UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE CIENCIAS MATEMÁTICAS

TESIS DOCTORAL

Una lógica computacional con polimorfismo y recursion, y un sistema de deducción automatica basado en ella

MEMORIA PARA OPTAR AL GRADO DE DOCTOR
PRESENTADA POR

Susana Nieva Soto

DIRECTOR:

Javier Leach Albert

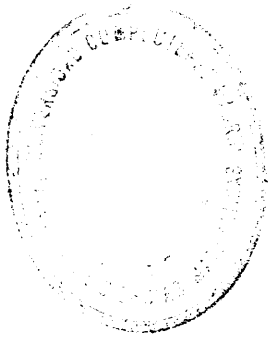
Madrid, 2015

UNIVERSIDAD COMPLUTENSE DE MADRID

Facultad de Ciencias Matemáticas

Departamento de Informática y Automática

UNA LOGICA COMPUTACIONAL CON POLIMORFISMO Y RECURSION Y UN SISTEMA DE DEDUCCION AUTOMATICA BASADO EN ELLA



Susana Nieva Soto

Madrid, 1992

Colección Tesis Doctorales. N.º 147/92

ISBN X-53-00602S-5

Edita e imprime la Editorial de la Universidad
Complutense de Madrid. Servicio de Reprografía.
Escuela de Estomatología. Ciudad Universitaria.
Madrid, 1992.
Ricoh 3700
Depósito Legal: M-12232-1992

UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE CIENCIAS MATEMATICAS
DEPARTAMENTO DE INFORMATICA Y AUTOMATICA

UNA LOGICA COMPUTACIONAL CON POLIMORFISMO Y RECURSION Y UN SISTEMA DE DEDUCCION AUTOMATICA BASADO EN ELLA

Memoria presentada por
SUSANA NIEVA SOTO

Para la obtención del
GRADO DE DOCTOR

Bajo la dirección de
JAVIER LEACH ALBERT

A mis padres

AGRADECIMIENTOS

En primer lugar agradezco a mi director, Javier Leach, su constante preocupación por los avances de este trabajo, sus consejos para la consecución del mismo y su apoyo incondicional.

Quiero expresar mi reconocimiento a Mario Rodríguez Artalejo que con sus ideas, sus explicaciones y sus correcciones ayudó a que se hiciera la luz cuando se planteaban las dificultades.

No puedo olvidar los detalles tanto técnicos como humanos que Antonio Gavilanes, Ana Gil y Teresa Hortalá han tenido conmigo; de ellos he aprendido mucho.

Quiero también agradecer la colaboración de Paco López por su participación en la programación de la primera versión del sistema. De manera especial, mi gratitud a Santos Pérez que con su trabajo ha hecho posible el funcionamiento del demostrador actual.

A todos los anteriores, a mi familia y a mis amigos que me han ayudado, animado y aguantado, muchas gracias.

INDICE

INTRODUCCION

Por demostración automática de teoremas, ATP (*Automated Theorem Proving*), [Lov-83], entendemos el uso del ordenador para demostrar cierto tipo de resultados no numéricos relacionados con la validez de sentencias simulando la manera de razonar humana.

Para definir los objetivos comunes de los sistemas enmarcados en el ámbito de la ATP, [Bib-90], admitimos, en primer lugar, que todo lo que una máquina puede realizar para simular los procesos del pensamiento humano puede ser descrito formalmente. Por otra parte, entre las distintas herramientas capaces de realizar dicha formalización, en ATP se elige la lógica matemática porque es una ciencia muy elaborada que presenta ventajas tanto por la naturalidad con la que es capaz de representar el problema no formal como por su naturaleza canónica; en particular, es bien conocido que la lógica de primer orden con sus diferentes extensiones es lo suficientemente rica para proporcionar al menos el núcleo de unos formalismos que describan los procesos del pensamiento humano en los que centramos nuestra atención. Una vez formalizado un problema mediante una sentencia lógica, podemos pensar que toda solución de éste puede generarse mediante la demostración de dicha sentencia. Presuponemos finalmente que existe un método general para descubrir automáticamente cualquier prueba hecha por el hombre.

Teniendo en cuenta las consideraciones anteriores, observamos que los principales objetivos en ATP comienzan por buscar una lógica conveniente que permita representar problemas; fijada ésta será necesario encontrar métodos generales de demostración y técnicas adecuadas a ella, comprobando después la viabilidad de representar problemas de diferentes áreas dentro del marco de la lógica elegida y la capacidad de resolución de los métodos definidos. El objetivo final consiste en obtener un sistema que presente una cierta capacidad de imitar los procesos del razonamiento humano manteniendo la generalidad que la lógica y los algoritmos llevan consigo.

La preocupación por encontrar un procedimiento general para demostrar teoremas data de muy antiguo; podemos remontarnos a Leibniz, seguido posteriormente por Peano y por la escuela de Hilbert. Ya en 1930 Herbrand propuso un método mecánico para demostrar teoremas que

más tarde gracias, a la invención del ordenador digital, pudo mecanizarse. Las primeras demostraciones automáticas se realizaron para la lógica proposicional, ampliándose después para la lógica de predicados; en concreto, en 1960 Gilmore consigue una implementación del método de Herbrand que será mejorada por Davis y Putman. En 1965 Robinson introduce la técnica de la resolución que por su eficiencia y facilidad de implementación supone un gran avance para la ATP. Se desarrollan entonces gran cantidad de variantes de la resolución, como la hiperresolución, la resolución unitaria, la SL-resolución, etc. Por otro lado para tratar razonamientos relativos a la igualdad se proponen métodos computacionales como la demodulación, la paramodulación y el método de Knuth-Bendix en 1970. Paralelamente se desarrollan también distintas variantes de la unificación introducida previamente por Robinson, por ejemplo, la teoría ecuacional.

Mientras que durante la década de los sesenta y comienzos de los setenta los intereses se centran principalmente en el punto de vista lógico y, más concretamente en encontrar métodos para resolver mecánicamente problemas de la lógica clásica con igualdad, en los últimos años se han mecanizado multitud de lógicas no clásicas, y ha habido mayor desarrollo de los sistemas diseñados para proporcionar herramientas interactivas agradables al usuario, que simulan técnicas de demostración humanas para resolver problemas en diferentes áreas. Ya en los años setenta surge el demostrador de Boyer y Moore que sigue siendo mejorado en la actualidad y que presenta cierta proximidad humana por su forma de construir demostraciones por medio de heurísticas. Los sistemas de la familia LCF (*Logic for Computable Logic*), en concreto el LCF de Edimburgo, están más enfocados a lograr mecanizar técnicas de deducción natural consiguiendo una interacción con el usuario por medio del uso de tácticas. La principal innovación de estos sistemas consiste en que incorporan la lógica formal a un lenguaje de programación funcional, ML, que sirve como metalenguaje. Los distintos sistemas que comprende esta familia (NuPRL, HOL, Veritas, Isabell, etc.) difieren primordialmente en la lógica que utilizan para construir las demostraciones.

Resulta difícil enumerar la cantidad de sistemas englobados en el campo de la ATP, [LMR-87], no trataremos aquí de hacer un estudio detallado de ellos sino simplemente señalaremos que gran parte de las diferencias entre unos y otros se debe a los problemas específicos

para los que han sido diseñados y por tanto a la lógica que soportan. La lógica de primer orden considerada casi exclusivamente en los sistemas primitivos se ha ido extendiendo de múltiples formas para abarcar los distintos campos que cubre la ATP. La utilidad actual de estos sistemas abarca diferentes áreas que van desde la ingeniería del *software*, donde la ATP se usa para síntesis de programas y verificación, hasta el diseño del *hardware*, donde ésta sirve, por ejemplo, para especificar el comportamiento de circuitos; sin olvidar su aplicación en el control de bases de datos y en la inteligencia artificial.

Este trabajo está orientado a diseñar un sistema de deducción automática que incorpore tanto la aplicación clásica de servir de herramienta para demostrar teoremas matemáticos como la de ser un instrumento mecánico que permita especificar de forma natural propiedades de programas funcionales, que puedan después ser verificadas.

De acuerdo con este marco, nuestro trabajo se ocupa de concretar para el nuevo sistema los objetivos generales en ATP indicados al comienzo de esta introducción, que podemos resumir en: fijar una lógica que sea capaz de formalizar nuestros problemas, encontrar unos mecanismos generales de deducción, y obtener como resultado una herramienta que simule ciertos razonamientos humanos.

La lógica computacional que proponemos para la formalización de los problemas matemáticos y las propiedades de programas funcionales, en que estamos interesados es una lógica de primer orden extendida con polimorfismo y recursión. Una vez fijada la lógica, se comprueba que tiene la capacidad de expresión deseada; pudiendo, por ejemplo, axiomatizar la aritmética estándar de los números naturales y especificar la teoría de listas. LLegados a este punto será necesario determinar los algoritmos que permitan automatizar las demostraciones; para ello se definen una serie de cálculos para la lógica en cuestión.

El primer mecanismo de deducción definido consiste en una extensión del método de los tableaux; este algoritmo será parcialmente implementado constituyendo una de las técnicas de comprobación y automatización de demostraciones que incorpora el sistema resultante. La falta de naturalidad de este mecanismo y la dificultad de su uso cuando se trabaja con fórmulas en las que aparecen funciones recursivas nos hace pensar en ampliar las técnicas de demostración de

nuestro sistema con métodos más cercanos a los razonamientos humanos. A partir del método de los tableaux se define un cálculo de deducción natural correcto y completo que refleja la semántica operacional de la recursión pero que como el anterior continúa siendo infinitario. Este sistema de deducción sirve a su vez de base para construir un último cálculo, también de deducción natural, que sustituye las reglas infinitarias poco manejables propias de la recursión por una regla de inducción de punto fijo.

El sistema de deducción automática resultante servirá para mecanizar pruebas de la lógica que hemos desarrollado utilizando técnicas basadas en los cálculos anteriormente mencionados. Estas demostraciones son guiadas por el usuario, que dispone de un lenguaje bastante natural de comunicación con el sistema, y son comprobadas por la máquina o bien son automatizadas en algunos de sus pasos.

La característica primordial que podemos destacar en nuestro demostrador es la proximidad del lenguaje de comunicación con el sistema al lenguaje usado habitualmente en los razonamientos matemáticos, y la facilidad para especificar funciones recursivas y tipos construidos. La organización del control de las pruebas y el lenguaje en que éstas se escriben tienen su origen en el sistema MIZAR-MSE ([Mos-85], [TB--85], [PR--88]) creado para la lógica de primer orden con igualdad y tipos. No obstante, nuestro demostrador presenta claros avances tanto desde el punto de vista lógico, por la extensión con polimorfismo y recursión, como desde el punto de vista humano por la interacción con el sistema y la naturalidad en la especificación de funciones y tipos.

La siguiente especificación, correspondiente al tipo cadenas (string) de elementos de un alfabeto cualquiera representado por un tipo variable *, es un ejemplo que permite mostrar algunas de las características de nuestro sistema antes señaladas.

```
type string/1;
const empty:string(*);
func head:string(*)->*, tail:string(*)->string(*),
     concat:* ^ string(*)->string(*), null:string(*)->bool;
A1# null(empty) = true;
A2# for x:*, y:string(*) holds (not x = bot and not y = bot) implies
                                  null(concat(x,y)) = false;
```

```
A3# for x:*, y:string(*) holds (not x = bot and not y = bot) implies
                                    head(concat(x,y)) = x;
A4# for x:*, y:string(*) holds (not x = bot and not y = bot) implies
                                    tail(concat(x,y)) = y;
```

El siguiente ejemplo refleja cómo el lenguaje destinado a la comunicación con el sistema permite al usuario declarar funciones recursivas de una forma similar a como se especifican habitualmente los programas recursivos.

```
funrec append:string(*) ^ string(*)->string(*)
   append(x,y) = if null(x) then y
                            else concat(head(x),append(tail(x),y));
```

Una vez especificada la función *append* será posible demostrar propiedades de ésta; por ejemplo su asociatividad expresada mediante la sentencia:

```
for x:string(*),y:string(*),z:string(*) holds append(append(x,y),z)
                                        = append(x,append(y,z))
```

Desde el punto de vista de la implementación señalamos que hemos conseguido un prototipo de un sistema semi-automático e interactivo que funciona en un sistema UNIX y que está programado en PROLOG.

Respecto a la estructuración, el trabajo se organiza en cuatro capítulos cada uno de los cuales comienza con una breve introducción y se divide en secciones. Describimos las lineas generales del contenido de cada uno de ellos.

El primer capítulo está dedicado a la definición de la sintaxis y la semántica de una lógica de predicados con polimorfismo y recursión (PLPR). Esta lógica se presenta como soporte o lenguaje objeto del sistema de demostración automática de teoremas que construiremos y que está orientado a realizar demostraciones matemáticas y pruebas de propiedades de programas funcionales. Los ejemplos descritos al final del capítulo son una muestra de su capacidad para expresar problemas dentro de estos campos.

Las principales características que la lógica PLPR aporta a la de primer orden son el uso de variables y constructores de tipo, y la existencia del operador λ para construir λ-abstracciones y del μ-operador para definir funciones recursivas. En cuanto a la

semántica, cabe destacar la interpretación de los tipos como cpo's planos. Las funciones que se definen entre dichos dominios son estrictas y por tanto continuas, lo que permite definir la semántica del μ-operador como el menor punto fijo de un operador continuo.

La lógica PLPR está ampliamente fundamentada, se demuestran sus propios lemas de coincidencia y sustitución análogos a los de la lógica de predicados [EFT-84]. Se verifica también un lema de sustitución de tipos y se prueban otros resultados propios de este lenguaje que serán utilizados a lo largo del trabajo.

En el segundo capítulo se introduce una extensión del método de los tableaux de Smullyan [Smu-68] que adaptado a las peculiaridades de nuestra lógica se convierte en un mecanismo de refutación para PLPR. Una característica propia de los tableaux es su facilidad de automatización y su ayuda en la construcción de cálculos completos. El método de los tableaux extendido resulta ser un algoritmo correcto y completo de semidecisión de la insatisfactibilidad en PLPR. Del teorema de completitud de los tableaux se deducen una serie de propiedades que sirven como condiciones suficientes para la completitud de cualquier cálculo para PLPR.

El tercer capítulo comienza definiendo un sistema de deducción natural para PLPR construido de manera que verifique las condiciones de completitud obtenidas en el capítulo anterior a partir del método de los tableaux. Se comprueba que este cálculo es correcto y completo, si bien, como todo cálculo completo para PLPR, es infinitario. Esta última afirmación puede considerarse como una consecuencia de la Π^1_1-completitud del problema de validez en nuestra lógica, hecho que se prueba formalmente en este capítulo.

Puesto que nuestro objetivo es conseguir un demostrador de teoremas basado en PLPR, la infinitud de los cálculos presenta un grave problema para su implementación. Por otro lado, existen reglas de derivación del cálculo presentado, basadas en aproximaciones sintácticas, que no resultan muy naturales desde el punto de vista de similitud con los razonamientos humanos. Mucho más manejable, tanto para su implementación como para su uso, es la regla de inducción. Con el fin de introducir la inducción en el cálculo, se construye el conjunto de fórmulas continuas, sobre el que se define una regla de inducción de punto fijo y para el que se realizan algunos estudios de complejidad. Esta regla junto con otra reflejando que la semántica del

operador μ es el punto fijo hacen que el cálculo sea suficientemente rico (como puede observarse en los ejemplos) para efectuar demostraciones propias de lenguajes funcionales que presentan cierta dificultad y para derivar una regla de inducción en los naturales.

Partiendo de la lógica y los cálculos anteriores, en el cuarto y último capítulo presentamos un sistema de deducción automática llamado MIZ-PR que en cierto modo simula el comportamiento humano, permitiendo automatizar demostraciones de la lógica PLPR utilizando un lenguaje cercano a como se escriben demostraciones matemáticas.

El capítulo comienza señalando ciertos aspectos que caracterizan el sistema como son las diferentes técnicas usadas en la mecanización de las demostraciones, que están basadas en los cálculos definidos en los capítulos anteriores. Las técnicas de demostración se dividen en dos bloques fundamentales; en uno se engloban aquellas que conducen la prueba hacia atrás descomponiendo el objetivo en subobjetivos, en el otro, se incluyen las técnicas de demostración hacia adelante que prueban el objetivo haciendo referencia a axiomas y teoremas ya demostrados. Asimismo se estudia la estructura en forma de árbol inherente a las demostraciones contemplando así las demostraciones anidadas como subárboles de la prueba global. También se analiza la forma en que tiene lugar el control de las demostraciones.

Posteriormente, fijaremos la sintaxis del lenguaje que sirve de comunicación del usuario con el sistema mediante una gramática en forma de Backus-Naur; una serie de ejemplos prácticos permiten obtener una visión bastante aproximada de la capacidad del sistema y del manejo del mismo.

El capítulo finaliza con una breve descripción de la implementación. Se hace énfasis en los programas Prolog utilizados para el control y automatización de las pruebas. En el anexo de final del trabajo puede encontrarse un listado de estos programas.

CAPITULO I

SINTAXIS Y SEMANTICA DE LA LOGICA COMPUTACIONAL PLPR

Al ser nuestro objetivo final construir un sistema de deducción automática hay que destacar que éste soportará un lenguaje objeto, esto es, un lenguaje lógico en el cual se expresen y prueben afirmaciones. EL lenguaje objeto es uno de los factores que más determina la utilidad de un sistema de deducción automática en las tareas de razonamiento formal.

Una somera revisión de los lenguajes objeto de algunos de los sistemas existentes ayudará a comprender sus distintos enfoques y a enmarcar nuestro trabajo.

LCF, [Pau-87], [Plo-77], soporta una familia de lenguajes objeto basada en una lógica de predicados con polimorfismo y λ-cálculo llamada PPλ parametrizada por la signatura propia de cada teoría. PPλ es un sistema de deducción natural desarrollado a partir del λ-cálculo con tipos de Scott con una regla de inducción estructural. Por tanto, LCF es muy adecuado para trabajar en contextos que utilicen funciones parciales, menor punto fijo de funcionales, evaluación perezosa y semántica denotacional.

NuPRL, [Con-86], aunque suele considerarse dentro de la familia de los sistemas LCF tiene como lenguaje objeto la lógica CTT (*Constructive Type Theory*). El sistema tiene la capacidad de construir mediante la información computacional obtenida en la demostración de una afirmación existencial, una representación del objeto que prueba la validez de esa afirmación.

El demostrador de Boyer y Moore (BM), [BM--79], [BM--88], soporta una lógica clásica con igualdad, símbolos de función y ciertas definiciones recursivas. Su gran potencia reside en la facilidad del manejo de la inducción, lo que es bastante costoso en LCF. Sin embargo, BM no permite el uso de funciones parciales y no puede ser aplicado para hacer demostraciones en una lógica constructiva; no obstante, podría simularse una lógica constructiva introduciendo axiomas auxiliares.

Los sistemas basados en reescritura de términos se han diseñado para manejar sistemas de ecuaciones, este es el caso de AFFIRM, [Ger-80], con su lógica ecuacional con tipos y conectivos

proposicionales clásicos. Estos sistemas resultan muy tediosos cuando aparecen cuantificadores, situación que suele solucionarse utilizando funciones de Skolem.

Las lógicas de orden superior convierten al sistema en una buena herramienta para tratar restricciones temporales y son muy empleadas en la especificación del comportamiento del *hardware* como es el caso de HOL [Gor-89]; otros sistemas basados en lógicas de orden superior son por ejemplo Veritas [HD--86] e Isabell [Pau-86], este último es un sistema para derivar reglas de inferencia basado en unificación de orden superior. Sin embargo, la fundamentación teórica de las lógicas de orden superior ofrece dificultades especiales con respecto a las de primer orden y por ello están menos desarrolladas.

Este capítulo está dedicado primordialmente a introducir una lógica computacional decribiendo su sintaxis y su semántica. Dicha lógica servirá como lenguaje objeto de un sistema enmarcado en el ámbito de la deducción automática. Se trata de una lógica de primer orden extendida con polimorfismo y recursión que llamaremos PLPR (*Predicate Logic with Polymorphism and Recursion*).

En el diseño de PLPR se ha tenido en cuenta no sólo la riqueza de la lógica en el sentido de su capacidad de expresar ideas y razonar sobre ellas sino también su proximidad al dominio de los problemas que queremos que resuelva nuestro sistema. Recordamos que los campos en los que hemos centrado nuestro interés son las demostraciones matemáticas y las propiedades de lenguajes funcionales.

Una característica de nuestra lógica, ya descrita en [LN--90], es el permitir introducir y construir nuevos tipos usando constructores. Los tipos de PLPR pueden ser polimórficos lo que da mayor potencia a las estructuras aunque, como es sabido, éstos pueden dificultar el tratamiento de los subtipos. Por otro lado PLPR posee el tipo booleano trivalorado que facilita los razonamientos en teoría de dominios. Una vez que se establece una asignación a las variables de tipo, cada tipo denota un cpo (*complete partial order*) plano, y cada término un elemento del cpo denotado por su tipo.

PLPR permite escribir expresiones funcionales con tipos que representan funciones parciales estrictas entre cpo's planos y que se construyen por medio de símbolos de función y de los operadores λ y μ que dan lugar al λ-cálculo y a la recursión.

Los términos de PLPR tienen tipos de primer orden. A nivel sintáctico, cabe destacar la existencia de un símbolo especial para representar el *bottom* (menor elemento de cada dominio) útil para definir funciones parciales, y el uso de las n-tuplas (términos con tipo producto) para unificar el tratamiento de las funciones.

En cuanto a las fórmulas, los conectivos utilizados son los propios de una lógica clásica de primer orden. Las fórmulas básicas son las predicativas, construidas utilizando símbolos de predicado con tipo y las aproximaciones o desigualdades que proporcionan una herramienta sintáctica cómoda para hacer razonamientos que utilicen las propiedades de la igualdad y de los órdenes parciales planos.

1. LA SINTAXIS DE PLPR

El lenguaje PLPR se compone de los siguientes objetos: tipos, expresiones funcionales, términos y fórmulas que permiten escribir sentencias y razonar sobre ellas. A continuación se introduce la sintaxis de dichos objetos y se definen distintas operaciones que permiten su manipulación.

1.1 DEFINICION DE LOS OBJETOS

Consideraremos un conjunto fijo, TVar, con una cantidad numerable de variables de tipo que se denotan por ρ o si es necesario con subíndices: $\rho_1, \rho_2, \ldots, \rho_n, \ldots$

Definición 1.1.1

Una *signatura de tipos* Σ_t es un conjunto a lo sumo numerable cuyos elementos son símbolos de constructures de tipo ct/n donde n representa la aridad del constructor ct. Si $ct/n \in \Sigma_t$ con n=0, se dice que ct es un símbolo de constante de tipo.

A partir de Σ_t se define el *conjunto* de Σ_t-*tipos de primer orden* que se denota por $T_p(\Sigma_t)$ y cuyos elementos se nombran con la letra τ simplemente, o con subíndices o primas, y se definen mediante las siguientes reglas BNF:

$\tau ::= \rho$	Siendo $\rho \in$ TVar. *Tipo variable*
| ct	Donde $ct/0 \in \Sigma_t$. *Tipo constante*
| bool	*Tipo booleano*
| $ct(\tau_1, \ldots, \tau_n)$	Donde $ct/n \in \Sigma_t$. *Tipo construido*
| $\tau_1 x \ldots x \tau_n$	*Tipo producto estricto*

Consideraremos también Σ_t-tipos funcionales, que son aquellos tipos de la forma $\tau_1 \rightarrow \tau_2$, siendo $\tau_1, \tau_2 \in T_p(\Sigma_t)$. Un Σ_t-*tipo*, τ, es o bien un Σ_t-tipo de primer orden o bien un Σ_t-tipo funcional.

Si en la construcción sintáctica de un Σ_t-tipo τ no aparecen variables de tipo se dice que τ es *monomórfico*, en caso contrario τ es *polimórfico*. □

Llamaremos $MT_p(\Sigma_t)$ al conjunto de Σ_t-tipos monomórficos de primer

orden. Los elementos de $MT_p(\Sigma_t)$ se denotan por ν, ν_1, ν_2,...

Definición 1.1.2

Una Σ_t-*sustitución de tipos* $\sigma = \frac{\tau_1 \ldots \tau_n}{\rho_1 \ldots \rho_n}$ es una aplicación parcial del conjunto de los Σ_t-tipos en si mismo. Utilizaremos la notación $\tau\sigma$ para expresar el resultado de aplicar la sustitución σ a un tipo τ y la definiremos del siguiente modo:

$$\rho\sigma = \begin{cases} \tau_i & \text{si } \rho = \rho_i \\ \rho & \text{en otro caso} \end{cases}$$

$bool\ \sigma = bool$

$ct\ \sigma = ct$

$ct(\tau_1, \ldots, \tau_n)\sigma = ct(\tau_1\sigma, \ldots, \tau_n\sigma)$

$(\tau_1 x \ldots x \tau_n)\sigma = \tau_1\sigma x \ldots x \tau_n\sigma$

$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$

Se dice que un Σ_t-tipo τ' es una Σ_t-*instancia de un* Σ-*tipo* τ y se escribe $\tau' \leq \tau$ si exite una Σ_t-sustitución de tipos σ tal que $\tau' = \tau\sigma$. En el caso de que $\nu \leq \tau$ se dice que ν es una *instancia monomórfica* de τ. Se dice que dos Σ_t-tipos τ_1 y τ_2 son *unificables* si existe una Σ_t-sustitución de tipos σ tal que $\tau_1\sigma = \tau_2\sigma$. □

Definición 1.1.3

Dada una signatura de tipos Σ_t, una *signatura de datos* Σ_d es un conjunto numerable cuyos elementos son símbolos de datos que llevan asociado un Σ_t-tipo. Los elementos de Σ_d pueden ser símbolos de constante ($c: \rightarrow \tau \in \Sigma_d$), símbolos de función ($f: \tau_1 \rightarrow \tau_2 \in \Sigma_d$), o símbolos de predicado ($p: \tau \in \Sigma_d$) donde τ, $\tau_1 \rightarrow \tau_2$ y τ son el *tipo más general* de c, f y p respectivamente.

Dadas una signatura de tipos Σ_t y una signatura de datos Σ_d, una *signatura* Σ es un par $\Sigma = \langle \Sigma_t, \Sigma_d \rangle$. □

En lo sucesivo simplificamos $T_p(\Sigma_t)$ y $MT_p(\Sigma_t)$ mediante $T_p(\Sigma)$ y $MT_p(\Sigma)$, respectivamente, y hablaremos de Σ-tipos y Σ-sustituciones.

Dada una signatura Σ, consideraremos para cada Σ-tipo τ de primer orden un conjunto numerable de variables de dato Var_τ de tal forma que para cada τ, $\tau' \in T_p(\Sigma)$, $\tau \neq \tau'$, $Var_\tau \cap Var_{\tau'} = \emptyset$. A la unión de todos estos conjuntos, cuando τ recorre $T_p(\Sigma)$, le llamamos $Var(\Sigma)$; sus

elementos se denotan por x, y, z, o por las mismas letras con subíndices. De la misma manera consideraremos un conjunto numerable de variables de función, $FVar_{\tau_1\to\tau_2}$, por cada tipo funcional; estos conjuntos serán disjuntos entre sí, su unión se denota por FVar(Σ), y sus elementos por las letras X, Y, que admiten subíndices.

Definición 1.1.4

Dada una signatura $\Sigma = \langle \Sigma_t, \Sigma_d \rangle$, el *conjunto de Σ-términos*, T_Σ, cuyos elementos se denotan por $t:\tau$, y el *conjunto de Σ-expresiones funcionales*, $E_F(\Sigma)$, cuyos elementos se denotan por $M:\tau_1\to\tau_2$, se definen por recursión mutua mediante las siguientes reglas:

$t:\tau ::= \bot:\tau$			*Término bottom*
$\mid x:\tau$		$x \in Var_\tau$.	*Término variable*
$\mid c:\tau'$		$c:\to\tau \in \Sigma_d$, $\tau' \leq \tau$.	*Término constante*
$\mid$ true:bool			*Término booleano cierto*
$\mid$ false:bool			*Término booleano falso*
$\mid (t_1:\tau_1,\ldots,t_n:\tau_n):\tau_1\times\ldots\times\tau_n$			*n-tupla de términos*
$\mid$ (if t:bool then $t_1:\tau$ else $t_2:\tau$):τ			*Término condicional*
$\mid (M\ t:\tau_1):\tau_2$		$M:\tau_1\to\tau_2 \in E_F(\Sigma)$	*Término funcional*

$M:\tau_1\to\tau_2 ::= \bot:\tau_1\to\tau_2$			*Función indefinida*
$\mid X:\tau_1\to\tau_2$		$X \in FVar_{\tau_1\to\tau_2}$	*Función variable*
$\mid f:\tau'_1\to\tau'_2$		$f:\tau_1\to\tau_2 \in \Sigma_d$, $\tau'_1\to\tau'_2 \leq \tau_1\to\tau_2$	*Símb. funcional*
$\mid (\lambda x_1:\tau_1\ldots x_n:\tau_n.\ t:\tau):\tau_1\times\ldots\times\tau_n\to\tau$			*Lambda abstracción*
$\mid (\mu X:\tau_1\to\tau_2.\ M:\tau_1\to\tau_2):\tau_1\to\tau_2$			*μ-Operador*

Dado un Σ-término $t:\tau$ (una Σ-expresión funcional $M:\tau_1\to\tau_2$) se dice que $t:\tau$ es de tipo τ ($M:\tau_1\to\tau_2$ es de tipo $\tau_1\to\tau_2$). Un Σ-término (una Σ-expresión funcional) es *monomórfico* (*monomórfica*) cuando no contiene ninguna variable de tipo. En caso contrario es *polimórfico* (*polimórfica*). □

A pesar de tener conjuntos de símbolos de variables disjuntos para cada Σ-tipo, puesto que todo Σ-término y toda Σ-expresión funcional lleva asociado su tipo, utilizaremos la misma notación para los símbolos de variable de cualquier tipo; distinguiremos sólo si la variable es de dato o función, utilizando mayúsculas en el segundo caso.

En lo sucesivo siempre que no sea necesario especificar, cuando hablemos de variables nos referimos tanto a los términos variables $(x:\tau)$ como a las expresiones funcionales variables $(X:\tau_1 \rightarrow \tau_2)$.

Ejemplo 1.1.5

Sea Σ_t = {int/0, stack/1}. Las siguientes construcciones son Σ_t-tipos:

$\rho \times \text{stack}(\rho)$,

$\text{stack}(\text{stack}(\text{int}) \times \text{bool})$,

$\text{stack}(\rho_1) \rightarrow \text{int} \times \text{stack}(\rho_2) \times \text{stack}(\text{bool})$

De ellos los dos primeros son de primer orden, siendo el segundo monomórfico.

Para la signatura de tipos anterior damos la signatura de datos

Σ_d = {0:→int, nil:→stack(ρ), suc: int→int, pred: int→int,
push: $\rho \times$stack(ρ)→stack(ρ)}

Si $\Sigma s = \langle \Sigma_t, \Sigma_d \rangle$, se pueden construir, por ejemplo, los siguientes Σs-términos y Σs-expresiones funcionales:

(suc (suc 0:int):int):int,

(push (x:int, nil:stack(int)):int×stack(int)):stack(int)

(λx1:bool x2:int. (if x1:bool then (pred x2:int):int
else ⊥:int):int):bool×int→int

(μX:stack(ρ)→int. (λy:stack(ρ).
(pred (X y:stack(ρ)):int):int):stack(ρ)→int):stack(ρ)→int

Definición 1.1.6

Dada una signatura Σ, el *conjunto de Σ-fórmulas* $F(\Sigma)$ de elementos $\varphi, \psi, \ldots$ es el menor conjunto que satisface las siguientes reglas:

$\varphi ::= t_1:\tau \leq t_2:\tau$	*Aproximación.*
$\mid (p\ t:\tau')$	$\tau' \leq \tau$, $p:\tau \in \Sigma_d$. *Fórmula predicativa.*
$\mid \neg\varphi$	*Negación.*
$\mid (\varphi \vee \psi)$	*Disyunción.*
$\mid (\varphi \wedge \psi)$	*Conjunción.*
$\mid \exists x:\tau\ \varphi$	*Cuantificación existencial.*
$\mid \forall x:\tau\ \varphi$	*Cuantificación universal.*

Las aproximaciones y las fórmulas predicativas constituyen las *fórmulas atómicas*. Una fórmula se dice *monomórfica* cuando no contiene ninguna variable de tipo. En caso contrario es *polimórfica*. □

1.2 EL LENGUAJE PLPR

Los objetos definidos en el apartado anterior servirán como base para definir nuestro lenguaje, no obstante, puesto que hemos utilizado la misma notación para los símbolos de variables de cualquier tipo, hemos de adoptar una serie de precauciones para no incurrir en ambigüedades. Por ejemplo, a un Σ-término $x:\tau$ puede aplicársele una Σ-sustitución de tipos σ que transforme τ en τ'. Si definimos $[x:\tau]\sigma = x:\tau\sigma$, tendremos que tener en cuenta que, en la izquierda de la igualdad, x representa un símbolo de variable del conjunto Var_τ mientras que en la derecha, x es un elemento de $Var_{\tau'}$.

Para no entrar en conflictos al definir la semántica de nuestro lenguaje, a la sintaxis abstracta del apartado anterior, le añadimos ciertas restricciones, concretamente, imponemos la no existencia de variables libres con el mismo símbolo y tipos distintos y unificables. El concepto de variable libre se define a continuación.

Definición 1.2.1.

Definimos el *conjunto de variables libres en un término* $t:\tau$, representado por $Lib(t:\tau)$, y el *conjunto de variables libres en una expresión funcional* $M:\tau_1\to\tau_2$, representado por $Lib(M:\tau_1\to\tau_2)$, por inducción mutua sobre los términos y las expresiones funcionales del siguiente modo:

$Lib(\bot:\tau) := \emptyset$

$Lib(c:\tau) := \emptyset$

$Lib(x:\tau) := \{x:\tau\}$

$Lib(true:bool) := \emptyset$

$Lib(false:bool) := \emptyset$

$Lib((t_1:\tau_1,\ldots,t_n:\tau_n):\tau_1\times\ldots\times\tau_n) := Lib(t_1:\tau_1) \cup\ldots\cup Lib(t_n:\tau_n)$

$Lib((\text{if } t:bool \text{ then } t_1:\tau \text{ else } t_2:\tau):\tau) :=$
$Lib(t:bool) \cup Lib(t_1:\tau) \cup Lib(t_2:\tau)$

$Lib((M\ t:\tau_1):\tau_2) := Lib(M:\tau_1\to\tau_2) \cup Lib(t:\tau_1).$

$Lib(\bot:\tau_1\to\tau_2) := \emptyset$

$Lib(f:\tau_1\to\tau_2) := \emptyset$

$Lib(X:\tau_1\to\tau_2) := \{X:\tau_1\to\tau_2\}$

$Lib((\lambda x_1:\tau_1\ldots x_n:\tau_n.\, t:\tau):\tau_1\times\ldots\times\tau_n\to\tau) := Lib(t:\tau)\setminus\{x_1:\tau_1,\ldots,x_n:\tau_n\}$

$Lib((\mu X:\tau_1\to\tau_2.M:\tau_1\to\tau_2):\tau_1\to\tau_2) := Lib(M:\tau_1\to\tau_2)\setminus\{X:\tau_1\to\tau_2\}$

Decimos que una variable está libre en un término $t:\tau$ (en una expresión funcional $M:\tau_1\to\tau_2$), si pertenece al conjunto $Lib(t:\tau)$ ($Lib(M:\tau_1\to\tau_2)$).

Definimos el *conjunto de variables (de dato y función) libres de una fórmula* φ y lo representamos por $Lib(\varphi)$ del siguiente modo:

$Lib(t_1:\tau \leq t_2:\tau) := Lib(t_1:\tau) \cup Lib(t_2:\tau)$

$Lib((p\ t:\tau)) := Lib(t:\tau)$

$Lib(\neg\varphi) := Lib(\varphi)$

$Lib((\varphi \vee \psi)) := Lib(\varphi) \cup Lib(\psi)$

$Lib((\varphi \wedge \psi)) := Lib(\varphi) \cup Lib(\psi)$

$Lib(\forall x:\tau\ \varphi) := Lib(\varphi)\setminus\{x:\tau\}$

$Lib(\exists x:\tau\ \varphi) := Lib(\varphi)\setminus\{x:\tau\}$

Se dice que una variable está libre en una fórmula φ si pertenece al cunjunto $Lib(\varphi)$, en caso contrario se dice que está ligada. □

Para cada Σ, PLPR se refiere a un subconjunto de $F(\Sigma)$ cuyas fórmulas cumplen una restricción necesaria para evitar la ambigüedad en el tipo de las variables libres. Esta restricción queda formalizada en la definición siguiente.

Definición 1.2.2

Para cada signatura Σ, el lenguaje PLPR asociado a Σ viene determinado por el conjunto $L(\Sigma)$ formado por las Σ-fórmulas φ de $F(\Sigma)$ tales que no existe ninguna variable x y dos tipos distintos τ_1 y τ_2 de manera que $x:\tau_1 \in Lib(\varphi)$ y $x:\tau_2 \in Lib(\varphi)$ siendo τ_1 y τ_2 unificables y no existe ninguna variable de función X tal que $X:\tau_1\to\tau_2 \in Lib(\varphi)$, $X:\tau'_1\to\tau'_2 \in Lib(\varphi)$ siendo $\tau_1\to\tau_2$ y $\tau'_1\to\tau'_2$ unificables y distintos. □

La restricción impuesta a las Σ-fórmulas se puede particularizar para los Σ-términos y Σ-expresiones funcionales. Para no complicar la notación conservaremos los nombres $T_\Sigma(\Sigma)$, $Ef(\Sigma)$, lo que no dará lugar a confusión porque desde ahora siempre consideraremos expresiones sintácticas que cumplen estas restricciones. Por otro lado, a partir de ahora hablaremos de Σ-fórmulas para referirnos a los elementos de $L(\Sigma)$ e incluso diremos únicamente fórmulas siempre que no haya ambigüedad o las propiedades que se enuncien sean independientes de la signatura. Igualmente podremos hablar de tipos, términos o expresiones funcionales.

Llamaremos $MT_m(\Sigma)$ al conjunto de términos monomórficos, $ME_f(\Sigma)$ al conjunto de expresiones funcionales monomórficas y $ML(\Sigma)$ al conjunto de fórmulas monomórficas.

1.3 SIMPLIFICACIONES SINTACTICAS

El hecho de que tanto los términos como las expresiones funcionales lleven pegado su tipo resulta muy expresivo y presenta grandes ventajas operacionales, pero en ocasiones dificulta su lectura, es por ello que en lo sucesivo admitiremos la siguientes simplificaciones en la sintaxis de PLPR:

- En los siguientes términos y expresiones funcionales eliminamos el tipo, puesto que su escritura resulta redundante:

  true, false,
  $(t_1:\tau_1,\ldots,t_n:\tau_n)$,
  (if t:bool then $t_1:\tau$ else $t_2:\tau$),
  $(\lambda x_1:\tau_1 \ldots x_n:\tau_n.\, t:\tau)$,
  $(\mu X:\tau_1 \rightarrow \tau_2.\, M)$.

- El tipo de las variables de dato ligadas por un cuantificador existencial o universal, o por una λ-abstracción puede eliminarse en su aparición en el interior de las fórmulas o expresiones, ya que su tipo queda explícito cuando las variables aparecen junto al cuantificador o al operador λ. Así por ejemplo, escribiremos:

  $\forall x:\tau\ \forall y:\tau\ x \leq y$ en lugar de $\forall x:\tau\ \forall y:\tau\ x:\tau \leq y:\tau$
  $(\lambda x:nat\ y:bool.(\text{if } y \text{ then } x \text{ else } \bot:nat))$ en lugar de
  $(\lambda x:nat\ y:bool.(\text{if } y:bool \text{ then } x:nat \text{ else } \bot:nat))$

- Admitimos la supresión de la notación explícita de los tipos en los términos funcionales siempre que éste pueda determinarse al inferir los tipos de la expresión funcional que se aplica y del término sobre el que se aplica, naturalmente ambos tipos deben ser compatibles de acuerdo con las reglas habituales de inferencia de tipos (ver p.e. [Rea-89]).

  Por ejemplo, si $f:\tau_1 \rightarrow \tau_2 \in \Sigma_d$ y $\tau_1' = \tau_1\sigma$, sabremos que $(f\ t:\tau_1')$ tiene tipo $\tau_2\sigma$ y admitiremos la escritura de este término sin explicitar su tipo.

Para estas inferencias se puede usar un algoritmo semejante al dado en [Mil-78] para las combinaciones de términos, utilizando como datos, por un lado, el tipo más general de los símbolos de función y de constante, y por otro, las reglas de construción y de instanciación de tipos.

En lo referente a la sintaxis de las fórmulas suprimimos los paréntesis en las conjunciones y disyunciones siempre que ésto no de lugar a conflictos. Por otro lado utilizaremos las siguientes simplificaciones:

$\varphi \longrightarrow \psi$ en vez de $\neg\varphi \vee \psi$

$\varphi \longleftrightarrow \psi$ en vez de $(\varphi \longrightarrow \psi) \wedge (\psi \longrightarrow \varphi)$

$t_1{:}\tau = t_2{:}\tau$ en vez de $t_1{:}\tau \subseteq t_2{:}\tau \wedge t_2{:}\tau \subseteq t_1{:}\tau$.

Ejemplo 1.3.1

Consideremos la signatura $\Sigma_+ = \langle \Sigma_t, \Sigma_d \rangle$, siendo:

$\Sigma_t = \{char/0,\ unit/0,\ \bullet/2\}$ y

$\Sigma_d = \{inl{:}\rho_1 \rightarrow\!\!\!\!\bullet\ (\rho_1,\rho_2),\ inr{:}\rho_2 \rightarrow\!\!\!\!\bullet\ (\rho_1,\rho_2),\ outl{:}\bullet(\rho_1,\rho_2)\rightarrow\rho_1,\ outr{:}\bullet(\rho_1,\rho_2)\rightarrow\rho_2\}$

Las siguientes son Σ_+-fórmulas que sirven para definir la suma estricta de dominios. En ellas se ha simplificado la notación de los tipos de acuerdo a los comentarios anteriores.

$\forall x{:}\rho_1\ (\neg x \subseteq \bot{:}\rho_1 \longrightarrow \neg(inl\ x) \subseteq \bot{:}\bullet(\rho_1,\rho_2))$

$\forall x{:}\rho_2\ (\neg x \subseteq \bot{:}\rho_2 \longrightarrow \neg(inr\ x) \subseteq \bot{:}\bullet(\rho_1,\rho_2))$

$\forall x{:}\rho_1\ \forall y{:}\rho_1\ ((inl\ x) \subseteq (inl\ y) \longleftrightarrow x \subseteq y)$

$\forall x{:}\rho_2\ \forall y{:}\rho_2\ ((inr\ x) \subseteq (inr\ y) \longleftrightarrow x \subseteq y)$

$\forall x{:}\rho_1\ \forall y{:}\rho_2\ ((inl\ x) \subseteq (inr\ y) \longleftrightarrow x = \bot{:}\rho_1)$

$\forall x{:}\rho_1\ \forall y{:}\rho_2\ ((inr\ y) \subseteq (inl\ x) \longleftrightarrow y = \bot{:}\rho_2)$

Las funciones outl y outr son las llamadas funciones destructoras. El carácter polimórfico del constructor $\bullet$ garantiza que lo que sigue son Σ_+-fórmulas.

$(outl\ (inl\ x_1{:}char){:}\bullet(char,\rho_2)) = x_1{:}char$

$(outr\ (inr\ x_2{:}unit){:}\bullet(\rho_1,unit)) = x_2{:}unit$

$(outl\ (inr\ x_2{:}unit){:}\bullet(\rho_1,unit)) = \bot{:}\rho_1$

$(outr\ (inl\ x_1{:}char){:}\bullet(char,\rho_2)) = \bot{:}\rho_2$.

1.4 SUSTITUCION

Las variables de tipo que aparecen en una Σ-fórmula pueden ser instanciadas por Σ-tipos de primer orden, asímismo las variables libres de una Σ-fórmula pueden ser sustituidas por Σ-términos del mismo tipo si se trata de variables de dato o por Σ-expresiones funcionales del mismo tipo si se trata de variables de función.

La Σ-sustitución de variables de tipo ha sido definida cuando se aplica a un tipo, definiremos ahora los conceptos de Σ-sustitución de variables, tanto de tipo como de dato y función, aplicada a una Σ-fórmula.

Definición 1.4.1

a) Σ-Sustitución de una variable de tipo por un Σ-tipo.

a1) El resultado de aplicar una Σ-sustitución de tipos σ a un Σ-término $t:\tau$ se denota por $[t:\tau]\sigma$ y se define mediante las siguientes reglas:

- $[\bot:\tau]\sigma := \bot:\tau\sigma$
- $[c:\tau]\sigma := c:\tau\sigma$
- $[x:\tau]\sigma := x:\tau\sigma$
- $[true]\sigma := true$ - $[false]\sigma := false$
- $[(t_1:\tau_1,\ldots,t_n:\tau_n)]\sigma := ([t_1:\tau_1]\sigma,\ldots,[t_n:\tau_n]\sigma)$
- $[(if\ t:bool\ then\ t_1:\tau\ else\ t_2:\tau)]\sigma :=$
  $(if\ [t:bool]\sigma\ then\ [t_1:\tau]\sigma\ else\ [t_2:\tau]\sigma)$
- $[(M\ t:\tau_1):\tau_2]\sigma := ([M:\tau_1\rightarrow\tau_2]\sigma\ [t:\tau_1]\sigma):\tau_2\sigma$

a2) El resultado de aplicar una Σ-sustitución de tipos σ a una Σ-expresión funcional $M:\tau_1\rightarrow\tau_2$ se denota por $[M:\tau_1\rightarrow\tau_2]\sigma$ y se define mediante las siguientes reglas:

- $[\bot:\tau_1\rightarrow\tau_2]\sigma := \bot:\tau_1\sigma\rightarrow\tau_2\sigma$
- $[f:\tau_1\rightarrow\tau_2]\sigma := f:\tau_1\sigma\rightarrow\tau_2\sigma$
- $[X:\tau_1\rightarrow\tau_2]\sigma := X:\tau_1\sigma\rightarrow\tau_2\sigma$
- $[(\lambda x_1:\tau_1\ldots x_n:\tau_n.\ t:\tau)]\sigma :=$
  $(\lambda z_1:\tau_1\sigma\ldots z_n:\tau_n\sigma.\ [t:\tau[z_1:\tau_1/x_1:\tau_1]\ldots[z_n:\tau_n/x_n:\tau_n]]\sigma)$ donde para cada i, z_i es igual a x_i si $x_i:\tau_i\sigma \notin Lib(t:\tau)$, y en caso contrario, z_i es un símbolo de variable tal que $z_i:\tau_i \notin Lib(t:\tau)$.
- $[(\mu X:\tau_1\rightarrow\tau_2.\ M)]\sigma := (\mu Y:\tau_1\sigma\rightarrow\tau_2\sigma.\ [M:\tau_1\rightarrow\tau_2[Y:\tau_1\rightarrow\tau_2/X:\tau_1\rightarrow\tau_2]]\sigma)$
donde Y es igual a X si $X:\tau_1\sigma\rightarrow\tau_2\sigma \notin Lib(M:\tau_1\rightarrow\tau_2)$; o $Y \notin FVar(\Sigma)$ es

tal que $Y:\tau_1\sigma\rightarrow\tau_2\sigma \notin Lib(M:\tau_1\rightarrow\tau_2)$ en caso contrario.

a3) El resultado de aplicar una Σ-sustitución de tipo σ a una Σ-fórmula φ se denota $[\varphi]\sigma$ o simplemente $\varphi\sigma$ y se define por inducción como sigue:

- $[t_1:\tau \leq t_2:\tau]\sigma := [t_1:\tau]\sigma \leq [t_2:\tau]\sigma$
- $[(p\ t:\tau)]\sigma := (p\ [t:\tau]\sigma)$
- $[\neg\varphi]\sigma := \neg[\varphi]\sigma$
- $[\varphi \vee \psi]\sigma := [\varphi]\sigma \vee [\psi]\sigma$
- $[\varphi \wedge \psi]\sigma := [\varphi]\sigma \wedge [\psi]\sigma$
- $[\forall x:\tau\ \varphi]\sigma := \forall z:\tau\sigma\ [\varphi[z:\tau/x:\tau]]\sigma$ donde $z = x$ si $x:\tau\sigma \notin Lib(\varphi)$; o z es una variable tal que $z:\tau\sigma \notin Lib(\varphi)$ en caso contrario.
- $[\exists x:\tau\ \varphi]\sigma := \exists z:\tau\sigma\ [\varphi[z:\tau/x:\tau]]\sigma$ con z como en el caso anterior.

Si φ es una fórmula que contiene en su sintaxis una variable de tipo ρ y $\sigma = \nu/\rho$ es una Σ-sustitución de tipos decimos que $\varphi\sigma$ es una Σ-*instancia* de φ.

b) Σ-Sustitución de una variable de dato por un Σ-término.

b1) La Σ-sustitución en un término de una variable de dato libre por un Σ-término del mismo tipo, se define como sigue:

- $\perp:\tau_1[t:\tau/x:\tau] := \perp:\tau_1$
- $c:\tau_1[t:\tau/x:\tau] := c:\tau_1$
- $x_1:\tau_1[t:\tau/x:\tau] := \begin{cases} t:\tau & \text{si } x_1:\tau_1 = x:\tau \\ x_1:\tau_1 & \text{en caso contrario} \end{cases}$
- $true[t:\tau/x:\tau] := true$ — $false[t:\tau/x:\tau] := false$
- $(t_1:\tau_1,\ldots,t_n:\tau_n)[t:\tau/x:\tau] := (t_1:\tau_1[t:\tau/x:\tau],\ldots,t_n:\tau_n[t:\tau/x:\tau])$
- $(\text{if } t_1:\text{bool then } t_2:\tau_1 \text{ else } t_3:\tau_1)[t:\tau/x:\tau] :=$
  $(\text{if } t_1:\text{bool}[t:\tau/x:\tau] \text{ then } t_2:\tau_1[t:\tau/x:\tau] \text{ else } t_3:\tau_1[t:\tau/x:\tau])$
- $(M\ t_1:\tau_1):\tau_2[t:\tau/x:\tau] := (M:\tau_1\rightarrow\tau_2[t:\tau/x:\tau]\ t_1:\tau_1[t:\tau/x:\tau]):\tau_2$

b2) La Σ-sustitución en una expresión funcional de una variable de dato libre por un Σ-término del mismo tipo se define mediante las reglas siguientes:

- $\perp:\tau_1\rightarrow\tau_2[t:\tau/x:\tau] := \perp:\tau_1\rightarrow\tau_2$
- $f:\tau_1\rightarrow\tau_2[t:\tau/x:\tau] := f:\tau_1\rightarrow\tau_2$
- $X:\tau_1\rightarrow\tau_2[t:\tau/x:\tau] := X:\tau_1\rightarrow\tau_2$

- $(\lambda x_1{:}\tau_1 \ldots x_n{:}\tau_n.\, t'{:}\tau')[t{:}\tau/x{:}\tau] :=$

$\qquad (\lambda z_1{:}\tau_1 \ldots z_n{:}\tau_n.\, t'{:}\tau'[z_1{:}\tau_1/x_1{:}\tau_1] \ldots [z_n{:}\tau_n/x_n{:}\tau_n][t{:}\tau/x{:}\tau])$

donde para cada $1 \le i \le n$ $z_i = x_i$ si $x_i{:}\tau_i \notin Lib(t{:}\tau)$ y $x{:}\tau \neq x_i{:}\tau_i$; en caso contrario, z_i es una variable tal que $x{:}\tau \neq z_i{:}\tau_i$ y $z_i{:}\tau_i \notin Lib(t{:}\tau) \cup Lib(t'{:}\tau')$.

- $(\mu X{:}\tau_1 \to \tau_2.\, M)[t{:}\tau/x{:}\tau] :=$

$\qquad (\mu Y{:}\tau_1 \to \tau_2.\, M{:}\tau_1 \to \tau_2[Y{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2][t{:}\tau/x{:}\tau])$

donde Y es igual a X si $X{:}\tau_1 \to \tau_2 \notin Lib(t{:}\tau)$; o $Y \in FVar(\Sigma)$ es tal que $Y{:}\tau_1 \to \tau_2 \notin Lib(M{:}\tau_1 \to \tau_2) \cup Lib(t{:}\tau)$.

b3) La Σ-sustitución en una fórmula de una variable de dato libre por un Σ-término del mismo tipo se define como sigue:

- $t_1{:}\tau_1 \subseteq t_2{:}\tau_1[t{:}\tau/x{:}\tau] := t_1{:}\tau_1[t{:}\tau/x{:}\tau] \subseteq t_2{:}\tau_1[t{:}\tau/x{:}\tau]$
- $(p\ t_1{:}\tau_1)[t{:}\tau/x{:}\tau] := (p\ t_1{:}\tau_1[t{:}\tau/x{:}\tau])$
- $(\neg\varphi)[t{:}\tau/x{:}\tau] := \neg\varphi[t{:}\tau/x{:}\tau]$
- $(\varphi \vee \psi)[t{:}\tau/x{:}\tau] := \varphi[t{:}\tau/x{:}\tau] \vee \psi[t{:}\tau/x{:}\tau]$
- $(\varphi \wedge \psi)[t{:}\tau/x{:}\tau] := \varphi[t{:}\tau/x{:}\tau] \wedge \psi[t{:}\tau/x{:}\tau]$
- $(\forall x_1{:}\tau_1\ \varphi)[t{:}\tau/x{:}\tau] := \forall z{:}\tau_1\ \varphi[z{:}\tau_1/x_1{:}\tau_1][t{:}\tau/x{:}\tau]$ donde z es igual a x_1 si $x_1{:}\tau_1 \notin Lib(t{:}\tau)$ y $x{:}\tau \neq x_1{:}\tau_1$. En caso contrario, z es un símbolo de variable tal que $x{:}\tau \neq z{:}\tau_1$ y $z{:}\tau_1 \notin Lib(\varphi) \cup Lib(t{:}\tau)$.
- $(\exists x_1{:}\tau_1\ \varphi)[t{:}\tau/x{:}\tau] := \exists z{:}\tau_1\ \varphi[z{:}\tau_1/x_1{:}\tau_1][t{:}\tau/x{:}\tau]$ con z como en el caso anterior.

c) Σ-sustitución de una variable de función por una Σ-expresión funcional.

c1) La Σ-sustitución en un término de una variable de función libre por una Σ-expresión funcional del mismo tipo se define como sigue:

- $\bot{:}\tau[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2] := \bot{:}\tau$
- $c{:}\tau[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2] := c{:}\tau$
- $x{:}\tau[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2] := x{:}\tau$
- $\mathbf{true}[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2] := \mathbf{true}$
- $\mathbf{false}[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2] := \mathbf{false}$
- $(t_1{:}\tau_1', \ldots, t_n{:}\tau_n')[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2] :=$

$\qquad (t_1{:}\tau_1'[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2], \ldots, t_n{:}\tau_n'[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2])$
- $(\mathbf{if}\ t{:}\mathbf{bool}\ \mathbf{then}\ t_1{:}\tau\ \mathbf{else}\ t_2{:}\tau)[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2] :=$

$\qquad (\mathbf{if}\ t{:}\mathbf{bool}[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2]\ \mathbf{then}\ t_1{:}\tau[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2]$

$\qquad\qquad \mathbf{else}\ t_2{:}\tau[M{:}\tau_1 \to \tau_2/X{:}\tau_1 \to \tau_2])$

- $(M\ t{:}\tau_1'){:}\tau_2'[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] :=$

  $(M{:}\tau_1'\to\tau_2'[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2]\ t{:}\tau_1'[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2]){:}\tau_2'$

c2) La Σ-sustitución en una expresión funcional de una variable de función libre por una Σ-expresión funcional del mismo tipo se define mediante las reglas siguientes:

- $\bot{:}\tau_1'\to\tau_2'[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] := \bot{:}\tau_1'\to\tau_2'$
- $f{:}\tau_1'\to\tau_2'[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] := f{:}\tau_1'\to\tau_2'$
- $X_1{:}\tau_1'\to\tau_2'[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] := \begin{cases} M{:}\tau_1\to\tau_2 & \text{si } X_1{:}\tau_1'\to\tau_2' = X{:}\tau_1\to\tau_2 \\ X_1{:}\tau_1'\to\tau_2' & \text{en otro caso} \end{cases}$
- $(\lambda x_1{:}\tau_1'\ldots x_n{:}\tau_n'.\,t{:}\tau)[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] :=$

  $(\lambda z_1{:}\tau_1'\ldots z_n{:}\tau_n'.\,t{:}\tau[z_1{:}\tau_1'/x_1{:}\tau_1']\ldots$

  $[z_n{:}\tau_n'/x_n{:}\tau_n'][M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2])$ donde para cada i, z_i es igual a x_i si $x_i{:}\tau_i' \notin Lib(M{:}\tau_1\to\tau_2)$; o z_i es un símbolo se variable tal que $z_i{:}\tau_i' \notin Lib(t{:}\tau) \cup Lib(M{:}\tau_1\to\tau_2)$ en caso contrario.
- $(\mu X'{:}\tau_1'\to\tau_2'.M')[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] :=$

  $(\mu Y{:}\tau_1'\to\tau_2'.M'{:}\tau_1'\to\tau_2'[Y{:}\tau_1'\to\tau_2'/X'{:}\tau_1'\to\tau_2'][M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2])$

donde Y es igual a X' si $X'{:}\tau_1'\to\tau_2' \notin Lib(M{:}\tau_1\to\tau_2)$. En caso contrario, Y es un símbolo de variable de función tal que $Y{:}\tau_1'\to\tau_2' \notin Lib(M{:}\tau_1\to\tau_2) \cup Lib(M'{:}\tau_1'\to\tau_2')$.

c3) La Σ-sustitución en una fórmula de una variable de función libre por una Σ-expresión funcional del mismo tipo se define como sigue:

- $t_1{:}\tau \subseteq t_2{:}\tau[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] :=$

  $t_1{:}\tau[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] \subseteq t_2{:}\tau[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2]$
- $(p\ t{:}\tau)[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] := (p\ t{:}\tau[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2])$
- $(\neg\varphi)[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] := \neg\varphi[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2]$
- $(\varphi \vee \psi)[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] := \varphi[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2]\vee\psi[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2]$
- $(\varphi \wedge \psi)[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] := \varphi[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2]\wedge\psi[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2]$
- $(\forall x{:}\tau\ \varphi)[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] := \forall z{:}\tau\ \varphi[z{:}\tau/x{:}\tau][M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2]$

donde z es igual a x si $x{:}\tau \notin Lib(M{:}\tau_1\to\tau_2)$, y z es una variable tal que $z{:}\tau \notin Lib(\varphi) \cup Lib(M{:}\tau_1\to\tau_2)$ en caso contrario.

- $(\exists x{:}\tau\ \varphi)[M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2] := \exists z{:}\tau\ \varphi[z{:}\tau/x{:}\tau][M{:}\tau_1\to\tau_2/X{:}\tau_1\to\tau_2]$

con z como en el caso anterior. □

2. LA SEMANTICA DE PLPR

A continuación se propone una semántica denotacional para PLPR, dando una denotación a los objetos del lenguaje y se definen los conceptos de válidez, satisfactibilidad, consecuencia lógica, etc. La semántica propuesta refleja el hecho de que nuestro lenguaje permite la definición de funciones recursivas mediante un operador de punto fijo. En el primer apartado de esta sección se enuncian algunas definiciones y resultados conocidos sobre funciones continuas, puntos fijos y dominios que serán utilizados para demostrar que el concepto de interpretación de términos y expresiones funcionales está bien definido. Por último, se demuestran distintos resultados semánticos que serán utilizados en capítulos posteriores.

2.1 CONCEPTOS PREVIOS

Las definiciones y resultados presentados aquí, propios de las teorías de dominios y de punto fijo, pueden verse con más detalle en [Sto-77], [Sco-82] y [Ber-85], por ejemplo.

Definición 2.1.1

Consideremos los conjuntos D, $D_1,\ldots,D_n$ con los órdenes parciales $\sqsubseteq$, $\sqsubseteq_1,\ldots,\sqsubseteq_n$ respectivamente.

Se dice que x es el *elemento mínimo* de D si $x \in D$ y para cada y de D, $x \sqsubseteq y$.

Se dice que $x \in D$ es una *cota superior* de un subconjunto A de D si para cada y de A se verifica $y \sqsubseteq x$.

La *menor cota superior* de $A \subseteq D$, escrito $\sqcup A$, es el elemento mínimo del conjunto de cotas superiores de A. (Si $A = \{x_0, x_1, x_2,\ldots\}$ denotamos $\sqcup A$ mediante $\sqcup_{n=0}^{\infty} x_n$).

Una *cadena creciente* en D es un subconjunto $\{x_0, x_1, x_2,\ldots\}$ de D que verifica: $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$

Se dice que D es un *orden parcial completo* (cpo) si tiene un elemento mínimo $\perp$ (llamado *bottom*) y toda cadena creciente en D tiene menor cota superior.

Una función $f: D_1 \rightarrow D_2$ se dice *monótona* cuando dados $x, y \in D_1$ tales que $x \sqsubseteq_1 y$, entonces $f(x) \sqsubseteq_2 f(y)$.

Una función monótona $f: D_1 \to D_2$ es *continua* cuando para toda cadena de D: $x_0 \subseteq x_1 \subseteq \ldots$ se tiene $f(\sqcup_{n=0}^{\infty} x_n) = \sqcup_{n=0}^{\infty} f(x_n)$.

Una función $f: D_1 \to D_2$ se dice *estricta* cuando $f(\perp_1) = \perp_2$.

Un cpo D se dice que es *plano* cuando para todos x, y $\in$ D tales que $x \subseteq y$ se verifica $x = y$ ó $x = \perp$.

Dados los cpo's planos $\langle D_i, \subseteq_i \rangle$, $1 \le i \le n$, definimos el producto cartesiano estricto $\langle D_1 \otimes \ldots \otimes D_n, \subseteq_\otimes \rangle$ donde el orden $\subseteq_\otimes$ definido sobre $D_1 \otimes \ldots \otimes D_n = \{\perp_\otimes\} \cup \{\langle d_1, \ldots, d_n \rangle \mid d_i \in D_i \setminus \{\perp_i\}\}$ es tal que: $\bar{d}_1 \subseteq_\otimes \bar{d}_2 \iff \bar{d}_1 = \bar{d}_2$ ó $\bar{d}_1 = \perp_\otimes$ para todo $\bar{d}_1, \bar{d}_2 \in D_1 \otimes \ldots \otimes D_n$. El producto estricto identifica toda n-tupla que tenga alguno de sus argumentos igual a $\perp_i$ con $\perp_\otimes$; de esta manera $D_1 \otimes \ldots \otimes D_n$ resulta ser un cpo plano.

Definimos el *espacio de funciones continuas* (*espacio de funciones estrictas*) entre los cpo's planos $\langle D_1, \subseteq_1 \rangle$, $\langle D_2, \subseteq_2 \rangle$ representado por $\langle [D_1 \xrightarrow[c]{} D_2], \subseteq_c \rangle$ ($\langle [D_1 \xrightarrow[e]{} D_2], \subseteq_e \rangle$) como el orden $\subseteq_c$ ($\subseteq_e$) definido sobre el conjunto de funciones continuas (estrictas) de D_1 en D_2 de la siguiente forma. Dados f, g $\in [D_1 \xrightarrow[c]{} D_2]$ (f, g $\in [D_1 \xrightarrow[e]{} D_2]$) para todo x de D_1 $f \subseteq_c g$ ($f \subseteq_e g$) $\iff f(x) \subseteq_2 g(x)$.

Identificaremos el *cpo plano de los booleanos*, $\mathbb{B}$, con cualquier dominio isomorfo al conjunto $\{\perp_{bool}, tt, ff\}$ con el orden siguiente: $\perp_{bool} \subseteq \perp_{bool}$, $\perp_{bool} \subseteq tt$, $\perp_{bool} \subseteq ff$.

Lema 2.1.2

Toda función f $\in [D_1 \xrightarrow[e]{} D_2]$ pertenece también a $[D_1 \xrightarrow[c]{} D_2]$.

La demostración de este lema es trivial pues si D_1 es un cpo plano, todas las cadenas crecientes en D_1 están formadas a lo sumo por $\perp_1$ y un elemento definido.

Teorema 2.1.3

a) La composición de funciones continuas es continua.

b) La función constante y la función identidad son continuas.

c) Dado un cpo (D, $\subseteq$), la función COND: $\mathbb{B} \times D \times D \to D$ definida como sigue $COND(tt, a, b) = a$, $COND(ff, a, b) = b$, $COND(\perp_{\mathbb{B}}, a, b) = \perp$ es continua en todos sus argumentos.

Demostración:

Detallamos la demostración de c)

Por construcción, la función COND es estricta en su primer argumento luego es continua. Para el segundo argumento se tiene:

$COND(tt, \sqcup_{i=0}^{\infty} a_i, b) = \sqcup_{i=0}^{\infty} a_i$ y $COND(tt, a_i, b) = a_i$ para todo $i < \omega$

luego $\sqcup_{i=0}^{\omega} COND(tt, a_i, b) = \sqcup_{i=0}^{\omega} a_i = COND(tt, \sqcup_{i=0}^{\omega} a_i, b)$.

$COND(ff, \sqcup_{i=0}^{\omega} a_i, b) = b$ y $COND(ff, a_i, b) = b$ para todo $i<\omega$ luego $\sqcup_{i=0}^{\omega} COND(tt, a_i, b) = b = COND(tt, \sqcup_{i=0}^{\omega} a_i, b)$.

$COND(\perp_{bool}, \sqcup_{i=0}^{\omega} a_i, b) = \perp$ y $COND(\perp_{bool}, a_i, b) = \perp$ para todo $i<\omega$ luego $\sqcup_{i=0}^{\omega} COND(\perp_{bool}, a_i, b) = \perp = COND(\perp_{bool}, \sqcup_{i=0}^{\omega} a_i, b)$.

La demostración de la continuidad para el tercer argumento es análoga. ■

Teorema 2.1.4 (Existencia del menor punto fijo)

Dado un cpo $<D, \leq>$, toda función continua $f:D \to D$ tiene un menor punto fijo, es decir, existe un $x \in D$ tal que $f(x) = x$ de manera que para todo $y \in D$ si $f(y) = y$ entonces $x \leq y$. Además dicho punto fijo es de la forma: $\sqcup_{i=0}^{\omega} f^i(\perp)$, siendo $f^i = \underbrace{f \circ \ldots \circ f}_{i \text{ veces}}$.

Teorema 2.1.5 (Continuidad del operador de punto fijo)

Dado un cpo $(D, \leq)$, el funcional $fix: [D \xrightarrow[c]{} D] \to D$ definido por $fix(f)$ = menor punto fijo de f para toda $f \in [D \xrightarrow[c]{} D]$ es continuo.

2.2 ESTRUCTURAS, INTERPRETACIONES Y MODELOS

Los dominios de interpretación utilizados para definir los modelos en PLPR son familias de cpo's planos, de manera que cada tipo denota un elemento de la familia de cpo's correspondiente, siendo el dominio de interpretación de los términos la unión de los cpo's que la componen, y el dominio de interpretación de las expresiones funcionales la unión de los espacios de funciones estrictas entre dichos cpo's.

Definición 2.2.1

Una *Σ-estructura de tipos* es un par $\mathcal{I} = <T^{\mathcal{I}}, \{ct^{\mathcal{I}}\}_{ct/n \in \Sigma_t}>$ donde $T^{\mathcal{I}}$ es una familia de cpo's planos cerrada bajo el producto estricto que contiene a $\mathbb{B}$, y por otro lado, $ct^{\mathcal{I}} \in T^{\mathcal{I}}$ si $ct/0 \in \Sigma_t$ y $ct^{\mathcal{I}}$ es una aplicación de $(T^{\mathcal{I}})^n$ en $T^{\mathcal{I}}$ si $ct/n \in \Sigma_t$ $(n \geq 1)$.

Dada una Σ-estructura de tipos $\mathcal{I}$, se define una *asignación de las variables de tipo para* $\mathcal{I}$ como una función η de TVar en $T^{\mathcal{I}}$. □

Hablaremos simplemente de asignación de tipos si no existe

ambigüedad con respecto a la estructura de tipos a la que se refiere. El conjunto $\{\eta \mid \eta: TVar \to T^{\mathcal{I}}\}$ de todas las asignaciones para una Σ-estructura de tipos $\mathcal{I}$ se denota $AT_p^{\mathcal{I}}$.

Definición 2.2.2

Sea $\mathcal{I}$ una Σ-estructura de tipos. Definimos la Σ-*Interpretación de un tipo* $\tau \in T_p(\Sigma)$ *con respecto a* $\mathcal{I}$, escrito $[\![\tau]\!]^{\mathcal{I}}$, como una función que a cada asignación para $\mathcal{I}$ le hace corresponder un cpo plano perteneciente a la familia $T^{\mathcal{I}}$. Definimos esta función por inducción sobre la construcción de los Σ-tipos de la siguiente manera:

$[\![\rho]\!]^{\mathcal{I}}\eta := \eta(\rho)$

$[\![ct]\!]^{\mathcal{I}}\eta := ct^{\mathcal{I}}$ para toda $\eta \in AT_p^{\mathcal{I}}$

$[\![bool]\!]^{\mathcal{I}}\eta := \mathbb{B}$ para toda $\eta \in AT_p^{\mathcal{I}}$

$[\![ct(\tau_1,\ldots,\tau_n)]\!]^{\mathcal{I}}\eta := ct^{\mathcal{I}}([\![\tau_1]\!]^{\mathcal{I}}\eta,\ldots,[\![\tau_n]\!]^{\mathcal{I}}\eta)$

$[\![\tau_1 x \ldots x \tau_n]\!]^{\mathcal{I}}\eta := [\![\tau_1]\!]^{\mathcal{I}}\eta \otimes \ldots \otimes [\![\tau_n]\!]^{\mathcal{I}}\eta$.

Se define la Σ-*Interpretación de un* Σ-*tipo de función* $\tau_1 \to \tau_2$ *con respecto a* $\mathcal{I}$ como una función que a cada asignación le hace corresponder un espacio de funciones estrictas definido como sigue:

$[\![\tau_1 \to \tau_2]\!]^{\mathcal{I}}\eta := [\,[\![\tau_1]\!]^{\mathcal{I}}\eta \xrightarrow[\bullet]{} [\![\tau_2]\!]^{\mathcal{I}}\eta\,]$. □

Es fácil comprobar que efectivamente $[\![\tau]\!]^{\mathcal{I}}\eta \in T^{\mathcal{I}}$ para todo Σ-tipo τ y toda asignación η para $\mathcal{I}$. Al menor elemento del cpo $[\![\tau]\!]^{\mathcal{I}}\eta$ lo denotaremos mediante $\perp^{\mathcal{I}}_{\eta(\tau)}$ escribiremos su orden como $\sqsubseteq^{\mathcal{I}}_{\eta(\tau)}$, análogamente, $\perp^{\mathcal{I}}_{\eta(\tau_1 \to \tau_2)}$ representa el *bottom* del cpo $[\,[\![\tau_1]\!]^{\mathcal{I}}\eta \xrightarrow[\bullet]{} [\![\tau_2]\!]^{\mathcal{I}}\eta\,]$. En todos los casos, se puede eliminar la notación del superíndice $\mathcal{I}$ si no existe anbigüedad.

De la definición de interpretación de tipos se deduce que si ν es un tipo monomórfico, $[\![\nu]\!]^{\mathcal{I}}$ es una función constante que asocia a todo η de $AT_p^{\mathcal{I}}$ un dominio que llamamos $D^{\mathcal{I}}_{\nu}$.

Para cada Σ-estructura de tipos $\mathcal{I}$ la *colección de dominios asociados a los tipos monomórficos de primer orden* es la familia de cpo's planos $\{\langle D^{\mathcal{I}}_{\nu}, \sqsubseteq^{\mathcal{I}}_{\nu}\rangle\}_{\nu \in MT_p(\Sigma)}$. Claramente verifican:

$D^{\mathcal{I}}_{ct} = ct^{\mathcal{I}}$

$D^{\mathcal{I}}_{bool} = \mathbb{B}$

$D^{\mathcal{I}}_{ct(\nu_1,\ldots,\nu_n)} = ct^{\mathcal{I}}(D^{\mathcal{I}}_{\nu_1},\ldots,D^{\mathcal{I}}_{\nu_n})$

$D^{\mathcal{I}}_{\nu_1 x \ldots x \nu_n} = D^{\mathcal{I}}_{\nu_1} \otimes \ldots \otimes D^{\mathcal{I}}_{\nu_n}$.

Lema 2.2.3

Sean Σ una signatura, $\mathcal{I}$ una Σ-estructura de tipos, η una valoración para $\mathcal{I}$ y σ la Σ-sustitución de tipos $\tau_1 \ldots \tau_n / \rho_1 \ldots \rho_n$. La función $\eta\left\{\frac{[\![\tau_1]\!]^{\mathcal{I}}\eta \ldots [\![\tau_n]\!]^{\mathcal{I}}\eta}{\rho_1 \ldots \rho_n}\right\}(\rho) = \begin{cases} [\![\tau_1]\!]^{\mathcal{I}}\eta & \text{si } \rho = \rho_1 \\ \eta(\rho) & \text{si } \rho \neq \rho_1, \ldots, \rho_n \end{cases}$ es una asignación de tipos para $\mathcal{I}$, que simplificamos con la notación $\eta\langle\sigma\rangle$, y verifica:

$[\![\tau\sigma]\!]^{\mathcal{I}}\eta = [\![\tau]\!]^{\mathcal{I}}\eta\langle\sigma\rangle$ para cualquier Σ-tipo τ.

Demostración:

De la definición de $\eta\langle\sigma\rangle$ se sigue directamente que dicha función es una valoración para $\mathcal{I}$. La igualdad del teorema se prueba por casos. Detallamos la demostración para cuando τ es una variable de tipo ρ, los casos constantes son triviales por definición de sustitución, y para el resto de los tipos de primer orden se aplica inducción en la construcción de los mismos:

$$[\![\rho\sigma]\!]^{\mathcal{I}}\eta = \begin{cases} [\![\tau_1]\!]^{\mathcal{I}}\eta & \text{si } \rho = \rho_1 \\ [\![\rho]\!]^{\mathcal{I}}\eta = \eta(\rho) & \text{si } \rho \neq \rho_1, \ldots, \rho_n \end{cases} = \eta\langle\sigma\rangle(\rho) = [\![\rho]\!]^{\mathcal{I}}\eta\langle\sigma\rangle$$

Para el caso del tipo función, se tiene igualmente:

$[\![(\tau_1 \to \tau_2)\sigma]\!]^{\mathcal{I}}\eta = [\,[\![\tau_1\sigma]\!]^{\mathcal{I}}\eta \underset{\bullet}{\longrightarrow} [\![\tau_2\sigma]\!]^{\mathcal{I}}\eta\,]$

$= [\,[\![\tau_1]\!]^{\mathcal{I}}\eta\langle\sigma\rangle \underset{\bullet}{\longrightarrow} [\![\tau_2]\!]^{\mathcal{I}}\eta\langle\sigma\rangle\,]$ por hipótesis de inducción

$= [\![\tau_1 \to \tau_2]\!]^{\mathcal{I}}\eta\langle\sigma\rangle$. ∎

El lema anterior presenta gran analogía con el lema de sustitución de términos de la lógica de predicados. De hecho en PLPR, los tipos de primer orden son los objetos correspondientes a los términos de aquella lógica.

Definición 2.2.4

Dada una Σ-estructura de tipos $\mathcal{I} = \langle T^{\mathcal{I}}, \{ct^{\mathcal{I}}\}_{ct/n \in \Sigma_t}\rangle$, se define el *dominio de datos asociado* a $\mathcal{I}$, $DAT^{\mathcal{I}}$, como la union de los cpo's pertenecientes a $T^{\mathcal{I}}$, es decir, $DAT^{\mathcal{I}} = \bigcup \{D \mid D \in T^{\mathcal{I}}\}$. De manera similar, se define el *dominio de funciones asociado* a $\mathcal{I}$ como el conjunto $FUN^{\mathcal{I}} = \bigcup \{[D_1 \underset{\bullet}{\longrightarrow} D_2] \mid D_1, D_2 \in T^{\mathcal{I}}\}$. □

Definición 2.2.5

Sea $\mathcal{I}$ una Σ-estructura de tipos. Una *Σ-estructura de datos*, $\mathcal{D}$,

con respecto a I viene definida por:

$$\mathcal{D} = \langle \{c:\tau^{\mathcal{D}}\}_{c:\rightarrow\tau\in\Sigma_d},\ \{f:\tau_1\rightarrow\tau_2^{\mathcal{D}}\}_{f:\tau_1\rightarrow\tau_2\in\Sigma_d},\ \{p:\tau^{\mathcal{D}}\}_{p:\tau\in\Sigma_d} \rangle$$

Donde los elementos de $\mathcal{D}$ son funciones definidas sobre el conjunto de asignaciones para I que devuelven valores en los siguientes conjuntos:

$$c:\tau^{\mathcal{D}}: AT_p^{\mathcal{I}} \rightarrow DAT^{\mathcal{I}},\ f:\tau_1\rightarrow\tau_2^{\mathcal{D}}: AT_p^{\mathcal{I}} \rightarrow FUN^{\mathcal{I}},\ p:\tau^{\mathcal{D}}: AT_p^{\mathcal{I}} \rightarrow \mathcal{P}(DAT^{\mathcal{I}})$$

siendo $\mathcal{P}(D)$ el conjunto formado por las partes del conjunto D. Además, estas funciones son tales que:

- Para cada $\eta \in AT_p^{\mathcal{I}}$:

  $c:\tau^{\mathcal{D}}\eta \in [\![\tau]\!]^{\mathcal{I}}\eta$, $f:\tau_1\rightarrow\tau_2^{\mathcal{D}}\eta \in [[\![\tau_1]\!]^{\mathcal{I}}\eta \xrightarrow[c]{} [\![\tau_2]\!]^{\mathcal{I}}\eta]$, $p:\tau^{\mathcal{D}}\eta \subseteq [\![\tau]\!]^{\mathcal{I}}\eta$ y $\perp^{\mathcal{I}}_{\eta(\tau)} \notin p:\tau^{\mathcal{D}}\eta$.
- Si η y η' coinciden en las variables de tipo de τ entonces $c:\tau^{\mathcal{D}}\eta = c:\tau^{\mathcal{D}}\eta'$.

  Si η y η' coinciden en las variables de tipo de τ_1 y τ_2, entonces $f:\tau_1\rightarrow\tau_2^{\mathcal{D}}\eta = f:\tau_1\rightarrow\tau_2^{\mathcal{D}}\eta'$.

  Si η y η' coinciden en las variables de tipo de τ entonces $p:\tau^{\mathcal{D}}\eta = p:\tau^{\mathcal{D}}\eta'$. □

La primera condición restringe la imagen de manera que para una asignación η, los símbolos de constante y función representen un elemento del cpo denotado por su tipo para dicha asignación, y un símbolo de predicado represente un subconjunto del cpo denotado por su tipo que no contenga el *bottom*. La última condición se impone para conseguir que la semántica de las fórmulas sea uniforme con respecto a las variables de tipo que en ella aparecen.

Definición 2.2.6

Sea I una Σ-estructura de tipos. Una *valoración con respecto a* I es una función $\xi = \xi^1 \cup \xi^2$ donde:

$\xi^1: \{x:\tau \mid x \in Var_\tau,\ \tau \in T_p(\Sigma)\} \rightarrow AT_p^{\mathcal{I}} \rightarrow DAT^{\mathcal{I}}$

$\xi^2: \{X:\tau_1\rightarrow\tau_2 \mid X \in FVar_{\tau_1\rightarrow\tau_2},\ \tau_1,\tau_2 \in T_p(\Sigma)\} \rightarrow AT_p^{\mathcal{I}} \rightarrow FUN^{\mathcal{I}}$ y verifican:

- $\xi^1(x:\tau)\eta \in [\![\tau]\!]^{\mathcal{I}}\eta$ y $\xi^2(X:\tau_1\rightarrow\tau_2)\eta \in [\![\tau_1\rightarrow\tau_2]\!]^{\mathcal{I}}\eta$
- $\xi^1(x:\tau\sigma)\eta = \xi^1(x:\tau)\eta\{\sigma\}$ y $\xi^2(X:\tau_1\sigma\rightarrow\tau_2\sigma)\eta = \xi^2(X:\tau_1\rightarrow\tau_2)\eta\{\sigma\}$. □

Las retricciones impuestas en la definición anterior son lógicas puesto que una variable debe denotar un valor del cpo denotado por su tipo y además se ha probado que $[\![\tau\sigma]\!]^{\mathcal{I}}\eta = [\![\tau]\!]^{\mathcal{I}}\eta\{\sigma\}$.

En lo que sigue se ve como la interpretación de un término y de una expresión funcional viene dada en relación a los valores asignados a las variables de tipo que en él o ella aparecen.

Definición 2.2.7

Sean $\mathcal{I}$ una Σ-estructura de tipos, $\mathcal{D}$ una Σ-estructura de datos con respecto a $\mathcal{I}$ y ξ una valoración con respecto a $\mathcal{I}$. Una *Σ-interpretación* $\mathcal{J}$ es una terna $\mathcal{J} = \langle \mathcal{I}, \mathcal{D}, \xi \rangle$ y una *Σ-estructura* $\mathfrak{A}$ es un par $\mathfrak{A} = \langle \mathcal{I}, \mathcal{D} \rangle$.

