UNIVERSIDAD DE SEVILLA

Facultad de Matemáticas

Depto. de Álgebra, Computación, Geometría y Topología

FUNDAMENTOS DE LA PROGRAMACIÓN LÓGICA

Tesina presentada por
Miguel Angel Gutiérrez Naranjo
para optar al
Grado de Licenciatura

V$^{\underline{o}}$ B$^{\underline{o}}$ del Tutor

Fdo. José Antonio Alonso Jiménez
Profesor Titular de la
Universidad de Sevilla

Sevilla, 1 de Julio, 1994

A mis padres

Introducción

> *Logic programming has aimed to bring together and solve two distinct aims in logic and computing: making logic more practical and making computing more logical.* R.A. Kowalski y C.J. Hogger pág. 545, [32]

Comenzamos esta memoria con una breve presentación de lo que conocemos como Programación Lógica, sus orígenes, su evolución histórica y sus objetivos. Esta introducción concluye con un breve resumen del contenido de la presente memoria.

Historia

Remontándonos en el tiempo podemos tomar a Aristóteles (384-322 a.C.) y su teoría silogística como los precursores de la lógica matemática y en consecuencia de la Programación Lógica. La teoría silogística, que estudia una clase particular de implicaciones con dos premisas y una conclusión, también fue tratada por filósofos contemporáneos de Aristóteles y largamente estudiada en siglos posteriores, aunque no se produjeron innovaciones de interés hasta el siglo XVII con los trabajos de Descartes y Leibnitz.

Dos siglos después Boole dio un paso importante en el sistema de razonamiento aristotélico poniendo en relación la lógica y el álgebra. Los trabajos de Boole fueron modificados y ampliados más tarde por lógicos y matemáticos como Jevon, Peirce, Schroeder y Huntington entre otros.

Llegamos así a finales del siglo XIX y principios del XX con la revolución de la fundamentación de las Matemáticas gracias a los trabajos de Frege, Cantor, Peano, Russell y Whitehead entre otros que marcan el período más apasionante y de mayor actividad en la historia de la lógica matemática.

En el primer tercio del siglo XX y tras la publicación de los *Principia Mathematica* por parte de Russell y Whitehead continuó la actividad frenética de innovación de la mano de investigadores de la talla de Post, Hilbert, Ackerman, Brower, Gödel y por supuesto Skolem y Herbrand[1]

Llegamos a la mitad del siglo XX y descubrimos que de forma paralela al desarrollo de la lógica se ha producido un espectacular avance de las llamadas *"máquinas de calcular"*, avance sobre el que reflexiona A. Turing en un artículo titulado *"¿Pueden pensar las máquinas?"*, publicado en 1950 y que podemos dar como punto de partida de lo que después se llamará Inteligencia Artificial.

En los años siguientes a la publicación de este artículo avanzó mucho la demostración automática con trabajos como los de Gilmore, Davis-Putnam y Prawitz, tomando siempre como referencia los trabajos de Skolem y Herbrand sobre lógica de primer orden.

Un hito importante en la historia de la demostración automática lo marcó Robinson en 1965 con la presentación de su método de resolución, punto de partida para los trabajos de otros investigadores como Chang, Lee, Loveland y Wos.

No obstante no es hasta la primera mitad de los setenta, con los trabajos de Kowalski y el primer PROLOG de Colmerauer cuando nace la Programación Lógica como rama de la demostración automática con personalidad propia.

En palabras de Kowalski y Hogger (cf. [32], pag. 544)

"La Programación Lógica puede ser brevemente definida como el uso de la lógica simbólica para la representación de problemas y sus bases de conocimiento asociadas, junto con el control de la inferencia lógica para la solución efectiva de esos problemas."

Tras la presentación de la Programación Lógica de Kowalski en 1972 y la del primer PROLOG por parte de Colmerauer, 1973-75 los acontecimientos se han precipitado. Telegráficamente, podemos citar

1965 J. Alan Robinson publica su principio de resolución para la demostración automática en forma clausal.

1972 Robert A. Kowalski formula su interpretación de la lógica en forma clausal como lenguaje de programación.

1973 Alain Colmerauer, Philippe Roussell y otros implantan el primer sistema PROLOG en la Universidad de Aix- Marseille.

[1] M. Davis se culpa en su artículo *The Prehistory and Early History of Automated Deduction* [16] de dar un nombre erróneo a lo que hoy conocemos como universo y teorema de Herbrand, cuando en realidad deberían llamarse *de Skolem*. Al parecer, la primera vez que aparecen estos nombres es en un artículo suyo publicado en 1963 [17].

1974 Robert A. Kowalski presenta su programación con lógica de predicados en el IFIP-74.

1977 Keith L. Clark publica sus resultados relacionando la negación con el fallo finito.

1984 John W. Lloyd publica la primera edición de su libro sobre la fundamentación de la Programación Lógica, referencia básica en estos últimos 10 años.

1984 Se funda el *Journal of Logic Programming* primero bajo la dirección editorial de J. Alan Robinson y más tarde de Jean-Louis Lassez.

En la última década, desde la aparición del *Journal of Logic Programming* el número de artículos publicados se ha multiplicado, existiendo actualmente un considerable número de lógicos y matemáticos investigando y publicando artículos. Podemos citar entre otros nombres a Apt, Shepherdson, Lassez, Maher, Marriot, Hogger, Martelli, Montanari, Kunen o Pedreschi. Especial mención merece el artículo publicado por Apt y Doets este mismo año en el que redefinen conceptos clásicos sobre Información Negativa presentados por Lloyd, trabajo que incorporamos y comentamos en esta memoria.

Objetivos

Como apuntábamos antes en palabras de Kowalski y Hogger, la Programación Lógica estudia el uso de la lógica para el planteamiento de problemas y el control sobre las reglas de inferencia para alcanzar la solución de forma automática.

Así, planteado de la forma más general posible, La Programación Lógica intenta resolver la siguiente cuestión:

Dado un problema S, saber si la afirmación ϕ es solución o no del problema, o en qué casos lo es. Además queremos que los métodos sean implantables en máquinas de forma que la resolución del problema se haga de forma automática.

Los pasos a seguir en la consecución de nuestro objetivo los podemos separar en dos grandes grupos que podemos llamar *Representación* y *Técnicas de resolución*.

Representación

Los procesos de representación no pertenecen a la Programación Lógica propiamente dicha y de hecho muchos autores no los consideran. En esta fase lo que hacemos es representar la información suministrada en el planteamiento del problema de forma que podamos aplicar sobre él las técnicas de resolución automática.

El primer paso en el caso general sería transcribir un problema planteado en lenguaje natural a lenguaje formal. Hoy día este es un problema sin resolver y de hecho las investigaciones apuntan a que sólo un fragmento del lenguaje natural puede ser traducido[2]. No obstante muchas afirmaciones pueden ser traducidas a fórmulas de primer orden y de ellas nos ocupamos.

Una vez expresado nuestro problema mediante un conjunto finito de fórmulas de primer orden expresamos estas en FNC, las cuantificamos universalmente y nos preguntamos si nuestro problema tiene o no solución, y en su caso hallarla.

En los términos en que tenemos representado nuestro problema, esto es, mediante un conjunto de fórmulas S, preguntarse por una solución del problema es preguntarse si una determinada fórmula del tipo

$$\exists y_1 \dots \exists y_n (B_1 \wedge \dots \wedge B_m)$$

es consecuencia lógica de S[3]. Llamaremos ϕ a esta fórmula.

Damos un paso más, porque preguntarnos si ϕ es consecuencia lógica de S es preguntarse si el conjunto $S \cup \{\neg\phi\}$ es consistente o no.

Nos estamos acercando a nuestro objetivo. Puesto que nuestro problema ahora es saber si un determinado conjunto de fórmulas tiene o no modelos, aplicando el Teorema de Skolem y las equivalencias lógicas usuales, podemos reducir nuestro problema al siguiente problema equivalente:

Dado el conjunto de cláusulas $S' \cup \{\neg\phi\}$ estudiar su consistencia.
donde por cláusula entendemos una fórmula del tipo

$$\forall x_1 \dots \forall x_s (A_1 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee B_m)$$

con los A_i y B_j átomos y los x_k las variables que ocurren en ellos. Abreviaremos la notación de la fórmula anterior como

$$A_1 \dots A_n \leftarrow B_1 \dots B_m$$

Técnicas de resolución

Aquí empieza el verdadero estudio de la Programación Lógica y de hecho es el punto de partida de casi todos los trabajos publicados, obviando todo lo

[2]Las interjecciones parece imposible

[3]Denotamos por B_i a los literales apropiados y por y_j a las variables que ocurren en ellos.

anterior. Nuestro trabajo consiste en estudiar la consistencia de un conjunto finito de cláusulas.

En una primera aproximación vamos a estudiar al caso históricamente más importante. El caso en que las cláusulas en las que transcribimos la información de nuestro problema y que llamaremos cláusulas de programa definido sean del tipo

$$A \leftarrow B_1 \dots B_m$$

esto es, tengan un y sólo un literal positivo. Llamaremos objetivos definidos a las cláusulas sin literales positivos y llamaremos cláusulas de Horn tanto a las cláusulas de programa definido como a los objetivos definidos (cf. Sección 1.1). Nótese que en este caso la fórmula

$$\neg\exists y_1 \dots \exists y_n (B_1 \wedge \dots \wedge B_m)$$

es el objetivo $\leftarrow B_1, \dots, B_m$.

Sea como fuere, nuestro objetivo es estudiar la consistencia de un número finito de cláusulas. A primera vista el problema parece insalvable, puesto que tendríamos que ver que ninguna de las infinitas interpretaciones posibles es modelo de este conjunto de cláusulas.

Herbrand ya había solucionado nuestro problema en 1931 de forma muy ingeniosa, ya que redujo el problema a estudiar sólo un tipo concreto de interpretaciones, las interpretaciones de Herbrand, que toman como universo el propio conjunto de términos cerrados del lenguaje en que está escrito el programa. Además, como desarrollamos en el primer capítulo de esta tesina podemos identificar una interpretación de Herbrand con el conjunto de los átomos que son válidos en esa interpretación (Prop. 1.2.5). De esta forma podemos aplicar toda la potencia de la Teoría de Conjuntos en el estudio de la existencia de modelos de Herbrand. De hecho podemos definir un operador del conjunto de interpretaciones de Herbrand en sí mismo, el *operador consecuencia inmediata*, que para programas definidos es continuo, con el que podemos caracterizar fácilmente los modelos de Herbrand de un programa.

Nuestro problema no está ni mucho menos resuelto todavía. En general existen infinitas interpretaciones de Herbrand y el estudio que podemos hacer en términos conjuntistas necesita en muchos casos aritmética transfinita.

El método que presentamos en esta memoria para decidir de forma mecánica la consistencia o no de nuestro conjunto de cláusulas de programa definido es una variante del método de resolución de Robinson llamado SLD-resolución (Resolución *L*ineal para cláusulas de programa *D*efinido con función de *S*elección). Se basa en dos principios: resolución propiamente dicha y unificación.

La resolución es una regla de inferencia, la única utilizada en este tipo de problemas, tal que dadas dos cláusulas nos permite inferir otra. La vemos primero en el caso de la lógica proposicional. El principio es muy simple. Dado el objetivo que representa la negación de la posible respuesta

$$\leftarrow A_1, \ldots, A_s$$

seleccionamos un átomo A_m,

$$\leftarrow A_1, \ldots, A_m, \ldots, A_s$$

y buscamos una cláusula de nuestro programa definido que tenga a A_m como único literal positivo, esto es, una cláusula del tipo

$$A_m \leftarrow B_1, \ldots, B_q$$

De ambas deducimos el siguiente objetivo (que llamamos *objetivo derivado*)

$$\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_s$$

Llamamos derivación (o SLD-derivación) a una sucesión de pasos de derivación como el que hemos descrito.

Nuestro interés está en inferir mediante el uso de esta regla la cláusula vacía, la cláusula sin literales, que denotamos como $\Box$. Puesto que una cláusula es válida en una interpretación si contiene un literal válido, la cláusula vacía es insatisfacible y pensaremos en ella como una contradicción. Por tanto, cabe pensar que si conseguimos llegar a contradicción razonando a partir de nuestro conjunto de cláusulas, dicho conjunto es inconsistente, con lo que habremos resuelto nuestro problema.

En el caso en que ocurran variables en los literales de las cláusulas, tendremos que hacer uso de la unificación.

La unificación en sí misma es todo un mundo dentro de las Matemáticas y a ella dedicamos el segundo apéndice de esta memoria. Visto de forma muy somera, una *sustitución* es una aplicación del conjunto de variables de un lenguaje en el conjunto de sus términos, que se puede extender de forma única al conjunto de los términos del lenguaje. Así dos términos s y t son unificables si existe una sustitución σ tal que $\sigma(s) = \sigma(t)$. En tal caso decimos que σ es un unificador de s y t. El concepto de *unificador de máxima generalidad, umg,* es más controvertido y en los últimos años han aparecido distintas definiciones no todas equivalentes. La idea que intentan atrapar las distintas definiciones es simple, un umg es un unificador que no hace enlaces superfluos, sino los mínimos necesarios en la unificación, dejando los términos unificados en la instancia más general posible. En esta memoria seguimos la definición de

T. Nipkow, que consigue plasmar esta idea mediante relaciones de orden y equivalencia, consiguiendo además que el umg de dos términos sea único salvo permutación de variables.

Podemos describir ahora la resolución en la que hacemos uso de la unificación. Sea el objetivo

$$\leftarrow A_1, \ldots, A_m, \ldots, A_s$$

donde hemos seleccionado el átomo A_m. Consideremos ahora una cláusula del programa

$$A \leftarrow B_1, \ldots, B_q$$

tal que A yA_m sean unificables. Tomamos su umg θ e inferimos

$$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_s)\theta$$

En este caso también buscamos la cláusula vacía mediante una *derivación* o sucesión de pasos como el que acabamos de describir. Caso de obtenerla también habremos obtenido una sustitución σ composición de los distintos umg que hemos considerado en la derivación. La restricción de esta sustitución σ a las variables de nuestro primer objetivo se llama *respuesta computada.* Su interpretación es clara, la respuesta computada es la sustitución de variables que dan respuesta a nuestro problema mediante el método de resolución aplicado. Vemos un ejemplo clásico.

PROBLEMA: *Dados tres puntos cualesquiera x, y, z sabemos que si existe un arco que una x e y y un camino de y a z entonces existe un camino de x a z. Si consideramos que existe un camino de todo punto a sí mismo y un arco de b a c, ¿existe algún punto desde el que pueda trazar un camino hasta c?.*

Las hipótesis del problema con notación clausal las puedo transcribir

$$\begin{cases} camino(x, z) \leftarrow arco(x, y), camino(y, z) \\ camino(u, u) \leftarrow \\ arco(b, c) \leftarrow \end{cases}$$

Nos preguntamos entonces si de este programa puedo deducir

$$\exists v(camino(v, c))$$

y en su caso hallarlo. Consideramos el objetivo $\leftarrow camino(v, c)$ y la siguiente derivación[4]:

$$
\begin{array}{ll}
\leftarrow \mathbf{camino(v,c)} & camino(x,z) \leftarrow arco(x,y), camino(y,z) \\
\downarrow & \theta_1 = \{v/x, z/c\} \\
\leftarrow \mathbf{arco(x,y)}, camino(y,c) & arco(b,c) \leftarrow \\
\downarrow & \theta_2 = \{x/b, y/c\} \\
\leftarrow \mathbf{camino(c,c)} & camino(u,u) \leftarrow \\
\downarrow & \theta_3 = \{u/c\} \\
\square &
\end{array}
$$

La restricción de la composición $\theta_1, \theta_2, \theta_3$ a las variables del primer objetivo $\leftarrow camino(v,c)$ nos da como respuesta computada $\theta = \{v/b\}$, que parece una respuesta acertada, pero este ejemplo pone de manifiesto otros problemas:

1. ¿Qué ocurre si seleccionamos otro átomo en un objetivo?
2. ¿Qué ocurre si en la derivación buscamos la unificación con otra cláusula?
3. Y lo que es más importante, ¿cómo sabemos que la respuesta computada es realmente la respuesta que andamos buscando?

La contrapartida declarativa al concepto procedural de respuesta computada es el de *respuesta correcta*. Decimos que la sustitución θ es una respuesta correcta para el programa definido P y el objetivo $\leftarrow A_1, \ldots, A_n$ si $\forall((A_1 \wedge \ldots \wedge A_n)\theta)$ es consecuencia lógica de P. Obviamente, nuestra intención es demostrar que toda respuesta computada es correcta, esto es probar la adecuación de método de resolución. Este resultado fue demostrado por Clark en 1979 (Tma. 2.3.1). Por tanto dado un programa definido y un objetivo definido podemos afirmar que toda respuesta computada es solución a nuestro problema. Nos planteamos

[4]Marcamos en negrilla el átomo seleccionado

ahora la cuestión contraria, esto es: Si existe solución a nuestro problema, ¿podemos encontrarla con este método? La respuesta es afirmativa y nos la da el correspondiente teorema de completitud (Tma. 2.4.6), que afirma que si el conjunto de claúsulas $P \cup \{G\}$ con P un programa definido y G un objetivo definido es insatisfacible, entonces existe una SLD-refutación (o derivación terminando en la claúsula vacía) de $P \cup \{G\}$.

Tenemos por tanto que la SLD-resolución es un método adecuado y completo, pero en la práctica no es tan sencillo puesto que el teorema de completitud sólo afirma la existencia de la refutación, pero no cómo hallarla.

Aquí empieza una segunda fase dentro de la investigación en Programación Lógica de la que también ofrecemos los resultados más importantes.

¿Cómo encontramos esa SLD-refutación caso de que exista? Esta pregunta enlaza con las dos primeras preguntas que nos planteábamos tras el ejemplo.

1. ¿Qué ocurre en una derivación si seleccionamos otro átomo?
2. ¿Qué ocurre cuando unificamos con otra cláusula?

En relación con el problema de la selección de átomos Lloyd define una regla de computación R como una aplicación que asocia a cada objetivo definido uno de los átomos que ocurren en él. Esta definición es muy rígida y muy criticada por otros autores[5] puesto que en cada objetivo tenemos que seleccionar siempre el mismo átomo independientemente del momento de la derivación en que aparezca.

Aunque rígida, esta definición nos permite probar un resultado importante y es la independencia de la regla de computación (Tma. 2.5.7), esto es, si existe una SLD-refutación de $P \cup \{G\}$, entonces fijada una regla de computación R, existe una SLD-refutación de $P \cup \{G\}$ que toma R como regla de computación.

Este resultado nos permite fijar una regla de selección de átomos con la tranquilidad de que si existe refutación podremos hallarla con esa regla.

Por tanto el último problema que nos queda por atacar es saber qué ocurre cuando elegimos unificar con una claúsula u otra. En ese sentido definimos los SLD-árboles de $P \cup \{G\}$ que no son más que árboles enraizados en el objetivo G y cuyas ramas son SLD- derivaciones. El estudio de estos árboles pertenece más a la Teoría de Grafos que a la Programación Lógica, pero apuntamos un último resultado general (Tma. 2.6.6): *Dado un programa definido P y un objetivo definido G, entonces o bien todo SLD-árbol tiene infinitas ramas con éxito (es decir, SLD-refutaciones), o bien todo SLD-árbol tiene el mismo número finito de ramas de éxito.*

[5]Apt, por ejemplo, en [3].

Información negativa

Con el estudio que llevamos hasta ahora sólo podemos deducir información positiva, es decir, sólo los literales positivos pueden ser consecuencia lógica de un programa. Lo vemos con un ejemplo.

Sea **L** el lenguaje de primer orden sin símbolos de función n- arios ($n > 0$), con un único símbolo de predicado 1-ario: "alumno" y el siguiente conjunto de constantes *SC*=$\{$*Esther, Juan, Rosa, Miguel, Pepe*$\}$. Consideremos el siguiente programa P sobre **L**:
$alumno(Esther) \leftarrow$
$alumno(Juan) \leftarrow$
$alumno(Rosa) \leftarrow$
Supongamos que nuestro objetivo es probar que $\neg alumno(Miguel)$ es consecuencia lógica de P. Con las herramientas que tenemos hasta ahora no podemos, ya que podemos encontrar modelos de $P \cup \{\leftarrow \neg alumno(Miguel)\}$, pero tampoco podríamos probar $alumno(Pepe)$ ya que también existen modelos de $P \cup \{\leftarrow alumno(Pepe)\}$[6].

