching simple, así que no tendremos que ocuparnos del caso que considera substituciones explícitas.
El matching contra la conversión α ahorrará al estudiante algunos pasos en la derivación, básicamente la aplicación de la regla de reescritura, cosa que no es deseable por motivos didácticos. Se quiere que el estudiante sea consciente de las reescrituras necesarias y la conversión α necesarias para completar ciertas derivaciones. La situación sería completamente diferente si estuviésemos motivados por un tipo de usuario profesional. Para este último caso la investigación del matching en SUB, estaría plenamente justificada.

En nuestro caso, el estudiante deberá realizar algunas transformaciones en los goals antes de poder aplicar el lema. Estas transformaciones son conversiones α y reescrituras que expliciten las substituciones. Estas últimas serán necesarias para usar lemas en los que la substitución aparezca explícitamente en la conclusión, y no pudiesen ser evitados excepto con el uso de

matching en SUB, que es aún desconocido. Ciertamente esto tampoco es un problema en nuestro caso. Queremos que el estudiante esté en total control de la derivación en cada paso y, en particular, el análisis de fórmulas en funciones y argumentos que exige la substitución es un rasgo de los lenguajes lógicos desde Frege que usamos permanentemente y de que queremos que los estudiantes sean plenamente conscientes.

Dicho esto, se da un algoritmo para implementar el matching simple.
Se define un problema de matching nominal P como un conjunto de problemas atómicos de dos tipos:

- Problemas de igualdad de la forma $t \approx ? t'$ con t y t' términos nominales y $\approx$ representa a la igualdad simple.
- Problemas de frescura de la forma $a \#? t$ con $a \in \mathbb{A}$ y t un término nominal.

La solución a un problema P es una substitución σ y un contexto de frescura Δ tales que

- $\Delta \vdash a \# \sigma(t)$ para cada $(a \#? t) \in P$
- $\Delta \vdash_{SUB} \sigma(t) = \sigma(t')$ para cada $(t \approx ? t') \in P$

El algoritmo que se implementa en el sistema se basa en el trabajo en [23], [24], [25], [26] y [27]. Dado un problema de matching P, el algoritmo decide si el problema tiene solución o no y en caso afirmativo devuelve la solución más general e idempotente.

El algoritmo realiza transformaciones a cada subproblema en P en 2 fases:

1) Aplica transformaciones σ sobre los problemas tipo $t \approx ? t'$ hasta que se acaben.
2) Aplica transformaciones Δ sobre problemas tipo $a \#? t$ hasta que se acaben.

Si en cualquiera de las dos fases, quedan subproblemas a los que no se pueden aplicar transformaciones, entonces al algoritmo falla. De lo contrario al finalizar la fase 2, el problema P está vacío y la solución *s* se construye como:

$$P \overset{\sigma_1}{\Rightarrow} \dots \overset{\sigma_n}{\Rightarrow} P' \overset{\Delta_1}{\Rightarrow} \dots \overset{\Delta_n}{\Rightarrow} \emptyset$$

$$s = (\sigma_1 \cup \dots \cup \sigma_n, \Delta_1 \cup \dots \cup \Delta_n)$$

Las transformaciones posibles para los subproblemas son:

$$(\approx atomo) \quad \{a \approx ? a\} \uplus P \overset{\varepsilon}{\Rightarrow} P$$

$$(\approx suspensión) \quad \{\pi \cdot X \approx ? \pi' \cdot X\} \uplus \overset{\varepsilon}{\Rightarrow} \{a \#? X | a \in ds(\pi, \pi')\} \cup P$$

$$(\approx metavariable) \quad \{\pi \cdot X \approx ? t\} \uplus P \overset{\sigma}{\Rightarrow} \sigma P \qquad con\ \sigma = [X \mapsto \pi^{-1} t]\ si\ X\ no\ ocurre\ en\ t$$

$$(\approx abstraccion) \quad \{[a]t \approx ? [a]\mu\} \uplus P \overset{\varepsilon}{\Rightarrow} \{t \approx ? \mu\} \cup P$$

$$(\approx funcion) \quad \{f(t_1 \cdots t_n) \approx ? f(\mu_1 \cdots \mu_n)\} \uplus P \overset{\varepsilon}{\Rightarrow} \{t_1 \approx ? \mu_1, \cdots, t_n \approx ? \mu_n\} \cup P$$

Figura 14 - Transformaciones de Fase 1 para problema de matching

$$(\#atomo) \qquad \{a\#b\} \uplus P \overset{\emptyset}{\Rightarrow} P$$

$$(\#suspension) \qquad \{a\#\pi \cdot X\} \uplus P \overset{\Delta}{\Rightarrow} P \qquad con\ \Delta = \{\pi^{-1} \cdot a\#X\}$$

$$(\#abstraccion\ 1) \qquad \{a\#?\,[a]t\} \uplus P \overset{\emptyset}{\Rightarrow} P$$

$$(\#abstraccion\ 2) \qquad \{a\#?\,[b]t\} \uplus P \overset{\emptyset}{\Rightarrow} \{a\#?t\} \cup P$$

$$(\#funcion) \qquad \{a\#?\,f(t_1 ... t_n)\} \uplus P \overset{\emptyset}{\Rightarrow} \{a\#?\,t_1, \cdots, a\#?\,t_n\} \cup P$$

Figura 15 - Transformaciones de Fase 2 para problema de matching

$\emptyset$ representa al contexto vacío y ε a la substitución vacía.

Considérese el siguiente ejemplo donde se busca una substitución para matchear el termino $\forall[a](\alpha \rightarrow \bot)$ con $\forall[x](P(x) \rightarrow \bot)$ bajo el contexto vacío.

El problema de matching P se define como:

$$P = \{\forall[a](\alpha \rightarrow \bot) \approx ?\, \forall[a](P(a) \rightarrow \bot)\}$$

$$\{\forall[a](\alpha \rightarrow \bot) \approx ?\, \forall[a](P(a) \rightarrow \bot)\}$$

Fase 1

$$\overset{\varepsilon}{\Rightarrow} \{[a](\alpha \rightarrow \bot) \approx ?\, [a](P(a) \rightarrow \bot)\}$$
$$\overset{\varepsilon}{\Rightarrow} \{(\alpha \rightarrow \bot) \approx ?\, (P(a) \rightarrow \bot)\}$$
$$\overset{\varepsilon}{\Rightarrow} \{\alpha \approx ?\, P(a),\ \ \bot \approx ?\, \bot\}$$
$$\overset{[\alpha \mapsto P(a)]}{\Longrightarrow} \emptyset$$

El resultado del algoritmo es $(\{\alpha \mapsto P(a)\}, \emptyset)$.

El siguiente ejemplo tiene una pequeña variante con respecto al caso anterior, pero genera un problema para el que no hay solución. En la fase 1 se aplican reglas hasta que se llega a una condición que no puede reducirse, el algoritmo termina sin devolver una solución.

$$P = \{\forall[a](\alpha \rightarrow \bot) \approx ?\, \forall[x](P(x) \rightarrow \bot)\}$$

$$\{\forall[a](\alpha \rightarrow \bot) \approx ?\, \forall[x](P(x) \rightarrow \bot)\}$$

Fase 1

$$\overset{\varepsilon}{\Rightarrow} \{[a](\alpha \rightarrow \bot) \approx ?\, [x](P(x) \rightarrow \bot)\}$$

Notar que es sencillo implementar el algoritmo de matching con conversión α a partir del anterior. Hace falta agregar la siguiente regla para la abstracción sobre átomos distintos en la Fase 1:

$$(\approx abstraccion\ 2) \quad \{[a]t \approx ?\, [b]\mu\} \uplus P \overset{\varepsilon}{\Rightarrow} \{t \approx ?\, (ab) \cdot \mu, a\#?\,\mu\} \cup P$$

Con esto, el problema del ejemplo 2 tiene solución:

$$\{\forall[a](\alpha \to \bot) \approx ?\, \forall[x](P(x) \to \bot)\}$$

Fase 1

$$\overset{\varepsilon}{\Rightarrow} \{[a](\alpha \to \bot) \approx ?\, [x](P(x) \to \bot)\}$$
$$\overset{\varepsilon}{\Rightarrow} \{(\alpha \to \bot) \approx ?\, (P(a) \to \bot), a\#?\, (P(x) \to \bot)\}$$
$$\overset{\varepsilon}{\Rightarrow} \{\alpha \approx ?\, P(a),\ \bot \approx ?\, \bot,\ a\#?\, (P(x) \to \bot)\}$$
$$\xRightarrow{[\alpha \mapsto P(a)]} \{a\#?\, (P(x) \to \bot)\}$$

Fase 2

$$\overset{\emptyset}{\Rightarrow} \{a\#?\, P(x), a\#?\, \bot)\}$$
$$\overset{\emptyset}{\Rightarrow} \{a\#?\, x\}$$
$$\overset{\emptyset}{\Rightarrow} \emptyset$$

Por motivos de completitud del trabajo, el algoritmo se implementa en dos modos: uno para estudiantes con igualdad simple y otro con igualdad módulo conversión α para uso avanzado. Por defecto el modo activado es el de igualdad simple.