La *Σ-interpretación de un término* $t:\tau$ *con respecto a* una *Σ-interpretación* $\mathcal{J}$ es una función $\mathcal{J}[\![t:\tau]\!]: AT_p^{\mathcal{I}} \to DAT^{\mathcal{I}}$ y la *Σ-interpretación de una expresión funcional* $M:\tau_1 \to \tau_2$ *con respecto a* $\mathcal{J}$ es una función $\mathcal{J}[\![M:\tau_1 \to \tau_2]\!]: AT_p^{\mathcal{I}} \to FUN^{\mathcal{I}}$ que se definen por recursión mutua:

$\mathcal{J}[\![\bot:\tau]\!]\eta := \bot^{\mathcal{I}}_{\eta(\tau)}$

$\mathcal{J}[\![x:\tau]\!]\eta := \xi(x:\tau)\eta$

$\mathcal{J}[\![c:\tau']\!]\eta := c:\tau^{\mathcal{D}}\eta(\sigma)$, donde $c:\to\tau \in \Sigma_d$ y $\tau' = \tau\sigma$

$\mathcal{J}[\![true]\!]\eta := tt$

$\mathcal{J}[\![false]\!]\eta := ff$

$$\mathcal{J}[\![(t_1:\tau_1,\ldots,t_n:\tau_n)]\!]\eta := \begin{cases} \langle \mathcal{J}[\![t_1:\tau_1]\!]\eta, \ldots, \mathcal{J}[\![t_n:\tau_n]\!]\eta \rangle \\ \qquad \text{si } \mathcal{J}[\![t_1:\tau_1]\!]\eta \neq \bot_{\eta(\tau_1)} \text{ p.t. } 1 \le i \le n \\ \bot^{\mathcal{I}}_{\eta(\tau_1 \times \ldots \times \tau_n)} \qquad \text{en otro caso} \end{cases}$$

$$\mathcal{J}[\![(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)]\!]\eta := \begin{cases} \mathcal{J}[\![t_1:\tau]\!]\eta & \text{si } \mathcal{J}[\![t:\text{bool}]\!]\eta = tt \\ \mathcal{J}[\![t_2:\tau]\!]\eta & \text{si } \mathcal{J}[\![t:\text{bool}]\!]\eta = ff \\ \bot^{\mathcal{I}}_{\eta(\tau)} & \text{si } \mathcal{J}[\![t:\text{bool}]\!]\eta = \bot_{bool} \end{cases}$$

$\mathcal{J}[\![(M\ t:\tau_1):\tau_2]\!]\eta := \mathcal{J}[\![M:\tau_1 \to \tau_2]\!]\eta\ (\mathcal{J}[\![t:\tau_1]\!]\eta)$

$\mathcal{J}[\![\bot:\tau_1 \to \tau_2]\!]\eta := \bot_{\eta(\tau_1 \to \tau_2)}$

$\mathcal{J}[\![f:\tau_1' \to \tau_2']\!]\eta := f:\tau_1 \to \tau_2^{\mathcal{D}}\eta(\sigma)$, donde $f:\tau_1 \to \tau_2 \in \Sigma_d$, $\tau_1' \to \tau_2' = \tau_1\sigma \to \tau_2\sigma$

$\mathcal{J}[\![X:\tau_1 \to \tau_2]\!]\eta := \xi(X:\tau_1 \to \tau_2)\eta$

$\mathcal{J}[\![(\lambda x_1:\tau_1 \ldots x_n:\tau_n.\, t:\tau)]\!]\eta \in [[\tau_1]^{\mathcal{I}}\eta \bullet \ldots \bullet [\tau_n]^{\mathcal{I}}\eta \xrightarrow{\bullet} [\tau]^{\mathcal{I}}\eta]$ y se define

$$\mathcal{J}[\![(\lambda x_1:\tau_1 \ldots x_n:\tau_n.\, t:\tau)]\!]\eta(d_1,\ldots,d_n) = \mathcal{J}\left\{\frac{d_1 \ldots d_n}{x_1:\tau_1 \ldots x_n:\tau_n}\right\}[\![t:\tau]\!]\eta$$

donde $\mathcal{J}\left\{\frac{d_1 \ldots d_n}{x_1:\tau_1 \ldots x_n:\tau_n}\right\}$ coincide con $\mathcal{J}$ salvo en $x_1:\tau_1,\ldots,x_n:\tau_n$ y η siendo $\mathcal{J}\left\{\frac{d_1 \ldots d_n}{x_1:\tau_1 \ldots x_n:\tau_n}\right\}[\![x_i:\tau_i]\!]\eta = d_i$ para $i=1,\ldots,n$.

$\mathfrak{I}[(\mu X{:}\tau_1{\to}\tau_2.M)]\eta$ = fix $T(\mathfrak{I}, X{:}\tau_1{\to}\tau_2, M{:}\tau_1{\to}\tau_2, \eta)$, es decir, el menor punto fijo del operador $T(\mathfrak{I}, X{:}\tau_1{\to}\tau_2, M{:}\tau_1{\to}\tau_2, \eta)$ definido a continuación:

$T(\mathfrak{I},\ X{:}\tau_1{\to}\tau_2,\ M{:}\tau_1{\to}\tau_2,\ \eta)$: $([\tau_1]^{\mathfrak{I}}\eta \xrightarrow[\bullet]{} [\tau_2]^{\mathfrak{I}}\eta) \to ([\tau_1]^{\mathfrak{I}}\eta \xrightarrow[\bullet]{} [\tau_2]^{\mathfrak{I}}\eta)$

$T(\mathfrak{I},\ X{:}\tau_1{\to}\tau_2,\ M{:}\tau_1{\to}\tau_2,\ \eta)(h)$: = $\mathfrak{I}\left\{\frac{h}{X{:}\tau_1{\to}\tau_2}\right\}[M{:}\tau_1{\to}\tau_2]\eta$

Donde $\mathfrak{I}\left\{\frac{h}{X{:}\tau_1{\to}\tau_2}\right\}$ es una Σ-interpretación que coincide con $\mathfrak{I}$ salvo en $X{:}\tau_1{\to}\tau_2$ y η, siendo $\mathfrak{I}\left\{\frac{h}{X{:}\tau_1{\to}\tau_2}\right\}[X{:}\tau_1{\to}\tau_2]\eta = h$. □

Si $t{:}\nu$ y $M{:}\nu_1{\to}\nu_2$ son monomórficos entonces $\mathfrak{I}[t{:}\nu]$ e $\mathfrak{I}[M{:}\nu_1{\to}\nu_2]$ son funciones constantes. Denotamos por $\mathfrak{I}[t{:}\nu]$ e $\mathfrak{I}[M{:}\nu_1{\to}\nu_2]$ respectivamente al valor de dichas funciones, es decir $\mathfrak{I}[t{:}\nu]\eta = \mathfrak{I}[t{:}\nu]$ y $\mathfrak{I}[M{:}\nu_1{\to}\nu_2]\eta = \mathfrak{I}[M{:}\nu_1{\to}\nu_2]$ para toda asignación η.

Definición 2.2.8

Sean $\mathfrak{I}=\langle \mathfrak{I}, \mathfrak{D}, \xi\rangle$ una Σ-interpretación, η una asignación para $\mathfrak{I}$, y φ una Σ-fórmula. La relación $\mathfrak{I}$ *satisface* φ *con respecto a la asignación* η, escrito $\mathfrak{I}, \eta \models \varphi$, se define por recursión sobre la construcción de las fórmulas como sigue:

$\mathfrak{I}, \eta \models t_1{:}\tau \subseteq t_2{:}\tau$	$\Leftrightarrow \mathfrak{I}[t_1{:}\tau]\eta \subseteq^{\mathfrak{I}}_{\eta(\tau)} \mathfrak{I}[t_2{:}\tau]\eta$.
$\mathfrak{I}, \eta \models (p\ t{:}\tau')$	$\Leftrightarrow \mathfrak{I}[t{:}\tau']\eta \in p{:}\tau^{\mathfrak{D}}\eta(\sigma)$ siendo $p{:}\tau \in \Sigma_d$ y $\tau'=\tau\sigma$.
$\mathfrak{I}, \eta \models \neg\, \varphi$	$\Leftrightarrow$ no $\mathfrak{I}, \eta \models \varphi$.
$\mathfrak{I}, \eta \models \varphi \vee \psi$	$\Leftrightarrow \mathfrak{I}, \eta \models \varphi$ ó $\mathfrak{I}, \eta \models \psi$.
$\mathfrak{I}, \eta \models \varphi \wedge \psi$	$\Leftrightarrow \mathfrak{I}, \eta \models \varphi$ y $\mathfrak{I}, \eta \models \psi$.
$\mathfrak{I}, \eta \models \exists x{:}\tau\ \varphi$	$\Leftrightarrow$ existe un $a \in [\tau]^{\mathfrak{I}}\eta$ tal que $\mathfrak{I}(a/x{:}\tau), \eta \models \varphi$.
$\mathfrak{I}, \eta \models \forall x{:}\tau\ \varphi$	$\Leftrightarrow$ para todo $a \in [\tau]^{\mathfrak{I}}\eta$, $\mathfrak{I}(a/x{:}\tau), \eta \models \varphi$.

Si $Lib(\varphi) = \varnothing$ la relación $\mathfrak{I}=\langle \mathfrak{I}, \mathfrak{D}, \xi\rangle$ satisface φ con respecto a η que acabamos de definir no depende de la valoración ξ, es por ello que en estos casos podemos decir simplemente que la estructura $\mathfrak{A}=\langle \mathfrak{I}, \mathfrak{D}\rangle$ satisface φ con la asignación η y escribimos $\mathfrak{A}, \eta \models \varphi$. □

Definición 2.2.9

Se dice que una Σ-interpretación $\mathfrak{I} = \langle \mathfrak{I}, \mathfrak{D}, \xi\rangle$ es un *modelo de* φ *en* PLPR y se escribe $\mathfrak{I} \models \varphi$ cuando $\mathfrak{I}, \eta \models \varphi$ para cualquier asignación η para $\mathfrak{I}$. Se dice una que Σ-*estructura* $\mathfrak{A}=\langle \mathfrak{I}, \mathfrak{D}\rangle$ *es un modelo de* φ ($\mathfrak{A} \models \varphi$)

cuando $\langle \Sigma, \mathcal{D}, \xi \rangle, \eta \models \varphi$ para cualquier valoración ξ con respecto a Σ y cualquier asignación η para Σ.

Se dice que φ es *satisfactible en* PLPR si existe una Σ-interpretación que es un modelo de φ, en caso contrario φ es *insatisfactible*. Esta definición se extiende a la satisfactibilidad de un conjunto de fórmulas de la manera habitual, de forma que un conjunto de Σ-fórmulas Φ es *satisfactible en* PLPR, escrito $Sat(\Phi)$, si existe una Σ-interpretación que es un modelo de φ para todo $\varphi \in \Phi$, en caso contrario Φ es *insatisfactible* (no $Sat(\Phi)$).

Una Σ-fórmula φ es *válida en* PLPR cuando toda Σ-interpretación es un modelo de φ.

Dadas dos Σ-fórmulas φ y ψ se dice que ψ es *consecuencia de* φ *en* PLPR, escrito $\varphi \models \psi$, cuando todo modelo de φ lo es también de ψ. Una Σ-fórmula ψ es *consecuencia en* PLPR *de un conjunto de* Σ*-fórmulas* Φ y se escribe $\Phi \models \psi$, cuando toda Σ-interpretación $\mathfrak{I}$ que verifique $\mathfrak{I} \models \varphi$ para toda fórmula $\varphi \in \Phi$ es un modelo de ψ. □

La idea que se desprende de la definición de modelo es, sencillamente, que una una fórmula polimórfica será implícitamente considerada como la cuantificación universal de las variables de tipo que aparecen en su interior.

2.3 CORRECCION DE LA DEFINICION DE INTERPRETACION

Para que la definición de interpretación dada sea correcta hay que asegurar la existencia del punto fijo del operador $T(\mathfrak{I}, X{:}\tau_1 \to \tau_2, M{:}\tau_1 \to \tau_2, \eta)$ asociado a la interpretación del μ-operador. Si demostramos que T es continuo, el teorema de existencia del menor punto fijo nos asegura los resultados esperados.

Al demostrar el teorema 2.3.2 probaremos la continuidad de T y garantizaremos que la semántica de PLPR está bien definida, en esta demostración usaremos un lema de coincidencia que probaremos en 2.3.1 y que será también usado posteriormente en numerosas ocasiones.

Lema 2.3.1 (Coincidencia)

Consideremos las signaturas $\Sigma_i = \langle \Sigma_{t_i}, \Sigma_{d_i} \rangle$ y las Σ_i-interpretaciones $\mathfrak{I}_i = \langle \Sigma_i, \mathcal{D}_i, \xi_i \rangle$, $i = 1,2$, siendo $T^{\Sigma_1} = T^{\Sigma_2}$. Sea $\Sigma =$

$\langle \Sigma_t, \Sigma_d \rangle$ donde $\Sigma_t = \Sigma_{t1} \cap \Sigma_{t2}$ y $\Sigma_d = \Sigma_{d1} \cap \Sigma_{d2}$. Se dice que $\mathcal{I}_1$ y $\mathcal{I}_2$ coinciden en los Σ-símbolos de tipo de τ (de $t:\tau$, de $M:\tau_1 \to \tau_2$, de φ) si $ct^{\Sigma_1} = ct^{\Sigma_2}$ para todo $ct/n \in \Sigma_t$ que aparece en τ (en $t:\tau$, en $M:\tau_1 \to \tau_2$, en φ). Se dice que $\mathcal{I}_1$ y $\mathcal{I}_2$ coinciden en los Σ-símbolos de dato de $t:\tau$ (de $M:\tau_1 \to \tau_2$, de φ) si $k:\tau^{D_1} = k:\tau^{D_2}$ para cualquier $k:\tau'$ que aparece en $t:\tau$ (en $M:\tau_1 \to \tau_2$, en φ), tal que $k:\tau \in \Sigma_d$ y $\tau' \leq \tau$. Se dice que $\mathcal{I}_1$ y $\mathcal{I}_2$ coinciden en las variables libres de $t:\tau$ (de $M:\tau_1 \to \tau_2$, de φ) si $\xi^1_1(x:\tau') = \xi^1_2(x:\tau')$ para cualquier $x:\tau' \in Lib(t:\tau)$ ($Lib(M:\tau_1 \to \tau_2)$, $Lib(\varphi)$), y $\xi^2_1(X:\tau_1' \to \tau_2') = \xi^2_2(X:\tau_1' \to \tau_2')$ para cualquier $X:\tau_1' \to \tau_2' \in Lib(t:\tau)$ ($Lib(M:\tau_1 \to \tau_2)$, $Lib(\varphi)$).

(1) Si $\mathcal{I}_1$ y $\mathcal{I}_2$ coinciden en los Σ-símbolos de un tipo τ, entonces $[\tau]^{\Sigma_1} = [\tau]^{\Sigma_1}$.

(2) Sea $t:\tau \in T_m(\Sigma)$, si $\mathcal{I}_1$ y $\mathcal{I}_2$ coinciden en los Σ-símbolos de tipo y de dato y en las variables libres de $t:\tau$ entonces $\mathcal{I}_1[t:\tau] = \mathcal{I}_2[t:\tau]$.

(3) Sea $M:\tau_1 \to \tau_2 \in Ef(\Sigma)$, si $\mathcal{I}_1$ y $\mathcal{I}_2$ coinciden en los Σ-símbolos de tipo y de dato y en las variables libres de $M:\tau_1 \to \tau_2$ entonces $\mathcal{I}_1[M:\tau_1 \to \tau_2] = \mathcal{I}_2[M:\tau_1 \to \tau_2]$.

(4) Sea $\varphi \in L(\Sigma)$, si $\mathcal{I}_1$ y $\mathcal{I}_2$ coinciden en los Σ-símbolos de tipo y de dato y en las variables libres de φ entonces $\mathcal{I}_1 \models \varphi \iff \mathcal{I}_2 \models \varphi$.

Demostración:

Para probar (1) hay que probar que $[\tau]^{\Sigma_1}\eta = [\tau]^{\Sigma_2}\eta$ para cualquier asignación $\eta: TVar \to I^{\Sigma_1}$ ($I^{\Sigma_1} = I^{\Sigma_2}$). La demostración se hace recorriendo los casos posibles.

- Si τ es una variable de tipo, la demostración es trivial.
- $[ct]^{\Sigma_1}\eta = ct^{\Sigma_1} = ct^{\Sigma_2} = [ct]^{\Sigma_2}\eta$.
- $[bool]^{\Sigma_1}\eta = B = [bool]^{\Sigma_2}\eta$.

El resto de los casos se obtienen por inducción en la construcción de los tipos.

Probamos (2) y (3) por inducción mutua en la construcción de los términos y de las expresiones funcionales. Para cualquier asignación $\eta: TVar \to I^{\Sigma_1}$, se verifica:

- $\mathcal{I}_1[\perp:\tau]\eta = \perp^{\Sigma_1}_{\eta(\tau)} = \perp^{\Sigma_2}_{\eta(\tau)}$ (por (1)) $= \mathcal{I}_2[\perp:\tau]\eta$
- $\mathcal{I}_1[x:\tau]\eta = \xi^1_1(x:\tau)\eta = \xi^1_2(x:\tau)\eta = \mathcal{I}_2[x:\tau]\eta$
- $\mathcal{I}_1[c:\tau']\eta = c:\tau^{D_1}\eta(\sigma) = c:\tau^{D_2}\eta(\sigma)$ si $c:\to\tau \in \Sigma_d$ y $\tau' = \tau\sigma$
  $= \mathcal{I}_2[c:\tau']\eta$

- $\mathcal{I}_1[\![\texttt{true}]\!]\eta = \texttt{tt} = \mathcal{I}_2[\![\texttt{true}]\!]\eta$

- $\mathcal{I}_1[\![\texttt{false}]\!]\eta = \texttt{ff} = \mathcal{I}_2[\![\texttt{false}]\!]\eta$

- $\mathcal{I}_1[\![(t_1:\tau_1,\ldots,t_n:\tau_n)]\!]\eta = \begin{cases} \langle \mathcal{I}_1[\![t_1:\tau_1]\!]\eta,\ldots,\mathcal{I}_1[\![t_n:\tau_n]\!]\eta\rangle & \text{si } \mathcal{I}_1[\![t_1:\tau_1]\!]\eta \neq \bot^{\Sigma_1}_{\eta(\tau_1)} \quad (1\le i\le n) \\ \bot^{\Sigma_1}_{\eta(\tau_1 x\ldots x\tau_n)} & \text{en otro caso} \end{cases}$

  $= \begin{cases} \langle \mathcal{I}_2[\![t_1:\tau_1]\!]\eta,\ldots,\mathcal{I}_2[\![t_n:\tau_n]\!]\eta\rangle = \mathcal{I}_2[\![(t_1:\tau_1,\ldots t_n:\tau_n)]\!]\eta & \text{si } \mathcal{I}_2[\![t_1:\tau_1]\!]\eta \neq \bot^{\Sigma_2}_{\eta(\tau_1)} \quad (1\le i\le n) \\ \bot^{\Sigma_2}_{\eta(\tau_1 x\ldots x\tau_n)} & \text{en otro caso} \end{cases}$

  por (1) e hipótesis de inducción

  $= \mathcal{I}_2[\![(t_1:\tau_1,\ldots,t_n:\tau_n)]\!]\eta$

- $\mathcal{I}_1[\![(\texttt{if } t:\texttt{boot then } t_1:\tau \texttt{ else } t_2:\tau)]\!]\eta$

  $= \begin{cases} \bot^{\Sigma_1}_{\eta(\tau)} & \text{si } \mathcal{I}_1[\![t:\texttt{bool}]\!]\eta = \bot_{\texttt{bool}} \\ \mathcal{I}_1[\![t_1:\tau]\!]\eta & \text{si } \mathcal{I}_1[\![t:\texttt{bool}]\!]\eta = \texttt{tt} \\ \mathcal{I}_1[\![t_2:\tau]\!]\eta & \text{si } \mathcal{I}_1[\![t:\texttt{bool}]\!]\eta = \texttt{ff} \end{cases}$

  $= \begin{cases} \bot^{\Sigma_2}_{\eta(\tau)} & \text{si } \mathcal{I}_2[\![t:\texttt{bool}]\!]\eta = \bot_{\texttt{bool}} \\ \mathcal{I}_2[\![t_1:\tau]\!]\eta & \text{si } \mathcal{I}_2[\![t:\texttt{bool}]\!]\eta = \texttt{tt} \\ \mathcal{I}_2[\![t_2:\tau]\!]\eta & \text{si } \mathcal{I}_2[\![t:\texttt{bool}]\!]\eta = \texttt{ff} \end{cases}$ por hipót. de induc. y (1)

  $= \mathcal{I}_2[\![(\texttt{if } t:\texttt{bool then } t_1:\tau \texttt{ else } t_2:\tau)]\!]\eta$

- $\mathcal{I}_1[\![(M\ t:\tau_1):\tau_2]\!]\eta = \mathcal{I}_1[\![M:\tau_1\to\tau_2]\!]\eta\ (\mathcal{I}_1[\![t:\tau_1]\!]\eta)$

  $= \mathcal{I}_1[\![M:\tau_1\to\tau_2]\!]\eta\ (\mathcal{I}_2[\![t:\tau_1]\!]\eta)$ por inducción sobre $t:\tau_1$

  $= \mathcal{I}_2[\![M\ \tau_1\to\tau_2]\!]\eta\ (\mathcal{I}_2[\![t:\tau_1]\!]\eta)$ por inducción sobre $M:\tau_1\to\tau_2$

  $= \mathcal{I}_2[\![(M\ t:\tau_1):\tau_2]\!]\eta.$

- $\mathcal{I}_1[\![\bot:\tau_1\to\tau_2]\!]\eta = \bot^{\Sigma_1}_{\eta(\tau_1\to\tau_2)} = \bot^{\Sigma_2}_{\eta(\tau_1\to\tau_2)}$ por (1)

  $= \mathcal{I}_2[\![\bot:\tau_1\to\tau_2]\!]\eta$

- $\mathcal{I}_1[\![f:\tau_1'\to\tau_2']\!]\eta = f:\tau_1\to\tau_2^{\mathcal{D}_1}\eta\{\sigma\}$

  $= f:\tau_1\to\tau_2^{\mathcal{D}_2}\eta\{\sigma\}$ siendo $f:\tau_1\to\tau_2 \in \Sigma_d$ y $\tau_1'\to\tau_2' = \tau_1\sigma\to\tau_2\sigma$

  $= \mathcal{I}_2[\![f:\tau_1'\to\tau_2']\!]\eta$

- $\mathcal{I}_1[\![X:\tau_1\to\tau_2]\!]\eta = \xi^2_1(X:\tau_1\to\tau_2)\eta = \xi^2_2(X:\tau_1\to\tau_2)\eta = \mathcal{I}_2[\![X:\tau_1\to\tau_2]\!]\eta$

- Sea $\langle d_1\ldots d_n\rangle \in [\![\tau_1]\!]^{\mathcal{I}_1}\eta\bullet\ldots\bullet[\![\tau_n]\!]^{\mathcal{I}_1}\eta = [\![\tau_1]\!]^{\mathcal{I}_1}\eta\bullet\ldots\bullet[\![\tau_n]\!]^{\mathcal{I}_1}\eta$ por (1), supongamos $\langle d_1\ldots d_n\rangle \neq \bot_\bullet$, entonces

  $\mathcal{I}_1[\![(\lambda x_1:\tau_1..x_n:\tau_n.\ t:\tau)]\!]\eta(d_1,\ldots,d_n) = \mathcal{I}_1\left\{\frac{d_1\ldots d_n}{x_1:\tau_1\ldots x_n:\tau_n}\right\}[\![t:\tau]\!]\eta$

$= \mathcal{J}_2\left\{\frac{d_1 \ldots d_n}{x_1:\tau_1 \ldots x_n:\tau_n}\right\}[\![t:\tau]\!]\eta$

por inducción sobre $\mathcal{J}_1\left\{\frac{d_1 \ldots d_n}{x_1:\tau_1 \ldots x_n:\tau_n}\right\}$ y $\mathcal{J}_2\left\{\frac{d_1 \ldots d_n}{x_1:\tau_1 \ldots x_n:\tau_n}\right\}$

$= \mathcal{J}_2[\![(\lambda x_1:\tau_1 \ldots x_n:\tau_n.\, t:\tau)]\!]\eta(d_1 \ldots d_n)$. El caso $\perp_\bullet$ es obvio por tratarse de funciones estrictas.

- Para probar $\mathcal{J}_1[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta = \mathcal{J}_2[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta$ basta con probar $T(\mathcal{J}_1, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta) = T(\mathcal{J}_2, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)$ y esto es cierto puesto que:

$T(\mathcal{J}_1, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)(h) = \mathcal{J}_1\{h/X:\tau_1\to\tau_2\}[\![M:\tau_1\to\tau_2]\!]\eta$

$= \mathcal{J}_2\{h/X:\tau_1\to\tau_2\}[\![M:\tau_1\to\tau_2]\!]\eta$

aplicando inducción a $\mathcal{J}_1\{h/X:\tau_1\to\tau_2\}$ y a $\mathcal{J}_2\{h/X:\tau_1\to\tau_2\}$

$= T(\mathcal{J}_2, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)(h)$.

Para probar (4), probamos $\mathcal{J}_1, \eta \models \varphi \Leftrightarrow \mathcal{J}_2, \eta \models \varphi$ para cualquier asignación η, y lo hacemos por inducción en la construcción de las fórmulas.

$\mathcal{J}_1, \eta \models t_1:\tau \subseteq t_2:\tau \quad \Leftrightarrow \mathcal{J}_1[\![t_1:\tau]\!]\eta \leq^{\mathcal{I}_1}_{\eta(\tau)} \mathcal{J}_1[\![t_2:\tau]\!]\eta$

$\Leftrightarrow \mathcal{J}_2[\![t_1:\tau]\!]\eta \leq^{\mathcal{I}_2}_{\eta(\tau)} \mathcal{J}_2[\![t_2:\tau]\!]\eta$ por (1) y (2)

$\Leftrightarrow \mathcal{J}_2, \eta \models t_1:\tau \subseteq t_2:\tau$.

$\mathcal{J}_1, \eta \models (p\ t:\tau') \quad \Leftrightarrow \mathcal{J}_1[\![t:\tau']\!]\eta \in p:\tau^{\mathcal{D}_1}\eta\{\sigma\}$ con $p:\tau \in \Sigma_d$ y $\tau' = \tau\sigma$

$\Leftrightarrow \mathcal{J}_1[\![t:\tau']\!]\eta \in p:\tau^{\mathcal{D}_2}\eta\{\sigma\}$

$\Leftrightarrow \mathcal{J}_2[\![t:\tau']\!]\eta \in p:\tau^{\mathcal{D}_2}\eta\{\sigma\}$ por (2)

$\Leftrightarrow \mathcal{J}_2, \eta \models (p\ t:\tau')$.

$\mathcal{J}_1, \eta \models \neg\varphi \quad \Leftrightarrow$ no $\mathcal{J}_1, \eta \models \varphi$

$\Leftrightarrow$ no $\mathcal{J}_2, \eta \models \varphi$ por inducción sobre φ

$\Leftrightarrow \mathcal{J}_2, \eta \models \neg\varphi$.

$\mathcal{J}_1, \eta \models \varphi \vee \psi \quad \Leftrightarrow \mathcal{J}_1, \eta \models \varphi$ ó $\mathcal{J}_1, \eta \models \psi$

$\Leftrightarrow \mathcal{J}_2, \eta \models \varphi$ ó $\mathcal{J}_2, \eta \models \psi$ inducción sobre φ y ψ

$\Leftrightarrow \mathcal{J}_2, \eta \models \varphi \vee \psi$.

$\mathcal{J}_1, \eta \models \varphi \wedge \psi \quad \Leftrightarrow \mathcal{J}_1, \eta \models \varphi$ y $\mathcal{J}_1, \eta \models \psi$

$\Leftrightarrow \mathcal{J}_2, \eta \models \varphi$ y $\mathcal{J}_2, \eta \models \psi$ inducción sobre φ y ψ

$\Leftrightarrow \mathcal{J}_2, \eta \models \varphi \wedge \psi$.

$\mathcal{J}_1, \eta \models \forall x:\tau\ \varphi \quad \Leftrightarrow \mathcal{J}_1\{a/x:\tau\}, \eta \models \varphi$ para todo $a \in [\![\tau]\!]^{\mathcal{I}_1}\eta$

$\Leftrightarrow \mathcal{J}_1\{a/x:\tau\}, \eta \models \varphi$ para todo $a \in [\![\tau]\!]^{\mathcal{I}_2}\eta$ por (1)

$\Leftrightarrow \mathcal{J}_2\{a/x:\tau\}, \eta \models \varphi$ p.t. $a \in [\![\tau]\!]^{\mathcal{I}_2}\eta$ por inducción aplicada a $\mathcal{J}_1\{a/x:\tau\}$ y $\mathcal{J}_2\{a/x:\tau\}$

$\Leftrightarrow \mathcal{J}_2, \eta \models \forall x:\tau\ \varphi$.

$\mathcal{I}_1, \eta \vdash \exists x{:}\tau\ \varphi$ $\quad\Leftrightarrow$ ex. un $a \in [\tau]^{\mathcal{I}_1}\eta$ tal que $\mathcal{I}_1\{a/x{:}\tau\}, \eta \vdash \varphi$

$\Leftrightarrow$ ex. $a \in [\tau]^{\mathcal{I}_2}\eta$ t.q. $\mathcal{I}_1\{a/x{:}\tau\}, \eta \vdash \varphi$ por (1)

$\Leftrightarrow$ existe un $a \in [\tau]^{\mathcal{I}_2}\eta$ tal que $\mathcal{I}_2\{a/x{:}\tau\}, \eta \vdash \varphi$
por inducción sobre $\mathcal{I}_1\{a/x{:}\tau\}$ y $\mathcal{I}_2\{a/x{:}\tau\}$

$\Leftrightarrow \mathcal{I}_2, \eta \vdash \exists x{:}\tau\ \varphi$. ■

Teorema 2.3.2

a) El funcional $T(\mathcal{I}, X{:}\tau_1\to\tau_2, t{:}\tau, \eta){:}\ [[\tau_1]^{\mathcal{I}}\eta \xrightarrow[\bullet]{} [\tau_2]^{\mathcal{I}}\eta] \to [\tau]^{\mathcal{I}}\eta$ definido como:

$$T(\mathcal{I}, X{:}\tau_1\to\tau_2, t{:}\tau, \eta)(h) = \mathcal{I}\left\{\frac{h}{X{:}\tau_1\to\tau_2}\right\}[t{:}\tau]\eta$$

está bien definido y es continuo.

b) El funcional

$T(\mathcal{I}, X{:}\tau_1\to\tau_2, M{:}\tau_1'\to\tau_2', \eta){:}\ [[\tau_1]^{\mathcal{I}}\eta \xrightarrow[\bullet]{} [\tau_2]^{\mathcal{I}}\eta] \to [[\tau_1']^{\mathcal{I}}\eta \xrightarrow[\bullet]{} [\tau_2']^{\mathcal{I}}\eta]$

donde $T(\mathcal{I}, X{:}\tau_1\to\tau_2, M{:}\tau_1'\to\tau_2', \eta)(h) = \mathcal{I}\left\{\frac{h}{X{:}\tau_1\to\tau_2}\right\}[M{:}\tau_1'\to\tau_2']\eta$

está bien definido y es continuo.

Demostración:

Demostramos a) y b) por inducción mutua sobre la construcción de los términos y las expresiones funcionales.

a) Si $t{:}\tau$ es $\bot{:}\tau$, $c{:}\tau$, $x{:}\tau$, true o false, entonces $\mathcal{I}\left\{\frac{h}{X{:}\tau_1\to\tau_2}\right\}[t{:}\tau]\eta = \mathcal{I}[t{:}\tau]\eta$. De la definición de interpretación de estos términos deducimos que $T(\mathcal{I}, X{:}\tau_1\to\tau_2, t{:}\tau, \eta)$ está bien definido en estos casos y es constante por lo cual es continuo.

- $T(\mathcal{I}, X{:}\tau_1\to\tau_2, (t_1{:}\tau_1', \ldots, t_n{:}\tau_n'), \eta)(h)$

$$= \begin{cases} \bot_{\eta(\tau_1' \times \ldots \times \tau_n')} & \text{si } \mathcal{I}(h/X{:}\tau_1\to\tau_2)[t_i{:}\tau_i']\eta = \bot_{\eta(\tau_i')} \text{ para algún } i \\ \langle \mathcal{I}\left\{\frac{h}{X{:}\tau_1\to\tau_2}\right\}[t_1{:}\tau_1']\eta, \ldots, \mathcal{I}\left\{\frac{h}{X{:}\tau_1\to\tau_2}\right\}[t_n{:}\tau_n']\eta\rangle & \text{en otro caso} \end{cases}$$

En ámbos casos (aplicando inducción en el segundo) se tiene que $T(\mathcal{I}, X{:}\tau_1\to\tau_2, (t_1{:}\tau_1', \ldots, t_n{:}\tau_n'), \eta) \in [\tau_1']^{\mathcal{I}}\eta \otimes \ldots \otimes [\tau_n']^{\mathcal{I}}\eta$ y está bien definido.

Para probar la continuidad de T en este caso, se aplican las propiedades del producto estricto con respecto al supremo:

$T(\mathcal{I}, X{:}\tau_1\to\tau_2, (t_1{:}\tau_1', \ldots, t_n{:}\tau_n'), \eta)(\bigsqcup_{j=0}^{\infty} h_j)$

$$= \mathcal{I}\left\{\frac{\bigsqcup_{j=0}^{\infty} h_j}{X{:}\tau_1\to\tau_2}\right\}[(t_1{:}\tau_1', \ldots, t_n{:}\tau_n')]\eta$$

$$= \begin{cases} \perp_{\eta(\tau_1' \times \ldots \times \tau_n')} & \text{si } \Im\{\sqcup h_j / X:\tau_1 \to \tau_2\}[\![t_1:\tau_1']\!]\eta = \perp_{\eta(\tau_1')} \text{ para algún } i \\ \langle \Im\left\{\frac{\sqcup_{j=0}^{\infty} h_j}{X:\tau_1 \to \tau_2}\right\}[\![t_1:\tau_1']\!]\eta, \ldots, \Im\left\{\frac{\sqcup_{j=0}^{\infty} h_j}{X:\tau_1 \to \tau_2}\right\}[\![t_n:\tau_n']\!]\eta \rangle & \text{en otro caso} \end{cases}$$

Estudiamos sólo el caso distinto de *bottom* que presenta más interés

$= \langle T(\Im, X:\tau_1 \to \tau_2, t_1:\tau_1', \eta)(\sqcup_{j=0}^{\infty} h_j), \ldots, T(\Im, X:\tau_1 \to \tau_2, t_n:\tau_n', \eta)(\sqcup_{j=0}^{\infty} h_j) \rangle$

$= \langle \sqcup_{j=0}^{\infty} T(\Im, X:\tau_1 \to \tau_2, t_1:\tau_1', \eta)(h_j), \ldots, \sqcup_{j=0}^{\infty} T(\Im, X:\tau_1 \to \tau_2, t_n:\tau_n', \eta)(h_j) \rangle$

por hipótesis de inducción

$= \sqcup_{j=0}^{\infty} \langle T(\Im, X:\tau_1 \to \tau_2, t_1:\tau_1', \eta)(h_j), \ldots, T(\Im, X:\tau_1 \to \tau_2, t_n:\tau_n', \eta)(h_j) \rangle$

por las propiedades de $\bullet$

$= \sqcup_{j=0}^{\infty} \langle \Im\left\{\frac{h_j}{X:\tau_1 \to \tau_2}\right\}[\![t_1:\tau_1']\!]\eta, \ldots, \Im\left\{\frac{h_j}{X:\tau_1 \to \tau_2}\right\}[\![t_n:\tau_n']\!]\eta \rangle$

$= \sqcup_{j=0}^{\infty} \Im\left\{\frac{h_j}{X:\tau_1 \to \tau_2}\right\}[\![(t_1:\tau_1', \ldots, t_n:\tau_n')]\!]\eta$

$= \sqcup_{j=0}^{\infty} T(\Im, X:\tau_1 \to \tau_2, (t_1:\tau_1', \ldots, t_n:\tau_n'), \eta)(h_j).$

- $T(\Im, X:\tau_1 \to \tau_2, (\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau), \eta)(h)$

$$= \begin{cases} \Im\{h/X:\tau_1 \to \tau_2\}[\![t_1:\tau]\!]\eta & \text{si } \Im\{h/X:\tau_1 \to \tau_2\}[\![t:\text{bool}]\!]\eta = tt \\ \Im\{h/X:\tau_1 \to \tau_2\}[\![t_2:\tau]\!]\eta & \text{si } \Im\{h/X:\tau_1 \to \tau_2\}[\![t:\text{bool}]\!]\eta = ff \\ \perp_{\eta(\tau)} & \text{si } \Im\{h/X:\tau_1 \to \tau_2\}[\![t:\text{bool}]\!]\eta = \perp_{\text{bool}} \end{cases}$$

$$= \begin{cases} T(\Im, X:\tau_1 \to \tau_2, t_1:\tau, \eta)(h) & \text{si } T(\Im, X:\tau_1 \to \tau_2, t:\text{bool}, \eta)(h) = tt \\ T(\Im, X:\tau_1 \to \tau_2, t_2:\tau, \eta)(h) & \text{si } T(\Im, X:\tau_1 \to \tau_2, t:\text{bool}, \eta)(h) = ff \\ \perp_{\eta(\tau)} & \text{si } T(\Im, X:\tau_1 \to \tau_2, t:\text{bool}, \eta)(h) = \perp_{\text{bool}} \end{cases}$$

Con lo que por la hipótesisde inducción está bien definido.

En este caso, la continuidad de T se sigue de la igualdad

$T(\Im, X:\tau_1 \to \tau_2, (\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau), \eta)(h) =$

$\text{COND}(T(\Im, X:\tau_1 \to \tau_2, t:\text{bool}, \eta)(h), T(\Im, X:\tau_1 \to \tau_2, t_1:\tau, \eta)(h),$

$T(\Im, X:\tau_1 \to \tau_2, t_2:\tau, \eta)(h))$, por hipótesis de inducción, ya que, los resultados de 2.1 aseguran que COND es continua y la composición de funciones continuas es continua.

- $T(\Im, X:\tau_1 \to \tau_2, (M\ t:\tau_1'):\tau_2', \eta)(h)$

$= \Im\left\{\frac{h}{X:\tau_1 \to \tau_2}\right\}[\![M:\tau_1' \to \tau_2']\!]\eta\ (\Im\left\{\frac{h}{X:\tau_1 \to \tau_2}\right\}[\![t:\tau_1']\!]\eta)$

$= T(\Im, X:\tau_1 \to \tau_2, M:\tau_1' \to \tau_2', \eta)(h)(T(\Im, X:\tau_1 \to \tau_2, t:\tau_1', \eta)(h))$ que deberá estar bien definido por hipótesis de inducción sobre $M:\tau_1' \to \tau_2'$ y sobre $t:\tau_1'$.

El funcional $T(\Im, X:\tau_1 \to \tau_2, (M\ t:\tau_1'):\tau_2', \eta)$ es continuo por ser

composición de las funciones continuas $T(\mathfrak{I}, X:\tau_1\to\tau_2, t:\tau_1', \eta)$ y $T(\mathfrak{I}, X:\tau_1\to\tau_2, M:\tau_1'\to\tau_2', \eta)$. A su vez, la continuidad de estos dos operadores se tiene por la hipótesis de inducción.

b) Si $M:\tau_1'\to\tau_2'$ es $\perp:\tau_1'\to\tau_2'$ o $f:\tau_1'\to\tau_2'$, entonces $\mathfrak{I}\left(\frac{h}{X:\tau_1\to\tau_2}\right)[M:\tau_1'\to\tau_2']\eta = \mathfrak{I}[M:\tau_1'\to\tau_2']\eta$. Basándonos en la definición de interpretación para estos casos podemos asegurar que $T(\mathfrak{I}, X:\tau_1\to\tau_2, M:\tau_1'\to\tau_2', \eta)$ está bien definido para $\perp:\tau_1'\to\tau_2'$ y $f:\tau_1'\to\tau_2'$ siendo además constante y por tanto continuo.

- $T(\mathfrak{I}, X:\tau_1\to\tau_2, X':\tau_1'\to\tau_2', \eta)(h)$

$$= \begin{cases} h & \text{si } X:\tau_1\to\tau_2 = X':\tau_1'\to\tau_2' \\ \xi(X':\tau_1'\to\tau_2')\eta & \text{en otro caso} \end{cases}$$

La definición de ξ^2 y la naturaleza de h hacen que esta función quede bien definida. Además, si $X:\tau_1\to\tau_2 = X':\tau_1'\to\tau_2'$, entonces $T(\mathfrak{I}, X:\tau_1\to\tau_2, X':\tau_1'\to\tau_2', \eta)$ es la identidad y por tanto es continua. Si por el contrario, $X:\tau_1\to\tau_2 \neq X':\tau_1'\to\tau_2'$, $T(\mathfrak{I}, X:\tau_1\to\tau_2, X':\tau_1'\to\tau_2', \eta)$ es una función constante y por tanto continua.

- $T(\mathfrak{I}, X:\tau_1\to\tau_2, (\lambda x_1:\tau_1'\ldots x_n:\tau_n'.\, t:\tau), \eta)(h)(d_1,\ldots,d_n)$ con $\langle d_1,\ldots,d_n\rangle$ una n-tupla de $[\tau_1'\times\ldots\times\tau_n']^{\mathrm{I}}\eta$ distinta de $\perp$.

$$= \mathfrak{I}\left(\frac{h}{X:\tau_1\to\tau_2}\right)[\lambda x_1:\tau_1'\ldots x_n:\tau_n'.\, t:\tau]\eta\ (d_1,\ldots,d_n)$$

$$= \mathfrak{I}\left(\frac{h}{X:\tau_1\to\tau_2}\right)\left(\frac{d_1\ldots d_n}{x:\tau_1'\ldots x:\tau_n'}\right)[t:\tau]\eta$$

$$= \mathfrak{I}\left(\frac{d_1\ldots d_n}{x:\tau_1'\ldots x:\tau_n'}\right)\left(\frac{h}{X:\tau_1\to\tau_2}\right)[t:\tau]\eta \qquad \text{por el lema de coincidencia}$$

$$= T\left(\mathfrak{I}\left(\frac{d_1\ldots d_n}{x:\tau_1'\ldots x:\tau_n'}\right), X:\tau_1\to\tau_2, t:\tau, \eta\right)(h) \qquad \text{que por hipótesis de inducción está bien definida.}$$

Para probar la continuidad es necesario demostrar que para cualquier $\langle d_1,\ldots,d_n\rangle \in [\tau_1'\times\ldots\times\tau_n']^{\mathrm{I}}\eta$ se tiene la igualdad

$T(\mathfrak{I}, X:\tau_1\to\tau_2, (\lambda x_1:\tau_1'\ldots x_n:\tau_n'.\, t:\tau), \eta)(\sqcup_{j=0}^{\omega} h_j)\ (d_1,\ldots,d_n)$

$= \sqcup_{j=0}^{\omega} T(\mathfrak{I}, X:\tau_1\to\tau_2, (\lambda x_1:\tau_1'\ldots x_n:\tau_n'.\, t:\tau), \eta)(h_j)\ (d_1,\ldots,d_n)$.

En efecto, $T(\mathfrak{I}, X:\tau_1\to\tau_2, (\lambda x_1:\tau_1'\ldots x_n:\tau_n'.\, t:\tau), \eta)(\sqcup_{j=0}^{\omega} h_j)\ (d_1,\ldots,d_n)$

$$= \mathfrak{I}\left(\frac{d_1\ldots d_n}{x:\tau_1'\ldots x:\tau_n'}\right)\left(\frac{\sqcup_{j=0}^{\omega} h_j}{X:\tau_1\to\tau_2}\right)[t:\tau]\eta \text{ como anteriormente}$$

$= T(\Im\left\{\frac{d_1 \dots d_n}{X:\tau_1' \dots X:\tau_n'}\right\}, X:\tau_1 \to \tau_2, t:\tau, \eta)(\sqcup_{j=0}^{\omega} h_j)$

$= \sqcup_{j=0}^{\omega} T(\Im\left\{\frac{d_1 \dots d_n}{X:\tau_1' \dots X:\tau_n'}\right\}, X:\tau_1 \to \tau_2, t:\tau, \eta)(h_j)$ por hip. de inducción

$= \sqcup_{j=0}^{\omega} \Im\left\{\frac{d_1 \dots d_n}{X:\tau_1' \dots X:\tau_n'}\right\}\left\{\frac{h_j}{X:\tau_1 \to \tau_2}\right\}[t:\tau]\eta$

$= \sqcup_{j=0}^{\omega} \Im\left\{\frac{h_j}{X:\tau_1 \to \tau_2}\right\}\left\{\frac{d_1 \dots d_n}{X:\tau_1' \dots X:\tau_n'}\right\}[t:\tau]\eta$ por el lema de coincidencia

$= \sqcup_{j=0}^{\omega} \Im\left\{\frac{h_j}{X:\tau_1 \to \tau_2}\right\}[(\lambda x_1:\tau_1' \dots x_n:\tau_n'.\, t:\tau)]\eta \ (d_1, \dots, d_n)$

$= \sqcup_{j=0}^{\omega} T(\Im, X:\tau_1 \to \tau_2, (\lambda x_1:\tau_1' \dots x_n:\tau_n'.\, t:\tau), \eta)(h_j)\ (d_1, \dots, d_n).$

- $T(\Im, X:\tau_1 \to \tau_2, (\mu X':\tau_1' \to \tau_2'.\, M), \eta)(h)$

$= \Im\left\{\frac{h}{X:\tau_1 \to \tau_2}\right\}[(\mu X':\tau_1' \to \tau_2'.\, M)]\eta$

$= fix(T(\Im\left\{\frac{h}{X:\tau_1 \to \tau_2}\right\}, X':\tau_1' \to \tau_2', M:\tau_1' \to \tau_2', \eta))$ que está bien definido por hipótesis de inducción sobre $M:\tau_1' \to \tau_2'$.

Para probar la continuidad de $T(\Im, X:\tau_1 \to \tau_2, (\mu X':\tau_1' \to \tau_2'.\, M), \eta)$, vemos que si $X:\tau_1 \to \tau_2 = X':\tau_1' \to \tau_2'$ entonces T es una función constante y por tanto continua. Si $X:\tau_1 \to \tau_2$ es distinto de $X':\tau_1' \to \tau_2'$, entonces definimos el operador $F:[[\tau_1]^{I}_{\eta} \to [\tau_2]^{I}_{\eta}] \to [[[\tau_1']^{I}_{\eta} \to [\tau_2']^{I}_{\eta}] \to [[\tau_1']^{I}_{\eta} \to [\tau_2']^{I}_{\eta}]]$ de manera que a cada h de $[[\tau_1]^{I}_{\eta} \to [\tau_2]^{I}_{\eta}]$ le hace corresponder $T(\Im\left\{\frac{h}{X:\tau_1 \to \tau_2}\right\}, X':\tau_1' \to \tau_2', M:\tau_1' \to \tau_2', \eta)$ y probamos que es continuo. Esta afirmación nos asegura la continuidad de T debido a la igualdad $T(\Im, X:\tau_1 \to \tau_2, (\mu X':\tau_1' \to \tau_2'.\, M), \eta) = fix(F(h))$ y a los resultados de la sección 2.1.

Vamos a probar entonces que $F(\sqcup_{j=0}^{\omega} h_j)(f) = \sqcup_{j=0}^{\omega} F(h_j)(f)$ para cualquier $f \in [[\tau_1']^{I}_{\eta} \to [\tau_2']^{I}_{\eta}]$

$F(\sqcup_{j=0}^{\omega} h_j)(f) = \Im\left\{\frac{\sqcup_{j=0}^{\omega} h_j}{X:\tau_1 \to \tau_2}\right\}\left\{\frac{f}{X':\tau_1' \to \tau_2'}\right\}[M:\tau_1' \to \tau_2']\eta$

$= \Im\left\{\frac{f}{X':\tau_1' \to \tau_2'}\right\}\left\{\frac{\sqcup_{j=0}^{\omega} h_j}{X:\tau_1 \to \tau_2}\right\}[M:\tau_1' \to \tau_2']\eta$ aplicando el lema de coincidencia ya que suponemos $X:\tau_1 \to \tau_2 \neq X':\tau_1' \to \tau_2'$

$= T(\Im\left\{\frac{f}{X':\tau_1' \to \tau_2'}\right\}, X:\tau_1 \to \tau_2, M:\tau_1' \to \tau_2', \eta)(\sqcup_{j=0}^{\omega} h_j)$

$= \sqcup_{j=0}^{\omega} T(\mathcal{I}\left\{\frac{f}{X':\tau_1'\to\tau_2'}\right\}, X:\tau_1\to\tau_2, M:\tau_1'\to\tau_2', \eta)(h_j)$ por inducción sobre $M:\tau_1'\to\tau_2'$

$= \sqcup_{j=0}^{\omega} \mathcal{I}\left\{\frac{f}{X':\tau_1'\to\tau_2'}\right\}\left\{\frac{h_j}{X:\tau_1\to\tau_2}\right\}[\![M:\tau_1'\to\tau_2']\!]\eta$

$= \sqcup_{j=0}^{\omega} \mathcal{I}\left\{\frac{h_j}{X:\tau_1\to\tau_2}\right\}\left\{\frac{f}{X':\tau_1'\to\tau_2'}\right\}[\![M:\tau_1'\to\tau_2']\!]\eta$ por el lema de cincid. ya que $X:\tau_1\to\tau_2 \neq X':\tau_1'\to\tau_2'$

$= \sqcup_{j=0}^{\omega} T(\mathcal{I}\left\{\frac{h_j}{X:\tau_1\to\tau_2}\right\}, X':\tau_1'\to\tau_2', M:\tau_1'\to\tau_2', \eta)(f)$

$= \sqcup_{j=0}^{\omega} F(h_j)\ (f)$. ■

2.4 EJEMPLOS DE APLICACION DEL LENGUAJE

Al comenzar este primer capítulo señalábamos nuestro interés en obtener una lógica capaz de expresar razonamientos matemáticos y propiedades de programas funcionales. Esta capacidad se pone de manifiesto en los ejemplos que aparecen a continuación escritos en nuestro lenguaje.

Ejemplo 2.4.1

Consideremos la signatura $\Sigma_N = \langle \Sigma_t, \Sigma_d \rangle$, siendo:

$\Sigma_t = \{nat/0\}$ y

$\Sigma_d = \{0:\to nat,\ suc: nat\to nat,\ pred: nat\to nat,\ +: nat\times nat\to nat,$
$*: nat\times nat\to nat,\ es\text{-}cero: nat\to bool\}$

Esta signatura nos va a permitir escribir propiedades de los números naturales y definir operaciones entre ellos. Más adelante comprobaremos que las siguientes fórmulas constituyen una caracterización axiomática de la aritmética estándar de los números naturales.

$\forall x{:}nat\ \forall y{:}nat((suc\ x) = (suc\ y) \to x = y)$

$\forall x{:}nat\ \neg(suc\ x) = 0$

$(es\_cero\ 0) = true$

$\forall x{:}nat(\neg x \subseteq \bot{:}nat \to (es\_cero\ (suc\ x)) = false)$

$\forall x{:}nat\ (pred\ (suc\ x)) = x$

$(pred\ 0) = \bot{:}nat$

$\forall y{:}nat\ ((\mu X{:}nat\rightarrow nat.(\lambda x{:}nat.(if\ (es\_cero\ x)\ then\ 0\ else\ (suc\ (X\ (pred\ x))))))\ y) = y$

$\forall x{:}nat\ (+\ (x,0)) = x$

$\forall x{:}nat\ \forall y{:}nat\ (+\ (x,(suc\ y))) = (suc\ (+\ (x,y)))$

$\forall x{:}nat\ (\neg x \leq \bot{:}nat \rightarrow (*\ (x,0) = 0)$

$\forall x{:}nat\ \forall y{:}nat\ (*\ (x,(suc\ y))) = (+\ (x,\ (*\ (x,y))))$

Teniendo en cuenta estas fórmulas podemos definir el factorial de un número natural mediante la expresión funcional siguiente:

$(\mu X{:}nat\rightarrow nat.(\lambda x{:}nat.(if\ (es\_cero\ x)\ then\ (suc\ 0)\ else\ (*\ (x,(X\ (pred\ x)))))))$

que simplificaremos por "fact:nat→nat" y que tiene propiedades como:

$\forall x{:}nat\ (\neg x \leq \bot{:}nat \rightarrow \neg(fact\ x){:}nat \leq \bot{:}nat),$

$(fact\ (suc\ 0)) = (suc\ 0)$, etc.

Si incorporamos el símbolo de predicado par:nat a nuestra signatura podemos escribir por ejemplo la fórmula:

$(par\ x_1{:}nat) \rightarrow (par\ (*\ (x_1{:}nat, x_2{:}nat)))$

y el conjunto de números pares podrá especificarse mediante la fórmula:

$\exists x{:}nat\ x = (+\ (y{:}nat, y{:}nat))$

Ejemplo 2.4.2

Consideremos la signatura $\Sigma_L = \langle \Sigma_t, \Sigma_d \rangle$, siendo:

$\Sigma_t = \{list/1\}$ y

$\Sigma_d = \{nil:\rightarrow\rho,\ append: list(\rho)\times list(\rho)\rightarrow list(\rho),\ cons: \rho\times list(\rho)\rightarrow list(\rho)\}$

Las siguientes Σ-fórmulas representan una especificación axiomática de la concatenación de listas de cualquier tipo de elementos.

$\forall x{:}list(\rho)\ (append\ (nil,x)) = x,$

$\forall x{:}list(\rho)\ \forall y{:}list(\rho)\ \forall z{:}\rho\ (append\ ((cons\ (z,x)),y)) = (cons\ (z,(append\ (x,y))))$

Una manera alternativa de obtener la expresión funcional "append" consiste en definirla utilizando el μ-operador. Si consideramos la signatura de datos

$\Sigma_d = \{nil:\rightarrow\rho,\ car: list(\rho)\rightarrow\rho,\ tail: list(\rho)\rightarrow list(\rho),$

$null: list(\rho)\rightarrow bool,\ cons: \rho\times list(\rho)\rightarrow list(\rho)\}$

Las siguientes fórmulas sirven para especificar el tipo lista:

$(null\ nil) = true$

$\forall x{:}\rho\ \forall y{:}list(\rho)\ ((\neg y \leq \perp{:}list(\rho) \wedge \neg x \leq \perp{:}\rho) \rightarrow (null\ (cons\ (x,y))) = false)$

$(car\ nil) = \perp{:}\rho$

$\forall x{:}\rho\ \forall y{:}list(\rho)\ ((\neg y \leq \perp{:}list(\rho) \wedge \neg x \leq \perp{:}\rho) \rightarrow (car\ (cons\ (x,y))) = x)$

$\forall x{:}\rho\ \forall y{:}list(\rho)\ ((\neg y \leq \perp{:}list(\rho) \wedge \neg x \leq \perp{:}\rho) \rightarrow (tail\ (cons\ (x,y))) = y)$

Si se verifican las fórmulas anteriores, entonces la siguiente expresión funcional representa la función de concatenación de dos listas:

$(\mu X{:}list(\rho) \times list(\rho) \rightarrow list(\rho).(\lambda x{:}list(\rho)\ y{:}list(\rho).(if\ (null\ x)$
$then\ y\ else\ (cons\ ((car\ x),(X\ ((tail\ x),y))))))))$

Ejemplo 2.4.3

Consideremos ahora una signatura que sea la unión de las signaturas de los dos ejemplos anteriores y que además contenga un símbolo de función "Plng: list(nat)→nat", es decir:

Σ_t = {nat/0, list/1} y

Σ_d = {0:→nat, nil:→ρ, car: list(ρ)→ρ, tail: list(ρ)→list(ρ),
null: list(ρ)→bool, cons: ρ×list(ρ)→list(ρ), suc: nat→nat,
pred: nat→nat, +: nat×nat→nat, *: nat×nat→nat, par: nat,
es_cero: nat→bool, Plng: list(nat)→nat}.

Si se verifican las fórmulas:

$(Plng\ nil) = 0,$

$\forall x{:}nat\ ((par\ x) \rightarrow \forall y{:}list(nat)\ (Plng\ (cons\ (x,y))) = (suc\ (Plng\ y))$

$\forall x{:}nat\ (\neg(par\ x) \rightarrow \forall y{:}list(nat)\ (Plng\ (cons\ (x,y))) = (Plng\ y)$

entonces la función "Plng" representará el número de elementos de una lista de naturales que verifican la propiedad "par". Si a su vez el predicado "par" viene especificado por la fórmula:

$\forall x{:}nat\ ((par\ x) \leftrightarrow \exists z{:}nat\ (\neg z \leq \perp{:}nat \wedge x = (+\ (z,z))))$

"Plng" será el número de elementos pares de una lista de naturales.

Otra manera de especificar esta función utilizando el μ-operador es la siguiente:

$(\mu X{:}list(nat) \rightarrow nat.(\lambda x{:}list(nat).(if\ (null\ x)\ then\ 0\ else$
$(if\ (par\ (car\ x))\ then\ (suc\ (X\ (tail\ x)))\ else\ (X\ (tail\ x))))))$

Las expresiones escritas en este ejemplo están bien tipadas ya que el carácter polimórfico del símbolo "null" permite construir funciones cuyo tipo es una instancia del tipo más general; lo mismo ocurre con "car: list(ρ)$\rightarrow\rho$" y "tail: list(ρ)$\rightarrow$list(ρ)".

Un cálculo para PLPR debe ser capaz de deducir la fórmula

$\forall x$: list(nat) $\forall y$: list(nat)(Plng (append (x,y)))=(+ ((Plng x),(Plng y)))

Si en esta fórmula se sustituyen los símbolos "Plng" y "append" por sus definiciones recursivas, tendremos anidamiento de μ-operadores. Aunque a simple vista esto pueda parecer engorroso y la definición axiomática de las funciones parezca más sencilla, las buenas propiedades del μ-operador nos facilitarán su tratamiento, permitiendo, como veremos más adelante, el empleo de una regla de inducción de punto fijo generalizada que nos proporcionará una demostración elegante de la fórmula anterior sin necesidad de tener que explicitar una regla de inducción para cada tipo de datos.

3. RESULTADOS SEMANTICOS

De la sémantica definida en la sección anterior se deducen multitud de resultados que merece la pena resaltar, tanto para comprobar las buenas propiedades de nuestra lógica, como por su utilidad a la hora de constriur cálculos correctos para PLPR.

3.1 ALGUNOS ASPECTOS TECNICOS

Lema 3.1.1 (Sustitución de variables de dato y función)

Sea $\mathfrak{I} = \langle I, \mathfrak{D}, \xi \rangle$ una Σ-interpretación. Se define:

a) $\mathfrak{I}\{\mathfrak{I}[t:\tau]/x:\tau\} = \langle I, \mathfrak{D}, \xi\{\mathfrak{I}[t:\tau]/x:\tau\}\rangle$ donde:

$$\xi\{\mathfrak{I}[t:\tau]/x:\tau\}^1\ (y:\tau')\eta = \begin{cases} \mathfrak{I}[t:\tau]\eta & \text{si } x:\tau = y:\tau' \\ \xi(y:\tau')\eta & \text{en otro caso} \end{cases}$$

$$\xi\{\mathfrak{I}[t:\tau]/x:\tau\}^2 = \xi^2$$

b) $\mathfrak{I}\left\{\frac{\mathfrak{I}[M:\tau_1\to\tau_2]}{X:\tau_1\to\tau_2}\right\} = \langle I, \mathfrak{D}, \xi\left\{\frac{\mathfrak{I}[M:\tau_1\to\tau_2]}{X:\tau_1\to\tau_2}\right\}\rangle$ donde:

$$\xi\left\{\frac{\mathfrak{I}[M:\tau_1\to\tau_2]}{X:\tau_1\to\tau_2}\right\}^2(Y:\tau_1'\to\tau_2')\eta = \begin{cases} \mathfrak{I}[M:\tau_1\to\tau_2]\eta & \text{si } X:\tau_1\to\tau_2 = Y:\tau_1'\to\tau_2' \\ \xi(Y:\tau_1'\to\tau_2')\eta & \text{en otro caso} \end{cases}$$

$$\xi\left\{\frac{\mathfrak{I}[M:\tau_1\to\tau_2]}{X:\tau_1\to\tau_2}\right\}^1 = \xi^1.$$

Entonces se verifica:

(A) Para cualquier $t':\tau' \in T_\Sigma(\Sigma)$:

(i) Si $x:\tau$, $t:\tau \in T_\Sigma(\Sigma)$ entonces

$$\mathfrak{I}[t':\tau'[t:\tau/x:\tau]] = \mathfrak{I}\{\mathfrak{I}[t:\tau]/x:\tau\}[t':\tau'].$$

(ii) Si $X:\tau_1\to\tau_2$, $M:\tau_1\to\tau_2 \in E_F(\Sigma)$ entonces

$$\mathfrak{I}[t':\tau'[M:\tau_1\to\tau_2/X:\tau_1\to\tau_2]] = \mathfrak{I}\left\{\frac{\mathfrak{I}[M:\tau_1\to\tau_2]}{X:\tau_1\to\tau_2}\right\}[t':\tau']$$

(B) Para cualquier $M':\tau_1'\to\tau_2' \in E_F(\Sigma)$:

(i) Si $x:\tau$, $t:\tau \in T_\Sigma(\Sigma)$ entonces

$$\mathfrak{I}[M':\tau_1'\to\tau_2'[t:\tau/x:\tau]] = \mathfrak{I}\{\mathfrak{I}[t:\tau]/x:\tau\}[M':\tau_1'\to\tau_2']$$

(ii) Si $X:\tau_1\to\tau_2$, $M:\tau_1\to\tau_2 \in E_F(\Sigma)$ entonces

$$\mathfrak{I}\left[M':\tau_1'\to\tau_2'\left[\frac{M:\tau_1\to\tau_2}{X:\tau_1\to\tau_2}\right]\right] = \mathfrak{I}\left\{\frac{\mathfrak{I}[M:\tau_1\to\tau_2]}{X:\tau_1\to\tau_2}\right\}[M':\tau_1'\to\tau_2']$$

(C) Para cualquier $\varphi \in L(\Sigma)$ se verifica:

(i) Si $x:\tau$, $t:\tau \in T_\Sigma(\Sigma)$ entonces

$\mathcal{I} \models \varphi[t:\tau/x:\tau] \iff \mathcal{I}\{\mathcal{I}[\![t:\tau]\!]/x:\tau\} \models \varphi$

(ii) Si $X:\tau_1 \to \tau_2$, $M:\tau_1 \to \tau_2 \in Ef(\Sigma)$ entonces

$$\mathcal{I} \models \varphi[M:\tau_1\to\tau_2/X:\tau_1\to\tau_2] \iff \mathcal{I}\left\{\frac{\mathcal{I}[\![M:\tau_1\to\tau_2]\!]}{X:\tau_1\to\tau_2}\right\} \models \varphi$$

Demostración:

Escribimos $\mathcal{I}'$ por $\mathcal{I}\{\mathcal{I}[\![t:\tau]\!]/x:\tau\}$ y ξ' por $\xi\{\mathcal{I}[\![t:\tau]\!]/x:\tau\}$.

Demostramos (A) y (B) por inducción mutua sobre la construcción de los términos y de las expresiones funcionales.

Para (A)-(i) probamos $\mathcal{I}[\![t':\tau'[t:\tau/x:\tau]]\!]\eta = \mathcal{I}\{\mathcal{I}[\![t:\tau]\!]/x:\tau\}[\![t':\tau']\!]\eta$ para cualquier $\eta \in AT_p^{\Sigma}$. Distinguimos casos:

- $\mathcal{I}[\![x':\tau'[t:\tau/x:\tau]]\!]\eta = \begin{cases}\mathcal{I}[\![x':\tau']\!]\eta & \text{si } x:\tau \neq x':\tau' \\ \mathcal{I}[\![t:\tau]\!]\eta & \text{si } x:\tau = x':\tau'\end{cases}$

  $= \begin{cases}\xi(x':\tau')\eta = \xi'(x':\tau')\eta & \text{si } x:\tau \neq x':\tau' \\ \xi'(x':\tau')\eta & \text{si } x:\tau = x':\tau'\end{cases}$ ya que si $x:\tau = x':\tau'$,

  $\xi'(x':\tau')\eta = \mathcal{I}[\![t:\tau]\!]\eta$ para cualquier $\eta \in AT_p^{\Sigma}$

  $= \mathcal{I}'[\![x':\tau']\!]\eta$.

- $\mathcal{I}[\![\perp:\tau'[t:\tau/x:\tau]]\!]\eta = \mathcal{I}'[\![\perp:\tau']\!]\eta$ por el lema de coincidencia.

- $\mathcal{I}[\![c:\tau'[t:\tau/x:\tau]]\!]\eta = \mathcal{I}'[\![c:\tau']\!]\eta$ por el lema de coincidencia.

- $\mathcal{I}[\![\mathrm{true}[t:\tau/x:\tau]]\!]\eta = \mathcal{I}'[\![\mathrm{true}]\!]\eta$ por el lema de coincidencia.

- $\mathcal{I}[\![\mathrm{false}[t:\tau/x:\tau]]\!]\eta = \mathcal{I}'[\![\mathrm{false}]\!]\eta$ por el lema de coincidencia.

- $\mathcal{I}[\![(t_1:\tau_1,\ldots,t_n:\tau_n)[t:\tau/x:\tau]]\!]\eta$

  $= \begin{cases}\langle \mathcal{I}[\![t_1:\tau_1[t:\tau/x:\tau]]\!]\eta, \ldots, \mathcal{I}[\![t_n:\tau_n[t:\tau/x:\tau]]\!]\eta\rangle \\ \qquad \text{si } \mathcal{I}[\![t_1:\tau_1[t:\tau/x:\tau]]\!]\eta \neq \perp^{\Sigma}_{\eta(\tau_1)} \text{ p.t. } 1\le i\le n \\ \perp^{\Sigma}_{\eta(\tau_1\times\ldots\times\tau_n)} \qquad \text{en otro caso}\end{cases}$

  $= \begin{cases}\langle \mathcal{I}'[\![t_1:\tau_1]\!]\eta, \ldots, \mathcal{I}'[\![t_n:\tau_n]\!]\eta\rangle \\ \qquad \text{si } \mathcal{I}'[\![t_1:\tau_1]\!]\eta \neq \perp^{\Sigma}_{\eta(\tau_1)} \text{ p.t. } 1\le i\le n \text{ por hipótesis} \\ \perp^{\Sigma}_{\eta(\tau_1\times\ldots\times\tau_n)} \qquad \text{en otro caso} \qquad \text{de inducción}\end{cases}$

  $= \mathcal{I}'[\![(t_1:\tau_1,\ldots,t_n:\tau_n)]\!]\eta$.

- $\mathcal{I}[\![(\mathrm{if}\ t_1:\mathrm{bool}\ \mathrm{then}\ t_2:\tau'\ \mathrm{else}\ t_3:\tau')[t:\tau/x:\tau]]\!]\eta =$

  $= \begin{cases}\mathcal{I}[\![t_2:\tau'[t:\tau/x:\tau]]\!]\eta & \text{si } \mathcal{I}[\![t_1:\mathrm{bool}[t:\tau/x:\tau]]\!]\eta = tt \\ \mathcal{I}[\![t_3:\tau'[t:\tau/x:\tau]]\!]\eta & \text{si } \mathcal{I}[\![t_1:\mathrm{bool}[t:\tau/x:\tau]]\!]\eta = ff \\ \perp^{\Sigma}_{\eta(\tau')} & \text{si } \mathcal{I}[\![t_1:\mathrm{bool}[t:\tau/x:\tau]]\!]\eta = \perp_{\mathrm{bool}}\end{cases}$

$$= \begin{cases} \mathcal{I}'[\![t_2:\tau']\!]\eta & \text{si } \mathcal{I}'[\![t_1:bool]\!]\eta = tt \\ \mathcal{I}'[\![t_3:\tau']\!]\eta & \text{si } \mathcal{I}'[\![t_1:bool]\!]\eta = ff \quad \text{por hipótesis de inducción} \\ \perp^{\Sigma}_{\eta(\tau')} & \text{si } \mathcal{I}'[\![t_1:bool]\!]\eta = \perp_{bool} \end{cases}$$

$= \mathcal{I}'[\![(\text{if } t_1:bool \text{ then } t_2:\tau' \text{ else } t_3:\tau')]\!]\eta.$

$- \mathcal{I}[\![(M\ t':\tau_1):\tau_2[t:\tau/x:\tau]]\!]\eta$

$= \mathcal{I}[\![M:\tau_1\to\tau_2[t:\tau/x:\tau]]\!]\eta\ (\mathcal{I}[\![t':\tau_1[t:\tau/x:\tau]]\!]\eta)$

$= \mathcal{I}[\![M:\tau_1\to\tau_2[t:\tau/x:\tau]]\!]\eta\ (\mathcal{I}'[\![t':\tau_1]\!]\eta)$ por inducción sobre $t':\tau_1$

$= \mathcal{I}'[\![M:\tau_1\to\tau_2]\!]\eta\ (\mathcal{I}'[\![t':\tau_1]\!]\eta)$ por inducción sobre $M:\tau_1\to\tau_2$

$= \mathcal{I}'[\![(M\ t':\tau_1):\tau_2]\!]\eta.$

La demostración de (A)-(ii) es similar a la anterior distinguiendo los casos correspondientes.

De la demostración de (B)-(i) detallamos los casos de las lambda y mu expresiones:

$- \mathcal{I}[\![(\lambda x_1:\tau_1\ldots x_n:\tau_n.\ t':\tau')[t:\tau/x:\tau]]\!]\eta\ (d_1,\ldots,d_n)$

$= \mathcal{I}[\![(\lambda z_1:\tau_1\ldots z_n:\tau_n.\ t':\tau'\left[\frac{z_1:\tau_1}{x_1:\tau_1}\right]\ldots\left[\frac{z_n:\tau_n}{x_n:\tau_n}\right][t:\tau/x:\tau])]\!]\eta\ (d_1,\ldots,d_n)$

$= \mathcal{I}\left\{\frac{d_1\ldots d_n}{z_1:\tau_1\ldots z_n:\tau_n}\right\}[\![t':\tau'\left[\frac{z_1:\tau_1}{x_1:\tau_1}\right]\ldots\left[\frac{z_n:\tau_n}{x_n:\tau_n}\right][t:\tau/x:\tau])]\!]\eta$

$= \mathcal{I}\left\{\frac{d_1\ldots d_n}{z_1:\tau_1\ldots z_n:\tau_n}\right\}\left\{\frac{\mathcal{I}[\![t:\tau]\!]}{x:\tau}\right\}[\![t':\tau'\left[\frac{z_1:\tau_1}{x_1:\tau_1}\right]\ldots\left[\frac{z_n:\tau_n}{x_n:\tau_n}\right]]\!]\eta$ por hipótesis de inducción y el lema de coincidencia pues $z_1:\tau_1 \notin Lib(t:\tau)$

$= \mathcal{I}\left\{\frac{d_1\ldots d_n}{z_1:\tau_1\ldots z_n:\tau_n}\right\}\left\{\frac{\mathcal{I}[\![t:\tau]\!]}{x:\tau}\right\}\left\{\frac{d_1\ldots d_n}{z_1:\tau_1\ldots z_n:\tau_n}\right\}[\![t':\tau']\!]\eta$ por hip. de induc.

pues $\mathcal{I}\left\{\frac{d_1\ldots d_n}{z_1:\tau_1\ldots z_n:\tau_n}\right\}\left\{\frac{\mathcal{I}[\![t:\tau]\!]}{x:\tau}\right\}[\![z_1:\tau_1]\!]\eta = d_i\ (1\le i\le n)$

$= \mathcal{I}'\left\{\frac{d_1\ldots d_n}{x_1:\tau_1\ldots x_n:\tau_n}\right\}[\![t':\tau']\!]\eta$ por el lema de coincidencia ya que $z_1:\tau_1 \notin Lib(t':\tau')$

$= \mathcal{I}'[\![(\lambda x_1:\tau_1\ldots x_n:\tau_n.\ t':\tau')]\!]\eta(d_1,\ldots,d_n).$

$- \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)[t:\tau/x:\tau]]\!]\eta$

$= \mathcal{I}[\![(\mu Y:\tau_1\to\tau_2.M:\tau_1\to\tau_2[Y:\tau_1\to\tau_2/X:\tau_1\to\tau_2][t:\tau/x:\tau])]\!]\eta$

$= fix\ T(\mathcal{I}, Y:\tau_1\to\tau_2, M:\tau_1\to\tau_2[Y:\tau_1\to\tau_2/X:\tau_1\to\tau_2][t:\tau/x:\tau], \eta)$

$= fix\ T(\mathcal{I}', X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta) = \mathcal{I}'[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta$ pues para todo $h \in [[\tau_1]^{\Sigma}\eta \to_{\bullet} [\tau_2]^{\Sigma}\eta]$ se tienen las siguientes igualdades

$T(\mathcal{I}, Y:\tau_1\to\tau_2, M:\tau_1\to\tau_2[Y:\tau_1\to\tau_2/X:\tau_1\to\tau_2][t:\tau/x:\tau], \eta)(h)$

$= \mathcal{I}\left\{\frac{h}{Y:\tau_1\to\tau_2}\right\}[\![M:\tau_1\to\tau_2[Y:\tau_1\to\tau_2/X:\tau_1\to\tau_2][t:\tau/x:\tau]]\!]\eta$

$= \mathfrak{I}\left\{\frac{h}{Y:\tau_1\to\tau_2}\right\}\left\{\frac{\mathfrak{I}[t:\tau]}{x:\tau}\right\}[M:\tau_1\to\tau_2\{Y:\tau_1\to\tau_2/X:\tau_1\to\tau_2\}]\eta$ por inducción y el lema de coincidencia pues $Y:\tau_1\to\tau_2 \notin Lib(t:\tau)$

$= \mathfrak{I}\left\{\frac{h}{Y:\tau_1\to\tau_2}\right\}\left\{\frac{\mathfrak{I}[t:\tau]}{x:\tau}\right\}\left\{\frac{h}{X:\tau_1\to\tau_2}\right\}[M:\tau_1\to\tau_2]\eta$ por inducción ya que $\mathfrak{I}\left\{\frac{h}{Y:\tau_1\to\tau_2}\right\}\left\{\frac{\mathfrak{I}[t:\tau]}{x:\tau}\right\}[Y:\tau_1\to\tau_2]\eta = h$

$= \mathfrak{I}'\left\{\frac{h}{X:\tau_1\to\tau_2}\right\}[M:\tau_1\to\tau_2]\eta$ por el lema de coincidencia pues $Y:\tau_1\to\tau_2 \notin Lib(M:\tau_1\to\tau_2)$

$= T(\mathfrak{I}', X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)(h).$

(B)-(ii). Demostraremos la igualdad $\mathfrak{I}[X':\tau_1'\to\tau_2'\left[\frac{M:\tau_1\to\tau_2}{X:\tau_1\to\tau_2}\right]]\eta = \mathfrak{I}\left\{\frac{\mathfrak{I}[M:\tau_1\to\tau_2]}{X:\tau_1\to\tau_2}\right\}[X':\tau_1'\to\tau_2']\eta$, en los demás casos la demostración es muy similar a las desarrolladas hasta ahora.

Representamos $\mathfrak{I}\left\{\frac{\mathfrak{I}[M:\tau_1\to\tau_2]}{X:\tau_1\to\tau_2}\right\}$ por $\mathfrak{I}''$ y $\xi\left\{\frac{\mathfrak{I}[M:\tau_1\to\tau_2]}{X:\tau_1\to\tau_2}\right\}$ por ξ''

$- \mathfrak{I}[X':\tau_1'\to\tau_2'\{M:\tau_1\to\tau_2/X:\tau_1\to\tau_2\}]\eta$

$= \begin{cases}\mathfrak{I}[X':\tau_1'\to\tau_2']\eta & \text{si } X:\tau_1\to\tau_2 \neq X':\tau_1'\to\tau_2' \\ \mathfrak{I}[M:\tau_1\to\tau_2]\eta & \text{si } X:\tau_1\to\tau_2 = X':\tau_1'\to\tau_2'\end{cases}$

$= \begin{cases}\xi(X':\tau_1'\to\tau_2')\eta & \text{si } X:\tau_1\to\tau_2 \neq X':\tau_1'\to\tau_2' \\ \mathfrak{I}[M:\tau_1\to\tau_2]\eta & \text{si } X:\tau_1\to\tau_2 = X':\tau_1'\to\tau_2'\end{cases}$

$= \xi''(X':\tau_1'\to\tau_2')\eta$ por definición de ξ''

$= \mathfrak{I}''[X':\tau_1'\to\tau_2']\eta$

Demostramos (C)-(i) probando que para cualquier $\eta \in AT_p^{\Sigma}$, se tiene $\mathfrak{I},\eta \vdash \varphi[t:\tau/x:\tau] \iff \mathfrak{I}\{\mathfrak{I}[t:\tau]/x:\tau\},\eta \vdash \varphi$. Lo probamos por inducción en la construcción de las fórmulas:

$\mathfrak{I},\eta \vdash (t_1:\tau' \leq t_2:\tau')[t:\tau/x:\tau]$

$\iff \mathfrak{I}[t_1:\tau'[t:\tau/x:\tau]]\eta \leq^{\Sigma}_{\eta(\tau')} \mathfrak{I}[t_2:\tau'[t:\tau/x:\tau]]\eta$

$\iff \mathfrak{I}'[t_1:\tau']\eta \leq^{\Sigma}_{\eta(\tau')} \mathfrak{I}'[t_2:\tau']\eta$ por (A)-(i)

$\iff \mathfrak{I}',\eta \vdash t_1:\tau' \leq t_2:\tau'.$

$\mathfrak{I},\eta \vdash (p\ t':\tau')[t:\tau/x:\tau]$

$\iff \mathfrak{I}[t':\tau'[t:\tau/x:\tau]]\eta \in p:\tau_1{}^{\mathfrak{D}}\eta(\sigma)$ con $p:\tau_1 \in \Sigma_d$ y $\tau' = \tau_1\sigma$

$\iff \mathfrak{I}'[t':\tau']\eta \in p:\tau_1{}^{\mathfrak{D}}\eta(\sigma)$ por (A)-(i)

$\iff \mathfrak{I}',\eta \vdash (p\ t':\tau').$

$\mathfrak{I},\eta \models (\neg\varphi)[t:\tau/x:\tau] \Leftrightarrow$ no $\mathfrak{I},\eta \models \varphi[t:\tau/x:\tau]$

$\Leftrightarrow$ no $\mathfrak{I}',\eta \models \varphi$ por hipótesis de inducción

$\Leftrightarrow \mathfrak{I}',\eta \models \neg\varphi$.

$\mathfrak{I},\eta \models (\varphi \vee \psi)[t:\tau/x:\tau] \Leftrightarrow \mathfrak{I},\eta \models \varphi[t:\tau/x:\tau]$ ó $\mathfrak{I},\eta \models \psi[t:\tau/x:\tau]$

$\Leftrightarrow \mathfrak{I}',\eta \models \varphi$ ó $\mathfrak{I}',\eta \models \psi$ por hipótesis de inducción

$\Leftrightarrow \mathfrak{I}',\eta \models \varphi \vee \psi$.

$\mathfrak{I},\eta \models (\varphi \wedge \psi)[t:\tau/x:\tau] \Leftrightarrow \mathfrak{I},\eta \models \varphi[t:\tau/x:\tau]$ y $\mathfrak{I},\eta \models \psi[t:\tau/x:\tau]$

$\Leftrightarrow \mathfrak{I}',\eta \models \varphi$ y $\mathfrak{I}',\eta \models \psi$ por hipótesis de inducción

$\Leftrightarrow \mathfrak{I}',\eta \models \varphi \wedge \psi$.

$\mathfrak{I},\eta \models (\forall x':\tau'\ \varphi)[t:\tau/x:\tau]$

$\Leftrightarrow \mathfrak{I}\{a/z:\tau'\},\eta \models \varphi[z:\tau'/x':\tau'][t:\tau/x:\tau]$ p.todo $a \in [\tau']^{\mathfrak{I}}\eta$

$\Leftrightarrow \mathfrak{I}\{a/z:\tau'\}\{\mathfrak{I}[t:\tau]/x:\tau\},\eta \models \varphi[z:\tau'/x':\tau']$ p.t. $a \in [\tau']^{\mathfrak{I}}\eta$

por hip. induc. y el lema de coincid. pues $z:\tau' \notin Lib(t:\tau)$

$\Leftrightarrow \mathfrak{I}\{a/z:\tau'\}\{\mathfrak{I}[t:\tau]/x:\tau\}\{a/x':\tau'\},\eta \models \varphi$ p.t. $a \in [\tau']^{\mathfrak{I}}\eta$ por

inducción pues $\mathfrak{I}\{a/z:\tau'\}\left\{\frac{\mathfrak{I}[t:\tau]}{x:\tau}\right\}[z:\tau']\eta = a$

$\Leftrightarrow \mathfrak{I}'\{a/x':\tau'\},\eta \models \varphi$ para todo $a \in [\tau']^{\mathfrak{I}}\eta$ por el lema

de coincidencia ya que $z:\tau' \notin Lib(\varphi)$

$\Leftrightarrow \mathfrak{I}',\eta \models \forall x':\tau'\varphi$.

$\mathfrak{I},\eta \models (\exists x':\tau'\ \varphi)[t:\tau/x:\tau]$

$\Leftrightarrow$ ex. $a \in [\tau']^{\mathfrak{I}}\eta$ $\mathfrak{I}\{a/z:\tau'\},\eta \models \varphi[z:\tau'/x':\tau'][t:\tau/x:\tau]$

$\Leftrightarrow$ existe un $a \in [\tau']^{\mathfrak{I}}\eta$ tal que

$\mathfrak{I}\{a/z:\tau'\}\{\mathfrak{I}[t:\tau]/x:\tau\},\eta \models \varphi[z:\tau'/x':\tau']$ por induc.

y el lema de coincid. pues $z:\tau' \notin Lib(t:\tau)$

$\Leftrightarrow$ ex. $a \in [\tau']^{\mathfrak{I}}\eta$ t.q. $\mathfrak{I}\{a/z:\tau'\}\{\mathfrak{I}[t:\tau]/x:\tau\}\{a/x':\tau'\},\eta \models \varphi$

por inducción ya que $\mathfrak{I}\{a/z:\tau'\}\left\{\frac{\mathfrak{I}[t:\tau]}{x:\tau}\right\}[z:\tau']\eta = a$

$\Leftrightarrow$ ex. $a \in [\tau']^{\mathfrak{I}}\eta$ t.q. $\mathfrak{I}'\{a/x':\tau'\},\eta \models \varphi$ por el lema de

coincidencia ya que $z:\tau' \notin Lib(\varphi)$

$\Leftrightarrow \mathfrak{I}',\eta \models \exists x':\tau'\varphi$.

(C)-(ii) se demuestra análogamente por inducción en la construcción de las fórmulas y apoyándonos en el resultado (A)-(ii). ∎

Lema 3.1.2 (Sustitución de tipos)

Sean Σ una signatura, $\mathfrak{I}=\langle \mathfrak{I}, \mathfrak{D}, \xi \rangle$ una Σ-interpretación, η una valoración para $\mathfrak{I}$ y σ una Σ-sustitución de tipos. Entonces:

(1) $\mathfrak{I}[[t:\tau]\sigma]\eta = \mathfrak{I}[t:\tau]\eta(\sigma)$, para cualquier $t:\tau \in T_{\sigma}(\Sigma)$.

(2) $\mathcal{I}[\![[M:\tau_1\to\tau_2]\sigma]\!]\eta = \mathcal{I}[\![M:\tau_1\to\tau_2]\!]\eta(\sigma)$, para cualquier $M:\tau_1\to\tau_2$ de $Er(\Sigma)$.

(3) $\mathcal{I},\eta \models \varphi\sigma \Leftrightarrow \mathcal{I},\eta(\sigma) \models \varphi$, para cualquier $\varphi \in L(\Sigma)$.

Demostración:

Sabemos que $[\![\tau\sigma]\!]^{\mathcal{I}}\eta = [\![\tau]\!]^{\mathcal{I}}\eta(\sigma)$ (por 2.2.3). Apoyándonos en este hecho, demostramos (1) y (2) por inducción mutua en la construcción de los términos y de las expresiones funcionales.

- $\mathcal{I}[\![[x:\tau]\sigma]\!]\eta = \xi(x:\tau\sigma)\eta = \xi(x:\tau)\eta(\sigma) = \mathcal{I}[\![x:\tau]\!]\eta(\sigma)$.

- $\mathcal{I}[\![[\bot:\tau]\sigma]\!]\eta = \bot^{\mathcal{I}}_{\eta(\tau\sigma)} = \bot^{\mathcal{I}}_{\eta(\sigma)(\tau)} = \mathcal{I}[\![\bot:\tau]\!]\eta(\sigma)$.