Para intentar solucionar este problema vamos a introducir otra regla de inferencia, la hipótesis del mundo cerrado, HMC: *Si un átomo cerrado A no es consecuencia lógica de un programa P entonces inferimos $\neg A$.*

Esta regla no es más que la generalización de la forma de razonar en una base de datos: La información que no se da de forma explícita se toma como falsa. Con esta regla de inferencia solucionamos nuestro problema ya que con ella podemos deducir información negativa. El problema es que la lógica de primer orden no es decidible y no existe ningún algoritmo que tomando como datos de entrada un programa P y un átomo cerrado A nos devuelva en tiempo finito si A es consecuencia lógica de P o no, ya que si el átomo cerrado A no es consecuencia lógica de P podemos entrar en un proceso infinito. En la práctica restringimos la aplicación de la HMC únicamente a aquellos átomos cerrados tales que al intentar hacer una refutación fallamos de forma finita.

Fallo finito

La regla de inferencia denominada *regla de fallo finito* (Sección 4.2) es menos potente que la HMC y nos permite inferir menos información, no obstante tiene la ventaja de ser fácilmente automatizable. Es la siguiente:

Dado un programa definido P y un objetivo definido $\leftarrow A$, si existe un SLD-árbol finito sin ramas con éxito de $P \cup \{\leftarrow A\}$ deducimos $\neg A$

[6]Ver sección 4.1 para los detalles.

La regla es fácilmente implantable una vez solucionado un pequeño problema. La regla habla de existencia, pero encontrar (o demostrar la existencia) de tal SLD-árbol no parece tarea fácil.

En ese sentido Lassez y Maher definieron una SLD- derivación *favorable* como una derivación que

- O bien es finita y no tiene éxito
- O bien cada átomo de la SLD-derivación es seleccionado en un número finito de pasos

y se define un SLD-árbol favorable como un SLD-árbol tal que todas sus ramas sean SLD-derivaciones favorables.

Con esa definición podemos probar el siguiente resultado (Tma. 4.2.13): *Dado un átomo cerrado A y un programa definido P, existe un SLD-árbol finito sin éxito de $P \cup \{\leftarrow A\}$ si y sólo si cada SLD-árbol favorable es finito y no contiene ramas con éxito.*

Este teorema muestra que la SLD-derivación es una implantación adecuada y completa de la regla de fallo finito.

Puesto que ahora tenemos un algoritmo de deducción de información negativa podemos dar un salto hacia conjuntos más generales de cláusulas: los programas normales.

Programas normales

Como vimos, una cláusula de programa definido era una cláusula de la forma

$$A \leftarrow B_1, \ldots, B_n$$

donde tanto A como los B_i eran literales positivos. Pues bien una cláusula de programa normal tiene la misma estructura, con la salvedad de que los literales B_i pueden ser negativos. Análogamente, un objetivo normal es una cláusula de la forma

$$\leftarrow B_1, \ldots, B_n$$

donde los B_i pueden ser literales negativos. Nuestro objetivo ahora sigue siendo el mismo: Demostrar la inconsistencia de un conjunto de cláusulas mediante procedimientos automatizables, el problema para utilizar la SLD- resolución está en que el literal seleccionado puede ser negativo, por lo que tendremos que modificar nuestro sistema de derivación. La idea es utilizar la regla de fallo finito en los casos en que seleccionemos literales negativos.

Para justificar su uso Clark [10] da un nuevo sentido a la interpretación de las cláusulas. Una cláusula del tipo

$$A \leftarrow B_1, \ldots, B_n$$

podemos interpretarla del siguiente modo: *A es válido si lo son $B_1, \ldots, B_n$.* Clark apunta una nueva interpretación: *A es válido si y sólo si lo son $B_1, \ldots, B_n$.*

La formalización de esta idea lleva consigo el desarrollo de una teoría auxiliar no exenta de controversia, puesto que tenemos que introducir un nuevo símbolo de predicado "=" junto con unos nuevos axiomas, los axiomas de la *teoría de igualdad* (Sec. 4.4).

Las fórmulas resultantes de la completación de P (esto es, la formalización del si y sólo si) junto con los axiomas de la teoría de igualdad constituyen un conjunto de fórmulas que llamaremos $comp(P)$. Se puede probar que $comp(P)$ no es más que una generalización de P en el sentido de que la información positiva que podemos obtener de $comp(P)$ es exactamente la misma que podemos obtener de P y además nos va a permitir deducir información negativa. El problema, como siempre, es hacer el proceso automático.

SLDNF-resolución

El método que vamos a seguir en el caso de programas y objetivos normales es la SLDNF-resolución, que no es otro que la SLD- resolución aumentada con la regla de fallo finito. Sólo imponemos una condición para poder probar más tarde resultados de adecuación y es que los literales negativos que seleccionemos deben ser cerrados[7]. La descripción formal del proceso destaca varios casos y utiliza inducción sobre lo que Lloyd [37] llama rango de la refutación.

No es más que un intento de formalización de una idea simple: Dado un programa normal P y un objetivo normal G el proceso a seguir es el siguiente:

1. Mientras que los literales seleccionados sean positivos aplicamos la SLD-derivación que hemos estudiado.

2. Cuando seleccionamos un literal negativo, $L_i \equiv \neg A$ del objetivo

   $$\leftarrow L_1 \ldots L_{i-1}, L_i, L_{i+1}, \ldots, L_n$$

   intentamos encontrar un SLDNF-árbol finito sin éxito de $P \cup \{\leftarrow A\}$. Si existe deducimos

   $$\leftarrow L_1 \ldots L_{i-1}, L_{i+1}, \ldots, L_n$$

[7] Llamamos a esa condición *"condición de seguridad"*, Cf. 4.5.1.

y continuamos nuestra derivación. Si no existe paramos ahí y consideramos que nuestra derivación no tiene éxito.

Por supuesto, puede ocurrir que en nuestro SLDNF-árbol, que no es más que un árbol cuyas ramas son SLDNF-derivaciones, seleccionemos en algún momento un literal negativo. No importa, volvemos a tomar otro SLDNF-árbol auxiliar. El número de veces que tenemos que hacer eso es lo que Lloyd llama *rango de la SLDNF-derivación.*

Lo vemos más claro con un ejemplo. Sea P el programa normal

$$\begin{array}{ll} 1. & p(x) \leftarrow q(x), \neg r(a) \\ 2. & r(y) \leftarrow s(y) \\ 3. & q(b) \leftarrow \end{array}$$

y el objetivo normal $\leftarrow p(z)$. Veamos un esquema de una SLDNF-derivación de $P \cup \{\leftarrow p(z)\}$. (Marcamos el literal seleccionado en negrita)

$$\begin{tikzcd}
\leftarrow \mathbf{p(z)} \arrow[d, "1"', "\theta_1 = \{z/x\}"] & \\
\leftarrow q(x), \neg\mathbf{r(a)} \arrow[r] \arrow[d] & \leftarrow \mathbf{r(a)} \arrow[d, "2"', "\sigma = \{y/a\}"] \\
\leftarrow q(x) \arrow[d, "3"', "\theta_2 = \{x/b\}"] & \leftarrow \mathbf{s(a)} \\
\square & \textit{fallo}
\end{tikzcd}$$

Las definiciones de respuestas correcta y computada se traspasan de forma natural a los programas y objetivos normales. La pregunta es ahora si las respuesta obtenidas mediante SLDNF- resolución son correctas. La solución a esta cuestión la dio Clark en su teorema de adecuación de la SLDNF-resolución (Tma. 4.6.5): *Sea P un programa normal y G un objetivo normal. Entonces* *toda respuesta computada de* $P \cup \{G\}$*es una respuesta correcta de* $comp(P) \cup \{G\}$. ¿Y la completitud? ¿Podemos probar que si existe alguna sustitución que sea respuesta correcta la encontraremos mediante la SLDNF-resolución? No se sabe. Aún no tenemos resultados generales. Se ha probado para determinados tipos de programas pero aún no tenemos un resultado general. En esa línea va el artículo publicado por Apt y Doets en el último número del *Journal of Logic Programming.* En su artículo afirman que aún no se han obtenido resultados generales porque no existen definiciones rigurosas de muchos de los conceptos estudiados y una gran cantidad de trabajos publicados buscan más la rápida aplicación práctica que la fundamentación teórica. Los detalles sobre sus nuevas definiciones están en la sección 4.8 y sólo comentar que pese a su esfuerzo por dar definiciones formales, queda mucho por hacer dentro de este campo.

Para terminar un último comentario general sobre la Programación Lógica y es el tema del que trata el último capítulo de esta memoria: la conexión que existe entre la Programación Lógica y la Teoría de la Recursión. Se puede probar, y de hecho presentamos dos resultados importantes[8], que una función es recursiva si y solo si existe un programa definido que la compute.

Breve descripción del contenido

El presente trabajo intenta unificar criterios, condensar en un único trabajo la información existente sobre esta materia y presentarla de forma ordenada.

El *Capítulo I* presenta la introducción clásica a la Programación Lógica a través de los modelos y teorema de Herbrand, a la vez que nos introduce en los programas más sencillos sobre los que trata la Programación Lógica: los programas definidos. El capítulo termina con la presentación del *operador consecuencia inmediata* y una colección de resultados para los que necesitaremos aritmética transfinita.

El *Capítulo II* entra ya de lleno en los métodos de resolución de los problemas presentados. Presentamos la SLD-resolución junto con los resultados correspondientes de adecución y completitud desde un punto de vista teórico, sin hacer referencia a los métodos de control que tienen estos sistemas en sus implantaciones prácticas (el “corte” en PROLOG, por ejemplo).

El *Capítulo III* profundiza en la relación existente entre las funciones recursivas y la Programación Lógica demostrando que las funciones recursivas pueden ser computadas por programas lógicos.

[8]Teoremas de Andreka y Nemeti, 3.3.8, y de Sebelik y Stepanek, 3.4.1.

Con el *Capítulo IV* entramos ya en un campo de investigación en el que aún quedan muchos problemas por resolver: La información negativa. Para presentar los resultados existentes introducimos un nuevo concepto: el programa normal, así como algunos de los resultados parciales que se conocen sobre completitud y adecuación de uno de los métodos de resolución que presentamos. Como muestra del estado de efervescencia en que se encuentra hoy día la investigación en este área, bajo el epígrafe *"Resultados recientes"* (cf. 4.8) incorporamos a esta memoria el último trabajo publicado por Apt y Doets [5] en el que dan definiciones alternativas de los conceptos clásicos buscando mayor rigor y fluidez en las demostraciones.

La memoria termina con dos apéndices, el *Apéndice A*, que recopila algunos resultados elementales de teoría de conjuntos y nos presenta las herramientas básicas de la lógica de primer orden sobre la que nos vamos a basar para construir la Programación Lógica. Dentro de esta presentación merece trato especial el tema de las sustituciones, intentando fundamentar con rigor a un tema crucial en la Programación Lógica: La Unificación, a la que dedicamos el *Apéndice B*. Este capítulo trata uno de los temas más controvertidos dentro de la Programación Lógica. Podemos encontrar casi una decena de definiciones alternativas de unificador y unificador de máxima generalidad en trabajos publicados en los últimos años. La presentación que hacemos se basa en los trabajos de T. Nipkow de 1992, aunque también tomamos datos de trabajos de Lloyd [37], Apt [3], Lasser, Maher y Marriot [34] entre otros. El capítulo termina con la presentación del Algoritmo de Unificación de Herbrand y algún comentario sobre otros algoritmos, sin que sea la intención de esta memoria el estudio de los mismos.

Contenido

1 Modelos de Herbrand. Operador consecuencia inmediata 18

1.1 Programas definidos 18

1.2 Modelos de Herbrand 20

1.2.1 El teorema de Herbrand 23

1.2.2 El teorema de Skolem 25

1.2.3 Interpretación de M_P 27

1.3 Operador consecuencia inmediata 28

1.4 Clausura ordinal de $T_P \downarrow$ 33

1.4.1 Asimetría de $T_P \downarrow$ 33

2 SLD-resolución 35

2.1 Respuestas 35

2.2 SLD-resolución 36

2.3 Adecuación de la SLD-resolución 39

2.4 Completitud de la SLD-resolución 42

2.5 Reglas de computación 46

2.6 Procedimientos de SLD-Refutación 51

3 Funciones recursivas 55

3.1 Interpretaciones declarativa y procedural 55

3.2 Funciones recursivas 56

3.3 Teorema de Andreka y Nemeti 57

3.4 Teorema de Sebelik y Stepanek 60

4 Información Negativa **64**
4.1 Introducción 64
4.2 Fallo finito 65
4.3 Programas normales 70
4.4 La teoría de igualdad 71
4.5 SLDNF-resolución 75
4.6 Adecuación y completitud de la SLDNF-resolución 80
4.7 La regla de Herbrand 85
4.8 Resultados recientes 87
4.8.1 Una nueva definición 88
4.8.2 Comparación con la definición de Lloyd 93

A Preliminares **95**
A.1 Nociones básicas de Teoría de Conjuntos 95
A.2 Sintaxis de la lógica de primer orden 96
A.2.1 Sustituciones 100
A.3 Semántica 102
A.3.1 Consistencia y validez 103
A.4 Formas prenexas 105

B Unificación **107**
B.1 La relación subsunción 107
B.2 Unificación 111
B.3 Algoritmos de unificación 113
B.3.1 Algoritmo de Herbrand 114

Capítulo 1

Modelos de Herbrand. Operador consecuencia inmediata

Como hemos visto en la introducción, nuestro trabajo consiste en estudiar la consistencia de un conjunto de cláusulas. En una primra aproximación vamos a estudiar el caso históricamente más importante: el caso en que las cláusulas que representan las hipótesis de nuestro problema tengan un y sólo un literal positivo, esto es, sean cláusulas de programa definido. En este primer capítulo reduciremos el problema de encontrar modelos al problema de encontrar un tipo determinado de modelos, los *modelos de Herbrand*. Como veremos, podemos identificar un modelo de Herbrand con un determinado conjunto, con lo que podemos utilizar toda la potencia de la Teoría de Conjuntos para probar nuestros resultados. En la última parte del capítulo definimos el *operador consecuencia inmediata*, que es continuo para programas definidos, con el que podremos caracterizar fácilmente los modelos de Herbrand del programa.

1.1 Programas definidos

1.1.1 Definición. Una *cláusula* es una fórmula de la forma

$$\forall x_1 \ldots \forall x_s (L_1 \vee \ldots \vee L_m)$$

donde cada L_i, $i \in \{1, \ldots, m\}$, es un literal y $x_1, \ldots, x_s$ son todas las variables que ocurren en $L_1 \vee \ldots \vee L_m$.

1.1.2 Nota. Otros autores prefieren definir las cláusulas como conjuntos de literales $C = \{L_1, \ldots, L_m\}$ y luego definir la fórmula correspondiente a la cláusula C como:

$$Form(C) = \forall x_1 \dots \forall x_s(L_1 \vee \dots \vee L_m)$$

1.1.3 Nota. En lo que sigue usaremos las cláusulas constantemente, por lo que adoptaremos el siguiente acuerdo de notación. Denotaremos la cláusula

$$\forall x_1, \dots \forall x_s(A_1 \vee \dots \vee A_k \vee \neg B_1 \vee \dots \vee \neg B_n)$$

donde $A_1, \dots, A_k, B_1, \dots, B_n$ son átomos y $x_1, \dots, x_s$ son todas las variables que ocurren en esos átomos como

$$A_1, \dots, A_k \leftarrow B_1, \dots, B_n$$

1.1.4 Definición. Una *cláusula de programa definido* es un cláusula de la forma

$$A \leftarrow B_1, \dots, B_n$$

esto es, que contiene sólo un literal positivo. A este literal positivo, A, lo llamamos *cabeza de la cláusula.* Al resto de los literales, $B_1, \dots, B_n$, los llamamos *cuerpo de la cláusula.*

Podemos dar dos interpretaciones diferentes a las cláusulas de programa definido[1] relacionadas con nuestro propósito. La primera sería una interpretación *declarativa*, esto es, podemos interpretar la cláusula $A \leftarrow B_1, \dots, B_n$ del siguiente modo: *Para cualquier interpretación* **M** *si* $B_1, \dots, B_n$ *son válidos en* **M** *entonces* A *es válido en* **M**. También podemos darle una interpretación *procedural*: Descomponemos el problema A en problemas más simples $B_1, \dots, B_n$, cuando tengamos resueltos $B_1, \dots, B_n$ podremos resolver A.

1.1.5 Definición. Una *cláusula unidad* es una cláusula con un único literal. Si el literal es positivo tendremos una *cláusula unidad positiva*

$$A \leftarrow$$

y si el literal es negativo una *cláusula unidad negativa*

$$\leftarrow B$$

1.1.6 Definición. Sea **L** un lenguaje de primer orden. Un *programa definido* es un conjunto finito de cláusulas de programa definido del lenguaje.

[1] Y en general a todas las cláusulas

1.1.7 Nota. En lo que sigue no haremos referencia de forma explícita al lenguaje en que está escrito un programa P. Si no indicamos lo contrario supondremos que las únicas constantes y símbolos de función y predicado del lenguaje son los que ocurren en las cláusulas del programa. En caso de ambigüedad, denotaremos este lenguaje como $\mathbf{L}_P$.

1.1.8 Definición. En un programa definido, el conjunto de cláusulas que tienen en la cabeza el símbolo de predicado p se denomina *definición* de p.

1.1.9 Definición. Un *objetivo definido* es una cláusula donde todos los literales son negativos.

$$\leftarrow B_1, \ldots, B_n$$

Cada uno de esos literales negativos, B_i, los llamamos *subobjetivos* del objetivo.

1.1.10 Definición. La *cláusula vacía* denotada por $\Box$ es la cláusula que no tiene literales.

Puesto que una cláusula es el cierre universal de una disyunción de literales, diremos que una interpretación $\mathbf{M}$ satisface una cláusula si y sólo si toda asignación satisface al menos uno de los literales. Esta condición no se puede verificar en la cláusula vacía, luego $\Box$ es insatisfacible.

1.1.11 Definición. Una *cláusula de Horn* es una cláusula con un literal positivo como máximo, esto es, una cláusula de programa definido o un objetivo definido.

1.2 Modelos de Herbrand

Nuestro objetivo es el siguiente: Dado un conjunto de fórmulas S y una fórmula F, saber si F es consecuencia lógica de S o no. Según vimos en el Teorema A.3.14 este problema es equivalente a saber si $S \cup \{F\}$ es insatisfacible. Este problema así planteado parece insalvable, puesto que para demostrar que $S \cup \{F\}$ es insatisfacible tenemos que probar que ninguna interpretación $\mathbf{M}$ de $\mathbf{L}$ es modelo de $S \cup \{F\}$. Herbrand [24] resolvió el problema para cláusulas en 1930.

1.2.1 Definición. Sea $\mathbf{L}$ un lenguaje de primer orden.

1. El *universo de Herbrand* de $\mathbf{L}$, $U_{\mathbf{L}}$, es el conjunto de todos los términos cerrados de $\mathbf{L}$[2].

[2] Si $\mathbf{L}_P$ no tuviera constantes añadimos una nueva constante a al alfabeto para formar el Universo de Herbrand

2. La *base de Herbrand* de **L**, $B_{\mathbf{L}}$, es el conjunto de todos los átomos cerrados de **L**.

1.2.2 Ejemplo. Sea P el programa definido

$$\begin{array}{c} p(a) \leftarrow q(f(x)) \\ r(g(x)) \leftarrow q(b) \end{array}$$

En este caso el universo de Herbrand es $U_{\mathbf{L}} = \{a,\ b,\ f(a),\ f(b),\ g(a),\ g(b),\ f(f(a)),\ f(f(b)),\ f(g(a)),\ f(g(b)),\ g(f(a)),\ g(f(b)),\ g(g(a)),\ g(g(b)),\ldots\}$ y la base de Herbrand $B_{\mathbf{L}} = \{p(t) : t \in U_{\mathbf{L}}\} \cup \{q(t) : t \in U_{\mathbf{L}}\} \cup \{r(t) : t \in U_{\mathbf{L}}\}$

1.2.3 Definición. Sea **L** un lenguaje de primer orden. Diremos que el par **M**$= (D, I)$ es una *preinterpretación de Herbrand* del lenguaje **L** si:

1. $D = U_{\mathbf{L}}$, esto es, el universo de la preinterpretación es el universo de Herbrand del lenguaje.
2. Para toda constante c de **L**,
$$I(c) = c$$
3. Para todo símbolo de función n-ario f, $I(f)$ es la aplicación $I(f) : {U_{\mathbf{L}}}^n \to U_{\mathbf{L}}$ tal que
$$I(f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$$

Una *interpretación de Herbrand* es una interpretación basada en una preinterpretación de Herbrand.