3 Descripción del Sistema

En los siguientes puntos se detallará el diseño de un asistente de demostración para el sistema precedente, incorporando las funcionalidades detalladas en la Introducción. Además se muestran los algoritmos de derivación automática de juicios de frescura y equivalencia en sintaxis nominal así como el matching de fórmulas.

El sistema está implementado de forma tal que la sintaxis nominal es independiente del sistema lógico. El objetivo es generar un framework que permita aplicar conceptos de la sintaxis nominal a distintos sistemas lógicos. Además, desde un punto de vista de diseño, favorece el desacoplamiento y reduce el impacto ante modificaciones.

3.1 Representación de fórmulas lógicas

Se define el lenguaje de objetos `Obj` que representa a los individuos en el dominio de interpretación del lenguaje de primer orden:

```
data Obj =   Var  Char
           | Cons String
           | Func String [Obj]
           | ObjVar Char
           | ObjSust Obj Char Obj
```

`Var` representa a las variables de individuos, `Cons` a las constantes, `Func` a los functores sobre términos, `ObjVar` es la metavariable sobre los términos y `ObjSust` la substitución sobre términos

También se define el lenguaje Form de las fórmulas de la lógica de primer orden

```
data Form =   Pred String [Obj]
            | MetaVar Char
            | Bottom
            | Neg Form
            | Imp Form Form
            | ForAll Char Form
            | Sust Form Char Obj
```

Notar que se agrega una regla de construcción para substituciones. Esto será de gran importancia cuando definamos las reglas de deducción natural para la lógica.

3.2 Representación de términos nominales

Sea `NominalTerm` el lenguaje de los términos nominales según

```
data NominalTerm  =    Atom Char
                     | MVar Perm Char
                     | Abs Char NominalTerm
                     | TermFormer String [NominalTerm]
```

Como se dijo, los términos nominales son independientes de la sintaxis de la lógica de primer orden. Se define entonces una función polimórfica de traducción de elementos de `Form` y `Obj` a elementos de `NominalTerm`.

```
nMap :: Obj → NominalTerm
nMap (Var x)    = Atom x
nMap (Cons c)   = TermFormer ("const_" ++ (show c)) []
nMap (Func nombre obj) = TermFormer ("func_" ++ nombre) (map nMap obj)
nMap (ObjVar x) = MVar [] x
nMap (ObjSust o1 a o2) = TermFormer "ObjSust" [Abs a (nMap o1), nMap o2]


nMap :: Form → NominalTerm
nMap (Pred nombre obj) = TermFormer ("pred_" ++ nombre) (map nMap obj)
nMap (MetaVar x)       = MVar [] x
nMap (Bottom)          = TermFormer "Bottom" []
nMap (Imp f1 f2)       = TermFormer "Imp" [nMap f1, nMap f2]
nMap (ForAll x f)      = TermFormer "ForAll" [Abs x (nMap f)]
nMap (Sust f1 a f2)    = TermFormer "Sust" [Abs a (nMap f1), nMap f2]
```

3.3 Pruebas y Derivaciones

Todas las estructuras están construidas como tipos parametrizados, lo cual permite implementar el comportamiento de la derivación de una prueba y la aplicación de reglas independientemente del tipo de fórmulas que contienen los nodos. Esto es muy útil ya que si bien el sistema fue construido para representar derivaciones de fórmulas de la lógica de orden 1½, internamente se utilizará la misma estructura para construir un algoritmo de derivación automática de frescura e igualdad de términos nominales. Esto trae otro importante beneficio: se crea un framework de prueba que podrá ser utilizado para construir asistentes de demostración en otros sistemas lógicos.

El estado de una prueba en el sistema se representa como un árbol n-ario (`NTree`), dónde la raíz es la conclusión α y las hojas son o bien hipótesis del juicio o bien fórmulas cuya demostración está pendiente (goals). Una prueba está completa cuando todas las hojas son hipótesis del juicio o hipótesis derivadas canceladas.

```
type Prueba a = NTree (NodoPrueba a)
```

Los nodos del árbol son una estructura especial que almacena toda la información de la prueba hasta ese punto. Es poco efectivo almacenar un conjunto general de hipótesis o un contexto general ya que ambos pueden variar a lo largo de la prueba de forma tal que para distintos nodos apliquen distintas hipótesis y distintos contextos.
Cada nodo está compuesto por:

- α_i , la fórmula en el nodo.
- $\Gamma_i = \{\beta_0^i, \dots, \beta_n^i\}$, conjunto de hipótesis locales.
- Δ_i, contexto local.
- $\Pi_i = \{\Delta_i \vdash \delta_1, \dots, \Delta_i \vdash \delta_k\}$, condiciones laterales generadas por la regla aplicada en α_i
- $regla$, un string representando el nombre de la regla aplicada en α_i
- $estaProbado$, un valor booleano que indica si la prueba de la fórmula α_i está completa.

- $i \in Nat$, un identificador.

Se define la estructura de una derivación como una agrupación de una prueba principal y n pruebas secundarias. La prueba principal es la que representa la construcción del juicio y el único que determina si es derivable, mientras que las pruebas secundarias sirven como "borradores" de prueba que podrán ser unidos con la prueba principal.
Las pruebas secundarias se generan durante la construcción de la derivación a partir de las hipótesis disponibles.

```
data Derivacion a = D (Prueba a) [Prueba a]
```

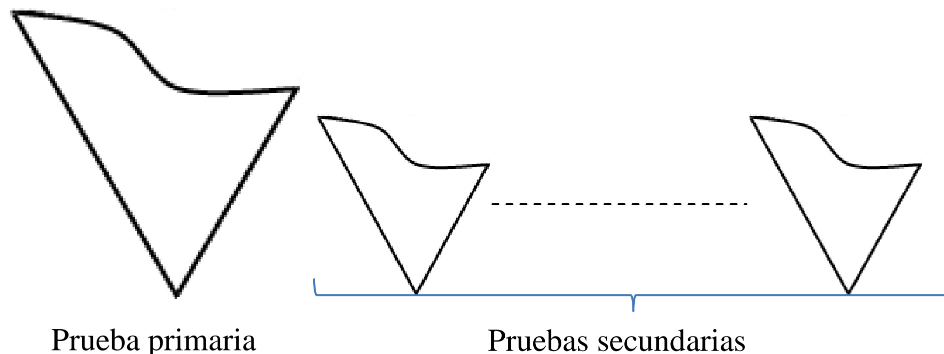

Figura 16 - Estructuras de datos de derivaciones

Notar que para las derivaciones en lógica de 1½, las estructuras anteriores se instancian con el tipo *Form*

Las condiciones laterales que aparecen en la estructura de los nodos se generan al aplicarse una regla de inferencia. Para que un nodo se considere como probado, no alcanza con que todas sus premisas estén probadas sino que además todas las condiciones laterales deben ser validadas (en el caso de los juicios de frescura o de igualdad, estos deben ser derivables). Si bien la verificación de las condiciones es automática, deberá ser el estudiante quien inicie la verificación, esto es, deberá ingresar un comando para validar las condiciones laterales en determinado nodo. El motivo de esto se basa en el concepto de que el sistema es un asistente y no un demostrador: el hecho de que el estudiante deba iniciar la validación implica que sea consciente de la existencia de la misma y la entienda; de lo contrario podría pasar desapercibida.

3.4 Reglas de Inferencia

Para implementar los mecanismos de razonamiento bottom-up y top-down , cada conectiva lógica debe tener una regla de inferencia de introducción y de eliminación en su variante backward y forward.
Dada una derivación, las reglas backward podrán aplicarse tanto en la prueba principal como en las pruebas secundarias, mientras que las reglas forward solo se podrán aplicar en pruebas secundarias.

Se define un tipo de datos `ReglaDerivacion` parametrizado en un tipo `a` con dos constructores `Backward` y `Forward`.

```
data ReglaDerivacion a =
          Backward (a -> [a] -> [Hipotesis a] -> FreshContext ->   (String,
     [a], [Hipotesis a], FreshContext, SideCondition))
      |     Forward  (a -> [a] -> [Hipotesis a] -> FreshContext -> (String, (a,
     [Hipotesis a]), [(a, [Hipotesis a])], FreshContext, SideCondition))
```

En las secciones siguientes se explica con detalle cada uno de los argumentos de la definición.

Además de las reglas por conectivas, se agregan tres más:

- Hipótesis: Permite cerrar una hoja del árbol y dar por finalizada la prueba de una fórmula, siempre y cuando dicha fórmula esté dentro del conjunto de hipótesis.
- Ampliar contexto: Permite agregar condiciones de frescura del tipo *a#X* al contexto, donde a una variable y X una meta-variable, bajo la precondición de que la variable a no ocurre en la sintaxis de ninguna fórmula en la prueba. Se basa en la idea de que siempre puedo encontrar nuevas variables que no ocurran en ninguna fórmula (el conjunto de las variables es infinito contable)
- Reescritura: Permite cambiar una fórmula por otra siempre y cuando pueda probarse que son equivalentes bajo la teoría nominal SUB. En general es usada para eliminar substituciones que aparecen en las fórmulas y α-equivalencia.

Las reglas de inferencia se implementan como funciones sobre un tipo genérico (para el caso particular de ANDY, serán las fórmulas de la lógica de orden 1½). Cada regla tendrá su variante forward y backward.