- $\mathcal{I}[\![[c:\tau_1]\sigma]\!]\eta = c:\tau^{\mathcal{D}}\eta(\sigma\sigma_1)$ donde $c:\to\tau \in \Sigma_d$ y $\tau_1=\tau\sigma_1$
  $= c:\tau^{\mathcal{D}}\eta(\sigma)(\sigma_1) = \mathcal{I}[\![c:\tau]\!]\eta(\sigma)$

- $\mathcal{I}[\![[true]\sigma]\!]\eta = tt = \mathcal{I}[\![true]\!]\eta(\sigma)$.

- $\mathcal{I}[\![[false]\sigma]\!]\eta = ff = \mathcal{I}[\![false]\!]\eta(\sigma)$.

- $\mathcal{I}[\![[(t_1:\tau_1,\ldots,t_n:\tau_n)]\sigma]\!]\eta = \langle\mathcal{I}[\![[t_1:\tau_1]\sigma]\!]\eta,\ldots,\mathcal{I}[\![[t_n:\tau_n]\sigma]\!]\eta\rangle$
  $= \langle\mathcal{I}[\![t_1:\tau_1]\!]\eta(\sigma),\ldots,\mathcal{I}[\![t_n:\tau_n]\!]\eta(\sigma)\rangle$ por hipótesis de inducción
  $= \mathcal{I}[\![(t_1:\tau_1,\ldots,t_n:\tau_n)]\!]\eta(\sigma)$ si $\mathcal{I}[\![[t_i:\tau_i]\sigma]\!]\eta = \bot_{\eta(\tau_i\sigma)}$ $(1\le i\le n)$.
  Para el caso $\bot$ la demostración es trivial

- $\mathcal{I}[\![[(\text{if } t:\text{bool then } t_1:\tau_1 \text{ else } t_2:\tau_1)]\sigma]\!]\eta =$

$$= \begin{cases} \mathcal{I}[\![[t_1:\tau_1]\sigma]\!]\eta & \text{si } \mathcal{I}[\![[t:\text{bool}]\sigma]\!]\eta = tt \\ \mathcal{I}[\![[t_2:\tau_1]\sigma]\!]\eta & \text{si } \mathcal{I}[\![[t:\text{bool}]\sigma]\!]\eta = ff \\ \bot^{\mathcal{I}}_{\eta(\tau_1\sigma)} & \text{si } \mathcal{I}[\![[t:\text{bool}]\sigma]\!]\eta = \bot_{bool} \end{cases}$$

$$= \begin{cases} \mathcal{I}[\![t_1:\tau_1]\!]\eta(\sigma) & \text{si } \mathcal{I}[\![t:\text{bool}]\!]\eta(\sigma) = tt \\ \mathcal{I}[\![t_2:\tau_1]\!]\eta(\sigma) & \text{si } \mathcal{I}[\![t:\text{bool}]\!]\eta(\sigma) = ff \\ \bot_{\eta(\sigma)(\tau_1)} & \text{si } \mathcal{I}[\![t:\text{bool}]\!]\eta(\sigma) = \bot_{bool} \end{cases}$$ por hipótesis de inducción

  $= \mathcal{I}[\![(\text{if } t:\text{bool then } t_1:\tau_1 \text{ else } t_2:\tau_1)]\!]\eta(\sigma)$.

- $\mathcal{I}[\![[(M\ t:\tau_1):\tau_2]\sigma]\!]\eta = \mathcal{I}[\![[M:\tau_1\to\tau_2]\sigma]\!]\eta\ (\mathcal{I}[\![[t:\tau_1]\sigma]\!]\eta)$
  $= \mathcal{I}[\![M:\tau_1\to\tau_2]\!]\eta(\sigma)\ (\mathcal{I}[\![[t:\tau_1]\sigma]\!]\eta)$ por hipótesis de induc. en $M:\tau_1\to\tau_2$
  $= \mathcal{I}[\![M:\tau_1\to\tau_2]\!]\eta(\sigma)\ (\mathcal{I}[\![t:\tau_1]\!]\eta(\sigma))$ por hipótesis de inducción en $t:\tau_1$
  $= \mathcal{I}[\![(M\ t:\tau_1):\tau_2]\!]\eta(\sigma)$.

- $\mathcal{I}[\![[\bot:\tau_1\to\tau_2]\sigma]\!]\eta = \bot_{\eta((\tau_1\to\tau_2)\sigma)} = \bot_{\eta(\sigma)(\tau_1\to\tau_2)}$
  $= \mathcal{I}[\![\bot:\tau_1\to\tau_2]\!]\eta(\sigma)$.

- $\mathcal{I}[\![[f:\tau_1'\to\tau_2']\sigma]\!]\eta = f:\tau_1\to\tau_2^{\mathcal{D}}\eta(\sigma\sigma_1) = \mathcal{I}[\![f:\tau_1'\to\tau_2']\!]\eta(\sigma)$ donde
  $f:\tau_1\to\tau_2 \in \Sigma_d$ y $\tau_1'\to\tau_2' = (\tau_1\to\tau_2)\sigma_1$

- $\mathcal{I}[\![[X:\tau_1\to\tau_2]\sigma]\!]\eta = \xi(X:(\tau_1\to\tau_2)\sigma))\eta = \xi(X:\tau_1\to\tau_2)\eta(\sigma)$ por definición
  $= \mathcal{I}[\![X:\tau_1\to\tau_2]\!]\eta(\sigma)$

- $\mathcal{I}[\![[(\lambda x_1:\tau_1\ldots x_n:\tau_n.\,t:\tau)]\sigma]\!]\eta$ $(d_1,\ldots,d_n)$ con $\langle d_1,\ldots,d_n\rangle$ una n-tupla de $[\![\tau_1\sigma]\!]^{\mathcal{I}}\eta\bullet\ldots\bullet[\![\tau_n\sigma]\!]^{\mathcal{I}}\eta\setminus\{\perp_\bullet\} = [\![\tau_1]\!]^{\mathcal{I}}\eta(\sigma)\bullet\ldots\bullet[\![\tau_n]\!]^{\mathcal{I}}\eta(\sigma)\setminus\{\perp_\bullet\}$. Haremos la demostración para el caso en que la sustitución se realiza sin necesidad de renombrar las variables de dato; en el caso de un renombramiento previo, la demostración sería similar, pero apoyándonos también en el lema de sustitución de datos (3.1.1).

  $= \mathcal{I}\left\{\frac{d_1\ldots d_n}{x_1:\tau_1\sigma\ldots x_n:\tau_n\sigma}\right\}[\![[t:\tau]\sigma]\!]\eta$

  $= \mathcal{I}\left\{\frac{d_1\ldots d_n}{x_1:\tau_1\ldots x_n:\tau_n}\right\}[\![t:\tau]\!]\eta(\sigma)$ por inducción sobre $t:\tau$ y porque

  $$\xi\left\{\frac{d_1\ldots d_n}{x_1:\tau_1\sigma\ldots x_n:\tau_n\sigma}\right\}(y:\tau\sigma)\eta = \xi\left\{\frac{d_1\ldots d_n}{x_1:\tau_1\ldots x_n:\tau_n}\right\}(y:\tau)\eta(\sigma)$$

  Esta última afirmación se debe a las condiciones de la definición de valoración y al hecho de que dos variables con el mismo símbolo no pueden tener tipos distintos y unificables.

  $= \mathcal{I}[\![(\lambda x_1:\tau_1\ldots x_n:\tau_n.\,t:\tau)]\!]\eta(\sigma)$ $(d_1,\ldots,d_n)$

- $\mathcal{I}[\![[(\mu X:\tau_1\to\tau_2.M)]\sigma]\!]\eta = \mathrm{fix}\ T(\mathcal{I},[X:\tau_1\to\tau_2]\sigma,[M:\tau_1\to\tau_2]\sigma,\eta)$
  Sea $h \in [[\tau_1\sigma]]^{\mathcal{I}}\eta \underset{\bullet}{\longrightarrow} [[\tau_2\sigma]]^{\mathcal{I}}\eta] = [[\tau_1]]^{\mathcal{I}}\eta(\sigma) \underset{\bullet}{\longrightarrow} [[\tau_2]]^{\mathcal{I}}\eta(\sigma)]$

  $T(\mathcal{I},X:(\tau_1\to\tau_2)\sigma,[M:\tau_1\to\tau_2]\sigma,\eta)(h)$

  $= \mathcal{I}\left\{\frac{h}{X:(\tau_1\to\tau_2)\sigma}\right\}[\![[M:\tau_1\to\tau_2]\sigma]\!]\eta$

  $= \mathcal{I}\left\{\frac{h}{X:\tau_1\to\tau_2}\right\}[\![M:\tau_1\to\tau_2]\!]\eta(\sigma)$ por inducción sobre $M:\tau_1\to\tau_2$

  ya que al igual que en el caso anterior se tiene

  $$\xi\left\{\frac{h}{X:\tau_1\sigma\to\tau_2\sigma}\right\}(Y:\tau_1'\sigma\to\tau_2'\sigma)\eta = \xi\left\{\frac{h}{X:\tau_1\to\tau_2}\right\}(Y:\tau_1'\to\tau_2')\eta(\sigma)$$

  $= T(\mathcal{I},X:\tau_1\to\tau_2,M:\tau_1\to\tau_2,\eta(\sigma))(h)$

  luego $\mathcal{I}[\![[(\mu X:\tau_1\to\tau_2.M)]\sigma]\!]\eta = \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta(\sigma)$.

Demostramos (3) por inducción en la construcción de las fórmulas. Supondremos que no es necesario un renombramiento de variables en el caso de las fórmulas cuantificadas. En caso contrario la demostración sería análoga utilizando también el lema de sustitución de datos.

$$\begin{aligned}\mathcal{I},\eta \models [t_1:\tau \le t_2:\tau]\sigma &\iff \mathcal{I}[\![[t_1:\tau]\sigma]\!]\eta \le^{\mathcal{I}}_{\eta(\tau\sigma)} \mathcal{I}[\![[t_2:\tau]\sigma]\!]\eta\\ &\iff \mathcal{I}[\![t_1:\tau]\!]\eta(\sigma) \le^{\mathcal{I}}_{\eta(\sigma)(\tau)} \mathcal{I}[\![t_2:\tau]\!]\eta(\sigma) \qquad \text{por (1)}\\ &\iff \mathcal{I},\eta(\sigma) \models t_1:\tau \le t_2:\tau.\end{aligned}$$

$\mathfrak{I},\eta \models [(p\ t:\tau')]\sigma$ $\Leftrightarrow \mathfrak{I}\big[[t:\tau']\sigma\big]\eta \in p:\tau^{\mathfrak{D}}\eta\{\sigma\sigma'\}$ si $p:\tau \in \Sigma_d$, $\tau'=\tau\sigma'$

$\Leftrightarrow \mathfrak{I}\big[t:\tau'\big]\eta\{\sigma\} \in p:\tau^{\mathfrak{D}}\eta\{\sigma\}\{\sigma'\}$ por induc. en $t:\tau'$

$\Leftrightarrow \mathfrak{I},\eta\{\sigma\} \models (p\ t:\tau')$.

$\mathfrak{I},\eta \models [\neg\varphi]\sigma$ $\Leftrightarrow$ no $\mathfrak{I},\eta \models \varphi\sigma$

$\Leftrightarrow$ no $\mathfrak{I},\eta\{\sigma\} \models \varphi$ por hipótesis de inducción

$\Leftrightarrow \mathfrak{I},\eta\{\sigma\} \models \neg\varphi$.

$\mathfrak{I},\eta \models [\varphi \vee \psi]\sigma$ $\Leftrightarrow \mathfrak{I},\eta \models \varphi\sigma$ ó $\mathfrak{I},\eta \models \psi\sigma$

$\Leftrightarrow \mathfrak{I},\eta\{\sigma\} \models \varphi$ ó $\mathfrak{I},\eta\{\sigma\} \models \psi$ por hip. de induc.

$\Leftrightarrow \mathfrak{I},\eta\{\sigma\} \models \varphi \vee \psi$

$\mathfrak{I},\eta \models [\varphi \wedge \psi]\sigma$ $\Leftrightarrow \mathfrak{I},\eta \models \varphi\sigma$ y $\mathfrak{I},\eta \models \psi\sigma$

$\Leftrightarrow \mathfrak{I},\eta\{\sigma\} \models \varphi$ y $\mathfrak{I},\eta\{\sigma\} \models \psi$ por hip. de induc.

$\Leftrightarrow \mathfrak{I},\eta\{\sigma\} \models \varphi \wedge \psi$

$\mathfrak{I},\eta \models [\forall x:\tau\ \varphi]\sigma$ $\Leftrightarrow \mathfrak{I}\{a/x:\tau\sigma\},\eta \models \varphi\sigma$ para todo $a \in \big[\tau\sigma\big]^{\mathfrak{I}}\eta$

$\Leftrightarrow \mathfrak{I}\{a/x:\tau\},\eta\{\sigma\} \models \varphi$ p.t. $a \in \big[\tau\big]^{\mathfrak{I}}\eta\{\sigma\}$ por hip. de induc. y porque $\xi\{a/x:\tau\sigma\}(y:\tau'\sigma)\eta = \xi\{a/x:\tau\}(y:\tau')\eta\{\sigma\}$ para cualquier $y:\tau'$

$\Leftrightarrow \mathfrak{I},\eta\{\sigma\} \models \forall x:\tau\ \varphi$.

$\mathfrak{I},\eta \models [\exists x:\tau\ \varphi]\sigma$ $\Leftrightarrow$ existe un $a \in \big[\tau\sigma\big]^{\mathfrak{I}}\eta$ tal que $\mathfrak{I}\{a/x:\tau\sigma\},\eta \models \varphi\sigma$

$\Leftrightarrow$ ex. un $a \in \big[\tau\big]^{\mathfrak{I}}\eta\{\sigma\}$ tal que $\mathfrak{I}\{a/x:\tau\},\eta\{\sigma\} \models \varphi$ por inducción en φ ya que para todo $y:\tau'$ $\xi\{a/x:\tau\sigma\}(y:\tau'\sigma)\eta = \xi\{a/x:\tau\}(y:\tau')\eta\{\sigma\}$

$\Leftrightarrow \mathfrak{I},\eta\{\sigma\} \models \exists x:\tau\ \varphi$. ■

Teorema 3.1.3

Sean $\varphi \in L(\Sigma)$ y $\mathfrak{I}$ una Σ-interpretación, entonces $\mathfrak{I} \models \varphi \Leftrightarrow \mathfrak{I},\eta \models \varphi$ para toda asignación η de las variables de tipo que aparecen en φ.

Este resultado es inmediato tal y como se ha definido la satisfactibilidad de una Σ-fórmula para una asignación y por la definición de la semántica de los términos y el concepto de modelo. ■

Obsérvese que si una Σ-fórmula φ es monomórfica y $\mathfrak{I}=\langle \mathcal{I}, \mathcal{D}, \xi\rangle$ es una Σ-interpretación, $\mathfrak{I},\eta \models \varphi$ es equivalente a $\mathfrak{I} \models \varphi$ para cualquier asignación η para $\mathcal{I}$. El concepto de modelo en estos casos es independiente de las asignaciones de tipo y coincide con el de satisfactibilidad.

En lo que sigue, si $\Sigma=\langle \Sigma_t, \Sigma_d\rangle$ y $\Sigma'=\langle \Sigma_t', \Sigma_d'\rangle$ son dos signaturas

tales que $\Sigma_t \subseteq \Sigma_t'$ y $\Sigma_d \subseteq \Sigma_d'$, diremos que una Σ'-interpretación $\mathfrak{I}' = \langle \mathcal{I}', \mathcal{D}', \xi \rangle$ es una extensión de una Σ-interpretación $\mathfrak{I} = \langle \mathcal{I}, \mathcal{D}, \xi \rangle$, si $\mathfrak{I}'$ coincide con $\mathfrak{I}$ en los Σ-símbolos y añade valores para el resto de los elementos de Σ', siendo además $T^{\mathcal{I}} = T^{\mathcal{I}'}$.

Lema 3.1.4

Sean $\Sigma = \langle \Sigma_t, \Sigma_d \rangle$, $\varphi \in L(\Sigma)$, $\sigma = ct'/\rho$ $(ct'/0 \notin \Sigma_t)$, $\Sigma' = \langle \Sigma_t \cup \{ct'/0\}, \Sigma_d \rangle$ y ρ una variable de tipo de φ. Entonces para cualquier Σ-interpretación $\mathfrak{I} = \langle \mathcal{I}, \mathcal{D}, \xi \rangle$ y para cualquier Σ'-interpretación $\mathfrak{I}'$ que sea una extensión de $\mathfrak{I}$, se verifica:

$\mathfrak{I} \models \varphi \Leftrightarrow \mathfrak{I}' \models \varphi\sigma$.

Demostración:

$\Leftarrow$) Sea $D \in T^{\mathcal{I}}$ un dominio cualquiera, $\mathfrak{I}\{D/ct'\} = \langle \mathcal{I}\{D/ct'\}, \mathcal{D}, \xi \rangle$ donde $\mathcal{I}\{D/ct'\} = \langle T^{\mathcal{I}}, \{ct^{\mathcal{I}}\}_{ct/n \in \Sigma_t} \cup ct'^{\mathcal{I}\{D/ct'\}} \rangle$ siendo $ct'^{\mathcal{I}\{D/ct'\}} = D$. El teorema es una consecuencia de la siguiente equivalencia:

$$\mathfrak{I}\{D/ct'\}, \eta\{ct'/\rho\} \models \varphi \Leftrightarrow \mathfrak{I}, \eta\{D/\rho\} \models \varphi \qquad (*)$$

Para conseguir la equivalencia (*), se prueba primero que, para cualquier $\tau \in T_p(\Sigma)$, $[\![\tau]\!]^{\mathcal{I}\{D/ct'\}}\eta\{ct'/\rho\} = [\![\tau]\!]^{\mathcal{I}}\eta\{D/\rho\}$. En concreto para el caso $\tau = \rho'$:

$$[\![\rho']\!]^{\mathcal{I}\{D/ct'\}}\eta\{ct'/\rho\} = \eta\{ct'/\rho\}(\rho')$$

$$= \begin{cases} [\![ct']\!]^{\mathcal{I}\{D/ct'\}}\eta & \text{si } \rho = \rho' \\ \eta(\rho') & \text{si } \rho \neq \rho' \end{cases} = \begin{cases} ct'^{\mathcal{I}\{D/ct'\}} = D & \text{si } \rho = \rho' \\ \eta(\rho') & \text{si } \rho \neq \rho' \end{cases}$$

$$= \eta\{D/\rho\}(\rho') = [\![\rho']\!]^{\mathcal{I}}\eta\{D/\rho\}$$

El resto de los casos se deducen por inducción en la construcción de los Σ-tipos de primer orden. Una vez probada esta igualdad para todo $\tau \in T_p(\Sigma)$, (*) se obtiene haciendo inducción sobre la construcción de los Σ-términos y las Σ-fórmulas.

Ahora bien, por el lema de sustitución de tipos, la equivalencia anterior se reduce a la que sigue, $\mathfrak{I}\{D/ct'\}, \eta \models \varphi\sigma \Leftrightarrow \mathfrak{I}, \eta\{D/\rho\} \models \varphi$ para toda $\eta \in AT_p^{\mathcal{I}} = AT_p^{\mathcal{I}'}$ y para todo $D \in T^{\mathcal{I}}$, con lo que $\mathfrak{I}\{D/ct'\} \models \varphi\sigma \Leftrightarrow \mathfrak{I} \models \varphi$ para todo $D \in T^{\mathcal{I}}$, o lo que es lo mismo $\mathfrak{I}' \models \varphi\sigma \Leftrightarrow \mathfrak{I} \models \varphi$ para toda Σ'-interpretación $\mathfrak{I}'$, extensión de $\mathfrak{I}$. ∎

Obviamente este resultado es igualmente cierto si σ es una sustitución multiple $\frac{ct_1 \ldots ct_n}{\rho_1 \ldots \rho_n}$ siendo ct_i $(1 \leq i \leq n)$ constantes de tipo nuevas y ρ_i $(1 \leq i \leq n)$ variables de tipo de φ para $1 \leq j \leq n$.

Teorema 3.1.5

Sean Σ una signatura, $\Phi \subseteq L(\Sigma)$, $\varphi \in L(\Sigma)$ y $\sigma = \frac{ct_1 \ldots ct_n}{\rho_1 \ldots \rho_n}$ con ct_i $(1 \leq i \leq n)$ constantes de tipo que no aparecen en Φ ni en φ y tal que $\varphi\sigma$ es monomórfica. Entonces $\Phi \vdash \varphi \Leftrightarrow \Phi \cup \{\neg\varphi\sigma\}$ es insatisfactible.

Demostración:

$\Rightarrow$) Supongamos $\Phi \vdash \varphi$ y $Sat(\Phi \cup \{\neg\varphi\sigma\})$ siendo $\mathfrak{I}=\langle \mathcal{I}, \mathcal{D}, \xi \rangle$ una interpretación tal que $\mathfrak{I} \vdash \Phi \cup \{\neg\varphi\sigma\}$. Por ser $\Phi \vdash \varphi$ y $\mathfrak{I} \vdash \Phi$ tendremos $\mathfrak{I} \vdash \varphi$ es decir $\mathfrak{I}, \eta \vdash \varphi$ para toda $\eta \in AT_p^{\mathcal{I}}$, pero por hipótesis $\mathfrak{I} \vdash \neg\varphi\sigma$ que por el lema de sustitución de tipos implica que para toda η de $AT_p^{\mathcal{I}}$ no $\mathfrak{I}, \eta\{\sigma\} \vdash \varphi$ lo cual se contradice con $\mathfrak{I} \vdash \varphi$. Concluimos entonces que $\Phi \cup \{\neg\varphi\sigma\}$ es insatisfactible.

$\Leftarrow$) Si $\Phi \cup \{\neg\varphi\sigma\}$ es insatisfactible, como $\varphi\sigma$ es monomórfica, no puede existir una interpretación $\mathfrak{I}$ que de valores a ct_i $(1 \leq i \leq n)$ y tal que $\mathfrak{I} \vdash \Phi$ y no $\mathfrak{I} \vdash \varphi\sigma$, por tanto, $\mathfrak{I} \vdash \Phi \Rightarrow \mathfrak{I} \vdash \varphi\sigma$ y, en base al lema anterior, $\mathfrak{I} \vdash \varphi$ con lo que $\Phi \vdash \varphi$. ■

Teorema 3.1.6

Sea $\mathfrak{I}=\langle \mathcal{I}, \mathcal{D}, \xi \rangle$ una Σ-interpretación tal que $T^{\mathcal{I}}$ coincide con la colección de dominios asociados a los Σ-tipos monomórficos de primer orden, y sea φ una Σ-fórmula cualquiera, entonces $\mathfrak{I} \vdash \varphi \Leftrightarrow \mathfrak{I} \vdash \varphi\sigma$ para toda Σ-sustitución de tipos σ tal que $\varphi\sigma$ es monomórfica.

Demostración:

$\Rightarrow$) Supongamos $\mathfrak{I} \vdash \varphi$, es decir, $\mathfrak{I}, \eta \vdash \varphi$ para toda $\eta \in AT_p^{\mathcal{I}}$ luego $\mathfrak{I}, \eta\{\sigma\} \vdash \varphi$ para toda $\eta \in AT_p^{\mathcal{I}}$ y toda Σ-sustitución de tipos σ tal que $\varphi\sigma$ es monomórfica. Por tanto, aplicando el lema de sustitución de tipos, $\mathfrak{I}, \eta \vdash \varphi\sigma$ para toda asignación η y toda Σ-sustitución σ tal que $\varphi\sigma$ es monomórfica, es decir, $\mathfrak{I} \vdash \varphi\sigma$ para toda Σ-sustitución de tipos σ tal que $\varphi\sigma$ es monomórfica.

$\Leftarrow$) Si $\mathfrak{I}, \eta \vdash \varphi\sigma$ para toda $\eta \in AT_p^{\mathcal{I}}$ y toda Σ-sustitución de tipos σ que hace monomórfica a $\varphi\sigma$, por el lema de sustitución de tipos tendremos $\mathfrak{I}, \eta\{\sigma\} \vdash \varphi$ para toda $\eta \in AT_p^{\mathcal{I}}$ y toda Σ-sustitución de tipos σ tal que $\varphi\sigma$ es monomórfica. Entonces, puesto que $T^{\mathcal{I}}$ coincide con la familia $\{D_\nu^{\mathcal{I}}\}_{\nu \in MT_p(\Sigma)}$, $\mathfrak{I}, \eta \vdash \varphi$ para todas las asignaciones para $\mathcal{I}$ de las variables de tipo de φ y, por 3.1.3, $\mathfrak{I} \vdash \varphi$. ■

En general, la complejidad del problema de validez de una Σ-fórmula, se reduce considerablemente si nos restringimos a los modelos con Σ-estructuras de tipos $\mathcal{I}$ tales que para todo $D \in T^{\mathcal{I}}$ existe

un tipo $\nu \in MT_P(\Sigma)$ tal que $D = D_\nu^I = [\![\nu]\!]^I$. Un ejemplo de este hecho es el teorema anterior. Además para estas estructuras de tipos, cada asignación de tipos η determinan de manera unívoca una sustitución de tipos, que por abuso del lenguaje también llamamos η, escribiremos $c:\tau\eta^D$ para indicar el valor de la función $c:\tau^D$ en η si $c:\rightarrow\tau \in \Sigma_d$ y lo mismo haremos con $f:\tau_1\rightarrow\tau_2{}^D$ y $p:\tau^D$. Entonces, las estructuras de datos correspondientes a estructuras de tipos con estas propiedades quedan determinadas si se conocen los valores que toman los símbolos de constantes, funciones y predicados para cada una de las instancias monomórficas de su tipo más general, es decir, para conocer la función $c:\tau^D$, si $c:\rightarrow\tau \in \Sigma_d$, será suficiente con conocer la sucesión de valores denotados $\{c:\nu^D \in D_\nu^I \mid \nu \leq \tau\}$; lo mismo ocurre con $f:\tau_1\rightarrow\tau_2{}^D$ y con $p:\tau^D$.

3.2 EQUIVALENCIA ENTRE APROXIMACIONES SINTACTICAS Y SEMANTICAS

A continuación vamos a definir unas expresiones que llamaremos aproximaciones sintácticas, que servirán para poder aplicar ciertos resultados sobre el menor punto fijo de un operador continuo que, como es sabido, puede caracterizarse como el supremo de la cadena creciente de las sucesivas aplicaciones al *bottom*. En PLPR los elementos de ésta cadena se pueden denotar mediante ciertas expresiones funcionales que son precisamente las aproximaciones sintácticas de $(\mu X:\tau_1\rightarrow\tau_2.M)$ que definimos a continuación.

Definicion 3.2.1

Las *aproximaciones sintácticas* i-*ésimas* de un término y de una expresión funcional se definen por recursión mutua:

$(\bot:\tau)^i := \bot:\tau$

$(x:\tau)^i := x:\tau$

$(c:\tau)^i := c:\tau$

$(\mathtt{true})^i := \mathtt{true} \qquad (\mathtt{false})^i := \mathtt{false}$

$((t_1:\tau_1,..,t_n:\tau_n))^i := ((t_1:\tau_1)^i,..,(t_n:\tau_n)^i)$

$(\mathtt{if}\ t:\mathtt{bool}\ \mathtt{then}\ t_1:\tau\ \mathtt{else}\ t_2:\tau)^i :=$

$\quad (\mathtt{if}\ (t:\mathtt{bool})^i\ \mathtt{then}\ (t_1:\tau)^i\ \mathtt{else}\ (t_2:\tau)^i)$

$(M:\tau_1\rightarrow\tau_2\ t:\tau_1)^i := ((M:\tau_1\rightarrow\tau_2)^i\ (t:\tau_1)^i)$

$(\bot:\tau_1\to\tau_2)^1 := \bot:\tau_1\to\tau_2$

$(X:\tau_1\to\tau_2)^1 := X:\tau_1\to\tau_2$

$(f:\tau_1\to\tau_2)^1 := f:\tau_1\to\tau_2$

$(\lambda x_1:\tau_1\ldots x_n:\tau_n.\,t:\tau)^1 := (\lambda x_1:\tau_1\ldots x_n:\tau_n.\,(t:\tau)^1)$

$$\begin{cases}(\mu X:\tau_1\to\tau_2.M)^0 := \bot:\tau_1\to\tau_2\\ (\mu X:\tau_1\to\tau_2.M)^{i+1} := (M:\tau_1\to\tau_2)^{i+1}\,[(\mu X:\tau_1\to\tau_2.M)^i/X:\tau_1\to\tau_2].\end{cases}$$ □

De esta definición se deduce que si en un término $t:\tau$ no aparece ninguna definición recursiva entonces $t:\tau$ coincide con todas sus aproximaciones sintácticas es decir para todo $i<\omega$ $(t:\tau)^i = t:\tau$. En estos casos se dice que $t:\tau$ es simple, en caso contrario $t:\tau$ es recursivo. Hay que observar también que, para cualquier $i<\omega$, $(\mu X:\tau_1\to\tau_2.M)^i$ no contiene en su interior ningún μ-operador.

Una vez definidas las aproximaciones sintácticas se definen unas aproximaciones semánticas que coinciden con las sucesivas aplicaciones del operador T, asociado a la semántica del μ-operador, al elemento *bottom*. Por tanto, el supremo de estas aproximaciones coincidirá con la interpretación de una μ-expresión. Se prueba también que tanto para cualquier $t:\tau$ como para cualquier $M:\tau_1\to\tau_2$, sus aproximaciones denotan para cada $\mathfrak{I}$ y cada η una cadena creciente cuya cota superior mínima coincide con $\mathfrak{I}[\![t:\tau]\!]\eta$ y $\mathfrak{I}[\![M:\tau_1\to\tau_2]\!]\eta$, respectivamente. Podemos deducir por tanto la equivalencia entre la semántica de las aproximaciones sintácticas y las aproximaciones semánticas de un operador.

Definición 3.2.2

Sea $\mathfrak{I}=\langle \mathcal{I}, \mathcal{D}, \xi\rangle$ una Σ-interpretación, dada $M:\tau_1\to\tau_2 \in Er(\Sigma)$ con $X:\tau_1\to\tau_2$ libre en $M:\tau_1\to\tau_2$, la función $\langle(\mu X:\tau_1\to\tau_2.M)\rangle^1$ que a cada asignación η para $\mathcal{I}$ le hace corresponder una función de $[[\tau_1]]^{\mathcal{I}}\eta \xrightarrow[\bullet]{} [[\tau_2]]^{\mathcal{I}}\eta]$ se define por inducción sobre i como sigue:

$$\langle(\mu X:\tau_1\to\tau_2.M)\rangle^0\eta = \bot_{\eta(\tau_1\to\tau_2)}$$

$$\langle(\mu X:\tau_1\to\tau_2.M)\rangle^{i+1}\eta = \mathfrak{I}\left\{\frac{\langle(\mu X:\tau_1\to\tau_2.M)\rangle^i\eta}{X:\tau_1\to\tau_2}\right\}[\![M:\tau_1\to\tau_2]\!]\eta.$$

Donde $\mathfrak{I}\left\{\frac{\langle(\mu X:\tau_1\to\tau_2.M)\rangle^i\eta}{X:\tau_1\to\tau_2}\right\}$ coincide con $\mathfrak{I}$ salvo en $X:\tau_1\to\tau_2$ y η siendo

$$\mathfrak{I}\left\{\frac{\langle(\mu X:\tau_1\to\tau_2.M)\rangle^i\eta}{X:\tau_1\to\tau_2}\right\}[\![X:\tau_1\to\tau_2]\!]\eta = \langle(\mu X:\tau_1\to\tau_2.M)\rangle^i\eta.$$ □

Proposición 3.2.3

Para cualesquiera $\mathcal{I}=\langle I, \mathcal{D}, \xi\rangle$, $M:\tau_1\to\tau_2 \in Ef(\Sigma)$, $X:\tau_1\to\tau_2$ libre en $M:\tau_1\to\tau_2$ y $\eta \in AT_p^I$ se verifica:

$\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta = \bigsqcup\{<(\mu X:\tau_1\to\tau_2.M)>^i\eta \mid i<\omega\}$.

Demostración:

Utilizando el teorema de existencia del menor punto fijo de un operador continuo y las propiedades del operador T asociado a la sémantica del μ-operador, para cualquier $\eta \in AT_p^I$, podemos asegurar la siguiente igualdad.

$\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta = \bigsqcup\{T(\mathcal{I}, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)^i(\perp_{\eta(\tau_1\to\tau_2)}) \mid i<\omega\}$

Por tanto si demostramos:

$<(\mu X:\tau_1\to\tau_2.M)>^i\eta = T(\mathcal{I}, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)^i(\perp_{\eta(\tau_1\to\tau_2)})$

el teorema estará resuelto. Lo demostramos por inducción sobre i.

- $<(\mu X:\tau_1\to\tau_2.M)>^0\eta = \perp_{\eta(\tau_1\to\tau_2)}$

  $= T(\mathcal{I}, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)^0(\perp_{\eta(\tau_1\to\tau_2)})$

Paso de inducción. Supuesto cierto para el caso i-ésimo lo probamos para el i+1-ésimo. Se tienen las igualdades:

- $<(\mu X:\tau_1\to\tau_2.M)>^{i+1}\eta$

  $= \mathcal{I}\left\{\frac{<(\mu X:\tau_1\to\tau_2.M)>^i\eta}{X:\tau_1\to\tau_2}\right\}[\![M:\tau_1\to\tau_2]\!]\eta$

  $= \mathcal{I}\left\{\frac{T(\mathcal{I}, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)^i(\perp_{\eta(\tau_1\to\tau_2)})}{X:\tau_1\to\tau_2}\right\}[\![M:\tau_1\to\tau_2]\!]\eta$ por inducción

  $= T(\mathcal{I}, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)\ (T(\mathcal{I}, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)^i(\perp_{\eta(\tau_1\to\tau_2)}))$

  $= T(\mathcal{I}, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)^{i+1}(\perp_{\eta(\tau_1\to\tau_2)})$. ∎

Teorema 3.2.4

Sean $t:\tau \in T_\Delta(\Sigma)$ y $M:\tau_1\to\tau_2 \in Ef(\Sigma)$. Para cualesquiera Σ-interpretación $\mathcal{I}=\langle I, \mathcal{D}, \xi\rangle$, $\eta \in AT_p^I$, y para cada $i < \omega$ se verifica:

a) $\mathcal{I}[\![(t:\tau)^i]\!]\eta \subseteq \mathcal{I}[\![(t:\tau)^{i+1}]\!]\eta$

b) $\mathcal{I}[\![(M:\tau_1\to\tau_2)^i]\!]\eta \subseteq \mathcal{I}[\![(M:\tau_1\to\tau_2)^{i+1}]\!]\eta$

Demostración:

Lo demostramos por inducción mutua en la construcción de los términos y de las expresiones funcionales.

a) Para los casos $x:\tau$, $c:\tau$, $\perp:\tau$, true o false, la demostración es evidente. Para el resto se tiene:

- $\mathcal{I}[\![(t_1:\tau_1, \ldots, t_m:\tau_m)^i]\!]\eta$

$$= \begin{cases} \langle \mathcal{I}[(t_1:\tau_1)^i]\eta, \ldots, \mathcal{I}[(t_n:\tau_n)^i]\eta \rangle & \text{si } \mathcal{I}[(t_j:\tau_j)^i]\eta \neq \perp_{\eta(\tau_j)} (1 \leq j \leq n) \\ \perp_{\eta(\tau_1 x \ldots x \tau_n)} & \text{en otro caso} \end{cases}$$

$$\leq \begin{cases} \langle \mathcal{I}[(t_1:\tau_1)^{i+1}]\eta, \ldots, \mathcal{I}[(t_n:\tau_n)^{i+1}]\eta \rangle \\ \qquad \text{si } \mathcal{I}[(t_j:\tau_j)^{i+1}]\eta \neq \perp_{\eta(\tau_j)} \text{ p.t. } 1 \leq j \leq n \quad \text{por} \\ \perp_{\eta(\tau_1 x \ldots x \tau_n)} \qquad \text{en otro caso} \end{cases}$$

hipót. de induc. y las propiedades del producto estricto, puesto que si $\mathcal{I}[(t_j:\tau_j)^i]\eta \neq \perp_{\eta(\tau_j)}$ y $\mathcal{I}[(t_j:\tau_j)^i]\eta \leq \mathcal{I}[(t_j:\tau_j)^{i+1}]\eta$ para todo $1 \leq j \leq n$, entonces $\mathcal{I}[(t_j:\tau_j)^{i+1}]\eta \neq \perp_{\eta(\tau_j)}$ para todo $1 \leq j \leq n$

$= \mathcal{I}[(t_1:\tau_1, \ldots, t_n:\tau_n)^{i+1}]\eta.$

- $\mathcal{I}[(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^i]\eta =$

$$= \begin{cases} \mathcal{I}[(t_1:\tau)^i]\eta & \text{si } \mathcal{I}[(t:\text{bool})^i]\eta = tt \\ \mathcal{I}[(t_2:\tau)^i]\eta & \text{si } \mathcal{I}[(t:\text{bool})^i]\eta = ff \\ \perp_{\eta(\tau)} & \text{si } \mathcal{I}[(t:\text{bool})^i]\eta = \perp_{\text{bool}} \end{cases}$$

distinguimos casos:

Si $\mathcal{I}[(t:\text{bool})^i]\eta = \perp_{\text{bool}}$ y $\mathcal{I}[(t:\text{bool})^{i+1}]\eta = \perp_{\text{bool}}$ entonces $\mathcal{I}[(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^{i+1}]\eta = \perp_{\eta(\tau)}$
$= \mathcal{I}[(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^i]\eta$

Si $\mathcal{I}[(t:\text{bool})^i]\eta = \perp_{\text{bool}}$ y $\mathcal{I}[(t:\text{bool})^{i+1}]\eta = tt$ o ff entonces $\mathcal{I}[(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^{i+1}]\eta = \mathcal{I}[(t_1:\tau)^{i+1}]\eta$ o $\mathcal{I}[(t_2:\tau)^{i+1}]\eta$, luego $\mathcal{I}[(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^{i+1}]\eta \geq \mathcal{I}[(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^i]\eta = \perp_{\eta(\tau)}$

Si $\mathcal{I}[(t:\text{bool})^i]\eta = tt$ entonces $\mathcal{I}[(t:\text{bool})^{i+1}]\eta = tt$ por hipótesis de inducción, luego, por hipótesis de inducción, $\mathcal{I}[(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^i]\eta = \mathcal{I}[(t_1:\tau)^i]\eta \leq \mathcal{I}[(t_1:\tau)^{i+1}]\eta$
$= \mathcal{I}[(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^{i+1}]\eta$

Si $\mathcal{I}[(t:\text{bool})^i]\eta = ff$ la demostración es análoga al caso anterior, luego la desigualdad se tiene en cualquiera de los casos.

- $\mathcal{I}[((M\ t:\tau_1):\tau_2)^i]\eta = \mathcal{I}[(M:\tau_1 \to \tau_2)^i]\eta\ (\mathcal{I}[(t:\tau_1)^i]\eta)$
$\leq \mathcal{I}[(M:\tau_1 \to \tau_2)^i]\eta\ (\mathcal{I}[(t:\tau_1)^{i+1}]\eta)$
por inducción sobre $t:\tau$ y monotonía de $\mathcal{I}[(M:\tau_1 \to \tau_2)^i]\eta$
$\leq \mathcal{I}[(M:\tau_1 \to \tau_2)^{i+1}]\eta\ (\mathcal{I}[(t:\tau_1)^{i+1}]\eta)$ por inducción sobre $M:\tau_1 \to \tau_2$
$= \mathcal{I}[((M\ t:\tau_1):\tau_2)^{i+1}]\eta.$

b) Los casos $X:\tau_1 \to \tau_2$, $\perp:\tau_1 \to \tau_2$ y $f:\tau_1 \to \tau_2$ son triviales ya que en todos ellos para cualquier $i < \omega$ $(M:\tau_1 \to \tau_2)^i$ coincide con $M:\tau_1 \to \tau_2$.

- $\mathcal{I}[(\lambda x_1:\tau_1 \ldots x_n:\tau_n.\, t:\tau)^i]\eta(d_1, \ldots, d_n)$ con $\langle d_1, \ldots, d_n \rangle$ como siempre

$= \mathcal{I}[(\lambda x_1:\tau_1 \ldots x_n:\tau_n.\,(t:\tau)^i)]\eta(d_1,\ldots,d_n)$

$= \mathcal{I}\left\{\frac{d_1 \ldots d_n}{x_1:\tau_1 \ldots x_n:\tau_n}\right\}[(t:\tau)^i]\eta \leq \mathcal{I}\left\{\frac{d_1 \ldots d_n}{x_1:\tau_1 \ldots x_n:\tau_n}\right\}[(t:\tau)^{i+1}]\eta$

por hipótesis de inducción

$= \mathcal{I}[(\lambda x_1:\tau_1 \ldots x_n:\tau_n.\,t:\tau)^{i+1})]\eta(d_1,\ldots,d_n)$.

- Probamos $\mathcal{I}[(\mu X:\tau_1\to\tau_2.M)^i]\eta \leq \mathcal{I}[(\mu X:\tau_1\to\tau_2.M)^{i+1}]\eta$ por inducción en i

Para i=0, $\mathcal{I}[(\mu X:\tau_1\to\tau_2.M)^0]\eta = \mathcal{I}[\bot:\tau_1\to\tau_2]\eta = \bot_{\eta(\tau_1\to\tau_2)}$ que está menos definida que qualquier función de $[[\tau_1]^{\mathcal{I}}\eta \xrightarrow{}_{\bullet} [\tau_2]^{\mathcal{I}}\eta]$

Supuesto cierto el caso i-ésimo se tiene:

$\mathcal{I}[(\mu X:\tau_1\to\tau_2.M)^{i+1}]\eta$

$= \mathcal{I}[(M:\tau_1\to\tau_2)^{i+1}[(\mu X:\tau_1\to\tau_2.M)^i/X:\tau_1\to\tau_2]]\eta$

$= \mathcal{I}\left\{\frac{\mathcal{I}[(\mu X:\tau_1\to\tau_2.M)^i]\eta)}{X:\tau_1\to\tau_2}\right\}[(M:\tau_1\to\tau_2)^{i+1}]\eta$ por el lema de sustitución

$\leq \mathcal{I}\left\{\frac{\mathcal{I}[(\mu X:\tau_1\to\tau_2.M)^i]\eta)}{X:\tau_1\to\tau_2}\right\}[(M:\tau_1\to\tau_2)^{i+2}]\eta$ por induc. en $(M:\tau_1\to\tau_2)^{i+1}$

$\leq \mathcal{I}\left\{\frac{\mathcal{I}[(\mu X:\tau_1\to\tau_2.M)^{i+1}]\eta)}{X:\tau_1\to\tau_2}\right\}[(M:\tau_1\to\tau_2)^{i+2}]\eta$

por inducción en i y monotonía de la semántica de $(M:\tau_1\to\tau_2)^{i+2}$

$= \mathcal{I}[(M:\tau_1\to\tau_2)^{i+2}[(\mu X:\tau_1\to\tau_2.M)^{i+1}/X:\tau_1\to\tau_2]]\eta$ por lema de sustituc.

$= \mathcal{I}[(\mu X:\tau_1\to\tau_2.M)^{i+2}]\eta$. ■

Teorema 3.2.5

Para cualesquiera $t:\tau$, $\mathcal{I}$, η, $M:\tau_1\to\tau_2$ se verifica:

a) $\mathcal{I}[t:\tau]\eta = \bigsqcup \{\mathcal{I}[(t:\tau)^i]\eta \mid i<\omega\}$

b) $\mathcal{I}[M:\tau_1\to\tau_2]\eta = \bigsqcup \{\mathcal{I}[(M:\tau_1\to\tau_2)^i]\eta \mid i<\omega\}$.

<u>Demostración</u>:

Por inducción simultanea sobre la construcción de los términos y de las expresiones funcionales.

Simplificamos $\bigsqcup \{\mathcal{I}[(t:\tau)^i]\eta \mid i<\omega\}$ por $\bigsqcup_{i=0}^{\infty} \mathcal{I}[(t:\tau)^i]\eta$ y $\bigsqcup \{\mathcal{I}[(M:\tau_1\to\tau_2)^i]\eta \mid i<\omega\}$ por $\bigsqcup_{i=0}^{\infty} \mathcal{I}[(M:\tau_1\to\tau_2)^i]\eta$

a) Si $t:\tau$ es de la forma $x:\tau$, $c:\tau$, $\bot:\tau$, true, false la demostración es evidente. Para los demás casos:

- $\mathcal{I}[(t_1:\tau_1,\ldots,t_n:\tau_n)]\eta$

$= \begin{cases} \langle \mathcal{I}[t_1:\tau_1]\eta,\ldots,\mathcal{I}[t_n:\tau_n]\eta\rangle & \text{si } \mathcal{I}[t_j:\tau_j]\ \eta \neq \bot_{\eta(\tau_j)} \quad \text{para todo } 1\leq j\leq n \\ \bot_{\eta(\tau_1 x \ldots x \tau_n)} & \text{en otro cas o} \end{cases}$

$$= \begin{cases} \langle \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t_1:\tau_1)^i]\!]\eta, \ldots, \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t_n:\tau_n)^i]\!]\eta \rangle \\ \qquad \text{si } \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t_j:\tau_j)^i]\!]\eta \neq \perp_{\eta(\tau_j)} \text{ para todo } 1 \leq j \leq n \\ \perp_{\eta(\tau_1 \times \ldots \times \tau_n)} \qquad \text{en otro caso} \end{cases}$$

por hipótesis de inducción

$= \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t_1:\tau_1, \ldots, t_n:\tau_n)^i]\!]\eta$ por las propiedades del producto $\bullet$.

- $\mathcal{I}[\![(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)]\!]\eta$

$$= \begin{cases} \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t_1:\tau)^i]\!]\eta & \text{si } \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t:\text{bool})^i]\!]\eta = tt \\ \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t_2:\tau)^i]\!]\eta & \text{si } \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t:\text{bool})^i]\!]\eta = ff \text{ por hip. de ind.} \\ \perp_{\eta(\tau)} & \text{si } \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t:\text{bool})^i]\!]\eta = \perp_{\text{bool}} \end{cases}$$

Si $\sqcup_{i=0}^{\omega} \mathcal{I}[\![(t:\text{bool})^i]\!]\eta = \perp_{\text{bool}}$ entonces $\mathcal{I}[\![(t:\text{bool})^i]\!]\eta = \perp_{\text{bool}}$ para todo $i<\omega$ entonces $\mathcal{I}[\![(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^i]\!]\eta = \perp_{\eta(\tau)}$ p. t. $i<\omega$, luego $\sqcup_{i=0}^{\omega} \mathcal{I}[\![(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^i]\!]\eta = \perp_{\eta(\tau)}$.

Si $\sqcup_{i=0}^{\omega} \mathcal{I}[\![(t:\text{bool})^i]\!]\eta = tt$, por ser B un cpo plano, existe un $j<\omega$ tal que para todo $i \geq j$, $\mathcal{I}[\![(t:\text{bool})^i]\!]\eta = tt$, entonces se tiene $\mathcal{I}[\![(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^i]\!]\eta = \mathcal{I}[\![(t_1:\tau)^i]\!]\eta$ para todo $i \geq j$ y $\sqcup_{i=0}^{\omega} \mathcal{I}[\![(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^i]\!]\eta = \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t_1:\tau)^i]\!]\eta$.

De igual forma se demuestra que si $\sqcup_{i=0}^{\omega} \mathcal{I}[\![(t:\text{bool})^i]\!]\eta = ff$ $\sqcup_{i=0}^{\omega} \mathcal{I}[\![(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^i]\!]\eta = \sqcup_{i=0}^{\omega} \mathcal{I}[\![(t_2:\tau)^i]\!]\eta$.

En cualquier caso $\mathcal{I}[\![(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)]\!]\eta$
$= \sqcup_{i=0}^{\omega} \mathcal{I}[\![(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau)^i]\!]\eta$

- $\mathcal{I}[\![(M\ t:\tau_1):\tau_2]\!]\eta = \mathcal{I}[\![M:\tau_1 \to \tau_2]\!]\eta\ (\mathcal{I}[\![t:\tau_1]\!]\eta)$
  $= \mathcal{I}[\![M:\tau_1 \to \tau_2]\!]\eta\ (\sqcup_{j=0}^{\omega} \mathcal{I}[\![(t:\tau_1)^j]\!]\eta)$ por inducción sobre $t:\tau$
  $= \sqcup_{i=0}^{\omega} \mathcal{I}[\![(M:\tau_1 \to \tau_2)^i]\!]\eta\ (\sqcup_{j=0}^{\omega} \mathcal{I}[\![(t:\tau_1)^j]\!]\eta)$ inducción en $M:\tau_1 \to \tau_2$
  $= \sqcup \{\mathcal{I}[\![(M:\tau_1 \to \tau_2)^i]\!]\eta\ (\sqcup_{j=0}^{\omega} \mathcal{I}[\![(t:\tau_1)^j]\!]\eta) \mid i<\omega\}$
  por definición de $\sqcup_{i=0}^{\omega} \{f_i \mid i<\omega\}$
  $= \sqcup \{\sqcup \{\mathcal{I}[\![(M:\tau_1 \to \tau_2)^i]\!]\eta\ (\mathcal{I}[\![(t:\tau_1)^j]\!]\eta) \mid j<\omega\} \mid i<\omega\}$
  puesto que $\mathcal{I}[\![(M:\tau_1 \to \tau_2)^i]\!]\eta$ es continua para todo $i<\omega$
  $= \sqcup \{\mathcal{I}[\![(M:\tau_1 \to \tau_2)^i]\!]\eta\ (\mathcal{I}[\![(t:\tau_1)^i]\!]\eta) \mid i<\omega\}$
  $= \sqcup_{i=0}^{\omega} \mathcal{I}[\![((M\ t:\tau_1):\tau_2)^i]\!]\eta$.

b) Los casos $X:\tau_1 \to \tau_2$, $\perp:\tau_1 \to \tau_2$ y $f:\tau_1 \to \tau_2$ son triviales ya que en todos ellos, $(M:\tau_1 \to \tau_2)^i$ coincide con $M:\tau_1 \to \tau_2$ para cualquier $i<\omega$.

- $\mathcal{I}[\![(\lambda x_1:\tau_1 \ldots x_n:\tau_n.\ t:\tau)]\!]\eta(d_1, \ldots, d_n)$ con $\langle d_1, \ldots, d_n \rangle$ como siempre
  $= \mathcal{I}\left\{\frac{d_1 \ldots d_n}{x_1:\tau_1 \ldots x_n:\tau_n}\right\}[\![t:\tau]\!]\eta$
  $= \sqcup_{i=0}^{\omega} \mathcal{I}\left\{\frac{d_1 \ldots d_n}{x_1:\tau_1 \ldots x_n:\tau_n}\right\}[\![(t:\tau)^i]\!]\eta$ por hipótesis de inducción

$= \sqcup \{\mathcal{I}[\![(\lambda x_1:\tau_1 \ldots x_n:\tau_n.(t:\tau)^i)]\!]\eta\ (d_1,\ldots,d_n) \mid i<\omega\}$

$= \sqcup_{i=0}^{\omega} \mathcal{I}[\![(\lambda x_1:\tau_1 \ldots x_n:\tau_n.t:\tau)^i]\!]\eta\ (d_1,\ldots,d_n)$.

- Probamos $\sqcup_{i=0}^{\omega} \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^i]\!]\eta = \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta$ probando las dos desigualdades y apoyándonos en la propiedad antisimétrica del orden parcial $\subseteq_{\eta(\tau_1\to\tau_2)}$.

  $\subseteq$) Probamos $\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^i]\!]\eta \subseteq \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta$ para todo $i<\omega$ por inducción en i. De esta desigualdad deduciremos que $\sqcup_{i=0}^{\omega} \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^i]\!]\eta \subseteq \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta$.

  Para el caso i=0 es evidente pues $\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^0]\!]\eta = \perp_{\eta(\tau_1\to\tau_2)}$ que está menos definida que cualquier función de $[[\tau_1]^{\mathcal{I}}\eta \xrightarrow[c]{} [\tau_2]^{\mathcal{I}}\eta]$.

  Supuesta cierta la desigualdad para el caso i tendremos:

$\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^{i+1}]\!]\eta = \mathcal{I}[\![(M:\tau_1\to\tau_2)^{i+1}[(\mu X:\tau_1\to\tau_2.M)^i/X:\tau_1\to\tau_2]]\!]\eta$

$= \mathcal{I}\left\{\frac{\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^i]\!]\eta}{X:\tau_1\to\tau_2}\right\}[\![(M:\tau_1\to\tau_2)^{i+1}]\!]\eta$ por el lema de sustitución

$\subseteq \mathcal{I}\left\{\frac{\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^i]\!]\eta}{X:\tau_1\to\tau_2}\right\}[\![M:\tau_1\to\tau_2]\!]\eta$ por inducción sobre $M:\tau_1\to\tau_2$

$\subseteq \mathcal{I}\left\{\frac{\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta}{X:\tau_1\to\tau_2}\right\}[\![M:\tau_1\to\tau_2]\!]\eta$ aplicando hipótesis de inducción sobre i y por la monotonía de la semántica de $M:\tau_1\to\tau_2$

$= \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta$ por la semántica del μ-operador.

  $\supseteq$) Probamos $<(\mu X:\tau_1\to\tau_2.M)>^i\eta \subseteq \sqcup_{j=0}^{\omega} \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^j]\!]\eta$ p.t. $i<\omega$ entonces $\sqcup_{i=0}^{\omega} <(\mu X:\tau_1\to\tau_2.M)>^i\eta \subseteq \sqcup_{j=0}^{\omega} \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^j]\!]\eta$ y por 3.2.3 $\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)]\!]\eta \subseteq \sqcup_{i=0}^{\omega} \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^i]\!]\eta$. Por inducción sobre i:

  El caso i=0 es trivial pues $<(\mu X:\tau_1\to\tau_2.M)>^0\eta = \perp_{\eta(\tau_1\to\tau_2)}$

  Supuesto cierto para i se tiene:

$<(\mu X:\tau_1\to\tau_2.M)>^{i+1}\eta = \mathcal{I}\left\{\frac{<(\mu X:\tau_1\to\tau_2.M)>^i\eta}{X:\tau_1\to\tau_2}\right\}[\![M:\tau_1\to\tau_2]\!]\eta$

$= T(\mathcal{I}, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)\ (<(\mu X:\tau_1\to\tau_2.M)>^i\eta)$

$\subseteq T(\mathcal{I}, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)\ (\sqcup_{j=0}^{\omega} \mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^j]\!]\eta)$ por hipótesis de inducción y por la monotonía del operador T

$= \sqcup_{j=0}^{\omega} T(\mathcal{I}, X:\tau_1\to\tau_2, M:\tau_1\to\tau_2, \eta)\ (\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^j]\!]\eta)$ por continuidad

$= \sqcup_{j=0}^{\omega} \mathcal{I}\left\{\frac{\mathcal{I}[\![(\mu X:\tau_1\to\tau_2.M)^j]\!]\eta}{X:\tau_1\to\tau_2}\right\}[\![M:\tau_1\to\tau_2]\!]\eta$

$= \bigsqcup_{j=0}^{\omega} \bigsqcup_{i=0}^{\omega} \mathfrak{I}\left\{\frac{\mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^j]\eta}{X:\tau_1\to\tau_2}\right\}[(M:\tau_1\to\tau_2)^i]\eta$

por inducción en $M:\tau_1\to\tau_2$

$\subseteq \bigsqcup_{k=0}^{\omega} \mathfrak{I}\left\{\frac{\mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^k]\eta}{X:\tau_1\to\tau_2}\right\}[(M:\tau_1\to\tau_2)^k]\eta$ dados $i, j<\omega$, $k=\max\{i, j\}$

y aplicando la monotonía de la semántica de $(M:\tau_1\to\tau_2)^i$

$\subseteq \bigsqcup_{k=0}^{\omega} \mathfrak{I}\left\{\frac{\mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^k]\eta}{X:\tau_1\to\tau_2}\right\}[(M:\tau_1\to\tau_2)^{k+1}]\eta$ por 3.2.4-b)

$= \bigsqcup_{k=0}^{\omega} \mathfrak{I}[(M:\tau_1\to\tau_2)^{k+1}[(\mu X:\tau_1\to\tau_2.M)^k/X:\tau_1\to\tau_2]]\eta$ lema sustitución

$= \bigsqcup_{k=0}^{\omega} \mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^{k+1}]\eta = \bigsqcup_{j=0}^{\omega} \mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^j]\eta.$ ∎

CAPITULO II

TABLEAUX PARA PLPR

El método de deducción más desarrollado para demostrar teoremas automáticamente es la resolución que tiene su origen en los estudios de Robinson que dieron lugar a multitud de refinamientos y variantes de este método de refutación [Sti-87]. La resolución está basada en el teorema de Herbrand y resulta especialmente adecuada cuando se trabaja con la lógica clásica.

Otro método de refutación de teoremas alternativo a la resolución, aunque menos conocido, es la formulación de los tableaux semánticos de Smullyan, [Smu-68], que tienen su origen en el sistema de tableaux de Beth, [Bet-64]. De hecho, los trabajos de Gilmore para automatizar la demostración de teoremas, que pueden ser considerados como el primer mecanismo que funciona para hacer demostraciones de la lógica de predicados, están basados en los tableaux semánticos de Beth, [Gil-60].

A partir de los tableaux iniciales de primer orden se han desarrollado extensiones naturales para cubrir lógicas no clásicas como por ejemplo lógicas intuicionistas en [McC-88], o lógicas modales en [Wri-85] y [Fit-88].

La automatización de los tableaux resulta al menos igual de práctica que la de la resolución aunque en ocasiones se plantean cuestiones de eficiencia. En [Fit-90] se estudian estos dos métodos de refutación para la lógica de primer orden y se presenta una implementación en Prolog del método de los tableaux, de tal forma que la falta de eficiencia puede considerarse compesada por la facilidad y claridad con la que se escribe y se sigue un programa que simule el comportamiento de los tableaux. Además, la automatización de los tableaux proporciona lugares donde incorporar heurísticas de forma natural; esto puede verse en [Fit-88] donde aparece una implementación de los tableaux para la lógica modal, y en [OS--88], que por medio de estrategias heurísticas, se consigue un sistema muy eficiente basado en una modificación de los tableaux analíticos para la lógica de primer orden.

Otro factor a tener en cuenta a la hora de defender los tableaux es la facilidad con la que se construyen cálculos completos a partir

de ellos. Sirvan como ejemplo los cálculos presentados en [Gav-89] y [Gil-89] obtenidos a partir de métodos de tableaux definidos para lógicas orientadas a la programación funcional.

En este capítulo se define una extensión de los tableaux dados por Smullyan para la lógica de predicados, que sirve de método de refutación para PLPR. Junto con las clases en que Smullyan divide las fórmulas de la lógica de predicados (básicas, conjuntivas, disyuntivas, universales y existenciales), se definen dos nuevas clases propias de la existencia de funciones recursivas. La corrección y completitud del algoritmo presentado convierte a los tableaux en un método para semi-decidir la insatisfactibilidad en PLPR. Para demostrar el teorema de completitud, se definen los conjuntos de Hintikka para nuestra lógica y se prueba su satisfactibilidad. El método constructivo de esta prueba garantizará la validez del teorema de Löwenheim-Skolem para PLPR, que será enunciado en el capítulo siguiente.

Las dos buenas propiedades de los tableaux destacadas en esta introducción, es decir, su fácil automatización y su utilidad para construir cálculos completos, se ponen de manifiesto en este capítulo y serán utilizadas posteriormente. Por un lado, el método que aquí introducimos sirve como base de la implementación en Prolog, que automatiza ciertos pasos de las demostraciones escritas en el sistema presentado en el capítulo IV, y por otro lado, en el capítulo III se construirá un cálculo de deducción natural completo, basado en las condiciones de completitud obtenidas de las propiedades de los tableaux.

1. METODO DE LOS TABLEAUX

Definimos el algoritmo de los tableaux para PLPR como una extensión de los tableaux semánticos de Smullyan. Para ello haremos una partición del conjunto de fórmulas monomórficas en siete clases disjuntas. Las primeras cinco clases son similares a las de los tableaux originales, las otras dos se basan en el concepto de aproximación sintáctica y en la equivalencia entre el límite de las aproximación sintácticas y la semántica del μ-operador probada en §I-3.2.5. La pertenencia de las fórmulas a cada clase se basa en criterios semánticos que permiten descomponer las fórmulas en otras más sencillas que se llamarán constituyentes.

Una vez definido el algoritmo, se prueban ciertas propiedades de los tableaux que serán utilizadas en la demostración de la corrección y completitud de este método. Los ejemplos del final de la sección tratan de reflejar de una manera práctica el comportamiento del mecanismo de deducción definido.

1.1 UNA CLASIFICACION DE LAS FORMULAS MONOMORFICAS ORIENTADA A LOS TABLEAUX

En lo sucesivo dada una signatura $\Sigma = \langle \Sigma_t, \Sigma_d \rangle$, consideraremos los siguientes conjuntos de símbolos:

- El conjunto numerable $CT = \{ct_0/0, ct_1/0, \ldots, ct_n/0, \ldots\}$ de constantes de tipo auxiliares que no aparecen en Σ_t.
- El conjunto numerable $C = \{c_0: \rightarrow \rho, c_1: \rightarrow \rho, \ldots, c_n: \rightarrow \rho, \ldots\}$ de símbolos de constantes tipadas auxiliares que no aparecen en Σ_d.

Si extendemos Σ_t con el conjunto CT y Σ_d con C obtenemos la signatura $\hat{\Sigma} = \langle \Sigma_t \cup CT, \Sigma_d \cup C \rangle$. Hablaremos entonces de $\hat{\Sigma}$-interpretaciones, $\hat{\mathfrak{I}}$, que son extensiones de una Σ-interpretación $\mathfrak{I}$ según el criterio establecido en el capítulo anterior.

Definición 1.1.1

Una *clasificación* del conjunto de $\hat{\Sigma}$-fórmulas monomórficas *orientada hacia los tableaux* es una partición de dicho conjunto en las siete clases disjuntas que se definen a continuación al satisfacer las siguientes condiciones.

1. *BASICA*: A esta clase pertenecen las fórmulas que no pertenecen a ninguna de las otras clases. Estas fórmulas no tienen constituyentes.

2. *ALFA*: Las fórmulas pertenecientes a esta clase son llamadas conjuntivas, se denotan por α y verifican la condición:

   $Sat(\Phi \cup \{\alpha\}) \leftrightarrow Sat(\Phi \cup \{\alpha, \alpha_1, \alpha_2\})$ para ciertas fórmulas α_1 y α_2 de $ML(\hat{\Sigma})$ llamadas constituyentes de α.

3. *BETA*: Esta clase está formada por las fórmulas llamadas diyuntivas que se denotan por β. Estas fórmulas satisfacen:

   $Sat(\Phi \cup \{\beta\}) \leftrightarrow Sat(\Phi \cup \{\beta, \beta_1\})$ o $Sat(\Phi \cup \{\beta, \beta_2\})$ para ciertas β_1 y β_2 llamadas constituyentes de β.

4. *GAMMA*: Pertenecen a esta clase las fórmulas llamadas universales y denotadas por γ. Satisfacen la condición:

   $Sat(\Phi \cup \{\gamma\}) \leftrightarrow Sat(\Phi \cup \{\gamma, \gamma(t:\nu)\})$ para todo $\hat{\Sigma}$-término monomórfico $t:\nu$ de un determinado $\hat{\Sigma}$-tipo ν. Para cada término de esta forma, $\gamma(t:\nu)$ es un constituyente de γ.

5. *DELTA*: Las fórmulas de esta clase también llamadas existenciales se denotan por δ. La condición de pertenencia a DELTA es:

   $Sat(\Phi \cup \{\delta\}) \leftrightarrow Sat(\Phi \cup \{\delta, \delta(c:\nu)\})$ para toda constante auxiliar $c:\rightarrow\rho \in C$ y un determinado $\hat{\Sigma}$-tipo ν de tal manera que $c:\nu$ no aparezca ni en Φ ni en δ. Para cada $c:\nu$ con estas características se tiene un constituyente $\delta(c:\nu)$ de δ.

6. *ω-ALFA*: Las fórmulas de esta clase se llaman ω-conjuntivas y se denotan por α^ω. Verifican la siguiente condición:

   $Sat(\Phi \cup \{\alpha^\omega\}) \leftrightarrow Sat(\Phi \cup \{\alpha^\omega\} \cup \{\alpha_i^\omega \mid i<\omega\})$ siendo las fórmulas α_i^ω $(i<\omega)$ los constituyentes de α^ω.

7. *ω-BETA*: Las fórmulas de esta clase se denominan ω-disyuntivas y se denotan por β^ω, tienen ω constituyentes que se denotan por β_i^ω $(i<\omega)$. La condición de pertenecia a esta clase es:

   $Sat(\Phi \cup \{\beta^\omega\}) \leftrightarrow Sat(\Phi \cup \{\beta^\omega, \beta_i^\omega\})$ para algún $i<\omega$. □

Definición 1.1.2

Para la lógica PLPR se definen las siguientes clases de las fórmulas monomórficas.

BASICA: Son las fórmulas atómicas y sus negaciones siempre que no contengan términos recursivos:

$(p\ t:\nu)$, $\neg(p\ t:\nu)$ siendo $t:\nu$ un término simple en ambos casos.

$t_1:\nu \leq t_2:\nu$, $\neg t_1:\nu \leq t_2:\nu$, en ambos casos $t_1:\nu$ y $t_2:\nu$ son términos simples.

ALFA:

α	α_1	α_2
$\varphi \wedge \psi$	φ	ψ
$\neg(\varphi \vee \psi)$	$\neg\varphi$	$\neg\psi$
$\neg\neg\varphi$	φ	φ

BETA:

β	β_1	β_2
$\varphi \vee \psi$	φ	ψ
$\neg(\varphi \wedge \psi)$	$\neg\varphi$	$\neg\psi$

GAMMA:

γ	$\gamma(t:\nu)$
$\forall x:\nu\ \varphi$	$\varphi[t:\nu/x:\nu]$
$\neg\exists x:\nu\ \varphi$	$\neg\varphi[t:\nu/x:\nu]$

En el primer caso decimos que $\forall$ es el cuantificador de γ. En el segundo, el cuantificador de γ es $\exists$.

DELTA:

δ	$\delta(c:\nu)$
$\exists x:\nu\ \varphi$	$\varphi[c:\nu/x:\nu]$
$\neg\forall x:\nu\ \varphi$	$\neg\varphi[c:\nu/x:\nu]$

En el primer caso decimos que $\exists$ es el cuantificador de δ. En el segundo, el cuantificador de δ es $\forall$.

ω-ALFA:

α^*	α_1^*
$\neg(p\ t:\nu)$ $\quad (t:\nu$ recursivo$)$	$\neg(p\ (t:\nu)^1)$
$t_1:\nu \subseteq t_2:\nu$ $\quad (t_1:\nu$ recursivo, $t_2:\nu$ simple$)$	$(t_1:\nu)^1 \subseteq t_2:\nu$
$\neg t_1:\nu \subseteq t_2:\nu$ $(t_2:\nu$ recursivo$)$	$\neg t_1:\nu \subseteq (t_2:\nu)^1$

ω-BETA:

β^*	β_1^*
$(p\ t:\nu)$ $\quad (t:\nu$ recursivo$)$	$(p\ (t:\nu)^1)$
$t_1:\nu \subseteq t_2:\nu$ $\quad (t_2:\nu$ recursivo$)$	$t_1:\nu \subseteq (t_2:\nu)^1$
$\neg t_1:\nu \subseteq t_2:\nu$ $\quad (t_1:\nu$ recursivo, $t_2:\nu$ simple$)$	$\neg(t_1:\nu)^1 \subseteq t_2:\nu$. $\square$

Para que las categorías que acabamos de definir determinen una clasificación de las $\hat{\Sigma}$-fórmulas monomórficas orientada a los tableaux

tenemos que comprobar que satisfacen las condiciones 1 a 7 de la definición 1.1.1, y que constituyen una partición de $ML(\hat{\Sigma})$.

Lema 1.1.3

Toda $\hat{\Sigma}$-fórmula monomórfica pertenece exactamente a una de las siete categorías anteriores, por tanto, las clases de la definición 1.1.2 dan lugar a una partición de $ML(\hat{\Sigma})$.

Demostración:

Se demuestra estudiando todos los casos posibles de construcciones de fórmulas en PLPR.

i) Si φ es atómica puede ser de la forma:
- $\varphi = t_1:\nu \subseteq t_2:\nu$. Se pueden distinguir los siguientes subcasos:
  - Si $t_1:\nu$ y $t_2:\nu$ son simples entonces φ es básica
  - Si $t_1:\nu$ es recursivo y $t_2:\nu$ simple, φ es α°
  - Si $t_1:\nu$ es recursivo y $t_2:\nu$ recursivo, φ es β°
  - Si $t_1:\nu$ es simple y $t_2:\nu$ recursivo, φ es β°.
- $\varphi = (p\ t:\nu)$. Distinguimos:
  - Si $t:\nu$ es simple entonces φ es básica
  - Si $t:\nu$ es recursivo entonces φ es β°.

ii) Si φ es una negación tenemos los casos:
- $\varphi = \neg t_1:\nu \subseteq t_2:\nu$. Se pueden distinguir los siguientes subcasos:
  - Si $t_1:\nu$ y $t_2:\nu$ son simples entonces φ es básica
  - Si $t_1:\nu$ es recursivo y $t_2:\nu$ simple, φ es β°
  - Si $t_1:\nu$ es recursivo y $t_2:\nu$ recursivo, φ es α°
  - Si $t_1:\nu$ es simple y $t_2:\nu$ recursivo, φ es α°.
- $\varphi = \neg(p\ t:\nu)$. Distinguimos:
  - Si $t:\nu$ es simple entonces φ es básica
  - Si $t:\nu$ es recursivo entonces φ es α°.
- $\varphi = \neg\neg\psi$ entonces φ es α.
- $\varphi = \neg(\psi_1 \vee \psi_2)$, φ es α.
- $\varphi = \neg(\psi_1 \wedge \psi_2)$, φ es β.
- $\varphi = \neg\exists x:\nu\ \psi$, φ es γ.
- $\varphi = \neg\forall x:\nu\ \psi$, φ es δ.

iii) Si φ es una disyunción entonces es β.

iv) Si φ es una conjunción entonces φ es α.

v) Si φ es una cuantificación existencial entonces φ es δ.

vi) Si φ es una cuantificación universal entonces φ es γ. ■

Lema 1.1.4

Sea $\Im$ una $\hat{\Sigma}$-interpretación cualquiera, entonces:

A) Las fórmulas de la partición anterior clasificadas en la clase ALFA verifican $\hat{\Im} \models \alpha$ si y solo si $\hat{\Im} \models \alpha_1 \wedge \alpha_2$.

B) Las fórmulas de la partición anterior clasificadas en la clase BETA verifican $\hat{\Im} \models \beta$ si y solo si $\hat{\Im} \models \beta_1 \vee \beta_2$.

C) Las fórmulas de la partición anterior clasificadas en la clase GAMMA verifican que si $\hat{\Im} \models \gamma$, entonces $\hat{\Im} \models \gamma(t:\nu)$ para todo $\hat{\Sigma}$-término cuyo tipo coincide con el de la variable afectada por el cuantificador de γ.

D) Las fórmulas de la partición anterior clasificadas en la clase DELTA son tales que para una constante auxiliar $c:\nu$ que no aparece en δ y cuyo tipo coincide con el de la variable afectada por el cuantificador de δ, se verifica:

i) $\hat{\Im} \models \delta(c:\nu) \Rightarrow \hat{\Im} \models \delta$

ii) Si $\hat{\Im} \models \delta$ entonces existe un $a \in D_\nu$ tal que $\hat{\Im}(a/c:\nu) \models \delta(c:\nu)$ donde $\hat{\Im}(a/c:\nu)$ es la $\hat{\Sigma}$-interpretación que coincide con $\hat{\Im}$ salvo en $c:\nu$, siendo $\hat{\Im}(a/c:\nu)[c:\nu] = a$.

A°) Las fórmulas de la partición anterior clasificadas en la clase ω-ALFA verifican $\hat{\Im} \models \alpha^\circ$ si y solo si $\hat{\Im} \models \alpha_i{}^\circ$ para todo $i<\omega$.

B°) Las fórmulas de la partición anterior clasificadas en la clase ω-BETA verifican $\hat{\Im} \models \beta^\circ$ si y solo si $\hat{\Im} \models \beta_i{}^\circ$ para algún $i<\omega$.

Demostración:

A) Si $\alpha = \varphi \wedge \psi$ la demostración es inmediata.

Si $\alpha = \neg(\varphi \vee \psi)$, $\hat{\Im} \models \alpha$ es equivalente a que para todo $\eta \in AT_p^{\hat{I}}$ no $\hat{\Im}, \eta \models (\varphi \vee \psi)$. Puesto que se trata de fórmulas monomórficas, esta afirmación es independiente de η, luego equivale a no $\hat{\Im} \models \varphi$ y no $\hat{\Im} \models \psi$ $\Leftrightarrow \hat{\Im} \models \neg\varphi$ y $\hat{\Im} \models \neg\psi$ y por tanto $\hat{\Im} \models \neg\varphi \wedge \neg\psi \Leftrightarrow \hat{\Im} \models \alpha_1 \wedge \alpha_2$.

Si $\alpha = \neg\neg\varphi$, entonces $\hat{\Im} \models \alpha \Leftrightarrow$ para todo $\eta \in AT_p^{\hat{I}}$ no no $\hat{\Im}, \eta \models \varphi \Leftrightarrow$ para todo $\eta \in AT_p^{\hat{I}}$, $\hat{\Im}, \eta \models \varphi \Leftrightarrow$ para todo $\eta \in AT_p^{\hat{I}}$, $\hat{\Im}, \eta \models \varphi$ y $\hat{\Im}, \eta \models \varphi$ $\Leftrightarrow \hat{\Im} \models \varphi \wedge \varphi \Leftrightarrow \hat{\Im} \models \alpha_1 \wedge \alpha_2$.

B) Si $\beta = \varphi \vee \psi$, la demostración es inmediata.

Si $\beta = \neg(\varphi \wedge \psi)$ la demostración es dual al segundo caso de las fórmulas alfa.

Eliminamos a partir de ahora la escritura de η puesto que se trata de fórmulas monomórficas.

C) Si $\gamma = \forall x{:}\nu\ \varphi$ y $\hat{\mathfrak{I}} \vdash \gamma$ entonces, para todo $a \in D_\nu$ $\hat{\mathfrak{I}}\{a/x{:}\nu\} \vdash \varphi$, y como $\hat{\mathfrak{I}}[t{:}\nu] \in D_\nu$ para todo $t{:}\nu$ monomórfico, se tiene $\hat{\mathfrak{I}}\{\hat{\mathfrak{I}}[t{:}\nu]/x{:}\nu\} \vdash \varphi$ que por el lema de sustitución implica $\hat{\mathfrak{I}} \vdash \varphi[t{:}\nu/x{:}\nu]$.

Si $\gamma = \neg\exists x{:}\nu\ \varphi$, $\hat{\mathfrak{I}} \vdash \neg\exists x{:}\nu\ \varphi$ implica que no existe un $a \in D_\nu$ tal que $\hat{\mathfrak{I}}\{a/x{:}\nu\} \vdash \varphi$ y por lo tanto para todo término $t{:}\nu$, no $\hat{\mathfrak{I}}\{\hat{\mathfrak{I}}[t{:}\nu]/x{:}\nu\} \vdash \varphi$ luego, para todo $t{:}\nu$ $\hat{\mathfrak{I}}\{\hat{\mathfrak{I}}[t{:}\nu]/x{:}\nu\} \vdash \neg\varphi$ y por el lema de sustitución, $\hat{\mathfrak{I}} \vdash \gamma(t{:}\nu)$ para todo $\hat{\Sigma}$-término de tipo ν.

D) i) Si $\delta = \exists x{:}\nu\ \varphi$ la demostración es una consecuencia inmediata de la semántica de las fórmulas existenciales monomórficas y del lema de sustitución.

Si $\delta = \neg\forall x{:}\nu\ \varphi$, $\hat{\mathfrak{I}} \vdash \delta(c{:}\nu)$ es lo mismo que no $\hat{\mathfrak{I}} \vdash \varphi[c{:}\nu/x{:}\nu]$. Supongamos que para todo $a \in D_\nu$ $\hat{\mathfrak{I}}\{a/x{:}\nu\} \vdash \varphi$, entonces para $a = \hat{\mathfrak{I}}[c{:}\nu]$ se tiene $\hat{\mathfrak{I}}\{\hat{\mathfrak{I}}[c{:}\nu]/x{:}\nu\} \vdash \varphi$ y por el lema de sustitución $\hat{\mathfrak{I}} \vdash \varphi[c{:}\nu/x{:}\nu]$ que es absurdo por hipótesis, luego no para todo $a \in D_\nu$ $\hat{\mathfrak{I}}\{a/x{:}\nu\} \vdash \varphi$ o, lo que es lo mismo, $\hat{\mathfrak{I}} \vdash \neg\forall x{:}\nu\ \varphi$.

ii) Si $\delta = \exists x{:}\nu\ \varphi$ y $\hat{\mathfrak{I}} \vdash \delta$, de la semántica de las fórmulas cuantificadas existencialmente deducimos $\hat{\mathfrak{I}}\{a/x{:}\nu\} \vdash \varphi$ para algún $a \in D_\nu$. Puesto que $c{:}\nu$ no aparece en φ, el lema de coincidencia nos asegura que $\hat{\mathfrak{I}}\{a/c{:}\nu\}\{a/x{:}\nu\} \vdash \varphi$, pero esto es equivalente a decir $\hat{\mathfrak{I}}\{a/c{:}\nu\}\{\hat{\mathfrak{I}}\{a/c{:}\nu\}[c{:}\nu]/x{:}\nu\} \vdash \varphi$ ya que $\hat{\mathfrak{I}}\{a/c{:}\nu\}[c{:}\nu] = a$. Por tanto, aplicando el lema de sustitución llegamos al resultado esperado, es decir, $\hat{\mathfrak{I}}\{a/c{:}\nu\} \vdash \delta(c{:}\nu)$ para algún $a \in D_\nu$.

Si $\delta = \neg\forall x{:}\nu\ \varphi$, la idea de la demostración es análoga que para el caso anterior.

A*) En la demostración de esta propiedad nos basamos en los teoremas 3.2.4 y 3.2.5 del capítulo anterior y en el hecho de que toda cadena creciente en un cpo plano tiene, a lo sumo, un solo elemento distinto de $\perp$.

Si $\alpha^* = \neg(p\ t{:}\nu)$ con $t{:}\nu$ recursivo, $p{:}\tau \in \Sigma_d$ y $\nu = \tau\sigma$ tenemos:

$$\begin{aligned}
\hat{\mathfrak{I}} \vdash \alpha^* &\iff \hat{\mathfrak{I}}[t{:}\nu] \notin p{:}\nu^{\hat{\mathfrak{D}}} \iff \sqcup_{i=0}^{\omega} \hat{\mathfrak{I}}[(t{:}\nu)^i] \notin p{:}\nu^{\hat{\mathfrak{D}}} \\
&\iff \text{para todo } i<\omega,\ \hat{\mathfrak{I}}[(t{:}\nu)^i] \notin p{:}\nu^{\hat{\mathfrak{D}}} \text{ pues } \perp_\nu \notin p{:}\nu^{\hat{\mathfrak{D}}} \\
&\iff \hat{\mathfrak{I}} \vdash \neg(p\ (t{:}\nu)^i) \text{ para todo } i<\omega.
\end{aligned}$$

Si $\alpha^* = t_1{:}\nu \leq t_2{:}\nu$, con $t_1{:}\nu$ recursivo y $t_2{:}\nu$ simple, entonces:

$$\begin{aligned}
\hat{\mathfrak{I}} \vdash \alpha^* &\iff \sqcup_{i=0}^{\omega} \hat{\mathfrak{I}}[(t_1{:}\nu)^i] \leq \hat{\mathfrak{I}}[t_2{:}\nu] \\
&\iff \hat{\mathfrak{I}}[(t_1{:}\nu)^i] \leq \hat{\mathfrak{I}}[t_2{:}\nu] \text{ para todo } i<\omega \\
&\iff \hat{\mathfrak{I}} \vdash (t_1{:}\nu)^i \leq t_2{:}\nu \text{ para todo } i<\omega
\end{aligned}$$

$\leftrightarrow$ para todo $i<\omega$ $\hat{\mathfrak{I}} \vdash \alpha_i^\bullet$.

Si $\alpha^\bullet = \neg t_1:\nu \leq t_2:\nu$ con $t_2:\nu$ recursivo, se tiene:

$\hat{\mathfrak{I}} \vdash \alpha^\bullet \leftrightarrow$ no $\hat{\mathfrak{I}}[t_1:\nu] \leq \sqcup_{i=0}^{\omega} \hat{\mathfrak{I}}[(t_2:\nu)^i]$

$\leftrightarrow$ para todo $i<\omega$ no $\hat{\mathfrak{I}}[t_1:\nu] \leq \hat{\mathfrak{I}}[(t_2:\nu)^i]$

$\leftrightarrow \hat{\mathfrak{I}} \vdash \neg t_1:\nu \leq (t_2:\nu)^i$ para todo $i<\omega$

$\leftrightarrow \hat{\mathfrak{I}} \vdash \alpha_i^\bullet$ para todo $i<\omega$.

B°) Nos basaremos en las mismas propiedades que en la demostración del apartado A°).

Si $\beta^\bullet = (p\ t:\nu)$ con $t:\nu$ recursivo, $p:\tau \in \Sigma_d$ y $\nu = \tau\sigma$, entonces:

$\hat{\mathfrak{I}} \vdash \beta^\bullet \leftrightarrow \sqcup_{i=0}^{\omega} \hat{\mathfrak{I}}[(t:\nu)^i] \in p:\nu^{\hat{\mathfrak{D}}}$

$\leftrightarrow \hat{\mathfrak{I}}[(t:\nu)^i] \in p:\nu^{\hat{\mathfrak{D}}}$ para algún $i<\omega$

$\leftrightarrow \hat{\mathfrak{I}} \vdash (p\ (t:\nu)^i)$ para algún $i<\omega$

$\leftrightarrow \hat{\mathfrak{I}} \vdash \beta_i^\bullet$ para algún $i<\omega$.

Si $\beta^\bullet = t_1:\nu \leq t_2:\nu$ con $t_2:\nu$ un $\hat{\Sigma}$-término recursivo, entonces:

$\hat{\mathfrak{I}} \vdash \beta^\bullet \leftrightarrow \hat{\mathfrak{I}}[t_1:\nu] \leq \sqcup_{i=0}^{\omega} \hat{\mathfrak{I}}[(t_2:\nu)^i]$

$\leftrightarrow \hat{\mathfrak{I}}[t_1:\nu] \leq \hat{\mathfrak{I}}[(t_2:\nu)^i]$ para algún $i<\omega$

$\leftrightarrow \hat{\mathfrak{I}} \vdash t_1:\nu \leq (t_2:\nu)^i$ para algún $i<\omega$

$\leftrightarrow \hat{\mathfrak{I}} \vdash \beta_i^\bullet$ para algún $i<\omega$.