1.2.4 Definición. Sea **L** un lenguaje de primer orden y sea **I** el conjunto de todas las interpretaciones de Herbrand de **L**. Se define la aplicación Ψ del conjunto **I** en el conjunto de las partes de la base de Herbrand del siguiente modo:

$$\begin{array}{rcll} \Psi & : & \mathbf{I} & \to \mathcal{P}(B_{\mathbf{L}}) \\ & & \mathbf{M} & \mapsto \Psi(\mathbf{M}) = \{A \in B_{\mathbf{L}} : \mathbf{M} \models A\} \end{array}$$

1.2.5 Proposición. *En las condiciones de la definición anterior, la aplicación Ψ así definida establece una biyección entre el conjunto* **I** *de las interpretaciones de Herbrand de* **L** *y el conjunto de las partes de la base de Herbrand* $\mathcal{P}(B_{\mathbf{L}})$.

Demostración:

Inyectividad.

Sean **M**$= (D, I)$ y $\mathbf{M}' = (D, I')$ dos interpretaciones de Herbrand tales que

$\Psi(\mathbf{M}) = \Psi(\mathbf{M}')$. Puesto que ambas interpretaciones se basan en la misma preinterpretación, para ver que **M=M'** basta ver que $\forall p \in SP\ I(p) = I'(p)$.

$$\begin{array}{ll} & \Psi(\mathbf{M}) = \Psi(\mathbf{M}') \\ \Longrightarrow & \{p(t_1,\ldots,t_n) \in B_{\mathbf{L}} : \mathbf{M} \models p(t_1,\ldots,t_n)\} = \\ & \{p(t_1,\ldots,t_n) \in B_{\mathbf{L}} : \mathbf{M}' \models p(t_1,\ldots,t_n)\} \\ \Longrightarrow & \forall p \in SP\ \forall (t_1,\ldots,t_n) \in U^n_{\mathbf{L}}\ (t_1,\ldots,t_n) \in I(p) \leftrightarrow (t_1,\ldots,t_n) \in I'(p) \\ \Longrightarrow & \forall p \in SP\ I(p) = I'(p). \end{array}$$

Sobreyectividad
Sea $X \subseteq B_{\mathbf{L}}$ y sea **M** la interpretación de Herbrand tal que $\forall A \in B_{\mathbf{L}}\ \mathbf{M} \models A \Longleftrightarrow A \in X$. Obviamente $\Psi(\mathbf{M}) = \{A \in B_{\mathbf{L}} : \mathbf{M} \models A\} = X$. □

1.2.6 Nota.

1. En virtud de la proposición anterior podemos identificar la interpretación de Herbrand **M** con el subconjunto $\Psi(\mathbf{M}) = \{A \in B_{\mathbf{L}} : \mathbf{M} \models A\}$ de $B_{\mathbf{L}}$.[3]
2. En lo que sigue identificaremos **M** con $\Psi(\mathbf{M})$ y usaremos la notación $I, I_1, I_2 \ldots$ para las interpretaciones de Herbrand.

1.2.7 Proposición. *Sea P un programa definido y $\{M_i\}_{i\in I}$ un conjunto no vacío de modelos de Herbrand de P. Entonces $\cap_{i\in I} M_i$ es un modelo de Herbrand de P.*

Demostración:
Obviamente $\cap_{i\in I} M_i$ es una interpretación de Herbrand de P. Veamos que es modelo de todas sus cláusulas. Sea C la cláusula

$$C = \forall x_1 \ldots x_n (p(x_1,\ldots,x_n) \vee \neg q_1(x_1,\ldots,x_n) \vee \ldots \vee \neg q_m(x_1,\ldots,x_n))$$

$\cap_{i\in I} M_i \models C \Longleftrightarrow$ Para toda n-upla $(a_1,\ldots,a_n) = \vec{a} \in U^n_{\mathbf{L}}$ se tiene

- O bien $p(\vec{a}) \in \cap_{i\in I} M_i$
- O bien $\exists i \in \{1,\ldots,m\}$ tal que $q_i(\vec{a}) \notin \cap_{i\in I} M_i$

Sea $\vec{a} \in U^n_{\mathbf{L}}$ y supongamos $p(\vec{a}) \notin \cap_{i\in I} M_i$. Entonces

$$\begin{array}{ll} \exists j \in I \text{ tal que } p(\vec{a}) \notin M_j & \Longrightarrow \\ \exists j \in I\ \exists i \in \{1,\ldots,m\} \text{ tal que } q_i(\vec{a}) \notin M_j & \Longrightarrow \\ \exists j \in I\ \exists i \in \{1,\ldots,m\} \text{ tal que } q_i(\vec{a}) \notin \cap_{i\in I} M_i & \end{array}$$

Por tanto $\cap_{i\in I} M_i$ es modelo de P. □

[3] De hecho hay autores que definen las interpretaciones de Herbrand como subconjuntos de B_P.

1.2.8 Lema. *Sea P un programa definido. Entonces B_P es un modelo de Herbrand de P.*

Demostración:
Trivial. Para toda cláusula C de P

$$C = \forall x_1 \ldots x_n(p(x_1,\ldots,x_n) \vee [\bigvee_{j=1}^{m} \neg q_j(x_1,\ldots,x_n)])$$

$B_P \models C$ puesto que para toda n-upla $(a_1,\ldots,a_n) \in U^n_{\mathbf{L}}$, $p(a_1,\ldots,a_n) \in B_P$.
$\square$

1.2.9 Definición. En virtud del lema anterior, dado un programa definido P el conjunto de los modelos de Herbrand de P es no vacío, y por la proposición 1.2.7 la intersección de todos los modelos de Herbrand de P es otro modelo de Herbrand, que llamaremos el *menor modelo de Herbrand* de P y denotaremos como M_P, i.e.,

$$M_P = \bigcap\{M : M \text{ es modelo de Herbrand de } P\}$$

1.2.10 Nota. Es importante destacar que la intersección de modelos de Herbrand es modelo de Herbrand sólo si estamos hablando conjuntos de cláusulas de Horn. Para verlo baste tomar el conjunto $S = \{p(a) \vee p(b)\}$ formado por una sola cláusula que no es cláusula de Horn. En este caso $I_1 = \{p(a)\}$ e $I_2 = \{p(b)\}$ son modelos de Herbrand de S, pero $I_1 \cap I_2 = \emptyset$ no lo es.

1.2.1 El teorema de Herbrand

1.2.11 Definición.

1. La sustitución θ es *básica* si $Rang(\theta) \subseteq U_{\mathbf{L}}$.

2. La cláusula C' es una *instancia básica* de la cláusula C si existe una sustitución básica θ tal que $C' = \theta(C)$.

3. En general diremos que una cláusula es *básica* si los literales que ocurren en ella son cerrados.

1.2.12 Teorema. *Sea S un conjunto de cláusulas y supongamos que S tiene un modelo. Entonces S tiene un modelo de Herbrand.*

Demostración:
Supongamos que **M** es modelo de S. Sea **M'** la interpretación de Herbrand dada por

$$\mathbf{M}' = \{A \in B_{\mathbf{L}} : \mathbf{M} \models A\}$$

Veamos que es modelo de todas las cláusulas de S. Sea C una de estas cláusulas:

$$C = \forall x_1 \ldots x_n([\bigvee_{i=1}^{n} p_i(x_1, \ldots, x_n] \vee [\bigvee_{j=1}^{m} \neg q_j(x_1, \ldots, x_n)])$$

M es modelo de C si y sólo si $\forall (a_1, \ldots, a_n) \in U_{\mathbf{L}}^n$

$$\mathbf{M} \models [\bigvee_{i=1}^{n} p_i(a_1, \ldots, a_n)] \vee [\bigvee_{j=1}^{m} \neg q_j(a_1, \ldots, a_n]$$

Sea $(a_1, \ldots, a_n) \in U_{\mathbf{L}}^n$. Entonces se tiene al menos una de estas opciones:

- $\exists i \in \{1, \ldots, n\}$ tal que $\mathbf{M} \models p_i(a_1, \ldots, a_n) \Longrightarrow p_i(a_1, \ldots, a_n) \in \mathbf{M}'$
- $\exists j \in \{1, \ldots, m\}$ tal que $\mathbf{M} \models \neg q_j(a_1, \ldots, a_n) \Longrightarrow q_j(a_1, \ldots, a_n) \notin \mathbf{M}'$

Por tanto de una forma u otra

$$\mathbf{M}' \models ([\bigvee_{i=1}^{n} p_i(a_1, \ldots, a_n] \vee [\bigvee_{j=1}^{n} \neg q_j(a_1, \ldots, a_n)])$$

Y esto concluye la demostración. □

1.2.13 Teorema. *Sea S un conjunto de cláusulas. Entonces S es insatisfacible si y sólo si no tiene modelos de Herbrand.*

Demostración:
($\Longrightarrow$) Trivial. Si S es insatisfacible, no tiene modelos. En particular no tiene modelos de Herbrand.
($\Longleftarrow$) Por el teorema anterior, si S no tiene modelos de Herbrand, no tiene ningún modelo. □

Corolario. 1.2.14 (Löwenheim-Skolem) *Si Γ un conjunto consistente de cláusulas, entonces tiene un modelo numerable.*

Demostración:
Si tiene trivialmente a partir del teorema 1.2.12, puesto que dado un lenguaje **L**, $U_{\mathbf{L}}$ siempre es numerable. □

Es importante destacar que las tesis de los teoremas anteriores sólo son válidas si S es un conjunto de cláusulas y no un conjunto de fórmulas cualesquiera.

1.2.15 Ejemplo. Sea $S = \{p(a), \exists x \neg p(x)\}$. Nótese que $\exists x \neg p(x)$ no es una cláusula. Evidentemente, S es satisfacible, basta tomar la interpretación $\mathbf{M}= (D, I)$ con $D = \{0, 1\}$ e $I(a) = 0$ e $I(p) = \{0\}$. Obviamente $\mathbf{M}$ es modelo de S. Pero S no tiene modelos de Herbrand. Las únicas interpretaciones de Herbrand de S son $I_1 = \emptyset$ e $I_2 = \{p(a)\}$, pero ninguna de las dos es modelo de S

Para intentar evitar situaciones como esta Skolem propone la incorporación al lenguaje de unos símbolos de función con los cuales podemos poner el cierre universal de cualquier fórmula en forma prenexa sin cuantificadores universales.

1.2.2 El teorema de Skolem

1.2.16 Definición. Para cada fórmula F definimos su *forma de Skolem*, *Skolem(F)*, como el resultado del siguiente "algoritmo" recursivo:

- Sea F_1 una fórmula en forma normal equivalente a $\forall(F)$.
- Si F_1 no tiene cuantificadores existenciales, entonces $Skolem(F)$ es F_1.
- Si F_1 es de la forma $\forall x_1 \ldots \forall x_n \exists y G$ entonces

  $$Skolem(F) = Skolem(\forall x_1 \ldots \forall x_n G[y/f(x_1, \ldots, x_n])$$

  donde f es un símbolo de aridad n que no ocurre en F. En este caso se dice que f es una *función de Skolem.*

1.2.17 Ejemplo.

Si $F_1 \equiv \exists x \neg p(x)$	entonces $Skolem(F_1) \equiv \neg p(c)$
Si $F_2 \equiv \forall x \exists y p(x, y)$	entonces $Skolem(F) \equiv \forall x p(x, f(x))$
Si $F_3 \equiv \exists x \forall y \exists z (p(x, y) \vee q(z))$	entonces $Skolem(F_3) \equiv \forall y (p(c, y) \vee q(f(y)))$

1.2.18 Nota.

1. Las formas de Skolem son formas prenexas cuyo prefijo no contiene cuantificadores existenciales.
2. A veces, se suprime el prefijo de las formas de Skolem (sobrentendiéndose que todas las variables están universalmente cuantificadas).

Teorema. 1.2.19 (Teorema de Skolem) *Una fórmula F es consistente si y sólo si su forma de Skolem $Skolem(F)$ es consistente.*

Demostración:
Vamos a establecer el resultado para una fórmula del tipo $\forall x \exists y p(x,y)$. La demostración del caso general se basa en los mismos razonamientos.
($\Longrightarrow$) Sean $\phi \equiv \forall x \exists y p(x,y)$ y $Skolem(\phi) \equiv \forall x p(x, f(x))$ donde f es un símbolo de función n-ario que no pertenece a **L**.
Sea **M**$= (D, I)$ un modelo de ϕ, por tanto para toda $a \in D$, existe $b \in D$ tal que $(a,b) \in I(p)$. Sea **L'**=**L**$\{f\}$ y sea **M'**$= (D', I')$ la interpretación de **L'** en la que

- $D = D'$
- I' se define como I con la ampliación

$$\begin{array}{rcccl} I(f) & : & D & \to & D \\ & & a & \mapsto & I(f(a)) \end{array}$$

  donde $I(f(a)) \in D$ es alguno de los elementos $b \in D$ tal que $(a,b) \in I(p)$[4].

Se tiene entonces que $\mathbf{M}' \models \forall x p(x, f(x))$, puesto que para toda $a \in D$, $(a, f(a)) \in I(p)$.
($\Longleftarrow$) Supongamos ahora que $\mathbf{M}' \models \forall x p(x, f(x))$ donde **M'** es la interpretación que acabamos de construir. Resulta evidente entonces que $\mathbf{M} \models \forall x \exists y p(x,y)$
$\square$

1.2.20 Nota.

1. Hemos probado un poco más de lo que afirma la tesis del teorema. Hemos visto que si F es una fórmula y $\mathbf{M} \models F$ podemos encontrar otra interpretación **M'** *con el mismo dominio* tal que $\mathbf{M}' \models Skolem(F)$.
2. No obstante, F y $Skolem(F)$ no son lógicamente equivalentes, esto es, $Skolem(F) \models F$, pero no es cierto que $F \models Skolem(F)$. Lo podemos ver en el siguiente ejemplo.

1.2.21 Ejemplo. Sean $F \equiv \exists x p(x)$ y $Skolem(F) \equiv p(a)$. Sea $\mathbf{M} = (D, I)$ la siguiente interpretación:

- $D = \{0, 1\}$
- $\left\{ \begin{array}{rcl} I(a) & = & 0 \\ I(p) & = & \{0\} \end{array} \right.$

[4] Para tomar ese elemento b tenemos que hacer uso del axioma de elección.

Obviamente **M** es modelo de $Skolem(F)$, pero no lo es de F.

1.2.22 Corolario. *Sea* $\Gamma = \{F_1, \ldots, F_n\}$ *un conjunto de fórmulas y* $\Gamma' = \{Skolem(F_i) : 1 \leq i \leq n\}$ *el conjunto de sus formas de Skolem, obtenidas usando para cada una distintas funciones de Skolem. Entonces* Γ *es consistente si y solo si* Γ' *es consistente.*[5]

Demostración:
Trivial a partir del teorema anterior. □

1.2.3 Interpretación de M_P

En general, cuando un programador escribe una fórmula o un conjunto de fórmulas tiene en mente una interpretación determinada, que llamamos *interpretación intencionada.* Obviamente ese conjunto de fórmulas admitirá una gran cantidad de interpretaciones posibles y no podemos tener seguridad de cual es la interpretación intencionada. Esta puede ser M_P o no, pero la siguiente proposición nos induce a pensar que M_P es la interpretación natural de un programa P. El siguiente resultado es debido a van Emden y Kowalski [50].

1.2.23 Teorema. *Sea* P *un programa definido. Entonces se tiene que*

$$M_P = \{A \in B : A \textit{ es consecuencia lógica de } P\}$$

Demostración:
A es consecuencia lógica de P
$\Longleftrightarrow$ $P \cup \{\neg A\}$ es insatisfacible
$\Longleftrightarrow$ $P \cup \{\neg A\}$ no tiene modelos de Herbrand
$\Longleftrightarrow$ Para todo modelo de Herbrand **M** de P, **M** no es modelo de $\neg A$
$\Longleftrightarrow$ Para todo modelo de Herbrand **M** de P, $A \in$ **M**
$\Longleftrightarrow$ $A \in M_P$ □

[5] Una vez estudiada la *skolemización* de una fórmula podemos presentar el problema general de la demostración automática: Dado un conjunto de fórmulas S, saber si la fórmula F es consecuencia lógica o no de S. Este problema es equivalente a saber si $\Gamma = S \cup \{F\}$ es inconsstente, esto es, si tiene o no modelos. Pero según acabamos de ver, Γ es consistente si el conjunto formado por las formas de Skolem de sus fórmulas, que llamaremos $Skolem(\Gamma)$ es consistente. Pero a su vez el conjunto $Skolem(\Gamma)$ no es más que un conjunto de cláusulas, por tanto hemos reducido nuestro problema a saber si un conjunto de cláusulas es consistente o no.

1.3 Operador consecuencia inmediata

Esta sección presenta un enfoque distinto dentro de la Programación Lógica, relacionando estrechamente la sintaxis y la semántica, las interpretaciones procedural y declarativa. La idea original se debe a van Emden y Kowalski [50].

1.3.1 Definición. Sea L un retículo completo y $T : L \to L$ una aplicación. Se dice que T es *monótona* si $T(x) \leq T(y)$ cuando $x \leq y$.

1.3.2 Definición. Sea L un retículo completo y $X \subseteq L$. Se dice que X es *dirigido* si cada subconjunto finito de X tiene un cota superior en X.

1.3.3 Definición. Sea L un retículo completo y $T : L \to L$ una aplicación. Decimos que T es *continua* si $T(sup(X)) = sup(T(X))$, para todo subconjunto dirigido X de L.

1.3.4 Definición. Sea L un retículo completo y sea $T : L \to L$ una aplicación. Diremos que a es un *punto fijo* de T si $T(a) = a$. Se dirá que $a \in L$ es *el menor punto fijo* de T, $mpf(T)$, si para todo punto fijo b de T se tiene que $a \leq b$. De forma análoga se define *el mayor punto fijo*, $Mpf(T)$.

El siguiente resultado es una versión débil de un teorema debido a Tarski [49], que generaliza un resultado anterior de Knaster y Tarski.

1.3.5 Proposición. *Sea L un retículo completo y $T : L \to L$ una aplicación monótona. Entonces T tiene un menor punto fijo, $mpf(T)$, y un mayor punto fijo, $Mpf(T)$. Más aún, se tiene que*

$$mpf(T) = inf\{x : T(x) = x\} = inf\{x : T(x) \leq x\}$$

$$Mpf(T) = sup\{x : T(x) = x\} = sup\{x : x \leq T(x)\}$$

Demostración:
Sea $G = \{x : T(x) \leq x\}$ y $g = inf(G)$. Veamos que $g \in G$.

$$\begin{array}{lll} & x \in G & \\ \Longrightarrow & g \leq x & \text{Por ser } g \text{ el ínfimo de } G \\ \Longrightarrow & T(g) \leq T(x) & \text{Por la monotonía de } T \\ \Longrightarrow & T(g) \leq x & \text{Por la definición de } G \end{array}$$

Luego, $T(g) \leq g$, ya que $g = inf(G)$ y por tanto, $g \in G$. Veamos ahora que g es punto fijo, esto es, $T(g) = g$. Tenemos que $g \in G$, luego $T(g) \leq g$. Veamos $g \leq T(g)$.

$$T(g) \leq g \Longrightarrow T(T(g)) \leq T(g) \Longrightarrow T(g) \in G \Longrightarrow g \leq T(g)$$

Por tanto $T(g) = g$. Sea ahora $g' = inf\{x : T(x) = x\}$. Por ser g punto fijo tenemos que $g' \leq g$. Por otra parte tenemos que $\{x : T(x) = x\} \subseteq \{x : T(x) \leq x\}$, luego $g \leq g'$ y esto termina la demostración.
La demostración de las igualdades del $Mpf(T)$ es análoga. □

1.3.6 Proposición. *Sea L un retículo completo y $T : L \to L$ una aplicación monótona. Supongamos que $a \in L$ y $a \leq T(a)$. Entonces existe un punto fijo a' de T tal que $a \leq a'$. Análogamente, si $b \in L$ y $T(b) \leq b$, entonces existe un punto fijo b' de T tal que $b' \leq b$.*

Demostración:
Se tiene por la proposición anterior.

$$
\begin{aligned}
a \leq T(a) &\implies a \in \{x : x \leq T(x)\} \\
&\implies a \leq sup\{x : x \leq T(x)\} = Mpf(T) = a' \\
T(b) \leq b &\implies b \in \{x : T(x) \leq x\} \\
&\implies b \geq inf\{x : T(x) \leq x\} = mpf(T) = b'
\end{aligned}
$$

□

En los siguientes resultados utilizamos aritmética transfinita. Las definiciones y propiedades que se necesitan pueden encontrarse en [27].