3.4.1 Reglas Backward

Estas reglas se aplican sobre cualquier hoja del árbol de un árbol de prueba (primario o secundario). En función de la fórmula y las hipótesis y contexto de frescura que la afectan, el resultado principal de la aplicación de una regla sobre dicha fórmula es una lista (posiblemente vacía) de fórmulas derivadas. Estas nuevas fórmulas se incluyen en el árbol como hijos directos de la fórmula.
Además, las reglas backward pueden generar hipótesis derivadas, condiciones laterales para el nodo o nuevas condiciones de frescura.

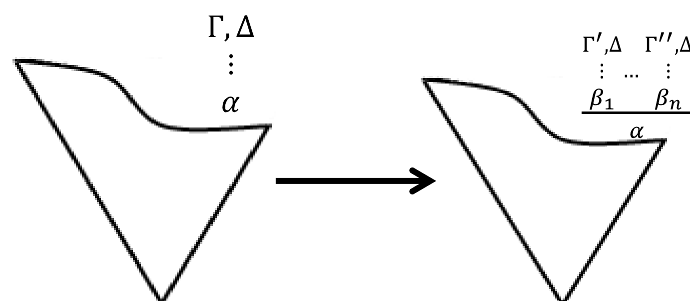

Figura 17 - Esquema de aplicación de reglas backward

Las reglas backward son funciones que reciben una fórmula, una lista de argumentos de tipo *Form* o de tipo *Obj*, una lista de hipótesis y un contexto de frescura. Luego, retorna una quíntupla con:

- El nombre de la regla aplicada (String)
- Una lista de duplas donde el primer componente es una premisa derivada y el segundo componente es la lista de hipótesis temporales que la afectan. Estas fórmulas serán hijas de la fórmula original.
- Una lista de nuevas hipótesis que afectarán al conjunto de hipótesis de las formulas derivadas.
- Una lista de afirmaciones de frescura que afectan a las formulas derivadas.
- Una lista de juicios que definen las condiciones laterales que aplican sobre el nodo.

La [Tabla 1] resume la definición de cada regla backward. Cada regla se especifica de la siguiente manera:

- A la izquierda aparecen los argumentos de la regla en este orden: la fórmula sobre la que se aplica la regla de inferencia, una lista de argumentos para completar la aplicación de la regla (puede ser vacía), un conjunto de hipótesis y un contexto de frescura.
- A la derecha se muestra el resultado de aplicar la regla donde aparece primero una lista de duplas (nueva premisa, hipótesis derivadas), luego la lista de afirmaciones de frescura y por último la lista de condiciones laterales.

Por ejemplo, la introducción de la implicación

$$\alpha \rightarrow \beta, [\,], \Gamma, \Delta \Rightarrow [(\beta, [\alpha])], [\,], [\,]$$

Se aplica sobre la fórmula $\alpha \rightarrow \beta$, no recibe argumentos y no necesita información especial sobre el conjunto de hipótesis disponibles ni sobre el contexto. Luego devuelve una única dupla con premisa β afectada por la nueva hipótesis temporal α.

Otro caso interesante es el de la eliminación de la implicación:

$$\beta, [\alpha], \Gamma, \Delta \Rightarrow [(\alpha \rightarrow \beta, [\,]), (\alpha, [\,])], [\,], [\,]$$

La regla se puede aplicar sobre cualquier fórmula β y necesita indicarse una fórmula cualquiera α en su lista de argumentos. El retorno se compone de dos duplas: la primera es la implicación $\alpha \rightarrow \beta$sin hipótesis temporales, y la segunda es α también sin hipótesis temporales.

Nombre de la regla	**Definición**
Eliminación del absurdo	$\alpha, [\,], \Gamma, \Delta \Rightarrow [(\bot, [\,])], [\,], [\,]$
Introducción de la implicación	$\alpha \rightarrow \beta, [\,], \Gamma, \Delta \Rightarrow [(\beta, [\alpha])], [\,], [\,]$
Eliminación de la Implicación	$\beta, [\alpha], \Gamma, \Delta \Rightarrow [(\alpha \rightarrow \beta, [\,]), (\alpha, [\,])], [\,], [\,]$
Introducción de cuantificación universal	$\forall x. \alpha, [\,], \Gamma, \Delta \Rightarrow [(\alpha, [\,])], [\,], [\Delta \vdash x \# \Gamma]$
Eliminación de cuantificación universal	$\alpha[x \rightarrow t], [\,], \Gamma, \Delta \Rightarrow [(\forall x. \alpha, [\,])], [\,], [\,]$

Rescritura	$\alpha, [\beta], \Gamma, \Delta \Rightarrow [(\beta, [\,])], [\,], [\Delta \vdash_{SUB} \alpha = \beta]$
Hipótesis	$\alpha, [\,], \Gamma, \Delta \Rightarrow [\,], [\,], [\alpha \in \Gamma]$
Ampliar contexto	$\alpha, [a], \Gamma, \Delta \Rightarrow [(\alpha, [\,])], [a\#X_1 \ldots a\#X_n], [a \notin \Gamma, a \notin \Delta]$ $donde\ X_1 \ldots X_n son\ las\ metavariables\ que\ ocurren\ en\ la\ derivación$

Tabla 8 - Definición de reglas backward

Notar que la única regla terminal es la de Hipótesis, ya que no genera nuevas premisas.

Las condiciones laterales se generan para el nodo donde aplica la regla de inferencia, sin embargo no se comprueban de manera inmediata.

3.4.2 Reglas Forward

Las reglas forward son un mecanismo de razonamiento top-down. Se aplican sobre la raíz de pruebas secundarias y derivan una nueva raíz para la prueba. Algunas reglas pueden construir también formulas hermanas, o sea, fórmulas que compartirán como padre directo a la nueva raíz derivada.

Considérese por ejemplo el caso de la eliminación de la implicación. Esta regla se aplica sólo sobre pruebas secundarias cuya raíz sea una fórmula que tiene como conectiva principal una implicación

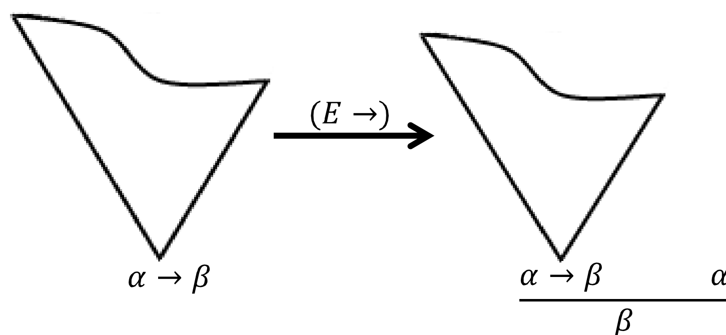

Figura 18 - Esquema de aplicación de reglas forward

Cuando se aplica la regla sobre el árbol secundario de la izquierda, el resultado será una nueva raíz β y una obligación de prueba de la fórmula α para completar la correcta aplicación de la regla. Esto es una variante importante frente a la versión anterior de ANDY donde para aplicar reglas forward con más de una premisa, se necesitaba dar pruebas completas de todas ellas. Ahora, al habilitar la aplicación de reglas backward en pruebas secundarias, se puede eliminar esa restricción.

Las reglas forward reciben los mismos argumentos que las backward, pero devuelven un conjunto de datos distintos:

- El nombre de la regla aplicada.
- Una fórmula que será la nueva raíz de la prueba secundaria.

- Una lista de duplas en las que el primer componente representa a las fórmulas hermanas y el segundo componente a las hipótesis derivadas para esas fórmulas. Este caso es importante para reglas como la eliminación de la disyunción dónde además de generar dos fórmulas hermanas, cada una de ellas tendrá un conjunto distinto de hipótesis disponibles.
- Una lista de afirmaciones de frescura derivadas. Esta lista deberá propagarse en todo el árbol original y actualizar las condiciones laterales prexistentes.
- Una lista de juicios que representan a las condiciones laterales que aplican a la fórmula.

La [Tabla 2] resume la implementación de las reglas forward. Se especifican las reglas forward de la siguiente manera

- A la izquierda se muestran los argumentos en orden: primero la fórmula en la raíz de la prueba secundaria, luego una lista (posiblemente vacía) de fórmulas, el conjunto de hipótesis disponibles y por último el contexto de frescura.
- A la derecha se muestra el retorno de la regla. Primero una dupla con la nueva raíz y las hipótesis que deriva. Después la lista de duplas con nuevas premisas para la nueva raíz y una lista de hipótesis para cada una. Luego, la lista de afirmaciones de frescura y por último la lista de condiciones laterales.

Por ejemplo, la introducción de la implicación

$$\beta, [\alpha], \Gamma, \Delta \Rightarrow \alpha \to \beta, [\,], [\,], [\,]$$

Se aplica sobre una fórmula cualquiera β y se le da como argumento otra fórmula cualquiera α. Además se indica las hipótesis disponibles y el contexto de frescura. Luego, la aplicación de la regla devuelve primero la fórmula $\alpha \to \beta$, la nueva raíz de la prueba. La regla no genera otras premisas, no modifica el contexto de frescura ni genera condiciones laterales.

En el caso de la eliminación de la implicación

$$\alpha \to \beta, [\,], \Gamma, \Delta \Rightarrow \beta, [(\alpha, \Gamma)], [\,], [\,]$$

Notar que el segundo componente del retorno es una lista con la dupla (α, Γ), por lo que además de modificar la raíz de la prueba a β, se agrega una premisa α con Γ como conjunto de hipótesis disponibles.