Si $\beta^\bullet = \neg t_1:\nu \leq t_2:\nu$ con $t_1:\nu$ recursivo y $t_2:\nu$ simple,

$\hat{\mathfrak{I}} \vdash \beta^\bullet \leftrightarrow$ no $\sqcup_{i=0}^{\omega} \hat{\mathfrak{I}}[(t_1:\nu)^i] \leq \hat{\mathfrak{I}}[t_2:\nu]$

$\leftrightarrow$ no $\hat{\mathfrak{I}}[(t_1:\nu)^i] \leq \hat{\mathfrak{I}}[t_2:\nu]$ para algún $i<\omega$

$\leftrightarrow \hat{\mathfrak{I}} \vdash \neg(t_1:\nu)^i \leq t_2:\nu$ para algún $i<\omega$

$\leftrightarrow \hat{\mathfrak{I}} \vdash \beta_i^\bullet$ para algún $i<\omega$. ■

Proposición 1.1.5

La clasificación de las fórmulas monomórficas de PLPR dada en 1.1.2 es una clasificación orientada a los tableaux, en el sentido de la definición 1.1.1.

Demostración:

Este resultado puede considerarse como un corolario de los dos lemas anteriores. En el primero se prueba que las clases a las que se refiere esta proposición forman una partición de $ML(\hat{\Sigma})$. En el segundo se pone de manifiesto que estas siete clases verifican condiciones de satisfactibilidad incluso más fuertes que las establecidas en la definición 1.1.1. ■

1.2 EL ALGORITMO DE LOS TABLEAUX

Vamos a trabajar con árboles ω-ramificados en el sentido de que estos árboles pueden tener nodos con una cantidad numerable de hijos; además, los nodos de estos árboles están etiquetados por conjuntos de fórmulas monomórficas. Los tableaux serán árboles de este tipo que cumplen unas condiciones particulares. Denotaremos los árboles o tableaux por la letra $\mathcal{T}$.

Definición 1.2.1

Sea $\Phi \subseteq L(\Sigma)$ se dice que Φ es *coherente* cuando no existe ninguna fórmula φ tal que $\varphi \in \Phi$ y $\neg\varphi \in \Phi$, y ninguna fórmula de Φ es de la forma $(p\ \bot{:}\tau)$.

Una rama de un árbol se dice que está *abierta* si la unión de los conjuntos que etiquetan los nodos de la rama es coherente. En caso contrario la rama está *cerrada*.

Dado un árbol $\mathcal{T}$ se define la *profundidad* de $\mathcal{T}$ y se denota $p(\mathcal{T})$ por inducción sobre su construcción de la siguiente forma:

$p(\mathcal{T}) = 0$ si $\mathcal{T}$ tiene un solo nodo.

Si $\mathcal{T}_i$, $i \in I \subseteq \mathbb{N}$, son los subárboles de $\mathcal{T}$, entonces $p(\mathcal{T})$ es el supremo de los ordinales $p(\mathcal{T}_i) + 1$, $i \in I$.

Una rama de un árbol es *finita* si contiene un número finito de nodos. Un árbol $\mathcal{T}$ es *finito* si $p(\mathcal{T})$ es un ordinal finito. □

Definición 1.2.2

a) Una $\hat{\Sigma}$-sustitución de tipos σ es *adecuada* a una fórmula φ y a un conjunto de fórmulas Ψ si y solo si todos sus $\hat{\Sigma}$-símbolos de tipo aparecen en Ψ y $\varphi\sigma$ es monomórfica.

b) Un $\hat{\Sigma}$-término monomófico $t{:}\nu$ es *adecuado* a una fórmula γ y a un conjunto de fórmulas Ψ si y solo si ν es el tipo de la variable afectada por el cuantificador de γ y $t{:}\nu$ está construido utilizando $\hat{\Sigma}$-símbolos de constante o función, $\bot$, true, false o variables que aparezcan libres en las fórmulas de Ψ.

c) Los *axiomas de orden* se denotan por θ y son los que aparecen en la figura 1.

Para cada axioma de orden θ, si θ' es una $\hat{\Sigma}$-instancia de θ se dice que θ' es *adecuada* a un conjunto de fórmulas Ψ, si cada $\hat{\Sigma}$-símbolo de θ' también aparece en Ψ. □

$\forall x{:}\tau\ x \subseteq x$

$\forall x{:}\tau\ \forall y{:}\tau\ \forall z{:}\tau\ ((x \subseteq y \wedge y \subseteq z) \rightarrow x \subseteq z)$

$\forall x{:}\tau\ \forall y{:}\tau\ ((x \subseteq y \wedge y \subseteq x) \rightarrow x = y)$

$\forall x_1{:}\tau_1 .. \forall x_n{:}\tau_n\ ((x_1 \subseteq \bot{:}\tau_1 \vee .. \vee x_n \subseteq \bot{:}\tau_n) \leftrightarrow (x_1,..,x_n) \subseteq \bot{:}\tau_1 \times .. \times \tau_n$

$\forall x_1{:}\tau_1 .. \forall x_n{:}\tau_n\ \forall y_1{:}\tau_1 ... \forall y_n{:}\tau_n\ ((\neg x_1 \subseteq \bot{:}\tau_1 \wedge ... \wedge \neg x_n \subseteq \bot{:}\tau_n) \rightarrow$
$((x_1,...,x_n) \subseteq (y_1,...,y_n) \leftrightarrow (x_1 \subseteq y_1 \wedge .. \wedge x_n \subseteq y_n)))$

$\forall x{:}\tau\ \forall y{:}\tau\ (x \subseteq y \rightarrow (x = y \vee x = \bot))$

$\forall x{:}\tau\ \bot \subseteq x$

$\forall x{:}\tau_1\ (\bot\ x){:}\tau_2 = \bot{:}\tau_2$

$(M\ \bot{:}\tau_1) = \bot{:}\tau_2$ para $M{:}\tau_1 \rightarrow \tau_2 \in Ef(\hat{\Sigma})$

$\forall x{:}\mathrm{bool}\ (x = \mathrm{true} \vee x = \mathrm{false} \vee x = \bot{:}\mathrm{bool})$

$\forall x{:}\tau_1\ \forall y{:}\tau_1 (x = y \rightarrow (M\ x{:}\tau_1) = (M\ y{:}\tau_1))$ con $M{:}\tau_1 \rightarrow \tau_2 \in Ef(\hat{\Sigma})$

$\forall x{:}\tau\ \forall y{:}\tau\ ((x = y \wedge (p\ x{:}\tau)) \rightarrow (p\ y{:}\tau))$ para $p{:}\tau \in \Sigma_d$

$\forall x{:}\tau\ \forall y{:}\tau\ \forall z{:}\mathrm{bool}\ (z = \bot{:}\mathrm{bool} \rightarrow (\mathrm{if}\ z\ \mathrm{then}\ x\ \mathrm{else}\ y) = \bot{:}\tau)$

$\forall x{:}\tau\ \forall y{:}\tau\ \forall z{:}\mathrm{bool}\ (z = \mathrm{true} \rightarrow (\mathrm{if}\ z\ \mathrm{then}\ x\ \mathrm{else}\ y) = x)$

$\forall x{:}\tau\ \forall y{:}\tau\ \forall z{:}\mathrm{bool}\ (z = \mathrm{false} \rightarrow (\mathrm{if}\ z\ \mathrm{then}\ x\ \mathrm{else}\ y) = y)$

$\forall y_1{:}\tau_1 ... \forall y_n{:}\tau_n\ (\neg(y_1,...,y_n) \subseteq \bot{:}\tau_1 \times ... \times \tau_n \rightarrow$
$((\lambda x_1{:}\tau_1 ... x_n{:}\tau_n.\, t{:}\tau)(y_1,..,y_n)) = t{:}\tau[y_1 ... y_n / x_1 ... x_n])$

$\forall x{:}\tau_1\ (\neg((\mu X{:}\tau_1 \rightarrow \tau_2.M)^1\ x) \subseteq \bot{:}\tau_2 \rightarrow$
$((\mu X{:}\tau_1 \rightarrow \tau_2.M)^1\ x) = ((\mu X{:}\tau_1 \rightarrow \tau_2.M)\ x)$ $M{:}\tau_1 \rightarrow \tau_2 \in Ef(\hat{\Sigma}),\ 1<\omega$

Figura 1: Axiomas de orden

Lema 1.2.3

Todo axioma de orden es una fórmula válida para PLPR.

Demostración:

La validez de estos axiomas es una consecuencia de la semántica definida para PLPR. Por ejemplo, los tres primeros expresan las propiedades reflexiva, transitiva y antisimétrica de todo orden parcial. Puesto que los dominios de interpretación son órdenes parciales, toda interpretación será modelo de dichas fórmulas. Los dos axiomas siguientes reflejan propiedades del producto estricto que es con el que trabajamos. El siguiente es la expresión sintáctica del hecho de que los órdenes son planos. La semántica del término $\bot{:}\tau$ y de la expresión funcional $\bot{:}\tau_1 \rightarrow \tau_2$ hace que se satisfagan los dos axiomas siguientes, y puesto que las expresiones funcionales se interpretan como funciones estrictas, se verifica el axioma $(M\ \bot{:}\tau_1) = \bot{:}\tau_2$. La

semántica del tipo bool y de los términos true y false sirve como justificación del siguiente axioma. A continuación aparecen unos axiomas que podríamos llamar de sustitución los dos primeros reflejan las propiedades comunes de sustitución de términos iguales en funciones y en fórmulas predicativas, para indicar la congruencia de las funciones y de los predicados, respectivamente. Los tres axiomas siguientes son consecuencia de la semántica del condicional.

Vamos a probar con detalle la validez del axioma de orden propio del lambda cálculo:

$$\forall y_1{:}\tau_1 .. \forall y_n{:}\tau_n (\neg(y_1,\ldots,y_n) \leq \bot{:}\tau_1 \times \ldots \times \tau_n \longrightarrow ((\lambda x_1{:}\tau_1 \ldots x_n{:}\tau_n.\, t{:}\tau)(y_1,\ldots,y_n)) = t{:}\tau[y_1 \ldots y_n / x_1 \ldots x_n])$$

para ello tendremos que probar que para toda $\mathfrak{I}$, para toda η y para toda n-tupla $\bar{d} = \langle d_1,\ldots,d_n\rangle \in [\![\tau_1]\!]^{\mathfrak{I}}\eta \bullet \ldots \bullet [\![\tau_1]\!]^{\mathfrak{I}}\eta \setminus \{\bot_\bullet\}$ obtenemos la igualdad:

$$\mathfrak{I}'[\![(\lambda x_1{:}\tau_1 \ldots x_n{:}\tau_n.\, t{:}\tau)]\!]\eta\ (\mathfrak{I}'[\![(y_1{:}\tau_1 \ldots y_n{:}\tau_n)]\!]\eta) = \mathfrak{I}'[\![t{:}\tau[y_1 \ldots y_n/x_1 \ldots x_n]]\!]\eta \qquad \text{donde } \mathfrak{I}' = \mathfrak{I}\left\{\frac{d_1 \ldots d_n}{y_1{:}\tau_1 \ldots y_n{:}\tau_n}\right\}$$

Esto es cierto porque por un lado

$\mathfrak{I}'[\![(y_1{:}\tau_1 \ldots y_n{:}\tau_n)]\!]\eta = \bar{d}$ y por otro,

$$\mathfrak{I}'[\![(\lambda x_1{:}\tau_1 .. x_n{:}\tau_n.\, t{:}\tau)]\!]\eta\ (\bar{d}) = \mathfrak{I}'\left\{\frac{d_1 \ldots d_n}{x_1{:}\tau_1 \ldots x_n{:}\tau_n}\right\}[\![t{:}\tau]\!]\eta$$

$$= \mathfrak{I}'\left\{\frac{\mathfrak{I}'[\![y_1{:}\tau_1]\!]\eta \ldots \mathfrak{I}'[\![y_n{:}\tau_n]\!]\eta}{x_1{:}\tau_1 \ldots x_n{:}\tau_n}\right\}[\![t{:}\tau]\!]\eta$$

puesto que $\mathfrak{I}'[\![y_i{:}\tau_i]\!]\eta = d_i$ para todo i, $1 \leq i \leq n$

$= \mathfrak{I}'[\![t{:}\tau[y_1 .. y_n/x_1 .. x_n]]\!]\eta$ por el lema de sustitución.

La validez de $\forall x{:}\tau_1(\neg((\mu X{:}\tau_1 \to \tau_2.\, M)^i\ x) \leq \bot{:}\tau_2 \longrightarrow ((\mu X{:}\tau_1 \to \tau_2.\, M)^i\ x) = ((\mu X{:}\tau_1 \to \tau_2.\, M)\ x)\ \ (i<\omega)$ es consecuencia de las propiedades de los cpo's planos, y de los resultados relativos a la equivalencia semántica del μ-operador y el supremo de sus aproximaciones sintácticas (Cfr. §I-3.2.5). ■

En lo que sigue, dado un conjunto de Σ-fórmulas Φ, utilizamos la siguiente notación:

$M\hat{\Sigma}(\Phi) := \{\varphi\sigma \mid \sigma$ es una $\hat{\Sigma}$-sustit. de tipos, $\varphi \in \Phi$, $\varphi\sigma$ es monomórfica$\}$ para representar el conjunto de las $\hat{\Sigma}$-instancias monomórficas de las fórmulas de Φ.

Definición 1.2.4

Dado un árbol $\mathcal{T}$ cuyos nodos están etiquetados por conjuntos de $\hat{\Sigma}$-fórmulas y un conjunto $\Phi \subseteq L(\Sigma)$, si R es una rama de $\mathcal{T}$ y Ψ_R es el conjunto de fórmulas que etiqueta su hoja, las *reglas de expansión de* Ψ_R *con respecto a* Φ se dividen en extensiones y bifurcaciones y son las siguientes:

Extensión de Ψ_R con respecto a Φ:

α-extensión: Si $\alpha \in \Psi_R$, alargar R con un nodo etiquetado por el conjunto $\Psi_R \cup \{\alpha_1, \alpha_2\}$.

α°-extensión: Si $\alpha^\circ \in \Psi_R$, alargar R con un nodo etiquetado por el conjunto $\Psi_R \cup \{\alpha_i^\circ \mid i < \omega\}$.

γ-extensión: Si $\gamma \in \Psi_R$, alargar R con un nodo etiquetado por el conjunto $\Psi_R \cup \{\gamma(t:\nu)\}$, siendo $t:\nu$ un término adecuado a γ y a Ψ_R.

δ-extensión: Si $\delta \in \Psi_R$ alargar R con un nodo etiquetado por el conjunto $\Psi_R \cup \{\delta(c:\nu)\}$, siendo c un símbolo de constante auxiliar que no aparece en Ψ_R ni en δ, y ν es el tipo de la variable afectada por el cuantificador de δ.

θ-extensión: Alargar R con un nodo etiquetado por $\Psi_R \cup \{\theta'\}$ siendo θ un axioma de orden y θ' una $\hat{\Sigma}$-instancia monomórfica de θ adecuada a Ψ_R.

φ-extensión: Si $\varphi \in \Phi$, alargar R con un nodo etiquetado por el conjunto $\Psi_R \cup \{\varphi\sigma\}$ siendo σ una $\hat{\Sigma}$-sustitución de tipos adecuada a φ y a Ψ_R.

Bifurcación de Ψ_R:

β-bifurcación: si $\beta \in \Psi_R$, bifurcar R con dos nuevos nodos etiquetados por $\Psi_R \cup \{\beta_1\}$, $\Psi_R \cup \{\beta_2\}$ respectivamente .

β°-bifurcación: si $\beta^\circ \in \Psi_R$, bifurcar R con ω nuevos nodos etiquetados por $\Psi_R \cup \{\beta_i^\circ\}$, $i < \omega$, respectivamente.

Dados dos árboles ω-ramificados $\mathcal{T}$ y $\mathcal{T}'$ de ramas finitas cuyos nodos están etiquetados por conjuntos de fórmulas, decimos que $\mathcal{T}'$ es una *expansión* de $\mathcal{T}$ con respecto a un conjunto de fórmulas no vacio Φ, si para algún conjunto R de ramas abiertas de $\mathcal{T}$, $\mathcal{T}'$ se ha obtenido a partir de $\mathcal{T}$ expandiendo de forma simultanea los conjuntos que etiquetan las hojas de las ramas de R de acuerdo con las reglas de expansión con respecto a Φ. □

Definición 1.2.5

Un árbol $\mathcal{T}$ se dice que es un *tableau para un conjunto de*

Σ-*fórmulas* Φ si es el límite de una sucesión de árboles ω-ramificados $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_n \ldots$ donde $\mathcal{T}_0$ está formado por un solo nodo etiquetado por un subconjunto $\Phi_0 \subseteq M\hat{\Sigma}(\Phi)$, y para cada $i<\omega$, $\mathcal{T}_{i+1}$ es una expansión de $\mathcal{T}_i$ con respecto a Φ. □

Definición 1.2.6

Un conjunto H de $\hat{\Sigma}$-fórmulas monomórficas es un *conjunto de Hintikka* si satisface:

(a) Si $\alpha \in H$ entonces $\alpha_1 \in H$ y $\alpha_2 \in H$.

(a°) Si $\alpha^\circ \in H$ entonces $\alpha_i{}^\circ \in H$ para todo $i < \omega$.

(b) Si $\beta \in H$ entonces $\beta_1 \in H$ ó $\beta_2 \in H$.

(b°) Si $\beta^\circ \in H$ entonces $\beta_i{}^\circ \in H$ para algún $i < \omega$.

(c) Si $\gamma \in H$ entonces $\gamma(t:\nu) \in H$ para cada $\hat{\Sigma}$-término $t:\nu$ adecuado a γ y a H.

(d) Si $\delta \in H$ entonces $\delta(c:\nu) \in H$ para alguna constante auxiliar c con tipo ν, donde ν es el tipo de la variable afectada por el cuantificador de δ.

(e) Todas las $\hat{\Sigma}$-instancias monomórficas de los axiomas de orden adecuadas a H están en H.

(f) H es coherente. □

Definición 1.2.7

Una *rama* de un tableau $\mathcal{T}$ para un conjunto de fórmulas $\Phi \subseteq L(\Sigma)$ es *completa* cuando la unión de los conjuntos de fórmulas que etiquetan sus nodos es un conjunto de Hintikka y contiene al conjunto $M\hat{\Sigma}(\Phi)$.

Un *tableau* $\mathcal{T}$ para $\Phi \subseteq L(\Sigma)$ es *cerrado* cuando todas sus ramas son finitas y cerradas, en caso contrario es *abierto*.

Un *tableau* $\mathcal{T}$ para $\Phi \subseteq L(\Sigma)$ es *completo* cuando todas sus ramas abiertas son completas. □

1.3 RESULTADOS PRACTICOS Y EJEMPLOS

De las definiciones dadas en el apartado 1.2 pueden deducirse una serie de propiedades de los tableaux que serán de gran utilidad en las demostraciones de la corrección y completitud de este método. A continuación probaremos que todo tableau es un árbol con una forma determinada y más adelante veremos cómo cualquier conjunto de fórmulas

(posiblemente polimórficas) tiene un tableau completo.

Lema 1.3.1

Sea $\Phi \subseteq L(\Sigma)$ y $\mathcal{T}$ un tableau para Φ con todas sus ramas finitas y con el conjunto Ψ etiquetando su raíz. Entonces $\mathcal{T}$ es de una de las formas (A) o (B) siguientes:

(A) $\mathcal{T}$ tiene un único nodo etiquetado por $\Phi_0 \subseteq \hat{M\Sigma}(\Phi)$.

(B) $\mathcal{T}$ tiene uno más subárboles de una de las tres formas que siguen:

i) $\mathcal{T}$ tiene exactamente un subárbol $\mathcal{T}_1$ con etiqueta Φ_1 en su raíz de forma que $\mathcal{T}_1$ es un tableau para $\Phi \cup \Phi_1$, además en la construcción de $\mathcal{T}$ se hizo una extensión de Ψ con respecto a Φ.

ii) $\mathcal{T}$ tiene exactamente dos subárboles $\mathcal{T}_1$ y $\mathcal{T}_2$ con etiquetas Φ_1 y Φ_2 en su raíz, respectivamente, tales que $\mathcal{T}_1$ es un tableau para $\Phi \cup \Phi_1$ y $\mathcal{T}_2$ es un tableau para $\Phi \cup \Phi_2$. En la construcción de $\mathcal{T}$ se hizo una β-bifurcación de Ψ.

iii) $\mathcal{T}$ tiene ω subárboles $\mathcal{T}_i$ con etiqueta Φ_i en su raíz ($i<\omega$) de forma que para cada $i<\omega$, $\mathcal{T}_i$ es un tableau para $\Phi \cup \Phi_i$, y en la construcción de $\mathcal{T}$ se hizo una β^*-bifurcación de Ψ.

Demostración:

Vamos a probar primero que todo subárbol $\mathcal{T}_1$ de $\mathcal{T}$ con el conjunto de fórmulas Φ_1 etiquetando su raíz es un tableau para $\Phi \cup \Phi_1$.

Por la definición de tableau, $\mathcal{T}$ es el límite de una sucesión de árboles $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_k, \ldots$ tales que $\mathcal{T}_0$ consiste en un único nodo etiquetado por $\Phi_0 \subseteq \hat{M\Sigma}(\Phi)$ y para todo k, $\mathcal{T}_{k+1}$ es una expansión de $\mathcal{T}_k$ con respecto a Φ. Entonces, cualquier subárbol $\mathcal{T}_1$ de $\mathcal{T}$ con Φ_1 etiquetando su raíz es el límite de una sucesión $\mathcal{T}^1_0, \mathcal{T}^1_1, \ldots, \mathcal{T}^1_k, \ldots$ donde por un lado $\mathcal{T}^1_0$ es un árbol con un solo nodo etiquetado por el subconjunto $\Phi_0 \cup \Phi_1 \subseteq \hat{M\Sigma}(\Phi \cup \Phi_1)$ (puesto que $\Phi_0 \subseteq \hat{M\Sigma}(\Phi)$ y las fórmulas de Φ_1 son monomórficas) y por otro lado $\mathcal{T}^1_k$ se obtiene de $\mathcal{T}^1_{k+1}$ aplicando la regla de expansión que hizo pasar de $\mathcal{T}_k$ a $\mathcal{T}_{k+1}$. Por tanto, y puesto que toda expansión con respecto a un conjunto lo es también con respecto a un conjunto mayor, $\mathcal{T}^1_{k+1}$ es una expansión de $\mathcal{T}^1_k$ con respecto a $\Phi \cup \Phi_1$ con lo que deducimos que $\mathcal{T}_1$ es un tableau para $\Phi \cup \Phi_1$.

Ahora bien, si $\mathcal{T}$ tiene un único nodo, $\mathcal{T}$ coincidirá con $\mathcal{T}_0$ que por hipótesis consite en un nodo etiquetado por $\Phi_0 \subseteq \hat{M\Sigma}(\Phi)$ y tenemos un árbol de la forma (A). En caso contrario, Ψ habrá sido extendido con respecto a Φ o bifurcado. Supongamos que Ψ fue extendido con respecto

a Φ, entonces por lo que acabamos de probar estamos en el caso (B)-i) del lema. Si Ψ fue β-bifurcado o β°-bifurcado, como $\mathcal{T}_1$ es un tableau para $\Phi \cup \Phi_1$, estamos en los casos (B)-ii) y (B)-iii) del lema, respectivamente. ■

En lo que sigue dada una rama R nos referiremos a Ψ_R para denotar el conjunto de fórmulas que etiquetan su hoja.

Lema 1.3.2

Todo conjunto de Σ-fórmulas Φ tiene un tableau completo $\mathcal{T}_\Phi$ que llamamos *tableau canónico* de Φ.

Demostración:

Consideramos una enumeración de la unión de los siguientes conjuntos de $\hat{\Sigma}$-fórmulas:

(i) $\{\alpha \mid \alpha \in \text{ALFA}\}$

(ii) $\{\beta \mid \beta \in \text{BETA}\}$

(iii) $\{\langle\gamma, \gamma(t:\nu)\rangle \mid \gamma \in \text{GAMMA}, \gamma(t:\nu)$ un constituyente de $\gamma\}$

(iv) $\{\delta \mid \delta \in \text{DELTA}\}$

(v) $\{\alpha^{\circ} \mid \alpha^{\circ} \in \omega\text{-ALFA}\}$

(vi) $\{\beta^{\circ} \mid \beta^{\circ} \in \omega\text{-BETA}\}$

(vii) $\{\theta' \mid \theta$ es un axioma de orden y θ' es una $\hat{\Sigma}$-instancia monomórfica de $\theta\}$.

Diremos que un par de fórmulas $\langle\varphi, \psi\rangle$ es relevante para un conjunto de fórmulas Ψ cuando $\varphi \in \Psi$ y $\psi \notin \Psi$. Diremos que α es relevante para Ψ cuando $\alpha \in \Psi$ y $\alpha_1 \notin \Psi$ o $\alpha_2 \notin \Psi$. Diremos que β es relevante para Ψ cuando $\beta \in \Psi$ y $\beta_1 \notin \Psi$ y $\beta_2 \notin \Psi$. Diremos que δ es relevante para Ψ si y solo si $\delta \in \Psi$ y para todo símbolo de constante c nuevo, si ν es el tipo de la variable afectada por el cuantificador de δ, $\delta(c:\nu) \notin \Psi$. Diremos que α° es relevante para Ψ si y solo si $\alpha^{\circ} \in \Psi$ y $\alpha_i{}^{\circ} \notin \Psi$ para algún $i<\omega$. Diremos que β° es relevante para Ψ si y solo si $\beta \in \Psi$ y $\beta_i{}^{\circ} \notin \Psi$ para ningún $i<\omega$. Diremos que una $\hat{\Sigma}$-instancia monomórfica θ' de un axioma de orden θ es relevante para Ψ si y solo si $\theta' \notin \Psi$.

El siguiente algoritmo devuelve un tableau completo para Φ.

1. $k := 0$
2. $\mathcal{T}_k := \bullet\ M\hat{\Sigma}(\Phi)$ (es un solo nodo etiquetado por el conjunto $M\hat{\Sigma}(\Phi)$)
3. Si $\mathcal{T}_k$ es completo, parar.

4. Sea $\Psi = \{\Psi_R \mid R$ es una rama abierta de $\mathcal{T}_k\}$. Para cada $\Psi_R \in \Psi$ se toma, si existe, la menor fórmula (o par de fórmulas) de la enumeración relevante a Ψ_R y se aplica simultáneamente a todos los elementos de Ψ, la correspondiente regla de expansión con respecto a ϕ. El resultado de esta operación es el tableau $\mathcal{T}_{k+1}$.
5. $k := k + 1$
6. Ir a 3

Definimos $\mathcal{T}_\phi$ como el límite de los tableaux obtenidos mediante este algoritmo. Tanto si el algoritmo para y la sucesión es finita, como si es infinita, $\mathcal{T}_\phi$ es un tableau completo para ϕ, como probamos a continuación.

Por construcción, $\mathcal{T}_\phi$ es un tableau para ϕ. En efecto, el paso 2 asegura la condición de la definición de $\mathcal{T}_0$, y por el paso 4 sabemos que para cualquier k, $\mathcal{T}_{k+1}$ es una expansión de $\mathcal{T}_k$ con respecto a ϕ.

Además $\mathcal{T}_\phi$ es completo porque por un lado, si el algoritmo para, la condición de parada nos garantiza que obtenemos un tableau completo. Supongamos pues, que $\mathcal{T}_\phi$ es el límite de una sucesión infinita. Entonces, por el paso 2 del algoritmo, se verifica que toda rama abierta R de $\mathcal{T}_\phi$ es tal que la unión de los conjuntos que etiquetan sus nodos contiene a $\hat{M\Sigma}(\phi)$. Vamos a ver como esta unión satisface las distintas condiciones que caracterizan a los conjuntos de Hintikka.

Las condiciones (a), (a*), (b), (b*), (c) y (d) se comprueban utilizando el siguiente razonamiento: denotamos por R_1 la parte de la rama R correspondiente al subárbol $\mathcal{T}_1$. Sea φ una fórmula no básica de la unión de los conjuntos que etiquetan los nodos de R y $s<\omega$ la posición de φ en la enumeración considerada para definir el algoritmo de construcción del tableau. Si φ aparece por primera vez en R_1, entonces en R_{1+s} están los constituyentes de φ.

La condición (e) se verifica porque para cada instancia θ' adecuada a la unión de los conjuntos que etiquetan los nodos de la rama R, si $s<\omega$ es la posición de θ' en la enumeración inicial entonces θ' aparece en R_s.

La condición (f) se satisface por ser R una rama abierta. ■

Veamos algunos ejemplos de construcciones de tableaux para fórmulas de PLPR.

Ejemplo 1.3.3

Consideremos la signatura $\Sigma = \langle \Sigma_t, \Sigma_d \rangle$, siendo:

Σ_t = {nat/0, list/1} y

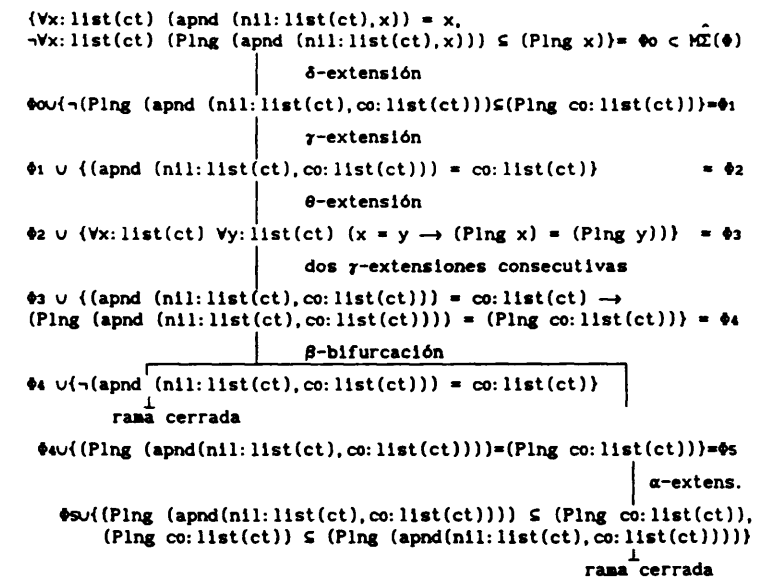
Σ_d = {nil:$\rightarrow\rho$, Plng:list(ρ)$\rightarrow$nat, apnd:list(ρ)$\times$list(ρ)$\rightarrow$list(ρ),
cons:$\rho\times$list(ρ)$\rightarrow$list(ρ)}

Supongamos que queremos deducir la fórmula:

$\forall x$:list(ρ) (Plng (apnd (nil,x))) $\leq$ (Plng x), a partir del axioma:

$\forall x$:list(ρ) (apnd (nil,x)) = x. Para ello construimos un tableau para el siguiente conjunto:

Φ = {$\forall x$:list(ρ) (apnd(nil,x)) = x,
$\neg\forall x$:list(ct) (Plng (apnd (nil:list(ct),x)):list(ct)) $\leq$ (Plng x):nat}



Este primer ejemplo es similar a los tableaux de primer orden clásicos salvo en la forma en que se construye el conjunto inicial Φ_0 instanciando de forma adecuada las fórmulas polimórficas. Lo que sigue es un ejemplo sencillo de un tableau en el cual se realizan expansiones propias de los tableaux para PLPR que hemos definido.

Ejemplo 1.3.4

Consideremos la signatura $\Sigma_{\mathbb{N}}$ del ejemplo §I-2.4.1 y como allí, utilizaremos la simplificación fact:nat→nat para representar

(μX:nat→nat.(λx:nat.(if (es_cero x) then (suc 0) else
(* (x,(X (pred x)))))))

Emplearemos también la notación 1 en lugar de (suc 0). Se trata de probar (fact 1) = 1 a partir de los axiomas siguientes referentes a las propiedades de los naturales.

(i) ∀x:nat ∀y:nat((suc x) = (suc y) → x = y)

(ii) ∀x:nat ¬(suc x) = 0

(iii) (es_cero 0) = true

(iv) ∀x:nat(¬x ≤ ⊥:nat →(es_cero (suc x)) = false)

(v) ∀x:nat(pred (suc x)) = x

(vi) ∀x:nat(+ (x,0)) = x

(vii) ∀x:nat ∀y:nat (+ (x,(suc y))) = (suc (+ (x,y)))

(viii) ∀x:nat (¬x ≤ ⊥:nat → (* (x,0) = 0)

(ix) ∀x:nat ∀y:nat(* (x,suc y))) = (+ (x,(* (x,y))))

Se trata por tanto, de encontrar un tableau cerrado para el conjunto de fórmulas $\Phi = \{(i),\ldots,(ix), \neg(\text{fact } 1) = 1\}$.

En la construcción de este tableau, que aparece en la figura 2, suprimiremos la escritura detallada de los pasos propios de los tableaux de la lógica clásica de primer orden y en los nodos escribimos sólo las fórmulas a las cuales vamos a hacer referencia.

fact^1 representa la i-ésima aproximación sintáctica de fact:nat→nat, éstas aproximaciones pueden ser expresadas de la siguiente forma:

fact^0 = (λx:nat.(if (es_cero x) then 1 else ⊥:nat

fact^1 = (λx:nat.(if (es_cero x) then 1 else
(* (x,(if (es_cero (pred x)) then 1 else ⊥)))))

fact^2 = (λx:nat.(if (es_cero x) then 1 else
(* (x,(if (es_cero (pred x)) then 1 else
(* ((pred x),(if (es_cero (pred (pred x))) then 1 else ⊥)))))))))

....

Desarrollamos la rama etiquetada con (0) en la figura 3.

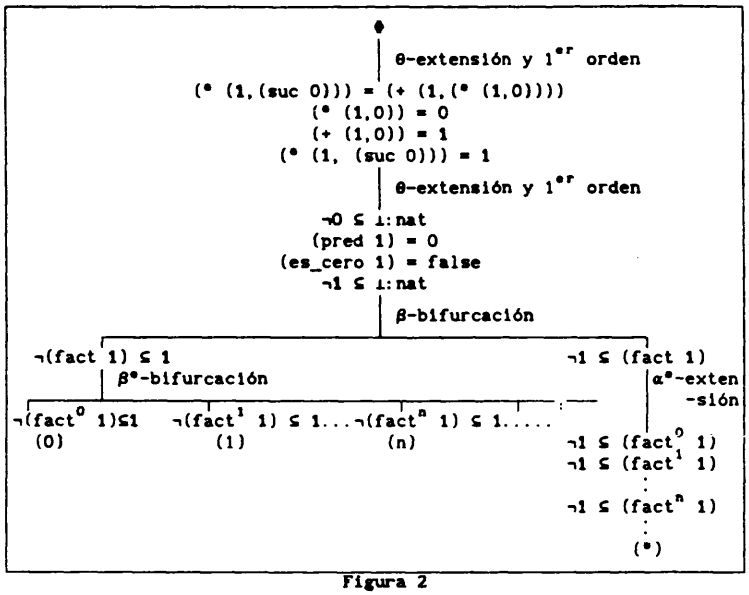


Figura 2

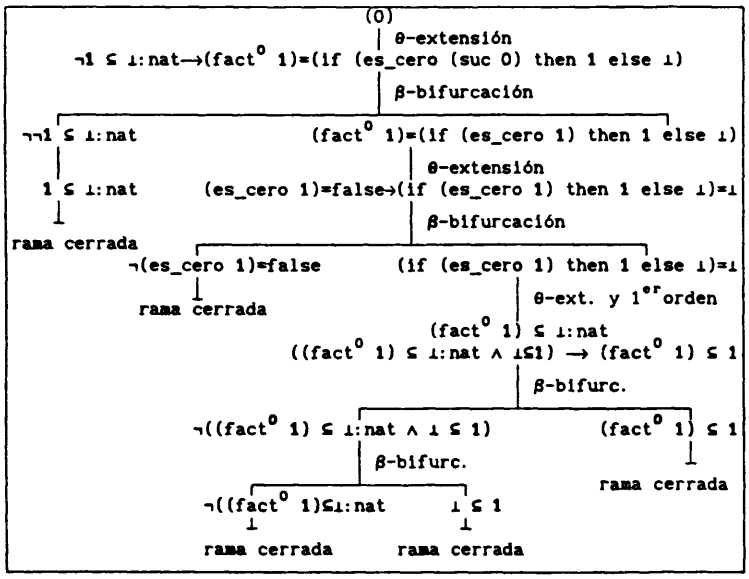


Figura 3

Prolongamos ahora la rama (1) en la figura 4. Los primeros pasos son análogos a los dados para la rama (0) y no los escribimos.

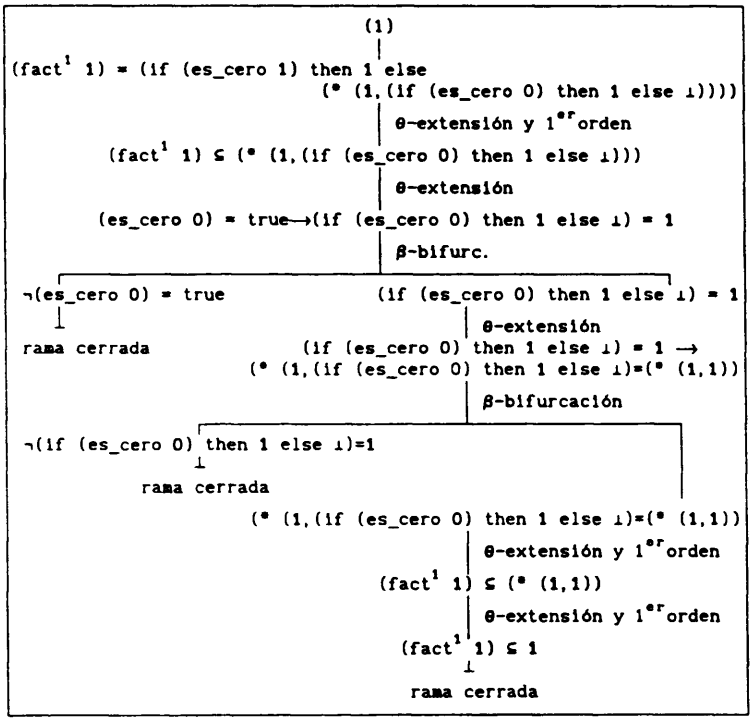


Figura 4

Para todo n ≥ 1, podemos prolongar (n) de la misma forma que para (1), de manera que se van cerrando todos los subárboles, y al final se obtiene una rama en cuya hoja aparece $(fact^n\ 1) \subseteq 1$ que sirve para cerrar la rama (n).

La rama (*) puede ser prolongada trabajando de la misma forma que hemos hecho para (1) en adelante, es decir, iremos cerrando subramas hasta obtener una rama que tenga en su hoja $1 \subseteq (fact^n\ 1)$, para cualquier n ≥ 1, que servirá para cerrar la rama (*) y por tanto el tableau.

2. CORRECCION Y COMPLETITUD DE LOS TABLEAUX

En esta sección demostraremos que el algoritmo de los tableaux para PLPR resulta ser un método correcto y completo que permite semi-decidir si un conjunto de fórmulas es insatisfactible. Además se obtendrán condiciones que convierten a los tableaux en una guía para obtener cálculos completos para PLPR.

2.1 CORRECCION DEL METODO DE LOS TABLEAUX

Proposición 2.1.1

Sea $\mathcal{T}$ un tableau para un conjunto $\Phi \subseteq L(\Sigma)$ con todas sus ramas finitas y sea Φ_0 el subconjunto de fórmulas de Φ utilizado en la construcción de $\mathcal{T}$. Para cualquier Σ-interpretación $\mathfrak{I}$, si $\mathfrak{I} \models \Phi_0$ entonces existen una rama de $\mathcal{T}$ con Ψ_0 etiquetando su hoja y una $\hat{\Sigma}$-interpretación $\hat{\mathfrak{I}}$ que es una extensión de $\mathfrak{I}$, tales que $\hat{\mathfrak{I}} \models \Psi_0$.

Demostración:

Vamos a ver que esta propiedad se cumple para todo $\mathcal{T}_i$ de la sucesión $\mathcal{T}_0, \mathcal{T}_1, \ldots \mathcal{T}_k, \ldots$ cuyo límite es $\mathcal{T}$. Lo probamos por inducción sobre i.

Para $i = 0$, $\mathcal{T}_0$ es igual a un solo nodo $\Psi_0 \subseteq M\hat{\Sigma}(\Phi)$, Ψ_0 estará formado por $\hat{\Sigma}$-instancias monomórficas de Φ_0 con lo que si $\mathfrak{I} \models \Phi_0$ los resultados semánticos del capítulo anterior nos aseguran que $\mathfrak{I} \models \Psi_0$ y por el lema de coincidencia cualquier $\hat{\Sigma}$-interpretación $\hat{\mathfrak{I}}$ que sea extensión de $\mathfrak{I}$ será tal que $\hat{\mathfrak{I}} \models \Psi_0$.

Para $i>0$, supongamos que en $\mathcal{T}_i$ existe una rama R_i con Ψ_i etiquetando su hoja y tal que si $\mathfrak{I} \models \Phi_0$ entonces existe una $\hat{\Sigma}$-interpretación $\hat{\mathfrak{I}}$ que es una extensión de $\mathfrak{I}$ de forma que $\hat{\mathfrak{I}} \models \Psi_i$. Sabemos que $\mathcal{T}_{i+1}$ es una expansión de $\mathcal{T}_i$ con respecto a Φ, si Ψ_i no se expande al pasar de $\mathcal{T}_i$ a $\mathcal{T}_{i+1}$ entonces R_i es una rama de $\mathcal{T}_{i+1}$ y no hay nada más que probar; en caso contrario, pueden haber ocurrido dos cosas:

A) Ψ_i ha sido extendida con respecto Φ y se ha producido una rama R_{i+1} de $\mathcal{T}_{i+1}$ con Ψ_{i+1} etiquetando su hoja, o

B) Ψ_i ha sido bifurcada.

Si ha tenido lugar A), podemos a su vez distinguir casos:

- Si $\alpha \in \Psi_i$ y $\hat{\mathfrak{I}} \models \Psi_i$, por las propiedades de las fórmulas de la clase

ALFA se tiene $\hat{\mathfrak{I}} \vdash \alpha_1$ y $\hat{\mathfrak{I}} \vdash \alpha_2$ luego $\hat{\mathfrak{I}} \vdash \Psi_i \cup \{\alpha_1, \alpha_2\}$ y como Ψ_{i+1} es precisamente $\Psi_i \cup \{\alpha_1, \alpha_2\}$ hemos obtenido el resultado esperado.

- Si $\alpha^* \in \Psi_i$ y $\hat{\mathfrak{I}} \vdash \Psi_i$, por las propiedades de la clase ω-ALFA, podemos asegurar que $\hat{\mathfrak{I}} \vdash \alpha_j^*$ para todo $j<\omega$ luego $\hat{\mathfrak{I}} \vdash \Psi_i \cup \{\alpha_j^* | j<\omega\}$ o lo que es lo mismo $\hat{\mathfrak{I}} \vdash \Psi_{i+1}$.

- Si $\gamma \in \Psi_i$ y $\hat{\mathfrak{I}} \vdash \Psi_i$, se tiene $\hat{\mathfrak{I}} \vdash \gamma(t:\nu)$ para todo $\hat{\Sigma}$-término monomórfico $t:\nu$, por las propiedades de las fórmulas universales. En particular $\hat{\mathfrak{I}} \vdash \gamma(t:\nu)$ para el constituyente $\gamma(t:\nu)$ que forma parte de Ψ_{i+1}, luego $\hat{\mathfrak{I}} \vdash \Psi_{i+1}$.

- Si $\delta \in \Psi_i$ y $\hat{\mathfrak{I}} \vdash \Psi_i$, por las propiedades de las fórmulas existenciales sabemos que si $\hat{\mathfrak{I}} \vdash \delta$ entonces existe un $a \in D_\nu$ tal que $\hat{\mathfrak{I}}(a/c:\nu) \vdash \delta(c:\nu)$ siendo $\delta(c:\nu)$ el constituyente de δ que forma parte de Ψ_{i+1}. Por el lema de coincidencia tenemos $\hat{\mathfrak{I}}(a/c:\nu) \vdash \Psi_{i+1}$ ya que $c:\nu$ no aparece en Ψ_{i+1}, luego existe una $\hat{\Sigma}$-interpretación $\hat{\mathfrak{I}}(a/c:\nu)$ que es una extensión de $\mathfrak{I}$ y que verifica $\hat{\mathfrak{I}}(a/c:\nu) \vdash \Psi_{i+1}$.

- Si $\Psi_{i+1} = \Psi_i \cup \{\theta'\}$ siendo θ' una $\hat{\Sigma}$-instancia monomórfica de un axioma de orden θ adecuada a Ψ_i, entonces $\hat{\mathfrak{I}} \vdash \Psi_i \Rightarrow \hat{\mathfrak{I}} \vdash \Psi_{i+1}$ porque 1.2.3 nos asegura $\hat{\mathfrak{I}} \vdash \theta$ para cualquier axioma de orden θ y por tanto, $\hat{\mathfrak{I}} \vdash \theta'$ para cualquier θ' que sea una $\hat{\Sigma}$-instancia monomórfica de θ.

- Si Ψ_i se extiende añadiendo una $\hat{\Sigma}$-instancia monomórfica $\varphi\sigma$ de una fórmula $\varphi \in \Phi$ con σ adecuada a φ y a Ψ_i y si $\mathfrak{I} \vdash \Phi_0 \cup \{\varphi\}$, por hipótesis de inducción, $\hat{\mathfrak{I}} \vdash \Psi_i$, y por el lema de coincidencia, $\mathfrak{I} \vdash \varphi$ implica $\hat{\mathfrak{I}} \vdash \varphi$. Por tanto, $\hat{\mathfrak{I}}$ también será modelo de todas las $\hat{\Sigma}$-instancias monomórficas de φ, en particular tendremos $\hat{\mathfrak{I}} \vdash \varphi\sigma$ y entonces $\hat{\mathfrak{I}} \vdash \Psi_{i+1}$.

Si ha tenido lugar B) podemos a su vez distinguir:

- Si $\beta \in \Psi_i$ y R_i da lugar a dos ramas de $\mathcal{T}_{i+1}$ que llamamos $R_{i+1,1}$ y $R_{i+1,2}$ con $\Psi_{i+1,1} = \Psi_i \cup \{\beta_1\}$ y $\Psi_{i+1,2} = \Psi_i \cup \{\beta_2\}$ como conjuntos de fórmulas que etiquetan sus hojas. Entonces, si $\hat{\mathfrak{I}} \vdash \Psi_i$, por las propiedades de las fórmulas disyuntivas, deducimos $\hat{\mathfrak{I}} \vdash \beta_1$ o $\hat{\mathfrak{I}} \vdash \beta_2$. En el primer caso $\hat{\mathfrak{I}} \vdash \Psi_{i+1,1}$ y será $R_{i+1,1}$ la rama que demuestra la proposición y en el segundo caso lo será $R_{i+1,2}$.

- Si $\beta^* \in \Psi_i$ y R_i da lugar a ω ramas de $\mathcal{T}_{i+1}$ que llamamos $R_{i+1,j}$, con $\Psi_{i+1,j} = \Psi_i \cup \{\beta_j^*\}$ $(j<\omega)$ como conjuntos de fórmulas que etiquetan sus hojas, si $\hat{\mathfrak{I}} \vdash \Psi_i$ entonces, por las propiedades de la clase ω-BETA, se tiene $\hat{\mathfrak{I}} \vdash \beta_k^*$ para algún $k<\omega$. Concluimos entonces que siempre existe una rama $R_{i+1,k}$ de $\mathcal{T}_{i+1}$ y una $\hat{\Sigma}$-interpretación $\hat{\mathfrak{I}}$ extensión de $\mathfrak{I}$ tal que $\hat{\mathfrak{I}} \vdash \Psi_{i+1}$.

Hemos probado hasta ahora que la proposición es cierta para cualquier árbol de la sucesión $\mathcal{T}_0,\mathcal{T}_1,\ldots\mathcal{T}_k,\ldots$ cuyo límite es $\mathcal{T}$. Entonces, puesto que por hipótesis $\mathcal{T}$ siempre tiene sus ramas finitas, tendremos que para cualquier rama R de $\mathcal{T}$ existirá un $i<\omega$ tal que, para todo $j>i$, $R_j = R_i = R$ lo que nos demuestra la proposición para $\mathcal{T}$. ■

Teorema 2.1.2 (Corrección del método de los tableaux)

Si un conjunto de fórmulas $\Phi \subseteq L(\Sigma)$ tiene un tableau cerrado entonces Φ es insatisfactible.

Demostración:

Sea $\mathcal{T}$ un tableau cerrado para Φ, por definición, todas sus ramas serán finitas y cerradas. Si Φ fuera satisfactible, también lo sería el conjunto $\Phi_0 \subseteq \Phi$ utilizado en la construcción de $\mathcal{T}$ y por la proposición anterior existiría una rama R de $\mathcal{T}$ tal que si Ψ_R es el conjunto de fórmulas que etiqueta la hoja de R, entonces Ψ_R es satisfactible. Pero Ψ_R contiene todas las etiquetas de los nodos de R y por ser R una rama cerrada dará lugar a un conjunto no coherente que claramente no puede ser satisfactible con lo que, de suponer que Φ es satisfactible, hemos llegado a un absurdo lo que indica que Φ ha de ser insatisfactible. ■

Este teorema nos asegura que el método de los tableaux es un método de refutación para PLPR correcto en el sentido de que si de un conjunto Φ de Σ-fórmulas podemos deducir una fórmula φ, o lo que es lo mismo, podemos encontrar un tableau cerrado para $\Phi \cup \{\neg\varphi\sigma\}$, donde $\varphi\sigma$ es monomórfica y σ es una sustitución de tipos cuyos símbolos de tipo están en CT entonces $\Phi \cup \{\neg\varphi\sigma\}$ es insatisfactible y por el teorema §I-3.1.5 se tiene $\Phi \vdash \varphi$.

2.2 COMPLETITUD DEL METODO DE LOS TABLEAUX

Los tableaux para PLPR resultan ser, como se demuestra a continuación, un método de deducción completo para las fórmulas de PLPR. Este teorema se deduce del hecho (probado en la sección anterior) de que todo conjunto de fórmulas tiene un tableau completo, y de la satisfactibilidad de los conjuntos de Hintikka.

Con el fin de aliviar la demostración de que todo conjunto de

Hintikka H es satisfactible, hacemos una serie de consideraciones previas que nos permitirán definir la familia de dominios en la que se satisface H.

Lema 2.2.1

Sea H un conjunto de Hintikka cualquiera y sea Σ_H la signatura con la cual se forman las fórmulas de H. Para cada $\nu \in MT_P(\Sigma_H)$ denotamos por $T:\nu$ el conjunto de Σ_H-términos monomórficos de tipo ν construidos con las variables libres de las fórmulas de H, true, false, $\perp$, los símbolos de Σ_H y los operadores λ y μ. Sea $\approx_H$ una relación definida para cada conjunto $T:\nu$ de la siguiente forma:

$t_1:\nu \approx_H t_2:\nu \Leftrightarrow t_1:\nu \leq t_2:\nu \in H$ y $t_2:\nu \leq t_1:\nu \in H$

Entonces se verifica que para cada $\nu \in MT_P(\Sigma_H)$, $\approx_H$ es una relación de equivalencia en $T:\nu$.

Demostración:

Comprobemos las propiedades de una relación de equivalencia.

- Reflexiva: Por la condición (e) de la definición de conjunto de Hintikka, $\forall x:\nu\ x \leq x$ es una fórmula γ perteneciente a H, entonces por la condición (c) de los conjuntos de Hintikka, como por contrucción, todo $t:\nu \in T:\nu$ es adecuado a dicha fórmula y a H, $t:\nu \leq t:\nu \in H$ con lo que $t:\nu \approx_H t:\nu$.

- Simétrica: Se obtiene directamente de la definición de $\approx_H$.

- Transitiva: Supongamos que $t_1:\nu \approx_H t_2:\nu$ y $t_2:\nu \approx_H t_3:\nu$. Por la condición (e), $\forall x:\nu\ \forall y:\nu\ \forall z:\nu((x \leq y \wedge y \leq z) \rightarrow x \leq z) \in H$ y aplicando la condición c) tres veces consecutivas tenemos que la fórmula disyuntiva $(t_1:\nu \leq t_2:\nu \wedge t_2:\nu \leq t_3:\nu) \rightarrow t_1:\nu \leq t_3:\nu$ pertenece a H con lo que en virtud de la propiedad (b) de la definición de conjunto de Hintikka:

i) $\neg(t_1:\nu \leq t_2:\nu \wedge t_2:\nu \leq t_3:\nu) \in H$, o bien

ii) $t_1:\nu \leq t_3:\nu \in H$

Si se verifica i), aplicando (b) de nuevo, se deduce que al menos una de las fórmulas $\neg t_1:\nu \leq t_2:\nu$ o $\neg t_2:\nu \leq t_3:\nu$ debe estar en H lo cual es absurdo porque de la hipótesis de partida se puede deducir que $t_1:\nu \leq t_2:\nu \in H$ y $t_2:\nu \leq t_3:\nu \in H$, y por ser H un conjunto coherente no puede contener una fórmula y su negación. Por tanto, como i) no es cierto, ha de verificarse ii).

Razonando de manera similar podemos probar

$((t_3:\nu \subseteq t_2:\nu \wedge t_2:\nu \subseteq t_1:\nu) \rightarrow t_3:\nu \subseteq t_1:\nu) \in H$ y llegar a la conclusión de que $t_3:\nu \subseteq t_1:\nu \in H$. Esto junto con ii) nos permite concluir $t_1:\nu \approx_H t_3:\nu$. ■

Lema 2.2.2

La relación de equivalencia $\approx_H$ definida en el lema anterior cumple las siguientes propiedades de congruencia.

1. Respecto al producto.

$(t_1:\nu_1,..,t_n:\nu_n) \approx_H \perp:\nu_1 x...x\nu_n \Leftrightarrow t_1:\nu_1 \approx_H \perp:\nu_1$ para algún i $(1 \leq i \leq n)$ y si p. t. $1 \leq i \leq n$, no $s_1:\nu_1 \approx_H \perp:\nu_1$ y no $t_1:\nu_1 \approx_H \perp:\nu_1$, $s_1:\nu_1 \approx_H t_1:\nu_1$ para todo $1 \leq i \leq n$ si y solo si $(s_1:\nu_1,\ldots,s_n:\nu_n) \approx_H (t_1:\nu_1,\ldots,t_n:\nu_n)$.

2. Respecto a las funciones.

$t_1:\nu_1 \approx_H t_2:\nu_1 \Rightarrow (M\ t_1:\nu_1):\nu_2 \approx_H (M\ t_2:\nu_1):\nu_2$ para toda Σ_H-expresión funcional $M:\nu_1 \rightarrow \nu_2$ construida con las mismas condiciones que $T:\nu$.

3. Respecto a los predicados.

Si $t_1:\nu \approx_H t_2:\nu$ y $(p\ t_1:\nu) \in H$ entonces $(p\ t_2:\nu) \in H$.

4. Respecto a los booleanos.

Si $t \in T:bool$ entonces $t \approx_H true$ o $t \approx_H false$ o $t \approx_H \perp:bool$.

5. Respecto al condicional.

$t:bool \approx_H true \Rightarrow (if\ t:bool\ then\ t_1:\nu\ else\ t_2:\nu) \approx_H t_1:\nu$.

$t:bool \approx_H false \Rightarrow (if\ t:bool\ then\ t_1:\nu\ else\ t_2:\nu) \approx_H t_2:\nu$.

$t:bool \approx_H \perp:bool \Rightarrow (if\ t:bool\ then\ t_1:\nu\ else\ t_2:\nu) \approx_H \perp:\nu$.

Demostración:

1. Por la condición (e) de los conjuntos de Hintikka, $\forall x_1:\nu_1 \ldots \forall x_n:\nu_n\ ((x_1 \subseteq \perp:\nu_1 \vee .. \vee x_n \subseteq \perp:\nu_n) \leftrightarrow (x_1,\ldots,x_n) \subseteq \perp:\nu_1 x \ldots x\nu_n) \in H$, entonces, aplicando la condición (c) n veces, $(t_1:\nu_1 \subseteq \perp:\nu_1 \vee .. \vee t_n:\nu_n \subseteq \perp:\nu_n) \leftrightarrow (t_1:\nu_1,..,t_n:\nu_n) \subseteq \perp:\nu_1 x..x\nu_n \in H$, puesto que esta última es una fórmula conjuntiva, tendremos $(t_1:\nu_1 \subseteq \perp:\nu_1 \vee .. \vee t_n:\nu_n \subseteq \perp:\nu_n) \rightarrow (t_1:\nu_1,\ldots,t_n:\nu_n) \subseteq \perp:\nu_1 x \ldots x\nu_n \in H$ y $(t_1:\nu_1,\ldots,t_n:\nu_n) \subseteq \perp:\nu_1 x \ldots x\nu_n \rightarrow (t_1:\nu_1 \subseteq \perp:\nu_1 \vee .. \vee t_n:\nu_n \subseteq \perp:\nu_n) \in H$, aplicando la condición (b) tenemos:

i) $\neg(t_1:\nu_1 \subseteq \perp:\nu_1 \vee .. \vee t_n:\nu_n \subseteq \perp:\nu_n) \in H$ o $(t_1:\nu_1,..,t_n:\nu_n) \subseteq \perp:\nu_1 x..x\nu_n \in H$ y

ii) $\neg(t_1:\nu_1,..,t_n:\nu_n) \subseteq \perp:\nu_1 x..x\nu_n \in H$ o $(t_1:\nu_1 \subseteq \perp:\nu_1 \vee .. \vee t_n:\nu_n \subseteq \perp:\nu_n) \in H$

Si ve verifica i), en virtud de la condición (a), obtenemos $\neg t_1:\nu_1 \subseteq \perp:\nu_1 \in H$ p. t. $1 \leq i \leq n$, o $(t_1:\nu_1,\ldots,t_n:\nu_n) \subseteq \perp:\nu_1 x \ldots x\nu_n \in H$.

Si se verifica ii) por la condición (b) llegamos a $\neg(t_1:\nu_1,\ldots,t_n:\nu_n) \subseteq \perp:\nu_1 x \ldots x\nu_n \in H$ o $t_1:\nu_1 \subseteq \perp:\nu_1 \in H$ para algún i, $1 \leq i \leq n$.

Como i) y ii) tienen que verificarse simultáneamente y H es coherente deducimos $(t_1:\nu_1,..,t_n:\nu_n)\subseteq\perp:\nu_1\times..\times\nu_n \in H \Leftrightarrow t_1:\nu_1 \subseteq \perp:\nu_1 \in H$ para algún i, además $\perp:\nu_1 \subseteq t_1:\nu_1$ y $\perp:\nu_1\times...\times\nu_n \subseteq (t_1:\nu_1,..,t_n:\nu_n)$ pertenecen a H gracias al axioma de orden $\forall x:\tau\ \perp:\tau \subseteq x$. De todo ello se deduce la equivalencia que queríamos probar.

La segunda parte de la propiedad 1 es consecuencia de

$$\forall x_1:\nu_1...\forall x_n:\nu_n\ \forall y_1:\nu_1...\forall y_n:\nu_n((\neg x_1\subseteq\perp:\nu_1 \wedge...\wedge \neg x_n\subseteq\perp:\nu_n) \rightarrow$$
$$((x_1,...,x_n) \subseteq (y_1,...,y_n) \leftrightarrow (x_1 \subseteq y_1 \wedge...\wedge x_n \subseteq y_n)) \in H$$

razonando de manera similar al caso anterior.

Para probar 2 vemos que por la condición (e) de los conjuntos de Hintikka, $\forall x:\nu_1\ \forall y:\nu_1((x = y \rightarrow (M\ x):\nu_2 = (M\ y):\nu_2) \in H$ y aplicando (c) dos veces $((t_1:\nu_1 = t_2:\nu_1 \rightarrow (M\ t_1:\nu_1):\nu_2 = (M\ t_2:\nu_1):\nu_2) \in H$ con lo que utilizando la propiedad (b) deducimos:

i) $\neg t_1:\nu_1 = t_2:\nu_1 \in H$ o bien

ii) $(M\ t_1:\nu_1):\nu_2 = (M\ t_2:\nu_1):\nu_2) \in H$.

Si se verificara i), puesto que se trata de una fórmula disyuntiva, tendríamos $\neg t_1:\nu_1 \subseteq t_2:\nu_1 \in H$ o $\neg t_2:\nu_1 \subseteq t_1:\nu_1 \in H$ pero esto es imposible pues H es coherente y por verificarse $t_1:\nu_1 \approx_H t_2:\nu_1$ sabemos que tanto $t_1:\nu_1 \subseteq t_2:\nu_1$ como $t_2:\nu_1 \subseteq t_1:\nu_1$ son fórmulas de H.

El razonamiento anterior nos indica que la fórmula conjuntiva $(M\ t_1:\nu_1):\nu_2 = (M\ t_2:\nu_1):\nu_2)$ pertenece a H con lo que obtenemos la relación esperada.

La propiedad 3 se obtiene a partir la instancia del axioma de orden $\forall x:\nu\ \forall y:\nu((x = y \wedge (p\ x)) \rightarrow (p\ y))$ que pertenece a H por (e), y utilizando las propiedades (c), (a) y (b) de los conjuntos de Hintikka.

Al igual que en los casos anteriores sabemos que el axioma de orden $\forall x:bool\ (x = true \vee x = false \vee x = \perp:bool)$ está en H por (e). Las condiciones (c), (b) y (a) de los conjuntos de Hintikka permiten descomponer esta fórmula hasta llegar a la conclusión de que para cualquier t:bool de T:bool, $t:bool \subseteq true \in H$ y $true \subseteq t:bool \in H$, o bien $t:bool \subseteq false \in H$ y $false \subseteq t:bool \in H$, o bien $t:bool \subseteq \perp:bool \in H$ y $\perp:bool \subseteq t:bool \in H$. De esta forma tendremos probada la condición 4.

Demostramos con detalle el primer resultado de la propiedad 5. Como $\forall x:\nu\ \forall y:\nu\ \forall z:bool\ (z=true \rightarrow (if\ z\ then\ x\ else\ y) = x) \in H$ por (e), al aplicar (c) tres veces consecutivas podemos asegurar que la

fórmula $(t{:}bool{=}true \longrightarrow (\text{if } t{:}bool \text{ then } t_1{:}\nu \text{ else } t_2{:}\nu) = t_1{:}\nu) \in H$ que por tratarse de un fórmula disyuntiva nos lleva a $\neg t{:}bool{=}true \in H$ o $(\text{if } t{:}bool \text{ then } t_1{:}\nu \text{ else } t_2{:}\nu) = t_1{:}\nu \in H$. Pero $\neg t{:}bool{=}true$ no puede aparecer en H porque llegaríamos a una contradición con la coherencia de H al ser $t{:}bool \approx_H true$. Deducimos por tanto que la fórmula $(\text{if } t{:}bool \text{ then } t_1{:}\nu \text{ else } t_2{:}\nu) = t_1{:}\nu$ pertenece a H. Aplicando la condición (a), $(\text{if } t{:}bool \text{ then } t_1{:}\nu \text{ else } t_2{:}\nu) \subseteq t_1{:}\nu \in H$ y $t_1{:}\nu \subseteq (\text{if } t{:}bool \text{ then } t_1{:}\nu \text{ else } t_2{:}\nu) \in H$ que permite concluir $(\text{if } t{:}bool \text{ then } t_1{:}\nu \text{ else } t_2{:}\nu) \approx_H t_1{:}\nu$.

La demostración de la segunda y tercera parte de esta quinta propiedad es análoga al caso anterior utilizando los axiomas de orden $\forall x{:}\nu\ \forall y{:}\nu\ \forall z{:}bool\ (z = false \longrightarrow (\text{if } z \text{ then } x \text{ else } y) = y)$ y $\forall x{:}\nu\ \forall y{:}\nu\ \forall z{:}bool\ (z = \bot{:}bool \longrightarrow (\text{if } z \text{ then } x \text{ else } y) = \bot{:}\tau)$. ∎

Teorema 2.2.3 (Satisfactibilidad de los conjuntos de Hintikka)
Todo conjunto de Hintikka es satisfactible.

Demostración:

Sea H un conjunto de Hintikka cualquiera y sea Σ_H la signatura con la que se forman las fórmulas de H. Consideremos los conjuntos $MT_p(\Sigma_H)$ que denota el conjunto de los Σ_H-tipos monomórficos de primer orden, y $T(\Sigma_H) = \bigcup \{T{:}\nu \mid \nu \in MT_p(\Sigma_H)\}$.

Consideremos para cada $\nu \in MT_p(\Sigma_H)$ el conjunto $T{:}\nu$ y la relación $\approx_H$ de los lemas anteriores y sus respectivos espacios cocientes $T{:}\nu/\approx_H$. Representamos por $[t{:}\nu] \in T{:}\nu/\approx_H$ la clase de equivalencia del término $t{:}\nu \in T{:}\nu$. Se verifica que para cada $\nu \in MT_p(\Sigma_H)$, $\langle T{:}\nu/\approx_H, \subseteq_\nu \rangle$ es un cpo plano con $[\bot{:}\nu]$ como elemento mínimo y $\subseteq_\nu$ definido por $[t_1{:}\nu] \subseteq_\nu [t_2{:}\nu] \Leftrightarrow t_1{:}\nu \subseteq t_2{:}\nu \in H$.

Utilizando los axiomas de orden pertenecientes a H que expresan las propiedades reflexiva, transitiva y antisimétrica, se comprueba fácilmente que el orden que acabamos de definir es un orden parcial.

Vamos a comprobar que $\subseteq_\nu$ determina un orden plano. Supongamos $[t_1{:}\nu] \subseteq_\nu [t_2{:}\nu]$. Por las condiciones (e) y (c) de los conjuntos de Hintikka, la fórmula disyuntiva $t_1{:}\nu \subseteq t_2{:}\nu \longrightarrow (t_1{:}\nu{=}t_2{:}\nu \vee t_1{:}\nu{=}\bot{:}\nu)$ pertenece a H. Entonces, por la condición (b) se tiene $\neg t_1{:}\nu \subseteq t_2{:}\nu \in H$ o $(t_1{:}\nu = t_2{:}\nu \vee t_1{:}\nu = \bot{:}\nu) \in H$. Pero sabemos que $\neg t_1{:}\nu \subseteq t_2{:}\nu \notin H$ puesto que $[t_1{:}\nu] \subseteq_\nu [t_2{:}\nu]$ y H es coherente, por tanto aseguramos $(t_1{:}\nu = t_2{:}\nu \vee t_1{:}\nu = \bot{:}\nu) \in H$, y por la condición (b) $t_1{:}\nu = t_2{:}\nu \in H$

o $t_1:\nu = \bot:\nu \in H$. Si $t_1:\nu = t_2:\nu \in H$ aplicando (a) podemos afirmar que $t_1:\nu \leq t_2:\nu \in H$ y que $t_2:\nu \leq t_1:\nu \in H$ con lo que $[t_1:\nu] \leq_\nu [t_2:\nu]$ y $[t_2:\nu] \leq_\nu [t_1:\nu]$, por la propiedad antisimétrica del orden parcial $\leq_\nu$, deducimos $[t_1:\nu] = [t_2:\nu]$. En el caso de que $t_1:\nu = \bot:\nu \in H$ razonando como antes llegamos a la conclusión de que $[t_1:\nu] = [\bot:\nu]$.

Se define la Σ_H-estructura de tipos $\mathcal{T}_H = \langle T^{\mathcal{T}_H}, \{ct^{\mathcal{T}_H} | \in \Sigma_{Ht}\}\rangle$ donde $T^{\mathcal{T}_H} = \{\langle T:\nu/\approx_H, \leq_\nu\rangle | \nu \in MT_p(\Sigma_H)\}$ y $ct^{\mathcal{T}_H} = T:ct/\approx_H$ si ct es una constante de tipo y $ct^{\mathcal{T}_H}(T:\nu_1/\approx_H, \ldots, T:\nu_n/\approx_H) = T:ct(\nu_1, \ldots, \nu_n)/\approx_H$ si ct es un constructor de tipos de aridad n.

Para esta estructura de tipos se verifica $[\![\nu]\!]^{\mathcal{T}_H} = T:\nu/\approx_H$ para todo $\nu \in MT_p(\Sigma_H)$ pues:

- $T:bool/\approx_H = \mathbb{B}$ ya que la propiedad de congruencia de $\approx_H$ referente a los booleanos nos asegura que los únicos elementos del cpo plano $T:bool/\approx_H$ son [true], [false] y $[\bot:bool]$ como elemento mínimo. Por tanto, estos elementos pueden identificarse con tt, ff y $\bot_{bool}$, respectivamente.
- $T:ct/\approx_H = ct^{\mathcal{T}_H}$ por la definición de $\mathcal{T}_H$.
- $T:ct(\nu_1, \ldots, \nu_n)/\approx_H = ct^{\mathcal{T}_H}(T:\nu_1/\approx_H, \ldots, T:\nu_n/\approx_H)$ por definición de $\mathcal{T}_H$.
- $T:\nu_1 x \ldots x \nu_n/\approx_H = T:\nu_1/\approx_H \bullet \ldots \bullet T:\nu_n/\approx_H$ por las propiedades de congruencia de $\approx_H$ con respecto al producto.

Puesto que $T^{\mathcal{T}_H}$ coincide con la familia de dominios asociados a los tipos monomórficos, para definir una Σ_H-estructura de datos $\mathcal{D}_H$ con respecto a $\mathcal{T}_H$, bastará con conocer los valores de los símbolos de datos para cada una de las instancias monomórficas de su tipo más general. Por tanto, definimos $\mathcal{D}_H$ como sigue:

$c:\nu^{\mathcal{D}_H} := [c:\nu]$

$f:\nu_1 \to \nu_2^{\mathcal{D}_H} \in [T:\nu_1/\approx_H \xrightarrow[H]{\bullet} T:\nu_2/\approx_H]$ verificando:

$$f:\nu_1 \to \nu_2^{\mathcal{D}_H}([t:\nu_1]) := \begin{cases} [(f\ t:\nu_1)] & \text{si } (f\ t:\nu_1) \text{ aparece en } H \\ \text{cualquiera} & \text{en caso contrario} \end{cases}$$

$p:\nu^{\mathcal{D}_H} \leq T:\nu/\approx_H \setminus \{[\bot:\nu]\}$ verificando $[t:\nu] \in p:\nu^{\mathcal{D}_H} \Leftrightarrow (p\ t:\nu) \in H$.

Se hace notar que estas definiciones son correctas debido a las propiedades de congruencia de $\approx_H$ con respecto a las funciones y los predicados.

Por las características de $T^{\mathcal{T}_H}$, toda asignación de tipos $\eta \in AT_p^{\mathcal{T}_H}$ puede identificarse con una Σ_H-sustitución de tipos del mismo nombre.

Se define la valoración ξ_H con respecto a $\mathfrak{I}_H$, como una función que verifica:

$$\xi_H^1(x:\tau)\eta = \begin{cases}[x:\tau\eta] & \text{si } x:\tau\eta \text{ aparece libre en las fórmulas de H}\\ \text{cualquiera} & \text{en caso contrario}\end{cases}$$

$$\xi_H^2(X:\tau_1\to\tau_2)\eta([t:\tau_1\eta]) = \begin{cases}[(X\ t:\tau_1\eta)] & \text{si } (X\ t:\tau_1\eta) \text{ aparece en H}\\ \text{cualquiera} & \text{en caso contrario}\end{cases}$$

A continuación se prueba:

(A) La Σ_H-interpretación $\mathfrak{I}_H = \langle \mathfrak{I}_H, \mathfrak{D}_H, \xi_H\rangle$ satisface $\mathfrak{I}_H[t:\nu]\eta = [t:\nu]$ para cualesquiera $t:\nu \in T(\Sigma_H)$ y $\eta \in AT_p^{\mathfrak{I}_H}$.

(B) Si $\varphi \in H$ entonces $\mathfrak{I}_H \models \varphi$.

Probamos (A) por inducción en la construcción de los términos:

(Señalamos que dado que todos los términos de $T(\Sigma_H)$ son monomórficos podemos simplificar $\mathfrak{I}_H[t:\nu]\eta$ por el valor constante $\mathfrak{I}_H[t:\nu]$ para cualquier asignación $\eta \in AT_p^{\mathfrak{I}_H}$).

$\mathfrak{I}_H[x:\nu] = \xi_H(x:\nu) = [x:\nu]$ pues si $x:\nu \in T(\Sigma_H)$ entonces $x:\nu$ aparece libre en las fórmulas de H.

$\mathfrak{I}_H[\bot:\nu]$ = el elemento mínimo de $[\nu]^{\mathfrak{I}_H}$ es decir $[\bot:\nu]$.

$\mathfrak{I}_H[c:\nu] = c:\nu^{\mathfrak{D}_H} = [c:\nu]$.

$\mathfrak{I}_H[\text{true}] = tt = [\text{true}]$.

$\mathfrak{I}_H[\text{false}] = ff = [\text{false}]$.

$$\mathfrak{I}_H[(t_1:\nu_1,\ldots,t_n:\nu_n)] = \begin{cases}\langle \mathfrak{I}_H[t_1:\nu_1],\ldots,\mathfrak{I}_H[t_n:\nu_n]\rangle & \text{si } \mathfrak{I}_H[t_1:\nu_1] \neq \bot_{\nu_1} \text{ p. t. } 1\le i\le n\\ \bot_{\nu_1\times\ldots\times\nu_n}^{\mathfrak{I}_H} & \text{en caso contrario}\end{cases}$$

$$= \begin{cases}\langle [t_1:\nu_1],\ldots,[t_n:\nu_n]\rangle & \text{si } [t_1:\nu_1] \neq [\bot:\nu_1] \text{ para todo } 1\le i\le n\\ [\bot:\nu_1\times\ldots\times\nu_n] & \text{en caso contrario}\end{cases}$$

por hipótesis de inducción

$= [(t_1:\nu_1,\ldots,t_n:\nu_n)]$ por las propiedades de $\approx_H$ con respecto a $\bullet$.

$$\mathfrak{I}_H[(\text{if } t:\text{bool then } t_1:\nu \text{ else } t_2:\nu)]$$

$$= \begin{cases}\mathfrak{I}_H[t_1:\nu] & \text{si } \mathfrak{I}_H[t:\text{bool}] = tt\\ \mathfrak{I}_H[t_2:\nu] & \text{si } \mathfrak{I}_H[t:\text{bool}] = ff\\ \bot_\nu & \text{si } \mathfrak{I}_H[t:\text{bool}] = \bot_{\text{bool}}\end{cases}$$

$$= \begin{cases}[t_1:\nu] & \text{si } [t:\text{bool}] = tt\\ [t_2:\nu] & \text{si } [t:\text{bool}] = ff\\ [\bot:\nu] & \text{si } [t:\text{bool}] = \bot_{\text{bool}}\end{cases} \quad \text{por hipótesis de inducción}$$

$= [(\text{if } t:\text{bool then } t_1:\nu \text{ else } t_2:\nu)]$ por las propiedades de congruencia de $\approx_H$ referentes al condicional.

$\mathcal{I}_H[(M\ t{:}\nu_1){:}\nu_2] = \mathcal{I}_H[M{:}\nu_1{\to}\nu_2]\ (\mathcal{I}_H[t{:}\nu_1])$. Por las propiedades de los conjuntos de Hintikka podemos asegurar que $(M\ \bot{:}\nu_1) = \bot{:}\nu_2 \in H$ y $(t{:}\nu_1 = \bot{:}\nu_1 \to (M\ \bot{:}\nu_1) = (M\ t{:}\nu_1)) \in H$ llegando a la conclusión de que si $[t{:}\nu_1] = [\bot{:}\nu_1]$ entonces $[(M\ t{:}\nu_1)] = [\bot{:}\nu_2]$. Por tanto, como las expresiones funcionales se interpretan como funciones estrictas, se tiene $\mathcal{I}[(M\ \bot{:}\nu_1)] = [(M\ \bot{:}\nu_1)]$. Si por el contrario $[t{:}\nu_1] \neq [\bot{:}\nu_1]$, y por h.i., $\mathcal{I}_H[t{:}\nu_1] \neq \bot_{\nu_1}$, distinguimos las distintas formas que puede tomar $M{:}\nu_1{\to}\nu_2$.

- Puesto que por (e) $(\bot\ t{:}\nu_1){:}\nu_2 = \bot{:}\nu_2 \in H$ y se trata de una fórmula conjuntiva podemos deducir que $[(\bot\ t{:}\nu_1){:}\nu_2] = [\bot{:}\nu_2]$. Por otro lado de la semántica de PLPR sabemos que $\mathcal{I}_H[\bot{:}\nu_1{\to}\nu_2]\ (\mathcal{I}_H[t{:}\nu_1]) = \bot_{\nu_2} = [\bot{:}\nu_2]$ con lo que $\mathcal{I}_H[(\bot\ t{:}\nu_1){:}\nu_2] = [(\bot\ t{:}\nu_1){:}\nu_2]$.

- Para $f{:}\nu_1{\to}\nu_2$ se tiene $\mathcal{I}_H[f{:}\nu_1{\to}\nu_2] = f{:}\nu_1{\to}\nu_2^{\mathcal{D}_H}$ que por definición verifica $f{:}\nu_1{\to}\nu_2^{\mathcal{D}_H}\ ([t{:}\nu_1]) = [(f\ t{:}\nu_1){:}\nu_2]$ y por inducción en $t{:}\nu_1$, $\mathcal{I}_H[(f\ t{:}\nu_1){:}\nu_2] = f{:}\nu_1{\to}\nu_2^{\mathcal{D}_H}([t{:}\nu_1])$, entonces $\mathcal{I}_H[(f\ t{:}\nu_1){:}\nu_2] = [(f\ t{:}\nu_1){:}\nu_2]$.

- Para $X{:}\nu_1{\to}\nu_2$, $\mathcal{I}_H[X{:}\nu_1{\to}\nu_2]\ (\mathcal{I}_H[t{:}\nu_1]) = \xi_H(X{:}\nu_1{\to}\nu_2)\ ([t{:}\nu_1])$ por hipótesis de inducción, y por definición de ξ_H se tiene la igualdad $\mathcal{I}_H[(X\ t{:}\nu_1){:}\nu_2] = [(X\ t{:}\nu_1){:}\nu_2]$.

- Por la condición (e) de los conjuntos de Hintikka:

$\forall y_1{:}\nu_1 \ldots \forall y_n{:}\nu_n(\neg(y_1 \ldots y_n) \leq \bot{:}\nu_1 x \ldots x\nu_n \to$

$((\lambda x_1{:}\nu_1 \ldots x_n{:}\nu_n.\, t{:}\nu)(y_1, \ldots, y_n)) = t{:}\nu[y_1 \ldots y_n/x_1 \ldots x_n]) \in H$

Por las condiciones (c) y (b) podemos afirmar que se verifica al menos una de las dos afirmaciones siguientes:

i) $\neg\neg(t_1{:}\nu_1, \ldots, t_n{:}\nu_n) \leq \bot{:}\nu_1 x \ldots x\nu_n \in H$

ii) $((\lambda x_1{:}\nu_1 \ldots x_n{:}\nu_n.\, t{:}\nu)(t_1{:}\nu_1, \ldots, t_n{:}\nu_n)) = t{:}\nu\left[\frac{t_1{:}\nu_1 \ldots t_n{:}\nu_n}{x_1{:}\nu_1 \ldots x_n{:}\nu_n}\right] \in H$

Si se verifica i) entonces, por tratarse de una fórmula α se tiene $(t_1{:}\nu_1, \ldots, t_n{:}\nu_n) \leq \bot{:}\nu_1 x \ldots x\nu_n \in H$ y llegaríamos a la conclusión de que $[(t_1{:}\nu_1, \ldots, t_n{:}\nu_n)] = [\bot{:}\nu_1 x \ldots x\nu_n]$ que no es cierto por hipótesis de partida. Por tanto ha de verificarse ii) y deducimos $[((\lambda x_1{:}\nu_1 \ldots x_n{:}\nu_n.\, t{:}\nu)(t_1{:}\nu_1, \ldots, t_n{:}\nu_n))] = [t{:}\nu\left[\frac{t_1{:}\nu_1 \ldots t_n{:}\nu_n}{x_1{:}\nu_1 \ldots x_n{:}\nu_n}\right]]$. Por otro lado, por inducción en $t_i{:}\nu_i$ para $i = 1, \ldots, n$, se tiene

$\mathcal{I}_H[(\lambda x_1{:}\nu_1 \ldots x_n{:}\nu_n.\, t{:}\nu)]\ (\mathcal{I}_H[(t_1{:}\nu_1, \ldots, t_n{:}\nu_n)]) =$

$\mathcal{I}_H\left\{\frac{[t_1{:}\nu_1] \ldots [t_n{:}\nu_n]}{x_1{:}\nu_1 \ldots x_n{:}\nu_n}\right\}[t{:}\nu] = \mathcal{I}_H[t{:}\nu\left[\frac{t_1{:}\nu_1 \ldots t_n{:}\nu_n}{x_1{:}\nu_1 \ldots x_n{:}\nu_n}\right]]$ por lema de sust.

$= [t{:}\nu\left[\frac{t_1{:}\nu_1 \ldots t_n{:}\nu_n}{x_1{:}\nu_1 \ldots x_n{:}\nu_n}\right]]$ por hipótesis de inducción

Luego $\mathcal{I}_{\mathcal{H}}[((\lambda x_1{:}\nu_1 \ldots x_n{:}\nu_n.\, t{:}\nu)(t_1{:}\nu_1,\ldots,t_n{:}\nu_n))] =$

$$[((\lambda x_1{:}\nu_1 \ldots x_n{:}\nu_n.\, t{:}\nu)(t_1{:}\nu_1,\ldots,t_n{:}\nu_n))].$$

- Por las propiedades de los conjuntos de Hintikka como el axioma $\forall x{:}\tau_1(\neg((\mu X{:}\tau_1\to\tau_2.M)^1 x) \leq \bot{:}\tau_2 \to ((\mu X{:}\tau_1\to\tau_2.M)^1 x) = ((\mu X{:}\tau_1\to\tau_2.M)\ x))$ pertenece a H, se deduce que p.t. $i<\omega$ $((\mu X{:}\nu_1\to\nu_2.M)^i\ t{:}\nu_1) \leq \bot{:}\nu_2 \in H$ o $((\mu X{:}\nu_1\to\nu_2.M)^i\ t{:}\nu_1) = ((\mu X{:}\nu_1\to\nu_2.M)\ t{:}\nu_1) \in H$. Entonces para todo $i<\omega$ se cumple una de las dos igualdades siguientes:

- $[((\mu X{:}\nu_1\to\nu_2.M)^i\ t{:}\nu_1)] = [\bot{:}\nu_2]$ ó
- $[((\mu X{:}\nu_1\to\nu_2.M)^i\ t{:}\nu_1)] = [((\mu X{:}\nu_1\to\nu_2.M)\ t{:}\nu_1)]$

De estas igualdades, aplicando hipótesis de inducción y por las propiedades de las aproximaciones sintácticas del μ-operador y de los cpo's planos se concluye $\sqcup_{i=0}^{\omega} \mathcal{I}_{\mathcal{H}}[((\mu X{:}\nu_1\to\nu_2.M)^i t{:}\nu_1)] = \sqcup_{i=0}^{\omega}[((\mu X{:}\nu_1\to\nu_2.M)^i t{:}\nu_1)] = [((\mu X{:}\nu_1\to\nu_2.M)\ t{:}\nu_1)]$, luego $\mathcal{I}_{\mathcal{H}}[((\mu X{:}\nu_1\to\nu_2.M)\ t{:}\nu_1)] = [((\mu X{:}\nu_1\to\nu_2.M)\ t{:}\nu_1)]$.

Probamos (B) por inducción en la construcción de las fórmulas, teniendo en cuenta que puesto que se trata de fórmulas monomórficas podemos suprimir la escritura de η.