1.3.7 Definición. Sea L un retículo completo y sea $T : L \to L$ una aplicación monótona. Definimos:

1. $T \uparrow 0 = \bot$
2. $T \uparrow (\alpha + 1) = T(T \uparrow \alpha)$
3. $T \uparrow \alpha = sup\{T \uparrow \beta : \beta < \alpha\}$, si α es un ordinal límite
4. $T \downarrow 0 = \top$
5. $T \downarrow (\alpha + 1) = T(T \downarrow \alpha)$, si α es un ordinal sucesor
6. $T \downarrow \alpha = inf\{T \downarrow \beta : \beta < \alpha\}$, si α es un ordinal límite

1.3.8 Proposición. *Sea L un retículo completo y $T : L \to L$ una aplicación monótona. Entonces, para todo ordinal α, $T \uparrow \alpha \leq mpf(T)$ y $T \downarrow \alpha \geq Mpf(T)$. Más aún, existen ordinales β_1 y β_2 tales que si $\gamma_1 \geq \beta_1$ entonces $T \uparrow \gamma_1 = mpf(T)$ y si $\gamma_2 \geq \beta_2$ entonces $T \downarrow \gamma_2 = Mpf(T)$.*

Demostración:
Damos aquí una demostración para el $mpf(T)$. La demostración para el $Mpf(T)$ es análoga. Seguiremos los siguientes pasos:
PASO 1: *Para todo ordinal α, $T \uparrow \alpha \leq mpf(T)$.*

- $T \uparrow 0 = \bot = inf(L) \leq mpf(T)$.
- *Si* α *es sucesor* $T \uparrow \alpha = T(T \uparrow (\alpha - 1)) \leq T(mpf(T)) = mpf(T)$ Por hipótesis de inducción y ser T monótona.
- *Si* α *es límite* $T \uparrow \alpha = sup\{T \uparrow \beta : \beta < \alpha\} \leq mpf(T)$ Por hipótesis de inducción. (Ya que si $mpf(T) \subset sup\{T \uparrow \beta : \beta < \alpha\} = \cup\{T \uparrow \beta : \beta < \alpha\}$[6], existe $\beta < \alpha$ tal que $T \uparrow \beta \not\leq mpf(T)$)

PASO 2: *Para todo ordinal* α, $T \uparrow \alpha \leq T \uparrow (\alpha + 1)$

- $T \uparrow 0 = \bot = inf(L) \leq T \uparrow 1$.
- *Si* α *es sucesor* $T \uparrow \alpha = T(T \uparrow (\alpha - 1)) \leq T(T \uparrow \alpha) = T \uparrow (\alpha + 1)$ Por hipótesis de inducción y ser T monótona.
- *Si* α *es límite*

$$\begin{array}{rcll} T \uparrow \alpha & = & \cup\{T \uparrow \beta : \beta < \alpha\} & \\ & \leq & \cup\{T \uparrow (\beta + 1) : \beta < \alpha\} & \text{Por h.i.} \\ & = & \cup\{T(T \uparrow \beta) : \beta < \alpha\} & \\ & \leq & T(\cup\{T \uparrow \beta : \beta < \alpha\}) & (**) \\ & = & T(T \uparrow \alpha) & \\ & = & T \uparrow (\alpha + 1) & \end{array}$$

Veamos $(**)$.

$$\begin{array}{lll} \text{Para todo } \beta < \alpha & T \uparrow \beta \subseteq \cup\{T \uparrow \beta : \beta < \alpha\} & \Longrightarrow \\ \text{Para todo } \beta < \alpha & T(T \uparrow \beta) \subseteq T(\cup\{T \uparrow \beta : \beta < \alpha\}) & \Longrightarrow \\ \multicolumn{2}{c}{\cup\{T(T \uparrow \beta) : \beta < \alpha\} \subseteq T(\cup\{T \uparrow \beta : \beta < \alpha\})} & \end{array}$$

PASO 3: *Para cualesquiera ordinales* α *y* β, $\alpha < \beta \quad \Longrightarrow \quad T \uparrow \alpha \leq T \uparrow \beta$
Por inducción sobre β.

- *Si* α *es sucesor*
  $\alpha < \beta \quad \Longrightarrow$
  $$\left\{ \begin{array}{ll} \alpha = \beta - 1 \Longrightarrow T(\alpha) = T(\beta - 1) \leq T(\beta) & \text{Por PASO 2} \\ \alpha < \beta - 1 \Longrightarrow T(\alpha) \leq T(\beta - 1) \leq T(\beta) & \text{Por h.i. y PASO 2} \end{array} \right.$$
- *Si* α *es límite* $\alpha < \beta \Longrightarrow T \uparrow \alpha \subseteq \cup\{T \uparrow \gamma : \gamma < \beta\} = T \uparrow \beta$.

[6] Usamos el símbolo $\subset$, para la *inclusión estricta*, esto es, $A \subset B \Longleftrightarrow A \subseteq B \wedge A \neq B$

PASO 4: *Para cualesquiera ordinales α y β, si $\alpha < \beta$ y $T \uparrow \alpha = T \uparrow \beta$ entonces $T \uparrow \alpha = mpf(T)$.*
$\alpha < \beta \quad \Longrightarrow$

$$\left\{ \begin{array}{ll} \alpha + 1 = \beta \Longrightarrow T \uparrow (\alpha+1) = T \uparrow \beta & \\ \alpha + 1 < \beta \Longrightarrow T \uparrow (\alpha+1) \leq T \uparrow \beta & \text{Por PASO 3} \end{array} \right\} \Longrightarrow$$

$$\left. \begin{array}{rlcl} \Longrightarrow & T \uparrow (\alpha+1) & \leq & T \uparrow \beta \\ \text{Por PASO 2} & T \uparrow \alpha & \leq & T \uparrow (\alpha+1) \end{array} \right\} \Longrightarrow$$

$$\left. \begin{array}{rl} \Longrightarrow & T \uparrow \alpha \leq T \uparrow (\alpha+1) \leq T \uparrow \beta \\ \text{Por hipótesis} & T \uparrow \alpha = T \uparrow \beta \end{array} \right\} \Longrightarrow$$

$T \uparrow \alpha = T \uparrow (\alpha+1) = T(T \uparrow \alpha) \Longrightarrow T \uparrow \alpha$ es punto fijo $\Longrightarrow T \uparrow \alpha = mpf(T)$
Por PASO 1

PASO 5: *Existe un ordinal β tal que para cualquier otro ordinal γ, si $\gamma \geq \beta$ entonces $T \uparrow \gamma = mpf(T)$*
Sea α el menor ordinal cuyo cardinal sea mayor que el cardinal de **L**. Supongamos que $T \uparrow \delta \neq mpf(T)$ para todo ordinal $\delta < \alpha$. Definamos la aplicación siguiente

$$\begin{array}{rcccl} h & : & \alpha & \to & \mathbf{L} \\ & & \delta & \mapsto & h(\delta) = T \uparrow \delta \end{array}$$

Por PASO 4 la aplicación h es inyectiva, lo que contradice la elección de α. Luego $T \uparrow \gamma = mpf(T)$, para algún $\beta < \alpha$. El resto de lo que afirma este paso se obtiene trivialmente a partir de los PASOS 1 y 3. □

1.3.9 Proposición. *Sea L un retículo completo y sea $T : L \to L$ una aplicación continua. Entonces $mpf(T) = T \uparrow \omega$.*

Demostración:
Por la proposición anterior basta ver que $T \uparrow \omega$ es un punto fijo. Nótese que el conjunto $\{T \uparrow n : n \in \omega\}$ es dirigido, ya que T es monótona. Así

$$\begin{array}{rcl} T(T \uparrow \omega) & = & T(sup\{T \uparrow n : n \in \omega\}) \\ & = & sup\{T(T \uparrow n) : n \in \omega\} \\ & = & T \uparrow \omega \end{array}$$

□

1.3.10 Nota. Dado un programa definido P, usaremos la notación 2^{B_P} para referirnos al conjunto de todas las interpretaciones de Herbrand de P, que es un retículo completo para la inlusión de conjuntos $\subseteq$.

1.3.11 Definición. Sea P un programa definido. Llamamos *operador consecuencia inmediata* a la aplicación

$$T_P : 2^{B_P} \to 2^{B_P}$$

definida del siguiente modo. Sea I una interpretación de Herbrand de P. Entonces $T_P(I) = \{A \in B_P : A \leftarrow A_1, \ldots, A_n$ es una instancia básica de una cláusula de P y $\{A_1, \ldots, A_n\} \subseteq I\}$.

Claramente T_P es monótona.

1.3.12 Proposición. *Sea P un programa definido. Entonces la aplicación T_P es continua.*

Demostración:
Sea X un subconjunto dirigido de 2^{B_P} y veamos que $T_P(sup(X)) = sup(T_P(X))$, para lo cual necesitamos el siguiente aserto: **Aserto.** $\{A_1, \ldots, A_n\} \subseteq sup(X) \Longleftrightarrow \{A_1, \ldots, A_n\} \subseteq I$ para algún $I \in X$.

Suponiendo cierto al aserto se tiene: $A \in T_P(sup(X)) \Longleftrightarrow A \leftarrow A_1, \ldots, A_n$ es una instancia básica de una cláusula de P y $\{A_1, \ldots, A_n\} \subseteq sup(X) \Longleftrightarrow$ $A \leftarrow A_1, \ldots, A_n$ es una instancia básica de una cláusula de P y $\{A_1, \ldots, A_n\} \subseteq I$ para algún $I \in X \Longleftrightarrow A \in T_P(I)$ para algún $I \in X \Longleftrightarrow A \in sup(T_P(X))$.

Demostración del aserto
($\Longrightarrow$) Puesto que X es un conjunto dirigido todo conjunto finito tiene una cota superior en X. Tenemos
$\{A_1, \ldots, A_n\} \subseteq sup(X) = \cup_{I \in X} I$, en particular, $\forall i \in \{1, \ldots, n\}$ $\exists I_i \in X$ tal que $A_i \in I_i$. Entonces $\{A_1, \ldots, A_n\} \subseteq \cup_{i=1}^{n} I_i = sup\{I_1, \ldots I_n\}$. Puede ocurrir que el supremo no esté en X, pero sabemos que existe un $J \in X$ que es cota superior de $\{I_1, \ldots, I_n\}$,i.e., $\{A_1, \ldots, A_n\} \subseteq sup\{I_1, \ldots I_n\} \subseteq J$.

($\Longleftarrow$) Trivial. $\exists I \in X$ tal que $\{A_1, \ldots, A_n\} \subseteq I \Longrightarrow \{A_1, \ldots, A_n\} \subseteq \cup_{I \in X} I = sup(X)$. □

1.3.13 Proposición. *Sea P un programa definido e I una interpretación de Herbrand de P. Entonces se tiene que I es un modelo de P si y sólo si $T_P(I) \subseteq I$.*

Demostración:
I es modelo de $P \Longleftrightarrow$ Para toda instancia básica $A \leftarrow A_1, \ldots, A_n$ de una cláusula de P tal que $\{A_1, \ldots, A_n\} \subseteq I$ tenemos que $A \in I \Longleftrightarrow T_P(I) \subseteq I$. □

1.3.14 Teorema. *Sea* P *un programa definido. Entonces se tienen las igualdades siguientes*

$$M_P = mpf(T_P) = T_P \uparrow \omega$$

Demostración:

$$\begin{array}{rcll} M_P & = & \cap\{I : I \text{ es modelo de Herbrand de } P\} & \\ & = & \cap\{I : T_P(I) \subseteq I\} & \\ & = & inf\{I : T_P(I) \subseteq I\} & \\ & = & mpf(T_P) & \text{Por 1.3.5} \\ & = & T_P \uparrow \omega & \text{Por 1.3.9 y 1.3.12} \end{array}$$

□

1.4 Clausura ordinal de $T_P \downarrow$

1.4.1 Definición. Por el teorema 1.3.8 sabemos que si P es un programa definido existe un ordinal α tal que para todo $\beta \geq \alpha$ se tiene $T_P \downarrow \beta = Mpf(T_P)$. Llamaremos a este ordinal α la *clausura ordinal* de $T_P \downarrow$ y lo denotaremos $\|T_P \downarrow\|$.

Es fácil ver que si $T_P \downarrow$ es continua $\|T_P \downarrow\| \leq \omega$.

1.4.1 Asimetría de $T_P \downarrow$

Dado un programa definido P el operador T_P es continuo, sin embargo $T_P \downarrow$ no tiene porqué ser continuo. Lo vemos en el siguiente ejemplo.

1.4.2 Ejemplo. Consideremos el siguiente programa

$$\begin{array}{l} p(f(x)) \leftarrow p(x) \\ q(a) \leftarrow p(x) \end{array}$$

Se tiene que para $n \geq 1$ $T_P \downarrow n = \{q(a)\} \cup \{f^k(a) : k \geq n\}$. Luego $T_P \downarrow \omega = \{q(a)\}$ y por consiguiente $T_P \downarrow \omega + 1 = \emptyset$. De ahí que $\|T_P \downarrow\| = \omega + 1$ y T_P no sea continua.

Por tanto dado un programa definido, no tenemos garantizado que su clausura ordinal sea menor o igual que ω. Dedicaremos esta sección a presentar una cota para la clausura ordinal de un programa definido. Para ello necesitamos algunas definiciones:

1.4.3 Definición. Sea R un orden parcial bien fundamentado en ω en el que denotaremos como $dom(R)$ su dominio y escribiremos $a <_R b$ en lugar de $(a, b) \in R$. Asociaremos al orden R el ordinal $||R||$ definido del siguiente modo

$$||a|| = \begin{cases} 0 & \text{si } a \text{ es un elemento } <_R\text{-minimal de } dom(R) \\ sup\{||b|| + 1 : b <_R a\} & \text{en otro caso} \end{cases}$$

$$||R|| = sup\{||a|| : a \in dom(R)\}$$

Diremos que un ordinal α es *recursivo* si $\alpha = ||R||$ para algún orden parcial bien fundamentado R que sea un predicado recursivo.

1.4.4 Definición. Denotaremos como ω_1^{ck} al menor ordinal no recursivo[7].

El siguiente teorema nos caracteriza los ordinales $||T_P \downarrow ||$.

1.4.5 Teorema.

1. *Para todo ordinal $\alpha \leq \omega_1^{ck}$ existe un programa definido P tal que*

$$||T_P \downarrow || = \alpha$$

2. *Para todo programa definido P, $||T_P \downarrow || \leq \omega_1^{ck}$.*

1.4.6 Ejemplo. Para cada $n, m \in \omega$ sea P_m^n el programa definido

$$\begin{array}{rcl} P_m^n & = & \{p_j(f(x)) \leftarrow p_j(x) : j \in \{0, \ldots, m-1\}\} \\ & \cup & \{p_{j+1}(c_i) \leftarrow p_j(x) : j \in \{0, \ldots, m-1\} \ i \in \{0, \ldots, n\}\} \\ & \cup & \{p_m(c_{i+1}) \leftarrow p_m(c_i) : i \in \{0, \ldots, n-1\}\} \\ & \cup & \{p_m(c_n) \leftarrow p_m(c_n)\} \end{array}$$

Se comprueba fácilmente que el $Mpf(P_m^n) = \{p_m(c_n)\}$ y que la clausura ordinal es $||T_{P_m^n} \downarrow || = \omega m + n$.

La demostración de este teorema necesita resultados de la Teoría de la Recursión y la Teoría de Conjuntos que exceden los límites de esta memoria. Blair presenta una demostración en [8] y Apt comenta este y otros resultados en [3].

[7] ω_1^{ck} (ω_1 de Church y Kleene) recibe este nombre por ser el análogo constructivo del primer ordinal no numerable ω_1. Este ordinal es numerable y fue presentado por Church y Kleene en 1937 en [14]. Podemos encontrar distintos comentarios sobre este ordinal en [41]

Capítulo 2

SLD-resolución

En este capítulo presentamos los procedimientos básicos que se usan en Programación Lógica para la demostración de teoremas. Existen muchas variantes y refinamientos de los métodos que no entramos a describir, todas ellas basadas en el método de resolución presentado por Robinson [44] en 1965.

2.1 Respuestas

Como vimos, desde un punto de vista procedural, podemos considerar que una unificación dentro de un problema no es más que la asignación de un valor determinado a una incógnita del problema, esto es, planteada una pregunta el unificador es la *respuesta*.

2.1.1 Nota. En lo que sigue consideraremos que el objetivo siempre es una fórmula del lenguaje $\mathbf{L}_P$.

2.1.2 Definición. Sea P un programa definido y G el objetivo definido. Una *respuesta* para $P \cup \{G\}$ es una sustitución θ tal que $Dom(\theta) \subseteq V(G)$.

2.1.3 Definición. Sea P un programa definido, G un objetivo definido $\leftarrow A_1, \ldots, A_k$ y θ una respuesta para $P \cup \{G\}$. Decimos que θ es una *respuesta correcta* para $P \cup \{G\}$ si $\forall((A_1 \wedge \ldots \wedge A_k)\theta)$ es consecuencia lógica de P.

A partir del Teorema 1.2.23 y de la definición de respuesta correcta podría pensarse que dado un programa P la respuesta θ es correcta si y sólo si

$$M_P \models \forall((A_1 \wedge \ldots \wedge A_k)\theta)$$

Desafortunadamente, esto no es así en general.

2.1.4 Ejemplo. Sea P el programa

$$p(a) \leftarrow$$

Sea G el objetivo $\leftarrow p(x)$ y θ la sustitución identidad. Se tiene que $M_P = \{p(a)\}$ y $M_P \models \forall x p(x)\theta$. No obstante, θ no es una respuesta correcta, ya que $\forall x p(x)$ no es consecuencia lógica de P. La razón es que $\neg\forall x p(x)$ no es una cláusula y no podemos restringir nuestra atención a las interpretaciones de Herbrand.[1]

2.1.5 Teorema. *Sea P un programa definido y sea G un objetivo definido $\leftarrow A_1, \ldots, A_k$. Supongamos que θ es una respuesta para $P \cup \{G\}$ tal que $(A_1 \wedge \ldots \wedge A_k)\theta$ es básica. Entonces, las siguientes condiciones son equivalentes:*

1. *θ es correcta.*
2. *Para todo modelo de Herbrand M de P, $M \models (A_1 \wedge \ldots \wedge A_k)\theta$.*
3. *$M_P \models (A_1 \wedge \ldots \wedge A_k)\theta$*

Demostración:
Las demostraciones de *(1) ⇒ (2)* y *(2) ⇒ (3)* son triviales. Veamos *(3) ⇒ (1)*
$M_P \models (A_1 \wedge \ldots \wedge A_n)\theta$
$\Longrightarrow$ Para todo $\mathbf{M}$ modelo de Herbrand de P, $\mathbf{M} \models (A_1 \wedge \ldots \wedge A_n)\theta$
$\Longrightarrow$ $P \cup \{\neg(A_1 \wedge \ldots \wedge A_n)\theta\}$ no tiene modelos de Herbrand
$\Longrightarrow$ $P \cup \{\neg(A_1 \wedge \ldots \wedge A_n)\theta\}$ no tiene modelos
$\Longrightarrow$ $(A_1 \wedge \ldots \wedge A_n)\theta$ es consecuencia lógica de P
$\Longrightarrow$ θ es correcta. □

2.2 SLD-resolución

La SLD-resolución es una adaptación del método de resolución para programas definidos descrito por primera vez por Kowalski, aunque es en un artículo de Apt y van Emden [6] donde aparece por primera vez este término.[2]

[1] Sea $\mathbf{M} = (D, I)$ la interpretación de $\mathbf{L}_P$ dada por

$$\begin{cases} D = \{0,1\} \\ I(a) = 0 \\ I(p) = \{0\} \end{cases}$$

Se comprueba trivialmente que $\mathbf{M}$ es modelo de $p(a)$, pero no lo es de $\forall x p(x)$.

[2] El término *SLD-resolución*, (SLD-resolution, en inglés) es un acrónimo de "*SL-Resolution for Definite clauses*", donde a su vez "SL-Resolution" es un término abreviado de "*Linear resolution with Selection function*".

2.2.1 Definición. Sea G el objetivo $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ y C la cláusula $A \leftarrow B_1, \ldots, B_q$. Se dice que el objetivo G' es *derivado* de G y C usando el umg θ si se verifican las siguientes condiciones:

1. A_m es un átomo de G, llamado el átomo *seleccionado.*

2. θ es un umg de A_m y A.

3. G' es el objetivo $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.

Diremos entonces que G' es una *resolvente* de G y C.