Nombre de la regla	**Definición**
Eliminación del absurdo	$\bot, [\alpha], \Gamma, \Delta \Rightarrow \alpha, [\,], [\,], [\,]$
Introducción de la implicación	$\beta, [\alpha], \Gamma, \Delta \Rightarrow \alpha \to \beta, [\,], [\,], [\,]$
Eliminación de la Implicación	$\alpha \to \beta, [\,], \Gamma, \Delta \Rightarrow \beta, [(\alpha, \Gamma)], [\,], [\,]$
Introducción de cuantificación universal	$\alpha, [x], \Gamma, \Delta \Rightarrow \forall x. \alpha, [\,], [\,], [\Delta \vdash x \# \Gamma]$
Eliminación de cuantificación universal	$\forall x. \alpha, [t], \Gamma, \Delta \Rightarrow \alpha[x \to t], [\,], [\,], [\,]$
Rescritura	$\alpha, [\beta], \Gamma, \Delta \Rightarrow \beta, [\,], [\,], [\Delta \vdash_{SUB} \alpha = \beta]$

Hipótesis	$\alpha, [\,], \Gamma, \Delta \Rightarrow [\,], [\,], [\,], [\alpha \in \Gamma]$
Ampliar contexto	$\alpha, [a], \Gamma, \Delta \Rightarrow \alpha, [\,], [a\#X_1 \ldots a\#X_n], [a \notin \Gamma, a \notin \Delta]$ $donde\ X_1 \ldots X_n son\ las\ metavariables\ que\ ocurren\ en\ la\ derivación$

Tabla 9 - Definición de reglas forward

3.5 Automatización de Demostración de Condiciones Laterales

Durante la construcción de la prueba se pueden generar varias condiciones laterales que deberán ser verificadas para completar la derivación de un juicio. La verificación de estas consiste en la construcción de una derivación utilizando reglas muy distintas de las de la Deducción Natural; si existe una derivación, la condición lateral se cumple y por lo tanto la regla de inferencia se puede aplicar.
Por lo tanto la verificación de las condiciones implica un conocimiento por parte del estudiante de otro sistema deductivo con nuevas reglas de inferencia, lo cual no es para nada deseable ya que se desvía del objetivo real de la herramienta que es el aprendizaje del cálculo de Deducción Natural. Esta sobrecarga de conceptos puede perjudicar la experiencia del alumno con el sistema.

Por este motivo es importante que la verificación de las condiciones laterales pueda hacerse automáticamente sin intervención del alumno, a la vez de que se explicitan en la construcción de la prueba ya que son conceptualmente importantes para la aplicación y entendimiento de las reglas de inferencia.

Para lograr esto, se implementan en el sistema algoritmos que permitan la derivación automática de juicios de frescura y juicios de equivalencia de términos. Los detalles de esto se dan en el Anexo I.

3.6 Aplicación de Lemas

La aplicación de lemas en una derivación es una característica especial de ANDY1½.

En la aplicación de un lema se debe indicar el lema $\Gamma' \vdash_{\Delta} \alpha'$ y la fórmula β de la prueba principal donde se hará la unión.

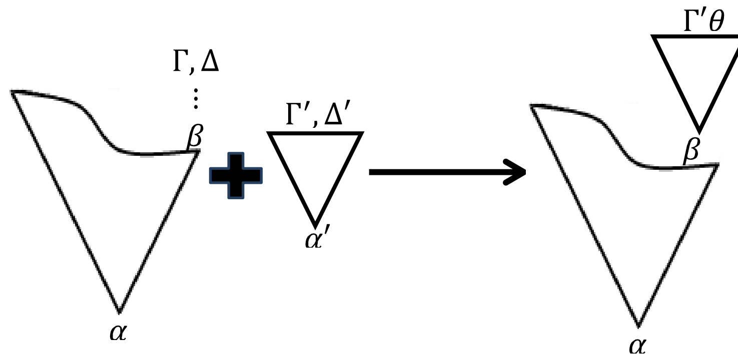

Figura 19 - Aplicación de lemas en derivaciones

La solución del problema de matching (θ, δ) para la aplicación de lemas debe ser tal que:

$\Gamma \vdash_{\Delta} \Gamma'\theta \rightarrow$ El conjunto de hipotesis del lema al que se le aplica θ debe ser derivable a partir del conjunto de hipotesis en β

$\Delta \vdash \Delta'\theta \rightarrow$ El contexto de frescura del lema aplicando θ, debe ser derivable a partir de el contexto de frescura en β

$\Delta \vdash \delta \rightarrow$ El contexto de frescura de matching debe ser derivable a partir del contexto de frescura en β

Dadas estas restricciones se define la aplicación de un lema como una regla de inferencia con condiciones laterales:

$$\frac{\gamma_1\theta \dots \gamma_n\theta}{\beta} \; (\Delta \vdash \Delta'\theta, \Delta \vdash \delta)$$

Notar que no es suficiente crear un problema de matching entre las formulas β y α'. En las hipótesis del lema puede haber metavariables que no ocurren en α' lo cual generaría una substitución incompleta, que no tiene en cuenta todas las metavariables en el lema. Por lo tanto las formulas en los conjuntos de hipótesis tanto del lema como las disponibles para β deberán intervenir en el problema. Para esto se crean equivalencias entre las fórmulas pertenecientes al producto cartesiano de los conjuntos de hipótesis.

Entonces la definición inicial del problema de matching de lemas P es:

$$P = \Delta \cup \{\alpha' \approx ?\, \beta\} \cup \{\psi \approx ?\, \phi \mid (\phi, \psi) \in \Gamma \times \Gamma'\}$$

El algoritmo para la construcción de la solución se aplica en dos fases: la primera reduce todas las condiciones de equivalencia hasta que no quede ninguna y la segunda reduce todas las condiciones de frescura.

La primera fase es computada por la aplicación sucesiva de la función transformarIgualdad sobre la lista problema. El argumento debe ser un elemento del tipo `Either NominalTerm Freshness` y el retorno será del tipo `Maybe ([(Either NominalTerm Freshness)], Sustitucion)`. Se usa un tipo Maybe para que en los casos que no se puede aplicar la transformación, se devuelva Nothing indicando que el problema P no tiene solución. En los

casos que se puede aplicar la regla, devuelve una lista de nuevas condiciones (que puede ser vacía) y una substitución.

```
transformarIgualdad (Left (Igual (Atom a) (Atom b))) =
              if a==b
             then Just ([],[])
             else Nothing
transformarIgualdad (Left (Igual (MVar perm a) (MVar perm' b))) =
              if a == b
              then
              Just ([Right (Fresh x (MVar []a)) | x <- diferenciaPerm perm perm'],[])
              else Just ([],[(a, permutar (MVar perm' b) (permInversa perm))])
transformarIgualdad (Left (Igual (Abs a t) (Abs b t'))) =
              if a == b
             then Just ([Left (Igual t t')],[])
             else Just ([Left (Igual (permutar t [(b,a),(a,b)]) t'),Right (Fresh b t)],[])
transformarIgualdad (Left (Igual (TermFormer nom ts) (TermFormer nom' ts'))) =
              if nom == nom'
             then Just (zipWith (\x y -> Left (Igual x y)) ts ts',[])
             else Nothing
transformarIgualdad (Left (Igual (MVar perm x) (t))) =
              if not . ocurre x $ t
             then Just ([],[(x, permutar t (permInversa perm))])
             else Nothing
transformarIgualdad t  = Nothing
```

Notar que el último caso de la función se aplica en los casos que ninguna de las anteriores es válida.

La segunda fase se computa con otra función que modifica las condiciones de frescura. Recibe como argumento un elemento del tipo `Either NominalTerm Freshness` y el retorno es del tipo `Maybe ([(Either NominalTerm Freshness)], FreshContext)`. Por el mismo motivo que en la fase anterior, el retorno es un tipo `Maybe`. En los casos que se aplica la transformación, se devuelve una lista de nuevas condiciones que serán exclusivamente afirmaciones de frescura (en esta fase ya no se pueden derivar condiciones de igualdad) y un contexto de frescura.

```
transformarFreshness (Right (Fresh x (Atom y))) =
              if x /= y
              then ([], [])
              else Nothing
transformarFreshness (Right (Fresh x (MVar p y))) =
              ([], [Fresh (aplicarPermInversa x (p)) (MVar [] y)])
transformarFreshness (Right (Fresh a (Abs x t)))  =
              if x == a
              then ([],[])
              else ([Right (Fresh a t)], [])
transformarFreshness (Right (Fresh a (TermFormer func ts))) =
              (map (Right . (Fresh a)) ts, [])
```

Finalmente, luego de la ejecución de ambas fases de transformación se combinan las substituciones y contextos de frescura obtenidos en cada paso y obteniéndose la solución al problema de matching.

4 Ejemplos de uso

Se agregan algunos ejemplos de derivaciones en el sistema, en código Haskell. En estas se define primero el juicio a probar dando fórmulas para la conclusión (`alfa`), las hipótesis (`gama`) y un contexto de frescura (`delta`).