$t_1{:}\nu \leq t_2{:}\nu \in H \Leftrightarrow [t_1{:}\nu] \leq_\nu [t_2{:}\nu]$ por definición de $\leq_\nu \Leftrightarrow \mathcal{I}_{\mathcal{H}}[t_1{:}\nu] \leq_\nu \mathcal{I}_{\mathcal{H}}[t_2{:}\nu]$ por (A) $\Leftrightarrow \mathcal{I}_{\mathcal{H}} \vdash t_1{:}\nu \leq t_2{:}\nu$.

$(p\ t{:}\nu) \in H \Leftrightarrow [t{:}\nu] \in p{:}\nu^{\mathcal{D}_{\mathcal{H}}}$ por definición de $p{:}\nu^{\mathcal{D}_{\mathcal{H}}}$. Entonces, del resultado (A) deducimos $(p\ t{:}\nu) \in H \Leftrightarrow \mathcal{I}_{\mathcal{H}} \vdash (p\ t{:}\nu)$.

Si $\neg\varphi \in H$ distinguimos casos según sea φ.

Si $\neg t_1{:}\nu \leq t_2{:}\nu \in H$, supongamos que $\mathcal{I}_{\mathcal{H}} \vdash t_1{:}\nu \leq t_2{:}\nu$ entonces podemos afirmar que $[t_1{:}\nu] \leq_\nu [t_2{:}\nu]$ y por tanto $t_1{:}\nu \leq t_2{:}\nu \in H$, pero esto es absurdo porque H es coherente. Por tanto no $\mathcal{I}_{\mathcal{H}} \vdash t_1{:}\nu \leq t_2{:}\nu$ y por tratarse de una fórmula monomórfica $\mathcal{I}_{\mathcal{H}} \vdash \neg t_1{:}\nu \leq t_2{:}\nu$.

Si $\neg(p\ t{:}\nu) \in H$ y $\mathcal{I}_{\mathcal{H}} \vdash (p\ t{:}\nu)$ tendremos $[t{:}\nu] \in p{:}\nu^{\mathcal{D}_{\mathcal{H}}}$ que por la definición de $p{:}\nu^{\mathcal{D}_{\mathcal{H}}}$ implica $(p\ t{:}\nu) \in H$ pero esto es imposible por ser H coherente. Entonces como trabajamos con fórmulas monomórficas concluimos $\mathcal{I}_{\mathcal{H}} \vdash \neg(p\ t{:}\nu)$.

Si $\neg\neg\varphi \in H$, por tratarse de una fórmula α tendremos $\varphi \in H$ y por hipótesis de inducción podemos asegurar que $\mathcal{I}_{\mathcal{H}} \vdash \varphi$ que equivale a decir que no no $\mathcal{I}_{\mathcal{H}} \vdash \varphi$ y entonces $\mathcal{I}_{\mathcal{H}} \vdash \neg\neg\varphi$.

Si $\neg(\varphi \vee \psi) \in H$, tenemos una fórmula conjuntiva cuyos constituyentes pertenecen a H, es decir $\neg\varphi \in H$ y $\neg\psi \in H$. Por hipótesis de inducción se tiene $\mathcal{I}_{\mathcal{H}} \vdash \neg\varphi$ y $\mathcal{I}_{\mathcal{H}} \vdash \neg\psi$, y por la semántica de la

disyunción y la negación de fórmulas monomórficas podemos asegurar que $\mathfrak{I}_H \vdash \neg(\varphi \vee \psi)$.

Si $\neg(\varphi \wedge \psi) \in H$, tendremos una fórmula β en H, luego en H estará también alguno de sus constituyentes, es decir, $\neg\varphi \in H$ o $\neg\psi \in H$. Por hipótesis de inducción se tiene $\mathfrak{I}_H \vdash \neg\varphi$ o $\mathfrak{I}_H \vdash \neg\psi$, y por la semántica de la conjunción y la negación de fórmulas monomórficas podemos asegurar que $\mathfrak{I}_H \vdash \neg(\varphi \wedge \psi)$.

Si $\neg\exists x{:}\nu\ \varphi \in H$, como se trata de una fórmula γ, deducimos que $\neg\varphi[t{:}\nu/x{:}\nu] \in H$ para todo término $t{:}\nu$ adecuado a φ y a H que son precisamente los miembros de $T{:}\nu$. Supongamos que existe un $a \in T{:}\nu/\approx_H$ tal que $\mathfrak{I}_H\{a/x{:}\nu\} \vdash \varphi$, es decir, existe un $t{:}\nu \in T{:}\nu$ tal que $\mathfrak{I}_H\{[t{:}\nu]/x{:}\nu\} \vdash \varphi$ y por (A), $\mathfrak{I}_H\{\mathfrak{I}_H[t{:}\nu]/x{:}\nu\} \vdash \varphi$. Entonces, aplicando el lema de sustitución, $\mathfrak{I}_H \vdash \varphi[t{:}\nu/x{:}\nu]$ pero esto es absurdo pues $\neg\varphi[t{:}\nu/x{:}\nu] \in H$ y por hipótesis de inducción, no $\mathfrak{I}_H \vdash \varphi[t{:}\nu/x{:}\nu]$. Concluimos diciendo que no exite un $a \in T{:}\nu/\approx_H$ tal que $\mathfrak{I}_H\{a/x{:}\nu\} \vdash \varphi$ y por tanto, $\mathfrak{I}_H \vdash \neg\exists x{:}\nu\ \varphi$.

Si $\neg\forall x{:}\nu\ \varphi \in H$, por la propiedad (d) de los conjuntos de Hintikka se tiene $\neg\varphi[c{:}\nu/x{:}\nu] \in H$ para alguna constante auxiliar $c{:}\nu$, entonces, aplicando inducción, $\mathfrak{I}_H \vdash \neg\varphi[c{:}\nu/x{:}\nu]$. Supongamos que para todo $a \in T{:}\nu/\approx_H$ $\mathfrak{I}_H\{a/x{:}\nu\} \vdash \varphi$, que es equivalente a decir que para todo $t{:}\nu \in T{:}\nu$, $\mathfrak{I}_H\{[t{:}\nu]/x{:}\nu\} \vdash \varphi$ o bien, $\mathfrak{I}_H\{\mathfrak{I}_H[t{:}\nu]/x{:}\nu\} \vdash \varphi$. Entonces, apoyándonos en el lema de sustitución tenemos que para todo $t{:}\nu$ de $T{:}\nu$, $\mathfrak{I}_H \vdash \varphi[t{:}\nu/x{:}\nu]$ y en particular $\mathfrak{I}_H \vdash \varphi[c{:}\nu/x{:}\nu]$ lo cual es absurdo pues $\mathfrak{I}_H \vdash \neg\varphi[c{:}\nu/x{:}\nu]$. Concluimos que $\mathfrak{I}_H \vdash \neg\forall x{:}\nu\ \varphi$.

Si $(\varphi \vee \psi) \in H$, como se trata de una fórmula β, $\varphi \in H$ o $\psi \in H$, y por hipótesis de inducción $\mathfrak{I}_H \vdash \varphi$ o $\mathfrak{I}_H \vdash \psi$. De la semántica de la disyunción de fórmulas monomórficas deducimos $\mathfrak{I}_H \vdash (\varphi \vee \psi)$.

Si $(\varphi \wedge \psi) \in H$, por la condición (a) de los conjuntos de Hintikka, $\varphi \in H$ y $\psi \in H$, y por hipótesis de inducción $\mathfrak{I}_H \vdash \varphi$ y $\mathfrak{I}_H \vdash \psi$. De la semántica de las fórmulas conjuntivas deducimos $\mathfrak{I}_H \vdash (\varphi \wedge \psi)$.

Si $\exists x{:}\nu\ \varphi \in H$, por ser ésta una fórmula existencial, se verifica que $\varphi[c{:}\nu/x{:}\nu] \in H$ y por hipótesis de inducción $\mathfrak{I}_H \vdash \varphi[c{:}\nu/x{:}\nu]$. Por el lema de sustitución $\mathfrak{I}_H\{\mathfrak{I}_H[c{:}\nu]/x{:}\nu\} \vdash \varphi$ para alguna constante auxiliar $c{:}\nu$, que por (A) equivale a decir que existe un elemento $[c{:}\nu]$ de $T{:}\nu/\approx_H$ tal que $\mathfrak{I}_H\{[c{:}\nu]/x{:}\nu\} \vdash \varphi$ por lo que $\mathfrak{I}_H \vdash \exists x{:}\nu\ \varphi$.

Si $\forall x{:}\nu\ \varphi \in H$, por la condición (c) de los conjuntos de Hintikka

se tiene que $\varphi[t:\nu/x:\nu] \in H$ para todo $t:\nu \in T:\nu$. Entonces, por hipótesis de inducción, $\mathfrak{I}_H \vdash \varphi[t:\nu/x:\nu]$ y por el lema de sustitución $\mathfrak{I}_H\{\mathfrak{I}_H[t:\nu]/x:\nu\} \vdash \varphi$ para todo $t:\nu \in T:\nu$, es decir, para todo $[t:\nu]$ de $T:\nu/\approx_H$ se satisface $\mathfrak{I}_H\{[t:\nu]/x:\nu\} \vdash \varphi$ por lo que $\mathfrak{I}_H \vdash \forall x:\nu\ \varphi$. ■

Teorema 2.2.4 (Completitud del método de los tableaux)

Todo conjunto $\Phi \subseteq L(\Sigma)$ insatisfactible tiene un tableau cerrado.

Demostración:

El lema 1.3.2 nos asegura que podemos encontrar un tableau completo para Φ que llamamos tableau canónico y denotamos por $\mathcal{T}_\Phi$. Vamos a probar que $\mathcal{T}_\Phi$ es cerrado. Supongamos que $\mathcal{T}_\Phi$ es abierto. Si $\mathcal{T}_\Phi$ es de ramas finitas, para ser abierto tendrá que tener alguna rama abierta; si por el contrario $\mathcal{T}_\Phi$ no es de ramas finitas, tendrá alguna rama infinita y por tanto abierta. En cualquier caso $\mathcal{T}_\Phi$ tiene una rama abierta, R, que será completa por serlo $\mathcal{T}_\Phi$, y la unión de los conjuntos que etiquetan los nodos de R será un conjunto de Hintikka H que contiene a $M\hat{\Sigma}(\Phi)$. Puesto que todo conjunto de Hintikka es satisfactible, la unión de los conjuntos que etiquetan los nodos de R es satisfactible y con ello también lo será $M\hat{\Sigma}(\Phi)$. Pero $M\hat{\Sigma}(\Phi)$ contiene, por definición, todas las $\hat{\Sigma}$-instancias monomórficas de las fórmulas de Φ; dado que el modelo que satisface H cumple las hipótesis del teorema §I-3.1.6 concluimos que Φ es satisfactible, pero esto es absurdo por hipótesis luego $\mathcal{T}_\Phi$ ha de ser cerrado. ■

Del teorema de completitud y de §I-3.1.5, se puede deducir que si un conjunto de Σ-fórmulas Φ y una Σ-fórmula φ verifican $\Phi \vdash \varphi$, entonces, existe un tableau cerrado para el conjunto $\Phi \cup \{\neg\varphi\sigma\}$ siendo $\varphi\sigma$ monomórfica y $\sigma = \frac{ct_1,\ldots,ct_n}{\rho_1,\ldots,\rho_n}$ con ct_i constantes de tipo nuevas para $i=1,\ldots,n$.

2.3 CONDICIONES DE COMPLETITUD DE UN CALCULO

Representamos por $\vdash$ el símbolo de deribabilidad formal de un cálculo. Demostraremos que todo sistema de deducción para PLPR que verifique unas condiciones obtenidas a partir de las propiedades de los tableaux es completo.

Proposición 2.3.1

Sean Φ un conjunto no vacio de Σ-fórmulas, φ una Σ-fórmula cualquiera y $\mathcal{T}$ un tableau de ramas finitas para Φ. Sea $\mathcal{C}$ un cálculo para PLPR que cumple las condiciones siguientes.

(1) Si $\Phi \vdash \varphi$ y $\Phi \subset \Phi'$ entonces $\Phi' \vdash \varphi$.

(2) Para toda fórmula α, $\Phi \cup \{\alpha_1, \alpha_2\} \vdash \varphi \Rightarrow \Phi \cup \{\alpha\} \vdash \varphi$.

(3) Para toda fórmula β, $\Phi \cup \{\beta_1\} \vdash \varphi$, $\Phi \cup \{\beta_2\} \vdash \varphi \Rightarrow \Phi \cup \{\beta\} \vdash \varphi$.

(4) Para toda fórmula γ, $\Phi \cup \{\gamma(t:\nu)\} \vdash \varphi \Rightarrow \Phi \cup \{\gamma\} \vdash \varphi$ para todo $\hat{\Sigma}$-término monomórfico $t:\nu$ de tipo ν.

(5) Para toda fórmula δ, $\Phi \cup \{\delta(c:\nu)\} \vdash \varphi \Rightarrow \Phi \cup \{\delta\} \vdash \varphi$ siendo c una constante auxiliar de tipo ν tal que c no aparece en Φ ni en φ ni en δ.

(6) Para toda fórmula α°, $\Phi \cup \{\alpha_i{}^\circ\} \vdash \varphi \Rightarrow \Phi \cup \{\alpha^\circ\} \vdash \varphi$ para todo $i<\omega$.

(7) Para toda fórmula β°, $\Phi \cup \{\beta_i\} \vdash \varphi$ para todo $i<\omega \Rightarrow \Phi \cup \{\beta^\circ\} \vdash \varphi$.

(8) $\Phi \cup \{\theta'\} \vdash \varphi \Rightarrow \Phi \vdash \varphi$ para toda $\hat{\Sigma}$-instancia monomórfica de los axiomas de orden.

(9) $\Phi \cup \{\psi\sigma\} \vdash \varphi \Rightarrow \Phi \cup \{\psi\} \vdash \varphi$ para cualquier $\hat{\Sigma}$-sustitución de tipos σ.

Entonces, si para toda rama R de $\mathcal{T}$, se verifica $\Psi_R \vdash \varphi$, se tiene $\Phi \vdash \varphi$.

Demostración:

Por inducción en la profundidad de $\mathcal{T}$:

Si $p(\mathcal{T}) = 0$, por el lema 1.3.1, sabemos que $\mathcal{T}$ tiene un solo nodo etiquetado por Ψ_0 siendo $\Psi_0 = M\hat{\Sigma}(\Phi_0)$ con $\Phi_0 \subseteq \Phi$. Supuesto que $\Psi_0 \vdash \varphi$, aplicando (9) para cada fórmula de Ψ_0, se obtiene $\Phi_0 \vdash \varphi$ y por (1) deducimos $\Phi \vdash \varphi$.

Si $p(\mathcal{T}) > 0$, por el lema 1.3.1, $\mathcal{T}$ tiene subárboles $\mathcal{T}_i$, con i recorriendo un cierto conjunto $I \subseteq \mathbb{N}$, que son tales que para todo i de I, si Φ_i es el conjunto que etiqueta la raíz de $\mathcal{T}_i$, entonces $\mathcal{T}_i$ es un tableau para $\Phi \cup \Phi_i$. Supongamos que para toda rama R de $\mathcal{T}$ se verifica $\Psi_R \vdash \varphi$. Si R_i es la rama de $\mathcal{T}_i$ correspondiente a la rama R, tendremos que por construcción $\Psi_{R_i} \vdash \varphi$ para todas las ramas R_i de $\mathcal{T}_i$ y por hipótesis de inducción podemos asegurar que $\Phi \cup \Phi_i \vdash \varphi$. Ahora bien, sea Ψ el conjunto de fórmulas que etiqueta la raíz de $\mathcal{T}$, a partir del hecho $\Phi \cup \Phi_i \vdash \varphi$ probaremos $\Phi \vdash \varphi$ distinguiendo casos según la regla de expansión con respecto a Φ que se le aplicó a Ψ.

Supongamos que $\alpha \in \Psi$ entonces I es unitario, $\Phi_1 = \Psi \cup \{\alpha_1, \alpha_2\}$ y

como hemos visto $\Phi \cup \Phi_i \vdash \varphi$, entonces en virtud de la condición (2) se obtiene $\Phi \cup \Psi \vdash \varphi$, pero $\Psi \subseteq M\hat{\Sigma}(\Phi)$ con lo que por (9) deducimos $\Phi \vdash \varphi$.

Si $\beta \in \Psi$, $\gamma \in \Psi$, $\delta \in \Psi$, $\alpha^\circ \in \Psi$, la demostración es análoga aplicando las condiciones (3), (4), (5) y (6) respectivamente. Vamos a ver algún otro caso de forma detallada.

Si $\beta^\circ \in \Psi$ y Ψ fue β°-bifurcado entonces $I = \mathbb{N}$ y $\Phi_i = \Psi \cup \{\beta_i^\circ\}$ para $i \in I$. Como $\Phi \cup \Phi_i \vdash \varphi$ para todo $i<\omega$, aplicando (7), $\Phi \cup \Psi \vdash \varphi$ y por ser $\Psi \subseteq M\hat{\Sigma}(\Phi)$, por la condición (9) concluimos $\Phi \vdash \varphi$.

Si Ψ fue expandida con respecto a Φ alargando su rama con una fórmula $\psi \in M\hat{\Sigma}(\Phi)$ entonces $I = \{1\}$ y $\Phi_1 = \Psi \cup \{\psi\}$, aplicando la hipótesis de inducción, $\Phi \cup \Psi \cup \{\psi\} \vdash \varphi$, pero $\Psi \cup \{\psi\} \subseteq M\hat{\Sigma}(\Phi)$, entonces, aplicando (9) llegamos a $\Phi \vdash \varphi$.

Por último si Ψ fue expandido alargando su rama con una $\hat{\Sigma}$-instancia monomórfica de un axioma de orden, la condición (8) nos asegura el resultado esperado. ∎

Teorema 2.3.2

Sea $\Phi \subseteq L(\Sigma)$ y sea $\mathcal{C}$ un cálculo para PLPR que satisface las condiciones siguientes:

(0) $\Phi \vdash \varphi$ si $\varphi \in \Phi$.

(1) a (9) del teorema 2.3.1.

(10) $\Phi \cup \{\neg\varphi\} \vdash \psi$, $\Phi \cup \{\neg\varphi\} \vdash \neg\psi \Rightarrow \Phi \vdash \varphi$ si φ es monomórfica.

(11) $\Phi \cup \{(p\ \perp{:}\tau)\} \vdash \varphi$ para cualquier $\varphi \in L(\Sigma)$.

(12) $\Phi \vdash \varphi\sigma \Rightarrow \Phi \vdash \varphi$, si $\varphi\sigma$ es monomórfica, $\sigma = \frac{ct_1,\ldots,ct_n}{\rho_1,\ldots,\rho_n}$, siendo ct_i constantes de tipo que no aparecen en $\Phi \cup \{\varphi\}$ para todo i, $1 \le i \le n$.

Entonces $\mathcal{C}$ es un cálculo completo para PLPR.

Demostración:

Supongamos que $\Phi \vdash \varphi$ para $\varphi \in L(\Sigma)$ entones para una $\hat{\Sigma}$-sustitución de tipos σ tal que $\varphi\sigma$ es monomórfica y los $\hat{\Sigma}$-símbolos de tipo que aparecen en σ están en CT se tiene que $\Phi \cup \{\neg\varphi\sigma\}$ es insatisfactible (§I-3.1.5). Por otro lado, la completitud del método de los tableaux nos asegura que existe de un tableau cerrado $\mathcal{T}$ para $\Phi \cup \{\neg\varphi\sigma\}$ que por tanto, tiene sus ramas finitas y cerradas. Para cada rama R de $\mathcal{T}$ se verifica o bien: a) existe una fórmula ψ_R tal que $\psi_R \in \Psi_R$ y $\neg\psi_R \in \Psi_R$, o bien: b) existe una fórmula $(p\ \perp{:}\tau) \in \Psi_R$.

Si se verifica a), por (0), tenemos $\Psi_R \vdash \psi_R$ y $\Psi_R \vdash \neg\psi_R$ para cada rama R, y por (1), $\Psi_R \cup \{\neg\chi\} \vdash \psi_R$ y $\Psi_R \cup \{\neg\chi\} \vdash \neg\psi_R$ para toda rama R

siendo χ una fórmula monomórfica cualquiera. Entonces, de acuerdo con (10), se obtiene $\Psi_R \vdash \chi$ para todo R. De una manera similar se prueba que $\Psi_R \vdash \neg\chi$ para toda rama R. Estamos pues en condiciones de aplicar la proposición anterior y asegurar que $\Phi \cup \{\neg\varphi\sigma\} \vdash \chi$ y $\Phi \cup \{\neg\varphi\sigma\} \vdash \neg\chi$.

Si de da el caso b), por (11) se obtiene que $\Psi_R \vdash \chi$ y $\Psi_R \vdash \neg\chi$ para toda rama R, y por 2.3.1, al igual que en el caso anterior, tendremos $\Phi \cup \{\neg\varphi\sigma\} \vdash \chi$ y $\Phi \cup \{\neg\varphi\sigma\} \vdash \neg\chi$.

Puesto que en cualquier caso $\Phi \cup \{\neg\varphi\sigma\} \vdash \chi$ y $\Phi \cup \{\neg\varphi\sigma\} \vdash \neg\chi$ aplicamos (10) y obtenemos $\Phi \vdash \varphi\sigma$. La $\hat{\Sigma}$-sustitución de tipos σ satisface las condiciones impuestas en (12), por tanto, $\Phi \vdash \varphi$ como queríamos demostrar. ■

CAPITULO III

CALCULOS NATURALES Y ALGUNAS CUESTIONES SOBRE COMPLEJIDAD

En el capítulo anterior vimos que el algoritmo de los tableaux es un método de refutación para PLPR que permite semi-decidir si un conjunto de fórmulas es insatisfactible. También hemos señalado su facilidad de automatización y su utilidad para construir cálculos completos. Sin embargo, se puede objetar que el método de los tableaux no resulta muy natural desde el punto de vista de su similitud con los razonamientos humanos. Dado que esta similitud es una de las características a tener en cuenta cuando se diseña un sistema de demostración automática, estaremos interesados en construir nuevos cálculos con esta cualidad.

En este capítulo, apoyándonos en las condiciones de completitud obtenidas utilizando los tableaux, definiremos un cálculo de deducción natural para PLPR, que llamaremos $\mathcal{PRC}$, y demostraremos su corrección y completitud. Aunque más natural que los tableaux el nuevo método de deducción seguirá siendo un cálculo infinitario, al igual que todo cálculo completo para PLPR.

La razón por la que nuestra lógica no admite cálulos completos finitarios queda detallada en la segunda sección de este capítulo donde se prueba incluso, que la validez en PLPR es un problema Π^1_1-completo. En la demostración de este resultado emplearemos técnicas de reducción de problemas, en concreto, reduciremos un dominó recursivo Σ^1_1-completo al problema de satisfactibilidad.

Por último, con el fin de obtener un cálculo que sirva de base a un sistema de deducción automática eliminaremos las reglas infinitarias del sistema de deducción natural definido en la primera sección, incorporando en su lugar una regla de inducción de punto fijo. Las reglas infinitarias de $\mathcal{PRC}$ trabajan con fórmulas que tienen definiciones recursivas en su interior; como veremos en los ejemplos, el hecho de sustituir estas reglas por la inducción de punto fijo no nos impide derivar este tipo de fórmulas. La regla de inducción se define sobre un subconjunto del lenguaje, el conjunto de fórmulas continuas. Se demostrará que este conjunto es Σ_2-duro, lo que seguirá dificultando la implementación del cálculo. Sin embargo, es fácil reconocer de manera automática un gran número de fórmulas continuas.

Si nos limitamos a este conjunto, conseguiremos un cálculo con mejores propiedades computacionales que $\mathcal{PRC}$ y más práctico para el usuario, acostumbrado a hacer demostraciones usando la regla de inducción.

1. UN CALCULO DE DEDUCCION NATURAL PARA PLPR

El cálculo $\mathcal{PRC}$ que se presenta a continuación puede considerarse como un sistema de deducción natural que admite la técnica de la cancelación de hipótesis. Esta técnica refleja el modo bastante habitual de hacer demostraciones matemáticas suponiendo ciertas unas determinadas hipótesis, [Dal-80].

El sistema de reglas de derivación determina el cálculo, estas reglas sirven para derivar fórmulas a partir de otras. La parte superior de una regla son las premisas mientras que la parte inferior es llamada conclusión. Las reglas de $\mathcal{PRC}$ pueden agruparse según su utilidad en distintos grupos:

Existirán reglas propias de la lógica de primer orden; como en todo cálculo de deducción natural tendremos, por cada conectivo y cuantificador, una regla de introducción en la que éstos aparecen en la conclusión, y una regla de eliminación, en las que el conectivo o cuantificador aparece en las premisas.

Al ser $\mathcal{PRC}$ un cálculo para una lógica polimórfica, estará provisto de reglas de sustitución de tipos, es decir, reglas que permiten o bien concluir una fórmulas a la que se le ha aplicado una sustitución de tipos eliminando o instanciando una variable de tipo, o bien permiten deshacer esta sustitución introduciendo una variable de tipo.

Las reglas derivadas de los axiomas de orden definidos en §II-1.2.2 se presentan en varios bloques: uno refleja las propiedades del orden parcial plano y del producto estricto; otro incluye las propiedades del símbolo $\perp$; la regla de sustitución de términos iguales constituye otro bloque; mientras que las reglas referentes a los términos booleanos y condicional forman un grupo más; por otro lado, la existencia del $\lambda\mu$-cálculo se traduce en una regla de abstración y una regla de recursión.

Por último, tenemos un grupo de reglas que llamaremos de aproximación porque expresan la relación entre una fórmula que tiene un término recursivo y la misma fórmula donde dicho término se ha sustituido por sus aproximaciones sintácticas. Este grupo incluye reglas con un número infinito de premisas.

Una vez establecido el sistema de reglas y el concepto de derivación, se introducen una serie de reglas derivadas que, salvo por

el polimorfismo, son análogas a reglas de la lógica clásica tales como el tercero excluso, el teorema fundamental, el modus ponens, las reglas de De Morgan, la eliminación de la doble negación, etc.

1.1 EL SISTEMA DE REGLAS

Por ser $\mathcal{PRC}$ un cálculo de deducción natural, las deducciones o esquemas de derivación pueden considerarse como árboles cuya raíz representa la conclusión de la deducción y cuyas hojas son hipótesis o suposición de hipótesis que pueden eliminarse; el paso de los hijos al padre se realiza aplicando una regla de derivación. Usaremos la notación $\begin{matrix} D \\ \varphi \end{matrix}$ para representar un árbol de derivación con φ como conclusión. La cancelación de hipótesis queda reflejada por el principio de que si $\begin{matrix} \psi \\ D \\ \varphi \end{matrix}$ es un árbol de derivación con hipótesis ψ, entonces, $\begin{matrix} [\psi] \\ D \\ \frac{\varphi}{\chi} \end{matrix}$ es un árbol de derivación con ψ cancelado. Entre las premisas de una regla pueden aparecer notaciones del tipo $\begin{matrix} [\varphi] \\ \dot{\psi} \end{matrix}$ para indicar que existe un árbol de derivación con φ cancelado y con ψ como conclusión.

Definición 1.1.1

Las *reglas de derivación* que constituyen el cálculo $\mathcal{PRC}$ son las definidas a continuación.

Reglas propias de los conectivos

$$(\vee I)\ \frac{\varphi}{\varphi \vee \psi} \qquad \frac{\psi}{\psi \vee \varphi}$$

$$(\vee E)\ \frac{\varphi \vee \psi \qquad \begin{matrix}[\varphi] \\ \dot{\chi}\end{matrix} \qquad \begin{matrix}[\psi] \\ \dot{\chi}\end{matrix}}{\chi} \quad (\varphi \text{ y } \psi \text{ monomórfic.})$$

$$(\wedge I)\ \frac{\varphi \qquad \psi}{\varphi \wedge \psi}$$

$$(\wedge E)\ \frac{\varphi \wedge \psi}{\varphi} \qquad \frac{\varphi \wedge \psi}{\psi}$$

$$(\neg I)\ \frac{\begin{array}{c}[\varphi]\\ \vdots\\ \neg\psi\ \wedge\ \psi\end{array}}{\neg\varphi}\quad (\varphi \text{ monomórfica}) \qquad (\neg E)\ \frac{\begin{array}{c}[\neg\varphi]\\ \vdots\\ \neg\psi\ \wedge\ \psi\end{array}}{\varphi}\quad (\varphi \text{ monomórfica})$$

Reglas para los cuantificadores

$$(\forall I)\ \frac{\varphi[c:\tau/x:\tau]}{\forall x:\tau\ \varphi}$$

(c no aparece en φ ni en las hipótesis no canceladas de la derivación de $\varphi[c:\tau/x:\tau]$)

$$(\forall E)\ \frac{\forall x:\tau\ \varphi}{\varphi[t:\tau/x:\tau]} \qquad (\exists I)\ \frac{\varphi[t:\tau/x:\tau]}{\exists x:\tau\ \varphi}$$

$$(\exists E)\ \frac{\exists x:\tau\ \varphi \qquad \begin{array}{c}[\varphi[c:\tau/x:\tau]]\\ \vdots\\ \psi\end{array}}{\psi}$$

(c no aparece en φ ni en ψ ni en las hipótesis no canceladas de las derivaciones de $\varphi[c:\tau/x:\tau]$ y de $\exists x:\tau\ \varphi$)

Reglas de sustitución de tipos

$$(TSI)\ \frac{\varphi[ct/\rho]}{\varphi}$$

(ct no aparece en φ ni en las hipót. no canceladas de la derivación de $\varphi[ct/\rho]$)

$$(TSE)\ \frac{\varphi}{\varphi\sigma}$$

(para cualquier sustitución de tipos σ)

Reglas del orden parcial plano

$$(REF)\ \frac{}{t:\tau \subseteq t:\tau} \qquad (ANT)\ \frac{t_1:\tau \subseteq t_2:\tau \quad t_2:\tau \subseteq t_1:\tau}{t_1:\tau = t_2:\tau}$$

$$(FLAT)\ \frac{t_1:\tau \subseteq t_2:\tau}{t_1:\tau = t_2:\tau \vee t_1:\tau = \bot:\tau}$$

$$(PRO1)\ \frac{}{t_1:\tau_1 \subseteq \bot:\tau_1 \vee \ldots \vee t_m:\tau_m \subseteq \bot:\tau_m \leftrightarrow (t_1:\tau_1,\ldots,t_m:\tau_n) \subseteq \bot:\tau_1 \times \ldots \times \tau_n}$$

$$(PRO2)\ \frac{\neg t_1:\tau_1 \subseteq \bot:\tau_1 \wedge \ldots \wedge \neg t_n:\tau_n \subseteq \bot:\tau_n}{t_1:\tau_1 \subseteq s_1:\tau_1 \wedge \ldots \wedge t_n:\tau_n \subseteq s_n:\tau_n \leftrightarrow (t_1:\tau_1,\ldots,t_n:\tau_n) \subseteq (s_1:\tau_1,\ldots,s_n:\tau_n)}$$

Reglas propias del bottom

$$(\text{BOT})\ \frac{}{\bot:\tau \leq t:\tau} \qquad (\text{STR})\ \frac{}{(M\ \bot:\tau_1):\tau_2 = \bot:\tau_2}$$

$$(\text{DEF})\ \frac{}{(\lambda\ t:\tau_1):\tau_2 = \bot:\tau_2}$$

Reglas de los booleanos y el condicional

$$(\text{Bool})\ \frac{}{t:\text{bool} = \bot:\text{bool} \vee t:\text{bool} = \text{true} \vee t:\text{bool} = \text{false}}$$

$$(\bot\text{If})\ \frac{t:\text{bool} = \bot:\text{bool}}{(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau) = \bot:\tau}$$

$$(\text{TIf})\ \frac{t:\text{bool} = \text{true}}{(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau) = t_1:\tau}$$

$$(\text{FIf})\ \frac{t:\text{bool} = \text{false}}{(\text{if } t:\text{bool then } t_1:\tau \text{ else } t_2:\tau) = t_2:\tau}$$

Regla de sustitución de términos

$$(\text{SUB})\ \frac{t_1:\tau = t_2:\tau \quad \varphi[t_1:\tau/x:\tau]}{\varphi[t_2:\tau/x:\tau]}$$

Reglas del λμ-cálculo

$$(\text{ABS})\ \frac{\neg(t_1:\tau_1,..,t_n:\tau_n) \leq \bot:\tau_1 x...x\tau_n}{(\lambda x_1:\tau_1..x_n:\tau_n.\ t:\tau)(t_1:\tau_1,..,t_n:\tau_n) = t:\tau[t_1:\tau_1..t_n:\tau_n/x_1:\tau_1..x_n:\tau_n]}$$

$$(\text{REC})\ \frac{\neg((\mu X:\tau_1\rightarrow\tau_2.M)^1\ t:\tau_1) \leq \bot:\tau_2}{((\mu X:\tau_1\rightarrow\tau_2.M)^1\ t:\tau_1) = ((\mu X:\tau_1\rightarrow\tau_2.M)\ t:\tau_1)}\ (1<\omega)$$

Regla de contradicción

$$(\text{Ctr})\ \frac{(p\ \bot:\tau)}{\varphi}$$

Reglas para las aproximaciones del punto fijo

$$(\omega Ap)\ \frac{(t_1:\tau)^i \subseteq t_2:\tau\ (i<\omega)}{t_1:\tau \subseteq t_2:\tau} \qquad (\neg\omega Ap)\ \frac{\neg t_1:\tau \subseteq (t_2:\tau)^i\ (i<\omega)}{\neg t_1:\tau \subseteq t_2:\tau}$$

$$(Apr)\ \frac{t_1:\tau \subseteq (t_2:\tau)^i}{t_1:\tau \subseteq t_2:\tau}\ (i<\omega) \qquad (\neg Apr)\ \frac{\neg(t_1:\tau)^i \subseteq t_2:\tau}{\neg t_1:\tau \subseteq t_2:\tau}\ (i<\omega)$$

$$(PAp)\ \frac{(p\ (t:\tau)^i)}{(p\ t:\tau)}\ (i<\omega) \qquad (\neg PAp)\ \frac{\neg(p\ (t:\tau)^i)\ (i<\omega)}{\neg(p\ t:\tau)}$$ □

La restricción en algunas reglas de que las premisas sean monomórficas no impide hacer razonamientos sobre fórmulas polimórficas, puesto que las reglas propias del polimorfismo permiten incluir y eliminar variables de tipo.

Definición 1.1.2

El *conjunto de derivaciones* es el menor conjunto $\mathbb{D}$ que verifica:

(i) Para cualquier φ de $L(\Sigma)$, el árbol con un único elemento φ pertenece a $\mathbb{D}$.

(ii) Si para algún $I \subseteq \mathbb{N}$, $D_i \in \mathbb{D}$ entonces $\dfrac{\begin{matrix}D_i \\ \varphi_i\end{matrix}}{\chi}\ \dfrac{\varphi_i\ (i\in I)}{\chi}$ pertenece a $\mathbb{D}$ si χ se obtiene aplicando una regla de derivación a $\{\varphi_i \mid i \in I\}$.

Sea Γ un conjunto de fórmulas, se dice que φ es *derivable a partir de* Γ y se escribe $\Gamma \vdash \varphi$, si existe una derivación con φ como conclusión y todas sus hipótesis no canceladas en Γ. □

1.2 REGLAS DERIVADAS

Con objeto de simplificar las deducciones, vamos a considerar reglas que llamamos derivadas y que son tales que existe una derivación cuyas hipótesis coinciden con las premisas de la regla y cuya conclusión es la conclusión de la regla.

El carácter de estas reglas permite usarlas en una deducción como si se tratara de las reglas primitivas que constituyen el cálculo $\mathcal{PRC}$.

Las reglas de De Morgan son derivables incluso en el caso de fórmulas polimórficas:

$$(DM1)\ \frac{\neg(\varphi \vee \psi)}{\neg\varphi \wedge \neg\psi} \qquad (DM2)\ \frac{\neg(\varphi \wedge \psi)}{\neg\varphi \vee \neg\psi}$$

Comprobamos $\neg(\varphi \vee \psi) \vdash \neg\varphi \wedge \neg\psi$. Sea σ es una sustitución de tipos que sólo utiliza símbolos nuevos y que hace monomórficas a $\varphi\sigma$ y $\psi\sigma$, Consideremos la siguiente derivación.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg(\varphi \vee \psi)}{\neg(\varphi\sigma \vee \psi\sigma)}\,(TSE) \quad \dfrac{[\varphi\sigma]}{\varphi\sigma \vee \psi\sigma}\,(\vee I)}{\neg(\varphi \vee \psi)\sigma \wedge (\varphi \vee \psi)\sigma}\,(\wedge I)}{\neg\varphi\sigma}\,(\neg I) \qquad \dfrac{\dfrac{\dfrac{\neg(\varphi \vee \psi)}{\neg(\varphi\sigma \vee \psi\sigma)}\,(TSE) \quad \dfrac{[\psi\sigma]}{\varphi\sigma \vee \psi\sigma}\,(\vee I)}{\neg(\varphi \vee \psi)\sigma \wedge (\varphi \vee \psi)\sigma}\,(\wedge I)}{\neg\psi\sigma}\,(\neg I)}{\neg\varphi\sigma \wedge \neg\psi\sigma}\,(\wedge I)}{\neg\varphi \wedge \neg\psi}\,(TSI)$$

La prueba de que DM2 es derivable es similar.

Puesto que la lógica que hemos definido es una lógica clásica, la regla del tercero excluso es correcta y puede derivarse del sistema de reglas de $\mathcal{PRE}$. Tenemos por lo tanto la regla:

$$(Exc)\ \frac{}{\varphi \vee \neg\varphi}$$

que se deriva por medio de:

$$\dfrac{\dfrac{\dfrac{[\neg(\varphi \vee \neg\varphi)\sigma]}{\neg\varphi\sigma \wedge \neg\neg\varphi\sigma}\,(DM1)}{(\varphi \vee \neg\varphi)\sigma}\,(\neg E)}{(\varphi \vee \neg\varphi)}\,(TSI)$$

σ hace monomórfica a $\varphi\sigma$ usando símbolos nuevos

El teorema fundamental y el modus ponens quedan reflejados mediante las dos reglas derivadas (FTh) y (MP).

$$(FTh)\ \frac{\begin{matrix}[\varphi]\\ \vdots\\ \psi\end{matrix}}{\neg\varphi \vee \psi}\ (\varphi \text{ monomórfica}) \qquad (MP)\ \frac{\neg\varphi \vee \psi \quad \varphi}{\psi}$$

El siguiente esquema sirve para derivar la regla (FTh):

$$\dfrac{\dfrac{\dfrac{[\varphi]}{\psi}\,(\text{premisa})}{\neg\varphi \vee \psi}\,(\vee I) \qquad \dfrac{\dfrac{[\neg\varphi]}{\neg\varphi \vee \psi}\,(\vee I) \qquad \dfrac{}{\varphi \vee \neg\varphi}\,(\text{Exc})}{\neg\varphi \vee \psi}\,(\vee E)$$

Pueden enunciarse también reglas para introducir y eliminar la doble negación.

$$(\neg\neg I)\ \dfrac{\varphi}{\neg\neg\varphi} \qquad\qquad (\neg\neg E)\ \dfrac{\neg\neg\varphi}{\varphi}$$

Una derivación de $(\neg\neg I)$ puede ser la siguiente:

$$\dfrac{\varphi \qquad \dfrac{}{\neg\varphi \vee \neg\neg\varphi}\,(\text{Exc})}{\neg\neg\varphi}\,(\text{MP})$$

Utilizando estas reglas puede derivarse una regla dual al modus ponens que llamamos (MP').

$$(\text{MP}')\ \dfrac{\varphi \vee \psi \quad \neg\varphi}{\psi}$$

De las reglas propias del orden parcial plano, como la igualdad es la conjunción de dos desigualdades, podemos concluir las propiedades reflexiva, simétrica y transitiva de la igualdad.

$$(=\text{REF})\ \dfrac{}{t{:}\tau = t{:}\tau}$$

$$(=\text{SIM})\ \dfrac{t_1{:}\tau = t_2{:}\tau}{t_2{:}\tau = t_1{:}\tau}$$

$$(=\text{TRAN})\ \dfrac{t_1{:}\tau = t_2{:}\tau \wedge t_2{:}\tau = t_3{:}\tau}{t_1{:}\tau = t_3{:}\tau}$$

1.3 CORRECCION Y COMPLETITUD DEL CALCULO

La prueba de la corrección del cálculo se basa en la corrección de todas sus reglas mientras que la prueba de la completitud se basa en el cumplimiento de las condiciones de completitud del teorema §II-2.3.2.

Teorema 1.3.1 (Corrección de $\mathcal{PRC}$)

Para todo $\Phi \subseteq L(\Sigma)$ y toda $\varphi \in L(\Sigma)$ se verifica $\Phi \vdash \varphi \Rightarrow \Phi \models \varphi$.

Demostración:

Por definición de $\Phi \vdash \varphi$ es suficiente probar que para cada derivación $D \in \mathbb{D}$ con hipótesis en Φ y conclusión φ se verifica $\Phi \models \varphi$. Lo probamos por inducción en D.

Si D sólo tiene un elemento, éste será φ y puesto que todas las reglas de $\mathcal{PRC}$ sin premisas tiene como conclusión fórmulas válidas se tendrá $\models \varphi$.

Paso de inducción: Si D_i ($i \in I \subseteq \mathbb{N}$) son derivaciones, y para todo $i \in I$, si Γ_i contiene las hipótesis de D_i $\left(\begin{smallmatrix}\varphi_i\\ \vdots\\ \psi\end{smallmatrix}\right)$ verifica $\Gamma_i \vdash \varphi_i$, vamos a probar que si D_i ($i \in I$) es una derivación entonces, si Φ contiene las hipótesis de D_i para todo $i \in I$, $\Phi \models \psi$. Para probar esto distinguimos casos según la regla de $\mathcal{PRC}$ que se haya aplicado a $\langle \varphi_i \mid i \in I \rangle$ para obtener ψ.

Probaremos en detalle algunos casos significativos.

(vE). Hipótesis de inducción: Todo conjunto Γ_1 que contenga las hipótesis de $\begin{smallmatrix}D_1\\ \varphi \vee \psi\end{smallmatrix}$ verifica $\Gamma_1 \vdash \varphi \vee \psi$; todo conjunto Γ_2 que contenga las hipótesis de $\begin{smallmatrix}D_2\\ \chi\end{smallmatrix}$ φ verifica $\Gamma_2 \vdash \chi$; todo conjunto Γ_3 que contenga las hipótesis de $\begin{smallmatrix}D_3\\ \chi\end{smallmatrix}$ ψ verifica $\Gamma_3 \vdash \chi$. Sea Φ conteniendo las hipótesis de

$$\dfrac{\begin{smallmatrix}D_1\\ \varphi \vee \psi\end{smallmatrix} \quad \begin{smallmatrix}[\varphi]\\ D_2\\ \chi\end{smallmatrix} \quad \begin{smallmatrix}[\psi]\\ D_3\\ \chi\end{smallmatrix}}{\chi}$$

y sea $\mathfrak{I}$ una interpretación tal que $\mathfrak{I} \models \Phi$. Evidentemente Φ contiene todas las hipótesis de $\begin{smallmatrix}D_1\\ \varphi \vee \psi\end{smallmatrix}$ por lo que se tiene $\mathfrak{I} \models \varphi \vee \psi$, es decir, $\mathfrak{I}, \eta \models \varphi$ ó $\mathfrak{I}, \eta \models \psi$ para todo $\eta \in AT_p^I$,

luego, por ser ambas fórmulas monomórficas tendremos $\mathfrak{I} \models \varphi$ ó $\mathfrak{I} \models \psi$. Por otro lado, $\Phi \cup \{\varphi\}$ contiene todas las hipótesis de $\underset{\chi}{D_2}$, y $\Phi \cup \{\psi\}$ contiene las de $\underset{\chi}{D_3}$ por lo que si $\mathfrak{I} \models \varphi$, como por hipótesis $\mathfrak{I} \models \Phi$, la inducción nos asegura que $\mathfrak{I} \models \chi$, en el caso de que $\mathfrak{I} \models \psi$ ocurrirá lo mismo, por lo que podemos concluir que $\Phi \models \chi$.

(VI) Hipótesis de inducción: Todo conjunto Γ que contenga las hipótesis de $\underset{\varphi[c:\tau/x:\tau]}{D}$ verifica $\Gamma \models \varphi[c:\tau/x:\tau]$ siendo c un símbolo de constante que no aparece en las hipótesis de D ni en φ. Sea Φ un conjunto que contiene las hipótesis de $\dfrac{\overset{D}{\varphi[c:\tau/x:\tau]}}{\forall x:\tau\ \varphi}$ y en el cual no aparece c; claramente Φ contiene las hipótesis de $\underset{\varphi[c:\tau/x:\tau]}{D}$. Si $\mathfrak{I}$ es una interpretación tal que $\mathfrak{I} \models \Phi$, por el lema de coincidencia se tiene que para toda asignación η y para todo $a \in [\tau]^{\mathfrak{I}}\eta$, $\mathfrak{I}\{a/c:\tau\},\eta \models \Phi$, ya que c no aparece en Φ. Entonces, por hipótesis de inducción, $\mathfrak{I}\{a/c:\tau\},\eta \models \varphi[c:\tau/x:\tau]$ para todo $a \in [\tau]^{\mathfrak{I}}\eta$ y para todo $\eta \in AT_p^{\mathfrak{I}}$, y por el lema de sustitución $\mathfrak{I}\{a/c:\tau\}\{a/x:\tau\},\eta \models \varphi$ para todo $a \in [\tau]^{\mathfrak{I}}\eta$ y para todo $\eta \in AT_p^{\mathfrak{I}}$. Como c no aparece en φ, aplicando de nuevo el lema de coincidencia obtenemos $\mathfrak{I}\{a/x:\tau\},\eta \models \varphi$ para todo $a \in [\tau]^{\mathfrak{I}}\eta$ y para todo $\eta \in AT_p^{\mathfrak{I}}$, luego $\mathfrak{I} \models \forall x:\tau\ \varphi$ lo que nos prueba $\Phi \models \forall x:\tau\ \varphi$.

(TSI) Hipótesis de inducción: Todo conjunto Γ que contenga las hipótesis de $\underset{\varphi[ct/\rho]}{D}$ verifica $\Gamma \models \varphi[ct/\rho]$ siendo ct una constante de tipo que no aparece en las hipótesis de D. Evidentemente, si Φ es un conjunto que contiene las hipótesis de $\dfrac{\overset{D}{\varphi[ct/\rho]}}{\varphi}$, Φ contiene las hipótesis de $\underset{\varphi[ct/\rho]}{D}$; podemos suponer sin pérdida de generalidad que ct no aparece en Φ. Si $\mathfrak{I}$ es una interpretación tal que $\mathfrak{I} \models \Phi$, por el lema de coincidencia se tiene $\mathfrak{I}\{D/ct\} \models \Phi$ para todo dominio $D \in T^{\mathfrak{I}}$, ya que ct no aparece en Φ, y por inducción $\mathfrak{I}\{D/ct\} \models \varphi[ct/\rho]$ para todo D de $T^{\mathfrak{I}}$ y por el lema de sustitución de tipos $\mathfrak{I}\{D/ct\},\eta\{ct/\rho\} \models \varphi$ para toda $\eta \in AT_p^{\mathfrak{I}}$ y todo $D \in T^{\mathfrak{I}}$ que es lo mismo que decir $\mathfrak{I},\eta\{D/\rho\} \models \varphi$ para todos $\eta \in AT_p^{\mathfrak{I}}$ y $D \in T^{\mathfrak{I}}$ y entonces, $\mathfrak{I} \models \varphi$ por lo que $\Phi \models \varphi$.

(DEF) Sea Φ un conjunto que contiene las hipótesis de la derivación $\dfrac{\begin{array}{c}D\\(\bot\ t:\tau_1):\tau_2=\bot:\tau_2\end{array}}{}$ y sea $\mathfrak{I}$ una interpretación tal que $\mathfrak{I} \models \Phi$. Puesto que $\mathfrak{I}\llbracket \bot:\tau_1\to\tau_2\rrbracket\eta\ (\mathfrak{I}\llbracket t:\tau_1\rrbracket\eta) = \bot_{\eta(\tau_2)} = \mathfrak{I}\llbracket \bot:\tau_2\rrbracket\eta$ para toda asignación η se tiene $\Phi \models (\bot\ t:\tau_1):\tau_2 = \bot:\tau_2$.

(SUB) Hipótesis de inducción: Todo conjunto Γ_1 que contenga las hipótesis de $\begin{array}{c}D_1\\t_1:\tau = t_2:\tau\end{array}$ verifica $\Gamma_1 \models t_1:\tau = t_2:\tau$ y todo conjunto Γ_2 que contenga las hipótesis de $\begin{array}{c}D_2\\\varphi[t_1:\tau/x:\tau]\end{array}$ verifica $\Gamma_2 \models \varphi[t_1:\tau/x:\tau]$.

Sea Φ conteniendo las hipótesis de $\dfrac{\begin{array}{cc}D_1 & D_2\\ t_1:\tau = t_2:\tau & \varphi[t_1:\tau/x:\tau]\end{array}}{\varphi[t_2:\tau/x:\tau]}$, entonces contiene las hipótesis de $\begin{array}{c}D_1\\t_1:\tau = t_2:\tau\end{array}$ y de $\begin{array}{c}D_2\\\varphi[t_1:\tau/x:\tau]\end{array}$ por lo que se verifica $\Phi \models t_1:\tau = t_2:\tau$ y $\Phi \models \varphi[t_1:\tau/x:\tau]$. Sea $\mathfrak{I}$ una interpretación tal que $\mathfrak{I} \models \Phi$ entonces, $\mathfrak{I}\llbracket t_1:\tau\rrbracket\eta = \mathfrak{I}\llbracket t_2:\tau\rrbracket\eta$ y $\mathfrak{I}\left\{\dfrac{\mathfrak{I}\llbracket t_1:\tau\rrbracket}{x:\tau}\right\}, \eta \models \varphi$ para toda $\eta \in AT_p^{\mathfrak{I}}$, por lo que $\mathfrak{I}\left\{\dfrac{\mathfrak{I}\llbracket t_2:\tau\rrbracket}{x:\tau}\right\}, \eta \models \varphi$ para toda $\eta \in AT_p^{\mathfrak{I}}$, y por el lema de sustitución $\mathfrak{I} \models \varphi[t_2:\tau/x:\tau]$. Por tanto, $\Phi \models \varphi[t_2:\tau/x:\tau]$.

(Tif) Hipótesis de inducción: Todo conjunto Φ que contiene las hipótesis de $\begin{array}{c}D\\t:bool=true\end{array}$ verifica $\Phi \models t:bool = true$. Este conjunto Φ contiene las hipótesis de $\dfrac{\begin{array}{c}D\\t:bool=true\end{array}}{(\text{if } t:bool \text{ then } t_1:\tau \text{ else } t_2:\tau)=t_1:\tau}$. Sea $\mathfrak{I}$ tal que $\mathfrak{I} \models \Phi$ entonces $\mathfrak{I}\llbracket t:bool\rrbracket = tt$ por hipótesis de inducción con lo que $\mathfrak{I} \models (\text{if } t:bool \text{ then } t_1:\tau \text{ else } t_2:\tau) = t_1:\tau$.

(Ctr) Hipótesis de inducción: Todo conjunto Φ que contiene las hipotesis de $\begin{array}{c}D\\(p\ \bot:\tau')\end{array}$ verifica $\Phi \models (p\ \bot:\tau')$. Por construcción Φ contiene las hipótesis de $\dfrac{\begin{array}{c}D\\(p\ \bot:\tau')\end{array}}{\varphi}$. Supongamos que no $\Phi \models \varphi$, esto quiere decir que existe una interpretación $\mathfrak{I}$ tal que $\mathfrak{I} \models \Phi$ y no $\mathfrak{I} \models \varphi$, si $\mathfrak{I} \models \Phi$ la hipótesis de inducción nos lleva a que si τ es el tipo más general de p y $\tau'=\tau\sigma$, entonces $\mathfrak{I}\llbracket \bot:\tau'\rrbracket\eta \in p:\tau^{\mathfrak{D}}\eta(\sigma)$, pero esto es absurdo porque por definición $p:\tau^{\mathfrak{D}}\eta$ no contiene el elemento *bottom* para ninguna asignación η. Por tanto, $\Phi \models \varphi$.

(REC) Hipótesis de inducción: Si $i<\omega$ y Γ_i contiene las hipótesis de

$\dfrac{D_1}{\neg((\mu X{:}\tau_1\to\tau_2.M)^1\ t{:}\tau_1)\subseteq\perp{:}\tau_2}$ entonces $\Gamma_1 \vdash \neg((\mu X{:}\tau_1\to\tau_2.M)^1\ t{:}\tau) \subseteq \perp{:}\tau_2$.

Supongamos que Φ contiene las hipótesis de $\dfrac{D_1}{\neg((\mu X{:}\tau_1\to\tau_2.M)^1 t{:}\tau_1)\subseteq\perp{:}\tau_2}$, aplicando inducción aseguramos que si $\mathfrak{I} \vdash \Phi$ entonces, para toda asignación de tipos η, $\mathfrak{I}[\![((\mu X{:}\tau_1\to\tau_2.M)^1\ t{:}\tau_1)]\!]\eta \neq \perp_{\eta(\tau_2)}$ por lo que necesariamente ha de ser $\mathfrak{I}[\![(\mu X{:}\tau_1\to\tau_2.M)^1]\!]\eta \neq \perp_{\eta(\tau_1\to\tau_2)}$. Bajo estas condiciones, puesto que la semántica del μ-operador coincide con el supremo de sus aproximaciones sintácticas y por las propiedades de los cpo's planos, $\mathfrak{I}[\![((\mu X{:}\tau_1\to\tau_2.M)\ t{:}\tau_1)]\!]\eta = \mathfrak{I}[\![((\mu X{:}\tau_1\to\tau_2.M)^1\ t{:}\tau_1)]\!]\eta$ para toda asignación η de $AT_p^{\mathfrak{I}}$. Podemos concluir, por tanto que $\Phi \vdash ((\mu X{:}\tau_1\to\tau_2.M)^1\ t{:}\tau_1) = ((\mu X{:}\tau_1\to\tau_2.M)\ t{:}\tau_1)$.

(ωAp) Hipótesis de inducción: Para cualquier $i<\omega$ si Γ_1 es un conjunto que contiene las hipótesis de $\dfrac{D_1}{(t_1{:}\tau)^1\subseteq t_2{:}\tau}$ entonces se verifica $\Gamma_1 \vdash (t_1{:}\tau)^1\subseteq t_2{:}\tau$. Sea Φ un conjunto que contiene las hipótesis de $\dfrac{D_1}{t_1{:}\tau \subseteq t_2{:}\tau}$ $(i<\omega)$, por construcción Φ contiene las hipótesis de $\dfrac{D_1}{(t_1{:}\tau)^1\subseteq t_2{:}\tau}$ para cualquier $i<\omega$ luego, por hipótesis de inducción, se verifica $\Phi \vdash (t_1{:}\tau)^1\subseteq t_2{:}\tau$ para todo $i<\omega$. Sea $\mathfrak{I}$ una interpretación tal que $\mathfrak{I} \vdash \Phi$ entonces $\mathfrak{I} \vdash (t_1{:}\tau)^1\subseteq t_2{:}\tau$ para todo $i<\omega$. Del hecho de que $\sqcup_{i=0}^{\omega}\mathfrak{I}[\![(t_1{:}\tau)^1]\!]\eta = \mathfrak{I}[\![t_1{:}\tau]\!]\eta$ para toda $\eta \in AT_p^{\mathfrak{I}}$, y de las propiedades de los cpo's planos deducimos $\mathfrak{I} \vdash t_1{:}\tau \subseteq t_2{:}\tau$, con lo que podemos asegurar que $\Phi \vdash t_1{:}\tau \subseteq t_2{:}\tau$.

En el resto de los casos, al igual que en estos ejemplos, la semántica de PLPR nos lleva a probar la corrección de la derivación para cada regla aplicada y por tanto la corrección del cálculo. ■

Como ya hemos dicho la prueba de la completitud de $\mathcal{PRC}$ se basa en la comprobación de las condiciones de completitud obtenidas a partir de los tableaux. Demostramos para ello algunos resultados previos.

Lema 1.3.2

Para todo conjunto de fórmulas $\Phi \subseteq L(\Sigma)$, $\mathcal{PRC}$ satisface:

a) Si para algún $i<\omega$ se verifica $\Phi \vdash \neg\alpha_i^*$ entonces $\Phi \vdash \neg\alpha^*$.

b) Si para todo $i<\omega$ se verifica $\Phi \vdash \neg\beta_i^*$ entonces $\Phi \vdash \neg\beta^*$.

Demostración:

a) Si α^e es de la forma $t_1:\nu \subseteq t_2:\nu$ con $t_1:\nu$ recursivo, aplicando la regla ($\neg$Apr) para cada $i<\omega$, se obtiene el resultado esperado.

Si α^e es de la forma $\neg t_1:\nu \subseteq t_2:\nu$ con $t_2:\nu$ recursivo, sabemos que para cualquier $i<\omega$, si $\Phi \vdash \neg\neg t_1:\nu \subseteq (t_2:\nu)^i$, existe una derivación de $\neg\neg t_1:\nu \subseteq (t_2:\nu)^i$ con todas sus hipótesis en Φ. Podemos utilizar la derivación de abajo para demostrar que $\Phi \vdash \neg\alpha^e$. En ella y en lo sucesivo, escribiremos la palabra hipótesis como justificación de un paso, cuando suponemos que existe una derivación con todas sus hipótesis en el conjunto de fórmulas que aparece en la parte superior de este paso y como conclusión la fórmula que aparece en la parte inferior.

$$\dfrac{\dfrac{\dfrac{\Phi}{\neg\neg t_1:\nu \subseteq (t_2:\nu)^i}\,(\text{hipótesis})}{t_1:\nu \subseteq (t_2:\nu)^i}\,(\neg\neg E)}{t_1:\nu \subseteq t_2:\nu}\,(\text{Apr})}{\neg\neg t_1:\nu \subseteq t_2:\nu}\,(\neg\neg I)$$

Si α^e es de la forma $\neg(p\ t:\nu)$ con $t:\nu$ recursivo, la prueba es similar utilizando la regla (PAp).

b) Si β^e es de la forma $t_1:\nu \subseteq t_2:\nu$ con $t_2:\nu$ recursivo y $\Phi \vdash \neg\beta_i^e$ para todo $i<\omega$, $\Phi \vdash \neg\beta^e$ es una consecuencia del siguiente esquema:

$$\dfrac{\dfrac{\Phi}{\neg t_1:\nu \subseteq (t_2:\nu)^0}\ \dfrac{\Phi}{\neg t_1:\nu \subseteq (t_2:\nu)^1}\ldots\dfrac{\Phi}{\neg t_1:\nu \subseteq (t_2:\nu)^n}\ldots(\text{hipótesis})}{\neg t_1:\nu \subseteq t_2:\nu}\,(\neg\omega\text{Ap})$$

Si β^e es de la forma $\neg t_1:\nu \subseteq t_2:\nu$ o $(p\ t:\nu)$, usando las reglas (ωAp) y ($\neg$PAp), respectivamente, junto con ($\neg\neg$E) y ($\neg\neg$I) en el primer caso, se obtiene el resultado esperado. ∎

Lema 1.3.3

Sea $\Phi \subseteq L(\Sigma)$. $\mathcal{PRC}$ verifica que para cualesquiera φ, ψ de $L(\Sigma)$.

$\Phi \cup \{\varphi\} \vdash \psi \Leftrightarrow \Phi \cup \{\neg\psi\} \vdash \neg\varphi$.

Demostración:

$\Rightarrow$) Supongamos que $\Phi \cup \{\varphi\} \vdash \psi$, es decir, existe una derivación de ψ con todas sus hipótesis en $\Phi \cup \{\varphi\}$. Entonces el siguiente esquema es una derivación de $\neg\varphi$ con todas sus hipótesis en $\Phi \cup \{\neg\psi\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[\varphi\sigma]}{\psi}\ (TSI) \qquad \Phi}{\psi \qquad \neg\psi}\ (hipótesis)}{\neg\psi \wedge \psi}\ (\wedge I)}{\neg\varphi\sigma}\ (\neg I)}{\neg\varphi}\ (TSI)$$

$\varphi\sigma$ monomórf. y los símbolos de σ nuevos

$\Leftarrow$) Supongamos que $\Phi \cup \{\neg\psi\} \vdash \neg\varphi$. Lo que sigue es una derivación de ψ con todas sus hipótesis no canceladas en $\Phi \cup \{\varphi\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[\neg\psi\sigma]}{\neg\psi}\ (TSI) \qquad \Phi}{\neg\varphi \qquad \varphi}\ (hipótesis)}{\neg\varphi \wedge \varphi}\ (\wedge I)}{\psi\sigma}\ (\neg E)}{\psi}\ (TSI)$$

$\neg\psi\sigma$ monomórf. y los símbolos de σ nuevos

con lo que se prueba $\Phi \cup \{\varphi\} \vdash \psi$. ∎

Teorema 1.3.4 (Completitud de $\mathcal{PRE}$)

Para todo $\Phi \subseteq L(\Sigma)$ y toda $\varphi \in L(\Sigma)$ se verifica $\Phi \models \varphi \Rightarrow \Phi \vdash \varphi$.

Demostración:

Vamos a probar que $\mathcal{PRE}$ satisfaces las condiciones (0) a (12) del teorema §II-2.3.2.

Las condiciones (0) y (1) son inmediatas por la propia definición de derivación.

Para demostrar la condición (2) estudiamos cada caso:

Si $\alpha = \varphi \wedge \psi$ y $\Phi \cup \{\varphi,\psi\} \vdash \chi$, la siguiente derivación sirve para demostrar $\Phi \cup \{\varphi \wedge \psi\} \vdash \chi$.

$$\dfrac{\dfrac{\varphi \wedge \psi}{\varphi}\ (\wedge E) \qquad \dfrac{\varphi \wedge \psi}{\psi}\ (\wedge E) \qquad \Phi}{\chi}\ (hipótesis)$$

Si $\alpha = \neg(\varphi \vee \psi)$ y $\Phi \cup \{\neg\varphi, \neg\psi\} \vdash \chi$, utilizamos la siguiente derivación para probar $\Phi \cup \{\neg(\varphi \vee \psi)\} \vdash \chi$.

$$\dfrac{\dfrac{\dfrac{\neg(\varphi \vee \psi)}{\neg\varphi \wedge \neg\psi}\,(DM1)}{\neg\varphi}\,(\wedge E) \qquad \dfrac{\dfrac{\dfrac{\neg(\varphi \vee \psi)}{\neg\varphi \wedge \neg\psi}\,(DM1)}{\neg\psi}\,(\wedge E) \qquad \Phi}{\chi}\,(\text{hipótesis})$$

Si $\alpha = \neg\neg\varphi$ la condición se obtiene de la regla $(\neg\neg E)$.

La condición (3) se prueba de un modo similar a (2) usando (vE) y (DM2). Veamos el caso $\beta = \neg(\varphi \wedge \psi)$. Si $\Phi \cup \{\neg\varphi\} \vdash \chi$ y $\Phi \cup \{\neg\psi\} \vdash \chi$ tenemos:

$$\dfrac{\dfrac{\neg(\varphi \wedge \psi)}{\neg\varphi \vee \neg\psi}\,(DM2) \qquad \dfrac{[\neg\varphi]\ \Phi}{\chi}\,(\text{hipót.}) \qquad \dfrac{[\neg\psi]\ \Phi}{\chi}\,(\text{hipót.})}{\chi}\,(vE)$$

y por tanto, $\Phi \cup \{\beta\} \vdash \chi$.

Para demostrar la condición (4), supongamos que $\Phi \cup \{\gamma(t:\nu)\} \vdash \varphi$.
Si γ es de la forma $\forall x:\nu\ \psi$, de γ se deriva $\psi[t:\nu/x:\nu]$ gracias a (VE), entonces usando la hipótesis de partida se tiene $\Phi \cup \{\gamma\} \vdash \varphi$.
Si $\gamma = \neg\exists x:\nu\ \psi$, para probar $\Phi \cup \{\gamma\} \vdash \varphi$ utilizamos la derivación:

$$\dfrac{\dfrac{\neg\exists x:\nu\ \psi \qquad \dfrac{[\psi[t:\nu/x:\nu]]}{\exists x:\nu\ \psi}\,(\exists I)}{\neg\psi[t:\nu/x:\nu] \wedge \exists x:\nu\ \psi}\,(\wedge I) \qquad \Phi}{\varphi}\,(\neg I)\ (\text{hipótesis})$$

La condición (5) se demuestra de una forma similar al caso (4) por medio de las reglas (∃E) y (∀I). Por ejemplo si $\delta = \exists x:\nu\ \psi$ y se verifica $\Phi \cup \{\delta(c:\nu)\} \vdash \varphi$, utilizamos la siguiente derivación.

$$\dfrac{\dfrac{\Phi \qquad [\ \psi[c:\nu/x:\nu]\ ]}{\varphi}\,(\text{hipótesis}) \qquad \exists x:\nu\ \psi}{\varphi}\,(\exists E)$$

Para probar (6) probamos que $\Phi \cup \{\alpha_i{}^\circ \mid i<\omega\} \vdash \varphi$ implica que para cualquier conjunto $I \subseteq \mathbb{N}$, $\Phi \cup \{\alpha_i{}^\circ \mid i \in \mathbb{N}/I\} \cup \{\alpha^\circ\} \vdash \varphi$. Entonces, para $I = \mathbb{N}$ se tendrá el resultado esperado. Lo probamos por inducción en la longitud de I:

Si I = ∅ es evidente, si I tiene un sólo elemento podemos suponer sin pérdida de generalidad que I = {1}. Supongamos $\Phi \cup \{\alpha_1^\circ \mid 1<\omega\} \vdash \varphi$ entonces el lema 1.3.3 nos asegura que $\Phi \cup \{\alpha_1^\circ \mid 1<1<\omega\} \cup \{\neg\varphi\} \vdash \neg\alpha_1^\circ$ y por el lema 1.3.2-a), $\Phi \cup \{\alpha_1^\circ \mid 1<i<\omega\} \cup \{\neg\varphi\} \vdash \neg\alpha^\circ$ de donde podemos deducir $\Phi \cup \{\alpha_1^\circ \mid 1<i<\omega\} \cup \{\alpha^\circ\} \vdash \varphi$ utilizando otra vez 1.3.3.

Paso de inducción: Suponemos que si $\Phi \cup \{\alpha_1^\circ \mid 1<\omega\} \vdash \varphi$, se verifica $\Phi \cup \{\alpha_1^\circ \mid i \in N/I\} \cup \{\alpha^\circ\} \vdash \varphi$ para todo I de longitud n, y lo probamos para cualquier I de longitud n+1. Para cualquier $k \in N/I$, el lema 1.3.3 nos asegura que si se verifica $\Phi \cup \{\alpha_1^\circ \mid i \in N/I\} \cup \{\alpha^\circ\} \vdash \varphi$ entonces, $\Phi \cup \{\alpha_1^\circ \mid i \in N/I,\ i \neq k\} \cup \{\alpha^\circ\} \cup \{\neg\varphi\} \vdash \neg\alpha\alpha^\circ$. Aplicando el resultado 1.3.2-a) tenemos $\Phi \cup \{\alpha_1^\circ \mid i \in N/I,\ i \neq k\} \cup \{\alpha^\circ\} \cup \{\neg\varphi\} \vdash \neg\alpha^\circ$, y por 1.3.3 deducimos $\Phi \cup \{\alpha_1^\circ \mid i \in N/I,\ i \neq k\} \cup \{\alpha^\circ\} \vdash \varphi$ para cualquier $k<\omega$, de donde concluimos el resultado esperado.

La condición (7) se prueba usando el siguiente razonamiento: Supongamos $\Phi \cup \{\beta_i^\circ\} \vdash \varphi$ para todo $i<\omega$. Aplicando el resultado 1.3.3 obtenemos $\Phi \cup \{\neg\varphi\} \vdash \neg\beta_i^\circ$ para todo $i<\omega$. Entonces el lema 1.3.2-b) nos asegura que $\Phi \cup \{\neg\varphi\} \vdash \neg\beta^\circ$ y deducimos $\Phi \cup \{\beta^\circ\} \vdash \varphi$ por 1.3.3.

La condición (8) es una consecuencia de las reglas del orden parcial. Veamos algunos casos.

- $\Phi \cup \{\forall x{:}\nu\ \forall y{:}\nu\ \forall z{:}\nu\ ((x \leq y \wedge y \leq z) \rightarrow x \leq z)\} \vdash \varphi \Rightarrow \Phi \vdash \varphi$. En esta demostración suprimimos la escritura de los tipos para simplificar la notación. las constantes auxiliares c_1, c_2, c_3 son de tipo ν.

$$\dfrac{\dfrac{\dfrac{\dfrac{[c_1 \leq c_2 \wedge c_2 \leq c_3]}{c_1 \leq c_2}(\wedge E)}{c_1 = c_2 \vee c_1 = \perp}(FLAT) \quad \dfrac{\dfrac{[c_1 \leq c_2 \wedge c_2 \leq c_3]}{c_2 \leq c_3}(\wedge E) \quad [c_1 = c_2]}{c_1 \leq c_3}(SUB) \quad \dfrac{[c_1 = \perp] \quad \dfrac{}{\perp \leq c_3}(BOT)}{c_1 \leq c_3}(SUB)}{c_1 \leq c_3}(\vee E)}{(c_1 \leq c_2 \wedge c_2 \leq c_3) \rightarrow c_1 \leq c_3}(FTh)$$
$$\dfrac{}{\forall x\ (x \leq c_2 \wedge c_2 \leq c_3) \rightarrow x \leq c_3}(\forall I)$$
$$\dfrac{}{\forall x\ \forall y\ (x \leq y \wedge y \leq c_3) \rightarrow x \leq c_3}(\forall I)$$
$$\dfrac{}{\forall x\ \forall y\ \forall z (x \leq y \wedge y \leq z) \rightarrow x \leq z}(\forall I)$$

- $\Phi \cup \{\forall x{:}\nu_1\ \forall y{:}\nu_1 (x = y \rightarrow (M\ x){:}\nu_2 = (M\ y){:}\nu_2)\} \vdash \varphi \Rightarrow \Phi \vdash \varphi$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(M\ c_1{:}\nu_1){:}\nu_2 = (M\ c_1{:}\nu_1){:}\nu_2}\ (=REF)}{[c_1{:}\nu_1 = c_2{:}\nu_1]\quad (M\ c_1{:}\nu_1){:}\nu_2 = (M\ c_2{:}\nu_1){:}\nu_2}\ (SUB)}{\neg c_1{:}\nu_1 = c_2{:}\nu_1 \vee (M\ c_1{:}\nu_1){:}\nu_2 = (M\ c_2{:}\nu_1){:}\nu_2}\ (FTh)}{\forall x{:}\nu_1\ (x = c_2{:}\nu_1 \rightarrow (M\ x){:}\nu_2 = (M\ c_2{:}\nu_1){:}\nu_2}\ (\forall I)}{\forall x{:}\nu_1\ \forall y{:}\nu_1\ (x = y \rightarrow (M\ x){:}\nu_2 = (M\ y){:}\nu_2}\ (\forall I)\qquad \Phi}{\varphi}\ (\text{hipótes.})$$

La condición (9), $\Phi \cup \{\varphi\sigma\} \vdash \psi \Rightarrow \Phi \cup \{\varphi\} \vdash \psi$, se obtiene directamente de la regla (TSE); mientras que la condición (12), $\Phi \vdash \varphi\sigma$ (con los símbolos de σ nuevos) implica $\Phi \vdash \varphi$, es consecuencia de (TSI). Efectivamente:

$$\dfrac{\dfrac{\varphi}{\varphi\sigma}\ (TSE)\qquad \Phi}{\psi}\ (\text{hipótesis})$$

$$\dfrac{\Phi}{\varphi\sigma}\ (\text{hipótesis})\qquad \dfrac{\varphi\sigma}{\varphi}\ (TSI)\ \text{por las caracterist. de } \sigma$$

Supuesto que $\Phi \cup \{\neg\varphi\} \vdash \psi$ y $\Phi \cup \{\neg\varphi\} \vdash \neg\psi$ con φ monomórfica, el siguiente esquema sirve para probar $\Phi \vdash \varphi$ que justifica la condición (10).

$$\dfrac{\dfrac{\dfrac{\Phi\quad [\neg\varphi]}{\psi}\ (\text{hipót.})\qquad \dfrac{\Phi\quad [\neg\varphi]}{\neg\psi}\ (\text{hipót.})}{\neg\psi \wedge \psi}\ (\wedge I)}{\varphi}\ (\neg E)$$

Para probar (11) vemos que de acuerdo con la regla (Ctr), teniendo como hipótesis $(p\ \perp{:}\tau)$ podemos derivar φ, entonces existirá una derivación de φ con todas sus hipótesis en $\Phi \cup \{(p\ \perp{:}\tau)\}$ para cualquier Φ, con lo que $\Phi \cup \{(p\ \perp{:}\tau)\} \vdash \varphi$. ∎

2. COMPLEJIDAD DEL PROBLEMA DE VALIDEZ EN LA LOGICA PLPR

Como se ha indicado, PLPR no admite cálculos completos finitarios. De una manera informal, podemos decir que esta afirmación puede deducitrse a partir del hecho de ser la aritmética estándar expresable en PLPR. En esta sección demostraremos que el problema de validez en PLPR es Π_1^1-completo. Para ello, en el primer apartado, probaremos que la satisfactibilidad es un problema Σ_1^1-duro, y en el segundo lo clasificamos en la clase Σ_1^1. Para la primera demostración, se reduce un problema de dominó recursivo al problema de satisfactibilidad, y en la segunda se utiliza el teorema de Löwenheim-Skolem que será enunciado y demostrado para nuestra lógica.

2.1 REDUCCION DE LOS DOMINOS A LA SATISFACTIBILIDAD

Un dominó es un cuadrado de dimensión 1 x 1, con orientación fija y con sus cuatro aristas coloreadas. Se plantea el siguiente problema de decisión, conocido como problema de dominó: ¿es posible recubrir una porción P del plano Z x Z utilizando dominós de ciertos tipos y cumpliendo una serie de restricciones?

Desde que Hao Wang introdujo los problemas de dominó, se han desarrollado muchas versiones de dominós, como los acotados, los no acotados y los recursivos. La indecibilidad de estos problemas se prueba generalmente estableciendo una correspondencia entre la representación de un dominó y una máquina de Turing, correspondencia que permite reducir los dominós a problemas tales como el problema de parada.

La principal utilidad de los dominós consiste en permitir clasificar el problema de satisfactibilidad o validez de diferentes lógicas usando técnicas de reducción de problemas. En [Lew-78] y [Emd-83] se usan los dominós acotados para demostrar que la satisfactibilidad en la lógica proposicional es un problema NP-completo. Los dominós no acotados sirven para demostrar que la satisfactibilidad de la lógica de predicados es un problema Σ_1^0-completo, este empleo se encuentra, por ejemplo, en [Emd-83], [LP-81] y [Wan-63]. Los dominós recursivos que presenta Harel, [Har-83] y [Har-84], sirven para probar que la satisfactibilidad de la

lógica infinitaria constructiva es Σ_1^1-completa; en el primero de ellos se hace énfasis en la reducción de dominós a lógicas orientadas a programas como por ejemplo QDL y QTL.

A continuación formalizaremos un problema concreto de dominó recursivo propuesto por Harel.

Sea $T = \{d_0, \ldots, d_m\}$ un conjunto de tipos de dominó donde cada d_i viene caracterizado por una cuadrupla $\langle R_i, L_i, U_i, D_i \rangle$ que indica los colores de las cuatro aristas (*Right*, *Left*, *Up*, *Down*) de d_i. Estos colores pertenecen a un conjunto C_T de k elementos $C_T = \{c_0, \ldots, c_{k-1}\}$. La porción P a recubrir en nuestro problema es $\mathbb{N} \times \mathbb{N}$.

Definición 2.1.1

Un *recubrimiento* F *de* P *con dominós de tipo* T es una aplicación de P en T.

Un *recubrimiento* F de P es T-*coherente* cuando verifica:

a) Para cada par de casillas de P contiguas en la misma vertical u horizontal, los dominós que las recubren tienen las aristas adyacentes del mismo color.

b) En la primera columna de P aparecen infinitos dominós del tipo d_0.

Un *conjunto* $T = \{d_0, \ldots, d_m\}$ se llama *coherente* cuando existe un recubrimiento F de P con dominós de tipo T tal que F es T-coherente. □

Nos planteamos el siguiente problema de decisión:

Dado un conjunto de tipos de dominó $T = \{d_0, \ldots, d_m\}$ con colores en $C_T = \{c_0, \ldots, c_{k-1}\}$ (k una potencia de 2) y $P = \mathbb{N} \times \mathbb{N}$. ¿Es el conjunto T coherente?

En [Har-84] se demuestra que este problema de dominó es Σ_1^1-completo y en [Har-83] se reduce al problema de satisfactibilidad de la lógica de primer orden extendida con la aritmética estándar de los naturales.

Basándonos en estos resultados, vamos a demostrar que el problema de satisfactibilidad en PLPR es Σ_1^1-duro reduciendo el dominó planteado al problema de satisfactibilidad en PLPR; esto es, para cada conjunto de tipos de dominó construimos, de forma efectiva, una fórmula φ_T tal que T es coherente $\Leftrightarrow$ φ_T es satisfactible.

Consideremos la siguiente signatura $\Sigma_T = \langle \Sigma_t, \Sigma_d \rangle$ con:

Σ_t = {nat/0, color/0} y

Σ_d = {do: $\rightarrow$color×color×color×color, ..., d_m: $\rightarrow$color×color×color×color, 0: $\rightarrow$nat, suc: nat$\rightarrow$nat, pred: nat$\rightarrow$nat, es_cero: nat$\rightarrow$bool, less: nat×nat RIGHT: nat×nat$\rightarrow$color, LEFT: nat×nat$\rightarrow$color, UP: nat×nat$\rightarrow$color, DOWN: nat×nat$\rightarrow$color, RLUD: nat×nat$\rightarrow$color×color×color×color, }.

Construimos φ_T como la conjunción de las fórmulas de la figura 1.

(1) $\forall x$:nat $\forall y$:nat((suc x) = (suc y) $\rightarrow$ x = y)

(2) $\forall x$:nat $\neg$(suc x) = 0

(3) (es_cero 0) = true

(4) $\forall x$:nat($\neg x \leq \perp$:nat $\rightarrow$(es_cero (suc x)) = false)

(5) $\forall x$:nat(pred (suc x)) = x

(6) (pred 0) = $\perp$:nat

(7) $\forall y$:nat((μX:nat$\rightarrow$nat. (λx:nat. (if (es_cero x) then 0 else (suc (X (pred x)))))) y) = y

(8) $\forall x$:nat $\neg$(less (x,0))

(9) $\forall x$:nat (less (0,(suc x)))

(10) $\forall x$:nat $\forall y$:nat ((less ((suc x),(suc y))) $\leftrightarrow$ (less (x,y)))

(11) $\forall x$:nat $\forall y$:nat (($\neg x \leq \perp$:nat $\wedge$ $\neg y \leq \perp$:nat) $\rightarrow$ ((RLUD (x,y)) = do $\vee$...$\vee$ (RLUD (x,y)) = d_m))

(12) $\forall x$:nat $\forall x$:nat (RLUD (x,y)) = ((RIGHT (x,y)),(LEFT (x,y)),(UP (x,y)),(DOWN (x,y)))

(13) $\forall x$:nat $\forall y$:nat (($\neg x \leq \perp$:nat $\wedge$ $\neg y \leq \perp$:nat) $\rightarrow$ ((RIGHT (x,y))=(LEFT ((suc x),y))$\wedge$(UP (x,y))=(DOWN (x,(suc y)))))

(14) $\forall x$:nat ($\neg x \leq \perp$:nat $\rightarrow$ $\exists y$:nat ((less (x,y))$\wedge$(RLUD (0,y)) = do))

Figura 1

Lema 2.1.2

Si T = {do,...,d_m} es coherente entonces φ_T es satisfactible.

Demostración:

Por ser T coherente existe un recubrimiento F para P que es T-coherente. A partir de F se define una Σ_T-estructura de tipos I y una Σ_T-estructura de datos $\mathcal{D}$, que darán lugar a un modelo de φ_T.

Interpretamos las constantes de tipo de la siguiente manera:

$nat^I = N \cup \perp_N = N_\perp$ y consideramos el orden plano $\leq_N$ en N con $\perp_N$ como su mínimo elemento.

$color^I = \{c_0, \ldots, c_{k-1}, \perp_{color}\} = C_{T_\perp}$, también considerado como un cpo plano con el orden $\leq_{color}$.

La familia de domonios $T^{\mathcal{I}}$ es el cierre para el producto estricto creada a partir de $color^{\mathcal{I}}$ $nat^{\mathcal{I}}$ y B

Las constantes de dato se interpretan como sigue:

$0: \rightarrow nat^{\mathcal{D}} = 0 \in \mathbb{N}$

$d_i: \rightarrow color \times color \times color \times color^{\mathcal{D}} = d_i = \langle R_i, L_i, U_i, D_i \rangle \in C_{T_\perp} \otimes C_{T_\perp} \otimes C_{T_\perp} \otimes C_{T_\perp}$

Definimos las siguientes funciones estrictas:

$suc: nat \rightarrow nat^{\mathcal{D}}$ (n) = n + 1 si $n \in \mathbb{N}$ (sucesor de n)

$pred: nat \rightarrow nat^{\mathcal{D}}$ (n) = n - 1 si $n \in \mathbb{N} \setminus \{0\}$ (predecesor de n)

$pred: nat \rightarrow nat^{\mathcal{D}}$ (0) = $\perp_{\mathbb{N}}$

$RLUD: nat \times nat \rightarrow color \times color \times color \times color^{\mathcal{D}}$ (n,m) = $d_i = \langle R_i, L_i, U_i, D_i \rangle$ si y solo si $F(n,m) = d_i$ y $\langle n,m \rangle \in \mathbb{N} \times \mathbb{N}$

$RIGHT: nat \times nat \rightarrow color^{\mathcal{D}}$ (n,m) = R_i sii $F(n,m) = d_i$ y $\langle n,m \rangle \in \mathbb{N} \times \mathbb{N}$

$LEFT: nat \times nat \rightarrow color^{\mathcal{D}}$ (n,m) = L_i sii $F(n,m) = d_i$ y $\langle n,m \rangle \in \mathbb{N} \times \mathbb{N}$

$UP: nat \times nat \rightarrow color^{\mathcal{D}}$ (n,m) = U_i sii $F(n,m) = d_i$ y $\langle n,m \rangle \in \mathbb{N} \times \mathbb{N}$

$DOWN: nat \times nat \rightarrow color^{\mathcal{D}}$ (n,m) = D_i sii $F(n,m) = d_i$ y $\langle n,m \rangle \in \mathbb{N} \times \mathbb{N}$

$es\_cero: nat \rightarrow bool^{\mathcal{D}}(0) = tt$, $es\_cero: nat \rightarrow bool^{\mathcal{D}}(n) = ff$ para n pertenciente a $\mathbb{N} \setminus \{0\}$.

Interpretamos el predicado less como el menor estándar entre los números naturales.

$less: nat \times nat^{\mathcal{D}} = <$

Simplificamos la notación eliminando los tipos en la interpretación de los símbolos de dato, así por ejemplo, tendremos $suc^{\mathcal{D}}$, $0^{\mathcal{D}}$, etc.