2.2.2 Definición. Sea P un programa definido y G un objetivo definido. Una *SLD-derivación* de $P \cup \{G\}$ es una sucesión (finita o infinita) de objetivos $G = G_0, G_1, G_2 \ldots$, una sucesión $C_1, C_2 \ldots$ de variantes de cláusulas del programa P y una sucesión $\theta_1, \theta_2, \ldots$ de umg y tales que

1. G_{i+1} se deriva de G_i y C_{i+1} usando θ_{i+1}.

2. $V(C_{i+1}) \cap (V(G) \cup V(C_1) \cup V(C_2) \cup \ldots \cup V(C_i)) = \emptyset$, esto es, ninguna de las variables que ocurren en C_{i+1} ocurre en G, C_1, C_2, ... o C_i. Denominaremos a esta condición *separación de variables* ("Standardising apart" en inglés)[3].

Cada variante de una cláusula de programa $C_1, C_2 \ldots$ se le llama *cláusula de entrada* de la derivación. (En la página siguiente se muestra un esquema de SLD-derivación).

2.2.3 Definición. Una *SLD-refutación*[4] de $P \cup \{G\}$es una SLD-derivación finita de $P \cup \{G\}$que tiene la cláusula vacía $\Box$ como último objetivo. Si $G_n = \Box$ se dice que la refutación tiene longitud n.

A lo largo del capítulo, cuando hablemos de "derivación" y "refutación" nos referiremos siempre a SLD-derivación y SLD-refutación.

[3] De esta forma evitamos que las derivaciones dependan de las variables utilizadas. En la práctica, esta condición se cumple generalmente numerando las cláusulas y asociando a las variables el subíndice correspondiente.

[4] El principal objetivo de la Programación Lógica es la demostración de teoremas mediante refutación, pero la SLD-derivación no sólo nos permite refutar, sino también computar, ya que como vimos, el proceso de unificación no es más que la asignación de valores a las variables y esto puede ser tomado como la búsqueda de soluciones a problemas.

2.2.4 Definición. Definimos una *SLD-derivación no restringida* como una SLD-derivación, salvo que no exigimos que los θ_i sean umg, basta con que sean unificadores.

SLD-derivación

$$\begin{tikzcd}
G_0 = G \arrow[d] & C_1, \theta_1 \arrow[dl] \\
G_1 \arrow[d] & C_2, \theta_2 \arrow[dl] \\
G_2 \arrow[d] & C_3, \theta_3 \arrow[dl] \\
\vdots \arrow[d] & \\
G_{n-1} \arrow[d] & C_n, \theta_n \arrow[dl] \\
G_n \arrow[d] & \\
\vdots &
\end{tikzcd}$$

2.2.5 Definición. Una SLD-derivación puede ser *finita* o *infinita*. Una SLD-derivación finita puede *tener éxito* o ser *fallida*.

1. Una SLD-derivación *con éxito* es una SLD-derivación que termina con la cláusula vacía, esto es, una refutación.
2. Una SLD-derivación *fallida* es una SLD-derivación que termina en una cláusula no vacía con la propiedad de que el átomo seleccionado en el último objetivo no unifica con la cabeza de ninguna cláusula del programa.

2.2.6 Definición. Sea P un programa definido. El *conjunto de éxito* de P es el conjunto de todos los $A \in B_P$ tales que $P \cup \{\leftarrow A\}$ tiene una SLD-refutación.

El conjunto de éxito de P es la contrapartida procedural del menor modelo de Herbrand de P. Veremos más adelante que realmente ambos conjuntos coinciden.

2.2.7 Definición. Sea P un programa definido y G un objetivo definido. Una *respuesta computada* θ para $P \cup \{G\}$ es la sustitución obtenida restringiendo la composición $\theta_1 \dots \theta_n$ a las variables de G, donde $\theta_1, \dots, \theta_n$ es la sucesión de umg usada en una SLD-refutación de $P \cup \{G\}$.

2.3 Adecuación de la SLD-resolución

La siguiente versión del teorema de adecuación de la SLD-resolución se debe a Clark [11].

Teorema. 2.3.1 (Adecuación de la SLD-resolución) *Sea P un programa definido y G un objetivo definido. Entonces cada respuesta computada para $P \cup \{G\}$ es una respuesta correcta para $P \cup \{G\}$.*

Demostración:
Sea G el objetivo $\leftarrow A_1, \dots, A_k$ y sea $\theta_1, \dots, \theta_n$ la sucesión de umg usada en la refutación de $P \cup \{G\}$. Veamos que $\forall((A_1 \wedge \dots \wedge A_k)\theta_1 \dots \theta_n)$ es consecuencia lógica de P. Lo probamos por inducción en la longitud de la refutación.
$(n = 1)$ Si la refutación es de longitud 1 es porque G es un objetivo de la forma $\leftarrow A_1$, el programa tiene una cláusula unidad $A \leftarrow$ y existe un umg θ_1 tal que $A\theta_1 = A_1\theta_1$. Por tanto, puesto que $A\theta_1$ es una instancia de una cláusula de P, tenemos que $\forall(A_1\theta_1)$ es consecuencia lógica de P.
$(n - 1 \Rightarrow n)$ Supongamos probado el resultado para refutaciones de longitud $n - 1$ y sea $\theta_1, \dots, \theta_n$ la sucesión de umg usada en una refutación de $P \cup \{G\}$

de longitud n. Sea $A \leftarrow B_1, \ldots, B_q$ la primera cláusula de entrada y A_m el átomo seleccionado en G. Por hipótesis de inducción

$$\forall((A_1 \wedge \ldots \wedge A_{m-1} \wedge B_1 \wedge \ldots \wedge B_q \wedge A_{m+1} \wedge \ldots \wedge A_k)\theta_1 \ldots \theta_n)$$

es consecuencia lógica de P. Luego si $q > 0$ $\forall((B_1 \wedge \ldots \wedge B_q)\theta_1 \ldots \theta_n)$ es consecuencia lógica de P y por tanto $\forall(A_m\theta_1 \ldots \theta_n)$ (que es lo mismo que $\forall(A\theta_1 \ldots \theta_n)$) es consecuencia lógica de P y por tanto

$$P \models \forall((A_1 \wedge \ldots \wedge A_k)\theta_1 \ldots \theta_n)$$

como queríamos probar. □

2.3.2 Corolario. *Sea P un programa definido y G un objetivo definido. Supongamos que existe una SLD-refutación de $P \cup \{G\}$. Entonces $P \cup \{G\}$ es insatisfacible.*

Demostración:
Sea G el objetivo $\leftarrow A_1, \ldots, A_k$. Por el teorema 2.3.1 la respuesta computada θ que hemos obtenido en la SLD-refutación de $P \cup \{G\}$ es correcta, luego

$$P \models \forall((A_1 \wedge \ldots \wedge A_k)\theta)$$

Aplicando ahora el teorema A.3.14 tenemos que

$$P \cup \{\neg\forall((A_1 \wedge \ldots \wedge A_k)\theta)$$

es insatisfacible. De ahí que $P \cup \{G\}$ sea insatisfacible. □

2.3.3 Corolario. *El conjunto de éxito de un programa definido está contenido en su menor modelo de Herbrand.*

Demostración:
Sea P un programa definido y sea $A \in B_P$.

$$\begin{array}{lll} & A \text{ pertenece al conjunto de éxito de } P & \\ \iff & P \cup \{\leftarrow A\} \text{ tiene una SLD-refutación} & \\ \implies & P \models A & \text{Por 2.3.1} \\ \iff & A \in M_P & \text{Por 1.2.23} \end{array}$$

□

2.3.4 Definición. Si A es un átomo, definimos

$$[A] = \{A' \in B_P : A' = A\theta \text{ para alguna sustitución } \theta\ \}$$

Así $[A]$ es el conjunto de todas las instancias cerradas de A.

2.3.5 Teorema. *Sea P un programa definido y sea G el objetivo definido $\leftarrow A_1 \ldots A_k$. Supongamos que $P \cup \{G\}$ tiene una SLD-refutación de longitud n y $\theta_1 \ldots \theta_n$ es la sucesión de umg de la SLD-refutación. Entonces tenemos que $\cup_{j=1}{}^k[A_j\theta_1 \ldots \theta_n] \subseteq T_p \uparrow n$.*

Demostración:
Probamos este teorema por inducción sobre la longitud de la derivación.
$(n = 1)$ En este caso G es de la forma $\leftarrow A_1$ y el programa tiene una cláusula unidad $A \leftarrow$ tal que existe un umg θ de A y A_1 tal que $A_1\theta = A\theta$. Obviamente $[A] \in T_P \uparrow 1$ y por tanto

$$[A_1\theta] = [A\theta] \subseteq [A] \subseteq T_P \uparrow 1$$

$(n-1 \Rightarrow n)$ Supongamos ahora el resultado cierto para refutaciones de longitud $(n-1)$ y consideremos una refutación de $P \cup \{G\}$ de longitud n.
Tomemos entonces $A_j\theta_1$, un átomo de G_1, el segundo objetivo de la refutación. Por hipótesis de inducción

$$[(A_j\theta_1)\theta_2 \ldots \theta_n] \subseteq T_P \uparrow (n-1)$$

y por la monotonía de T_P, $T_P \uparrow (n-1) \subseteq T_P \uparrow n$.
Supongamos ahora que A_j es el átomo seleccionado en G y sea $B \leftarrow B_1, \ldots, B_q$ la primera cláusula de entrada. Tenemos que $B\theta_1 = A_j\theta_1$

- *Si* $q = 0$. $[B] \subseteq T_P \uparrow 1$ con lo que tenemos

$$[A_j\theta_1 \ldots \theta_n] \subseteq [A_j\theta_1] \subseteq [B] \subseteq T_p \uparrow 1 \subseteq T_p \uparrow n$$

- *Si* $q > 0$. Por hipótesis de inducción, para todo $i \in \{1, \ldots, q\}$

$$[B_i\theta_1 \ldots \theta_n] \subseteq T_p \uparrow (n-1)$$

  y por la definición de T_P,

$$[A_j\theta_1 \ldots \theta_n] = [B\theta_1 \ldots \theta_n] \subseteq T_p \uparrow n$$

□

2.4 Completitud de la SLD-resolución

Para la demostración de los resultados de esta sección necesitamos dos lemas técnicos que enunciamos a continuación. No damos aquí sus demostraciones aunque sí ejemplos ilustrativos. Los detalles de las demostraciones pueden encontrarse en [37].

Lema. 2.4.1 (Lema umg) *Sea P un programa definido y sea G un objetivo definido. Supongamos que $P \cup \{G\}$ tiene un SLD-refutación no restringida. Entonces $P \cup \{G\}$ tiene una SLD-refutación de la misma longitud tal que, si $\theta_1, \ldots, \theta_n$ son los unificadores de la SLD-refutación no restringida y $\theta'_1, \ldots, \theta'_n$ son los umg de la SLD-refutación, existe una sustitución γ tal que $\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n \gamma$.*

2.4.2 Ejemplo. Sea el programa P

$$\left\{ \begin{array}{l} 1.\ p(x, y) \leftarrow q(x, y) \\ 2.\ q(z, a) \leftarrow \end{array} \right.$$

Consideremos la siguiente derivación no restringida del objetivo $G \equiv \leftarrow p(v, w)$

$$\begin{array}{cl} \leftarrow p(v, w) & \\ 1 \Big\downarrow & \theta_1 = \{x/u, y/u, v/u, w/u\} \\ \leftarrow q(u, u) & \\ 2 \Big\downarrow & \theta_2 = \{z/a, u/a\} \\ \Box & \end{array}$$

Tenemos que $\theta_1\theta_2 = \{x/a, y/a, v/a, w/a, z/a, u/a\}$. Veamos ahora una refutación del mismo objetivo donde los unificadores son de máxima generalidad

$$\begin{array}{c}
\leftarrow p(v,w) \\
1 \;\Big\downarrow\; \theta'_1 = \{v/x, w/y\} \\
\leftarrow q(x,y) \\
2 \;\Big\downarrow\; \theta'_2 = \{x/z, y/a\} \\
\square
\end{array}$$

Aquí $\theta'_1\theta'_2 = \{v/z, w/a, x/z, y/a\}$. Basta tomar como sustitución γ

$$\gamma = \{z/a, u/a\}$$

para que se verifique la tesis del lema.

La demostración del caso general se realiza por inducción sobre la longitud de la refutación. Para longitud uno se aplica directamente la definición de unificador de máxima generalidad. Para longitud superior a uno se aplica esta misma definición junto con la hipótesis de inducción.

Lema. 2.4.3 (Lema de la elevación) *Sea P un programa definido,G un objetivo definido y θ una sustitución. Supongamos que existe una SLD-refutación de $P \cup \{G\theta\}$. Entonces existe una SLD-refutación de $P \cup \{G\}$ de la misma longitud tal que, si $\theta_1, \ldots, \theta_n$ son los umg de la SLD-refutación de $P \cup \{G\theta\}$ y $\theta'_1, \ldots, \theta'_n$ son los umg de la SLD-refutación de $P \cup \{G\}$, entonces existe una sustitución γ tal que $\theta\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n\gamma$.*

2.4.4 Ejemplo. Sean el programa P

$$\left\{ \begin{array}{l} 1.\ p(x,y) \leftarrow q(x,y) \\ 2.\ q(z,z) \leftarrow \end{array} \right.$$

el objetivo $G \equiv \leftarrow p(v,w)$ y la sustitución $\theta = \{v/a\}$. Consideremos la siguiente refutación de $G\theta \equiv \leftarrow p(a,w)$

$$
\begin{array}{lll}
 & \leftarrow p(a,w) & \\
1 & \Big\downarrow & \theta_1 = \{y/w, x/a\} \\
 & \leftarrow q(a,w) & \\
2 & \Big\downarrow & \theta_2 = \{z/a, w/a\} \\
 & \square &
\end{array}
$$

donde las cláusulas y unificadores utilizados en cada paso se dan de forma explícita en el diagrama. Tenemos que

$$\theta_1\theta_2 = \{x/a, y/a, z/a\}$$

Veamos que podemos encontrar una refutación de G verificando la tesis del lema

$$
\begin{array}{lll}
 & \leftarrow p(v,w) & \\
1 & \Big\downarrow & \theta'_1 = \{x/v, y/w\} \\
 & \leftarrow q(v,w) & \\
2 & \Big\downarrow & \theta'_2 = \{v/z, w/z\} \\
 & \square &
\end{array}
$$

Con $\theta'_1\theta'_2 = \{x/z, w/z\}$. Puesto que

$$\theta\theta_1\theta_2 = \{v/a, x/a, y/a, z/a\}$$

tomando $\gamma = \{z/a, v/a, y/a\}$ se tiene lo que afirma el teorema.

En el caso general la idea de la demostración es simple. Basta considerar que la composición de θ con la sucesión de unificadores $\theta_1, \ldots, \theta_n$ resultante de una refutación de $G\theta$ dan lugar a una refutación no restringida de G. Luego basta aplicar el lema 2.4.1.

2.4.5 Teorema. *El conjunto de éxito de un programa definido es igual a su menor modelo de Herbrand.*

Demostración:
Sea P un programa definido. Por el corolario 2.3.3 es suficiente probar que el menor modelo de Herbrand de P está contenido en el conjunto de éxito de P. Sea $A \in M_P$. Por el Teorema 1.3.14

$$A \in M_P = T_P \uparrow \omega = \cup\{T_P \uparrow n : n \in \omega\}$$

Por tanto $A \in T_P \uparrow n$ para algún $n \in \omega$. Probemos por inducción sobre n que si $A \in T_P \uparrow n$ entonces $P \cup \{\leftarrow A\}$ tiene una SLD-refutación y por tanto A pertenece al conjunto de éxito de P.
$(n = 1)$ Si $A \in T_P \uparrow 1$ es porque existe una cláusula unidad en P, $A_1 \leftarrow$ tal que A es una instancia cerrada de A_1. En este caso es evidente que $P \cup \{\leftarrow A\}$ tiene una refutación.
$(n - 1 \Rightarrow n)$ Supongamos el resultado cierto para $n - 1$. Sea $A \in T_P \uparrow n$. Por la definición de ${}_P$ existe una instancia básica de una cláusula de P, $B \leftarrow B_1, \ldots, B_q$ tal que $A = B\theta$ y $\{B_1\theta, \ldots, B_q\theta\} \subseteq T_P \uparrow (n-1)$ para algún θ. Por hipótesis de inducción, para todo $i \in \{1, \ldots, q\}$ $P \cup \{\leftarrow B_i\theta\}$ tiene una SLD-refutación. Puesto que para todo $i \in \{1, \ldots, q\}$ $B_i\theta$ es cerrado, estas refutaciones se pueden combinar para obtener una refutación de $P \cup \{\leftarrow (B_1, \ldots, B_q)\theta\}$, de donde $P \cup \{\leftarrow A\}$ tiene una refutación no restringida. Por último aplicamos el lema 2.4.1 y obtenemos una refutación de $P \cup \{\leftarrow A\}$. □

Teorema. 2.4.6 (Completitud de la SLD-Resolución) *Sea P un programa definido y G un objetivo definido. Supongamos que $P \cup \{G\}$ es insatisfacible. Entonces existe una SLD-refutación de $P \cup \{G\}$.*

Demostración:
Sea G el objetivo definido $\leftarrow A_1, \ldots, A_n$. Puesto que $P \cup \{G\}$ es insatisfacible $M_P \; 2 \; P \cup \{G\}$, de donde tenemos que $M_P \; 2 \; G$. Por tanto para alguna sustitución θ, $\{A_1\theta, \ldots, A_n\theta\} \subseteq M_P$. Por el teorema 2.4.5 para todo $i \in \{1, \ldots, n\}$ hay una SLD-refutación para $P \cup \{A_i\theta\}$. Pero los $A_i\theta$ son átomos cerrados, así que podemos combinar esas refutaciones para obtener una refutación de

$$P \cup \{\leftarrow (A_1, \ldots, A_n)\theta\}$$

Por último, si existe una refutación de $P \cup \{G\theta\}$, por el lema 2.4.3 existe una refutación de $P \cup \{G\}$. □

2.4.7 Lema. *Sea P un programa definido y A un átomo. Supongamos que $\forall(A)$ es consecuencia lógica de P. Entonces existe una SLD-refutación de $P \cup \{\leftarrow A\}$ con la sustitución identidad como la respuesta computada.*

Demostración:
Sea $\{x_1, \ldots, x_n\}$ el conjunto de las variables que ocurren en A. Sea $\{a_1, \ldots, a_n\}$ un conjunto de n constantes del lenguaje que no ocurren[5] en P ni en A, y sea θ la sustitución

$$\theta = \{x_1/a_1, \ldots, x_n/a_n\}$$

Obviamente $A\theta$ es consecuencia lógica de P. Puesto que $A\theta$ es cerrado, por el teorema 2.4.5 $P \cup \{\leftarrow A\theta\}$ tiene una refutación. Puesto que a_i no ocurre en A ni en P, sustituyendo a_i por x_i, $i \in \{1, \ldots, n\}$, obtendríamos una refutación de $P \cup \{\leftarrow A\}$ con la sustitución identidad como respuesta computada. □

El siguiente resultado es un inverso parcial del teorema de adecuación. Se debe a Clark [11].

2.4.8 Teorema. *Sea P un programa definido y sea G un objetivo definido. Para toda respuesta correcta θ para $P \cup \{G\}$ existe una respuesta computada σ para $P \cup \{G\}$ y una sustitución γ tal que $\theta = \sigma\gamma$.*

Demostración:
Sea G el objetivo $\leftarrow A_1, \ldots, A_k$. Puesto que θ es correcta $\forall((A_1 \wedge \ldots \wedge A_k)\theta)$ es consecuencia lógica de P y por el lema 2.4.7 para todo $i \in \{1, \ldots, k\}$ existe una refutación de $P \cup \{\leftarrow A_i\theta\}$ tal que la respuesta computada es la sustitución identidad. Podemos por tanto combinar esas refutaciones para obtener una refutación de $P \cup \{\leftarrow G\theta\}$ tal que la respuesta computada sea la identidad.
Sea $\theta_1, \ldots, \theta_n$ la sucesión de umg de la refutación de $P \cup \{\leftarrow G\theta\}$. Puesto que su composición es la sustitución identidad tenemos que $G\theta\theta_1 \ldots \theta_n = G\theta$ y por el lema de elevación 2.4.3 existe una refutación de $P \cup \{G\}$ con los umg $\theta'_1, \ldots, \theta'_n$ tal que

$$\theta\theta_1 \ldots \theta_n = \theta'_1, \ldots, \theta'_n\gamma'$$

para alguna sustitución γ'. Sea σ la restricción de $\theta'_1 \ldots \theta'_n$ a las variables de G. Entonces $\theta = \sigma\gamma$ donde γ es la restricción de γ' apropiada. □

2.5 Reglas de computación

Existen muchas formas de seleccionar los átomos de un objetivo en una resolución. Presentamos aquí un criterio clásico en el que dado un objetivo siempre seleccionamos en él el mismo átomo independientemente del punto de la derivación en que nos hallemos. Esta definición de regla de computación, aunque rígida, nos permite probar que la existencia o no de refutación no depende de la regla elegida.