Luego se aplican sucesivamente reglas de derivación que pueden ser
Backward:

- `eliminacionBottomBW`
- `eliminacionImpBW`
- `introduccionImpBW`
- `introduccionForAllBW`
- `eliminacionForAllBW`
- `igualdadBW`
- `agregarFreshBW`
- `hipotesis`

Forward:

- `eliminacionBottomFW`
- `eliminacionImpFW`
- `introduccionImpFW`
- `introduccionForAllFW`
- `eliminacionForAllFW`
- `igualdadFW`
- `agregarFreshFW`

Para cada regla se indica una lista de argumentos que depende de la especificación [ver Tabla], el número de prueba a la que aplica (0 para la prueba principal y n>=1 para las pruebas secundarias, el número de nodo donde se aplica la regla (esencial para las pruebas backward, el valor se descarta para las forward)

La función `iniciarDerivacion` crea la estructura de derivación como se detalla en la descripción del sistema.

<u>Ejemplo 1</u>

```
let alfa = Imp (MetaVar 'P') (Imp (MetaVar 'Q') (MetaVar 'P'))
let gama  = []
let delta = []

let derivacion = iniciarDerivacion alfa gama delta

putStrLn . show $ (
                    (derivar hipotesis [] 0 3) .
                    (derivar introduccionImpBW [] 0 2) .
                    (derivar introduccionImpBW [] 0 1)
                    $ derivacion)
```

Salida>

```
Prueba principal:
--------------------------------------------------------------
(IImpBW)  -  Imp (MetaVar P) (Imp (MetaVar Q) (MetaVar P))  {1}
|
`- (IImpBW)  -  Imp (MetaVar Q) (MetaVar P)  {2}
   |
   `- (Hipotesis MetaVar P)  -  MetaVar P  {3}
```

Ejemplo 2

En este ejemplo de introduce un contexto de frescura y la aplicación de la regla de reescritura o que genera una condición lateral que se verifica ejecutando la función `verificarCondicion` e indicándosele el número de prueba y el número de nodo.
En negrita se muestran las condiciones laterales que aparecen.

```
let alfa = Sust (MetaVar 'P') 'a' (Var 'x')
          let gama  = [MetaVar 'P']
          let delta = [Fresh 'a' (MVar [] 'P')]

          let derivacion = iniciarDerivacion alfa gama delta

          putStrLn . show $  ((verificarSideCondition 0 1) .
                              (derivar hipotesis [] 0 2) .
                              (derivar rescrituraBW [Left (MetaVar 'P')] 0 1)
                              $ derivacion)
```

Salida>

Prueba principal:

(IgualdadBW) - Sust (MetaVar P) a (Var x) {1} **([a#P |- Sust([a]P, x) = P (True)])**
|
`- (Hipotesis MetaVar P) - MetaVar P {2}

Ejemplo 3

En este caso es necesario agregar una afirmación al contexto de frescura. La regla agregarFreshBW introduce un átomo fresco en todas las metavariables que ocurren en la derivación.

```
let alfa = ForAll 'a' (MetaVar 'P')
let gama  = [MetaVar 'P', MetaVar 'Q']
let delta = [Fresh 'a' (MVar [] 'P')]

let derivacion = iniciarDerivacion alfa gama delta

putStrLn . show $ (
        (verificarCondicion 0 4) .
        (verificarCondicion 0 3) .
        (verificarCondicion 0 2) .
        (derivar hipotesis [] 0 5) .
        (derivar rescrituraBW [Left (MetaVar 'P')] 0 4) .
        (derivar introduccionForAllBW [] 0 3) .
        (derivar rescrituraBW [Left (ForAll 'b' (Sust (MetaVar 'P') 'a' (Var 'b')))] 0 2) .
        (derivar agregarFreshBW [Right (Var 'b')] 0 1)
        $ derivacion)
```

Prueba principal:

(Fr) - ForAll 'a' (MetaVar 'P') {1}
|
`- (IgualdadBW) - ForAll 'a' (MetaVar 'P') {2}
([a#P, b#P, b#Q |- ForAll([a]P) = ForAll([b]Sust([a]P, b)) (True)])
|
`- (IForAllBW) - ForAll 'b' (Sust (MetaVar 'P') 'a' (Var 'b')) {3}
([a#P, b#P, b#Q |- b#P (True),
a#P, b#P, b#Q |- b#Q (True)])
|
`- (IgualdadBW) - Sust (MetaVar 'P') 'a' (Var 'b') {4}
([**a#P, b#P, b#Q |- Sust([a]P, b) = P (True**)])
|
`- (Hipotesis MetaVar 'P') - MetaVar 'P' {5}

Ejemplo 4

En este caso se inicia una prueba secundaria que luego se une con la prueba primaria con el comando `unirPrueba`

```
let alfa = ForAll 'a' (Sust (MetaVar 'P') 'b' (Var 'a'))
let gama  = [ForAll 'b' (ForAll 'a' (MetaVar 'P'))]
let delta = []

let derivacion = iniciarDerivacion alfa gama delta

putStrLn . show $ (
       (unionPrueba 1 4) .
       (derivar eliminacionForAllFW [Right (Var 'c')] 1 0) .
       (derivar rescrituraFW [Left (ForAll 'a' (Sust (MetaVar 'P') 'b' (Var 'c')))] 1 0) .
       (derivar eliminacionForAllFW [Right (Var 'c')] 1 0) .
       (supuesto (ForAll 'b' (ForAll 'a' (MetaVar 'P')))) .
       (derivar introduccionForAllBW [] 0 3) .
       (derivar rescrituraBW
        [Left (ForAll 'c' (Sust (Sust (MetaVar 'P') 'b' (Var 'c')) 'a' (Var 'c')))] 0 2) .
       (derivar agregarFreshBW [Right (Var 'c')] 0 1)
       $ derivacion)
```

Prueba principal:

(Fr) - ForAll 'a' (Sust (MetaVar 'P') 'b' (Var 'a')) {1}
|
`- (IgualdadBW) - ForAll 'a' (Sust (MetaVar 'P') 'b' (Var 'a')) {2}
([c#P |- ForAll([a]Sust([b]P, a)) = ForAll([c]Sust([a]Sust([b]P, c), c)) (False)])
|
`- (IForAllBW) - ForAll 'c' (Sust (Sust (MetaVar 'P') 'b' (Var 'c')) 'a' (Var 'c')) {3}
([c#P |- Right c#ForAll([b]ForAll([a]P)) (False)])
|
`- (EForAllFW) - Sust (Sust (MetaVar 'P') 'b' (Var 'c')) 'a' (Var 'c') {4}
|
`- (Igualdad) - ForAll 'a' (Sust (MetaVar 'P') 'b' (Var 'c')) {5}
([c#P |- Sust([b]ForAll([a]P), c) = ForAll([a]Sust([b]P, c)) (False)])
|
`- (EForAllFW) - Sust (ForAll 'a' (MetaVar 'P')) 'b' (Var 'c') {6}
|

```
            `- (Supuesto)  -  ForAll 'b' (ForAll 'a' (MetaVar 'P'))  {7}
```

Pruebas secundarias:

```
------------------------------------------------------------
Subprueba 1:
(EForAllFW)  -  Sust (Sust (MetaVar 'P') 'b' (Var 'c')) 'a' (Var 'c')  {1}
|
`- (Igualdad)  -  ForAll 'a' (Sust (MetaVar 'P') 'b' (Var 'c'))  {2}
      ([ |- Sust([b]ForAll([a]P), c) = ForAll([a]Sust([b]P, c))  (False)])
  |
  `- (EForAllFW)  -  Sust (ForAll 'a' (MetaVar 'P')) 'b' (Var 'c')  {3}
    |
    `- (Supuesto)  -  ForAll 'b' (ForAll 'a' (MetaVar 'P'))  {4}
```

---

5 Conclusiones

Nuestro problema era construir un asistente de demostraciones, con facilidades didácticas, para el sistema de deducción natural de la lógica de primer orden. Basado en una implementación previa, las facilidades que se definen como esenciales son:

- Manejo de árboles como representación gráfica de las derivaciones.
- Construcción de derivaciones con razonamientos forward y backward.
- Implementación automática de procedimientos triviales, pero no de la ejecución de pasos de inferencia que aseguren que el control de la derivación lo tiene el estudiante.
- Uso de lemas = reuso de teoremas.

En la práctica, las derivaciones que se efectúan en los cursos y libros de texto de lógica son esquemáticas, es decir, operan a nivel de meta-lenguaje. Manejan, en consecuencia, cuatro conceptos que no aparecen en el lenguaje objeto de primer orden:

- Meta-variables
- Condiciones, imponiendo que ciertas variables no aparezcan libres en ciertas fórmulas
- Substituciones explícitas (que deben evitar captura de variables libres)
- Igualdad sintáctica que tiene en cuenta la equivalencia alfa y la substitución explícita.

Por tanto, nuestro asistente, además de presentar las funcionalidades didácticas mencionadas debería permitir este tipo de derivaciones.