Consideremos la Σ_T-estructura $\mathcal{A} = \langle \mathcal{I}, \mathcal{D} \rangle$. Puesto que φ_T no tiene variables libres y es monomórfica, basta con probar $\mathcal{A} \models \varphi_T$ para poder afirmar que φ_T es satisfactible. Por construcción de φ_T, $\mathcal{A} \models \varphi_T$ es equivalente a decir que $\mathcal{A}$ satisface cada una de las fórmulas (1) a (14) de la figura 1, esto es cierto como probamos a continuación.

(1) Para cualesquiera $n, m \in \mathbb{N}$ si $n + 1 = m + 1$ entonces $n = m$ y puesto que si $\perp_{\mathbb{N}} + 1 = n + 1$, entonces, $n = \perp_{\mathbb{N}}$, se concluye que para todos $n, m \in nat^{\mathcal{I}}$, $suc^{\mathcal{D}}(n) = suc^{\mathcal{D}}(m) \Rightarrow n = m$, y esto es equivalente a decir $\mathcal{A} \models$ (1).

(2) $\mathcal{A} \models$ (2) porque ningún elemento de $\mathbb{N}_\perp$ tiene como sucesor el cero y $0^{\mathcal{D}} = 0$.

(3) Esta fórmula se satisface por definición de $es\_cero^{\mathcal{D}}$ y de $0^{\mathcal{D}}$.

(4) $\mathcal{A} \models$ (4), por la misma razón que el caso anterior, ya que ningún elemento de $\mathbb{N}_\perp$ distinto de $\perp_{\mathbb{N}}$ verifica que su sucesor es cero.

(5) Sabemos que si $n \in \mathbb{N}$, se tiene $(n + 1) - 1 = n$ y si $n = \perp_{\mathbb{N}}$, como $pred^{\mathcal{D}}$ y $suc^{\mathcal{D}}$ son funciones estrictas, $(\perp_{\mathbb{N}} + 1) - 1 = \perp_{\mathbb{N}}$. Luego para todo $n \in nat^{\mathcal{I}}$, $pred^{\mathcal{D}}(suc^{\mathcal{D}}(n)) = n$ es decir $\mathcal{I} \models (5)$.

(6) $\mathcal{I} \models (6)$ por definicón de $pred^{\mathcal{D}}$ y de $0^{\mathcal{D}}$.

(7) Para probar $\mathcal{I} \models (7)$, probamos que para cualquier valoración ξ con respecto a $\mathcal{I}$ y culaquier $\eta \in AT_p^{\mathcal{I}}$, si $M: nat \to nat$ es la expresión funcional $(\lambda x: nat.(if\ (es\_cero\ x)\ then\ 0\ else\ (suc\ (X\ (pred\ x)))))$, $\mathcal{I}$ es el par $\langle \mathbb{N}, \xi \rangle$, y h es el menor punto fijo del operador $T(\mathcal{I}, X: nat \to nat, M: nat \to nat, \eta)$, entonces, $h(n) = n$ para todo $n \in nat^{\mathcal{I}}$, es decir, h es la identidad en $\mathbb{N}_\perp$. Efectivamente, por definición de T, h es la menor función $f: [nat^{\mathcal{I}} \xrightarrow[e]{} nat^{\mathcal{I}}]$ que verifica $\mathcal{I}\left(\frac{f}{X: nat \to nat}\right)[M: nat \to nat]\eta\ (n) = f(n)$ para todo $n \in \mathbb{N}_\perp$. Es decir, h es la menor f que para todo $n \in \mathbb{N}_\perp$ satisface:

(i) Si $es\_cero^{\mathcal{D}}(n) = tt$ entonces $f(n) = 0^{\mathcal{D}} = 0$.

(ii) Si $es\_cero^{\mathcal{D}}(n) = ff$ entonces $f(n) = suc^{\mathcal{D}}(f(pred^{\mathcal{D}}(n)))$.

(iii) Si $es\_cero^{\mathcal{D}}(n) = \perp_{bool}$ entonces $f(n) = \perp_{\mathbb{N}}$.

Por construcción de $\mathcal{D}$, podemos asegurar que si f verifica (i), (ii) y (iii), entonces, $f(\perp_{\mathbb{N}}) = \perp_{\mathbb{N}}$, $f(0) = 0$ y si $n > 0$, $f(n-1) + 1 = n$. La menor función estricta de $\mathbb{N}_\perp$ en $\mathbb{N}_\perp$ que satisface esto es la identidad, por tanto, $h(n) = n$ para todo $n \in nat^{\mathcal{I}}$.

(8), (9) y (10) se satisfacen por la definición de $less^{\mathcal{D}}$ y de $0^{\mathcal{D}}$, y por las propiedades de los naturales. En concreto para ver $\mathcal{I} \models (10)$, sabemos que para todos $n, m \in \mathbb{N}$, $(n + 1) < (m + 1)$ si y solo si $n < m$ con lo que para todos n y $m \in \mathbb{N}$ $less^{\mathcal{D}}(suc^{\mathcal{D}}(n), suc^{\mathcal{D}}(m)) \leftrightarrow less^{\mathcal{D}}(n, m)$. Si n ó m son iguales a $\perp_{\mathbb{N}}$ puesto que $suc^{\mathcal{D}}$ es una función estricta, los dos lados de la equivalencia serán falsos, por tanto dicha equivalencia se verifica para todos $n, m \in nat^{\mathcal{I}}$.

(11) Por ser F un recubrimiento de $\mathbb{N} \times \mathbb{N}$, $F(n, m) \in T$ para todos $n, m \in \mathbb{N}$, y por la definición de $\mathcal{D}$, $F(n, m) = d_1 = \langle R_1, L_1, U_1, D_1 \rangle$ sii $RLUD^{\mathcal{D}}(n, m) = \langle R_1, L_1, U_1, D_1 \rangle = d_1^{\mathcal{D}}$. Podemos asegurar entonces que $RLUD^{\mathcal{D}}(n, m) = d_0^{\mathcal{D}}$, ó $RLUD^{\mathcal{D}}(n, m) = d_1^{\mathcal{D}}$, ó ... $RLUD^{\mathcal{D}}(n, m) = d_m^{\mathcal{D}}$ para todos $n, m \in \mathbb{N}$, o lo que es lo mismo $\mathcal{I} \models (11)$.

(12) $\mathcal{I}$ satisface la fórmula (12) porque por construcción de $\mathcal{D}$ se verifica $RLUD^{\mathcal{D}}(n, m) = \langle RIGHT^{\mathcal{D}}(n, m), LEFT^{\mathcal{D}}(n, m), UP^{\mathcal{D}}(n, m), DOWN^{\mathcal{D}}(n, m) \rangle$ para todos n y m de $\mathbb{N}$. Por tratarse de funciones estrictas y debido a que $nat^{\mathcal{I}} = \mathbb{N}_\perp$, la igualdad anterior se verifica para todos los elementos de $nat^{\mathcal{I}}$.

(13) Por ser F un recubrimiento de P T-coherente, se cumple la condición a) de la definición de recubrimiento T-coherente: si $F(n,m) = \langle R,L,U,D\rangle$, $F(n+1,m) = \langle R',L',U',D'\rangle$ y $F(n,m+1) = \langle R'',L'',U'',D''\rangle$, entonces $R = L'$ y $U = D''$ por lo que ha de verificarse que para todos $n, m \in nat^{\mathcal{I}}$, si n y m son distintos de $\perp_{\mathbb{N}}$, entonces $RIGHT^{\mathcal{D}}(n,m) = LEFT^{\mathcal{D}}(suc^{\mathcal{D}}(n),m)$ y $UP^{\mathcal{D}}(n,m) = DOWN^{\mathcal{D}}(n,suc^{\mathcal{D}}(m))$. Por tanto $\mathcal{H} \models$ (13).

(14) Sabemos que F debe cumplir la condición b) de la definición de recubrimiento T-coherente. Si en la primera columna de $\mathbb{N} \times \mathbb{N}$ hay infinitas casillas recubiertas por un dominó de tipo do, esto quiere decir que $F(0,m) = do$ para infinitos $m \in \mathbb{N}$ y por tanto dado un $n \in \mathbb{N}$ siempre existe un número natural $m > n$ tal que $F(0,m) = do$. Por la definición de $\mathcal{D}$ tendremos que para todo $n \in \mathbb{N}$, existe un $m \in \mathbb{N}_{\perp}$ tal que $less^{\mathcal{D}}(n,m)$ y $RLUD^{\mathcal{D}}(0^{\mathcal{D}},m) = do^{\mathcal{D}}$ y por tanto, $\mathcal{H} \models$ (14). ∎

Este lema hubiera sido igualmente válido si hubiéramos construido la fórmula φ_T sin (3), (4) y (7), consiguiendo así una fórmula de la lógica de predicados. Sin embargo, el recíproco no sería cierto ya que podríamos obtener modelos no estándar de los naturales y la fórmula (14) no representaría la condición de dominó recursivo.

Proposición 2.1.3

El modelo estándar de los números naturales es caracterizable en PLPR.

Demostración:

Sea $\Sigma_{\mathcal{M}} = \langle \Sigma_t, \Sigma_d\rangle$ con $\Sigma_t = \{nat/0\}$, $\Sigma_d = \{0:\rightarrow nat,\ suc: nat \rightarrow nat,\ pred: nat \rightarrow nat,\ es\_cero: nat \rightarrow bool,\ less: nat \times nat\}$. Sea $\mathcal{N} = \langle\langle T^{\mathcal{N}}, \mathbb{N}_{\perp}\rangle, \{0, suc, pred, =0, <\}\rangle$ la estructura que representa a los números naturales con el orden habitual y con la función $=0$ para representar una función booleana que es cierta para $n=0$, siendo $T^{\mathcal{N}}$ la familia de dominios construida como el cierre para el producto estricto a partir de $\mathbb{N}_{\perp}$ y $\mathbb{B}$.

La demostración del lema anterior es una prueba de que $\mathcal{N}$ satisface las fórmulas (1) a (10) de la figura 1. Además para cualquier $\Sigma_{\mathcal{M}}$-estructura $\mathcal{H} = \langle \mathcal{I}, \mathcal{D}\rangle$ que satisfaga estas fórmulas existe un isomorfismo h entre $\mathcal{N}$ y $\mathcal{H}$. Vamos a comprobarlo.

Definimos la función h entre $\mathbb{N}_{\perp}$ y $nat^{\mathcal{I}}$ mediante:

$$h(n) := \underbrace{suc^{\mathcal{D}}(\ldots(suc^{\mathcal{D}}}_{\text{n veces}}(0^{\mathcal{D}}))\ldots);\ h(\perp_{\mathbb{N}}) = \perp_{nat}.$$

h es la identidad entre los booleanos. h es en efecto un homomorfismo puesto que preserva las operaciones de $\mathfrak{N}$:

- $h(0) = 0^{\mathfrak{D}}$ por definición de h.

- $h(suc(n)) = suc^{\mathfrak{D}}(\ldots(suc^{\mathfrak{D}}(0^{\mathfrak{D}}))\ldots)$ ($\llcorner$ n+1 veces $\lrcorner$) por definición de h y suc
  $= suc^{\mathfrak{D}}(h(n))$ por definición de h.

- Probamos $h(pred(n)) = pred^{\mathfrak{D}}(h(n))$ para el caso $\perp_{\mathbb{N}}$ primero, y por inducción en n para el resto.

  $h(pred(\perp_{\mathbb{N}})) = h(\perp_{\mathbb{N}}) = \perp_{nat} = pred^{\mathfrak{D}}(\perp_{nat}) = pred^{\mathfrak{D}}(h(\perp_{\mathbb{N}}))$

  $h(pred(0)) = h(\perp_{\mathbb{N}})$ pues el predecesor de 0 no está definido
  $= \perp_{nat}$ por definición de h
  $= pred^{\mathfrak{D}}(0^{\mathfrak{D}})$ pues $\mathfrak{A} \vdash$ (6)
  $= pred^{\mathfrak{D}}(h(0))$ por definición de h.

  $h(pred(suc(n))) = h(n)$ propiedades de pred y suc
  $= pred^{\mathfrak{D}}(suc^{\mathfrak{D}}(h(n)))$ pues $\mathfrak{A} \vdash$ (5)
  $= pred^{\mathfrak{D}}(h(suc(n)))$ pues h preserva el sucesor.

- $h(=0(\perp_{\mathbb{N}})) = \perp_{bool} = es\_cero^{\mathfrak{D}}(\perp_{nat}) = es\_cero^{\mathfrak{D}}(h(\perp_{\mathbb{N}}))$

  $h(=0(0)) = tt$ por definición de =0
  $= es\_cero^{\mathfrak{D}}(0^{\mathfrak{D}})$ ya que $\mathfrak{A} \vdash$ (3)
  $= es\_cero^{\mathfrak{D}}(h(0))$ por definición de h

  $h(=0(suc(n))) = ff$ si $n \neq \perp_{\mathbb{N}}$ por definición de =0
  $= es\_cero^{\mathfrak{D}}(suc^{\mathfrak{D}}(h(n)))$ si $h(n) \neq \perp_{\mathbb{N}}$ pues $\mathfrak{A} \vdash$ (4)
  $= es\_cero^{\mathfrak{D}}(h(suc(n)))$ si $n \neq \perp_{\mathbb{N}}$ pues h preserva el suc.

- Probamos $n < m \leftrightarrow less^{\mathfrak{D}}(h(n),h(m))$ utilizando inducción en n y m.

  $n < 0 \leftrightarrow less^{\mathfrak{D}}(h(n),0^{\mathfrak{D}})$ pues $\mathfrak{A} \vdash$ (8)
  $\leftrightarrow less^{\mathfrak{D}}(h(n),h(0))$ por definición de h

  $0 < suc(n) \leftrightarrow less^{\mathfrak{D}}(0^{\mathfrak{D}}, suc^{\mathfrak{D}}(\ldots(suc^{\mathfrak{D}}(0^{\mathfrak{D}}))\ldots)$ ($\llcorner$ n veces $\lrcorner$) pues $\mathfrak{A} \vdash$ (9)
  $\leftrightarrow less^{\mathfrak{D}}(h(0), h(n))$ por definición de h

  Supongamos que $n < m \leftrightarrow less^{\mathfrak{D}}(h(n),h(m))$, entonces:

  $suc(n) < suc(m) \leftrightarrow n < m$ propiedad del orden <
  $\leftrightarrow less^{\mathfrak{D}}(h(n),h(m))$ por hipótesis de inducción
  $\leftrightarrow less^{\mathfrak{D}}(suc^{\mathfrak{D}}(h(n)),suc^{\mathfrak{D}}(h(m)))$ pues $\mathfrak{A} \vdash$ (10)
  $\leftrightarrow less^{\mathfrak{D}}(h(suc(n)),h(suc(m)))$ pues h preserva el sucesor.

Una vez probado que h es un homomorfismo, falta demostrar que h

es biyectiva. Para comprobar que h es inyectiva, supongamos que h(n) = h(m) y que n ≠ m, entonces, $\underbrace{suc^{\mathfrak{D}}(..suc^{\mathfrak{D}}(0^{\mathfrak{D}})..)}_{\text{n veces}} = \underbrace{suc^{\mathfrak{D}}(..suc^{\mathfrak{D}}(0^{\mathfrak{D}})..)}_{\text{m veces}}$, si por ejemplo, n > m, por ser $\mathfrak{A}$ un modelo de (1) obtenemos $\underbrace{suc^{\mathfrak{D}}(...suc^{\mathfrak{D}}(0^{\mathfrak{D}})..)}_{\text{n-m veces}} = 0^{\mathfrak{D}}$ que es absurdo porque $\mathfrak{A} \vdash$ (2). Luego, n = m y h es inyectiva.

La prueba de que h es sobre se basa en que puesto que $\mathfrak{A}$ satisface la fórmula (7) es posible probar que todo elemento de $nat^{\mathfrak{I}} \setminus \{\perp_{nat}\}$ puede alcanzarse a partir de $0^{\mathfrak{D}}$ por sucesivas aplicaciones de la función $suc^{\mathfrak{D}}$. Sea id la función de $nat^{\mathfrak{I}}$ en $nat^{\mathfrak{I}}$ que denota (μX:nat→nat.(λx:nat.(if (es_cero x) then 0 else (suc (X (pred x))))))) Puesto que $\mathfrak{A} \vdash$ (7), para todo $d \in nat^{\mathfrak{I}}$, se verifica id(d) = d que, de acuerdo con el valor de id, significa que $d = \perp_{nat}$, que $d = 0^{\mathfrak{D}}$, o que $d = suc^{\mathfrak{D}}(id(pred^{\mathfrak{D}}(d)))$. Para los dos primeros casos, se tiene $h(\perp_{\mathbb{N}}) = \perp_{nat}$ y $h(0) = 0^{\mathfrak{D}}$; para el último caso, un tratamiento análogo con $pred^{\mathfrak{D}}(d)$, nos permitirá asegurar que existe un $n \in \mathbb{N}_{\perp}$ tal que $d = 0^{\mathfrak{D}}$ o $d = \underbrace{suc^{\mathfrak{D}}(...(suc^{\mathfrak{D}}(0^{\mathfrak{D}}))...)}_{\text{n veces}}$. Por tanto, para todo $d \in nat^{\mathfrak{I}}$ existe un $n \in \mathbb{N}$ tal que h(n) = d, es decir, h es sobre.

El recíproco es trivial, pues claramente, toda $\Sigma_{\mathbb{N}}$-estructura isomorfa a $\mathfrak{N}$ satisface las fórmulas (1) a (10). ■

Si consideramos el conjunto ST formado por las fórmulas (1) a (10) de la figura 1 junto con las fórmulas que sirven para caracterizar la suma y el producto (ver ejemplo §I-2.4.1), podríamos extender la proposición anterior diciendo que la aritmética estándar de los números naturales es caracterizable en PLPR. Si representamos por $\mathfrak{N}_{+,\cdot}$ la estructura $\mathfrak{N}$ aumentada con las operaciones suma y producto y por $Th(\mathfrak{N}_{+,\cdot})$ el conjunto de fórmulas que esta estructura satisface, entonces, $\psi \in Th(\mathfrak{N}_{+,\cdot}) \Leftrightarrow ST \vdash \psi$.

Obsérvese que toda estructura que satisfaga las fórmulas (1) a (6) debe contener a la estructura de los números naturales, ahora bien, los modelos no estándar se eliminan al tener que satisfacerse la fórmula (7) (que no es expresable en la lógica de primer orden) ya que esta fórmula sólo puede satisfacerse para los naturales y el bottom.

Lema 2.1.4

Si φ_T es satisfactible entonces T es coherente.

Demostración:

Sea $\mathfrak{A} = \langle I, \mathbb{D} \rangle$ un modelo de φ_T. Por construcción de φ_T, $\mathfrak{A}$ tiene que satisfacer las fórmulas (1) a (10) luego, por la proposición anterior, existe un isomorfismo h entre $\mathfrak{N}$ y $\mathfrak{A}$.

Definimos el recubrimiento F de P, $F:\mathbb{N} \times \mathbb{N} \rightarrow T$, a partir de $\langle I, \mathbb{D} \rangle$ y de h de la siguiente forma:

Sean $n, m \in \mathbb{N}$, $F(n,m) = d_i \Leftrightarrow \text{RLUD}^{\mathbb{D}}(h(n),h(m)) = d_i^{\mathbb{D}}$

Por hipótesis $\mathfrak{A} \vdash$ (11), esto nos asegura que si h(n) y h(m) son distintas de $\perp_{nat}$, $\text{RLUD}^{\mathbb{D}}(h(n),h(m)) = d_i^{\mathbb{D}}$ para algún d_i ($0<i<m$) de T. Por tanto, como $h(n) \neq \perp_{nat}$ para todo $n \in \mathbb{N}$, $F(n,m) \in T$, está bien definida y es total.

Para demostrar que F verifica la condición a) de la definición de recubrimiento T-coherente, representamos por π_i la proyección i-ésima de una n-tupla y comprobamos que para todos $n, m \in \mathbb{N}$, $\pi_1(F(n,m)) = \pi_2(F(n+1,m))$ y $\pi_3(F(n,m)) = \pi_4(F(n,m+1))$. Por la definición de F y puesto que $\mathfrak{A} \vdash$ (12), probar la condición anterior equivale a comprobar que para todos n y $m \in \mathbb{N}$, se verifica:

(i) $\text{RIGHT}^{\mathbb{D}}(h(n),h(m)) = \text{LEFT}^{\mathbb{D}}(h(\text{suc}(n)),h(m))$ y

(ii) $\text{UP}^{\mathbb{D}}(h(n),h(m)) = \text{DOWN}^{\mathbb{D}}(h(n),h(\text{suc}(m)))$.

Por ser $\mathfrak{A}$ un modelo de (13) se tiene que $\text{RIGHT}^{\mathbb{D}}(h(n),h(m)) = \text{LEFT}^{\mathbb{D}}(\text{suc}^{\mathbb{D}}(h(n)),h(m))$ y que $\text{UP}^{\mathbb{D}}(h(n),h(m)) = \text{DOWN}^{\mathbb{D}}(h(n),\text{suc}^{\mathbb{D}}(h(m)))$. De estas dos últimas igualdades, al ser h un homomorfismo, se deduce (i) y (ii).

Para comprobar que F cumple la condición b) de la definición de recubrimiento T-coherente hay que probar que $F(0,m) = d_0$ para infinitos $m \in \mathbb{N}$. Esto es cierto porque por ser $\mathfrak{A}$ un modelo de la fórmula (14) sabemos que para todo $n \in \mathbb{N}$ existe un $m \in \mathbb{N}$ tal que se cumple $\text{less}^{\mathbb{D}}(h(n),h(m))$ y $\text{RLUD}^{\mathbb{D}}(h(0),h(m)) = d_0^{\mathbb{D}}$. Las propiedades de h y la definición de F nos conduce a lo que queríamos probar. ∎

Proposición 2.1.5

El problema de satisfactibilidad en PLPR es Σ_1^1-duro.

Demostración:

Los lemas 2.1.2 y 2.1.4 nos aseguran que el problema del dominó

definido es m-reducible al problema de satisfactibilidad en PLPR. Entonces, puesto que el problema de dominó planteado es Σ_1^1-completo (Cfr. [Har-84]), la proposición queda demostrada. m

2.2 Π_1^1-COMPLETITUD

Una vez demostrado que la satisfactibilidad en PLPR es un problema Σ_1^1-duro, o lo que es lo mismo, la validez en PLPR es Π_1^1-duro, para probar que este último problema es Π_1^1-completo, basta con probar que pertenece a la clase Π_1^1 de la jerarquía aritmética. Para demostrar esta pertenencia probamos el teorema de Löwenheim-Skolem para PLPR y utilizamos el hecho de que la relación de satisfactibilidad para una asignación es equivalente a la satisfactibilidad en la lógica de predicados, lo que permite representar esta relación de manera aritmética.

Teorema 2.2.1 (Löwenheim-Skolem)

Sea $\Phi \subseteq L(\Sigma)$ un conjunto satisfactible, entonces existe un modelo que satisface a Φ cuyo dominio es numerable.

Demostración:

Reconstruyendo la demostración de la completitud del método de los tableaux, podemos asegurar que si Φ es satisfactible y $\mathcal{T}_\Phi$ es su tableau canónico, $\mathcal{T}_\Phi$ tiene al menos una rama abierta que por definición es tal que la unión H de los conjuntos que etiquetan sus nodos es un conjunto de Hintikka y $\hat{H\Sigma}(\Phi) \subseteq H$. En la demostración de la satisfactibilidad de los conjuntos de Hintikka se construye un modelo, $\mathfrak{I}_H$, cuyo dominio, $DAT^{\mathfrak{I}_H}$, consiste en clases de equivalencia $[t:\nu]$ con $t:\nu$ recorriendo los términos monomórficos. De la numerabilidad de $MT_m(\Sigma)$ podemos deducir que el modelo construido tiene un dominio numerable. Podemos afirmar entonces, que todo conjunto de Hintikka tiene un modelo numerable. En concreto, la interpretación que satisface H, con dominio numerable, también será un modelo de $\hat{H\Sigma}(\Phi)$, y por las características de $\mathfrak{I}_H$, también lo será de Φ. s

Teorema 2.2.2

El problema de satisfactibilidad en PLPR es Σ_1^1-completo y por tanto, el problema de validez de las fórmulas en PLPR es Π_1^1-completo.

Demostración:

El teorema de Löwenheim-Skolem para PLPR nos asegura que si φ es satisfactible tiene una estructura numerable, que como tal podrá codificarse mediante una función total f sobre los números naturales. Por otro lado, por ser $L(\Sigma)$ numerable, cada fórmula φ puede identificarse con un número natural n, finalmente puesto que en cada fórmula hay como mucho un número finito de variables de tipo, el conjunto de asignaciones correspondientes a las variables de tipo de un fórmula es numerable, y por tanto, cada asignación de este conjunto puede codificarse por medio de un número natural m. Además, por ser los conectivos de PLPR los de la lógica de predicados, de la misma forma que existe una relación aritmética que expresa el concepto de satisfactibilidad para la lógica de primer orden, podemos decir que el problema de validez de una fórmula $\varphi \in L(\Sigma)$ puede escribirse en términos de una relación aritmética ternaria R de la siguiente forma: $\forall f \; \forall \; m \; R(f,m,n)$ que, dado que el problema de validez en PLPR es Π^1_1-duro, nos lleva a concluir el teorema. ∎

Una consecuencia inmediata de este teorema es la no existencia de cálculos completos finitarios para PLPR que de existir proporcionarían un algoritmo computable para el problema de validez de una fórmula. Igualmente puede afirmarse que PLPR no puede verificar teoremas de compacidad. Estos hechos no restan potencia a PLPR en comparación a otras lógicas destinadas a ser la base de un sistema de deducción automático como PPλ o CCT. En realidad todas las lógicas que permiten razonar acerca de funciones computables suelen contener la teoría aritmética y por tanto ser incompletas como es el caso de LCF, NuPRL, etc.

3. INDUCCION DE PUNTO FIJO PARA PLPR

Hasta ahora hemos conseguido un sistema de deducción natural que como tal será adecuado para simular razonamientos humanos, pero que por ser infinitario, sólo puede automatizarse parcialmente. Eliminando las reglas infinitarias de $\mathcal{PRC}$ e introduciendo en su lugar un axioma de punto fijo y una regla de inducción para las fórmulas continuas, conseguimos un cálculo que llamaremos $\mathcal{PIC}$ que, como muestran los ejemplos, es capaz de deducir fórmulas no triviales; en concreto utilizando el axioma de punto fijo puede derivarse una regla de inducción para los naturales.

3.1 FORMULAS CONTINUAS

La regla de inducción de punto fijo que vamos a introducir está definida sobre un subconjunto de nuestro lenguaje. Al igual que en [Bak-80] la regla de punto fijo de Scott se define para la clase de las fórmulas correctas, o en [Man-74] la inducción paso a paso se define para las fórmulas admisibles, o en [Pau-87] la regla de inducción estructural se define para las fórmulas que son cadena completa; en PLPR la regla de inducción de punto fijo viene dada para el conjunto de fórmulas continuas que definimos a continuación.

Definición 3.1.1

Una fórmula φ es *continua con respecto a una variable funcional* $Y:\tau_1\to\tau_2$ que aparece libre en φ, si se verifica:

$$\{\varphi[(\mu X:\tau_1\to\tau_2.M)^i/Y:\tau_1\to\tau_2] \mid i<\omega\} \vdash \varphi[(\mu X:\tau_1\to\tau_2.M)/Y:\tau_1\to\tau_2].$$

Esta definición puede extenderse para un número finito de variables funcionales: φ es continua con respecto a las variables funcionales $Y_1:\tau_1^1\to\tau_2^1,\ldots,Y_n:\tau_1^n\to\tau_2^n$, que aparecen libres en φ si se verifica:

$$\{\varphi[(\mu X_1:\tau_1^1\to\tau_2^1.M_1)^i,\ldots,(\mu X_n:\tau_1^n\to\tau_2^n.M_n)^i/Y_1:\tau_1^1\to\tau_2^1,\ldots,Y_n:\tau_1^n\to\tau_2^n] \mid i<\omega\}$$

$$\vdash \varphi[(\mu X_1:\tau_1^1\to\tau_2^1.M_1),\ldots,(\mu X_n:\tau_1^n\to\tau_2^n.M_n)/Y_1:\tau_1^1\to\tau_2^1,\ldots,Y_n:\tau_1^n\to\tau_2^n]. \ \square$$

Definiremos la regla de inducción sobre el conjunto de fórmulas continuas y comprobaremos al introducir esta regla al cálculo $\mathcal{PRC}$, éste se mantiene correcto. La corrección de la regla de inducción se

basa en la definición de fórmula continua. Sin embargo, desde el punto de vista computacional se nos plantea un nuevo problema, porque no existe una caracterización sintáctica de las fórmulas continuas. Es más, como vamos a ver a continuación, existe al menos una signatura Σ para la que el conjunto de las Σ-fórmulas continuas es Σ_2-duro.

Consideremos la signatura $\Sigma_{\mathbb{N}}$ de la proposición 2.1.3 y simplifiquemos la sintaxis del $\Sigma_{\mathbb{N}}$-término (suc (...(suc 0)...)) por n. (n veces)

Definimos la $\Sigma_{\mathbb{N}}$-fórmula χ como la conjunción de las siguientes fórmulas:

$\forall x{:}nat\ \forall y{:}nat((suc\ x) = (suc\ y) \rightarrow x = y)$

$\forall x{:}nat\ \neg(suc\ x) = 0$

$(es\_cero\ 0) = true$

$\forall x{:}nat(\neg x \leq \bot{:}nat \rightarrow (es\_cero\ (suc\ x)) = false)$

$\forall x{:}nat(pred\ (suc\ x)) = x$

$(pred\ 0) = \bot{:}nat$

$\forall x{:}nat\ \neg(less\ (x,0))$

$\forall x{:}nat\ (less\ (0,(suc\ x)))$

$\forall x{:}nat\ \forall y{:}nat\ ((less\ ((suc\ x),(suc\ y))) \leftrightarrow (less\ (x,y)))$

Por otro lado, definimos R como la menor clase de $\Sigma_{\mathbb{N}}$-expresiones funcionales que verifica:

i) $(\lambda x_1{:}nat...x_k{:}nat.n) \in R$ $(1 \leq k,\ 0 \leq n)$ *(Funciones constantes)*

ii) $(\lambda x{:}nat.(suc\ x)) \in R$ *(Sucesor)*

iii) $(\lambda x_1{:}nat...x_k{:}nat.x_i) \in R$ $(1 \leq i \leq k)$ *(Proyección)*

iv) Si $M{:}\underbrace{nat_{\mathbb{N}}...x nat}_{k} \rightarrow nat \in R$ y $M_i{:}\underbrace{nat_{\mathbb{N}}...x nat}_{m} \rightarrow nat \in R$ $1 \leq i \leq k$

entonces $(\lambda x_1{:}nat...x_m{:}nat.(M\ ((M_1\ (x_1,...,x_m)),...,(M_k\ (x_1,..,x_m)))))$

pertenece a R $(1 \leq k, m)$ *(Sustitución)*

v) Si $M_1{:}\underbrace{nat_{\mathbb{N}}...x nat}_{k+1} \rightarrow nat \in R$, $M_2{:}\underbrace{nat_{\mathbb{N}}...x nat}_{k-1} \rightarrow nat \in R$ entonces:

$(\mu X{:}\underbrace{nat_{\mathbb{N}}..x nat}_{k} \rightarrow nat.(\lambda x_1{:}nat..x_k{:}nat.(if\ (es\_cero\ x_1)\ then\ (M_2\ (x_2,..,x_k))\ else\ (M_1\ ((pred\ x_1),(X\ (x_1,..,x_k)),x_2,...,x_k)))))$

pertenece a R *(Recursión)*

vi) Si $M{:}\underbrace{nat_{\mathbb{N}}...x nat}_{k+1} \rightarrow nat \in R$ y $M_0{:}\underbrace{nat_{\mathbb{N}}...x nat}_{k+1} \rightarrow nat$ es la expresión:

$(\lambda y_0{:}nat...y_k{:}nat.(if\ (es\_cero\ (M\ (y_0,...,y_k)))\ then\ y_k\ else\ (X\ (y_0,...,y_{k-1},(suc\ y_k)))))$,

entonces $(\lambda x_1{:}nat...x_k{:}nat.((\mu X{:}\underbrace{nat_{\mathbb{N}}...x nat}_{k+1} \rightarrow nat.M_0)\ (x_1,...,x_k,0)))$

pertenece a R *(Minimización)*

Sea $\mathfrak{A} = \langle I, \mathcal{D} \rangle$ una Σ_M-estructura que satisface χ entonces, podemos asegurar que por contrucción de χ, existe un subconjunto S de $nat^I \in T^I$ tal que $\mathfrak{N}$ es isomorfo a $\langle S, \mathcal{D} \rangle$.

Sea $\mathfrak{R}$ la clase de las funciones recursivas parciales de $\mathbb{N}^k$ en $\mathbb{N}$ ($1 \leq k$) que sabemos es recursivamente enumerble. Un algoritmo análogo al utilizado para enumerar las funciones de $\mathfrak{R}$ sirve para enumerar R. Por ello y por la semántica de las expresiones funcionales, si ϕ_n^k (ϕ_n si k=1) es la función recursiva de aridad k y de índice n, y si h es el isomorfismo entre $\mathfrak{N}$ y $\langle S, \mathcal{D} \rangle$, podemos afirmar que para n y cada k podemos encontrar de forma efectiva una expresión funcional de aridad k que llamaremos $M_n^k: \underbrace{nat \times \ldots \times nat}_{k} \to nat$ y que verifica $\mathfrak{A}\left[M_n^k: \underbrace{nat \times \ldots \times nat}_{k} \to nat\right](h((n_1), \ldots, h(n_k)) = \phi_n^k(n_1, \ldots, n_k)$ para toda k-tupla $\langle n_1, \ldots, n_k \rangle \in \mathbb{N}^k$

La fórmula χ y la notación $M_n^k: \underbrace{nat \times \ldots \times nat}_{k} \to nat$ para las expresiones de R (eliminamos el superíndice cuando k=1) se utiliza en la demostración de la siguiente proposición.

Teorema 3.1.3

El conjunto de fórmulas continuas es Σ_2-duro.

Demostración :

La demostración consite en reducir un conjunto Σ_2-completo al conjunto de fórmulas continuas. Utilizamos para ello el conjunto Σ_2-completo FIN = {n | el dominio de ϕ_n es finito}, [Rog-67]. Para cada n definimos una fórmula φ^n y probamos que $n \in FIN \Leftrightarrow \varphi^n$ es continua.

Sea $n \in \mathbb{N}$, definimos φ^n como $\chi \to \chi^n$ donde χ ha sido definida con anterioridad y χ^n es $\exists x{:}nat\ \forall y{:}nat\ ((less\ (x,y)) \to (M_n\ (ID\ y)) = \bot{:}nat)$ siendo $ID{:}nat \to nat$ una simplificación de la expresión funcional $(\mu X{:}nat \to nat.(\lambda x{:}nat.(if\ (es\_cero\ x)\ then\ 0\ else\ (suc\ (X\ (pred\ x))))))$

Al resultado de sustituir la μ-expresión $ID{:}nat \to nat$ en χ^n por la i-ésima aproximación sintáctica $(ID{:}nat \to nat)^i$, es decir, a la fórmula $\exists x{:}nat\ \forall y{:}nat\ ((less\ (x,y)) \to (M_n\ (ID^i\ y)) = \bot{:}nat)$ la llamamos χ^{ni}.

Veamos que $n \in FIN \Leftrightarrow \chi \to \chi^n$ continua.

$\Rightarrow$) Sea $n \in FIN$ y $\mathfrak{A} = \langle I, \mathcal{D} \rangle$ un modelo de χ. Por construcción, χ admite modelos no estándar, pero sabemos que existe un isomorfismo h entre $\mathfrak{N}$ y $\langle S, \mathcal{D} \rangle$ siendo $S \subseteq nat^I$. Por la definición de FIN y las

propiedades de M_n:nat→nat, $\mathfrak{A}[M_n:nat\to nat](h(m)) = \perp_{nat}$ para infinitos $m \in \mathbb{N}$. Es fácil comprobar que por ser $\mathfrak{A}$ un modelo de χ, $\mathfrak{A}[ID:nat\to nat](h(m)) = h(m)$ para cualquier $m \in \mathbb{N}$. Por tanto, si $n \in FIN$ tendremos que $\mathfrak{A}[M_n:nat\to nat](\mathfrak{A}[ID:nat\to nat](h(m))) = \perp_{nat}$ para un número infinito de elementos $h(m) \in S$. Deducimos por ello, que existe un $u \in nat^{\mathfrak{I}}$ tal que para todo $v \in nat^{\mathfrak{I}}$ si $less^{\mathfrak{D}}(u,v)$ entonces $\mathfrak{A}[M_n:nat\to nat]\ (\mathfrak{A}[ID:nat\to nat]\ (v)) = \perp_{nat}$ y por tanto, $\mathfrak{A} \vdash \chi^n$ con lo que $\vdash \chi \to \chi^n$. Por otro lado, para cualquier $n \in \mathbb{N}$, si $\mathfrak{A} \vdash \chi$ entonces, $\mathfrak{A}[M_n:nat\to nat]\ (\mathfrak{A}[(ID:nat\to nat)^i](h(m)) = \perp_{nat}$ para todo $m>i$, entonces, siguiendo un razonamiento análogo al utilizado antes para χ^n, probamosque $\mathfrak{A} \vdash \chi\to\chi^{ni}$ para todo $i<\omega$, que nos permite concluir que φ^n es continua.

*) Lo probamos por reducción al absurdo. Sea $n \notin FIN$, claramente la estructura $\mathfrak{N}$ satisface χ. El isomorfismo h de la implicación anterior será en este caso la identidad, y como antes, $\mathfrak{N} \vdash \chi$ implica $\mathfrak{N}[M_n:nat\to nat]\ (\mathfrak{N}[(ID:nat\to nat)^i](n) = \perp_{nat}$ para todo $m>i$, de donde deducimos $\mathfrak{N} \vdash \chi\to\chi^{ni}$ para todo $i<\omega$. Ahora bien, por definición de M_n:nat→nat y de FIN, $\mathfrak{N}[M_n:nat\to nat](m) = \perp_{nat}$ sólo para un número finito de elementos $m \in \mathbb{N}$, pero $\mathfrak{N}[ID:nat\to nat](m) = m$ para todo $m \in \mathbb{N}$ y por tanto, $\mathfrak{N}$ no es un modelo de χ^n. Concluimos que si $n \notin FIN$, existe una estructura $\mathfrak{N}$, que para todo $i<\omega$ satisface $\chi\to\chi^{ni}$, pero $\mathfrak{N}$ no es un modelo de $\chi\to\chi^n$, es decir, φ^n no es continua.

Queda probado que FIN es m-reducible al conjunto de $\Sigma\mathbb{N}$-fórmulas continuas que por lo tanto es Σ_2-duro. ■

La continuidad de las fórmulas es un concepto semántico que como hemos visto no tiene una caracterización sintáctica. Sin embargo, un gran número de expresiones sintácticas puede reconocerse de una manera automática como fórmulas continuas.

Proposición 3.1.4

a) Las siguientes fórmulas son continuas con respecto a una variable $Y:\tau_1\to\tau_2$ que aparece libre en ellas:

$t_1:\tau \subseteq t_2:\tau$

$\neg(t:\tau \subseteq \perp:\tau \wedge \perp:\tau \subseteq t:\tau)$

$\neg t_1:\tau \subseteq t_2:\tau$ si $Y:\tau_1\to\tau_2$ no aparece libre en $t_1:\tau$

$\neg(p\ t:\tau)$

b) Si φ y ψ son continuas con respecto a $Y:\tau_1\to\tau_2$ entonces las siguientes fórmulas son también continuas:

$(\varphi \wedge \psi)$

$(\varphi \vee \psi)$ si φ y ψ son además monomórficas

$\forall x:\tau\ \varphi$

$\exists x:\tau\ \varphi$

c) Si $\neg\varphi$ es continua con respecto a $Y:\tau_1\to\tau_2$ y monomórfica, entonces las siguientes fórmulas son también continuas:

$\neg\forall x:\nu\ \varphi$

$\neg\exists x:\nu\ \varphi$

d) Si φ es continua con respecto a $Y:\tau_1\to\tau_2$, entonces, $\varphi[M:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]$ es continua con respecto a todas las variables funcionales libres en $M:\tau_1\to\tau_2$ de tipo $\tau_1\to\tau_2$.

La demostración de esta proposición se basa en la semántica de las fórmulas y en la equivalencia entre las aproximaciones sintácticas y semánticas. ■

3.2 AXIOMA Y REGLA DE INDUCCION DE PUNTO FIJO

Definición 3.2.1

Se define el cálculo $\mathcal{PFC}$ mediante el sistema de reglas de derivación de $\mathcal{PRC}$ excluyendo las reglas para las aproximaciones de punto fijo y añadiendo las siguientes reglas:

Regla de inducción de punto fijo

$$(\mathrm{IND})\ \frac{\varphi[\bot:\tau_1\to\tau_2/Y:\tau_1\to\tau_2] \qquad \begin{bmatrix}\varphi[X:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]\\ \vdots \\ \varphi[M:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]\end{bmatrix}}{\varphi[(\mu X:\tau_1\to\tau_2.M)/Y:\tau_1\to\tau_2]}$$

siendo φ continua con respecto a $Y:\tau_1\to\tau_2$, $X:\tau_1\to\tau_2$ no aparece libre en φ ni en las hipótesis no canceladas de las derivaciones de las premisas, $X:\tau_1\to\tau_2 \notin Lib(M:\tau_1\to\tau_2)$ y $M:\tau_1\to\tau_2$ no contiene definiciones recursivas.

Axioma de punto fijo

$$(\mathrm{FIX})\ \frac{}{((\mu X:\tau_1\to\tau_2.M)\ t:\tau_1) = (M[(\mu X:\tau_1\to\tau_2.M)/X:\tau_1\to\tau_2]\ t:\tau_1)}$$ □

Conviene señalar que la regla de inducción puede extenderse de manera natural a una regla de inducción simultánea para fórmulas que sean continuas con respecto a varias variables de función.

Teorema 3.2.2

EL cálculo $\mathcal{PIC}$ es correcto.

<u>Demostración</u>:

Vamos a probar:

(i) $\vdash ((\mu X{:}\tau_1\to\tau_2.M)\ t{:}\tau_1) = (M{:}\tau_1\to\tau_2[(\mu X{:}\tau_1\to\tau_2.M)/X{:}\tau_1\to\tau_2]\ t{:}\tau_1)$

(ii) Supuesta la siguiente hipótesis de inducción:

Para todo conjunto Γ_0 que contenga las hipótesis de $\begin{array}{c} D_0 \\ \varphi[\bot/Y{:}\tau_1\to\tau_2] \end{array}$, se verifica $\Gamma_0 \vdash \varphi[\bot{:}\tau_1\to\tau_2/Y{:}\tau_1\to\tau_2]$ y para todo conjunto Γ que contenga las hipótesis de la derivación $\begin{array}{c} \varphi[X{:}\tau_1\to\tau_2/Y{:}\tau_1\to\tau_2] \\ D \\ \varphi[M{:}\tau_1\to\tau_2/Y{:}\tau_1\to\tau_2] \end{array}$ se verifica $\Gamma \vdash \varphi[M{:}\tau_1\to\tau_2/Y{:}\tau_1\to\tau_2]$.

Entonces, si Φ es un conjunto que contiene las hipótesis de la derivación

$$\dfrac{\begin{array}{c} \\ D_0 \\ \varphi[\bot/Y{:}\tau_1\to\tau_2] \end{array} \quad \begin{array}{c} [\varphi[X{:}\tau_1\to\tau_2/Y{:}\tau_1\to\tau_2]] \\ D \\ \varphi[M{:}\tau_1\to\tau_2/Y{:}\tau_1\to\tau_2] \end{array}}{\varphi[(\mu X{:}\tau_1\to\tau_2.M)/Y{:}\tau_1\to\tau_2]}$$

y en el que no aparece libre la variable $X{:}\tau_1\to\tau_2$, entonces $\Phi \vdash \varphi[(\mu X{:}\tau_1\to\tau_2.M)/Y{:}\tau_1\to\tau_2]$.

Una vez probados (i) y (ii), utilizando la demostración de la corrección de $\mathcal{PRC}$ podemos concluir que $\mathcal{PIC}$ es correcto.

Demostración de (i):

Para cualquier interpretación $\mathfrak{I}=\langle\mathcal{I},\mathcal{D},\xi\rangle$ y $\eta \in AT_P{}^{\mathcal{I}}$ se tiene:

$\mathfrak{I}[\![((\mu X{:}\tau_1\to\tau_2.M)\ t{:}\tau_1)]\!]\ \eta = \mathbf{fix}\ T(\mathfrak{I},X{:}\tau_1\to\tau_2,M,\eta)\ (\mathfrak{I}[\![t{:}\tau_1]\!]\eta)$

$= T(\mathfrak{I},X{:}\tau_1\to\tau_2,M,\eta)(\mathbf{fix}\ T(\mathfrak{I},X{:}\tau_1\to\tau_2,M,\eta))\ (\mathfrak{I}[\![t{:}\tau_1]\!]\eta)$

por la definición de punto fijo

$= \mathfrak{I}\{\mathbf{fix}\ T(\mathfrak{I},X{:}\tau_1\to\tau_2,M,\eta)/X{:}\tau_1\to\tau_2\}[\![M{:}\tau_1\to\tau_2]\!]\eta\ (\mathfrak{I}[\![t{:}\tau_1]\!]\eta)$

por la definición del operador T

$= \mathfrak{I}[\![M{:}\tau_1\to\tau_2[(\mu X{:}\tau_1\to\tau_2.M)/X{:}\tau_1\to\tau_2]]\!]\eta\ (\mathfrak{I}[\![t{:}\tau_1]\!]\eta)$

por el lema de sustitución y la semántica del μ-operador

$= \mathfrak{I}[\![(M{:}\tau_1\to\tau_2[(\mu X{:}\tau_1\to\tau_2.M)/X{:}\tau_1\to\tau_2]\ t{:}\tau_1)]\!]\eta$.

Demostración de (ii):

Partimos de las hipótesis de inducción enunciadas en (ii) y consideramos un conjunto Φ que contiene las hipótesis no canceladas de

la derivación $\dfrac{\overset{\text{Do}}{\varphi[\bot/Y:\tau_1\to\tau_2]} \quad \overset{\text{D}}{\varphi[M:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]}}{\varphi[(\mu X:\tau_1\to\tau_2.M)/Y:\tau_1\to\tau_2]}$ $[\varphi[X:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]]$ y en el que $X:\tau_1\to\tau_2$ no aparece libre. Sea $\mathfrak{I}$ una interpretación tal que $\mathfrak{I} \models \Phi$. Vamos a probar que para todo $i<\omega$ se verifica $\mathfrak{I} \models \varphi[(\mu X:\tau_1\to\tau_2.M)^i/Y:\tau_1\to\tau_2]$. Lo probamos por inducción en i.

Para i=0, tenemos que por construcción Φ contiene las hipótesis de $\overset{\text{Do}}{\varphi[\bot/Y:\tau_1\to\tau_2]}$ entonces, por inducción, $\mathfrak{I} \models \varphi[\bot:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]$ y como $(\mu X:\tau_1\to\tau_2.M)^0 = \bot:\tau_1\to\tau_2$, tenemos $\mathfrak{I} \models \varphi[(\mu X:\tau_1\to\tau_2.M)^0/Y:\tau_1\to\tau_2]$.

Paso de inducción: Supongamos $\mathfrak{I} \models \varphi[(\mu X:\tau_1\to\tau_2.M)^i/Y:\tau_1\to\tau_2]$ para un $i<\omega$ que es lo mismo que $\mathfrak{I} \models \varphi[X:\tau_1\to\tau_2/Y:\tau_1\to\tau_2][(\mu X:\tau_1\to\tau_2.M)^i/X:\tau_1\to\tau_2]$ pues $X:\tau_1\to\tau_2$ no aparece libre en φ. Entonces, $\mathfrak{I}\{\mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^i]/X:\tau_1\to\tau_2\} \models \varphi[X:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]$ por el lema de sustitución. Utilizando el lema de coincidencia, puesto que $\mathfrak{I} \models \Phi$ y $X:\tau_1\to\tau_2$ no aparece libre en Φ, $\mathfrak{I}\{\mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^i]/X:\tau_1\to\tau_2\} \models \Phi$. Simplificamos la escritura de $\mathfrak{I}\{\mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^i]/X:\tau_1\to\tau_2\}$ por $\mathfrak{I}'$. Tenemos $\mathfrak{I}' \models \varphi[X:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]$ y $\mathfrak{I}' \models \Phi$ luego, por hipótesis de inducción, como $\Phi \cup \{\varphi[X:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]\}$ contiene las hipótesis de $\overset{\text{D}}{\varphi[X:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]}$, $\mathfrak{I}' \models \varphi[M:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]$ lo cual implica que $\overset{\text{D}}{\varphi[M:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]}$

$\mathfrak{I}'\{\mathfrak{I}'[M:\tau_1\to\tau_2]/Y:\tau_1\to\tau_2\} \models \varphi$ por el lema de sustitución. La definición de las aproximaciones sintácticas y el lema de sustitución nos aseguran la igualdad $\mathfrak{I}'[M:\tau_1\to\tau_2] = \mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^{i+1}]$ ya que $M:\tau_1\to\tau_2$ no contiene definiciones recursivas y por tanto, $(M:\tau_1\to\tau_2)^{i+1} = M:\tau_1\to\tau_2$. Tenemos entonces, $\mathfrak{I}'\{\mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^{i+1}]/Y:\tau_1\to\tau_2\} \models \varphi$, y por el lema de coincidencia, $\mathfrak{I}\{\mathfrak{I}[(\mu X:\tau_1\to\tau_2.M)^{i+1}]/Y:\tau_1\to\tau_2\} \models \varphi$ de donde, por el lema de sustitución $\mathfrak{I} \models \varphi[(\mu X:\tau_1\to\tau_2.M)^{i+1}/Y:\tau_1\to\tau_2]$ como queríamos probar.

Por ser φ continua y verificarse $\mathfrak{I} \models \varphi[(\mu X:\tau_1\to\tau_2.M)^i/Y:\tau_1\to\tau_2]$ para todo $i<\omega$ deducimos que $\mathfrak{I} \models \varphi[(\mu X:\tau_1\to\tau_2.M)/Y:\tau_1\to\tau_2]$ con lo que llegamos a la conclusión de que $\Phi \models \varphi[(\mu X:\tau_1\to\tau_2.M)/Y:\tau_1\to\tau_2]$. ∎

Corolario 3.2.3

La regla de inducción sigue siendo correcta si se enuncia de la siguiente forma:

$$(\mathrm{IND}')\ \frac{\varphi[\bot:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]\ \ \varphi[X:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]\to\varphi[M:\tau_1\to\tau_2/Y:\tau_1\to\tau_2]}{\varphi[(\mu X:\tau_1\to\tau_2.M)/Y:\tau_1\to\tau_2]}$$

Esta forma de la regla de inducción será la utilizada en la implementación del sistema de deducción automática basado en PLPR que se presenta en el capítulo IV.

El cálculo $\mathcal{PJC}$ que acabamos de construir aunque, de acuerdo con las propiedades de PLPR que estudiamos en la sección anterior, no puede ser completo, resulta sin embargo más manejable a un usuario y tiene buenas propiedades de automatización.

Como veremos en los siguientes ejemplos utilizando la inducción de $\mathcal{PJC}$, se puede deducir la validez fórmulas de una cierta complejidad que presentan definiciones recursivas. $\mathcal{PJC}$ permite también derivar una regla de inducción sobre los naturales que asegura la completitud del cálculo para estructuras aritméticas.

Ejemplo 3.2.4 ([Man 74], [Pau 87])

Sea Σ_t una signatura de tipos cualquiera y $\Sigma_d=\{f:\rho\to bool, g:\rho\to\rho\}$, como siempre $\Sigma = \langle\Sigma_t, \Sigma_d\rangle$. Sea $H:\rho\to\rho$ la Σ-expresión funcional $(\mu X:\rho\to\rho.(\lambda x:\rho.(\text{if } (f\ x):bool \text{ then } x \text{ else } (X\ (X\ (g\ x):\rho):\rho):\rho):\rho))$. Se trata de probar que la fórmula $\forall y:\rho\ (H\ (H\ y):\rho):\rho = (H\ y):\rho$ es válida en PLPR utilizando $\mathcal{PJC}$. La continuidad de $\forall y:\rho\ (H\ (X\ y):\rho) = (X\ y):\rho$ nos permite utilizar la regla de inducción de punto fijo. Consideremos las derivaciones Do y D siguientes:

- Sea co una constante auxiliar de tipo ct siendo ct una constante de tipo auxiliar. Do consiste en la derivación del caso básico y es el siguiente árbol

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\bot\ co:ct):ct = \bot}(DEF)\quad \dfrac{}{(H\ (\bot\ co:ct))=(H\ (\bot\ co:ct))}(REF)}{(H\ (\bot\ co:ct))=(H\ \bot:ct)}(SUB)\quad \dfrac{}{(H\ \bot:ct) = \bot:ct}STR}{(\bot\ co:ct):ct = \bot:ct}(=TRAN)\quad (DEF)\ (H\ (\bot\ co:ct))=\bot:ct}{\bot:ct = (\bot\ co:ct):ct}(=SIM)}{(H\ (\bot\ co:ct)) = (\bot\ co:ct)}(=TRAN)}{\forall y:ct\ (H\ (\bot\ y)):ct = (\bot\ y):ct}(VI)$$

- La derivación D representa el paso de inducción. Supuesta cierta la fórmula $\forall y{:}ct\ (H\ (X\ y){:}ct){:}ct = (X\ y){:}ct$ se trata de probar la fórmula $\forall y{:}ct\ (H\ (M\ y){:}ct){:}ct = (M\ y){:}ct$ donde M simplifica la expresión funcional $(\lambda x{:}ct.(\text{if}\ (f\ x)\ \text{then}\ x\ \text{else}\ (X\ (X\ (g\ x)))))$. Consideramos las siguientes simplificaciones en la notación:

.to:ct es el $\hat{\Sigma}$-término (if (f co:ct) then co:ct else (X (X (g co:ct))))

.t:ct es igual a (if (f x:ct) then x:ct else (H (H (g x:ct))))

Construimos los siguientes subárboles de la derivación D.

- $D_{1.1}$ es el árbol $\dfrac{(f\ co{:}ct) = \bot{:}bool}{to{:}ct = \bot{:}ct}\ (\bot If)$

- D_1 consite en la derivación siguiente.

$$\dfrac{\dfrac{D_{1.1}\ \dfrac{}{(H\ to{:}ct) = (H\ to{:}ct)}\ (=REF)}{(H\ to{:}ct) = (H\ \bot{:}ct)}\ (SUB) \qquad \dfrac{}{(H\ \bot{:}ct) = \bot{:}ct}\ (STR)}{(H\ to{:}ct) = \bot{:}ct}\ (=TRAN)$$

$$\dfrac{\dfrac{D_{1.1}}{\bot{:}ct = to{:}ct}\ (=SIM) \qquad (H\ to{:}ct) = \bot{:}ct}{(H\ to{:}ct) = to{:}ct}\ (=TRAN)$$

- $D_{2.1}$, $D_{2.2}$ y $D_{2.3}$ son los subárboles siguientes:

$$D_{2.1}:\ \dfrac{(f\ co{:}ct) = true}{to{:}ct = co{:}ct}\ (TIf)$$

$$D_{2.2}:\ \dfrac{}{(H\ co{:}ct) = ((\lambda x{:}ct.\,t{:}ct)\ co{:}ct)}\ (FIX)$$

$$D_{2.3}:\ \dfrac{\dfrac{[\neg co{:}ct \leq \bot{:}\tau]}{((\lambda x{:}ct.\,t{:}ct)\ co{:}ct) = t\left[\frac{co:ct}{x:ct}\right]}\ (ABS) \qquad \dfrac{(f\ co{:}ct) = true}{t\left[\frac{co:ct}{x:ct}\right] = co{:}ct}\ (TIf)}{((\lambda x{:}ct.\,t{:}ct)\ co{:}ct) = co{:}ct}\ (=TRAN)$$

A partir de ahora, puesto que todos los términos son de tipo ct, eliminaremos la notación de los tipos.

- D_2 es la siguiente derivación.

$$\dfrac{\dfrac{\dfrac{}{(H\ to) = (H\ to)}\ (=REF)}{(H\ to) = (H\ co)}\ (SUB) \qquad \dfrac{}{(H\ co) = co}\ (=TRAN)}{(H\ to) = co}\ (=TRAN) \qquad \dfrac{D_{2.1}}{co = to}\ (=SIM)}{(H\ to{:}ct) = to{:}ct}\ (=TRAN)$$

- $D_{3.1}$ y $D_{3.2}$ consisten en los subárboles:

$$D_{3.1}:\ \dfrac{(f\ co{:}ct) = false}{to = (X\ (X\ (g\ co)))}\ (FIf)$$

$$D_{3.2}:\ \dfrac{\forall y{:}ct\ (H\ (X\ y)) = (X\ y)}{(H\ (X\ (X\ (g\ co)))) = (X\ (X\ (g\ co)))}\ (\forall E)$$

- D_3 queda constituioda por:

$$\dfrac{\dfrac{\dfrac{}{(H\ to) = (H\ to)}\ (=REF) \qquad D_{3.1}}{(H\ to) = (H\ (X\ (X\ (g\ co))))}\ (SUB) \qquad D_{3.2}}{(H\ to) = (X\ (X\ (g\ co)))}\ (=TRAN) \qquad \dfrac{D_{3.1}}{(X\ (X\ (g\ co)))=to}\ (=SIM)}{(H\ to{:}ct) = to{:}ct}\ =TRAN$$

Las derivaciones D_1, D_2 y D_3 pueden ser alargadas de manera que para cada D_i tenemos la derivación

$$\dfrac{\dfrac{\dfrac{[\neg co{:}ct \leq \bot{:}ct]}{(M\ co) = to{:}ct}\ (ABS)}{to{:}ct = (M\ co)}\ (=SIM) \qquad D_1}{(H\ (M\ co)) = (M\ co)}\ (SUB)$$

Sabemos entonces que supuesto $\forall y{:}ct\ (H\ (X\ y)) = (X\ y)$, si $\neg co \leq \bot{:}ct$, existe una derivación de $(H\ (M\ co)) = (M\ co)$ tanto en el caso de que $(f\ co) = \bot$, como si $(f\ co) = true$, como si $(f\ co) = false$. Si $co \leq \bot{:}ct$, utlizando las las reglas (BOT) y (STR), también derivamos $(H\ (M\ co)) = (M\ co)$. De las reglas (Exc) y ($\forall E$) acabaríamos deduciendo que las derivaciones D_i, $i=1,2,3$, son válidas para un co cualquiera. Todo esto nos lleva a la construcción de la derivación D:

$$\dfrac{}{(f\ co)=\perp : bool \vee (f\ co)=true \vee (f\ co)=false}\ (Bool)$$

$$\dfrac{\dfrac{\dfrac{}{(f\ co)=\perp : bool \vee (f\ co)=true \vee (f\ co)=false}\ (Bool) \qquad [\forall y:ct\ (H\ (X\ y)) = (X\ y)]}{\begin{array}{ccc}[(f\ co)=\perp] & [(f\ co)=true] & [(f\ co)=false] \\ \vdots & \vdots & \vdots \\ (H\ (M\ co))=(M\ co) & (H\ (M\ co))=(M\ co) & (H\ (M\ co))=(M\ co)\end{array}}(\vee E)}{\dfrac{(H\ (M\ co)) = (M\ co)}{\forall y:ct\ (H\ (M\ y)) = (M\ y)}\ (\forall I)}$$

Concluimos la prueba con el árbol siguiente:

$$\dfrac{\dfrac{Do \qquad D}{\forall y:ct\ (H\ (H\ y):ct):ct = (H\ y):ct}\ (IND)}{\forall y:\rho\ (H\ (H\ y):\rho):\rho = (H\ y):\rho}\ (TSI)$$

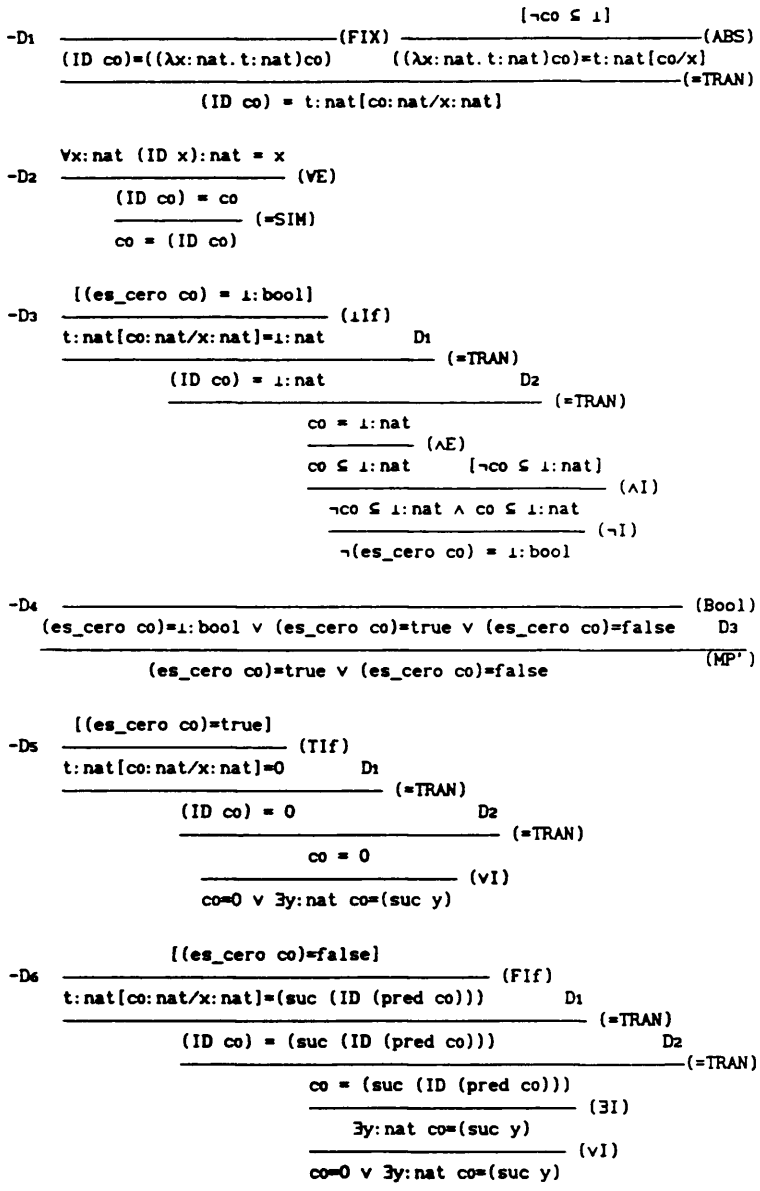
Ejemplo 3.2.5 (Una derivación de la regla de inducción en $\mathbb{N}$)

Consideramos la signatura $\Sigma_{\mathbb{N}}$ empleada otras veces para caracterizar los números naturales. ID:nat→nat una simplificación de $(\mu X{:}nat \to nat.(\lambda x{:}nat.(if\ (es\_cero\ x)\ then\ 0\ else\ (suc\ (X\ (pred\ x))))))$ Utilizando solo la regla (FIX) y las reglas propias de un cálculo de primer orden correspondientes a $\mathcal{PFC}$ vamos demostrar que existe una derivación de la regla de inducción para los naturales que enunciamos de la siguiente forma:

$$\dfrac{\forall x:nat\ (ID\ x) = x \qquad \varphi[0:nat/x:nat] \qquad \begin{array}{c}[\ \varphi[c:nat/x:nat]\ ] \\ \vdots \\ \varphi[(suc\ c:nat)/x:nat]\end{array}}{\forall x:nat\ (\neg x \leq \perp:nat \to \varphi)}$$

Siendo c:nat una constante que no aparece en φ ni en las hipótesis no canceladas de la derivación de φ[c:nat/x:nat]. Al introducir la fórmula ∀x:nat (ID x) = x en las premisas, se obliga a que los modelos que la satisfagan sean isomorfos al estándar de los números naturales.

Sea co una constante auxiliar de tipo nat, para simplificar la notación, no escribiremos su tipo en la derivación. También simplificamos (if (es_cero x:nat) then 0 else (suc (ID (pred x:nat)))) por t:nat. Descomponemos la derivación en una serie de subárboles, como sigue:


-D1
(FIX)
[¬co ≤ ⊥]
(ABS)
(ID co)=((λx:nat. t:nat)co)
((λx:nat. t:nat)co)=t:nat[co/x]
(=TRAN)
(ID co) = t:nat[co:nat/x:nat]
-D2
∀x:nat (ID x):nat = x
(∀E)
(ID co) = co
(=SIM)
co = (ID co)
-D3
[(es_cero co) = ⊥:bool]
(⊥If)
t:nat[co:nat/x:nat]=⊥:nat
D1
(=TRAN)
(ID co) = ⊥:nat
D2
(=TRAN)
co = ⊥:nat
(∧E)
co ≤ ⊥:nat
[¬co ≤ ⊥:nat]
(∧I)
¬co ≤ ⊥:nat ∧ co ≤ ⊥:nat
(¬I)
¬(es_cero co) = ⊥:bool
-D4
(Bool)
(es_cero co)=⊥:bool ∨ (es_cero co)=true ∨ (es_cero co)=false
D3
(MP')
(es_cero co)=true ∨ (es_cero co)=false
-D5
[(es_cero co)=true]
(TIf)
t:nat[co:nat/x:nat]=0
D1
(=TRAN)
(ID co) = 0
D2
(=TRAN)
co = 0
(∨I)
co=0 ∨ ∃y:nat co=(suc y)
-D6
[(es_cero co)=false]
(FIf)
t:nat[co:nat/x:nat]=(suc (ID (pred co)))
D1
(=TRAN)
(ID co) = (suc (ID (pred co)))
D2
(=TRAN)
co = (suc (ID (pred co)))
(∃I)
∃y:nat co=(suc y)
(∨I)
co=0 ∨ ∃y:nat co=(suc y)

-D7 Sea c una constante auxiliar de tipo nat

$$\dfrac{\dfrac{\dfrac{}{(\text{suc c}) = \text{co}}\,[\text{co}=(\text{suc c})]\ (=\text{SIM}) \qquad \begin{array}{c}[\ \varphi[\text{c:nat}/\text{x:nat}]\ ]\\ \vdots\\ \varphi[(\text{suc c}):\text{nat}/\text{x:nat}]\end{array}\ \text{premisa}}{\varphi[\text{co:nat}/\text{x:nat}]}\,(\text{SUB}) \qquad [\exists \text{y:nat co}=(\text{suc y})]}{\varphi[\text{co:nat}/\text{x:nat}]}\,(\exists\text{E})$$

Por fin, con el siguiente árbol terminamos la prueba.

$$\dfrac{\dfrac{\dfrac{\dfrac{D_4 \qquad D_5 \qquad D_6}{\text{co}=0 \vee \exists \text{y:nat co}=(\text{suc y})}\,(\vee\text{E}) \qquad \dfrac{[\text{co} = 0]\ \varphi[0:\text{nat}/\text{x:nat}]}{\varphi[\text{co:nat}/\text{x:nat}]}\,(\text{SUB}) \qquad D_7}{\varphi[\text{co:nat}/\text{x:nat}]}\,(\vee\text{E})}{\neg \text{co} \leq \bot\text{:nat} \rightarrow \varphi[\text{co:nat}/\text{x:nat}]}\,(\text{FTh})}{\forall \text{x:nat}\ (\neg \text{x} \leq \bot\text{:nat} \rightarrow \varphi)}\,(\forall\text{I})$$

Por medio de la regla que acabamos de derivar, podremos demostrar propiedades de los números naturales que no pueden probarse en la lógica de primer orden. Es posible demostrar, por ejemplo, que el factorial de un número es una función total. Esta afirmación se expresa mediante la fórmula $\forall x{:}nat\ (\neg x \leq \bot{:}nat \rightarrow \neg(fact\ x) \leq \bot{:}nat)$, donde $fact{:}nat \rightarrow nat$ representa la expresión funcional siguiente:
$(\mu X{:}nat \rightarrow nat.(\lambda x{:}nat.(if\ (es\_cero\ x)\ then\ x\ else\ (*\ (x,(X(pred\ x)))))))$
Si consideramos las fórmulas que caracterizan la aritmética estándar (ver §I-2.4), se puede probar $\neg(fact\ 0) \leq \bot{:}nat$ y se puede derivar la fórmula $\neg(fact\ (suc\ c{:}nat)) \leq \bot{:}nat$ a partir de $\neg(fact\ c{:}nat) \leq \bot{:}nat$ para un $c{:}nat$ auxiliar. Aplicando la regla de inducción para los naturales que acabamos de introducir en este ejemplo, llegamos al resultado esperado.

CAPITULO IV

MIZ-PR: UN SISTEMA DE DEMOSTRACION AUTOMATICA BASADO EN PLPR

En la consecución de una demostración dentro del ámbito de la ATP pueden distinguirse dos maneras de operar: búsquedas completamente automáticas y búsquedas de la demostración con interacción del usuario con la máquina. En este capítulo presentamos un sistema de deducción automática basado en la lógica PLPR, llamado MIZ-PR, en el que se pueden combinar las técnicas de comprobación automática de demostraciones con la interacción hombre-máquina. Estudiaremos cómo se mecanizan las demostraciones de la lógica y cuáles son las características de MIZ-PR.

Para analizar nuestro sistema, es importante señalar que en ATP suelen adoptarse dos puntos de vista bien diferenciados: el de la lógica como algo estático que fundamenta la construcción del sistema y el motivado por la orientación humana, es decir, por la simulación de las técnicas humanas usadas para resolver problemas; desde este segundo punto de vista, los tratamientos específicos del sistema se van mostrando conforme se progresa en su construcción.

Concentrándose en el punto de vista lógico, se han desarrollado distintos métodos que permiten comprobar automáticamente la validez de un teorema, entre los más conocidos están la resolución con todas sus distintas variantes (*Unit preference, Set-of-suport, Hyperresolution, Model elimination, SL-resolution, Connection graph*, etc.) [Sti-87], y el método de los tableaux con sus diferentes extensiones. Sistemas diseñados para implementar métodos lógicos son por ejemplo AURA [WWL-81] y SETHEO [Bib-90]. El *Computational Logic Theorem Prover* de Boyer y Moore [BM--88] tiene un carácter intermedio entre el aspecto puramente lógico y la orientación humana. Otros sistemas, tales como AUTHOMAT [Bru-80] y la familia de los sistemas LCF [Pau-87], [Con-86], [Gor-89], presentan mayor preocupación por el aspecto humano, tratando de conseguir una construcción interactiva de un árbol de demostración aplicando unas estrategias de demostración basadas en una lógica concreta.

Desde el enfoque de orientación humana podemos considerar diferentes aspectos. Entre ellos son dignos de mención: el manejo de

una base de datos; la existencia de reglas de reescritura y de simplificaciones algebráicas; el uso de sistemas de inferencia naturales; el control de la dirección de búsqueda; la existencia de tipos y su construcción; la utilización de procedimientos que integren el control de la búsqueda con la inferencia; los medios para la interacción hombre-máquina, como puede ser la existencia de un lenguaje claro con el que el usuario se comunique con el demostrador; la posibilidad de utilizar tácticas, ejemplos y contraejemplos; o incluso la capacidad de servirse de patrones usando razonamientos similares a otros y de "aprender" de teoremas ya demostrados.

El enfoque lógico de nuestro sistema ha sido planteado y tratado en los capítulos anteriores, en ellos hemos definido la lógica PLPR que sirve de soporte en el diseño de MIZ-PR y hemos presentado diversos cálculos que permiten mecanizar demostraciones. Algunos aspectos de esta lógica soporte, tales como la construcción de tipos polimórficos, la inducción y el sistema de inferencia natural se han introducido con el fin de conseguir un sistema que incorpore algunos aspectos claves en ATP desde el punto de vista del enfoque humano. Otras características propias de este enfoque se irán poniendo de manifiesto a medida que se vaya conociendo el sistema; es de destacar la naturalidad del lenguaje en el que se escriben las demostraciones.

En la primera sección de este capítulo se da una visión general de MIZ-PR estudiando cómo se mecanizan las demostraciones, qué estructura tiene éstas y qué tareas se ejecutan para mantener el control y verificar las demostraciones. Nuestro sistema heredará las cualidades de la lógica PLPR sobre la que está construido y ciertas reglas de los cálculos para ella definidos, en particular dispone de la inducción de punto fijo.

A continuación, utilizando una gramática en forma de Backus_Naur, se formaliza la sintaxis del lenguaje con el que se definen teorías, se escriben demostraciones y se solicitan facilidades, y se presentan ejemplos que dan una visión de la utilidad y el manejo de MIZ-PR.

El capítulo termina destacando algunos aspectos del programa Prolog que lleva a cabo el control de las demostraciones. El uso de este lenguaje de programación lógica permite considerar al mismo programa como su propia especificación semántica.

1. UNA INTRUDUCCION AL SISTEMA

La organización de MIZ-PR y la forma en que se escriben las demostraciones tiene su origen en la familia Mizar, [TB--85], [PR--88]; de ahí las tres primeras letras de su nombre (MIZ). La razón de que en el diseño de nuestro sistema nos hayamos inspirado en la familia Mizar radica en la naturalidad del lenguaje con que se escriben textos en estos sistemas que permite construir demostraciones propias de la lógica de primer orden de una manera muy próxima a la manera de razonar de los matemáticos.

Los primeros trabajos que se hicieron partiendo de esta familia de sistemas consistieron en una implementación en Prolog de MIZAR-MSE con esquemas y con un comprobador de demostraciones basado en el método de los tableaux. El producto resultante se llamó MIZAR/LOG [Nie-88], [Nie-89] y ha servido como punto de partida en el diseño e implementación del sistema actual.

MIZ-PR ha heredado la naturalidad del lenguaje Mizar, pero además, ha incorporado nuevas ventajas tanto desde el punto de vista lógico como humano. Las principales innovaciones a nivel de la lógica soporte presentes en MIZ-PR son el polimorfismo y la recursión, de ahí las dos últimas letras de su nombre (PR). Desde el punto de vista humano, las funciones recursivas basadas en el μ-operador se especifican en MIZ-PR de la manera habitual en que se especifican programas recursivos [LS--84]. Otras ventajas propias de nuestro sistema son, por ejemplo, el hecho de que las pruebas son validadas a cada paso y la capacidad de que las diferentes técnicas de demostración existentes en MIZ-PR sean utilizadas bien por el usuario para guiar la demostración, o bien se ejecuten de manera automática.

1.1 MECANIZACION DE LAS DEMOSTRACIONES

Tal y como hemos definido nuestra lógica, podemos decir que MIZ-PR soporta una familia de lenguajes objeto basados en PLPR y parametrizados por una signatura. Para cada signatura tenemos la capacidad de crear una teoría que estará compuesta por ciertos objetos que cumplen una serie de propiedades; estas propiedades pueden ser

introducidas de una manera axiomática, o pueden ser demostradas a partir de axiomas o de teoremas ya demostrados utilizando las reglas del cálculo. Sólo los teoremas bien demostrados podrán ser incorporados a la teoría.

La mecanización de las demostraciones está íntimamente ligada con la construcción de teorías; como gran parte de los sistemas de deducción automática, MIZ-PR puede ser considerado como una herramienta para construir y almacenar teorías o utilizar otras previamente definidas por él o predefinidas en el sistema.

En la construcción de cada teoría, es necesario fijar la signatura del lenguaje, para ello se declaran los nombres de ciertos objetos que servirán de elementos básicos en esta construcción y que serán usados al escribir axiomas y proposiciones. El lenguaje de las pruebas se irá presentando conforme se escriben ejemplos y se explican las distintas técnicas de demostración que soporta el sistema.

1.1.1 Declaración de objetos y axiomas

En MIZ-PR existen palabras reservadas que permiten definir los tipos, y los símbolos con tipo de constantes, funciones y predicados correspondientes a una teoría.

Utilizamos la palabra type tanto para definir un constructor de tipo con su aridad (posiblemente igual a cero), como para dar un nombre a un tipo construido. Ejemplos:

```
type nat, 0/2; (por defecto, la aridad de nat se considera igual a cero)
type sum_nat = 0(nat,nat);
```

Las variables de tipo se representan por medio de cadenas de asteriscos, el tipo booleano mediante la palabra bool y el producto de tipos mediante el conectivo ^.

Las variables que aparezcan libres en una teoría, pueden ser declaradas previamente para poder así evitar la escritura de su tipo en la sintaxis de las fórmulas, esta declaración se hace utilizando la palabra var, seguido del nombre de la variable acompañado de su tipo que aparece separado por dos puntos; éste determinará si se trata de una variable de dato o de una variable funcional. Ejemplos:

```
var x:nat, y:*;
var Y:sum_nat->nat;
```

Los símbolos de constante, función y predicado se declaran utilizando las palabras **const**, **func** y **pred**, respectivamente. No obstante, la palabra **func** se utiliza también para asignar un nombre a una expresión funcional, esto hará más cómoda la escritura de los términos. Ejemplos:

```
const 0:nat;
func es_cero:nat->bool, suc:nat->nat, pred:nat->nat;
pred par:nat;
func cero = (lambda x:nat.0);
```

La definición de funciones recursivas se hace mediante la palabra **funrec** seguida de un símbolo de función con su tipo y de una fórmula destinada a especificar los valores que toma dicha función. Por ejemplo, podemos definir la función de Ackerman mediante la expresión

```
funrec Ack:nat^nat->nat
   Ack(x,y) = if es_cero(x) then suc(y)
                 else (if es_cero(y) then Ack(pred(x),suc(0))
                           else Ack(pred(x),Ack(x,pred(y))))
```

La semántica de esta expresión funcional es la misma que la definida en PLPR para la μ-expresión:

```
(μX:nat×nat→nat.(λx:nat y:nat.(if (es_cero x) then (suc y)
          else (if (es_cero y) then (X ((pred x),(suc 0)))
                   else (X ((pred x),(X (x,(pred y)))))))))
```

Los objetos declarados forman la signatura de la teoría y permiten escribir fórmulas. A grandes rasgos, la sintaxis de las fórmulas en el lenguaje de las demostraciones sigue las siguientes normas: las fórmulas predicativas se escriben usando paréntesis cuadrados para englobar el término afectado por el predicado en cuestión. Las aproximaciones se escriben utilizando el símbolo **<=**. Se permite también el uso del símbolo **=** para representar igualdad de términos. Los conectivos de la lógica proposicional se representan mediante los símbolos **not**, **or**, **and**, **implies** e **iff** con su significado intuitivo. Para las cuantificaciones existencial y universal tenemos las palabras **ex** y **forall**, seguidas de **st** en el primer caso y de **holds** en el segundo para separar la declaración de variables de la fórmula a la que cuantifican sin necesidad de paréntesis. Ejemplos:

- `for x:nat holds es_cero(x) = true or ex y:nat st x = suc(y)`
- `for x:*, y:*, z:* holds (x <= y and y <= z) implies x <= z`

- par[x] iff par[suc(suc(x))]

utilizando la sintaxis de PLPR, las fórmulas anteriores equivalen a:

- $\forall x{:}nat\ ((es\_cero\ x) = true \vee \exists y{:}nat\ x = (suc\ y))$
- $\forall x{:}\rho\ \forall y{:}\rho\ \forall z{:}\rho\ ((x \subseteq y \wedge y \subseteq z) \rightarrow x \subseteq z)$
- $(par\ x{:}nat) \longleftrightarrow (par\ (suc\ (suc\ x{:}nat)))$

Cuando se construye una teoría se pueden definir y almacenar axiomas que consisten en fórmulas etiquetadas. Las etiquetas se separan de la fórmula usando el símbolo # y sirven para identificarla cuando ésta es referenciada en una demostración posterior. Los teoremas ya demostrados pueden ser incorporados a la teoría usando una etiqueta nueva.

Podemos, por ejemplo, introducir los siguientes axiomas:

```
A1# es_cero(0) = true;
A2# cero(x) = 0;
A3# for x:nat holds Ack(0,x) = suc(x);
```

Durante el desarrollo de una demostración, es posible etiquetar fórmulas cuya veracidad sirve como paso intermedio para la consecución del objetivo inicial. A cada par etiqueta-fórmula de este tipo se le llama *asignación*, aunque en ocasiones, por abuso del lenguaje, nos referiremos a ellas como axiomas. La diferencia entre un axioma y una asignación es que una asignación tiene un carácter local que sólo tiene vigencia a lo largo de la demostración en la que se introduce, mientras que los axiomas tienen un carácter global dentro de la sesión de trabajo.

1.1.2 Técnicas de demostración

Supongamos que estamos trabajando en una teoría determinada para la que previamente hemos definido una serie de tipos, variables libres, constantes, funciones, predicados y axiomas. Para esta teoría podemos hacer nuevas declaraciones introducir nuevos axiomas, o podemos demostrar un teorema.

Para las fórmulas de primer orden, el texto de una demostración MIZ-PR puede resultar muy semejante a los textos escritos en el lenguaje MIZAR-MSE; sin embargo, puesto que nuestro sistema se basa en una extensión de la lógica de predicados, se admiten técnicas de demostración propias de PLPR, como son la inducción, el λ-cálculo y la generalización de tipos. Además el nuevo sistema tiene una orientación

más interactiva que los sistemas MIZAR, pues el control de la prueba se realiza a cada paso y, por otra parte, permite que a petición del usuario, el demostrador ejecute de manera automática pasos de la demostración eligiendo la técnica más adecuada al objetivo.

Las técnicas de demostración de MIZ-PR se dividen en dos bloques. Los mecanismos del primer bloque trabajan desde el objetivo hacia atrás produciendo la transformación de la tesis inicial y tienen su fundamento en una regla primitiva o derivada del cálculo PTS. Las técnicas del segundo grupo producen demostraciones hacia adelante, consisten en justificar una fórmula haciendo referencia a ciertos axiomas o asignaciones; la demostración se mueve desde estas hipótesis hacia la conclusión a través de las reglas.

Las técnicas hacia atrás son:

- Generalización de variables de tipo.
- Suposición de hipótesis.
- Generalización de variables de dato.
- Descomposición del objetivo en subobjetivos.
- Reducción al absurdo.

Las técnicas hacia adelante son:

- Inducción.
- Axioma de punto fijo.
- β-reducción.
- Referencias a axiomas o teoremas ya probados.

```
pred p:*, q:*, r:*;
(for x:* holds (p[x] implies q[x]) and (q[x] implies r[x]))
   implies (for x:* holds p[x] implies r[x])
proof;
   let sort ct for *;
   assume A$ for x:ct holds
               (p[x] implies q[x]) and (q[x] implies r[x]);
   let a:ct such that B$ p[a];
   thus r[a] by A,B;
end;
```

Figura 1

El usuario tiene la capacidad de guiar una demostración eligiendo la técnica o táctica a seguir, o bien, éstas pueden llevarse a cabo de

una manera automática. Vamos a analizar cada una de las técnicas de demostración estudiando su significado e introduciendo, mediante ejemplos, la sintaxis que permite al usuario su elección.

La demostración de la figura 1 nos servirá de guía para estudiar las técnicas de demostración hacia atrás.