[5]Si no existen tales constantes se las añadimos al lenguaje

2.5.1 Definición. Dado un lenguaje **L**, una *regla de computación* es una aplicación entre el conjunto de objetivos definidos y el conjunto de átomos, de forma que la imagen de un objetivo por esa aplicación sea un átomo de ese objetivo, llamado el átomo *seleccionado* de ese objetivo.

$$\begin{array}{rcll} R & : & \text{Objetivos Definidos} & \rightarrow \ \text{Atomos} \\ & & (\leftarrow B_1, \ldots, B_n) & \mapsto \ R(\leftarrow B_1, \ldots, B_n) = B_i \quad i \in \{1, \ldots, n\} \end{array}$$

2.5.2 Definición. Sea P un programa definido, G un objetivo definido y R una regla de computación. Una *SLD-derivación de $P \cup \{G\}$ vía R* es una SLD-derivación de $P \cup \{G\}$ en la cual usamos la regla de computación R para seleccionar átomos

Al definir de este modo las reglas de derivación estamos imponiendo una restricción muy fuerte, ya que si un objetivo aparece en diferentes momentos tenemos que seleccionar en él siempre el mismo átomo. Dicho de otro modo, existen SLD-derivaciones que no son SLD-derivaciones *vía R* para ninguna regla de computación R.
Hay otros autores que consideran esta restricción demasiado fuerte y dan otras definiciones más flexibles de lo que puede ser una regla de computación.[6]

2.5.3 Definición. Sea P un programa definido, G un objetivo definido y R una regla de computación. Una *SLD-refutación de $P \cup \{G\}$ vía R* es una SLD-refutación de $P \cup \{G\}$ en la que usamos la regla de computación R para seleccionar átomos.

2.5.4 Definición. Sea P un programa definido, G un objetivo definido y R una regla de computación. Una *respuesta R- computada* de $P \cup \{G\}$ es una respuesta computada de $P \cup \{G\}$obtenida a partir de una SLD-refutación vía R.

Nuestra intención es demostrar que si tenemos un programa definido P, un objetivo definido G y $P \cup \{G\}$ es insatisfacible entonces podemos encontrar una SLD-refutación de $P \cup \{G\}$ *independientemente de la regla de computación que usemos*, esto es, que si existe una SLD-refutación de $P \cup \{G\}$ entonces existe una SLD-refutación vía R, para cualquier regla de computación que usemos.
Para probarlo necesitamos un lema técnico previo en el que empleamos la siguiente notación:
Si C es una cláusula de un programa definido, C^+ denotará la cabeza de la cáusula y C^- denotará el cuerpo.[7]

[6] Ver Apt [3].
[7] No damos aquí la demostración del lema, que puede encontrarse en [37].

Lema. 2.5.5 (Lema del Cambio) *Sea P un programa definido y G un objetivo definido. Supongamos que $P \cup \{G\}$ tiene una SLD-refutación*

$$G_0 = G, G_1, \ldots, G_{q-1}, G_q, G_{q+1}, \ldots, G_n = \Box$$

con cláusulas de entrada $C_1, \ldots, C_n$ y umg $\theta_1, \ldots, \theta_n$. Supongamos que
G_{q-1} es $\leftarrow A_1, \ldots, A_{i-1}, A_i, \ldots, A_{j-1}, \ldots, A_k$
G_q es $\leftarrow (A_1, \ldots, A_{i-1}, C_q^{-}, \ldots, A_{j-1}, A_j, A_k)\theta_q$
G_{q+1} es $\leftarrow (A_1, \ldots, A_{i-1}, C_q^{-}, \ldots, A_{j-1}, C_{q+1}^{-}, \ldots, A_k)\theta_q\theta_{q+1}$
Entonces existe una SLD-refutación de $P \cup \{G\}$ en la cual A_j es el átomo seleccionado en G_{q-1} en lugar de A_i y A_i es el seleccionado en G_q en lugar de A_j. Más aún, si σ es la respuesta computada para $P \cup \{G\}$ en la refutación dada y σ' es la respuesta computada de $P \cup \{G\}$ en la nueva refutación, entonces $G\sigma$ es una variante de $G\sigma'$.

2.5.6 Ejemplo. Consideremos el siguiente programa P

$$\begin{cases} 1.\ p(x_1, y_1, z_1) \leftarrow q(x_1, y_1), r(y_1, z_1, x_1) \\ 2.\ q(x_2, a) \leftarrow \\ 3.\ r(y_2, z_2, b) \leftarrow \end{cases}$$

Consideremos la siguiente refutación del objetivo $\leftarrow p(x, y, z)$, donde las cláusulas de entrada son, por este orden, 1, 2 y 3. (Marcamos en negrilla el átomo seleccionado).

$$\begin{array}{c}
\leftarrow \mathbf{p(x, y, z)} \\
1 \ \Big\downarrow \ \theta_1 = \{x/x_1, y/y_1, z/z_1\} \\
\leftarrow \mathbf{q(x_1, y_1)}, r(y_1, z_1, x_1) \\
2 \ \Big\downarrow \ \theta_2 = \{x_1/x_2, y_1/a\} \\
\leftarrow \mathbf{r(a, z_1, x_2)} \\
3 \ \Big\downarrow \ \theta_3 = \{y_2/a, z_2, z_1, x_2/b\} \\
\Box
\end{array}$$

Tenemos que la respuesta computada en esta derivación es

$$\sigma = \{x/b, y/a, z/z_1\}$$

y por tanto $G\sigma \equiv \leftarrow p(b, a, z_1)$.
Veamos ahora una derivación del mismo objetivo donde se han permutado los átomos seleccionados en G_1 y G_2, esto es, las cláusulas de entrada son, por este orden, 1, 3 y 2.

$$
\begin{array}{c}
\leftarrow \mathbf{p}(\mathbf{x}, \mathbf{y}, \mathbf{z}) \\
1 \ \Big\downarrow \ \theta'_1 = \{x/x_1, y/y_1, z/z_1\} \\
\leftarrow q(x_1, y_1), \mathbf{r}(\mathbf{y_1}, \mathbf{z_1}, \mathbf{x_1}) \\
3 \ \Big\downarrow \ \theta'_2 = \{y_1, y_2, z_1/z_2, x_1/b\} \\
\leftarrow \mathbf{q}(\mathbf{b}, \mathbf{y_2}) \\
2 \ \Big\downarrow \ \theta'_3 = \{x_2/b, y_2/a\} \\
\square
\end{array}
$$

La respuesta computada en esta derivación es

$$\sigma' = \{x/b, y/a, z/z_2\}$$

y por tanto $G\sigma' \equiv \leftarrow p(b, a, z_2)$. Obviamente se tiene que $G\sigma'$ es una variante de $G\sigma$ como afirma el lema.

Teorema. 2.5.7 (Independencia de la regla de computación)
Sea P un programa definido y G un objetivo definido. Supongamos que existe una SLD-refutación de $P \cup \{G\}$ con σ como respuesta computada. Entonces, para cualquier regla de computación R, existe una SLD-refutación de $P \cup \{G\}$ vía R con una respuesta R-computada σ' tal que $G\sigma'$ es una variante de $G\sigma$.

Demostración:
Sea $G = G_0, G_1, \ldots, G_n = \square$ la sucesión de objetivos de la SLD-refutación Π de $P \cup \{G\}$ que sabemos que existe por hipótesis y sea σ la respuesta computada en esa refutación. Suponemos que para todo $i \in \{0, \ldots, n-1\}$ G_{i+1} se obtiene a partir de G_i y la cláusula de entrada C_i mediante el umg θ_i.
Sea i el menor índice para el cual la selección de átomos determinada por R y la realizada en la SLD- refutación Π varían. Sea A^i el átomo seleccionado por R en G_i. Puesto que la SLD-derivación dada Π termina en la cláusula

vacía, ese átomo (o su variante correspondiente) es seleccionado en un paso posterior de la refutación. Supongamos que es en el objetivo G_r $(r > i)$ donde seleccionamos A^i. En virtud del lema del cambio 2.5.5 podemos permutar los átomos seleccionados de G_r y G_{r-1} con lo que obtendríamos una nueva refutación en la que seleccionamos A^i en el objetivo G_{r-1}. Permutando ahora los átomos seleccionados en G_{r-1} y G_{r-2}, G_{r-2} y G_{r-3}, etc... podemos conseguir una SLD-refutación Π' en la que seleccionamos A^i en el objetivo G_i. Si σ' es la respuesta computada en esta nueva derivación Π' es evidente que $G\sigma'$ es una variante de $G\sigma$.
En el paso siguiente volvemos a tomar el menor índice j para el cual la selección de átomos realizada en Π' y la indicada por R difieren. Obviamente $j > i$ y puesto que la refutación tiene un número finito de objetivos, el proceso termina. □

2.5.8 Definición. Sea P un programa definido y R una regla de computación. El *conjunto de R-éxito* de P es el conjunto de todos los $A \in B_P$ tales que $P \cup \{\leftarrow A\}$ tiene una SLD-refutación vía R.

2.5.9 Teorema. *Sea P un programa definido y R una regla de computación. Entonces el R-conjunto de éxito de P es igual a su menor modelo de Herbrand.*

Demostración:
Se tiene directamente a partir de los teoremas 2.4.5 y 2.5.7. □

2.5.10 Teorema. *Sea P un programa definido, G un objetivo definido y R una regla de computación. Supongamos que $P \cup \{G\}$ es insatisfacible. Entonces existe un SLD-refutación de $P \cup \{G\}$ vía R.*

Demostración:
Directamente a partir de 2.4.6 y 2.5.7. □

El teorema de completitud de la SLD-Resolución puede ser generalizado en distintas direcciones. Una de ellas es la siguiente:

2.5.11 Teorema. *Sea P un programa definido , G un objetivo definido y R una regla de computación. Entonces, para toda respuesta correcta θ de $P \cup \{G\}$ existe una respuesta R-computada σ para $P \cup \{G\}$ y una sustitución γ tal que $\theta = \sigma\gamma$.*

Demostración:
Trivial a partir de 2.4.8 y 2.5.7. □

2.6 Procedimientos de SLD-Refutación

En esta sección vamos a formalizar la definición de SLD-árbol, que no es más que el árbol formado por todas las posibles derivaciones de un programa y un objetivo. El problema consiste ahora en estudiar la existencia o no de ramas con éxito dentro de un árbol. Este estudio es más propio de la Teoría de Grafos que de la Programación Lógica, por lo que no entramos a estudiarlo en profundidad. Nos limitamos a presentar sin demostración[8] tres teoremas elementales (2.6.4, 2.6.5, 2.6.6) y un ejemplo clásico debido a Apt y van Emden que pone de manifiesto la importancia de estos procedimientos y estrategias.

2.6.1 Definición. Sean P un programa definido y G un programa definido. Un *SLD-árbol* para $P \cup \{G\}$es un árbol satisfaciendo las siguientes condiciones:

1. Cada nodo del árbol es un objetivo definido (puede ser la cláusula vacía).
2. El nodo raíz es G.
3. Sea $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ $(k \geq 1)$ un nodo del árbol y supongamos que A_m es el átomo seleccionado. Entonces, para cada cláusula de entrada $A \leftarrow B_1, \ldots, B_q$ tal que A_m y A son unificables mediante el umg θ el nodo tiene un hijo
$$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$$
4. Los nodos que son cláusulas vacías no tienen hijos.

Cada rama del SLD-árbol es una derivación de $P \cup \{G\}$. Las ramas correspondientes a derivaciones con éxito las llamaremos *ramas con éxito*, las ramas correspondientes a derivaciones infinitas las llamaremos *ramas infinitas* y las ramas que corresponden a derivaciones fallidas las llamaremos *ramas fallidas*. Diremos que un SLD-árbol *tiene éxito* si contiene alguna rama con éxito.

Podemos enunciar ahora el siguiente

Teorema. 2.6.2 (Teorema del éxito) *Sea P un programa definido y A un átomo cerrado. Son equivalentes:*

1. *A pertenece al conjunto de éxito de P.*
2. *$A \in T_P \uparrow \omega$*

[8] Las demostraciones pueden encontrarse en Lloyd [37].

3. *Cada SLD-árbol de $P \cup \{\leftarrow A\}$ tiene éxito.*

4. $P \models A$.

Demostración:
Por los teoremas 1.2.23, 1.3.14 y 2.4.5 tenemos las equivalencias (1) $\Longleftrightarrow$ (2) $\Longleftrightarrow$ (4). Por otra parte, a partir de la definición de SLD-árbol, la afirmación (3) equivale a afirmar

"Existe una refutación de $P \cup \{\leftarrow A\}$"

de donde la equivalencia entre (1) y (3) resulta evidente. □

2.6.3 Definición. Sea P un programa definido, G un objetivo definido y R una regla de computación. El SLD-árbol de $P \cup \{G\}$ *vía* R es el SLD-árbol de $P \cup \{G\}$ en el que se ha usado R como regla de computación.

2.6.4 Teorema. *Sea P un programa definido, G un objetivo definido y R una regla de computación. Supongamos que $P \cup \{G\}$ es insatisfacible. Entonces el SLD-árbol de $P \cup \{G\}$ vía R tiene al menos una rama con éxito.*

2.6.5 Teorema. *Sea P un programa definido, G un objetivo definido y R una regla de computación. Entonces cada respuesta correcta θ de $P \cup \{G\}$ está puesta de manifiesto ("displayed") en el SLD-árbol de $P \cup \{G\}$ vía R. Donde "puesta de manifiesto" significa que, dado θ hay una rama con éxito tal que θ es una instancia de la respuesta computada de la refutación correspondiente a esa rama.*

2.6.6 Teorema. *Sea P un programa definido y G un objetivo definido . Entonces o bien todo SLD-árbol de $P \cup \{G\}$ tiene infinitas ramas de éxito, o bien todo SLD-árbol de $P \cup \{G\}$ tiene el mismo número finito de ramas de éxito.*

2.6.7 Definición. Una *regla de búsqueda* es una estrategia para encontrar ramas con éxito dentro de los SLD- árboles. Un *procedimiento de SLD- refutación* viene dado por una regla de computación junto con una regla de búsqueda.

Finalizamos este capítulo con un ejemplo clásico debido a Apt y van Emden [6] que pone de manifiesto cómo puede cambiar la forma y el tamaño de un SLD-árbol al cambiar la regla de computación.

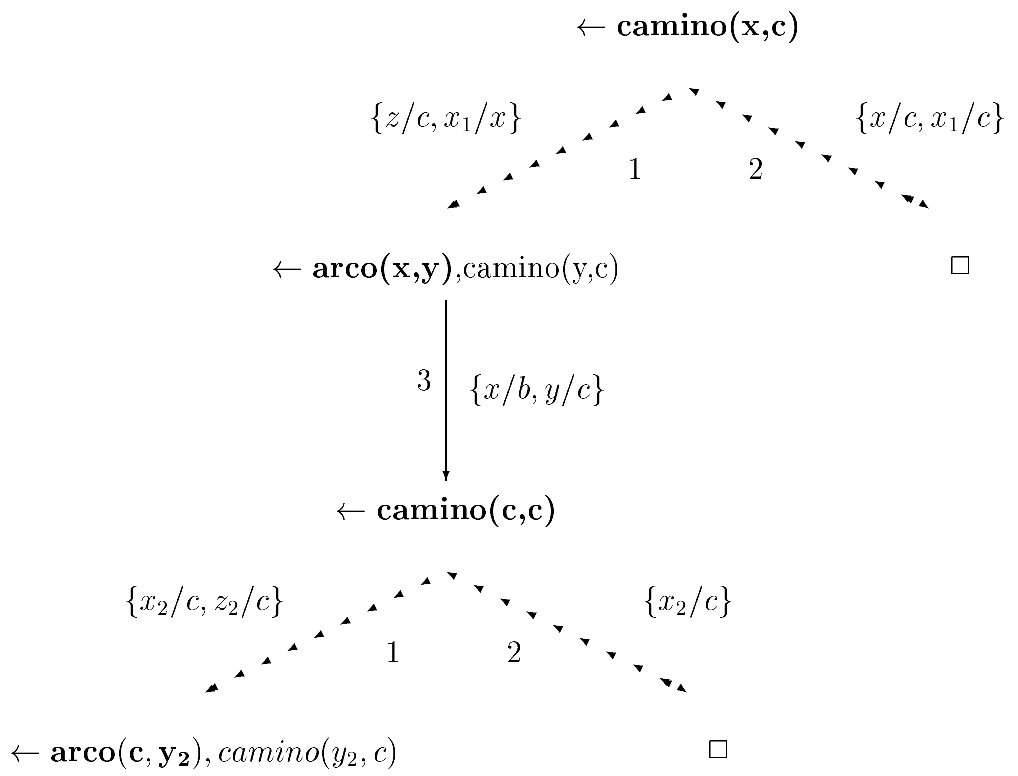
2.6.8 Ejemplo. Se P el siguiente programa

1. $camino(x, z) \leftarrow arco(x, y), camino(y, z)$
2. $camino(x, x) \leftarrow$
3. $arco(b, c) \leftarrow$

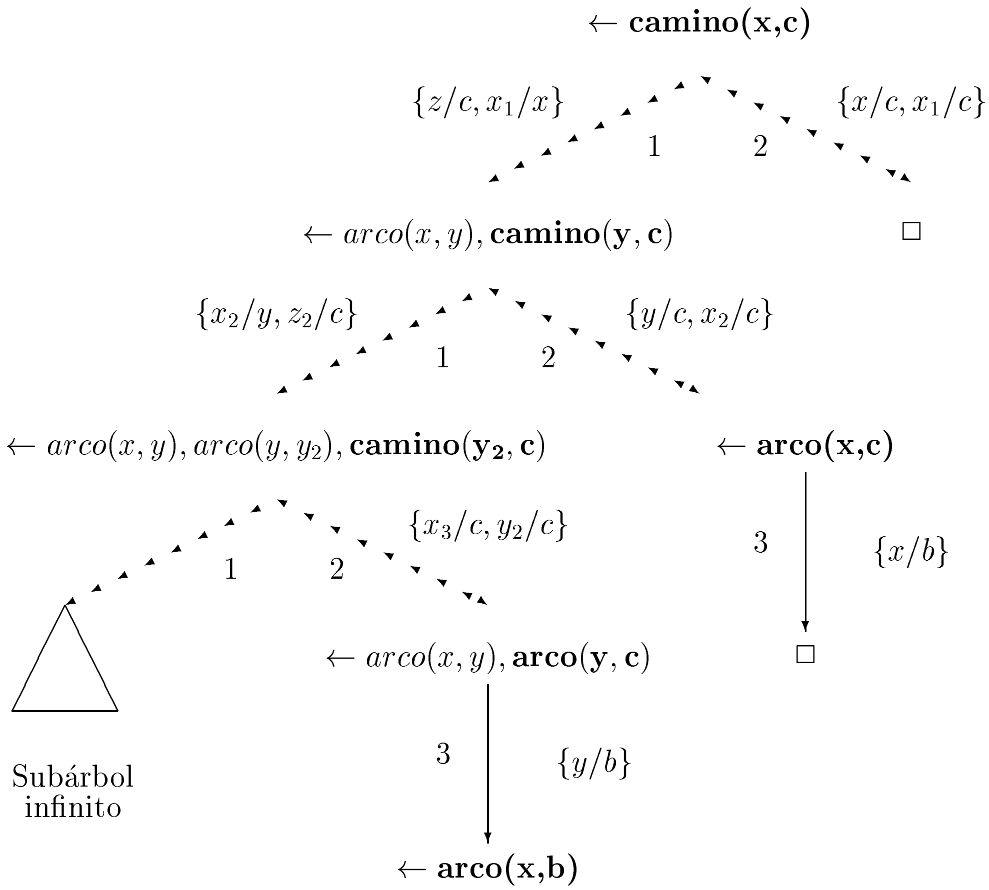
Una posible interpretación de P es la evidente, esto es, se verifica $camino(x, y)$ si existe un camino de x a y y se verifica $arco(x, y)$ si existe un arco de x a y. Las dos figuras siguientes muestran SLD-árboles de

$$P \cup \{\leftarrow camino(x, c)\}$$

El átomo seleccionado está en negrilla y las cláusulas se modifican según las sustituciones indicadas. Las cláusulas de entrada del nivel i se obtienen de la cláusula original añadiendo el subíndice i a las variables que ya han sido consideradas en la derivación. De esta forma se garantiza la condición de la separación de variables.