Nuestra contribución ha sido:

- Elegir la Lógica de orden uno y medio como el lenguaje idóneo para formalizar juicios y derivaciones esquemáticas.
- Adaptar el cálculo de secuentes de la presentación original en [8] a un cálculo de deducción natural equivalente.
- Diseñar el asistente completo para este sistema, con las facilidades ya mencionadas.
- Implementar algoritmos de decisión para la igualdad de expresiones módulo alfa equivalencia y substitución explícita.
- Implementar y justificar el uso de lemas instanciables.
- Implementar algoritmos de matching que permitan la aplicación de esta facilidad.
- Desarrollar un prototipo que presenta al asistente embebido en el lenguaje funcional Haskell.

El trabajo que resta por hacer es:

- Proveer al prototipo de una interfaz gráfica comparable al de la versión anterior. Muchas de las ventajas que presentaba, vienen de la mano con la posibilidad de manipulación de pruebas en el entorno gráfico.
- Explorar la alternativa de un asistente orientado al uso profesional. En este caso sería de interés una mayor automatización que en el caso didáctico, así como la alternativa (hasta ahora sólo experimental) de presentar el asistente embebido en un lenguaje funcional.

6 Bibliografía

[1] D. Gentzen. “Untersuchungen über das logische Schliessen”. Traducido como “Investigations into logical deduction”; Mathematische Zeitschrift 39, 1935. Traducido en M. Szabo, editor. Collected Papers of Gerhard Gentzen. North Holland (1969).

[2] D. Van Dalen. “Logic and Structure”. Springer (2008).

[3] J. Pais, A. Tasistro, "Design and implementation of a proof assistant for natural deduction," *Computers in Education (SIIE), 2012 International Symposium on* , vol., no., pp.1,6, 29-31 Oct. 2012

[4] J. Pais, A. Tasistro, “Proof Assistant Based on Didactic Considerations”. J.UCS – Journal of Universal Computer Science (A ser publicado)

[5] M. Huth, M. Ryan. “Logic in Computer Science: Modeling and Reasoning About Systems”; Cambridge University Press (2004).

[6] PANDORA (2013, Ago) [Online] Disponible: http://www.doc.ic.ac.uk/pandora/runpandora.html

[7] ADN (2013, Ago) [Online] Disponible: http://www.dccia.ua.es/logica/ADN/indexen.htm

[8]PANDA (2013, Ago) [Online] Disponible: http://www.irit.fr/panda/

[9]ProofBuilder (2013, Ago) [Online] Disponible: http://www.cis.gvsu.edu/~mcguire/ProofBuilder/

[10]DC Proof (2013, Ago) [Online] Disponible: http://www.dcproof.com/

[11] JAPE (2013, Ago) [Online] Disponible: http://japeforall.org.uk

[12] H. van Ditmarsch. “User interfaces in natural deduction programs”, in Backhouse, R. C. (Ed.) Proceedings of the 4th International Workshop on User Interface Design for Theorem Proving Systems (1998).

[13] K. Broda, J. Ma, G. Sinnadurai y A. Summers. (2007). “Pandora: A reasoning toolbox using natural deduction style”. Logic Journal of IGPL, 15(4), 293-304.

[14] F. Gallego, F. Llorens, S. Mira y R. Rizo. “An assistant for learning logic deduction”

[15] O. Gasquet, F. Schwarzentruber y M. Strecker (2011). “PANDA: a proof assistant in natural deduction for all. a Gentzen style proof assistant for undergraduate students”. En *Tools for Teaching Logic* (pp. 85-92). Springer Berlin Heidelberg.

[16] H. McGuire. (2007, September). “Proofbuilder, an interactive prover for students, with extensive capabilities”. En *6th International Workshop on First-Order Theorem Proving* FTP 2007 Septiembre 12–13, 2007 University of Liverpool (p. 51).

[17] R. Bornat & B. Sufrin (1999). “A minimal graphical user interface for the Jape proof calculator”. Formal Aspects of Computing, 11(3), 244-271.

[18] R. Bornat y B. Sufrin. “Displaying sequent-calculus proofs in natural-deduction style: experience with the Jape proof calculator”; International Workshop on Proof Transformation and Presentation, Dagstuhl (1997).

[19] M. J. Gabbay, A. Mathijssen. “One-and-a-halfth-order logic”. Journal of Logic and Computation, 18, 4 (2008), 521-562.

[20] M. J. Gabbay and A. Mathijssen. “Nominal algebra”. Reporte técnico, Heriott-Watt, 2007.

[21] M. J. Gabbay and A. M. Pitts. “A new approach to abstract syntax involving binders”. En *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, 1999.

[22] M. J. Gabbay and A. M. Pitts. “A new approach to abstract syntax with variable binding.
Formal Aspects of Computing”, 13(3–5):341–363, 2001.

[23] M. J. Gabbay and A. Mathijssen. “A formal calculus for informal equality with binding”. En *WoLLIC’07: 14th Workshop on Logic, Language, Information and Computation*, volume
4576 of LNCS, páginas 162–176, 2007.

[23] C. Urban, A. M. Pitts, M. J. Gabbay, “Nominal unification”. M. Baaz (Ed.), Computer Science Logic and 8th Kurt Godel Colloquium (CSL’03 & KGC), Vienna, Austria. Proceedings, Vol. 2803 de Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 513–527.

[24] C. Calves y M. Fernández. “Implementing nominal unification”. Proceedings of TERMGRAPH’06, 3rd International Workshop on Term Graph Rewriting, ETAPS 2006, Vienna, Electronic Notes in Computer Science. Elsevier, 2006.

[25] M. Fernández y M. J. Gabbay. “Nominal Rewriting”. Information and Computation 205:917–965, 2007. Elsevier.

[26] C. Christophe y M. Fernández. "Implementing nominal unification."Electronic Notes in Theoretical Computer Science 176.1 (2007): 25-37.

[27] C. Christophe y M. Fernández. "Nominal matching and alpha-equivalence." Logic, Language, Information and Computation. Springer Berlin Heidelberg, 2008. 111-122.

[28] M. J. Gabbay, A. Mathijssen. “Capture-avoiding substitution as a nominal algebra”. *ICTAC’2006: 3rd Int’l Colloquium on Theoretical Aspects of Computing*, volume 4281 of LNCS, páginas 198-212, 2006.

7 Anexo I

7.1 Algebra Nominal y Derivabilidad

Las pruebas de igualdad de fórmulas en las condiciones laterales de nuestras derivaciones, se basan en una axiomatización de la substitución expresados en algebra nominal llamada SUB. Se dará una idea general de esta teoría, necesario para mostrar los algoritmos implementados en el sistema. Los detalles formales sobre este marco algebraico pueden encontrarse en [28].

$$\begin{array}{lrl}
(var \mapsto) & & \vdash a[a \mapsto T] = T \\
(\# \mapsto) & a\#X & \vdash X[a \mapsto T] = X \\
(f \mapsto) & & \vdash f(X_1 \ldots X_n)[a \mapsto T] = f(X_1[a \mapsto T], \ldots, X_n[a \mapsto T]) \\
(abs \mapsto) & b\#T & \vdash ([b]U)[a \mapsto T] = [b](U[a \mapsto T]) \\
(ren \mapsto) & b\#T & \vdash X[a \mapsto b] = (b\ a) \cdot X \\
(\eta \mapsto) & a\#X & \vdash [a]sub(X, a) = X
\end{array}$$

Imagen 1 - Axiomas de la teoría SUB

El objetivo será definir un mecanismo para derivar automáticamente juicios de igualdad en SUB basado en realizar varias transformaciones sobre los términos nominales hasta lograr una forma normal cuya igualdad es decidible en una teoría con reglas dirigidas por sintaxis llamada CORE.

7.1.1 La Teoría CORE

Se define primero una teoría de igualdad módulo α-equivalencia sin axiomas conocida como CORE.
En [16] se proponen dos conjuntos de reglas de derivación para la igualdad nominal en CORE. La ventaja de uno de esos conjuntos es que sus reglas son dirigidas por la sintaxis y disjuntas (para cada caso hay una y solo una regla para aplicar) permiten automatizar la construcción de derivaciones.

Sea una tupla $(t \approx_\Delta u)$ donde $t, u \in \mathbb{F}$ y Δ es un contexto de frescura. Luego se definen las reglas dirigidas por sintaxis en [Fig.7] para determinar la igualdad de *t* y *u*.

$$(\mathrm{Ax})\frac{}{a \approx_\Delta a} \qquad (\mathrm{Ds})\frac{\Delta \vdash ds(\pi', \pi)\#X}{\pi \cdot \approx_\Delta \pi' \cdot X} \qquad (\mathrm{F})\frac{t_1 \approx_\Delta u_1 \cdots t_n \approx_\Delta u_n}{f(t_1, \cdots, t_n) \approx_\Delta f(u_1, \cdots, u_n)}$$

$$(\mathrm{Absaa})\frac{t \approx_\Delta u}{[a]t \approx_\Delta [a]u} \qquad (\mathrm{Absab})\frac{(ba) \cdot \approx_\Delta u \quad \Delta \vdash b\#t}{[a]t \approx_\Delta [b]u}$$

Figura 2- Reglas de derivación dirigidas por sintaxis

En estas reglas, $ds(\pi, \pi') = \{a | \pi(a) \neq \pi'(a)\}$ y $\Delta \vdash ds(\pi, \pi')\#t$ es un conjunto de derivaciones $\Delta \vdash a\#t$ para cada $a \in ds(\pi, \pi')$

7.1.1.1 Substitución

Se define una substitución σ como una función de metavariables a términos nominales tal que $\sigma(X) \equiv Id \cdot X$ para la mayoría de las metavariables. La aplicación de una substitución se define como:

$\#_1\#_2 :: TN \rightarrow (\mathbb{M} \rightarrow TN) \rightarrow TN$
$a\sigma = a$
$(\pi \cdot X)\sigma = \pi \cdot (X\sigma)$
$([a]t)\sigma = [a](t\sigma)$
$f(t_1, \dots, t_n)\sigma = f(t_1\sigma, \dots, t_n\sigma)$

Una substitución se propaga por la estructura del término nominal hasta que alcanza un átomo y desaparece, o una metavariable moderada y la aplica la substitución reemplazándola por un término concreto.