Generalización de variables de tipo. Algunas de las reglas del cálculo PTS trabajan sólo con fórmulas monomórficas; esto no representa un gran problema porque las reglas (TSI) y (TSE) nos permiten pasar de fórmulas monomórficas a fórmulas polimórficas y viceversa. La técnica de demostración que ahora nos ocupa consiste en una mecanización de la regla (TSI) que permite introducir una variable de tipo consiguiendo así una fórmula más genérica. Veamos el uso de este mecanismo en el ejemplo de la figura 1. El objetivo inicial o tesis a demostrar es la fórmula:

```
(for x:* holds (p[x] implies q[x]) and (q[x] implies r[x])) implies
    (for x:* holds p[x] implies r[x])
```

La palabra proof indica que comienza la prueba, el símbolo ';' separa cada uno de los pasos de esta demostración. El primer paso:

```
let sort ct for *;
```

es la sintaxis correspondiente a una generalización de tipos. El efecto que esta técnica tiene sobre la tesis actual es la sustitución de la variable de tipo * por la constante de tipo ct, que ha de ser nueva y que será local para esta demostración. En virtud de la regla (TSI), para demostrar el objetivo inicial, será suficiente con probar la fórmula instanciada:

```
(for x:ct holds (p[x] implies q[x]) and (q[x] implies r[x])) implies
    (for x:ct holds p[x] implies r[x])
```

que constituirá el nuevo objetivo después del uso de la técnica de generalización de tipos.

En MIZ-PR podemos hacer una generalización de tipo múltiple pues la sintaxis de las demostraciones nos permite la utilización consecutiva de esta técnica en un solo paso de la prueba.

Suposición de hipótesis. Esta forma de demostración refleja la técnica de la cancelación de hipótesis propia de nuestro sistema de deducción natural, y tiene su base en el Teorema Fundamental, es

decir, en la regla derivada (FTh). Para suponer el antecedente de una implicación que queremos demostrar usamos la palabra clave **assume**. Entonces, si logramos probar el consecuente, la implicación quedará validada. Siguiendo nuestro ejemplo en el que, después del primer paso de la demostración, el objetivo ha quedado transformado en una fórmula condicional, se observa que el segundo paso de la demostración es:

```
assume A# for x:ct holds
            (p[x] implies q[x]) and (q[x] implies r[x]);
```

que deja reducido el objetivo a la fórmula

```
for x:ct holds p[x] implies r[x].
```

Como hemos visto, la técnica de suposición de hipótesis se desencadena con la palabra **assume**. Además de esta asumpción simple en la que suponemos una sola hipótesis, la sintaxis admite también otras formas de asumpción como son la colectiva y la existencial que, aunque más sofisticadas, siempre pueden reducirse a la primera porque se fundamentan en reglas derivadas de (FTh). La asumpción colectiva consiste en suponer varias condiciones a la vez. Si, por ejemplo, A1, A2, A3 y B son fórmulas, para probar (A1 and A2 and A3) implies B, al usar **assume** A1,A2 el objetivo se transforma en A3 implies B. Por tanto, una asumpción colectiva equivale a un uso consecutivo de la técnica de suposición de hipótesis.

La asumpción existencial equivale a tomar como hipótesis una fórmula existencial. Se utiliza la sintaxis **given** C **such that** F; siendo F una sucesión de fórmulas y C una serie de declaraciones de constantes nuevas con carácter temporal que satisfacen las fórmulas de F. Esta sintaxis equivale a la agrupación de dos técnicas de demostración: la suposición de hipótesis y la justificación por referencia que estudiaremos más tarde. Por ejemplo, la expresión

```
given c:ct such that L# p[c];
```

donde c:ct es una constante nueva, equivale a los dos pasos

```
assume H# ex x:ct st p[x];
L# p[c] by H;
```

Generalización de variables de dato. Se trata de demostrar una fórmula universal probándola para una constante auxiliar o variable genérica del tipo adecuado. Esta técnica corresponde por tanto, a la regla (VI). Se utiliza la palabra **let** para indicar la utilización de

esta técnica y para declarar una constante nueva que tendrá un carácter local.

Continuando con la demostración de la figura 1, sabemos que después de los dos primeros pasos, el objetivo a demostrar es la fórmula

for x:ct **holds** p[x] **implies** r[x]

Supongamos que el siguiente paso de la demostración fuera

let a:ct;

Este texto hace que el sistema utilize la técnica de generalización de variables de dato, que produce una instanciación del objetivo actual consistente en una fórmula universal, en este caso, nuestro objetivo se transformará en la fórmula

p[a] **implies** r[a] (donde a es una constante de tipo ct)

Al obtener una implicación como objetivo podemos pensar en continuar la demostración utilizando de nuevo el item **assume**, es decir la técnica de suposición de hipótesis. Sin embargo, como muestra el tercer paso del ejemplo que seguimos, el lenguaje de las pruebas es lo suficientemente rico para permitirnos agrupar los pasos let a:ct; y **assume** B$ p[a]; en uno sólo, escribiendo la expresión

let a:ct **such that** B$ p[a];

que desencadena las dos técnicas implícitas y reduce nuestro objetivo a probar la fórmula r[a]. Desde el punto de vista lógico este uso del lenguaje se fundamenta en una regla derivada de las reglas (∀I) y (FTh).

El item let puede usarse también para declarar varias constantes a la vez, lo que resulta equivalente a agrupar en un solo paso el uso consecutivo de la técnica de generalización de variables de dato y, con ello, de la regla (∀I) produciéndose la eliminación de tantos cuantificadores como constantes se declaren.

Descomposición del objetivo en subobjetivos. Esta forma de demostración corresponde a la regla (∧I) que permite considerar la demostración de una fórmula conjuntiva como la demostración de una lista de fórmulas que son las componentes de la fórmula de partida; estas componentes constituirán por tanto los nuevos subobjetivos. De forma similar, cualquier fórmula no conjuntiva puede tratarse como una lista con un único elemento o subobjetivo. Cuando se demuestra un

subobjetivo de esta lista, éste se concluye utilizando la palabra thus que produce una transformación en la tesis actual consistente en eliminar dicha componente.

Supongamos n fórmulas A1,...,An con n ≥ 1, para demostrar la fórmula A1 and A2 and...and An bastará con demostrar Ai para todo 1≤i≤n; una prueba que concluya con thus Ai dejará reducido el objetivo a la fórmula A1 and...Ai-1 and Ai+1 and...and An. Al contrario de lo que ocurre en MIZAR-MSE, en nuestro sistema, el orden en el que se vayan probando los subobjetivos o componentes es irrelevante.

Volviendo al ejemplo de la figura 1; recordemos que en el momento actual, después de recorrer los tres primeros pasos, nuestro objetivo es probar la fórmula r[a] siendo a una constante de tipo ct. Haciendo referencia a los axiomas A y B podemos concluir

```
thus r[a] by A,B;
```

produciéndose la supresión de esta componente del objetivo actual, que por estar formado por este único subobjetivo, queda reducido al vacío permitiendonos finalizar la prueba con éxito.

Podemos resumir diciendo que el item thus significa la acción de eliminar una componente de la tesis actual que puede ser la última. Para que esto se realice correctamente, dicha componente o subobjetivo debe ser probada previamente para lo cual se hace una nueva subprueba, que forma parte de la demostración general, utilizando de forma recursiva las técnicas propias de MIZ-PR. En el caso del ejemplo, la fórmula r[a] se infiere por referencia a los axiomas A y B; esta técnica será estudiada más tarde.

Reducción al absurdo. La base de este tipo de pruebas son las reglas (¬I) y (¬E), es decir, hacemos una prueba indirecta del objetivo suponiéndolo falso y llegando a un absurdo, representado en MIZ-PR por la palabra contradiction, que es semánticamente equivalente a la conjunción de una fórmula cualquiera y su negación. De esta forma, para demostrar una fórmula A podemos suponer su negación utilizando assume not A y si concluimos contradiction el objetivo quedará probado.

Como vemos en la figura 2, la demostración de la figura 1 puede escribirse usando esta técnica en alguno de sus pasos.

```
pred p:*, q:*, r:*;
(for x:* holds (p[x] implies q[x]) and (q[x] implies r[x]))
   implies (for x:* holds p[x] implies r[x])
proof;
   let sort ct for *;
   assume A$ for x:ct holds
               (p[x] implies q[x]) and (q[x] implies r[x]);
   let a:ct such that B$ p[a];
   assume C$ not r[a];
   thus contradiction by A,B,C;
end;
```

Figura 2

Desde el punto de vista lógico, cada uno de los procesos correspondientes a una técnica de demostración de las anteriormente descritas corresponde a una regla de inferencia usada hacia atrás partiendo del objetivo hasta llegar a las hipótesis, por ello decimos que la demostración es *backwards*. Desde el punto de vista de la resolución del problema, cada proceso descompone el objetivo en objetivos más simples, luego es una simplificación. Podemos considerar que las demostraciones escritas utilizando MIZ-PR están orientadas hacia el objetivo en el sentido de que a la vista de éste se elige una técnica de demostración.

En MIZ-PR, siguiendo a Mizar, llamamos **thesis** al objetivo que queda por probar en cada momento de la demostración. Esta palabra representa, por tanto, una fórmula que va variando a lo largo de la demostración conforme el objetivo inicial es transformado. En 1.2.1, estudiaremos con más detalle los efectos que las diferentes técnicas de demostración producen en el objetivo.

En la figura 3 aparece recuadrado el valor interno, expresado en el lenguaje PLPR, que va tomando la palabra **thesis**, después de cada paso de la demostración de la figura 1.

Al contrario de lo que sucede con las técnicas explicadas hasta ahora, los mecanismos englobados en el segundo bloque pueden considerarse técnicas *forwards* en el sentido de que a partir de unos axiomas o teoremas previos se infiere una proposición. En lo que sigue se describen las distintas clases de justificación (hacia adelante)

existentes en nuestro sistema.

```
pred p:*, q:*, r:*;
(for x:* holds (p[x] implies q[x]) and (q[x] implies r[x]))
   implies (for x:* holds p[x] implies r[x])
proof;
   let sort ct for *;
┌──────────────────────────────────────────────────────────────┐
│ ∀x:ct (((p x) → (q x)) ∧ ((q x) → (r x))) →                  │
│                                  ∀x:ct ((p x) → (r x))       │
└──────────────────────────────────────────────────────────────┘
   assume A# for x:ct holds
                (p[x] implies q[x]) and (q[x] implies r[x]);
┌──────────────────────────────────────────────────────────────┐
│                        ∀x:ct ((p x) →(r x))                  │
└──────────────────────────────────────────────────────────────┘
   let a:ct such that B# p[a];
┌──────────────────────────────────────────────────────────────┐
│                              (r a:ct)                        │
└──────────────────────────────────────────────────────────────┘
   thus r[a] by A,B;
┌──────────────────────────────────────────────────────────────┐
│                                                              │
└──────────────────────────────────────────────────────────────┘
end;
```

Figura 3

Inducción. Esta técnica se fundamenta en la regla (IND) de inducción de punto fijo para fórmulas continuas del cálculo $\mathcal{PFC}$; el árbol de derivación con su primera hipótesis cancelada que aparece en las premisas de (IND) se sustituye aquí por una implicación, con lo que en realidad la inducción en MIZ-PR se basa en la regla (IND').

Un problema común en ATP cuando se utilizan esquemas de inducción es encontrar la variable sobre la que se realiza dicha inducción. En nuestro sistema el usuario tiene la capacidad de guiar estas demostraciones; para ello comienza por probar las premisas de la regla de inducción y utilizando la palabra induction concluye la fórmula, en cuyo interior aparece una definición recursiva. El sistema entonces se encarga de encontrar la variable y con ello el patrón de la inducción, teniendo como datos las premisas y la conclusión; comprueba también que este patrón corresponde a una fórmula continua y confirma que las hipótesis de la justificación corresponden a las premisas de la regla (IND').

Consideremos las declaraciones de la figura 4, en la que se definen las funciones *null*, *cons*, *head* y *tail* propias de de la teoría de listas de tipo polimórfico; el tipo *nat* se usa para representar a los naturales. Definimos la concatenación de listas y la longitud de una lista utilizando definiciones recursivas.

```
type list/1, nat/0;
const 0:nat;
func suc:nat->nat, +:nat^nat->nat;
var X:list(*)->nat;
func null:list(*)->bool,
   cons:*^list(*)->list(*),
   head:list(*)->*,
   tail:list(*)->list(*);
funrec append:list(*)^list(*)->list(*)
   append(x,y) = if null(x) then y else
                          cons(head(x),append(tail(x),y));
funrec lng:list(*)->nat
   lng(x) = if null(x) then 0 else suc(lng(tail(x)));
```

Figura 4

Supongamos que para esta teoría queremos probar el teorema

```
for x:list(*), y:list(*) holds lng(append(x,y)) = +(lng(x),lng(y)).
```

Aplicando la regla de inducción de punto fijo para la función recursiva *lng*, si probamos los axiomas A1 y A2 siguientes:

```
A1# for x:list(*), y:list(*) holds
      (bot:list(*)->nat)(append(x,y)) =
                  +((bot:list(*)->nat)(x),(bot:list(*)->nat)(y));
A2# (for x:list(*), y:list(*) holds X(append(x,y)) = +(X(x),X(y)))
   implies (for x:list(*), y:list(*) holds
      if null(append(x,y)) then 0 else suc(X(tail(append(x,y))))
       = +(if null(x) then 0 else suc(X(tail(x))),
           if null(y) then 0 else suc(X(tail(y)))));
```

podemos concluir la demostración diciendo

```
thus thesis induction A1,A2;
```

para indicar que el teorema que queríamos probar se deduce por inducción basándonos en las premisas A1 y A2.

Axioma de punto fijo. A1 utilizar la palabra fix, se hace

referencia al axioma de punto fijo (FIX) del cálculo $\mathcal{PVS}$.

Esta técnica permite probar la iguadad de dos términos, en uno de los cuales, una función recursiva ha sido sustituida por la expresión funcional que la define. Por ejemplo, si el factorial de un número natural ha sido definido por medio de la declaración

```
funrec fact:nat->nat
   fact(x) = if es_cero(x) then suc(0) else por(x,fact(pred(x)));
```

el símbolo fact representará en lo sucesivo al μ-operador $(\mu X{:}nat \rightarrow nat.M)$ donde $M{:}nat \rightarrow nat$ es la expresión funcional siguiente

```
(λx:nat.(if (es_cero x) then (suc 0) else (por (x,(X (pred x))))))
```

Del axioma de punto fijo pueden deducirse igualdades del tipo

```
∀y:nat ((μX:nat→nat.M) y) = (λx:nat.(if (es_cero x) then
          (suc 0) else (por (x,((μX:nat→nat.M) (pred x)))))) y)
```

que por las reglas del lambda-cálculo puede reducirse a la expresión

```
∀y:nat (if (es_cero y) then (suc 0) else
                              (por (y,((μX:nat→nat.M) (pred y)))))
```

Recordando que $(\mu X{:}nat \rightarrow nat.M)$ es precisamente fact, utilizando MIZ-PR, podemos expresar el razonamiento anterior como sigue:

```
for y:nat holds fact(y) = if es_cero(y) then suc(0) else
                                        por(y,fact(pred(y))) by fix;
```

β-reducción. Esta técnica es una mecanización del uso de la regla (ABS) del cálculo $\mathcal{PVS}$. Esta regla tiene una única premisa que debe ser probada previamente a la ejecución de la reducción. La sintaxis para usar esta técnica es señalar la lambda expresión a la que se le aplica la β-reducción utilizando la palabra reducing y utilizar la palabra by para indicar la etiqueta correspondiente a la premisa de la regla.

Sirva como ejemplo la demostración de la figura 5.

```
type nat;
const 0:nat;
func es_cero:nat->bool;
func cero = (lambda x:nat.0);
for x:nat holds not x <= bot implies cero(x) = 0
proof;
   let n:nat such that A$ not n <= bot;
   thus cero(n) = 0 reducing cero by A;
end;
```

Figura 5

Referencia a axiomas o teoremas ya demostrados. Esta técnica de demostración consiste en inferir una fórmula haciendo referencia a axiomas o asignaciones sin especificar las reglas de deducción utilizadas; siendo el sistema el encargado de encontrar una derivación que compruebe la corrección de dicha inferencia. La sintaxis utilizada en este tipo de demostraciones consiste en escribir la palabra **by** seguida de las *referencias* que sirven de hipótesis de la deducción. Las referencias pueden ser etiquetas de axiomas o asignaciones, o bien las palabras **order**, **bot-order** o **bool**.

El empleo de **order**, **bot-order** o **bool** supone añadir al conjunto de premisas, los axiomas de orden referentes a las propiedades del orden parcial, del elemento *bottom* y de los booleanos, respectivamente, adecuados al objetivo y a las premisas. La corrección de esta acción se basa en que todo axioma de orden es una fórmula válida para PLPR. Los axiomas de orden adecuados a la teoría de trabajo, forman parte del conjunto de axiomas de dicha teoría aunque no aparezcan de forma explícita.

En la figura 6 aparece una variación de la demostración de la figura 1; la variación consiste en dar un paso intermedio antes de llegar a la conclusión final, que consiste en una justificación por referencia a asignaciones, esto no altera el objetivo, y en ambas demostraciones, la fórmula r[a] necesaria para concluir el teorema se prueba usando esta misma técnica, aunque los axiomas referenciados varían de una a otra.

Cuando el usuario utiliza este método de demostración por referencia, da por supuesto que existe un árbol de derivación que permite concluir una fórmula partiendo de las premisas referenciadas, con el fin de comprobar la corrección de este paso de la demostración, el sistema debe ser capaz de mecanizar dicha inferencia. En 1.2.2 veremos de que forma tiene lugar esta mecanización.

El tercer paso de la demostración de la figura 6 (**let**...**such that**...) es un ejemplo de composición de técnicas. También hemos comentado otros ejemplos de sintaxis para indicar la aplicación encadenada de varias técnicas como es el uso del ítem **given** para indicar una suposición existencial. Pero como ya hemos indicado, todas estas otras otras posibilidades sintácticas pueden descomponerse en la ya descritas, no incorporando nuevas técnicas de demostración.

```
pred p:*, q:*, r:*;
(for x:* holds (p[x] implies q[x]) and (q[x] implies r[x]))
   implies (for x:* holds p[x] implies r[x])
proof;
   let sort ct for *;
   assume A# for x:ct holds
                (p[x] implies q[x]) and (q[x] implies r[x]);
   let a:ct such that B# p[a];
   C# q[a] by A,B;
   thus r[a] by A,C;
end;
```

Figura 6

En lo sucesivo, identificaremos las técnicas de demostración de MIZ-PR con los items **let sort**, **assume**, **let**, **thus**, **contradiction**, **induction**, **fix**, **reducing** y **by**, respectivamente. Estos items forman parte del lenguaje de las pruebas. El paralelo en LCF, de los items correspondientes a una técnica hacia atrás es el uso de las tácticas; igualmente, la sintaxis de MIZ-PR que permite agrupar en un solo paso varias técnicas tiene como paralelo en LCF las *tacticals*. Sin embargo, existen claras diferencias entre uno y otro sistemas. En LCF, las tácticas se programan mediante el metalenguaje ML, esto tiene la ventaja, de que el usuario puede incrementar el número de tácticas y no es dependiente de las que ya están programadas en el sistema, como es el caso de MIZ-PR. Por contra, la ventaja de MIZ-PR es que en este sistema, la interacción entre el usuario y la máquina, se apoya en un lenguaje que representa más la forma natural de hacer demostraciones, en la que no se indica de forma explícita la regla de derivación utilizada, mientras que por el contrario el uso de una táctica en LCF representa la aplicación explícita de una regla.

1.1.3 La estructura de las demostraciones.

Al igual que toda demostración realizada utilizando un sistema de deducción natural, una prueba escrita en MIZ-PR tiene la estructura de un árbol cuya raíz es el objetivo, cuyas ramas son reglas y cuyas hojas son axiomas o suposición de hipótesis que luego pueden eliminarse. Cada subárbol será una prueba de la fórmula que tenga en

la raíz que a su vez es un subobjetivo del nodo padre.

Hemos visto que las técnicas de demostración que utiliza nuestro sistema pueden conducir la prueba hacia atrás o hacia adelante; la elección de una técnica repercutirá, por tanto, en el modo en que se construye el árbol asociado a la prueba. Para los pasos de la demostración en los que se utiliza una técnica *backwards*, la construcción del árbol es hacia arriba desde la raíz. Sin embargo, cuando demostramos un subobjetivo haciendo referencia a unos axiomas, incorporamos a nuestro árbol de prueba un subárbol que ha sido construido desde abajo, es decir, desde sus hojas.

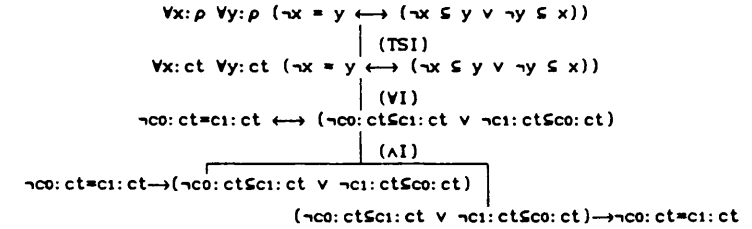
En los ejemplos de comprobaciones escritas en MIZ-PR que se han presentado hasta ahora, se puede observar cómo para los teoremas de los cuales se especifican varios pasos de su demostración, esta comienza con la palabra **proof** y se escribe la palabra **end** cuando se concluye. En el interior de esta demostración puede aparecer de nuevo la palabra **proof** para indicar el comenienzo de la prueba de un subobjetivo o paso intermedio de la inicial; esta nueva demostración finalizará con su propio **end**. Surgen así lo que en Mizar se llaman demostraciones anidadas que equivalen a subárboles dentro del árbol asociado a la demostración inicial, pudiendo albergar estos subárboles nuevos anidamientos en su construcción. Las etiquetas y constantes definidas dentro de un anidamiento tienen un carácter local.

Veamos un ejemplo en el que apoyándonos en el hecho de que una equivalencia o doble implicación es la conjunción de dos fórmulas condicionales, probamos una fórmula de este tipo utilizando una demostración para cada una de sus dos componentes.

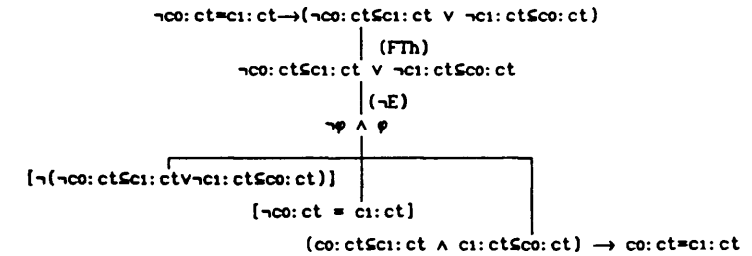
```
for x:*, y:* holds not x = y iff (not x <= y or not y <= x)
   proof;
      let sort ct for *;
      let x:ct, y:ct;
      thus not x = y implies (not x <= y or not y <= x)
      proof;
         assume A# not x = y;
         assume B# not (not x <= y or not y <= x);
         thus contradiction by A,B,order;
      end;
```

```
    thus (not x <= y or not y <= x) implies not x = y
    proof;
      assume A$ not x <= y or not y <= x;
      assume B$ x = y;
      thus contradiction by A,B;
    end;
  end;
```

Escribamos desde la raíz, el árbol de prueba correspondiente a este texto. Los nombres que aparecen en las ramas corresponden a las reglas aplicadas para pasar de los hijos, que aparecen debajo, a los padres, que aparecen encima.

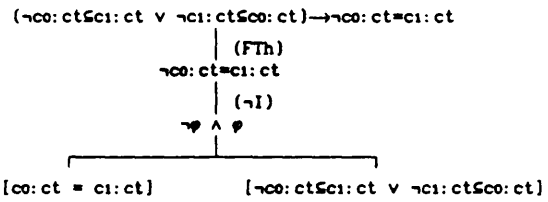


Tanto la rama izquiera de este árbol como la derecha dan lugar a nuevos subárboles que desarrollamos a continuación:



Nótese, que la hoja de más a la derecha corresponde a la instanciación de un axioma de orden, en concreto representa la propiedad antisimétrica de todo orden parcial, y el paso en el que se deduce una contradicción sin especificar las reglas utilizadas corresponde a la demostración hecha por referencia (by). Es fácil probar que existe una derivación, es decir, un subárbol que justifica este paso, sabemos que el encargado de comprobarlo es el sistema.

Desarrollemos ahora la rama derecha del árbol de la prueba.



Al igual que en caso anterior, el último paso de la demostración indica la construcción automática de un subárbol en el sentido inverso al que seguimos, que no aparece explícitamente, únicamente nos basamos en su existencia, comprobada por el sistema, para simplificar la construcción del árbol global y por tanto el texto de la demostración.

1.2 CONTROL DE LAS DEMOSTRACIONES

El usuario de MIZ-PR tiene la capacidad de guiar la demostración escribiendo un texto que lleva implícito el uso de las técnicas existentes, el sistema por su parte se encarga en estos casos de validar la corrección de la prueba y de mantener el control de la misma; esto se realiza por medio de un programa modular escrito en Prolog.

Llamaremos *contexto* o *entorno* a un par formado por una lista de asignaciones y una fórmula que representa el objetivo a probar o tesis. Una teoría determina un contexto inicial cuya lista de asignaciones está formada por los axiomas de la teoría y cuya tesis es vacía. Este contexto inicial se mantiene mientras dura una sesión de trabajo pudiendo ser aumentada la lista de asignaciones si se introducen nuevos axiomas o teoremas una vez demostrados. Al comenzar la demostración de un teorema se crea un nuevo sub-contexto que irá evolucionando a lo largo de ésta, de manera que en cada momento un entorno es una caracterización del estado actual de la demostración a la que representa. Una prueba anidada constituirá un contexto local que desaparecerá si ésta finaliza con éxito. Un control riguroso de la demostración se consigue manteniendo una estructura y actualización adecuada de los contextos.

La parte del sistema dedicada a controlar la evolución de los contextos creados a lo largo de una demostración es lo que llamamos *reasoner*. A continuación estudiaremos con detalle las tareas que este módulo realiza para comprobar la corrección de los ítems usados por el usuario para validar un teorema. Veremos después cómo el espíritu del *reasoner* sirve para diseñar otro módulo capaz de automatizar ciertos pasos de la demostración sin necesidad de la intervención del usuario.

1.2.1 El Reasoner

Este módulo constituye el núcleo del sistema, su tarea consiste en encaminar el programa de verificación o control de las pruebas y en mantener una estructura de contextos que refleje la estructura de una demostración.

Podemos definir la *estructura de contextos* de forma recursiva de la siguiente manera: una estructura de contextos es o bien un entorno inicial o bien la concatenación de un contexto local con una estructura de contextos.

El *reasoner* modificará la estructura de los contextos de alguna de las siguientes formas:

A) Creando o eliminando un contexto local asociado a una demostración anidada.

B) Añadiendo nuevas fórmulas etiquetadas ya demostradas a la lista de asignaciones para que éstas puedan ser referenciadas más tarde.

C) Transformando la tesis inicial.

Veamos cómo y cuándo se producen cada uno de estos cambios.

A) La creación de un contexto local se produce cuando se comienza la demostración de un teorema o subobjetivo. En su origen todo contexto local o subcontexto está formado por una lista de asignaciones que es en principio vacía y por la tesis que comienza siendo el teorema o subobjetivo que se quiere probar. La eliminación de dicho subcontexto tiene lugar cuando se finaliza la prueba asociada; esta prueba será correcta, pudiéndose llevar a cabo la desaparición del contexto, cuando en el momento en que el usuario de por finalizada la demostración (escribiendo la palabra end), no nos quede nada por probar, es decir, la tesis actual asociada al entorno en cuestión será vacía. Por supuesto, después de una eliminación, las asignaciones asociadas no podrán volver a ser referenciadas.

B) La lista de asignaciones de un entorno puede ser modificada por incorporación de fórmulas etiquetadas previamente probadas, a las que podrá hacerse referencia a lo largo de la demostración asociada a este entorno. En cada momento, las asignaciones accesibles son tanto las del contexto actual como las que forman parte de los contextos que han sido creados con anterioridad y no han sido todavía cerrados o eliminados. Insistimos en que para añadir una asignación a una lista, la fórmula correspondiente ha debido de ser validada previamente; esto se comprueba de una manera recursiva utilizando el módulo *reasoner* gracias a la forma también recursiva de la estructura de contextos.

C) Para poder controlar la corrección de una demostración, es necesario saber en cada paso y para cada subárbol de la demostración, qué nos queda por probar. Mediante la palabra thesis se expresa, dentro de cada demostración anidada, o entorno proof, el objetivo actual. Por lo anterior, y puesto que cada demostración anidada se asocia internamente con un contexto local, el sistema debe ser capaz de identificar en cada sub-contexto la palabra thesis con una fórmula que ha ido evolucionando desde su origen al utilizarse las distintas técnicas de demostración hacia atrás.

En [Mos-85] aparece un estudio minucioso de cómo se reconstruye la tesis en un sistema MIZAR-MSE a la vista de cada item o técnica utilizada en una demostración. Esta misma idea de descomposición de la tesis inicial y posible reconstrucción de la misma queda reflejada en MIZ-PR cuando se utilizan métodos hacia atrás, y tiene su fundamento en reglas del cálculo $\mathcal{PVS}$. Al estudiar estas técnicas de demostración fuimos viendo la repercusión que tienen en el objetivo como consecuencia de la regla de inferencia en la que se apoya cada una de ellas. Un control adecuado de la transformación del objetivo será necesario para que el sistema detecte la corrección de cada paso dado por el usuario, y valide una demostración que el usuario da por finalizada, si realmente no queda nada por probar, siendo el objetivo vacío.

Los pasos de una demostración escrita por el usuario pueden, como vimos, identificarse con uno o varios items del lenguaje de las pruebas que a su vez representan el uso de una técnica de demostración, el *reasoner* recibe información (en forma de término Prolog) de cada uno de estos items y desencadena las acciones

correspondientes a cada técnica. Hagamos un repaso de estas acciones.

Cuando se utiliza la técnica de generalización de variables de tipo, la única acción que se produce es un cambio en la tesis del entorno actual que consiste en sustituir dentro de ella la variable de tipo indicada por una constante de tipo nueva.

La técnica de suposición de hipótesis modifica tanto la tesis como la lista de asignaciones del contexto actual. Para que el uso de esta técnica sea correcto y pueda continuar el proceso, el objetivo en el momento de ser utilizada debe ser una implicación cuyo antecedente coincide con la fórmula supuesta, produciéndose entonces la eliminación de dicho antecedente en la tesis actual y la incorporación de éste a la lista de asignaciones.

El uso de una generalización de variables de dato tendrá éxito cuando la tesis actual sea una cuantificación universal que afecta a una variable cuyo tipo coincide con el de la nueva constante introducida. En este caso se produce una transformación de este objetivo que consiste en eliminar la cuantificación universal y sustituir la variable cuantificada por la nueva constante.

Si la prueba del objetivo se realiza probando cada uno de sus componentes, se desencadenan varias acciones. Por un lado, debe comprobarse la corrección de la prueba que justifique la validez del subobjetivo produciendose una llamada recursiva al proceso de control; en caso de que esta subprueba finalice con éxito, se eliminará esta componente de la tesis y será añadida a la lista de asignaciones. Puesto que la tesis siempre puede ser considerada como una conjunción, este proceso la reduce permitiendo concluir la demostración cuando se elimine la última componente.

Las demostraciones por reducción al absurdo se inician correctamente cuando la fórmula supuesta equivale a la negación de la tesis actual. En caso de que esto ocurra, el objetivo se transforma en la representación interna de la contradicción y la fórmula supuesta se añade a la lista de asignaciones.

Cuando se utiliza una técnica hacia adelante, la tesis actual no se modifica, por otro lado, la comprobación de la corrección de este tipo de demostraciones se lleva a cabo por diferentes módulos llamados desde el *reasoner*, este último sólo se encarga de incorporar la fórmula, una vez demostrada, al conjunto de asignaciones del contexto

actual. Hay que tener también en cuenta, que cuando en la inferencia de una fórmula se hace referencia a unas premisas, éstas deben ser localizadas en las listas de asignaciones vigentes; la búsqueda se hace de abajo arriba, es decir, a partir del contexto actual hacia el resto de entornos concatenados en la estructura de contextos hasta llegar al entorno inicial que contiene los axiomas de la teoría.

1.2.2 El Checker

Manteniendo la terminología de los sistemas Mizar, nos referimos a la parte del sistema encargada de la comprobación de las demostraciones por referencia con el nombre de *checker*. El *checker* de MIZ-PR está constituido por dos módulos bien diferenciados; el primero para tratar las propiedades de la igualdad y el segundo para el resto de los razonamientos.

La mecanización de las pruebas relativas a la igualdad se ha conseguido mediante la implementación del mecanismo de Zukowski [Zuk-87] que permite deducir igualdad de términos a partir de un conjunto de ecuaciones, generando clases de equivalencia. Este conjunto de ecuaciones está formado por las fórmulas referenciadas como premisas que sean igualdades o cuantificaciones universales de igualdades.

La parte principal del checker está constituida por su segundo módulo que es una implementación parcial del algoritmo de los tableaux para PLPR presentado en el capítulo II. Los tableaux construidos automáticamente no tienen en cuenta las clases ω-alfa y ω-beta; las fórmulas pertenecientes a estas clases son consideradas como básicas, con lo que se pierde la completitud del método pero no la corrección. Por otra parte, con el fin de ganar eficiencia en la construcción de un tableau, y puesto que las igualdades se comprueban separadamente, el sistema sólo incorpora los axiomas de orden correspondientes al orden parcial, a las propiedades del elemento *bottom* o a los booleanos, cuando las palabra **order**, **bot-order** o **bool** forman parte de las referencias en una justificación.

En la búsqueda de un tableau cerrado para un conjunto de fórmulas se comienza trabajando con las fórmulas de la lógica proposicional; de no poderse cerrar el tableau simplemente con ellas, se instanciarán fórmulas gamma y delta hasta consegir un árbol que no sobrepase una

cierta profundidad calculada previamente en función del número de términos construidos con variables no libres y del número de cuantificaciones existente en el conjunto de fórmulas inicial. En 3.2.4 aparece una descripción del programa que mecaniza este algoritmo.

Puesto que el programa resultante lógicamente no es un método de deducción completo, si el sistema contesta negativamente en la comprobación de una justificación por referencia a axiomas, no podrá asegurarse la no existencia de una derivación. El ususario deberá intentar descomponer este paso de la demostración en pasos intermedios.

1.2.3 La Inducción y el axioma de punto fijo

Cuando el usuario define una función recursiva utilizando la palabra reservada funrec, la especificación dada sirve para construir una representación interna que identifica el símbolo de función empleado con una μ-expresión escrita como un término Prolog. Para comprobar si una demostración por inducción está bien realizada, el sistema se servirá de esta representación que permitirá encontrar el patrón de la fórmula sobre la que se realiza la inducción además permitirá reconocer si esta fórmula es continua. El demostrador reconcerá como fórmulas continuas un gran número de expresiones sintácticas en particular las que provienen de las construcciones que aparecen en la proposición §III-3.1.4.

En cuanto al punto fijo, la representación interna de las μ-expresiones, $(\mu X.M:\tau_1\rightarrow\tau_2)$, permite al sistema la fácil comprobación de la igualdad de dos términos cuando en uno de ellos aparece una expresión de este tipo, mientras que en el otro aparece en su lugar el resultado de sustituir en $M:\tau_1\rightarrow\tau_2$, la variable funcional $X:\tau_1\rightarrow\tau_2$ por $(\mu X.M:\tau_1\rightarrow\tau_2)$.

1.2.4 Deducción automática e interacción del usuario

En las demostraciones que se han escrito hasta ahora, no se ha hecho ninguna referencia a pasos automatizados, sino que han sido guiadas por completo por el usuario; de tal modo que éste, valiéndose de un lenguaje propio de MIZ-PR, escribe demostraciones que el sistema se encarga de comprobar. Esto nos ha permitido especificar cuáles son

las tareas que el sistema ejecuta para cada una de las reglas de deducción, implícitas en los items elegidos por el usuario para guiar la demostración. Estas mismas tareas habrán de llevarse a cabo cuando una demostración se ejecute de manera automática siendo el sistema el encargado de elegir la técnica de demostración a seguir.

En un momento cualquiera de la demostración, el usuario puede optar por demostrar una fórmula (posiblemente la tesis actual) de una manera totalmente automática, esto se indica con la palabra automat, y es el sistema el encargado de elegir, a partir de este momento, las técnicas de demostración necesarias para conseguir el objetivo y de contestar afirmativamente si consigue finalizar con éxito la prueba; en caso contrario volverá al punto de partida. Puesto que el sistema no es completo, una negativa en la comprobación automática de una fórmula a partir de unas premisas no quiere decir necesariamente que no exista una demostración de dicha fórmula.

Para conseguir la demostración de un objetivo de forma automática, MIZ-PR intenta aplicar alguna de las técnicas de demostración existentes, con excepción de la inducción. El orden en el que se ensayan las distintas técnicas es fijo; el sistema comienza probando la técnica de generalización de tipos convirtiendo la fórmula que constituye el objetivo en una instancia monomórfica de dicha fórmula. A continuación, visto el objetivo como una conjunción, se prueba la técnica de comprobación de cada uno de los subobjetivos o componentes de la fórmula de partida. El siguiente lugar en este orden lo ocupa la técnica de generalización de variables de dato siempre que el subobjetivo actual sea una fórmula universal, y si éste es una implicación, se prueba la asumpción de hipótesis. Si las dos técnicas anteriores no tuvieran éxito, se intenta realizar una β-reducción de alguna λ-expresión contenida en la tesis; igualmente es posible, gracias al axioma de punto fijo, sustituir un μ-operador por la expresión funcional que lo define aplicada a dicho μ-operador. Finalmente se construye un tableau que refute el subobjetivo actual a partir de las premisas existentes en el entorno actual.

Este proceso de aplicación de reglas de deducción se repite con cada uno de los subobjetivos en que un objetivo inicial se haya descompuesto; en caso de fallo se produce *backtracking*, la implementación Prolog que soporta el sistema permite que esta búsqueda

hacia atrás se realice de una manera automática.

Si después de intentar aplicar las técnicas anteriores no se consiguiera demostrar el objetivo inicial, se intenta construir un tableau añadiendo a las asignaciones del contexto actual los axiomas de orden adecuados.

2. UTILIZACION

Esta sección, dedicada a la utilización del sistema MIZ-PR, comienza con una formalización de la sintaxis del lenguaje de las demostraciones, este lenguaje ha sido utilizado en los ejemplos introducidos hasta ahora. El segundo primer apartado de la sección está dedicado a mostrar ejemplos de demostraciones que han sido comprobadas por el sistema.

2.1 EL LENGUAJE DE LAS DEMOSTRACIONES

La sintaxis del lenguaje utilizado para declarar objetos y escribir demostraciones en el sistema MIZ-PR puede formalizarse mediante la gramática que aparece a continuación, donde los paréntesis cuadrados, [ ], indican que el item es opcional, mientras que los corchetes, { }, indican cero, una o varias ocurrencias.

```
Texto = {Definicion| Axioma| Demostracion| Facilidad} exit

Definicion = {Def-tipo| Def-constante| Def-variable| Def-funcion|
        Def-predicado}
Def-tipo = type Identif-tipo ["/" Aridad]
        {"," Identif-tipo ["/" Aridad]} ";"
        | type Identif-tipo "=" Tipo ";"
Def-constante = const Simbolo-cte-tipado {"," Simbolo-cte-tipado} ";"
Def-variable = Def-var-dato| Def-var-funcion
Def-var-dato = var Var-dato-tipada {"," Var-dato-tipada} ";"
Def-var-funcion = var Var-funcion-tipada {"," Var-funcion-tipada} ";"
Def-funcion = func Simbolo-func-tipado {"," Simbolo-func-tipado} ";"
        | func Identif-funcion "=" Expresion-funcional ";"
        | funrec Simbolo-func-tipado Formula ";"
Def-predicado = pred Simbolo-pred-tipado {"," Simbolo-pred-tipado} ";"

Axioma = Etiqueta "#" Formula ";"
Proposicion = [Etiqueta "#"] Formula

Demostracion = Proposicion Derivacion ";"
```

```
Derivacion = Prueba| Justificacion| automat

Prueba = proof ";" Razonamiento end
Razonamiento = {Suposicion| Generalizacion| Conclusion| Demostracion}
Suposicion = Suposicion-indiv| Suposicion-colect| Suposicion-exist
Suposicion-indiv = assume Proposicion ";"
Suposicion-colect = assume Condiciones ";"
Suposicion-exist = given Declaracion-ctes ";"
Generalizacion = Generalizacion-tipo| Generalizacion-dato
Generalizacion-tipo = let sort Tipo for Var-tipo
        {"," Tipo for Var-tipo} ";"
Generalizacion-dato = let Declaracion-ctes ";"
Conclusion = thus Demostracion
Declaracion-ctes = Constante {"," Constante} [such that Condiciones]
Condiciones = Proposicion {"," Proposicion}

Justificacion = Referencias| Induccion| Punto-fijo| Lambda-reduccion
Referencias = by Referencia {"," Referencia}
Referencia = Etiqueta| order| bot_order| bool
Induccion = induction Etiqueta "," Etiqueta
Punto-fijo = by fix
Lambda-reduccion = reducing Expresion-funcional by Etiqueta

Facilidad = save Nombre-fichero| include Nombre-fichero

Formula = Formula-atomica| Negacion| Formula-conectiva
        | Cuantificacion| Formula-predefinida| Formula-agrupada
Formula-conectiva = Conjuncion| Disyuncion| Implicacion| Equivalencia
Formula-atomica = Igualdad| Aproximacion| Formula-predicativa
Igualdad = Termino "=" Termino
Aproximacion = Termino "<=" Termino
Formula-predicativa = Identif-predicado [Termino]
Conjuncion = Formula and Formula
Disyuncion = Formula or Formula
Implicacion = Formula implies Formula
Equivalencia = Formula iff Formula
Negacion = not Formula
```

```
Cuantificacion = Formula-universal| Formula-existencial
Formula-universal = for Var-dato-tipada {"," Var-dato-tipada} holds
                                                          Formula
Formula-existencial = ex Var-dato-tipada {"," Var-dato-tipada} st
                                                          Formula
Formula-predefinida = contradiction| thesis
Formula-agrupada = "(" Formula ")"

Termino = Bottom| Variable-dato| Constante| N-tupla| Aplicacion
        | Termino-condicional| Termino-agrupado
Bottom = bot| bot ":" Tipo
Variable-dato = Identif-variable| Var-dato-tipada
Constante = Identif-constante| Simbolo-cte-tipado| true| false
N-tupla = Termino "," Termino {"," Termino}
Aplicacion = Expresion-funcional "(" Termino ")"
Termino-condicional = if Termino then Termino else Termino
        | if Termino then Termino
Termino-agrupado = "(" Termino ")"

Expresion-funcional = Funcion-bottom| Variable-funcion
        | Simbolo-funcion| Lambda-expresion| Fix-expresion
Funcion-bottom = bot| bot ":" Tipo "->" Tipo
Variable-funcion = Identif-variable| Var-funcion-tipada
Simbolo-funcion = Identif-funcion| Simbolo-func-tipado
Lambda-expresion = "(" lambda Var-dato-tipada
        {"," Var-dato-tipada} "." Termino ")"
Fix-expresion = "(" mu Var-funcion-tipada "." Expresion-funcional ")"

Simbolo-cte-tipado = Identif-constante ":" Tipo
Simbolo-func-tipado = Identif-funcion ":" Tipo "->" Tipo
Simbolo-pred-tipado = Identif-predicado ":" Tipo
Var-dato-tipada = Identif-variable ":" Tipo
Var-funcion-tipada = Identif-variable ":" Tipo "->" Tipo

Tipo = Var-tipo| Identif-tipo| Tipo "^" Tipo {"^" Tipo}| bool
        | Identif-tipo "(" Tipo {"," Tipo} ")"
Var-tipo = * {*}
```

Identif-tipo = Identificador
Identif-variable = Identificador
Identif-constante = Identificador
Identif-funcion = Identificador
Identif-predicado = Identificador
Aridad = Dígito
Nombre-fichero = Identificador
Etiqueta = Identificador

Un Identificador es una cadena de caracteres de una longitud arbitraria.

La prioridad de los conectivos es el habitual en la lógica de primer orden, teniendo en cuenta que los cuantificadores afectan a toda la fórmula que aparece a continuación de holds (para las fórmulas universales) y de st (para las fórmulas existenciales) sin necesidad de paréntesis.

2.2 EJEMPLOS DE USO

La capacidad de expresión de PLPR ha sido puesta de manifiesto en los ejemplos presentados a lo largo del trabajo. Así, pueden enunciarse y verificarse propiedades tanto matemáticas como de lenguajes funcionales. El lenguaje de las demostraciones que soporta el sistema MIZ-PR permite mecanizar, de una manera natural, las deducciones realizadas en esta lógica consiguiendo así una herramienta confortable para enunciar y comprobar propiedades propias de estructuras matemáticas o de lenguajes funcionales.

Los siguientes son ejemplos de demostraciones comprobadas o automatizadas por nuestro sistema.

Ejemplo 2.2.1

Este primer ejemplo es una demostración sencilla al estilo Mizar con la salvedad del polimorfismo existente en ella.

El programa se inicializa desde Prolog con el objetivo miz_pr, con lo que se obtiene el prompt :> propio de nuestro sistema. Cada vez que se valida un paso, este símbolo aparece en pantalla para que el

usuario pueda escribir el siguiente paso. El paso es correcto, si no se obtienen mensajes indicando lo contrario.

```
: ?- miz_pr.
:>pred p:*, q:*, r:*;

:>(for x:* holds (p[x] implies q[x]) and (q[x] implies r[x]))
                implies (for x:* holds p[x] implies r[x])
  proof;
:>   let sort ct for *;

:>   assume A# for x:ct holds
              (p[x] implies q[x]) and (q[x] implies r[x]);

:>   let a:ct such that B# p[a];

:>   assume C# not r[a];

:>   thus contradiction by A,B,C;

:>end;

:>exit;
fin de sesion
yes
: ?-
```

Ejemplo 2.2.2

En este ejemplo se muestra, utilizando la demostración del ejemplo anterior, como el sistema devuelve mensajes en caso de error en el razonamiento. Cuando se detecta un error en un paso, se recupera el estado anterior.

```
: ?- miz_pr.
:>pred p:*, q:*, r:*;

:>(for x:* holds (p[x] implies q[x]) and (q[x] implies r[x]))
         implies (for x:* holds p[x] implies r[x])
  proof;

:>let sort ct for *;

:>assume A# for x:ct holds p[x] implies q[x];
la tesis actual no es una implicacion
o no se asume el antecedente

:>assume A# for x:ct holds (p[x] implies q[x]) and (q[x] implies r[x]);
```

```
:>let a:ct such that B# p[a];

:>assume C# not r[a];

:>thus r[a] by A,B,C;
la formula concluida no ha podido ser inferida
a partir de las referencias dadas
esta formula no corresponde con ninguna
componente del objetivo actual

:>thus contradiction by A,B,D;
premisas inexistentes
la formula concluida no ha podido ser inferida
a partir de las referencias dadas

:>end;
la prueba no ha sido finalizada

:>thus contradiction by A,B,C;

:>end;

:>exit;
fin de sesion
yes
: ?-
```

Ejemplo 2.2.3

El siguiente ejemplo muestra, primeramente, la demostración de una propiedad matemática escrita paso a paso por el usuario. Sabemos que cuando aparece el prompt :>, el paso anterior ha tenido éxito.

```
: ?- miz_pr.
:>type int;

:>pred ng:* ^ *;

:>(ex x:int st for y:int holds ng[x,y]) implies
  (for x:int holds ex y:int st ng[y,x])
  proof;

:>   given x:int such that 1# for y:int holds ng[x,y];

:>   let a:int;

:>   2# ng[x,a] by 1;

:>    thus ex y:int st ng[y,a] by 2;

:>end;
```

A continuación se demuestra la misma propiedad automatizando alguno de sus pasos. Puesto que estamos en la misma sesión que en la prueba anterior, no es necesario volver a hacer las correspondientes declaraciones. El sistema escribe un mensaje al comprobar automáticamente que la fórmula correspondiente a la tesis actual es verdadera.

```
:>(ex x:int st for y:int holds ng[x,y]) implies
  (for x:int holds ex y:int st ng[y,x])
  proof;

:>   assume (ex x:int st for y:int holds ng[x,y]);

:>   thus thesis automat;
prueba completada con exito

:>end;
```

Por último se automatiza la prueba de esta propiedad totalmente.

```
:>(ex x:int st for y:int holds ng[x,y]) implies
  (for x:int holds ex y:int st ng[y,x])
  automat;
prueba completada con exito

:>exit;
fin de sesion
yes
: ?-
```

Ejemplo 2.2.4

A continuación se demuestra una propiedad típica de cualquier función estricta, la distributibidad con respecto al término condicional. Para ello se declara una variable de función, f, de tipo arbitrario. Esta propiedad se utiliza en el ejemplo 2.2.6 para el caso de la función *append*.

```
: ?- miz_pr.
:>var f:*->**;

:>for x:bool,y:*,z:* holds f(if x then y else z)
            = if x then f(y) else f(z)
  proof;

:>   let sort t for *;
```

```
:>   let a:bool, b:t, c:t;

:>   A# a=bot or a=true or a=false by bool;

:>   B# a=bot implies f(if a then b else c)
                    = if a then f(b) else f(c)
     proof;

:>      b# f(bot:t) = bot:** by bot_order;

:>      assume 1# a = bot;

:>      B1# if a then b else c = bot by 1,bool;

:>      B2# if a then f(b) else f(c) = bot by 1,bool;

:>      B3# f(if a then b else c) = bot by B1,b;

:>      thus thesis by B2,B3;

:>   end;

:>   T# a=true implies f(if a then b else c)
                      = if a then f(b) else f(c)
     proof;

:>      assume 1# a = true;

:>      T1# if a then b else c = b by 1,bool;

:>      T2# if a then f(b) else f(c) = f(b) by 1,bool;

:>      thus thesis by T1,T2;

:>   end;

:>   F# a=false implies f(if a then b else c)
                       = if a then f(b) else f(c) automat;
prueba completada con exito

:>   thus f(if a then b else c) = if a then f(b) else f(c) by A,B,T,F;

:>end;

:>exit;
fin de sesion

yes
: ?-
```

Ejemplo 2.2.5

Definimos una teoría de listas polimórficas que será almacenada para volver a ser utilizada en el siguiente ejemplo. También se define la función *append* para concatenar listas utilizando una especificación recursiva y se demuestran algunas propiedades de esta función.

```
: ?- miz_pr.
:>type list/1;

:>const []:list(*);

:>func cons:*^list(*)->list(*),hd:list(*)->*,
       tl:list(*)->list(*),null:list(*)->bool;

:>funrec append:list(*)^list(*)->list(*)
     append(x,y) = if null(x) then y else cons(hd(x),append(tl(x),y));

:>L1# null([])=true;

:>L2# for x:*,y:list(*) holds
          (not y=bot and not x=bot) implies null(cons(x,y)) = false;

:>L3# for x:*,y:list(*) holds
          (not y=bot and not x=bot) implies hd(cons(x,y)) = x;

:>L4# for x:*,y:list(*) holds
          (not y=bot and not x=bot) implies tl(cons(x,y)) = y;

:>AP0# for x:list(*),y:list(*),z:list(*) holds
          (x=bot or y=bot or z=bot) implies
           append(cons(x,y),z)=cons(x,append(y,z));

:>AP1# for x:list(*),y:list(*) holds
        append(x,y) = if null(x) then y
                      else cons(hd(x),append(tl(x),y)) by fix;

:>AP2# for x:list(*),y:list(*),z:list(*) holds
       append(cons(hd(x),y),z) = cons(hd(x),append(y,z))
  proof;

:>   let x:list(*),y:list(*),z:list(*);

:>   1# (hd(x)=bot or y=bot or z=bot) implies
        append(cons(hd(x),y),z) = cons(hd(x),append(y,z)) by AP0;

:>   2# (not hd(x)=bot and not y=bot and not z=bot) implies
        append(cons(hd(x),y),z) = cons(hd(x),append(y,z)) proof;
```

```
:>     assume 3# not hd(x)=bot and not y=bot and not z=bot;

:>     4# null(cons(hd(x),y)) = false by 3,L2;

:>     5# hd(cons(hd(x),y)) = hd(x) by 3,L3;

:>     6# tl(cons(hd(x),y)) = y by 3,L4;

:>     7# append(cons(hd(x),y),z) = if null(cons(hd(x),y)) then z
       else cons(hd(cons(hd(x),y)),append(tl(cons(hd(x),y)),z)) by AP1;

:>     8# append(cons(hd(x),y),z) =
       cons(hd(cons(hd(x),y)),append(tl(cons(hd(x),y)),z)) by 4,7,bool;

:>     thus thesis by 5,6,8;

:>  end;

:>  thus append(cons(hd(x),y),z) = cons(hd(x),append(y,z)) by 1,2;

:>end;

:>save listas;

:>exit;
fin de sesion
yes
: ?-
```

Ejemplo 2.2.6

Utilizando la teoría de listas definida y salvada en ejemplo anterior, demostramos que la operación *append* es asociativa. Para ello, se define una variable funcional, X, sobre la que se hace la inducción, y para simplificar la notación del caso básico, se declara la función B como la función *bottom* del mismo tipo que *append*.

```
: ?- miz_pr.
:>include listas;
teoria incorporada al entorno actual

:>func B = bot:list(*) ^ list(*) -> list(*);

:>var X:list(*) ^ list(*) -> list(*);

:>IF# for x:list(*), y:list(*), z:list(*) holds
      append(if null(x) then y else cons(hd(x),X(tl(x),y)),z) =
    if null(x) then append(y,z) else append(cons(hd(x),X(tl(x),y)),z);
```

```
:>ASOC# for x:list(*),y:list(*),z:list(*) holds
        append(append(x,y),z) = append(x,append(y,z))
  proof;
:>    BASE# for x:list(*),y:list(*),z:list(*) holds
           append(B(x,y),z) = B(x,append(y,z)) by bot_order;
:>    IND#  (for x:list(*),y:list(*),z:list(*) holds
           append(X(x,y),z) = X(x,append(y,z))) implies
           (for x:list(*),y:list(*),z:list(*) holds
      append(if null(x) then y else cons(hd(x),X(tl(x),y)),z) =
   if null(x) then append(y,z) else cons(hd(x),X(tl(x),append(y,z))))
      proof;
:>        let sort ct for *;
:>        assume HI# for x:list(ct),y:list(ct),z:list(ct) holds
                    append(X(x,y),z) = X(x,append(y,z)));
:>        let a:list(ct),b:list(ct),c:list(ct);
:>        1# append(cons(hd(a),X(tl(a),b)),c)
                 = cons(hd(a),append(X(tl(a),b),c)) by AP2;
:>        2# append(if null(a) then b else cons(hd(a),X(tl(a),b)),c)
                 = if null(a) then append(b,c) else
                      append(cons(hd(a),X(tl(a),b)),c) by IF;
:>        3# append(if null(a) then b else cons(hd(a),X(tl(a),b)),c)
                 = if null(a) then append(b,c) else
                      cons(hd(a),append(X(tl(a),b),c)) by 1,2;
:>        thus thesis by HI,3;
:>     end;
:>     thus for x:list(*),y:list(*),z:list(*) holds
   append(append(x,y),z) = append(x,append(y,z)) induction BASE,IND;
:>end;
:>exit;
fin de sesion
yes
: ?-
```

3. IMPLEMENTACION

En la actualidad existe un prototipo del sistema MIZ-PR programado en Quintus Prolog que funciona dentro de un entorno EMACS en una estación SUN con la versión 3 de Unix.

En la elaboración de este prototipo, se han ponderado los aspectos de claridad por encima de los de eficiencia para conseguir así un programa, que a la vez de ser ejecutable, sea capaz de especificar por si mismo la semántica del sistema. Esto ha sido posible gracias a las particularidades de la programación lógica. Comentaremos en esta sección algunos de los predicados del programa Prolog destinados al control de las demostraciones.

3.1 GENERALIDADES DE LOS PROGRAMAS PROLOG

El sistema consta de dos partes claramente diferenciadas. Una parte realiza el análisis sintáctico y traduce a términos Prolog cada uno de los pasos de una demostración. Estos términos servirán de entrada a la parte del sistema encargada del control de las demostraciones. Nos ocuparemos aquí de estudiar la implementación de esta segunda parte.

3.1.1 Representación de las fórmulas

La semántica de las fórmulas permite simplificar los conectivos, de manera que a nivel interno las fórmulas son conjunciones (posiblemente con una sola componente). Cada fórmula, F, viene representada por una lista cuyos elementos son las componentes de dicha fórmula. Si P es un símbolo de predicado y T, T1, T2 representan términos, cada componente de F es una lista con alguna de las siguientes formas:

```
[P,T]          % fórmula predicativa
[=,T1,T2]      % igualdad de términos
[<,T1,T2]      % aproximación
[n,F]          % negación
[pt,F]         % cuantificación universal
```

La aparición de una fórmula en las dos últimas listas supone la

forma recursiva de la representación de las fórmulas.

Los términos y expresiones funcionales del lenguaje también se representan mediante términos Prolog que llevan incorporada información de su tipo. Se ha utilizado una representación de functores (var, const, func, bot, lambda, etc.) y operadores (prod) para las distintas construcciones de términos y expresiones funcionales.

3.1.2 Representación de los entornos

Una serie de operadores predefinidos facilitan la representación de las estructura de entornos o contextos que caracterizan a cada demostración. Cada contexto se identifica con un término Prolog formado por dos subtérminos uno es una lista de asignaciones, AL, y el otro es la representación de la fórmula que constituye la tesis actual, Th; ambos términos se separan por el operador "#". La lista de asignaciones es una lista de fórmulas etiquetadas, la etiqueta es un átomo que se une a la fórmula mediante el operador infijo ":".

Una estructura de entornos E tiene la siguiente forma recursiva:

AL # Th o bien E :: E

Esta representación permite modificar con facilidad los entornos a la vista de la técnica de demostración elegida en cada paso, según se explicó en 1.2.1.

3.1.3 Representación de los items de la demostración

Para representar los pasos de las demostraciones, se han declarado una serie de operadores Prolog, de manera que cada uno de los items, correspondientes a las distintas técnicas de demostración, se convierte en un término Prolog con una estructura propia (como muestra la figura 7), lo que permite su fácil reconocimiento por ajuste de patrones.

La variable Justific que aparece en la figura 7 debe encajar con alguno de los patrones mostrados en la figura 8, que sirven para representar las distintas técnicas de demostración hacia delante.

Un paso en el que se solicita la automatización de la demostración de una fórmula, se traduce por el término:

autom Etiqueta : Formula

```
assume Etiqueta : Formula            % Suposición de hipótesis
let(const(Tipo,Nombre))              % Generaliz. de var. de dato
let(Tipo,Vartipo),NE)                % Generaliz. de var. de tipo
thus proof(Teorema)                  % Descomposición del objetivo
thus Etiqueta:Formula Justific       % Descomposición del objetivo
Etiqueta : Formula Justificacion     % Demostración hacia adelante
Etiqueta : Formula                   % Axioma
proof(Teorema)                       % Comienzo de prueba por pasos
end(Etiqueta : Teorema)              % Fin de prueba de un teorema
```

Figura 7

```
by ListaEtiquetas                    % Referen. a axiomas o teoremas
by fix                               % Axioma de punto fijo
induc [Etiqueta1,Etiqueta2]          % Inducción
reduc [ExpFuncional,Etiqueta]        % β-Reducción
```

Figura 8

3.2 ALGUNOS DETALLES DE LOS PREDICADOS PROLOG

Los módulos Prolog que soportan el peso principal del control y automatización de la demostración tienen los siguientes nombres: *control*, *reasoner*, *prover*, *checker*, *equal*, *induction*, *lambda*.

Existen otros como *substitution* o *formulas* destinados a tareas complementarias.

A continuación analizaremos brevemente cada uno de los siete módulos principales arriba mencionados.

3.2.1 Módulo control

Este módulo está destinado a realizar el interface entre los programas de lectura y escritura y los programas de comprobación y automatización de la demostración.

Se inicializa el proceso con un entorno inicial vacío. Por cada ítem, analizado y convertido en término Prolog por el analizador sintáctico, se hace una llamada al *reasoner*, quien se encargará de comprobar la corrección lógica, o al módulo que construye la demostración (*prover*) si se trata de la automatización de un paso.

El procedimiento principal de este módulo lo constituye el predicado recursivo *control_proof* cuyo parámetro es la estructura de entornos de la demostración.

```
control_proof([]) :- !.
control_proof(Env) :-
                read_item(_,Item),
                verify_item(Env,Item,NewEnv),
                control_proof(NewEnv).
```

El predicado *verify_item* es el encargado de distinguir si una vez escrito un paso de la demostración, se trata de final de sesión, de una prueba automática, de una facilidad, de una lista de items a analizar por el *reasoner*, o si no hay nada que demostrar por tratarse por ejemplo de una declaración de datos o tipos.

```
verify_item(_,[exit],[]) :- !,
                write('fin de sesion').
verify_item(Env,[],Env) :-!.
verify_item(AL#[],[save(T)],[]) :- !,
                recordz(asig,AL,_),
                save(T).
verify_item(AL1#[],[include(T)],AL#[]) :- !,
                restore(T),
                recorded(asig,AL2,Ref),
                erase(Ref),
                append(AL1,AL2,AL).
verify_item(Env,[autom L : F],NewEnv) :- !,
                prove(Env,L:F,NE),
                is_ok->NewEnv = NE;NewEnv = Env.
verify_item(Env,Item,NewEnv) :-
                reasoner(Env,Item,NE),
                is_ok->NewEnv = NE;NewEnv = Env.
```

Después del control de cada paso, si el análisis finaliza con éxito, el entorno inicial será transformado sirviendo de entrada al procedimiento *control_proof* quien se encarga de nuevo de hacer la

llamada a los procedimientos de lectura. En caso contrario, el entorno no varía y se escriben los mensajes de error antes de leer el siguiente item.

El predicado *is_ok* decide si un proceso ha tenido éxito a la vista de los códigos de error que ha desencadenado, y escribe los mensajes correspondientes.

```
is_ok :-        all_errors(Err),
                error_text(Err),
                Err = [0];Err =[].
```

3.2.2 Módulo reasoner

Una explicación detallada de los predicados de este módulo para el sistema MIZAR/LOG aparece en [Nie-89]. El reasoner de MIZ-PR, al igual que el del sistema inicial, analiza cada item de la demostración que ha sido convertido en un término Prolog. Las principales diferencias entre uno y otro sistemas son, por un lado, el carácter más interactivo del módulo actual que permite una respuesta inmediata después del análisis de cada item sin esperar al final de la demostración.

Por otro lado, las técnicas de demostración propias del nuevo sistema (generalización de tipos, λ-cálculo, inducción y punto fijo) obligan a incorporar nuevas cláusulas para definir los predicados *reduces* y *justification* que aparecen dentro de el módulo *reasoner*. El primero se ocupa de analizar los items modificando de forma adecuada el contexto. El segundo tiene lugar cuando se utiliza una técnica de demostración hacia adelante y se encarga de la ejecución de los procedimientos correspondientes.

Los predicados que constituyen el módulo *reasoner* llevan a cabo las acciones necesarias para analizar la corrección de un item que fueron explicadas en 1.2.1; siendo ésta la razón de que no insistamos ahora en detallar cada uno de estos predicados. Unicamente y a modo de ejemplo, señalamos la cláusula del predicado *reduces* correspondiente al análisis de la técnica de generalización de tipos y las cláusulas del predicado *justification* para las técnicas de referencia al axioma de punto fijo, de reducción de una lambda expresión y de inducción. El resto del programa puede verse en el anexo.

```
reduces(E, let(Ty, v(TyV)), NE) :- !,
                thesis_of(E, Th),
                subs_type(Ty, TyV, Th, NTh),
                constructor_of_thesis(E, NTh, NE).
reduces(E, J, NE) :- !,
                justification(E, J, NE, _).
```

Cuando se justifica una sentencia por referencia al axioma de punto fijo, es el predicado *fixed*, que forma parte del módulo *induction* el que se encarga de validarla.

```
justification(E, L : S by fix, NE, S1) :- !,
                thesis_of(E, Th),
                replace_thesis(S, Th, S1),
                fixed(S1),
                add_binding(L : S1, E, NE).
```

Si se indica que se está aplicando una inducción se hace una llamada al procedimiento *induction* en el módulo del mismo nombre.

```
justification(E, L : S induc LL, NE, S1) :- !,
                thesis_of(E, Th),
                replace_thesis(S, Th, S1),
                look_up(LL, E, SL),
                induction(S1, SL),
                add_binding(L : S1, E, NE).
```

Si en el razonamiento se produce de manera explícta una β-reducción de una λ-expresión, se comprueba la corrección mediante el procedimiento *check_reducing* que forma parte del módulo *lambda*.

```
justification(E, L : S reduc [FE, L1], NE, S2) :- !,
                thesis_of(E, Th),
                replace_thesis(S, Th, S2),
                find(L1 : S1, E),
                check_reducing(S1, S2, FE),
                add_binding(L : S2, E, NE).
```

3.2.3 Módulo prover

Para construir una demostración de una fórmula de forma automática, se utiliza el procedimiento *prover*, incluido en el módulo de igual nombre.

```
prover(E,L:F,NE) :-
                thesis_of(E,Th),
                replace_thesis(F,Th,F1),
                construct_proof(E,F1,1),
                add_binding(L:F1,E,NE).
```

Una vez que en la fórmula que queremos demostrar, se sustituye, si aparece, la palabra thesis por su valor actual, el predicado *construct_proof* llama a las distintas técnicas de demostración para construir la prueba.

```
construct_proof(_,[],_) :- !.
construct_proof(E,[F:RF],N) :-
                do_new_monomorphic([F:RF]),
                select_tech(E,F,NF,NE,N),
                N1 is N + 1,
                construct_proof(NE,NF,N1),
                N2 is N1 + 1,
                construct_proof(E,RF,N2).
construct_proof(E,F,_) :-
                insert_bool(E,AL),
                check_consequence(AL,F).
construct_proof(E,F,_) :-
                insert_bot(E,AL),
                check_consequence(AL,F).
construct_proof(E,F,_) :-
                insert_order(E,AL),
                check_consequence(AL,F).
```

El predicado *do_new_monomorphic* produce la instanciación de una fórmula con nuevas constantes de tipo, y ejecuta por tanto la primera de las técnicas de demostración posibles. A partir de ese momento,

select_tech ensaya el resto de las técnicas para cada una de las subfórmulas o componentes que constituyen el objetivo inicial.

```
select_tech(E,[pt,Ty,F],NF,E,N) :-
                subs(const(Ty,i(N)),F,NF).
select_tech(E,[n,[F1:F2]],NF2,NE,_) :-
                add_binding(**n:[F1],E,NE)
                negation(F2,NF2).
select_tech(E,F,NF,E,_) :-
                reducing([F],NF).
select_tech(E,F,[],E,_) :-
                fixed([F]).
select_tech(E,F,[],E,_) :-
                elim_label_env(E,AL),
                check_consequence(AL,[F]).
```

El orden en el que aparecen las clausulas de *select_tech* sigue los criterios establecidos en 1.2.4 de aplicación de las técnicas de demostración.

3.2.4 Módulo checker

Corresponde a este módulo la tarea de comprobar si es posible encontrar una derivación de una fórmula a partir de unas premisas. La mayor parte de las inferencias se comprueban utilizando el método de los tableaux, sin embargo, aquellas que se fundamentan en las propiedades de la igualdad, se comprueban por un algoritmo particular implementado en el módulo *equal* que se explica en 3.2.5.

El predicado *check_consequence* ensaya los dos métodos anteriores, *check_equality* supone la llamada al módulo *equal*, mientras que *refutation* prepara las fórmulas para construir un tableau y desencadena dicho algoritmo.

```
check_consequence(SL,S) :-
                check_equality(SL,S)
               ;
                refutation(SL,S).
```

```
refutation(SL,S) :-
                negation(S,NS),
                do_new_monomorphic(NS),!,
                closed_tableau([NS:SL]).
```

El predicado *closed_tableau* investiga primero si el conjunto de fórmulas inicial (FS) da lugar a una rama cerrada, en caso negativo, se hacen monomórficas las fórmulas de este conjunto y, para conseguir cierta eficiencia en la consecución de un tableau, se clasifican las fórmulas en las correspondientes categorías - alfa (AL), beta (BL), delta (DL), gamma (GL) y básica o lista de literales (LL) -, esta clasificación, conseguida mediante el predicado *classify*, se mantiene y actualiza para cada rama conforme se generan nuevas subfórmulas, en estos casos se utiliza el predicado *place*.

```
closed_tableau(FS) :-
                closed_branch(FS),!
               ;
                instance_set(FS,MFS),
                classify(MFS,AL,BL,DL,GL,LL,TL),
                close(AL,BL,DL,GL,LL,TL).
```

La búsqueda de un tableau cerrado comienza con la construcción de un tableau proposicional, el predicado definido con tal objetivo es *close*.

```
close([],[],DL,GL,LL,TL) :- !,
                depth(DL,GL,TL,DP),
                close_m([],[],DL,GL,LL,TL,0,DP).
close([],[B:BL],DL,GL,LL,TL) :- !,
                beta_formula(B,B1,B2),
                confront_beta(B1,B2,BL,DL,GL,LL,TL).
close([A:AL],BL,DL,GL,LL,TL) :-
                alfa_formula(A,A1,A2),
                confront_alfa(A1,A2,AL,BL,DL,GL,LL,TL).
```

Si con las fórmulas sin cuantificar no se logra cerrar el

tableau, las fórmulas delta y gamma entran en juego, la la búsqueda de un tableau de predicados cerrado la realiza *close_m*.

Los predicados *confront_alfa* y *confront_beta* producen la α-extensión y la β-bifurcación, respectivamente, en el tableau proposicional que se está construyendo y desencadenan la llamada recursiva al predicado *close* de búsqueda del tableau cerrado. Los análogos de estos predicados, para el caso de la construcción de un tableau de predicados, son *confront_alfa_m* y *confront_beta_m*, respectivamente que pueden verse desarrollados en el anexo.

```
confront_alfa(A,A,AL,BL,DL,GL,LL,_) :-
                opposite_t(A,AL,BL,DL,GL,LL),!.
confront_alfa(A,A,AL,BL,DL,GL,LL,TL) :- !,
                place(A,AL,BL,DL,GL,LL,NAL,NBL,NDL,NGL,NLL),
                close(NAL,NBL,NDL,NGL,NLL,TL).
confront_alfa(A1,A2,AL,BL,DL,GL,LL,TL) :-
                opposite_t(A1,AL,BL,DL,GL,LL),!
               ;
                opposite_t(A2,AL,BL,DL,GL,LL),!
               ;
                negation(A1,A2),!
               ;
                place(A1,AL,BL,DL,GL,LL,AL1,BL1,DL1,GL1,LL1),
                place(A2,AL1,BL1,DL1,GL1,LL1,NAL,NBL,NDL,NGL,NLL),
                close(NAL,NBL,NDL,NGL,NLL,TL).

confront_beta(B,B,BL,DL,GL,LL,TL) :- !,
                close_beta(B,BL,DL,GL,LL,TL).
confront_beta(B1,B2,BL,DL,GL,LL,TL) :-
                close_beta(B1,BL,DL,GL,LL,TL),
                close_beta(B2,BL,DL,GL,LL,TL).

close_beta(B,BL,DL,GL,LL,TL) :-
                opposite_t(B,[],BL,DL,GL,LL),!
               ;
                place(B,[],BL,DL,GL,LL,NAL,NBL,NDL,NGL,NLL),
                close(NAL,NBL,NDL,NGL,NLL,TL).
```

Como ya se ha señaldo, de no tener éxito en la búsqueda del tableau proposicional se continuará la búsqueda con *close_m* que se encarga de manejar también fórmulas delta y gamma.

```
close_m([],[],[],[],_,_,_,_) :- !,
                fail.
close_m(_,_,_,_,_,_,_,0)  :- !,
                fail.
close_m([],[],[],GL,LL,TL,I,DP) :- !,
                confront_gamma(GL,LL,TL,I,DP).
close_m([],[],[D:DL],GL,LL,TL,I,DP) :- !,
                delta_formula(D,Ty,F),
                confront_delta(Ty,F,DL,GL,LL,TL,I,DP).
close_m([],[B:BL],DL,GL,LL,TL,I,DP) :- !,
                beta_formula(B,B1,B2),
                confront_beta_m(B1,B2,BL,DL,GL,LL,TL,I,DP).
close_m([A:AL],BL,DL,GL,LL,TL,I,DP) :-
                alfa_formula(A,A1,A2),
                confront_alfa_m(A1,A2,AL,BL,DL,GL,LL,TL,I,DP).
```

La instanciación de las fórmulas delta con nuevas constantes se realiza por medio de *confront_delta*. Para conseguir nuevas constantes, se mantiene un índice I que aumenta por cada instanciación de una fórmula delta. El proceso de construcción del tableau continua recursivmente, la llamada recursiva a *close_m* tiene lugar por medio del predicado *process_gamma* según veremos más tarde.

```
confront_delta(Ty,F,DL,GL,LL,TL,I,DP) :-
                subs(const(Ty,aux(I)),F,SF),
                close_delta(SF,DL,GL,LL,[const(Ty,aux(I))|TL],I,DP).

close_delta(F,DL,GL,LL,TL,I,DP) :-
                opposite_t(F,[],[],DL,GL,LL),!
               ;
                I1 is I + 1, DP1 is DP - 1,
                place(F,[],[],DL,GL,LL,NAL,NBL,NDL,NGL,NLL),
                process_gamma(NAL,NBL,NDL,NGL,NLL,TL,I1,DP1).
```

A cada fórmula gamma se le asocia una marca que consiste en la lista de términos con los que ya ha sido instanciada. Por otra parte, el número de instancias de fórmulas gamma y delta, es decir, el número de γ y δ extensiones no puede superar un cierto valor, DP, calculado mediante el predicado *depth* utilizando el número de términos y cuantificaciones en el conjunto de fórmulas inicial.

Para una fórmula tipo γ marcada, el predicado *process_gamma* elige, siempre que sea posible (predicado *look_term*), un término que no aparece en su marca, instancia esta fórmula universal con dicho término y actualiza su marca. . Por otro lado, produce la llamada recursiva al procedimiento *close_m*, decreciendo, previamente, la profundidad, DP, en una unidad.

```
confront_gamma(GL,LL,TL,I,DP) :-
                process_gamma([],[],[],GL,LL,TL,I,DP).

process_gamma(AL,BL,DL,GL,LL,TL,I,DP) :-
                look_term(GL,TL,GL1,F),!,
                close_gamma(F,AL,BL,DL,GL1,LL,TL,I,DP).
process_gamma(AL,BL,DL,[G:GL],LL,TL,I,DP) :-
                append(GL,[G],NGL),
                DP1 is DP - 1,
                close_m(AL,BL,DL,NGL,LL,TL,I,DP1).

close_gamma(F,AL,BL,DL,GL,LL,TL,I,DP) :-
                opposite_t(F,[],[],[],GL,LL),!
               ;
                DP1 is DP - 1,
                place(F,AL,BL,DL,GL,LL,NAL,NBL,NDL,NGL,NLL),
                close_m(NAL,NBL,NDL,NGL,NLL,TL,I,DP1).
```

3.2.5 Módulo equal

Este módulo está destinado a comprobar las inferencias que se fundamentan en las propiedades de la igualdad. Esto hace ganar eficiencia al *checker*, ya que se evita la construcción de un tableau cuando la sentencia que se quiere probar es consecuencia de las propiedades de la igualdad.

El algoritmo simulado en el módulo *equal*, para deducir la igualdad de dos términos a partir de un conjunto de ecuaciones, es el definido en [Zuk-87], que ha sido también utilizado en otros sistemas de la familia Mizar.

El predicado principal de este módulo es *check_equal*, cuyos tres argumentos corresponden a los dos términos cuya igualdad quiere comprobarse y al conjunto de igualdades que forman las hipótesis. Con todos los términos y subtérminos existentes, entre premisas y conclusión, se construye una partición inicial que tiene tantas clases como términos. Después de ejecutar el algoritmo que devuelve una partición formada por clases de equivalencia, los términos cuya igualdad quiere demostrarse deben pertenecer a la misma clase.

```
check_equal(T,T,_) :- !.
check_equal(prod[T1|TL1],prod[T2|TL2],EL) :- !,
                check_equal(T1,T2,EL),
                check_equal(prod TL1,prod TL2,EL).
check_equal(T1,T2,EL) :-
                terms_in([[['=',T1,T2]]|EL],TL),
                cond_to_func(TL,TL1),
                partition(TL1,P),
                trans_cond(EL,NEL),
                algorithm(P,FP,NEL),
                class(T1,FP,C),
                class(T2,FP,C).
```

El predicado *algorithm* realiza la tarea más importante del método presentado en [Zuk-87] para conseguir clases de equivalencia, creando nuevas particiones al eliminar colisiones entre clases.

```
algorithm(P,FP,EL) :-
                new_partition(P,EL,NP),
                elim_clashes(NP,FP).
```

El predicado *new_partition* produce una transformación de la partición P consistente en solapar todos los pares de clases en P correspondientes a dos términos, cuya igualdad aparece en el conjunto

de ecuaciones de partida EL.

```
new_partition(P,[],P) :- !.
new_partition(P,[[['=',T1,T2]]:EL],NP) :-
                class(T1,P,C1),
                class(T2,P,C2),
                new_partition1(C1,C2,P,EL,NP).
new_partition1(C,C,P,EL,NP) :- !,
                new_partition(P,EL,NP).
new_partition1(C1,C2,P,EL,NP) :-
                join(C1,C2,P,P1),
                new_partition(P1,EL,NP).
```

Dos clases C1 y C2 colisionan cuando dos expresiones funcionales correspondientes a la misma función están en distintas clases mientras que los términos a los que se aplican están en la misma clase. El procedimiento *elim_clashes* busca y solapa todas las colisiones de una partición para devolver una partición sin colisiones. Para detectar las colisiones se utiliza *clashes*. El predicado *join* solapa las clases C1 y C2, eliminándolas de la partición y añadiendo una clase formada por la unión de las dos anteriores.

```
elim_clashes(P,FP) :-
                clashes(P,P,C1,C2),
                join(C1,C2,P,NP),!,
                elim_clashes(NP,FP)
               ;
                FP=P.

clashes(P1,[C:P],C,NC) :-
                member(func(Ty,FE,T1),C),
                class(func(Ty,FE,T2),P,NC),
                class(T1,P1,CL),
                class(T2,P1,CL).
clashes(P1,[_:P],C1,C2) :-
                clashes(P1,P,C1,C2).
```

```
join(C1,C2,P,[C:NP]) :-
                subtract(P,[C1,C2],NP),
                append(C1,C2,C).
```

class(T,P,C) tiene éxito, cuando la clase del término T en la partición P es precisamente C, este último argumento pude estar instanciado al ser llamado el predicado o puede ser variable.

3.2.6 Módulo induction

El predicado principal de este módulo, también llamado *induction*, controla si una demostración por inducción está bien realizada. Mediante el predicado *induc_pattern*, se trata de encontrar el patrón de la fórmula sobre la que se realiza la inducción. Por otro lado (predicado *continuous*) se demuestra que dicha fórmula es continua.

```
induction(F,Pr) :-
                induc_pattern(F,Pr,FX,X),
                continuous(FX,X).
```

Para encontrar el patrón de la fórmula y la variable sobre la que se realiza la inducción, se tienen como datos la fórmula resultante, en cuyo interior hay una definición recursiva, y las dos premisas que deben corresponder al caso *bottom* (FB) y al paso de inducción (la implicación FX→FM), respectivamente.

```
induc_pattern(F,Pr,FX,X) :-
                induc_premis(Pr,FB,FX,FM),
                differs(F,FB,fix(X,M),bot),
                differs(F,FX,fix(X,M),X),atom(X)
                subs_func(X,M,FX,FM).
```

El último predicado (*subs_func*) produce la sustitución de la variable funcional X por la expresión M en la fórmula FX obteniéndose la nueva fórmula FM que debe coincidir con el consecuente del paso de inducción.

La especificación de una función recursiva se transforma en una representación interna que identifica el símbolo empleado para la

función recursiva con un término Prolog que comienza con el functor *fix*. De esta representación se sirve el predicado *differs* para demostrar que las diferencias sintácticas entre la fórmula concluida, F, y el caso básico, FB, son las adecuadas; análogamente comprueba la diferencia entre F y el antecedente del paso de inducción FX.

La representación de las fórmulas en forma de listas permite al demostrador reconcer como fórmulas continuas un gran número de expresiones sintácticas de esto se ocupa el predicado *continuous*, que puede verse en el anexo.

En este módulo también aparece el predicado *fixed* destinado a comprobar o automatizar los pasos de la demostración que se apoyan en el axioma de punto fijo. Este predicado se sirve de la representación de las μ-expresiones para poder aplicar dicho axioma.

El término Prolog *fix(X,M)* representa una función declarada recursiva que como tal es equivalente a una μ-expresión (μX.M). *X* es la representación en forma de término Prolog de X, mientras que *M* lo es de M, y en su interior aparecerá *X*.

Se ha programado un algoritmo que permite derivar la igualdad de dos términos tales que uno de ellos es una función recursiva, fix(X,M), aplicada a un término T, mientras que el otro consiste en aplicar una función, que es el resultado de sustituir en M la variable X por fix(X,M), al mismo término T. Si la expresión M es una λ-expresión, puede haberse producido la β-reducción correspondiente.

```
fixed([[pt,_,F]]) :- !,fixed(F).
fixed([[=,func(Ty,fix(X,M),T),func(Ty,MM,T)]]) :-
                copy_term(fix(X,M),fix(X1,M1)),
                copy_term(fix(X1,M1),fix(X2,M2),
                X1 = fix(X2,M2),MN = M1.
fixed([[=,func(Ty,fix(X,M),T),RT]]) :-
                copy_term(fix(X,M),fix(X1,M1)),
                copy_term(fix(X1,M1),fix(X2,M2)),
                X1 = fix(X2,M2),
                apply(N1,T,RT),
                type_of(RT,Ty).
```

3.2.7 Módulo lambda

Este módulo se encarga tanto de la automatización de una β-reducción como de la comprobación de su corrección cuando ésta sirve de justificación en un paso dado por el usuario.

Una vez localizada la λ-expresión a reducir, se trata de sustituir en el cuerpo de esta expresión, las variables ligadas por el término al cual se aplica, esto se lleva a cabo mediante *apply*.

```
apply(lambda(VL,T),prod TL,RT) :- !,
                same_length(VL,TL),
                copy_term(lambda(VL,T),lambda(VL1,T1)),
                replace_vars(VL1,TL,T1,RT).
apply(lambda([var(Ty,X)],T),T2,RT) :-
      copy_term(lambda([var(Ty,X)],T),lambda([var(Ty1,X1)],T1)),
                type_of(T1,Ty1),
                replace_var(X1,T2,T1,RT).
```

Genéricamente, la representación de una λ-expresión en forma de término Prolog tiene la forma lambda(ListVar,Term), donde ListVar es la lista de las variables ligadas y Term es la representación del cuerpo de la λ-abstracción. Cada una de las variables de ListVar aparece escrita de la forma var(Tipo,X), siendo X una variable Prolog, que sirve para representar cada una de las ocurrencias de la variable ligada corrrespondiente, dentro de Term.