Nótese que este primer SLD-árbol es finito, en cambio el que ofrecemos a continuación es infinito.



2.6.9 Nota. Existen numerosos estudios sobre estrategias y procedimientos de refutación. Podemos encontrar una exposición clara en [22].

Capítulo 3

Funciones recursivas

En este capítulo estudiamos la relación que hay entre la Programación Lógica y la Teoría de la Recursión, demostrando que toda función recursiva puede ser computada por un programa definido.

3.1 Interpretaciones declarativa y procedural

El estudio de la Programación Lógica admite dos interpretaciones contrapuestas y a la vez complementarias a las que ya hemos hecho referencia: Una interpretación declarativa y otra procedural.

La interpretación declarativa hace referencia a *qué* debe ser computado por el programa, nos da el objetivo, a dónde queremos llegar, independientemente de que exista un algoritmo o no que nos conduzca a ese objetivo. Entre los conceptos que hemos definido, tienen un clara interpretación declarativa: El menor modelo de Herbrand, el conjunto de fallo finito, etc...

La interpretación procedural hace referencia a *cómo* computa el programa, a cuáles son los pasos que sigue para conseguir su objetivo. Tienen una fuerte interpretación procedural conceptos como: El conjunto de éxito de un programa, el SLD- conjunto de fallo finito, etc...

Resumiendo, podemos decir que la interpretación declarativa hace referencia al *significado*, al *sentido* de lo que hacemos y la interpretación procedural al *método*, al *procedimiento*.
Bajo esta perspectiva general, los teoremas de adecuación demuestran que los conceptos definidos de forma procedural realmente son los mismos que aquellos

que se han definido de forma declarativa y por otra parte los teoremas de completitud no hacen más que probar que los conceptos definidos de forma declarativa son los mismos que los definidos de forma procedural.

3.2 Funciones recursivas

3.2.1 Definición. Se definen las *funciones básicas* como las siguientes funciones

1. $\mathcal{O} : \omega \to \omega \quad \mathcal{O}(x) = 0$ *Función cero*
2. $\mathcal{S} : \omega \to \omega \quad \mathcal{S}(x) = x + 1$ *Función sucesor*
3. $\Pi_i^n : \omega^n \to \omega \quad \Pi_i^n(x_1, \ldots, x_n) = x_i \quad \forall n \geq 1 \quad \forall i \in \{1, \ldots, n\}$
   Función proyección

3.2.2 Definición.

1. Si $f : \omega^n \to \omega$ y $g_1, \ldots, g_n : \omega^m \to \omega$ son funciones parciales, la aplicación parcial $h : \omega^m \to \omega$ definida por

$$h(a_1, \ldots, a_m) = f(g_1(a_1, \ldots, a_m), \ldots, g_n(a_1, \ldots, a_m))$$

   diremos que se obtiene de $f, g_1, g_2, \ldots, g_n$ por *composición.*

2. Si $f : \omega^n \to \omega$ y $g : \omega^{n+2} \to \omega$ son funciones parciales, la aplicación parcial $h : \omega^{n+1} \to \omega$ definida por

$$\begin{array}{l} h(a_1, \ldots, a_n, 0) = f(a_1, \ldots, a_n) \\ h(a_1, \ldots, a_n, x+1) = g(a_1, \ldots, a_n, x, h(a_1, \ldots, a_n, x)) \end{array}$$

   diremos que se obtiene de f y g por *recursión* (o *recursión primitiva*).

3. Sean $f : \omega^{n+1} \to \omega$ y $(a_1, \ldots, a_n) = \vec{a}$, la aplicación

$$\mu x(f(\vec{a}, x) = 0) = \begin{cases} y, & \text{si } f(\vec{a}, y) = 0 \text{ y } \forall z < y(f(\vec{a}, z) \downarrow \neq 0) \\ \uparrow, & \text{en otro caso} \end{cases}$$

   diremos que se obtiene de f por $\mu-$ *recursión*[1]

Definición. 3.2.3 (Dedekind, Skolem, Gödel) La clase de las *funciones primitivas recursivas* es la menor clase de funciones que

[1] $\uparrow$ significa que la función no está definida y $\downarrow$ que sí está definida.

1. Contiene las funciones básicas.
2. Es cerrada bajo composición.
3. Es cerrada bajo recursión primitiva.

3.2.4 Definición. La clase de las *funciones recursivas* es la menor clase de funciones que

1. Contiene las funciones básicas
2. Es cerrada bajo composición, recursión primitiva y μ-recursión.

Nuestro objetivo es establecer un resultado esencial dentro de la fundamentación teórica de la Programación Lógica, y es el hecho de que toda función computable puede ser computada por un programa definido. Podemos encontrar distintos artículos tratando este tema, cada uno de los cuales asume una definición diferente de “función computable”. Así podemos encontrar demostraciones de que toda función Turing computable puede ser computada por un programa definido (Tarnlund, [48]), o bien usando máquinas de registro ilimitadas para definir funciones computables (Shepherdson, [46]). Kowalski [31] estableció el resultado mostrando cómo transformar un conjunto de ecuaciones recursivas en un programa definido. Andreka y Nemeti [2] y Sonenberg y Topor [47] demostraron la adecuación de los programas definidos para la computación sobre un universo de Herbrand. Sebelik y Stepanek [45] demostraron que toda función recursiva parcial puede ser computada por un programa definido.

Presentamos aquí los teoremas de Andreka y Nemeti y Sebelik y Stepanek.

3.3 Teorema de Andreka y Nemeti

3.3.1 Nota. Asumiremos que **L** es un lenguaje con al menos un símbolo de constante. Asímismo supondremos que para cada $n \in \omega$ el lenguaje tiene una cantidad infinita numerable de símbolos de predicado de aridad n.

3.3.2 Definición. Diremos que un programa definido P *computa* un subconjunto R de $U^n_{\mathbf{L}}$, mediante el símbolo de predicado r de aridad n si para toda n-upla $(t_1, \ldots, t_n) \in U^n_{\mathbf{L}}$ se tiene

$$(t_1, \ldots, t_n) \in R \quad \Longleftrightarrow \quad \text{Existe una SLD-refutación de } P \cup \{\leftarrow r(t_1, \ldots, t_n)\}$$

Esta definición, de clara intención procedural, tiene su contrapartida declarativa:

3.3.3 Definición. Diremos que un programa definido P *define* un subconjunto R de $U_{\mathbf{L}}^n$, mediante el símbolo de predicado r de aridad n si para toda n-upla $(t_1, \ldots, t_n) \in U_{\mathbf{L}}^n$ se tiene

$$(t_1, \ldots, t_n) \in R \quad \Longleftrightarrow \quad P \models r(t_1, \ldots, t_n)\}$$

3.3.4 Teorema. *Sea P un programa definido, $R \subseteq U_{\mathbf{L}}^n$ y r un símbolo de predicado n-ario de $\mathbf{L}$. Entonces las siguientes condiciones son equivalentes:*

1. *P computa R usando r.*

2. *P define R usando r.*

3. *Para todo $(t_1, \ldots, t_n) \in U_{\mathbf{L}}^n$ se tiene*

$$(t_1, \ldots, t_n) \in R \quad \Longleftrightarrow \quad r(t_1, \ldots, t_n) \in M_P$$

Demostración:
Se tiene por aplicación directa de los teoremas 1.3.14 y 2.6.2. □

Así, el problema de cuáles son los subconjuntos computables por programas definidos se reduce a estudiar qué subconjuntos del último modelo de Herbrand puede ser definido mediante átomos. Nosotros vamos a estudiar el caso en que el lenguaje $\mathbf{L}$ tenga al menos un símbolo de constante y al menos un símbolo de función, con lo que garantizaremos que $U_{\mathbf{L}}$ es infinito.
La asunción de que $\mathbf{L}$ contiene infinitos símbolos de predicado para cada aridad n, con $n \in \omega$ nos permite construir nuevas cláusulas sin problemas sintácticos. También exigiremos que tanto el conjunto de las constantes del lenguaje como el conjunto de los símbolos de función sean finitos

3.3.5 Definición. Diremos que una relación binaria R en $U_{\mathbf{L}}$ es una *enumeración de $U_{\mathbf{L}}$* si R define una función sucesor en $U_{\mathbf{L}}$; esto es, existe una aplicación biyectiva $\phi : \omega \to U_{\mathbf{L}}$ tal que para todo par $(x, y) \in R$, existe $n \in \omega$ tal que $x = \phi(n)$ e $y = \phi(n+1)$

Como primer paso hacia la caracterización de los predicados computables por programas lógicos probamos el siguiente resultado debido a Andreka y Nemeti [2]. La presentación que hacemos se debe a Blair [9].

Teorema. 3.3.6 (Teorema de enumeración) *Existe un programa* `sucesor` *tal que computa una enumeración de $U_{\mathbf{L}}$ usando la relación binaria* `suc`.

Demostración:
La construcción del programa `sucesor` es bastante tediosa. La idea que Blair utiliza es definir una función *peso* de un término cerrado [2]

$$peso(x) = 0 \text{ si } x \text{ es una constante}$$
$$peso(f(t_1, \ldots, t_n)) = 1 + max\{peso(t_i) : i \in \{1, \ldots, n\}\}$$

y un buen orden en el conjunto de símbolos de función y constantes mediante una biyección con ω. Más tarde con ayuda de la función peso establece un buen orden en el conjunto de los términos cerrados. La construcción del programa `sucesor` se lleva a cabo trasladando minuciosamente la definición de ese buen orden al lenguaje de las cláusulas. Los detalles pueden encontrarse en [3]. $\square$

3.3.7 Proposición. *Para todo programa P, M_P es recursivamente enumerable.*

Demostración:
Por el teorema 1.3.14 tenemos que $A \in M_P$ si y solo si existe $k \in \omega$ tal que $A \in T_P \uparrow k$, por tanto el resultado es inmediato, puesto que, mediante una codificación apropiada es fácil ver que el predicado *pertenecer a $T_P \uparrow k$* es recursivo. $\square$

El teorema de enumeración nos permite identificar cualquier universo de Herbrand, con las retricciones antes expuestas, con el conjunto de los números naturales, por lo que podemos aplicar ahora sobre este conjunto la teoría de la recursión.

Teorema. 3.3.8 (Andreka y Nemeti, 1978) *Para toda función recursiva f existe un programa P que computa el grafo de f usando una relación p_f.*

3.3.9 Nota. No damos aquí la demostración del teorema de Andreka y Nemeti puesto que es análoga a la que presentamos del teorema de Sebelik y Stepanek. Una presentación amena de esta demostración puede encontrarse en [3].

3.3.10 Corolario. *Un predicado R definido sobre $U_{\mathbf{L}}$ es recursivamente enumerable si y sólo si existe un programa P que lo computa usando una relación r.*

[2]Nótese la semejanza entre el concepto que Blair define como *peso (height)* y el que Lassez, Maher y Marriot definen como *profundidad (depth)*, [34].

Demostración:
($\Rightarrow$) Supongamos que para algún predicado recursivo S se tiene que

$$\vec{a} \in R \Longleftrightarrow \exists b(\vec{a}, b) \in S$$

Sea P_S es programa que computa la función característica de S, C_S, usando la relación p_S. Entonces el programa P_S aumentado con la cláusula

$$p_R(x_1, \ldots, x_k) \leftarrow p_S(x_1, \ldots, x_k, y, 0)$$

computa el predicado R usando es símbolo de predicdo p_R.
($\Leftarrow$) Se tiene directamente a partir de los teoremas 3.3.4 y 3.3.7. □

3.3.11 Corolario. *Si un programa P computa el grafo de una función total usando alguna relación, entonces la función es recursiva.*

Demostración:
Una función total es recursiva si y sólo si su grafo es recursivamente enumerable. □

3.3.12 Corolario. *Existe un progrma P tal que M_P es r.e.-completo.*

Demostración:
Sea R una relación r.e.-completa definida sobre $U_{\mathbf{L}}$. Por el corolario 3.3.10 y el teorema 3.3.4 tenemos que para todo $\vec{a}$,

$$\vec{a} \in R \quad \Longleftrightarrow \quad r(\vec{a}) \in M_P$$

donde P es un programa que computa R usando una relación r. Esto prueba que M_P es r.e.-completo. □

3.3.13 Nota. En [3] podemos encontrar diversos resultados relacionados con este teorema.

3.4 Teorema de Sebelik y Stepanek

Teorema. 3.4.1 (Sebelik y Stepanek, 1982) *Sea f una función recursiva parcial n-aria. Entonces existe un programa definido P_f y un símbolo de predicado $(n+1)$-ario p_f tal que todas las respuestas computadas de $P_f \cup \{\leftarrow p_f(s^{k_1}(0), \ldots, s^{k_n}(0), x)\}$ son de la forma $\{x/s^k(0)\}$ y para toda n-upla de enteros no negativos $(k_1, \ldots, k_n)$ y para todo entero no negativo k tenemos que $f(k_1, \ldots, k_n) = k$ si y sólo si $\{x/s^k(0)\}$ es una respuesta computada de $P_f \cup \{\leftarrow p_f(s^{k_1}(0), \ldots, s^{k_n}(0), x)\}$.*

Demostración:
En el programa P_f representamos el entero no negativo k como $s_k(0)$, donde s representa la función sucesor. Por el teorema 2.5.7 podemos suponer que toda respuesta computada es R- computada, donde R es la regla de selección en la que siempre seleccionamos el átomo *más a la izquierda*. Hacemos la demostración por inducción sobre q, el número de veces que aplicamos composición, recursión primitiva y $\mu-$recursión para definir f.

($q = 0$) En este caso f es una función básica.

1. Si f es la *función cero*, entonces P_f es el programa
$$p_f(x, 0) \leftarrow$$
2. Si f es la *función sucesor*, entonces P_f es el programa
$$p_f(x, s(x)) \leftarrow$$
3. Si f es la *función proyección*, $\Pi_i^n(x_1, \ldots, x_n) = x_i$ entonces P_f es el programa
$$p_f(x_1, \ldots, x_n, x_i) \leftarrow$$

Obviamente, para cada una de estas funciones, los programas dados cumplen las condiciones requeridas.
($q - 1 \Rightarrow q$) Supongamos ahora que la función recursiva parcial f está definida para q ($q > 0$) aplicaciones de composición, recursión primitiva y $\mu-$recursión.

Composición
Supongamos que la definición de f es

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

con $h, g_1, \ldots, g_m$ funciones recursivas parciales. Por hipótesis de inducción, para cada $i \in \{i, \ldots, m\}$ existe un programa definido P_{g_i} y un símbolo de predicado p_{g_i} que satisfacen las condiciones del teorema. Análogamente existe un programa definido P_h y un símbolo de predicado p_h que satisface las condiciones del teorema. Podemos suponer que los símbolos de predicado que ocurren en los programas $P_{g_1}, \ldots . P_{g_m}$ y P_h son distintos. Definimos P_f como la unión de esos símbolos de predicado junto con la cláusula

$$p_f(x_1, \ldots, x_n, z) \leftarrow p_{g_1}(x_1, \ldots, x_n, y_1), \ldots p_{g_m}(x_1, \ldots, x_n, y_m), p_h(y_1, \ldots, y_m, z)$$

Claramente podemos ver, por inducción, que toda respuesta computada de $P_f \cup \{\leftarrow p_f(s^{k_1}(0), \ldots, s^{k_n}(0), z\}$ es de la forma $\{z/s^k(0)\}$. Supongamos que

$f(k_1, \ldots, k_n) = k$ y para cada $i \in \{i, \ldots, m\}$ sea $g_i(k_1, \ldots, k_n) = n_i$. Por hipótesis de inducción tenemos que $\{y_i/s^{n_i}(0)\}$ es una respuesta computada de $P_{g_i} \cup \{\leftarrow p_{g_i}(s^{k_1}(0), \ldots, s^{k_n}(0), y_i\}$ y que $\{z/s^k(0)\}$ es una respuesta computada de $P_h \cup \{\leftarrow p_h(s^{n_1}(0), \ldots, s^{n_m}(0), z\}$, por tanto $\{z/s^k(0)\}$ es una respuesta computada para $P_f \cup \{\leftarrow p_f(s^{k_1}(0), \ldots, s^{k_n}(0), z\}$.
Por otra parte, supongamos que $\{z/s^k(0)\}$ es una respuesta computada de $P_f \cup \{\leftarrow p_f(s^{k_1}(0), \ldots, s^{k_n}(0), z\}$. De la refutación en la que obtenemos esta respuesta podemos extraer, para cada $i \in \{i, \ldots, m\}$, una respuesta computada $\{y_i/s^{n_i}(0)\}$ para $P_{g_i} \cup \{\leftarrow p_{g_i}(s^{k_1}(0), \ldots, s^{k_n}(0), y_i\}$ y una respuesta computada $\{z/s^k(0)\}$ para $P_h \cup \{\leftarrow p_h(s^{n_1}(0), \ldots, s^{n_m}(0), z\}$. Entonces, por hipótesis de inducción tenemos que, para cada $i \in \{i, \ldots, m\}$, $g_i(k_1, \ldots, k_n) = n_i$ y que $h(n_1, \ldots, n_m) = k$. De ahí que $f(k_1, \ldots, k_n) = k$.

Recursión primitiva
Supongamos que f tiene a siguiente definición:

$$\begin{array}{l} f(x_1, \ldots, x_n, 0) = h(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, y+1) = g(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \end{array}$$

donde h y g son funciones recursivas parciales. Por hipótesis de inducción, para las funciones h y g existen dos programas definidos P_h y P_g y dos símbolos de predicado p_h y p_g que satisfacen las propiedades del teorema. Podemos también suponer que P_h y P_g no tienen símbolos de predicado en común. Definimos P_f como la unión de P_h y P_g junto con las cláusulas

$$\begin{array}{l} p_f(x_1, \ldots, x_n, 0, z) \leftarrow p_h(x_1, \ldots, x_n, z) \\ p_f(x_1, \ldots, x_n, s(y), z) \leftarrow p_f(x_1, \ldots, x_n, y, u), p_g(x_1, \ldots, x_n, y, u, z) \end{array}$$

Un razonamiento similar al que hemos hecho con la composición nos demuestra que este programa tiene las propiedades deseadas.

μ−recursión
Supongamos que f está definida de la siguiente manera

$$f(x_1, \ldots, x_n) = \mu y(g(x_1, \ldots, x_n, y) = 0)$$

con g una función recursiva parcial. Por hipótesis de inducción tenemos que para la función g existe un programa definido P_g y un símbolo de predicado p_g que satisfacen las condiciones del teorema. Definimos P_f como P_g junto con las cláusulas siguientes[3]

$$
\begin{array}{llll}
p_f(\vec{x}, y) & \leftarrow & p_g(\vec{x}, y, 0), & \% \quad g(\vec{x}, y) = 0 \\
 & & r(\vec{x}, y) & \% \quad \forall z(z < y \rightarrow g(\vec{x}, z) > 0) \\
 & & & \\
r(\vec{x}, 0) & \leftarrow & & \\
 & & & \\
r(\vec{x}, y) & \leftarrow & suc(y', y), & \% \quad y = y' + 1 \\
 & & r(\vec{x}, y'), & \% \quad \forall z(z < y' \rightarrow g(\vec{x}, z) > 0) \\
 & & p_g(\vec{x}, y', z), & \% \quad g(\vec{x}, y') = z \\
 & & p_<(0, z) & \% \quad 0 < z
\end{array}
$$

Donde tras el símbolo % indicamos las interpretaciones intencionadas de las distintas cláusulas.
Un argumento similar a los usados anteriormente, puesto que los predicados utilizados son recursivos, demuestra que el programa definido P_f cumple la tesis del teorema. □

[3]Para simplificar la notación escribimos $\vec{x}$ en lugar de la n-upla $x_1, \ldots, x_n$.

Capítulo 4

Información Negativa

4.1 Introducción

Hasta el momento, con nuestro estudio sobre Programación Lógica sólo podemos demostrar teoremas que nos dan información positiva; es decir, sólo los literales positivos pueden ser ser consecuencia lógica de un programa. Así, si tenemos un programa definido P y un átomo $A \in B_P$ no podemos probar que $\neg A$ sea consecuencia lógica de P. La razón está en que $P \cup \{A\}$ es satisfacible teniendo a B_P como modelo.