7.1.1.2 Términos Ground

Los términos ground son aquellos que no mencionan metavariables o substituciones. La importancia de este tipo particular de términos está en que permitirá conectar el concepto de substitución en términos cualquiera con la substitución con evitación de captura en sintaxis.

Definición del conjunto inductivo de términos ground (GT):

$$\frac{}{a \in GT}\ (a \in \mathbb{A}) \qquad \frac{g \in GT}{[a]g \in GT}\ (a \in \mathbb{A}) \qquad \frac{g_1 \in GT, \dots, g_n \in GT}{f(g_1, \dots, g_n) \in GT}$$

Se define la función “átomos libres de” como
$fa(g) :: GT \rightarrow \mathbb{A}^2$
$fa(a) = \{a\}$
$fa([a]g) = fa(g)/\{a\}$
$$fa\big(f(g_1, \dots, g_n)\big) = \bigcup_{1 \le i \le n} fa(g_i)$$

Luego se define la substitución en términos ground:
$_[_/_] :: GT \rightarrow GT \rightarrow \mathbb{A} \rightarrow GT$
$a[h/a] = h$
$b[h/a] = b$
$([a]g)[h/a] = [a]g$
$([b]g)[h/a] = [b](g[h/a]) \qquad si\ b \notin fa(h)$
$([b]g)[h/a] = [c](g[c/b][h/a]) \qquad si\ b \in fa(h)\ y\ c\ fresca$
$f(g_1, \dots, g_n)[h/a] = f(g_1[h/a], \dots, g_n[h/a])$

Una variable c es fresca si $c \notin fa(g) \cup fa(h) \cup \{a, b\}$

Puede probarse que bajo la igualdad de la teoría CORE, se puede probar que dos términos son iguales si y solo si son α-equivalentes, o sea $\vdash_{CORE} g = h \Leftrightarrow g =_\alpha h$

7.1.1.3 Términos Cerrados

Se consideran términos cerrados (TC) aquellos en los que no ocurren metavariables. Para estos se define una función de traducción que los transforma en términos ground:

$\#_1{}^{\downarrow} :: TC \rightarrow TG$
$a^{\downarrow} = a$
$([a]t)^{\downarrow} = [a](t^{\downarrow})$
$f(t_1, \dots, t_n)^{\downarrow} = f(t_1^{\downarrow}, \dots, t_n^{\downarrow})$
$sub(t,u)^{\downarrow} = g[u^{\downarrow}/a] \qquad con\ [a]g\ = t^{\downarrow}$

Notar que $g[a \mapsto h]$es lo mismo que $sub([a]g, h)$

7.1.1.4 Términos Abiertos

Los términos abiertos son los que contienen por lo menos una metavariable. Se define una substitución especial que transforma a los términos posiblemente abiertos que mencionan a metavariables a términos cerrados.

Dado un contexto Δ y fórmulas *t* y *u*, se define un conjunto $\mathbb{F}'$ de términos nominales con nuevos constructores. Se define $\mathcal{A}$ como el conjunto de átomos que aparecen (sintácticamente) en Δ, t, u y $\mathcal{X}$como el conjunto de las metavariables que aparecen en Δ, t, u.

Luego para cada $X \in \mathcal{X}$ elegir:

- Un subconjunto de los átomos de $\mathcal{A}, \{a_{X1}, \dots, a_{Xn}\}$ tal que $a\#X \ \notin \Delta$
- Un nuevo constructor $d_X :: \ \mathbb{T}^n \rightarrow \mathbb{T}$ tal que d_X no existe como constructor

Se define una substitución ς:
$\varsigma(X) = d_X(a_{X1}, \dots, a_{Xn}) \qquad (X \in \mathcal{X})$
$\varsigma(Y) = Y \qquad (Y \notin \mathcal{X})$
Con esta substitución, $\varsigma(X)$ es un término cerrado que menciona sin abstracción aquellos átomos que Δ no puede probar frescos para X.

7.1.2 Decidibilidad en SUB

Con las definiciones anteriores se puede definir un algoritmo para decidir si dos términos son iguales en la teoría SUB.

Dado un juicio del tipo $\Delta \vdash_{SUB} t = u$ los pasos para construir una derivación son:

- Construir la substitución ς para transformar los términos posiblemente abiertos *t* y *u* a los términos cerrados $t\varsigma$ y $u\varsigma$
- Evaluar $t\varsigma^{\downarrow}$ y $u\varsigma^{\downarrow}$, obteniéndose términos ground
- Construir la derivación $\vdash_{CORE} t\varsigma^{\downarrow} \approx_{\emptyset} u\varsigma^{\downarrow}$ siguiendo las reglas dirigidas por sintaxis.

Este algoritmo es el que implementará el sistema a la hora de automatizar las pruebas de condiciones laterales que involucran igualdad de términos nominales.

Por ejemplo, se quiere verificar que $a\#X \vdash_{SUB} X[a \mapsto b] = X$.
El primer paso consiste en construir la substitución ς con:

- $\mathcal{A} = \{a, b\}$
- $\mathcal{X} = \{X\}$

Entonces $\varsigma = \{X \mapsto d_X(b)\}$ y $(X[a \mapsto b])\varsigma =$
Se aplica la substitución a los términos de la igualdad:

- $(X[a \mapsto b])\varsigma = d_X(b)[a \mapsto b]$
- $(X)\varsigma = d_X(b)$

Luego se aplica la función de traducción de términos cerrados a términos ground:

- $(d_X(b)[a \mapsto b])^{\downarrow} = d_X(b)[b/a] = d_X(b)$
- $(d_X(b))^{\downarrow} = d_X(b)$

Finalmente se construye la derivación para $\vdash_{CORE} d_X(b) \approx_{\emptyset} d_X(b)$

$$(F)\,\frac{(Ax)\,\overline{b \approx_{\emptyset} b}}{d_X(b) \approx_{\emptyset} d_X(b)}$$

Como existe una derivación para $\vdash_{CORE} d_X(b) \approx_{\emptyset} d_X(b)$, entonces es cierto que $a\#X \vdash_{SUB} X[a \mapsto b] = X$

7.1.3 Algoritmo de Derivación Automática de Condiciones Laterales

Como se mencionó anteriormente, se usarán las estructuras de datos de derivación pero instanciadas sobre un tipo suma de términos nominales y afirmaciones de frescura: `Either NominalTerm Freshness`

Esto se debe a que la derivación de juicios de igualdad de términos puede requerir la derivación de afirmaciones de frescura. Algunas reglas de inferencia sobre la igualdad de términos nominales derivan afirmaciones de frescura (ver reglas Ds y Absab en [fig]). Entonces, la misma instancia del tipo `Derivacion` sirve para juicios de frescura y juicios de igualdad.

Se define una única regla de inferencia backward dirigida por sintaxis, i.e. con distintos casos en función de la sintaxis de la fórmula sobre la que se aplica. El retorno de la función coincide con el esperado para una regla `Backward` del tipo `ReglaDerivacion.` A diferencia de las reglas para la lógica de orden 1½, esta no depende del contexto de frescura ni de otras fórmulas, por lo que serán omitidos en la especificación de los casos. El conjunto de hipótesis está compuesto por afirmaciones de frescura y se mantiene constante a lo largo de la derivación (no hay casos que agreguen hipótesis)

Pattern	Definición
$a\#X$	$(\text{"}\#ab\text{"}, [\,], [\,], [\,], [\,])$
$a\#\pi \cdot X$	$(\text{"}\#X\text{"}, [\pi^{-1}(a)\#X], [\,], [\,], [\,])$
$a\#[a]X$	$(\text{"}\#[]a\text{"}, [\,], [\,], [\,], [\,])$
$a\#[b]X$	$(\text{"}\#[]b\text{"}, [a\#t], [\,], [\,], [\,])$
$a\#f(t_1, \dots, t_n)$	$(\text{"}\#f\text{"}, [a\#t_1, \dots, a\#t_n], [\,], [\,], [\,])$

$a \approx a$	("Ax", [ ], [ ], [ ], [ ])
$\pi \cdot X \approx \pi' \cdot X$	("Ds", [$ds(\pi, \pi')\#X$], [ ], [ ], [ ])
$f(t_1, \dots, t_n) \approx f(u_1, \dots, u_n)$	(F, [$t_1 \approx u_1, \dots, t_n \approx u_n$], [ ], [ ], [ ])
$[a]t \approx [a]u$	("Abs_aa", [t ≈ u], [ ], [ ], [ ])
$[a]t \approx [b]u$	("Abs_ab", [$(b\ a) \cdot t \approx u, b\#t)$], [ ], [ ], [ ])
$x\#X$	("Hipotesis", [ ], [ ], [ ], [ ]) $si\ (x\#X) \in hips$

Tabla 10 - Especificación de reglas de derivación de condiciones laterales

Se aplica esta regla sucesivamente mientras haya en la prueba hojas no probadas. Si hay un caso que no entra en los formulados en la tabla anterior, la prueba falla y el juicio no es derivable.