El predicado *replace_vars*, se ha programado, utilizando la representación anterior, de manera que produce la sustitución en Term de las variables ligadas por los términos a los que se aplica la λ-expresión.

```
replace_vars([],[],T,T) :- !.
replace_vars([var(Ty,X)|VL],[T1|TL],T,RT) :-
                type_of(T1,Ty),
                replace_var(X,T1,T,NT),
                replace_vars(VL,TL,NT,RT).

replace_var(X,T1,T,RT) :-
                X = T1,
                RT = T.
```

CONCLUSIONES

El objetivo primordial de este trabajo ha sido diseñar un sistema de deducción automática capaz tanto de demostrar teoremas matemáticos como de especificar y comprobar propiedades de programas funcionales.

Para cumplir nuestro objetivo, hemos definido la lógica computacional PLPR, para ella se han encontrado unos mecanismos generales de deducción, y por último se ha implementado un sistema que soporta a esta lógica como lenguaje objeto y es capaz de simular ciertos razonamientos humanos dentro del ámbito en el que nos movemos.

PLPR es una extensión de la lógica de predicados; los operadores λ y μ, junto con el polimorfismo, son las principales herramientas con las que cuenta el lenguaje para conseguir la expresividad esperada. Además, la definición de interpretación de los diferentes objetos nos lleva a una semántica denotacional muy práctica.

Se han hecho estudios de la complejidad del problema de validez de las fórmulas, pudiendo asegurar la no existencia de cálculos finitarios para PLPR. No obstante, se ha introducido una extensión del método de los tableaux de Smullyan, algoritmo que permite semi-decidir la insatisfactibilidad de un conjunto de fórmulas.

También se han presentado otros cálculos de deducción natural para nuestra lógica y se han demostrado teoremas de completitud. Con el fin de tratar cómodamente la recursión, se ha incorporado una regla de inducción de punto fijo para ciertas fórmulas llamadas continuas.

Por último, para conseguir que la automatización de la lógica anterior, se asemeje, en algunos aspectos, a la forma de razonar humana, se ha diseñado un sistema, MIZ-PR, dotado de un lenguaje cercano al lenguaje natural, que permite al usuario hacer especificaciones, enunciar teoremas y guiar las demostraciones.

El sistema semi-automático obtenido presenta mejoras notables con respecto a la familia de sistemas Mizar en los que está basado; estas mejoras cubren los aspectos lógico, humano y técnico.

Desde el punto de vista de la fundamentación, la lógica de primer orden, que sirve de lenguaje objeto de los sistemas Mizar, ha sido extendida con polimorfismo y recursión dando una mayor capacidad de expresión al sistema.

Desde el punto de vista de la proximidad del sistema al usuario, el lenguaje utilizado en los sistemas Mizar para escribir demostraciones se ha extendido posibilitando especificar tipos (incluso polimórficos y construidos) y definir funciones recursivas de una manera similar a la manera habitual en que se especifican los programas recursivos. Además se han incorporado nuevas técnicas de demostración entre las que destacamos el uso generalizado de la regla de inducción.

Desde el punto de vista técnico, se ha conseguido una interacción hombre máquina no existente en los sistemas iniciales y una cierta automatización no contemplada en Mizar. En cuanto a la implementación, podemos señalar que mientras los programas que soportan a la mayoría de los sistemas Mizar están escritos en PASCAL, los programas de MIZ-PR están escritos en PROLOG, con lo que constituyen una especificación semántica del sistema.

Las vías de continuación que este trabajo ofrece son numerosas. Extender PLPR incorporando orden superior constituye una línea de investigación abierta, que podría afrontarse considerando translaciones sintácticas que permitan afrontar la semántica del nuevo lenguaje por referencia a la semántica de primer orden. Por otro lado, MIZ-PR es un prototipo que como tal puede ser mejorado y desarrollado. Entre los objetivos actuales están la inferencia automática de tipos, la mejora del tratamiento de los errores y el estudio e implementación de *checkers* más potentes que permitan automatizar mayor número de pasos.

REFERENCIAS

[Bak-80] Bakker, J.
Mathematical Theory of Program Correctness. Prentice Hall, 1980.

[Ber-85] Bermúdez, J.
Una Esposición de los Fundamentos Matemáticos de la Semántica Denotacional. Tesina de Licenciatura. Fac. de Matemáticas UCM, 1985.

[Bet-64] Beth, E.W.
The Foundations of Mathematics. North-Holland Amsterdam, 1959. Revised edition in 1964.

[Bib-90] Bibel, W.
Perspectives of Automated Deduction. Invited talk at the 10th Conference on Automated Deduction, 1990.

[BM--79] Boyer, R.S. and Moore, J.S.
A Computational Logic. Academic Press, 1979.

[BM--88] Boyer, R.S. and Moore, J.S.
A Computational Logic Handbook. Academic Press, 1988.

[Bru-80] Bruijn N.G. de
A Survey of the project AUTHOMAT. To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism, Academic-Press, 1980.

[Con-86] Constable, R.L. et al.
Implementing Mathematics with the Nuprl Proof Development System. Computer Science Dept. Cornell University, Ithaca, N.Y., 1986.

[Dal-80] van Dalen, D.
Logic and Structure. Springer-Verlag, 1980.

[EFT-84] Ebbinghaus, H.-D., Flum, J. and Thomas, W.
Mathematical Logic. Springer-Verlag, 1984.

[Emd-83] van Emde Boas, P.
Dominoes are Forever. 1st GTI Workshop, Paderborn, 75-95, 1983.

[Fit-88] Fitting, M.
First-Order Modal Tableaux. Jour. of Automated Reasoning 4, 191-213, 1988.

[Fit-90] Fitting, M.
First-Order logic and Authomated Theorem Proving.

Springer-Verlag, 1990.

[Gav-89] Gavilanes, A.
Una Lógica Trivaluada para Funciones Recursivas Parciales Tesis Doctoral, Fac. Matemáticas UCM, 1989.

[Ger-80] Gerhart, S.L. et al.
An overview of Affirm: a specification and verification system. Information Processing 80 (ed. Lavington), North Holland, 1980.

[Gil-89] Gil, A.
Una Lógica no Estandar Admisible para Programas Funcionales. Tesis Doctoral, Fac. Matemáticas UCM, 1989.

[Gil-60] Gilmore, P.C.
A Proof Method for Quantification Theory: Its Justification and Realization. IBM Jour. Research and Devlop., 28-35, 1960.

[Gor-89] Gordon, M.J.C.
Mechanizing Programming Logics in Higher Order Logic. Current Trends in Hardware Verification and Automated Theorem Proving. (Birtwistle, G. and Subrahmanyam, P.A., ed), 387-439. Springer Verlag, 1989.

[Har-83] Harel, D.
Recurring Dominoes: Making the highly undecidable highly understandable. LNCS 158. (FCT'83, Karpinski ed.), 1983.

[Har-84] Harel, D.
A Simple Highly Undecidable Domino Problem. Proc. Conf. Logic & Computation, Clayton Victoria, Australia, 1984.

[HD--86] Hanna, F.K. and Daeche, N.
Purely functional implementation of a logic. LNCS 230, 598-607, 1986.

[Lew-78] Lewis, H.R.
Complexity of Solvable Classes of the Decision Problem for the Predicate Calculus. 19th FOCS, 35-47, 1978.

[LMR-87] Lindsay, P.A., Moore, R.C. and Ritchie, B.
Review of Existing Theorem Provers. Thechnical Report Series UMCS, 1987.

[LN--90] Leach, J. and Nieva, S.
A Predicate Logic using Polymorphism. Thechnical Report,

Dpto. Informática y Automática UCM. (DIA/90/30), 1990.

[Lov-83] Loveland D.W.
Automated Theorem Proving: After 25 Years. Contemporary Mathematics, 29. American Mathematical Society, 1983.

[LP--81] Lewis, H.R. and Papadimitriou
Elements of the Theory of Computation. Prentice-Hall, 1981.

[LS--84] Loeckx, J. and Sieber, K.
The Foundation of Program Verification. John Wiley, 1984.

[Man-74] Manna, Z.
Mathematical Theory of Computation. Mc Graw-Hill, 1974.

[McC-88] McCarty L.T.
Clausal Intuitionistic Logic II. Tableau Proof Procedures. The Jour. of Logic Programming 5, 93-132, 1988.

[Mil-78] Milner, R.
A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 17, 348-375, 1978.

[Mos-85] Mostowski, M.
Textbook of Logic Based on Mizar-MSE, m.s.Bialystok 1985.

[Nie-89] Nieva, S.
MIZAR/LOG: Una Implementación del sistema MIZAR-MSE en un Lenguaje de Programación Lógico. Actas del IV Congreso de Lenguajes Naturales y Lenguajes Formales, 1988 Vol. 2, 807-815.

[Nie-89] Nieva, S.
The Reasoner of MIZAR/LOG. Computerized Logic Teaching Bulletin, 2(1), 22-35, 1989.

[OS--88] Oppacher F. and Suen E.
A Tableau-Based Theorem Prover. Jour. of Automated Reasoning 4, 69-100, 1988.

[Pau-86] Paulson, L. C.
Natural Deduction as Higher-Order Resolution. J. Logic Programming, 1986:3, 237-258.

[Pau-87] Paulson, L. C.
Logic and Computation. Cambridge University Press, 1987.

[Plo-77] Plotkin, G.

LCF Considered as a Programming Language. Theoretical Computer Sci. 5, 223-257, 1977.

[PR--88] Prazmowski, K. and Rudnicki, P.
MIZAR-MSE Primer and User Guide. Tr 88-11, The University of Alberta, Dpt. of Computing Science, Edmonton, 1988.

[Rea-89] Reade, C.
Elements of Fundational Programming. Addison Wesley, 1989.

[Rog-67] Rogers, H.
Theory of Recursive Functions and Effective Computatibility. McGraw Hill, 1967.

[Sco-82] Scott, D. S.
Domains for Denotational semantics. ICALP'82. Aarhus. Denmark, 1982.

[Smu-68] Smullyan, R. M.
First-Order Logic. Springer Verlag, 1968.

[Sti-87] Stickel, M.E.
An Introduction to Automated Deduction. Fundamentals of Artificial Intelligence. Springer Verlag, 1987.

[Sto-77] Stoy, J.E.
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, 1977.

[TB--85] Trybulec, A. and Blair, H.
Computer aided reasoning. In R.Parikh, editor, Logic of Programs, LNCS 193. Springer Verlag, 1985.

[Wan-63] Wang, H.
Dominoes and the AEA Case of the Decision Problem. In: Mathematical Theory of Automata, Polytechnic Press, 23-55, 1963.

[Wri-85] Wrighton G.
Non Classical Logic Theorem Proving. Jour. of Automated Reasoning 1, 35-37, 1985.

[WWL-81] Wos, L., Winker, S.K and Lusk, E.
An automated reasoning system. AFIPS Conf. Proc.: Natl. Comp. Conf.,6987-702, AFIPS Press, 1981.

[Zuk-87] Zukowski, S.
Checking of Identities. Mizar News, Vol 2 No 1, Stockholm 1987.

ANEXO

```
/*****************************************************************/
/*                        MODULO OPERATOR                        */
/*****************************************************************/

:- module(operator,[]).

/* DECLARACION DE OPERADORES PROLOG */

:- op(495,fx,'**').

:- op(500,xfx,:).

:- op(500,xfy,$).

:- op(650,fx,thus).

:- op(550,xfy,::).

:- op(550,fx,assume).

:- op(550,fx,autom).

:- op(490,fx,prod).

:- op(600,xfx,by).

:- op(600,xfx,induc).

:- op(600,xfx,reduc).
```

```
/*************************************************************/
/*                      MODULO CONTROL                       */
/*************************************************************/


:- module(control,[miz_pr/0]).


:- consult(operator).

:- use_module(reasoner,[reasoner/3]).

:- use_module(prover,[prove/3]).

:- use_module(errors,[all_errors/1,msg_error/1]).

:- use_module(analysis,[read_item/1]).

:- use_module(library(sets),[list_to_set/2]).


/* CUERPO PRINCIPAL DEL PROGRAMA DE CONTROL DE LA PRUEBA
   SALVA Y RECUPERA TEORIAS, Y LLAMA A LOS MODULOS DE ANALISIS
   O AUTOMATIZACION PARA CADA PASO */

control_proof([]) :- !.

control_proof(Env) :-
                    read_item(Item),
                    verify_item(Env,Item,NewEnv),
                    control_proof(NewEnv).


verify_item(_,[exit],[]) :- !,
                    write('fin de sesion').

verify_item(Env,[],Env) :-!.

verify_item(AL$[],[save(T)],[]) :- !,
                    recordz(asig,AL,_),
                    save(T),
       write('teoria incorporada al entorno actual'),nl.

verify_item(AL1$[],[include(T)],AL$[]) :- !,
                    restore(T),
                    recorded(asig,AL2,Ref),
                    erase(Ref),
                    append(AL1,AL2,AL).

verify_item(Env,[autom L : F],NewEnv) :-
                    prove(Env,L:F,NE),!,
                    select_env(Env,NE,NewEnv).

verify_item(Env,Item,NewEnv) :-
                    reasoner(Env,Item,NE),!,
                    select_env(Env,NE,NewEnv).
```

```
/* TRATAMIENTO DE LOS ERRORES Y SELECCION DEL NUEVO ENTORNO*/

select_env(Env,NE,NewEnv) :-
                        is_ok->
                           NewEnv = NE;
                           NewEnv = Env.


is_ok :-                all_errors(ErrL),
                        list_to_set(ErrL,ErrS),
                        error_text(ErrS),
                        no_errors(ErrS).


no_errors(ErrS) :-      ErrS = []
                       ;
                        ErrS = [0].


error_text([]) :- !.

error_text([ErrorCode|ErrorL]) :-
                        msg_error(ErrorCode),
                        error_text(ErrorL).


/* INICIALIZACION DEL PROCESO */

miz_pr :- control_proof([]#[]).
```

```
/*************************************************************/
/*                      MODULO REASONER                      */
/*************************************************************/


:- module(reasoner,[reasoner/3,add_binding/3,thesis_of/2,
                    replace_thesis/3]).


:- use_module(errors,[write_error/1]).

:- use_module(checker,[check_consequence/2]).

:- use_module(induction,[induction/2,fixed/1]).

:- use_module(lambda,[check_reducing/3]).

:- use_module(prover,[prove/3]).

:- use_module(formulas,[negation/2,order_axiom/1,bot_axiom/1,
                        bool_axiom/1,monomorphic/1]).

:- use_module(substit,[subs/3,subs_type/4]).

:- use_module(library(basics),[append/3,memberchk/2]).

:- use_module(library(lists),[delete/3,subseq/3]).


/* PRUEBA DE LA CORRECCION DE CADA PASO DEL RAZONAMIENTO
   MODIFICANDO EL ENTORNO Y CREANDO CODIGOS DE ERROR */

reasoner(E,[],E) :- !.

reasoner(E,[I|IL],NE) :-
                       reduces(E,I,ME),
                       nonvar(ME),
                       reasoner(ME,IL,NE).

reasoner(E,_,E).


reduces(E,L : S,NE) :- !,
                       sentence(S),
                       add_binding(L : S,E,NE).

reduces(E,assume L : S,NE) :- !,
                      (monomorphic(S),
                       thesis_of(E,Th),
                       replace_thesis(S,Th,S1),
                       obtain_conseq(Th,S1,NTh),
                       constructor_of_thesis(E,NTh,ME),
                       add_binding(L : S,ME,NE)
                      ;
                       write_error(assume)).
```

```
reduces(E,let(const(Ty,M)),NE) :- !,
                      (thesis_of(E,[[pt,Ty,F]]),
                       subs(const(Ty,M),F,S),
                       constructor_of_thesis(E,S,NE)
                      ;
                       write_error(let)).

reduces(E,let(Ty,v(TyV)),NE) :- !,
                      (thesis_of(E,Th),
                       subs_type(Ty,TyV,Th,NTh),
                       constructor_of_thesis(E,NTh,NE)
                      ;
                       write_error(type)).

reduces(E,thus proof(T),NE) :- !,
                      (thesis_of(E,Th),
                       replace_thesis(T,Th,S),
                       remove_comp(S,Th,NTh),
                       constructor_of_thesis(E,NTh,ME),
                       construct_environ(ME,S,NE)
                      ;
                       write_error(thus)).

reduces(E,thus autom L:S,NE) :- !,
                      (prove(E,L:S,ME),!,
                       thesis_of(E,Th),
                       replace_thesis(S,Th,S1),
                       remove_comp(S1,Th,NTh),
                       constructor_of_thesis(ME,NTh,NE)
                      ;
                       write_error(thus)).

reduces(E,thus J,NE) :- !,
                      (justification(E,J,ME,S),
                       thesis_of(E,Th),
                       remove_comp(S,Th,NTh),
                       constructor_of_thesis(ME,NTh,NE)
                      ;
                       write_error(thus)).

reduces(E,proof(T),NE) :- !,
                       thesis_of(E,Th),
                       replace_thesis(T,Th,T1),
                       construct_environ(E,T1,NE).

reduces(E,end(L : S),NE) :- !,
                      (E = _#[] :: RE,
                       thesis_of(RE,Th),
                       replace_thesis(S,Th,S1),
                       add_binding(L : S1,RE,NE)
                      ;
                       write_error(end)).

reduces(E,J,NE) :- !,justification(E,J,NE,_)
                      ;
                       write_error(justify).
```

```
/* DEMOSTRACIONES HACIA ADELANTE */

/* Justificacion por el axioma de punto fijo */

justification(E,L : S by fix,NE,S1) :- !,
                     (thesis_of(E,Th),
                      replace_thesis(S,Th,S1),
                      fixed(S1),!,
                      add_binding(L : S1,E,NE)
                     ;
                      write_error(fix),
                      fail).

/* Justificacion por referencia: llamada al checker */

justification(E,L : S by LL,NE,S1) :- !,
                     (thesis_of(E,Th),
                      replace_thesis(S,Th,S1),
                      look_up(LL,E,SL),
                      check_consequence(SL,S1),!,
                      add_binding(L : S1,E,NE)
                     ;
                      write_error(by),
                      fail).

/* Justificacion por induccion */

justification(E,L : S induc LL,NE,S1) :- !,
                     (thesis_of(E,Th),
                      replace_thesis(S,Th,S1),
                      look_up(LL,E,SL),
                      induction(S1,SL),
                      add_binding(L : S1,E,NE)
                     ;
                      write_error(induc),
                      fail).

/* Justificacion por reduccion de una lambda expresion  */

justification(E,L : S reduc [FE,L1],NE,S2) :- !,
                     (thesis_of(E,Th),
                      replace_thesis(S,Th,S2),
                      find(L1 : S1,E),
                      check_reducing(S1,S2,FE),
                      add_binding(L : S2,E,NE)
                     ;
                      write_error(reduc),
                      fail).


sentence([X:_]) :-    X=[_:_].


/* BUSQUEDA DE LAS REFERENCIAS EN LA ESTRUCTURA DE CONTEXTOS */

look_up([],_,[]) :- !.
```

```
look_up([order:RL],E,SL) :- !,
                        order_axiom(OA),
                        look_up(RL,E,RS),
                        append(OA,RS,SL).

look_up([bot_order:RL],E,SL) :- !,
                        bot_axiom(BA),
                        look_up(RL,E,RS),
                        append(BA,RS,SL).

look_up([bool:RL],E,SL) :- !,
                        bool_axiom(BA),
                        look_up(RL,E,RS),
                        append(BA,RS,SL).

look_up([L:RL],E,[S:RS]) :-
                        find(L : S,E),
                        look_up(RL,E,RS),!
                       ;
                        write_error(find),
                        fail.


find(L : S,AL#_ :: RE) :-
                        memberchk(L : S,AL)
                       ;
                        find(L : S,RE).

find(L : S,AL#_ ) :-  memberchk(L : S,AL).


/* INCORPORA UNA REFERENCIA AL ENTORNO ACTUAL */

add_binding(L:S,AL#Th::RE,[L:S:AL]#Th::RE).

add_binding(L:S,AL#Th,[L:S:AL]#Th).


/* BUSQUEDA DE LA TESIS EN EL ENTORNO ACTUAL */

thesis_of(_#Th :: _,Th) :- !.

thesis_of(_#Th,Th).


/* SUSTITUYE LA PALABRA TESIS EN UNA FORMULA POR LA TESIS
   ACTUAL */

replace_thesis([],_,[]) :- !.

replace_thesis([F:FL],Th,RF) :-
                        replace_th1(F,Th,RF1),
                        replace_thesis(FL,Th,RF2),
                        append(RF1,RF2,RF).
```

```
replace_th1([thesis],Th,Th) :- !.


replace_th1([n,S],Th,[[n,RS]]) :- !,
                        replace_thesis(S,Th,RS).

replace_th1([pt,Ty,F],Th,[[pt,Ty,RF]]) :- !,
                        replace_thesis(F,Th,RF).

replace_th1(S,_,[S]).


/* MODIFICACION DE UNA ESTRUCTURA DE ENTORNOS AL CREAR UN
   ENTORNO LOCAL O AL INCORPORAR LA TESIS AL ENTORNO LOCAL*/

construct_environ(E,S,[]#S::E).


constructor_of_thesis(AL#_ :: RE,S,AL#S :: RE) :- !.

constructor_of_thesis(AL#_,S,AL#S).


/* ELIMINA COMPONENTES DE UNA CONJUNCION */

remove_comp(S,Th,NTh) :-
                        subseq(Th,S,NTh).


/* ELIMINA EL ANTECEDENTE DE UNA IMPLICACION */

obtain_conseq(Imp,Ant,Con) :-
                        negation(Imp,NI),
                        remove_comp(Ant,NI,NCon),
                        negation(NCon,Con).
```

```
/*************************************************************/
/*                        MODULO PROVER                      */
/*************************************************************/

:- module(prover,[prove/3]).

:- use_module(reasoner,[add_binding/3,thesis_of/2,
                        replace_thesis/3]).

:- use_module(errors,[write_error/1]).

:- use_module(substit,[subs/3]).

:- use_module(formulas,[order_axiom/1,bot_axiom/1,bool_axiom/1,
                        negation/2]).

:- use_module(substit,[do_new_monomorphic/2]).

:- use_module(checker,[check_consequence/2]).

:- use_module(lambda,[reducing/2]).

:- use_module(induction,[induction/2]).

:- use_module(library(basics),[append/3]).


/* PRUEBA DE UNA FORMULA DE FORMA AUTOMATICA */

prove(E,L:F,NE) :-  thesis_of(E,Th),
                    replace_thesis(F,Th,F1),
                    construct_proof(E,F1,1),
                    add_binding(L:F1,E,NE),
                    write_error(ok)
                   ;
                    write_error(prover),NE = E.

/* CONSTRUCCION DE UNA DEMOSTRACION PROBANDO CON LAS DIFERENTES
   TECNICAS */

construct_proof(_,[],_) :- !.

construct_proof(E,[F:RF],N) :-
                    do_new_monomorphic([F:RF],[MF:MRF]),
                    select_tech(E,MF,NF,NE,N),
                    N1 is N + 1,
                    construct_proof(NE,NF,N1),
                    N2 is N1 + 1,
                    construct_proof(E,MRF,N2).

construct_proof(E,F,_) :-
                    insert_bool(E,AL),
                    check_consequence(AL,F).

construct_proof(E,F,_) :-
                    insert_bot(E,AL),
                    check_consequence(AL,F).
```

```
construct_proof(E,F,_) :-
                    insert_order(E,AL),
                    check_consequence(AL,F).


select_tech(E,[pt,Ty,F],NF,E,N) :-
                    subs(const(Ty,i(N)),F,NF).

select_tech(E,[n,[F1|F2]],NF2,NE,_) :-
                    negation(F2,NF2),
                    add_binding(**n:[F1],E,NE).

select_tech(E,F,NF,E,_) :-
                    reducing([F],NF).

select_tech(E,F,[],E,_) :-
                    fixed([F]).

select_tech(E,F,[],E,_) :-
                    elim_label_env(E,AL),
                    check_consequence(AL,[F]).


/* ADAPTA EL ENTORNO INICIAL, ELIMINANDO TESIS Y ETIQUETAS, E
   INCORPORANDO LOS AXIOMAS DE ORDEN */

insert_bool(E,AL) :-
                    bool_axiom(Bool),
                    elim_label_env(E,AL1),
                    append(Bool,AL1,AL).

insert_bot(E,AL) :-
                    bot_axiom(Bot),
                    elim_label_env(E,AL1),
                    append(Bot,AL1,AL).

insert_order(E,AL) :-
                    order_axiom(Or),
                    elim_label_env(E,AL1),
                    append(Or,AL1,AL).


elim_label_env(A#_::RE,AL) :- !,
                    elim_label(A,AL1),
                    elim_label_env(RE,AL2),
                    append(AL1,AL2,AL).

elim_label_env(A#_,AL) :-
                    elim_label(A,AL).


elim_label([],[]) :- !.

elim_label([_:F|RA],[F|RF]) :-
                    elim_label(RA,RF).
```

```
/*************************************************************/
/*                        MODULO CHECKER                     */
/*************************************************************/

:- module(checker,[check_consequence/2]).


:- use_module(equal,[check_equality/2]).

:- use_module(formulas,[terms/2,depth/4,type_of/2,negation/2,
              alfa_formula/3,beta_formula/3,delta_formula/3,
              gamma_formula/3]).

:- use_module(substit,[subs/3,instance_set/2,
                                    do_new_monomorphic/2]).

:- use_module(library(basics),[append/3,member/2,nonmember/2]).

:- use_module(library(sets),[union/3]).


/* VALIDACION DE UNA JUSTIFICACION POR REFERENCIA LLAMADA
   AL MODULO EQUAL O CONSTRUCCION DE UN TABLEAU */

check_consequence(SL,S) :-
                    check_equality(SL,S),!
                   ;
                    refutation(SL,S).


refutation(SL,S) :- negation(S,NS),
                    do_new_monomorphic(NS,MS),!,
                    closed_tableau([MS:SL]).

/* BUSQUEDA DE UN TABLEAU CERRADO */

closed_tableau(FS) :-
                    closed_branch(FS),!
                   ;
                    copy_term(FS,FS1),
                    instance_set(FS1,MFS),
                    classify(MFS,AL,BL,DL,GL,LL,TL),
                    close(AL,BL,DL,GL,LL,TL).

/* BUSQUEDA DE UN TABLEAU PROPOSICIONAL CERRADO */

close([],[],DL,GL,LL,TL) :- !,
                    depth(DL,GL,TL,DP),
                    close_m([],[],DL,GL,LL,TL,0,DP).

close([],[B:BL],DL,GL,LL,TL) :- !,
                    beta_formula(B,B1,B2),
                    confront_beta(B1,B2,BL,DL,GL,LL,TL).

close([A:AL],BL,DL,GL,LL,TL) :-
                    alfa_formula(A,A1,A2),
                    confront_alfa(A1,A2,AL,BL,DL,GL,LL,TL).
```

```
/* ALFA EXTENSION */

confront_alfa(A,A,AL,BL,DL,GL,LL,_) :-
                    opposite_t(A,AL,BL,DL,GL,LL),!.

confront_alfa(A,A,AL,BL,DL,GL,LL,TL) :- !,
                    place(A,AL,BL,DL,GL,LL,NAL,NBL,NDL,NGL,NLL),
                    close(NAL,NBL,NDL,NGL,NLL,TL).

confront_alfa(A1,A2,AL,BL,DL,GL,LL,TL) :-
                    opposite_t(A1,AL,BL,DL,GL,LL),!
                   ;
                    opposite_t(A2,AL,BL,DL,GL,LL),!
                   ;
                    negation(A1,A2),!
                   ;
                    place(A1,AL,BL,DL,GL,LL,AL1,BL1,DL1,GL1,LL1),
               place(A2,AL1,BL1,DL1,GL1,LL1,NAL,NBL,NDL,NGL,NLL),
                    close(NAL,NBL,NDL,NGL,NLL,TL).


/* BETA BIFURCACION */

confront_beta(B,B,BL,DL,GL,LL,TL) :- !,
                    close_beta(B,BL,DL,GL,LL,TL).

confront_beta(B1,B2,BL,DL,GL,LL,TL) :-
                    close_beta(B1,BL,DL,GL,LL,TL),
                    close_beta(B2,BL,DL,GL,LL,TL).


close_beta(B,BL,DL,GL,LL,TL) :-
                    opposite_t(B,[],BL,DL,GL,LL),!
                   ;
                    place(B,[],BL,DL,GL,LL,NAL,NBL,NDL,NGL,NLL),
                    close(NAL,NBL,NDL,NGL,NLL,TL).


/* BUSQUEDA DEL TABLEAU DE PREDICADOS CERRADO */

close_m([],[],[],[],_,_,_,_) :- !,
                    fail.

close_m(_,_,_,_,_,_,_,0)  :- !,
                    fail.

close_m([],[],[],GL,LL,TL,I,DP) :- !,
                    confront_gamma(GL,LL,TL,I,DP).

close_m([],[],[D:DL],GL,LL,TL,I,DP) :- !,
                    delta_formula(D,Ty,F),
                    confront_delta(Ty,F,DL,GL,LL,TL,I,DP).

close_m([],[B:BL],DL,GL,LL,TL,I,DP) :- !,
                    beta_formula(B,B1,B2),
                    confront_beta_m(B1,B2,BL,DL,GL,LL,TL,I,DP).
```

```
close_m([A:AL],BL,DL,GL,LL,TL,I,DP) :-
                    alfa_formula(A,A1,A2),
                    confront_alfa_m(A1,A2,AL,BL,DL,GL,LL,TL,I,DP).


/* ALFA EXTENSION ESTANDO LAS FORMULAS GAMMA MARCADAS */

confront_alfa_m(A,A,AL,BL,DL,GL,LL,_,_,_) :-
                    opposite_t(A,AL,BL,DL,GL,LL),!.

confront_alfa_m(A,A,AL,BL,DL,GL,LL,TL,I,DP) :- !,
                    place(A,AL,BL,DL,GL,LL,NAL,NBL,NDL,NGL,NLL),
                    process_gamma(NAL,NBL,NDL,NGL,NLL,TL,I,DP).

confront_alfa_m(A1,A2,AL,BL,DL,GL,LL,TL,I,DP) :-
                    opposite(A1,A2),!
                   ;
                    opposite_t(A1,AL,BL,DL,GL,LL),!
                   ;
                    opposite_t(A2,AL,BL,DL,GL,LL),!
                   ;
                    place(A1,AL,BL,DL,GL,LL,AL1,BL1,DL1,GL1,LL1),
               place(A2,AL1,BL1,DL1,GL1,LL1,NAL,NBL,NDL,NGL,NLL),
                    process_gamma(NAL,NBL,NDL,NGL,NLL,TL,I,DP).


/* BETA BIFURCACION ESTANDO LAS FORMULAS GAMMA MARCADAS */

confront_beta_m(B,B,BL,DL,GL,LL,TL,I,DP) :- !,
                    close_beta_m(B,BL,DL,GL,LL,TL,I,DP).

confront_beta_m(B1,B2,BL,DL,GL,LL,TL,I,DP) :-
                    close_beta_m(B1,BL,DL,GL,LL,TL,I,DP),
                    close_beta_m(B2,BL,DL,GL,LL,TL,I,DP).


close_beta_m(B,BL,DL,GL,LL,TL,I,DP) :-
                    opposite_t(B,[],BL,DL,GL,LL),!
                   ;
                    place(B,[],BL,DL,GL,LL,NAL,NBL,NDL,NGL,NLL),
                    process_gamma(NAL,NBL,NDL,NGL,NLL,TL,I,DP).


/* DELTA EXTENSION ESTANDO LAS FORMULAS GAMMA MARCADAS */

confront_delta(Ty,F,DL,GL,LL,TL,I,DP) :-
                    subs(const(Ty,aux(I)),F,SF),
            close_delta(SF,DL,GL,LL,[const(Ty,aux(I))|TL],I,DP).


close_delta(F,DL,GL,LL,TL,I,DP) :-
                    opposite_t(F,[],[],DL,GL,LL),!
                   ;
                    I1 is I + 1,
                    new_depth(DP,DP1),
                    place(F,[],[],DL,GL,LL,NAL,NBL,NDL,NGL,NLL),
                    process_gamma(NAL,NBL,NDL,NGL,NLL,TL,I1,DP1).
```

```
/* GAMMA EXTENSION ESTANDO LAS FORMULAS GAMMA MARCADAS */

confront_gamma(GL,LL,TL,I,DP) :-
                    process_gamma([],[],[],GL,LL,TL,I,DP).

process_gamma(AL,BL,DL,GL,LL,TL,I,DP) :-
                    look_term(GL,TL,GL1,F),!,
                    close_gamma(F,AL,BL,DL,GL1,LL,TL,I,DP).

process_gamma(AL,BL,DL,[G:GL],LL,TL,I,DP) :-
                    append(GL,[G],NGL),
                    new_depth(DP,DP1),
                    close_m(AL,BL,DL,NGL,LL,TL,I,DP1).

look_term([[[[pt,Ty,F]],M]:GL],TL,NGL,SF) :-
                    member(T,TL),
                    type_of(T,Ty),
                    nonmember(T,M),!,
                    subs(T,F,SF),
                    append(GL,[[[[pt,Ty,F]],[T:M]]],NGL).

close_gamma(F,AL,BL,DL,GL,LL,TL,I,DP) :-
                    opposite_t(F,AL,BL,DL,GL,LL),!
                   ;
                    new_depth(DP,DP1),
                    place(F,AL,BL,DL,GL,LL,NAL,NBL,NDL,NGL,NLL),
                    close_m(NAL,NBL,NDL,NGL,NLL,TL,I,DP1).


/* DECRECE LA PROFUNDIDAD EN UNA UNIDAD */

new_depth(DP,DP1) :-
                   DP = 0 -> DP1 = DP; DP1 is DP - 1.


/* ANALIZA SI UNA RAMA ES CERRADA ESTUDIANDO SI EL CONJUNTO
   DE FORMULAS QUE LA CONSTITUYEN ES UN CONJUNTO NO COHERENTE */

closed_branch([F:FS]) :-
                    noncoherent(F)
                   ;
                    opposite(F,FS)
                   ;
                    closed_branch(FS).

opposite_t(F,AL,BL,DL,GL,LL) :-
                    noncoherent(F)
                   ;
                    opposite(F,AL)
                   ;
                    opposite(F,BL)
                   ;
                    opposite(F,DL)
                   ;
                    opposite_m(F,GL)
                   ;
                    opposite(F,LL).
```

```
opposite(F,[F1:FS]) :-
                    negation(F,F1),!
                   ;
                    gamma_formula(F1,_,_),
                    oppos_unif(F,F1),!
                   ;
                    opposite(F,FS).

opposite_m(F,[[G,_]:FS]) :-
                    negation(F,G),!
                   ;
                    oppos_unif(F,G),!
                   ;
                    opposite_m(F,FS).


noncoherent([[n,[[_,bot(_)]]]]).


oppos_unif(F,G) :-  subs_var(G,G1),!,
                    negation(F,G1).


subs_var([[pt,_,F]],SF) :- !,
                    subs(_,F,SF1),
                    subs_var(SF1,SF).

subs_var(F,F).


/* ORDENACION DE LAS FORMULAS SEGUN SU CLASE */

classify([],[],[],[],[],[],[]) :- !.

classify([G:FS],AL,BL,DL,[[G,[]]:GL],LL,TL) :-
                    gamma_formula(G,_,_),!,
                    terms(G,TL1),
                    classify(FS,AL,BL,DL,GL,LL,TL2),
                    union(TL1,TL2,TL).

classify([D:FS],AL,BL,[D:DL],GL,LL,TL) :-
                    delta_formula(D,_,_),!,
                    terms(D,TL1),
                    classify(FS,AL,BL,DL,GL,LL,TL2),
                    union(TL1,TL2,TL).

classify([B:FS],AL,[B:BL],DL,GL,LL,TL) :-
                    beta_formula(B,_,_),!,
                    terms(B,TL1),
                    classify(FS,AL,BL,DL,GL,LL,TL2),
                    union(TL1,TL2,TL).

classify([A:FS],[A:AL],BL,DL,GL,LL,TL) :-
                    alfa_formula(A,_,_),!,
                    terms(A,TL1),
                    classify(FS,AL,BL,DL,GL,LL,TL2),
                    union(TL1,TL2,TL).
```

```
classify([L:FS],AL,BL,DL,GL,[L:LL],TL) :-
                    terms(L,TL1),
                    classify(FS,AL,BL,DL,GL,LL,TL2),
                    union(TL1,TL2,TL).


place(G,AL,BL,DL,GL,LL,AL,BL,DL,[[G,[]]:GL],LL) :-
                    gamma_formula(G,_,_),!.

place(D,AL,BL,DL,GL,LL,AL,BL,NDL,GL,LL) :-
                    delta_formula(D,_,_),!,
                    union([D],DL,NDL).

place(B,AL,BL,DL,GL,LL,AL,NBL,DL,GL,LL) :-
                    beta_formula(B,_,_),!,
                    union([B],BL,NBL).

place(A,AL,BL,DL,GL,LL,NAL,BL,DL,GL,LL) :-
                    alfa_formula(A,_,_),!,
                    union([A],AL,NAL).

place(L,AL,BL,DL,GL,LL,AL,BL,DL,GL,NLL) :-
                    union([L],LL,NLL).
```

```
/*************************************************************/
/*                      MODULO EQUAL                         */
/*************************************************************/


:- module(equal,[check_equality/2]).


:- use_module(formulas,[vars_in/2,terms_in/2,type_of/2,terms/2]).

:- use_module(substit,[subs/3]).

:- use_module(library(sets),[union/2,subtract/3]).

:- use_module(library(lists),[delete/3]).

:- use_module(library(basics),[member/2,memberchk/2,append/3]).

:- use_module(library(not),['\='/2]).


/* ANALISIS DE LA CORRECCION DE LAS INFERENCIAS QUE UTILIZAN
   LAS PROPIEDADES DE LA IGUALDAD */

check_equality(SL,S) :-
                    equality(S,T1,T2),!,
                    look_equals(SL,S,EL),
                    check_equal(T1,T2,EL).

check_equality(SL,S) :-
                    look_equals(SL,S,EL),
                    member(S1,SL),
                    check_equal_subs(S,S1,EL).


/* SE OBTIENEN LAS IGUALDADES EXISTENTES EN UNA LISTA DE
   FORMULAS */

look_equals(SL,S,EL) :-
                    terms(S,TL),
                    list_of_equal(SL,TL,EL).


/* SE OBTIENE UNA LISTA DE IGUALDADES A PARTIR DE UNA LISTA
   DE FORMULAS */

list_of_equal([],_,[]) :- !.

list_of_equal([[[pt,Ty,F]|SL],TL,EL) :-
                    instance_all_terms([F,Ty],TL,E),!,
                    append(E,SL,NSL),
                    list_of_equal(NSL,TL,EL).

list_of_equal([E|SL],TL,[E|EL]) :-
                    equality(E,_,_),!,
                    list_of_equal(SL,TL,EL).
```

```
list_of_equal([_:SL],TL,EL) :-
                    list_of_equal(SL,TL,EL).
instance_all_terms(_,[],[]) :- !.

instance_all_terms([F,Ty],TL,EL) :-
                    copy_term([F,Ty],[F1,Ty1]),
                    member(T,TL),
                    delete(TL,T,TL1),
                    instance_term(F1,Ty1,T,E),
                    instance_all_terms([F,Ty],TL1,EL1),
                    append(E,EL1,EL).


instance_term(F1,Ty1,T,[E]) :-
                    T\=if(_,_,_),
                    type_of(T,Ty1),!,
                    subs(T,F1,E).

instance_term(_,_,_,[]).


/* DECIDE SI UNA FORMULA ES IGUALDAD DE DOS TERMINOS SIN
   VARIABLES LIBRES */

equality([['=',T1,T2]],T1,T2) :-
                    vars_in(T1,[]),
                    vars_in(T2,[]).


/* SE COMPRUEBA SI PUEDE ESTABLECERSE IGUALDAD ENTRE TERMINOS
   A PARTIR DE UNA LISTA DE IGUALDADES UTILIZANDO EL ALGORITMO
   DE ZUKOWSKY */

check_equal(T,T,_) :- !.

check_equal(prod[T1:TL1],prod[T2:TL2],EL) :- !,
                    check_equal(T1,T2,EL),
                    check_equal(prod TL1,prod TL2,EL).

check_equal(T1,T2,EL) :-
                    terms_in([[['=',T1,T2]]:EL],TL),
                    cond_to_func([T1,T2],[FT1,FT2]),
                    cond_to_func(TL,TL1),
                    trans_cond(EL,NEL),
                    partition(TL1,P),
                    algorithm(P,FP,NEL),
                    class(FT1,FP,C),
                    class(FT2,FP,C).


algorithm(P,FP,EL) :-
                    new_partition(P,EL,NP),
                    elim_clashes(NP,FP).
```

```
/* SE CREA UNA PARTICION INICIAL A PARTIR DE LA LISTA DE
   TODOS LOS TERMINOS EXISTENTES */

partition([],[]) :- !.

partition([X:Y],[[X]:Z]) :-
                    partition(Y,Z).


/* PASO DE UNA PARTICION OTRA */

new_partition(P,[],P) :- !.

new_partition(P,[[['=',T1,T2]]:EL],NP) :-
                    class(T1,P,C1),
                    class(T2,P,C2),
                    new_partition1(C1,C2,P,EL,NP).


new_partition1(C,C,P,EL,NP) :- !,
                    new_partition(P,EL,NP).

new_partition1(C1,C2,P,EL,NP) :-
                    join(C1,C2,P,P1),
                    new_partition(P1,EL,NP).


join(C1,C2,P,[C:NP]) :-
                    subtract(P,[C1,C2],NP),
                    append(C1,C2,C).


/* SE ELIMINAN LAS CLASES CON INTERSECCION NO VACIA */

elim_clashes(P,FP) :-
                    clashes(P,P,C1,C2),
                    join(C1,C2,P,NP),!,
                    elim_clashes(NP,FP)
                   ;
                    FP=P.


clashes(P1,[C:P],C,NC) :-
                    member(func(Ty,FE,T),C),
                    copy_term(func(Ty,FE,T),
                              func(Ty1,FE1,T1)),
                    class(func(Ty1,FE1,T2),P,NC),
                    class(T1,P1,CL),
                    class(T2,P1,CL).

clashes(P1,[_:P],C1,C2) :-
                    clashes(P1,P,C1,C2).


/* SE BUSCA EN UNA PARTICION LA CLASE DE UN TERMINO */

class(prod [],_,[]) :- !.
```

```
class(prod[T:TL],P,[C:CL]) :- !,
                    class(T,P,C),
                    class(prod TL,P,CL).

class(T,[C1:_],C) :-
                    memberchk(T,C1),
                    C=C1.

class(T,[_:P],C) :- class(T,P,C).


/* SE APLICA LA REGLA DE SUSTITUCION DEL CALCULO PARA
   COMPROBAR SI UNA FORMULA SE DEDUCE DE OTRA A PARTIR
   DE LA IGUALDAD DE DOS TERMINOS */

check_equal_subs([],[],_) :- !.

check_equal_subs([F1:LF1],[F2:LF2],EL) :-
                    check_equal_subs1(F1,F2,EL),
                    check_equal_subs(LF1,LF2,EL).


check_equal_subs1([pt,Ty,F1],[pt,Ty,F2],EL) :- !,
                    subs(const(Ty,*),F1,SF1),
                    subs(const(Ty,*),F2,SF2),
                    check_equal_subs(SF1,SF2,EL).

check_equal_subs1([n,F1],[n,F2],EL) :- !,
                    check_equal_subs(F1,F2,EL).

check_equal_subs1(['<',T1,T2],['<',S1,S2],EL) :- !,
                    check_equal(T1,S1,EL),
                    check_equal(T2,S2,EL).

check_equal_subs1([P,T1],[P,T2],EL) :-
                    check_equal(T1,T2,EL).


/* REPRESENTACION DEL CONDICIONAL COMO UN TERMINO FUNCIONAL */

trans_cond([],[]) :- !.

trans_cond([F:RF],[NF:NRF]) :-
                    trans_cond1(F,NF),
                    trans_cond(RF,NRF).

trans_cond1([[E,T1,T2]],[[E,NT1,NT2]]) :- !,
                    cond_to_func([T1,T2],[NT1,NT2]).


cond_to_func([],[]) :- !.

cond_to_func([T:TL],[T1:TL1]) :-
                    cond_to_func1(T,T1),
                    cond_to_func(TL,TL1).
```

```
cond_to_func1(X,X) :- var(X),!.

cond_to_func1(prod TL,prod TL1) :- !,
                    cond_to_func(TL,TL1).

cond_to_func1(func(Ty,FE,T),func(Ty,FE,T1)) :- !,
                    cond_to_func1(T,T1).

cond_to_func1(if(T,T1,T2),func(Ty,if,prod[FT,FT1,FT2])) :- !,
                    cond_to_func1(T,FT),
                    cond_to_func1(T1,FT1),
                    cond_to_func1(T2,FT2),
                    type_of(if(T,T1,T2),Ty).

cond_to_func1(X,X).
```

```
/*************************************************************/
/*                      MODULO INDUCTION                     */
/*************************************************************/


:- module(induction,[induction/2,fixed/1]).


:- use_module(formulas,[negation/2,free_fvar/2,type_of/2,
          monomorphic/1,monomorphic_type/1,differs/4]).

:- use_module(lambda,[apply/3]).

:- use_module(errors,[write_error/1]).


/* BUSQUEDA DEL PATRON DE LA FORMULA SOBRE LA QUE SE HACE
   INDUCCION Y ANALISIS DE SU CONTINUIDAD */

induction(F,Pr) :-  copy_term(F,F1),
                    induc_pattern(F1,Pr,FX,X),
                    continuous(FX,X).


/* BUSQUEDA DE UN ESQUEMA DE INDUCCION */

induc_pattern(F,Pr,FX,X) :-
                    induc_premis(Pr,FB,FX,FM),
                    differs(F,FB,fix(X,M),bot),
                    differs(F,FX,fix(X,M),X), atom(X),
                    subs_func(X,M,FX,FM).


/* BUSQUEDA DE LAS PREMISAS CORRESPONDIENTES A LA REGLA
   DE INDUCCION */

induc_premis([F1,F2],FB,FX,FM) :-
                    induc_step(F2,FX,FM),!,
                    FB = F1
                   ;
                    induc_step(F1,FX,FM),!,
                    FB = F2
                   ;
                    write_error(induc_step),
                    fail.


induc_step([[n,[X:Y]]],[X],NY) :-
                    negation(Y,NY).


/* INTERCAMBIA UNA EXPRESION FUNCIONAL POR OTRA REDUCIENDO LA
   EXPRESION OBTENIDA */

subs_func(_,_,[],[]) :- !.

subs_func(X,Y,[F:RF],[SF:SRF]) :-
```

```
                        subs1_func(X,Y,F,SF),
                        subs1_func(X,Y,RF,SRF).
subs1_func(_,_,X,X) :- !.

subs1_func(X,Y,[pt,Ty,F],[pt,Ty,SF]) :- !,
                        subs_func(X,Y,F,SF).

subs1_func(X,Y,[n,F],[n,SF]) :- !,
                        subs_func(X,Y,F,SF).

subs1_func(X,Y,[P,T1],[P,ST]) :- !,
                        subs1_func(X,Y,T1,ST).

subs1_func(X,Y,[E,T1,T2],[E,ST1,ST2]) :- !,
                        subs1_func(X,Y,T1,ST1),
                        subs1_func(X,Y,T2,ST2).

subs1_func(_,_,var(Ty,N),var(Ty,N)) :- !.

subs1_func(_,_,const(Ty,N),const(Ty,N)) :- !.

subs1_func(_,_,bot(Ty),bot(Ty)) :- !.

subs1_func(X,Y,func(Ty,X,T),ST) :- !,
                        subs1_func(X,Y,T,T1),
                        apply(Y,T1,ST),
                        type_of(ST,Ty).

subs1_func(X,Y,func(Ty,FE,T1),func(Ty,SFE,ST)) :- !,
                        subs1_func(X,Y,FE,SFE),
                        subs1_func(X,Y,T1,ST).

subs1_func(X,Y,prod[T1:TL],prod[ST:STL]) :- !,
                        subs1_func(X,Y,T1,ST),
                        subs_func(X,Y,TL,STL).

subs1_func(_,_,prod[],prod[]) :- !.

subs1_func(X,Y,if(T1,T2,T3),if(ST1,ST2,ST3)) :- !,
                        subs1_func(X,Y,T1,ST1),
                        subs1_func(X,Y,T2,ST2),
                        subs1_func(X,Y,T3,ST3).

subs1_func(X,Y,lambda(VL,T1),lambda(VL,ST)) :- !,
                        subs1_func(X,Y,T1,ST).

subs1_func(X,Y,fix(FV,FE),fix(FV,SFE)) :- !,
                        subs1_func(X,Y,FE,SFE).

subs1_func(_,_,X,X) :- atom(X).


/* SE COMPRUEBA SI LA FORMULA ES CONTINUA CON RESPECTO A
   UNA VARIABLE DE FUNCION */

continuous([],_) :- !.
```

```
continuous([F:RF],X) :-
                    continuous1(F,X),
                    continuous(RF,X),!.

continuous(_,_) :-  write_error(contin).


continuous1([pt,_,F],X) :- !,
                    continuous(F,X).

continuous1([n,[[pt,Ty,F]]],X) :- !,
                    monomorphic_type(Ty),
                    negation(F,NF),
                    continuous(NF,X).

continuous1([n,[['<',T1,T2]]],X) :- !,
                    free_fvar(X,T1),
                    \+ free_fvar(X,T2).

continuous1([n,[['=',T1,T2]]],_) :- !,
                    T1 = bot(_)
                    ;
                    T2 = bot(_).

continuous1([n,[F1:F2]],X) :- !,
                    monomorphic([F1]),
                    monomorphic(F2),
                    negation([F1],NF1),
                    negation(F2,NF2),
                    continuous(NF1,X),
                    continuous(NF2,X).

continuous1([_,T1,T2],X) :-
                    free_fvar(X,T1)
                    ;
                    free_fvar(X,T2).


/* APLICACION DEL AXIOMA DE PUNTO FIJO */

fixed([[pt,_,F]]) :- !,fixed(F).

fixed([[=,func(Ty,fix(X,M),T),func(Ty,NM,T)]]) :-
                    copy_term(fix(X,M),fix(X1,M1)),
                    copy_term(fix(X1,M1),fix(X2,M2)),
                    X1 = fix(X2,M2),
                    NM = M1.

fixed([[=,func(Ty,fix(X,M),T),RT]]) :-
                    copy_term(fix(X,M),fix(X1,M1)),
                    copy_term(fix(X1,M1),fix(X2,M2)),
                    X1 = fix(X2,M2),
                    apply(M1,T,RT),
                    type_of(RT,Ty).
```

```
/*************************************************************/
/*                      MODULO LAMBDA                        */
/*************************************************************/


:- module(lambda,[reducing/2,check_reducing/3,apply/3]).


:- use_module(formulas,[differs_term/4,type_of/2]).

:- use_module(library(lists),[same_length/2,select/4]).


/* BUSQUEDA DE LA ABSTRACCION A LA QUE SE LE APLICA LA
   BETA-REDUCCION */

reducing(S1,S2) :- search(func(Ty,lambda(VL,T),T1),S1,T2,S2),
                   type_of(T2,Ty),
                   apply(lambda(VL,T),T1,T2).


/* APLICACION DE LA BETA REDUCCION DEL LAMBDA CALCULO */

apply(lambda(VL,T),prod TL,RT) :- !,
                   same_length(VL,TL),
                   copy_term(lambda(VL,T),lambda(VL1,T1)),
                   replace_vars(VL1,TL,T1,RT).

apply(lambda([var(Ty,X)],T),T2,RT) :-
                   copy_term(lambda([var(Ty,X)],T),
                             lambda([var(Ty1,X1)],T1)),
                   type_of(T2,Ty1),
                   replace_var(X1,T2,T1,RT).


/* SUSTITUYE LAS VARIABLES LIGADAS DE UNA LAMBDA EXPRESION
   POR UNA SERIE DE TERMINOS */

replace_vars([],[],T,T) :- !.

replace_vars([var(Ty,X)|VL],[T1|TL],T,RT) :-
                   type_of(T1,Ty),
                   replace_var(X,T1,T,NT),
                   replace_vars(VL,TL,NT,RT).


replace_var(X,T1,T,RT) :-
                   X = T1,
                   RT = T.


/*BUSQUEDA DE LA EXPRESION LAMBDA QUE HA SIDO REDUCIDA */

search(X,[F1|F],Y,[F2|RF]) :-
                   search1(X,F1,Y,F2)
                  ;
                   search(X,F,Y,RF).
```

```
search1(X,[pt,Ty,F1],Y,[pt,Ty,F2]) :- !,
                    search(X,F1,Y,F2).

search1(X,[n,F1],Y,[n,F2]) :- !,
                    search(X,F1,Y,F2).

search1(X,F1,Y,F2) :-
                    select(X,F1,Y,F2).


/* COMPROBACION DE UNA BETA-REDUCCION */

check_reducing([[n,[[_,T,bot(_)]]],[['=',T1,T2]],FE) :-
                    differs_term(T1,T2,func(Ty,FE,T),T3),
                    type_of(T3,Ty),
                    apply(FE,T,T3).
```

```
/*****************************************************************/
/*                        MODULO FORMULAS                        */
/*****************************************************************/

:- module(formulas,[negation/2,alfa_formula/3,beta_formula/3,
             delta_formula/3,gamma_formula/3,depth/4,terms/2,
             type_vars_term/2,free_fvar/2,vars_in/2,type_of/2,
             monomorphic/1,monomorphic_type/1,type_vars/2,
             m_types_in/2,differs/4,differs_term/4,terms_in/2,
             order_axiom/1,bot_axiom/1,bool_axiom/1]).

:- use_module(substit,[instance_lambda/1]).

:- use_module(library(sets),[union/3,union/2]).

:- use_module(library(lists),[subseq/3,is_list/1]).

:- use_module(library(not),['\='/2]).


/* NEGACION DE UNA FORMULA */

negation([[n,X]],X) :- !.

negation(X,[[n,X]]).


/* RECONOCIMIENTO DE UNA FORMULA TIPO ALFA */

alfa_formula([['=',T1,T2]],[['<',T1,T2]],[['<',T2,T1]]) :- !.

alfa_formula([X|Y],[X],Y) :-
                        is_list(X),
                        Y\=[].


/* RECONOCIMIENTO DE UNA FORMULA TIPO BETA */

beta_formula([[n,[['=',T1,T2]]]],
             [[n,[['<',T1,T2]]]],[[n,[['<',T2,T1]]]]) :- !.

beta_formula([[n,[X|Y]]],W,Z) :-
                        Y\=[],
                        negation([X],W),
                        negation(Y,Z).


/* RECONOCIMIENTO DE UNA FORMULA TIPO DELTA */

delta_formula([[n,[[pt,X,F]]]],X,Y) :-
                        negation(F,Y).


/* RECONOCIMIENTO DE UNA FORMULA TIPO GAMMA */

gamma_formula([[pt,X,Y]],X,Y).
```

```
/* CALCULO DE LA PROFUNDIDAD EN FUNCION DEL NUMERO DE
   CUANTIFICADORES Y DEL NUMERO DE TERMINOS EXISTENTES
   EN TRES CONJUNTOS DE FORMULAS DL, GL, TL */

depth(DL,GL,TL,DP) :-  count_pt(DL,M),
                       count_pt_m(GL,N),
                       X is M + N,
                       length(TL,Y),
                       DP is 2*(X+1)*(Y+1).


/* CONTADOR DEL NUMERO DE CUANTIFICADORES DE UNA FORMULA */

count_pt([],0) :- !.

count_pt([F:FL],X) :-  count_pt1(F,X1),
                       count_pt(FL,X2),
                       X is X1 + X2.


count_pt1([],0) :- !.

count_pt1([F:FL],X) :- count_pt11(F,X1),
                       count_pt1(FL,X2),
                       X is X1 + X2.


count_pt11([pt,_,F],X1) :- !,
                       count_pt1(F,X),
                       X1 is X + 1.

count_pt11([n,[[pt,_,F]]],X1) :- !,
                       count_pt1(F,X),
                       X1 is X + 1.

count_pt11(_,0).


count_pt_m([],0) :- !.

count_pt_m([[F,_]:FL],X) :-
                       count_pt1(F,X1),
                       count_pt_m(FL,X2),
                       X is X1 + X2.


/* OBTIENE UNA LISTA DE TERMINOS A PARTIR DE UNA LISTA
   DE FORMULAS */

terms_in([],[]) :- !.

terms_in([F:FL],TL) :- terms(F,TL1),
                       terms_in(FL,TL2),
                       union(TL1,TL2,TL).
```

```
terms([],[]) :- !.

terms([F:RF],TL) :-     terms1(F,TL1),
                        terms(RF,TL2), union(TL1,TL2,TL).
terms1([pt,_,F],TL) :- !,
                        terms(F,TL).

terms1([n,F],TL) :- !,
                        terms(F,TL).

terms1([_,T1,T2],TL) :- !,
                        collect_term(T1,TL1),
                        collect_term(T2,TL2),
                        union(TL1,TL2,TL).

terms1([_,T],TL) :-     collect_term(T,TL).


/* OBTIENE UNA LISTA DE TERMINOS BASICOS Y VARIABLES LIBRES
   QUE APARECEN EN UN TERMINO */

collect_term_list([],[]) :- !.

collect_term_list([T:RT],CT) :-
                        collect_term(T,T1),
                        collect_term_list(RT,T2),
                        union(T1,T2,CT).


collect_term(X,[]) :-  var(X),!.

collect_term(const(Ty,N),[const(Ty,N)]) :- !.

collect_term(bot(Ty),[bot(Ty)]) :- !.

collect_term(prod TL,TS) :- !,
                        collect_term_list(TL,TS1),
                        compl_term_list(prod TL,TS1,TS).

collect_term(if(T1,T2,T3),TS) :- !,
                        collect_term_list([T1,T2,T3],TS1),
                        compl_term_list(if(T1,T2,T3),TS1,TS).

collect_term(func(_,X,T),TS) :-
                        var(X),!,
                        collect_term(T,TS).

collect_term(func(Ty,FE,T),TS) :- !,
                        collect_term(T,TS1),
                        compl_term_list(func(Ty,FE,T),TS1,TS).

collect_term(var(Ty,N),[var(Ty,N)]) :-
                        nonvar(N),
                        \+ integer(N).

collect_term(_,[]).
```

```
compl_term_list(T,TS1,TS) :-
                        vars_in(T,[])->
                        union([T],TS1,TS);
                        TS = TS1.

vars_in(X,[]) :- var(X),!.

vars_in(var(Ty,N),[var(Ty,N)]) :- !.

vars_in(prod[T:RT],VL) :- !,
                        vars_in(T,V1),
                        vars_in(prod RT,V2),
                        union(V1,V2,VL).

vars_in(if(T1,T2,T3),VL) :- !,
                        vars_in(T1,V1),
                        vars_in(T2,V2),
                        vars_in(T3,V3),
                        union([V1,V2,V3],VL).

vars_in(func(_,FE,T),VL) :- !,
                        vars_in(FE,V1),
                        vars_in(T,V2),
                        union(V1,V2,VL).

vars_in(lambda(V,T),VL) :- !,
                        copy_term(lambda(V,T),lambda(V1,T1)),
                        instance_lambda(V1),
                        vars_in(T1,VL).

vars_in(fix(_,FE),VL) :- !,
                        vars_in(FE,VL).

vars_in(_,[]).


/* DETECTA SI UNA VARIABLE X DE FUNCION APARECE LIBRE
   EN UNA EXPRESION */

free_fvar(_,Y) :-       var(Y),!,fail.

free_fvar(X,prod[T:RT]) :- !,
                        free_fvar(X,T)
                        ;
                        free_fvar(X,prod RT).

free_fvar(X,if(T1,T2,T3)) :- !,
                        free_fvar(X,T1)
                        ;
                        free_fvar(X,T2)
                        ;
                        free_fvar(X,T3).

free_fvar(X,func(_,FE,T)) :- !,
                        free_fvar_func(X,FE)
                        ;
                        free_fvar(X,T).
```

```
free_fvar_func(X,lambda(_,T)) :- !,
                        free_fvar(X,T).

free_fvar_func(X,fix(Y,FE)) :- !,
                        free_fvar_func(X,FE),
                        X \= Y.
free_fvar_func(X,X).


/* DETERMINA EL TIPO DE UN TERMINO */

type_of(var(Ty,_),Ty) :- !.

type_of(const(Ty,_),Ty) :- !.

type_of(func(Ty,_,_),Ty) :- !.

type_of(if(_,T1,T2),Ty) :- !,
                        type_of(T1,Ty),
                        type_of(T2,Ty).

type_of(bot(Ty),Ty) :- !.

type_of(prod[],[]) :- !.

type_of(prod[T:RT],[Ty:RTy]) :- !,
                        type_of(T,Ty),
                        type_of(prod RT,RTy).


/* DECIDE SI UNA FORMULA ES MONOMORFICA */

monomorphic([]) :- !.

monomorphic([F:RF]) :- monomorphic1(F),
                        monomorphic(RF).


monomorphic1([n,F]) :- !,
                        monomorphic(F).

monomorphic1([pt,Ty,F]) :- !,
                        monomorphic_type(Ty),
                        monomorphic(F).

monomorphic1([_,T1,T2]) :- !,
                        type_vars_term(T1,[]),
                        type_vars_term(T2,[]).

monomorphic1([_,T]) :- !,
                        type_vars_term(T,[]).


/* DECIDE SI UN TIPO ES MONOMORFICO */

monomorphic_type(TyV) :- var(TyV),!,fail.
```

```
monomorphic_type(ct(_,Ty)) :- !,
                        monomorphic_type(Ty).

monomorphic_type([]) :- !.

monomorphic_type([Ty|RTy]) :- !,
                        monomorphic_type(Ty),
                        monomorphic_type(RTy).
monomorphic_type(Ty) :-
                        atom(Ty).


/* OBTIENE UNA LISTA CON LOS TIPOS MONOMORFICOS DE
   UNA FORMULA */

m_types_in([],[]) :- !.

m_types_in([F|RF],MTy) :-
                        m_types1_in(F,Ty1),
                        m_types_in(RF,Ty2),
                        union(Ty1,Ty2,MTy).


m_types1_in([n,F],MTy) :- !,
                        m_types_in(F,MTy).

m_types1_in([pt,_,F],MTy) :- !,
                        m_types_in(F,MTy).

m_types1_in([_,T],MTy) :- !,
                        m_types_in_term(T,MTy).

m_types1_in([_,T1,T2],MTy) :- !,
                        m_types_in_term(T1,Ty1),
                        m_types_in_term(T2,Ty2),
                        union(Ty1,Ty2,MTy).


/* OBTIENE UNA LISTA CON LOS TIPOS MONOMORFICOS DE UN
   TERMINO */

m_types_in_term(X,[]) :-
                        var(X),!.

m_types_in_term(prod X,[]) :-
                        var(X),!.

m_types_in_term(prod[],[]) :- !.

m_types_in_term(prod [T|RT],MTy) :- !,
                        m_types_in_term(T,Ty1),
                        m_types_in_term(prod RT,Ty2),
                        union(Ty1,Ty2,Ty3),
                        collect_m_types(Ty3,MTy).
```

```
m_types_in_term(func(Ty,FE,T),MTy) :- !,
                        m_types_in_func(FE,Ty1),
                        m_types_in_term(T,Ty2),
                        union([Ty|Ty1],Ty2,Ty3),
                        collect_m_types(Ty3,MTy).

m_types_in_term(if(T1,T2,T3),MTy) :- !,
                        m_types_in_term(T1,MT1),
                        m_types_in_term(T2,MT2),
                        m_types_in_term(T3,MT3),
                        union([MT1,MT2,MT3],MTy).

m_types_in_term(T,MTy) :-
                        type_of(T,Ty),
                        collect_m_types1(Ty,MTy).


/* OBTIENE UNA LISTA CON LOS TIPOS MONOMORFICOS DE UNA
   EXPRESION FUNCIONAL */

m_types_in_func(X,[]) :-
                        var(X),!.

m_types_in_func(lambda(VL,T),MTy) :- !,
                        m_types_in_term(prod VL,MTy1),
                        m_types_in_term(T,MTy2),
                        union(MTy1,MTy2,MTy).

m_types_in_func(fix(_,FE),MTy) :- !,
                        m_types_in_func(FE,MTy).

m_types_in_func(_,[]).


/*OBTIENE LOS TIPOS MONOMORFICOS DE UNA LISTA DE TIPOS*/

collect_m_types([],[]) :- !.

collect_m_types([Ty|RTy],MTy) :-
                        collect_m_types1(Ty,MTy1),
                        collect_m_types(RTy,MTy2),
                        union(MTy1,MTy2,MTy).

collect_m_types1(TyV,[]) :-
                        var(TyV),!.

collect_m_types1(ct(N,Ty),[ct(N,Ty)|MTy]) :-
                        monomorphic_type(ct(N,Ty)),!,
                        collect_m_types1(Ty,MTy).

collect_m_types1(ct(_,Ty),MTy) :-
                        collect_m_types1(Ty,MTy).

collect_m_types1([],[]) :- !.
```

```
collect_m_types1([Ty:RTy],[Ty:MTy]) :-
                        monomorphic_type(Ty),!,
                        collect_m_types1(RTy,MTy).

collect_m_types1([_:RTy],MTy) :-
                        collect_m_types1(RTy,MTy).

collect_m_types1(Ty,[Ty]) :-
                        monomorphic_type(Ty),!.

collect_m_types1(_,[]).


/* OBTIENE UNA LISTA CON LAS VARIABLES DE TIPO DE UNA FORMULA */

type_vars([],[]) :- !.

type_vars([F:RF],TyV) :-
                        type_vars1(F,Ty1),
                        type_vars(RF,Ty2),
                        union(Ty1,Ty2,TyV).


type_vars1([n,F],TyV) :- !,
                        type_vars(F,TyV).

type_vars1([pt,_,F],TyV) :- !,
                        type_vars(F,TyV).

type_vars1([_,T],TyV) :- !,
                        type_vars_term(T,TyV).

type_vars1([_,T1,T2],TyV) :- !,
                        type_vars_term(T1,Ty1),
                        type_vars_term(T2,Ty2),
                        union(Ty1,Ty2,TyV).


/* OBTIENE UNA LISTA CON LAS VARIABLES DE TIPO DE UN TERMINO */

type_vars_term(X,[]) :-
                        var(X),!.

type_vars_term(bot(Ty),TyV) :- !,
                        type_vars_type(Ty,TyV).

type_vars_term(const(Ty,_),TyV) :- !,
                        type_vars_type(Ty,TyV).

type_vars_term(var(Ty,_),TyV) :- !,
                        type_vars_type(Ty,TyV).

type_vars_term(prod [T:RT],TyV) :- !,
                        type_vars_term(T,TV1),
                        type_vars_term(prod RT,TV2),
                        union(TV1,TV2,TyV).
```

```
type_vars_term(prod [],[]) :- !.

type_vars_term(func(Ty,FE,T),TyV) :- !,
                        type_vars_type(Ty,Ty1),
                        type_vars_func(FE,Ty2),
                        type_vars_term(T,Ty3),
                        union([Ty1,Ty2,Ty3],TyV).

type_vars_term(if(T1,T2,T3),TyV) :-
                        type_vars_term(T1,Ty1),
                        type_vars_term(T2,Ty2),
                        type_vars_term(T3,Ty3),
                        union([Ty1,Ty2,Ty3],TyV).


/* OBTIENE UNA LISTA CON LAS VARIABLES DE TIPO DE UNA
   EXPRESION FUNCIONAL */

type_vars_func(X,[]) :-  var(X),!.

type_vars_func(lambda(VL,T),TyV) :- !,
                        type_vars_term(prod VL,TyV1),
                        type_vars_term(T,TyV2),
                        union(TyV1,TyV2,TyV).

type_vars_func(fix(_,FE),TyV) :- !,
                        type_vars_func(FE,TyV).

type_vars_func(_,[]).


/* OBTIENE UNA LISTA CON LAS VARIABLES DE TIPO DENTRO DE
   UN TIPO */

type_vars_type(X,[]) :-
                        var(X),!.

type_vars_type(v(N),[N]) :- !.

type_vars_type(ct(_,Ty),TyV) :- !,
                        type_vars_type(Ty,TyV).

type_vars_type([Ty|RTy],TyV) :- !,
                        type_vars_type(Ty,Ty1),
                        type_vars_type(RTy,Ty2),
                        union(Ty1,Ty2,TyV).

type_vars_type(_,[]).


/* BUSCA LAS DIFERENCIAS SINTACTICAS ENTRE DOS FORMULAS */

differs([],[],_,_) :- !.

differs([F1|RF1],[F2|RF2],X,Y) :-
                        differs1(F1,F2,X,Y),
                        differs(RF1,RF2,X,Y).
```

```
differs1(X,X,_,_) :- !,fail.

differs1([pt,Ty,F1],[pt,Ty,F2],X,Y) :- !,
                        differs(F1,F2,X,Y).

differs1([n,F1],[n,F2],X,Y) :- !,
                        differs(F1,F2,X,Y).

differs1([P,T1],[P,T2],X,Y) :- !,
                        differs_term(T1,T2,X,Y).

differs1([E,T1,T2],[E,DT1,DT2],X,Y) :-
                        differs_term(T1,DT1,X,Y),
                        differs_term(T2,DT2,X,Y).


/* BUSCA LA DIFERENCIA SINTACTICA ENTRE DOS TERMINOS */

differs_term(X,X,_,_) :- !.

differs_term(prod[T1:RT1],prod[T2:RT2],X,Y) :- !,
                        differs_term(T1,T2,X,Y),
                        differs_term(prod RT1,prod RT2,X,Y).

differs_term(func(Ty,FE1,T1),func(Ty,FE2,T2),X,Y) :- !,
                        differs_term(T1,T2,X,Y),
                        differs_func(FE1,FE2,X,Y).

differs_term(if(T1,T2,T3),if(DT1,DT2,DT3),X,Y) :- !,
                        differs_term(T1,DT1,X,Y),
                        differs_term(T2,DT2,X,Y),
                        differs_term(T3,DT3,X,Y).

differs_term(X,Y,X,Y).


/* BUSCA LA DIFERENCIA SINTACTICA ENTRE DOS EXPRESIONES
   FUNCIONALES */

differs_func(X,Y,X,Y) :- !.

differs_func(lambda(VL,T1),lambda(VL,T2),X,Y) :- !,
                        differs_term(T1,T2,X,Y).

differs_func(fix(FV,FE1),fix(FV,FE2),X,Y) :- !,
                        differs_func(FE1,FE2,X,Y).

differs_func(X,X,_,_).
```

```
/* LISTA DE LA REPRESENTACION INTERNA DE LOS AXIOMAS DE ORDEN */

order_axiom([
             [[pt,X1,[[<,var(X1,1),var(X1,1)]]]],
             [[pt,X3,[[pt,X3,[[pt,X3,[[n,[
                  [<,var(X3,1),var(X3,2)],
                  [<,var(X3,2),var(X3,3)],
                  [n,[[<,var(X3,1),var(X3,3)]]]]]]]]]]],
             [[pt,X,[[pt,X,[[n,[
                  [<,var(X,1),var(X,2)],
                  [<,var(X,2),var(X,1)],
                  [n,[[=,var(X,1),var(X,2)]]]]]]]]]]
            ]).


bot_axiom([[[pt,X1,[[<,bot(X1),var(X1,1)]]]],
           [[pt,X2,[[pt,X2,[[n,[
                  [<,var(X2,1),var(X2,2)],
                  [n,[['=',var(X2,1),var(X2,2)]]],
                  [n,[['=',var(X2,1),bot(X2)]]]]]]]]]],
           [['=',func(X4,_,bot(_)),bot(X4)]],
           [[pt,Z,[['=',func(Y,bot,var(Z,1)),bot(Y)]]]]
          ]).


/* AXIOMAS DE IGUALDAD REFERENTES A LOS BOOLEANOS */

bool_axiom([[[pt,bool,[[n,[
                  [n,[['=',var(bool,1),const(bool,true)]]],
                  [n,[['=',var(bool,1),const(bool,false)]]],
                  [n,[['=',var(bool,1),bot(bool)]]]]]]]]],
            [[pt,B,[[pt,B,[['=',if(bot(bool),var(B,1),var(B,2)),
                  bot(B)]]]]]],
            [[pt,T,[[pt,T,
            [['=',if(const(bool,true),var(T,1),var(T,2)),
                  var(T,1)]]]]]],
            [[pt,F,[[pt,F,
            [['=',if(const(bool,false),var(F,1),var(F,2)),
                  var(F,2)]]]]]]
           ]).
```

```
/*****************************************************************/
/*                    MODULO SUBSTITUTION                        */
/*****************************************************************/


:- module(substitution, [subs/3, subs_type/4, instance_set/2,
                         do_new_monomorphic/2, instance_lambda/1]).


:- use_module(formulas, [type_vars/2, type_vars_term/2,
                          m_types_in/2]).

:- use_module(library(basics), [member/2]).

:- use_module(library(sets), [intersection/3, union/3]).


/* SUSTITUIR LA PRIMERA VARIABLE CUANTIFICADA POR UNA CONSTANTE
   EN UNA FORMULA Y EN UN TERMINO */

subs(_, [], []) :- !.

subs(T, [F|RF], [SF|SRF]) :-
                        subs1(T, F, SF),
                        subs(T, RF, SRF).


subs1(_, X, X) :-       var(X), !.

subs1(T, [pt, Ty1, F], [pt, Ty1, SF]) :- !,
                        subs(T, F, SF).

subs1(T, [n, F], [n, SF]) :- !,
                        subs(T, F, SF).

subs1(T, [P, T1], [P, ST]) :- !,
                        subs1(T, T1, ST).

subs1(T, [E, T1, T2], [E, ST1, ST2]) :- !,
                        subs1(T, T1, ST1),
                        subs1(T, T2, ST2).

subs1(T, var(_, 1), T) :- !.

subs1(_, const(Ty, N), const(Ty, N)) :- !.

subs1(_, bot(Ty), bot(Ty)) :- !.

subs1(T, func(Ty, FE, T1), func(Ty, SFE, ST)) :- !,
                        subs1(T, FE, SFE),
                        subs1(T, T1, ST).

subs1(T, prod TL, prod STL) :- !,
                        subs(T, TL, STL).
```

```
subs1(T,if(T1,T2,T3),if(ST1,ST2,ST3)) :- !,
                        subs1(T,T1,ST1),
                        subs1(T,T2,ST2), subs1(T,T3,ST3).

subs1(_,var(Ty,N),var(Ty,N)) :-
                        \+ integer(N),!.

subs1(_,var(Ty,N),var(Ty,N1)) :-
                        N1 is N - 1,!.

subs1(T,lambda(VL,T1),lambda(VL,ST)) :- !,
                        subs1(T,T1,ST).

subs1(T,fix(FV,FE),fix(FV,SFE)) :- !,
                        subs1(T,FE,SFE).

subs1(_,X,X).


/* SUSTITUIR UNA VARIABLE DE TIPO DENTRO DE UNA FORMULA Y DE UN
   TERMINO */

subs_type(_,_,[],[]) :- !.

subs_type(Ty,N,[F:RF],[SF:SRF]) :-
                        subs1_type(Ty,N,F,SF),
                        subs_type(Ty,N,RF,SRF).

subs1_type(_,_,X,X) :-var(X),!.

subs1_type(Ty,N,[pt,Ty1,F],[pt,STy1,SF]) :- !,
                        subs_type_type(Ty,N,Ty1,STy1),
                        subs_type(Ty,N,F,SF).

subs1_type(Ty,N,[n,F],[n,SF]) :- !,
                        subs_type(Ty,N,F,SF).

subs1_type(Ty,N,[P,T],[P,ST]) :- !,
                        subs1_type(Ty,N,T,ST).

subs1_type(Ty,N,[E,T1,T2],[E,ST1,ST2]) :- !,
                        subs1_type(Ty,N,T1,ST1),
                        subs1_type(Ty,N,T2,ST2).

subs1_type(Ty,N,var(Ty1,M),var(STy1,M)) :- !,
                        subs_type_type(Ty,N,Ty1,STy1).

subs1_type(Ty,N,const(Ty1,N),const(STy1,N)) :- !,
                        subs_type_type(Ty,N,Ty1,STy1).

subs1_type(Ty,N,bot(Ty1),bot(STy1)) :- !,
                        subs_type_type(Ty,N,Ty1,STy1).

subs1_type(Ty,N,func(Ty1,FE,T),func(STy1,SFE,ST)) :- !,
                        subs_type_type(Ty,N,Ty1,STy1),
                        subs1_type(Ty,N,FE,SFE),
                        subs1_type(Ty,N,T,ST).
```

```
subs1_type(Ty,N,prod TL,prod STL) :- !,
                        subs_type(Ty,N,TL,STL).

subs1_type(Ty,N,if(T1,T2,T3),if(ST1,ST2,ST3)) :- !,
                        subs1_type(Ty,N,T1,ST1),
                        subs1_type(Ty,N,T2,ST2),
                        subs1_type(Ty,N,T3,ST3).

subs1_type(Ty,N,lambda(VL,T),lambda(SVL,ST)) :- !,
                        subs_type(Ty,N,VL,SVL),
                        subs1_type(Ty,N,T,ST).

subs1_type(Ty,N,fix(FV,FE),fix(FV,SFE)) :- !,
                        subs1_type(Ty,N,FE,SFE).

subs1_type(_,_,X,X).


/* APLICAR UNA SUSTITUCION DE TIPO A UN TIPO */

subs_type_type(Ty,N,v(N),Ty) :- !.

subs_type_type(Ty,N,ct(M,Ty1),ct(M,STy)) :- !,
                        subs_type_type(Ty,N,Ty1,STy).

subs_type_type(Ty,N,[Ty1:RTy],[STy:SRTy]) :- !,
                        subs_type_type(Ty,N,Ty1,STy),
                        subs_type_type(Ty,N,RTy,SRTy).

subs_type_type(_,_,Ty,Ty).


/* HACER UNA FORMULA MONOMORFICA CON NUEVAS CONSTANTES DE TIPO */

do_new_monomorphic(F,MF) :-
                        type_vars(F,VL),
                        instance_types(F,VL,MF).


instance_types(F,[],F) :- !.

instance_types(F,[V:VL],MF) :-
                        subs_type(V,V,F,NF),
                        instance_types(NF,VL,MF).


/* HACER UNA FORMULA (UN TERMINO) MONOMORFICA CON TIPOS
   PERTENECIENTES A UN CONJUNTO */

do_monomorphic(F,MTy,MF) :-
                        type_vars(F,VL),
                        instance_with(F,VL,MTy,MF).


do_monomorphic_term(T,MTy,MT) :-
                        type_vars_term(T,VL),
                        instance_with_term(T,VL,MTy,MT).
```

```
instance_with(F,[],_,F) :- !.

instance_with(F,[V:VL],MTy,MF) :-
                        member(Ty,MTy),
                        subs_type(Ty,V,F,NF),
                        instance_with(NF,VL,MTy,MF).

instance_with_term(T,[],_,T) :- !.

instance_with_term(T,[V:VL],MTy,MT) :-
                        member(Ty,MTy),
                        subs1_type(Ty,V,T,NT),
                        instance_with(NT,VL,MTy,MT).


/* INSTANCIA LAS VARIABLES LIGADAS DE UNA ABSTRACCION */

instance_lambda(VL) :-
                        var_symb(VL,VS),
                        instance_var(VS,1).


var_symb([],[]) :- !.

var_symb([var(_,X):VL],[X:VS]) :-
                         var_symb(VL,VS).


instance_var([],_) :- !.

instance_var([V:VL],N) :-
                        V = N,
                        N1 is N + 1,
                        instance_var(VL,N1).


/* BUSCA LOS TIPOS MONOMORFICOS DE UN CONJUNTO DE FORMULAS
   E INSTANCIA SUS VARIABLES DE TIPO */

instance_set(FS,MFS) :-
                        m_types_in_set(FS,MTy),
                        do_monomorphic_set(FS,MTy,MFS).


m_types_in_set([],[]) :- !.

m_types_in_set([F:RF],MTy) :-
                        m_types_in(F,Ty1),
                        m_types_in_set(RF,Ty2),
                        union(Ty1,Ty2,MTy).

do_monomorphic_set([],_,[]) :- !.

do_monomorphic_set([F:RF],MTy,[MF:MRF]) :-
                        do_monomorphic(F,MTy,MF),
                        do_monomorphic_set(RF,MTy,MRF).
```

```
/*****************************************************************/
/*                        MODULO ERRORS                          */
/*****************************************************************/


:- module(errors,[all_errors/1,write_error/1,msg_error/1]).


/* OBTIENE UNA LISTA CON LOS CODIGOS DE ERROR GRABADOS EN LA
   BASE DE DATOS DESPUES DEL ANALISIS DE UN ITEM */

all_errors([EC|EL]) :-
                  recorded(error,EC,Ref),
                  erase(Ref),
                  all_errors(EL).

all_errors([]).


/* SE GRABA EN LA BASE DE DATOS EL CODIGO DE ERROR */

write_error(ok) :- !,
                  recordz(error,0,_).

write_error(assume) :- !,
                  recordz(error,10,_).

write_error(let) :- !,
                  recordz(error,15,_).

write_error(type) :- !,
                  recordz(error,20,_).

write_error(thus) :- !,
                  recordz(error,25,_).

write_error(end) :- !,
                  recordz(error,30,_).

write_error(justify) :- !,
                  recordz(error,35,_).

write_error(by) :- !,
                  recordz(error,40,_).

write_error(induc) :- !,
                  recordz(error,45,_).

write_error(contin) :- !,
                  recordz(error,50,_).

write_error(induc_step) :- !,
                  recordz(error,55,_).

write_error(reduc) :- !,
                  recordz(error,60,_).
```

```
write_error(find) :- !,
                  recordz(error,65,_).

write_error(fix) :- !,
                  recordz(error,70,_).

write_error(prover) :- !,
                  recordz(error,75,_).


msg_error(0) :- !,
          write('prueba completada con exito'),nl.

msg_error(10) :- !,
         write('la tesis actual.no es una implicacion'),nl,
         write('o no se asume el antecedente'),nl.

msg_error(15) :- !,
         write('la tesis actual no es una formula universal'),nl,
         write('o la constante introducida no es valida'),nl.

msg_error(20) :- !,
        write('error al instanciar una variable de tipo'),nl.

msg_error(25) :- !,
        write('esta formula no corresponde con ninguna'),nl,
        write('componente del objetivo actual'),nl.

msg_error(30) :- !,
        write('la prueba no ha sido finalizada'),nl.

msg_error(35) :- !,
        write('esta justificacion no ha sido aceptada'),nl.

msg_error(40) :- !,
        write('la formula concluida no ha podido ser inferida'),
        nl,write('a partir de las referencias dadas'),nl.

msg_error(45) :- !,
        write('error en la demostracion por induccion'),nl.

msg_error(50) :- !,
        write('la formula a la que se le aplica'),nl,
        write('induccion no es continua'),nl.

msg_error(55) :- !,
        write('las premisas no corresponden a la regla'),
        nl,write('de induccion'),nl.

msg_error(60) :- !,
        write('la beta-reduccion no ha podido realizarse'),nl.

msg_error(65) :- !,
        write('premisas inexistentes'),nl.
```

```
msg_error(70) :- !,
        write('el axioma de punto fijo no ha sido'),nl,
        write('aplicado correctamente'),nl.

msg_error(75) :- !,
        write('la prueba no ha podido realizarse'),nl,
        write('de forma automatica'),nl.
```