Las derivaciones de juicios de frescura son de la forma $\Delta \vdash a\#t$ donde Δ es un contexto de frescura, a es un átomo y t un término nominal.
Si bien en el sistema la condición lateral figura como un juicio de frescura sobre una formula lógica, internamente la derivación se resuelve sobre una traducción de la formula a un término nominal.

Las derivaciones de igualdad tienen la forma $\Delta \vdash_{SUB} t = u$ donde Δ es un contexto de frescura y *t, u* son términos nominales.
La derivación automática en SUB se logra aplicando el algoritmo mencionado en () y construyendo una derivación para $\vdash_{CORE}\ t\varsigma^{\downarrow} = u\varsigma^{\downarrow}$

8 Anexo II

8.1 Manual de usuario

1. Iniciar una derivación.

Dado un juicio de la forma $\Gamma \vdash_{\Delta} \alpha$, utilizar esta función para iniciar la construcción de su derivación.

```
iniciarDerivacion :: a -> [a] -> FreshContext -> (Derivacion a)
```

Que recibe como argumentos:

- Una fórmula α, la conclusión del juicio
- Una lista de fórmulas Γ, la lista de hipótesis del juicio.
- Una lista de afirmaciones de frescura Δ, el contexto de frescura del juicio.

El contexto de frescura se compone de afirmaciones sobre términos nominales, no sobre fórmulas lógicas, i.e (Fresh 'a' (MVar [] 'X') para *a#X*

Ejemplo 1:

```
let alfa = ForAll 'a' (MetaVar 'Q')
let gama  = [(MetaVar 'P'), ForAll 'a' (Imp (MetaVar 'P') (MetaVar 'Q'))]
let delta = [Fresh 'a' (MVar [] 'P')]

iniciarDerivacion alfa gama delta
```

Ejemplo 2: Γ y Δ pueden ser vacíos.

```
let alfa = Imp (ForAll 'x' (MetaVar 'A')) (ForAll 'x' (MetaVar 'B'))
let gama  = [ForAll 'x' (Imp (MetaVar 'A') (MetaVar 'B'))]
let delta = []

iniciarDerivacion alfa gama delta
```

2. Aplicar reglas de inferencia

2.1 Reglas sobre conectivas lógicas

Las reglas de inferencia se aplican sobre derivaciones mediante la función `derivar`. Para aplicarla se debe indicar:

- El nombre de la regla de inferencia
- Una lista de argumentos formado por elementos tipo Form (fórmulas) o elementos tipo Obj (términos del lenguaje de objetos). Notar que en la función, se espera recibir un elemento del tipo `Either Form Obj`, por lo tanto se debe construir el tipo correcto para cada argumento (Left para las fórmulas y Right para los términos)
- Un entero que representa a la prueba sobre la que se aplica la regla. El 0 corresponde a la prueba principal y n>=1 a las pruebas secundarias.
- Un entero que representa al nodo sobre el que se aplica la regla. Los nodos están numerados por niveles. En el caso de que la regla se aplique sobre una prueba secundaria, este valor es ignorado ya que la regla de inferencia se aplica siempre sobre la conclusión.

Los nombres de las reglas de inferencia sobre conectivas son:

Reglas Backward	eliminacionBottomFW eliminacionImpBW introduccionImpBW introduccionForAllBW eliminacionForAllBW
Reglas Forward	eliminacionBottomFW eliminacionImpFW introduccionImpFW introduccionForAllFW eliminacionForAllFW

Ejemplo1: Aplicar la introducción de la implicación backwards sobre el nodo 1 de la prueba principal.

```
let alfa = Imp (ForAll 'x' (MetaVar 'A')) (ForAll 'x' (MetaVar 'B'))
let gama  = [ForAll 'x' (Imp (MetaVar 'A') (MetaVar 'B'))]
let delta = []

let derivacion = iniciarDerivacion alfa gama delta

derivar introduccionImpBW [] 0 1 derivacion
```

Ejemplo 2: Aplicar la eliminación del para todo forward. Se le pasa como argumento la variable a para construir la substitución derivada. Continúa sobre el ejemplo anterior.

```
(derivar eliminacionForAllFW [Right (Var 'a')] 1 1) .
(supuesto (ForAll 'x' (MetaVar 'A')) .
(derivar introduccionImpBW [] 0 1) $
derivacion
```

Nota: La función supuesto se ve más adelante.

2.2 Otras reglas

Se definen reglas que no dependen de la conectiva principal de la fórmula sobre la que se aplican.

Hipótesis

Prueba un goal de la derivación usando una hipótesis disponible. Para aplicar la regla se debe indicar la prueba y el nodo afectados, no es necesario indicar argumentos extra. Esta regla solo se usa en sentido backward.

Ejemplo: Se aplica la regla de hipótesis sobre la fórmula 2 de la prueba principal.

```
let alfa = Sust (MetaVar 'P') 'a' (Var 'x')
let gama  = [MetaVar 'P']
let delta = [Fresh 'a' (MVar [] 'P')]

let derivacion = iniciarDerivacion alfa gama delta

(derivar hipotesis [] 0 2) .
(derivar rescrituraBW [Left (MetaVar 'P')] 0 1)
$ derivacion)
```

Reescritura
Permite reescribir un término por otro. Genera como condición lateral la prueba de la igualdad de la fórmula original f y la nueva fórmula f' según la teoría SUB, bajo el contexto de frescura disponible: $\Delta \vdash_{\mathrm{SUB}} t = t'$

Para aplicar la regla se debe indicar el valor de la nueva fórmula a derivar.
La condición lateral queda pendiente y se debe validar para dar por finalizada la prueba.

Ejemplo: Se quiere probar que P $\vdash_{\mathrm{a\#P}} P[a \mapsto x]$. Se inicia la prueba y en el primer paso se reescribe la conclusión de $P[x \mapsto a]$ a P. Se genera la condición lateral de probar la igualdad de estos términos bajo el contexto $\{a\#P\}$

```
let alfa = Sust (MetaVar 'P') 'a' (Var 'x')
let gama  = [MetaVar 'P']
let delta = [Fresh 'a' (MVar [] 'P')]

let derivacion = iniciarDerivacion alfa gama delta

(derivar hipotesis [] 0 2) .
(derivar rescrituraBW [Left (MetaVar 'P')] 0 1)
$ derivacion)
```

Supuesto
Genera una nueva prueba secundaria a partir de una hipótesis del juicio. Sobre esta pruebas secundarias se pueden aplicar reglas forward.
A diferencia de las anteriores, esta regla no se aplica con la función `derivar`

Ejemplo: Se crean dos pruebas secundarias a partir de las dos hipótesis del teorema.

```
let alfa = ForAll 'a' (MetaVar 'Q')
let gama  = [(MetaVar 'P'), ForAll 'a' (Imp (MetaVar 'P') (MetaVar 'Q'))]
let delta = [Fresh 'a' (MVar [] 'P')]

let derivacion = iniciarDerivacion alfa gama delta

(supuesto (MetaVar 'P')) .
(supuesto (ForAll 'a' (Imp (MetaVar 'P') (MetaVar 'Q'))) .
derivacion)
```

3. Aplicar lemas

Permite utilizar un juicio como una regla de inferencia. Se espera que la fórmula donde se quiere aplicar el lema sea una instancia de la conclusión del mismo, i.e. que exista una substitución que transforme a las metavariables de la conclusión del lema en expresiones concretas de forma tal que se iguale con la fórmula donde aplica la regla.
Tener en cuenta que la relación de igualdad aquí se define como "igualdad sintáctica", lo que llamaremos igualdad simple.

Para aplicar un lema se deben indicar:

- Una fórmula α', la conclusión del lema
- Una lista de fórmulas Γ', la lista de hipótesis del lema.
- Una lista de afirmaciones de frescura Δ', el contexto de frescura del lema.
- El número del nodo sobre el árbol principal donde se quiera aplicar el lema.

Luego, se construye una substitución simple que se aplica sobre la conclusión y las hipótesis del lema y luego agregar a estas últimas como obligaciones de prueba en la derivación.

Ejemplo: Se aplica el lema formado por alfa', gama' y delta' en el nodo 1 de la prueba principal.

```
let alfa = ForAll 'x' (Pred "P" [Var 'x'])
let gama =
       [(Imp (ForAll 'x' (Pred "Q" [Var 'x', Var 'y'])) (ForAll 'x' (Pred "P" [Var
'x']))),
        (ForAll 'x' (Pred "Q" [Var 'x', Var 'y']))]
let delta = []
let alfa' = ForAll 'x' (MetaVar 'D')
let gama' = [Imp (ForAll 'x' (MetaVar 'G')) (ForAll 'x' (MetaVar 'D')),
            ForAll 'x' (MetaVar 'G')]
let delta' = []

let derivacion = iniciarDerivacion alfa gama delta

aplicarLema alfa' gama' delta' 1 deriv
```