4.1.1 Ejemplo. Sea **L** el lenguaje de primer orden sin símbolos de función n- arios ($n > 0$), con un único símbolo de predicado 1-ario: "alumno" y el siguiente conjunto de constantes *SC*=\{*Esther, Juan, Rosa, Miguel, Pepe*\}. Consideremos el siguiente programa P sobre **L**:
$alumno(Esther) \leftarrow$
$alumno(Juan) \leftarrow$
$alumno(Rosa) \leftarrow$
Supongamos que nuestro objetivo es probar que $\neg alumno(Miguel)$ es consecuencia lógica de P. Con las herramientas que tenemos hasta ahora no podemos, ya que B_P es modelo de $P \cup \{\leftarrow \neg alumno(Miguel)\}$, pero tampoco podríamos probar $alumno(Pepe)$ ya que *I*=\{*alumno(Esther), alumno(Juan), alumno(Rosa)*\} es modelo de $P \cup \{\leftarrow alumno(Pepe)\}$.

Para intentar solucionar nuestro problema vamos a ampliar el conjunto de nuestras reglas de inferencia para poder deducir información negativa. La primera regla de inferencia que presentamos para deducir información negativa, fue presentada por Reiter [43].

4.1.2 Definición. [Regla de Inferencia HMC][1] Si un átomo cerrado A no es consecuencia lógica de un programa, entonces inferimos $\neg A$.

4.1.3 Nota. El acrónimo HMC es la abreviatura de *hipótesis del mundo cerrado*, traducción del acrónimo inglés CWA, *closed world assumption*.

4.1.4 Ejemplo. Así, en el ejemplo anterior el átomo cerrado $alumno(Pepe)$ no es consecuencia lógica del programa P y por tanto, usando la regla HMC podemos inferir $\neg alumno(Pepe)$.

La HMC es la forma usual de razonamiento cuando se trabaja con bases de datos: La información que no está presente de forma explícita en la base de datos se toma como falsa. La HMC es un ejemplo de regla de inferencia no monótona.

4.1.5 Definición. Una regla de inferencia $\vdash$ se dice *monótona* si dados dos programas P y P', para toda fórmula ϕ se tiene

$$P \vdash \phi \Longrightarrow P \cup P' \vdash \phi$$

esto es, la regla de inferencia es *no monótona* si añadiendo nuevos axiomas puede reducirse el número de teoremas.

En consecuencia, dado un programa P al que podamos aplicar la HMC y dado un átomo $A \in B_P$, si queremos probar $\neg A$ lo único que tenemos que ver es que A no es consecuencia lógica de P. Pero desgraciadamente el problema de la validez en lógica de primer orden es indecidible, por tanto no hay ningún algoritmo que tomando como datos de entrada un programa definido P y un átomo $A \in B_P$ nos devuelva en un tiempo finito si A es o no consecuencia lógica de P, ya que si el átomo cerrado A no es consecuencia lógica de P podemos entrar en un proceso infinito. En la práctica restringimos la aplicación de la HMC únicamente a aquellos átomos cerrados tales que al intentar hacer una refutación fallamos de forma finita.

4.2 Fallo finito

4.2.1 Nota. El planteamiento que hacemos aquí fue presentado por Lloyd en 1987 (ver [37]) y es generalmente aceptado. No obstante otros autores han publicado trabajos que difieren de esta presentación.[2]

[1] Apt nace una presentación detallada de esta regla de inferencia en [3].
[2] Ver la sección "Resultados recientes" en este mismo capítulo.

La regla de inferencia denominada *regla de fallo finito* es menos potente que la HMC y nos permite inferir menos información, no obstante tiene la ventaja de ser fácilmente automatizable. Es la siguiente:

Dado un programa definido P y un objetivo definido $\leftarrow A$, si existe un SLD-árbol finito sin ramas con éxito de $P \cup \{\leftarrow A\}$ deducimos $\neg A$

Veamos algunos resultados relacionados con esta regla:

4.2.2 Definición. Sea P un programa definido. Se define *el conjunto de átomos de B_P con fallo finito* de profundidad d, F_P^d, de la siguiente manera:

1. $A \in F_P^1$ si $A \notin T_P \downarrow 1$.
2. $A \in F_P^d$ para $d > 1$ si para toda cláusula $B \leftarrow B_1, \ldots, B_n$ de P y toda sustitución θ tal que $A = B\theta$ y $B_1\theta, \ldots, B_n\theta$ sean átomos cerrados , existe un $k \in \{1, \ldots, n\}$ y $B_k\theta \in F_P^{d-1}$.

4.2.3 Definición. Sea P un programa definido. Se define *el conjunto de fallo finito*, F_P de P como el conjunto

$$F_P = \cup_{d \geq 1} F_P^d$$

4.2.4 Proposición. *Sea P un programa definido, entonces*

$$F_P = B_P \backslash T_P \downarrow \omega$$

Demostración:
En la demostración de esta proposición necesitamos el siguiente **Aserto.**

$$B_P \backslash F_P^d = T_P \downarrow d \ \forall d \geq 1$$

Suponiendo cierto el aserto se tiene que

$$B_P \backslash F_P = B_P \backslash \cup_{d \geq 1} F_P^d = \cap_{d \geq 1}(B_P \backslash F_P^d) = \cap_{d \geq 1} T_P \downarrow d = T_P \downarrow \omega$$

Demostración del aserto
($\supseteq$) Lo vemos por inducción.
($d = 1$) Sea $A \in B_P$. Por definición, si $A \notin F_P^1$ entonces $A \in T_P \downarrow 1$.
($d \Rightarrow d + 1$) Sea $A \in B_P$. Si $A \notin F_P^{d+1}$ entonces existe una cláusula $B \leftarrow B_1, \ldots, B_n$ y una sustitución θ con $A = B\theta$ y $\{B_1, \ldots, B_n\} \subseteq B_P$ tal que para todo $k \in \{1, \ldots, n\}$ $B_k\theta \in B_P \backslash F_P^d$. Pero por hipótesis de inducción $B_P \backslash F_P^d \subseteq T_P \downarrow d$ con lo que tenemos que existe una cláusula $B \leftarrow B_1, \ldots, B_n$ y una sustitución θ con $A = B\theta$ y $\{B_1, \ldots, B_n\} \subseteq B_P$ tal que para todo $k \in \{1, \ldots, n\}$ $B_k\theta \in T_P \downarrow d$, por tanto $A \in T_P \downarrow d + 1$, como queríamos

probar.
($\subseteq$) También lo vemos por inducción
($d = 1$) Sea $A \in B_P$. Por definición, si $A \in T_P \downarrow 1$ entonces $A \notin F_P^1$.
($d \Rightarrow d+1$) Sea $A \in B_P$. Si $A \in T_P \downarrow d+1$ entonces existe una cláusula $B \leftarrow B_1, \ldots, B_n$ y una sustitución θ con $A = B\theta$ y $\{B_1\theta, \ldots, B_n\theta\} \subseteq T_P \downarrow d$. Pero por hipótesis de inducción $T_P \downarrow d \subseteq B_P \backslash F_P^d$, con lo que tenemos que existe una cláusula $B \leftarrow B_1, \ldots, B_n$ y una sustitución θ con $A = B\theta$ y $\{B_1, \ldots, B_n\} \subseteq B_P$ tal que para todo $k \in \{1, \ldots, n\}$ $B_k\theta \in T_P \downarrow d$, por tanto $A \notin F_P^d$. □

4.2.5 Definición. Sea P un objetivo definido y G un objetivo definido. Un *SLD-árbol finitamente fallido* de $P \cup \{G\}$ es un SLD-árbol finito que no contiene ramas con éxito.

4.2.6 Definición. Sea P un programa definido, se define el *SLD-conjunto de fallo finito* de P como el conjunto de todos los átomos $A \in B_P$ para los cuales existe un SLD-árbol finitamente fallido de $P \cup \{\leftarrow A\}$, esto es un árbol finito sin ramas con éxito.

Nótese que sólo se exige la existencia de *un* SLD-árbol finitamente fallido de $P \cup \{\leftarrow A\}$ para que $A \in B_P$ pertenezca al SLD-conjunto de fallo finito de P. La siguiente proposición y los dos lemas previos se deben a Apt y van Emden. No presentamos la demostración de los lemas, pero pueden encontrarse en [6].

4.2.7 Lema. *Sea P un programa definido, G un objetivo definido y θ una sustitución. Supongamos que $P \cup \{G\}$tiene un SLD-árbol finitamente fallido de profundidad $\leq k$. Entonces $P \cup \{G\theta\}$ tiene también un SLD-árbol finitamente fallido de profundidad $\leq k$.*

4.2.8 Lema. *Sea P un programa definido y $\{A_1, \ldots, A_m\} \subseteq B_P$. Supongamos que $P \cup \{\leftarrow A_1, \ldots, A_m\}$ tiene un SLD-árbol finitamente fallido de profundidad $\leq k$ entonces existe $i \in \{1, \ldots, m\}$ tal que $P \cup \{\leftarrow A_i\}$ tiene un SLD-árbol finitamente fallido de profundidad $\leq k$.*

4.2.9 Proposición. *Sea P un programa definido y $A \in B_P$. Si $P \cup \{\leftarrow A\}$ tiene un SLD-árbol finitamente fallido de profundidad $\leq k$ entonces se tiene que $A \notin T_P \downarrow k$.*

Demostración:
Lo probamos por inducción sobre k.
($k = 1$) Supongamos que $P \cup \{\leftarrow A\}$ tiene un árbol finitamente fallido de

profundidad uno, esto es, A no unifica con la cabeza de ninguna cláusula de P. Obviamente $A \notin T_P \downarrow 1$.
$(k-1 \Rightarrow k)$ Lo vemos por reducción al absurdo. Supongamos que para todo $A' \in B_P$ tal que $P \cup \{\leftarrow A'\}$ tiene un SLD-árbol finitamente fallido de profundidad menor o igual que k se verifica que $A' \notin T_P \downarrow k$, esto es, se verifica la tesis de la proposición para $k-1$, y que existe $A \in B_P$ con un árbol finitamente fallido de $P \cup \{\leftarrow A\}$ de profundidad menor o igual que k y a su vez $A \in T_P \downarrow k$.
Si $A \in T_P \downarrow k$, existe una cláusula $B \leftarrow B_1, \ldots, B_n$ y una sustitución θ tal que $A = B\theta$ y $\{B_1\theta, \ldots, B_n\theta\} \subseteq T_P \downarrow (k-1)$. Por tanto podemos encontrar un umg γ de A y B, $A = B\gamma$ tal que $\theta = \gamma\sigma$ para algún σ. Pero entonces $\leftarrow (B_1, \ldots, B_n)\gamma$ es raíz de un SLD-árbol finitamente fallido de profundidad menor o igual que $k-1$. Por el lema 4.2.7 también existe un SLD-árbol finitamente fallido de profundidad menor o igual que $k-1$ para $P \cup \{\leftarrow (B_1, \ldots, B_n)\theta\}$. Aplicando ahora el lema 4.2.8 existe $i \in \{1, \ldots, n\}$ tal que existe un árbol finitamente fallido para $P \cup \{\leftarrow B_i\theta\}$ de profundidad menor o igual que $k-1$. Pero por hipótesis de inducción, si esto ocurre $B_i\theta \notin T_P \downarrow k-1$, con lo que llegamos a contradicción. □

La definición 4.2.6 sólo exige la existencia de un árbol finitamente fallido. Sería muy útil que tuviéramos alguna forma de encontrar ese árbol finitamente fallido, caso de que exista. La siguiente definición, presentada por Lassez y Maher en [33] va en esa dirección.

4.2.10 Definición. Una SLD-derivación es *favorable*[3] si

- O bien es fallida
- O bien cada átomo B de la derivación[4], B es seleccionado en un número finito de pasos.

4.2.11 Definición. Diremos que un SLD-árbol es *favorable* si cada rama del árbol es una SLD-derivación favorable.

4.2.12 Proposición. *Sea P un programa definido y $\leftarrow A_1, \ldots A_m$ un objetivo definido. Supongamos que existe una SLD-derivación favorable no fallida $\leftarrow A_1, \ldots A_m = G_0, G_1, \ldots$ con los umg $\theta_1, \theta_2, \ldots$. Entonces, dado $k \in \omega$ existe $n \in \omega$ tal que $[A_i\theta_1, \ldots \theta_n] \subseteq T_P \downarrow k$, para $i \in \{1, \ldots, m\}$.*

Demostración:
Por el teorema 2.3.5 podemos asumir que la derivación es infinita. Claramente

[3] Traducción libre del término inglés *"fair"*.
[4] o alguna instancia suya

es suficiente probar que dado $i \in \{1, \dots, m\}$ y $k \in \omega$ existe $n \in \omega$ tal que $[A_i\theta_1, \dots \theta_n] \subseteq T_P \downarrow k$.
Fijemos $i \in \{1, \dots, m\}$ y veamos que el resultado es cierto por innducción sobre k.

($k = 0$) Trivial.

($k - 1 \to k$) Asumamos que el resultado es cierto para $k - 1$ y supongamos que $A_i\theta_1 \dots \theta_{p-1}$ es seleccionado en el objetivo G_{p-1}. (Por ser la derivación favorable, A_i es seleccionado en algún momento). Sea G_p el objetivo $\leftarrow B_1, \dots, B_q$ con $q \geq 1$. Por hipótesis de inducción existe $s \in \omega$ tal que $\cup_{j=1}^{q}[B_j\theta_{p+1}, \dots \theta_{p+s}] \subseteq T_P \downarrow (k-1)$. De ahí que

$$[A_i\theta_1, \dots \theta_{p+s}] \subseteq T_P(\cup_{j=1}^{q}[B_j\theta_{p+1}, \dots \theta_{p+s}] \subseteq T_P(T_P \downarrow (k-1)) = T_P \downarrow k$$

Y esto concluye la demostración. □

Combinando ahora los resultados de Apt y van Emden [6] y Lassez y Maher [33] obtenemos la siguiente caracterización del conjunto de fallo finito de un programa.

4.2.13 Teorema. *Sea P un programa definido y $A \in B_P$. Las siguientes condiciones son equivalentes:*

1. $A \in F_P$
2. $A \notin T_P \downarrow \omega$
3. *A está en el SLD-conjunto de fallo finito.*
4. *Cada SLD-árbol favorable de $P \cup \{\leftarrow A\}$ es finitamente fallido.*

Demostración:
El esquema de la demostración es el siguiente:

$$\begin{cases} 1 \Longleftrightarrow 2 \\ 4 \Rightarrow 3 \Rightarrow 2 \Rightarrow 4 \end{cases}$$

Con lo que tendremos probada la equivalencia entre las cuatro afirmaciones.
(1) $\Leftrightarrow$ (2) Se tiene por la proposición 4.2.4.
(4) $\Rightarrow$ (3) Obvio.
(3) $\Rightarrow$ (2) Por la proposición 4.2.9.
(2) $\Rightarrow$ (4) Lo vemos por reducción al absurdo. Supongamos que es falsa la afirmación (4), esto es, existe una SLD-derivación favorable de $\leftarrow A$ que no falla en un número finito de pasos. Entonces, por la proposición 4.2.12 para todo $k \in \omega$ $A \in T_P \downarrow k$, esto es $A \in T_P \downarrow \omega$. Contradicción con (2). □

Este teorema muestra que la SLD-derivación favorable es una implantación adecuada y completa de la regla de fallo finito.

4.3 Programas normales

En esta sección seguimos nuestro estudio sobre la información negativa ampliando el concepto de programa definido a programa normal.

4.3.1 Definición. Una *cláusula de programa* es una cláusula de la forma

$$A \leftarrow L_1, \ldots, L_n$$

donde A es un átomo y $L_1, \ldots L_n$ son literales.

Nótese que las cláusulas de programa definido son un caso particular de las cláusulas de programa.

4.3.2 Definición. Un *programa normal* [5] es un conjunto finito de cláusulas de programa.

4.3.3 Definición. Un *objetivo normal* es una cláusula de la forma

$$\leftarrow L_1, \ldots, L_n$$

donde $L_1, \ldots, L_n$ son literales.

4.3.4 Definición. La *definición* de un símbolo de predicado p en un programa normal P es el conjunto de todas las cláusulas de programa de P que tienen el símbolo p en su cabeza.

4.3.5 Definición. Adoptaremos la siguiente definición de completación de un programa:

Sea $p(t_1, \ldots, t_n) \leftarrow L_1, \ldots, L_m$ una cláusula de un programa normal P. Consideraremos un nuevo símbolo de predicado $=$, que no aparezca en $\mathbf{L}_P$, con la interpretación “intencionada” de la relación identidad. Vamos a transformar la cláusula dada siguiendo los siguientes pasos:

1. Transformamos la cláusula dada en

$$p(x_1, \ldots, x_n) \leftarrow (x_1 = t_1) \wedge \ldots \wedge (x_n = t_n) \wedge L_1 \wedge \ldots \wedge L_m$$

   donde $x_1, \ldots, x_n$ son n variables que no aparecen en la cláusula dada.

[5] Aún podemos generalizar más el concepto con los *programas generales* cuyas cláusulas son del tipo

$$A_1, \ldots A_n \leftarrow L_1, \ldots, L_k$$

donde los A_i son átomos y los L_j literales cualesquiera.

2. Si $y_1, \ldots, y_d$ son las variables que existen en la cláusula original hacemos

$$p(x_1, \ldots, x_n) \leftarrow \exists y_1 \ldots \exists y_d((x_1 = t_1) \wedge \ldots \wedge (x_n = t_n) \wedge L_1 \wedge \ldots \wedge L_m)$$

3. Supongamos que ya hemos hecho esta transformación para toda cláusula que pertenezca a la definición de p. Obtenemos por tanto $k \geq 1$ fórmulas del tipo

$$\begin{array}{c} p(x_1, \ldots, x_n) \leftarrow E_1 \\ \vdots \\ p(x_1, \ldots, x_n) \leftarrow E_k \end{array}$$

donde cada E_i tiene la forma genérica

$$\exists y_1 \ldots \exists y_d((x_1 = t_1) \wedge \ldots \wedge (x_n = t_n) \wedge L_1 \wedge \ldots \wedge L_m)$$

Por último se define la *definición completada* de p como la fórmula

$$\forall x_1 \ldots \forall x_n(p(x_1, \ldots x_n) \leftrightarrow E_1 \vee \ldots \vee E_k)$$

4.3.6 Nota. Puede ocurrir que alguno de los símbolos de predicado que ocurran en el programa nunca aparezcan en la cabeza de una cláusula. Para cada uno de esos símbolos de predicado q añadimos explícitamente la cláusula

$$\forall x_1 \ldots \forall x_n \neg q(x_1, \ldots, x_n)$$

que es la definición de q dada de forma implícita en el programa . También llamaremos a esta fórmula la *definición completada de q.*

4.3.7 Ejemplo. La definición completada del predicado *alumno* del ejemplo que vimos al principio del capítulo es

$$\forall x(alumno(x) \leftrightarrow (x = Esther) \vee (x = Juan) \vee (x = Rosa))$$

4.4 La teoría de igualdad

En la definición de completación de un programa normal hemos introducido un nuevo símbolo de predicado $=$ que no aparecía en el programa. Evidentemente queremos que la interpretación del símbolo sea la relación identidad, pero tenemos que dar unas reglas para su uso desde un punto de vista sintáctico. Llamaremos a los siguientes axiomas *axiomas de la teoría de la igualdad.*(Usaremos la notación$\neq$ en lugar de $\neg =$):

1. $\forall(f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m))$ para todo par de símbolos de función distintos f y g.
2. $\forall(f(x_1, \ldots, x_n) \neq c$ para todo símbolo de función f y toda constante c.
3. $\forall(t(x) \neq x)$ para todo término $t(x)$ es que ocurra x y sea distinto de x.
4. $\forall((x_1 \neq y_1) \vee \ldots \vee (x_n \neq y_n) \rightarrow f(x_1, \ldots, x_n) \neq f(y_1, \ldots, y_n))$ para todo símbolo de función f.
5. $\forall(x = x)$.
6. $\forall((x_1 = y_1) \wedge \ldots \wedge (x_n = y_n) \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$ para todo símbolo de función f.
7. $\forall((x_1 = y_1) \wedge \ldots \wedge (x_n = y_n) \rightarrow (p(x_1, \ldots, x_n) \rightarrow p(y_1, \ldots, y_n))$ para todo símbolo de predicado p, incluido $=$.

4.4.1 Definición. Sea P un programa normal. Denominaremos *completación de* P, $comp(P)$, al conjunto de fórmulas formado por las definiciones completadas de los símbolos de predicado que ocurren en P junto con los axiomas de la teoría de igualdad.

4.4.2 Definición. Sea P un programa normal y G el objetivo normal. Una *respuesta* para $P \cup \{G\}$ es una sustitución θ tal que $Dom(\theta) \subseteq V(G)$.

4.4.3 Definición. Sea P un programa normal, G un objetivo normal $\leftarrow L_1, \ldots, L_k$ y θ una respuesta para $P \cup \{G\}$. Decimos que θ es una *respuesta correcta* para $comp(P) \cup \{G\}$ si $\forall((L_1 \wedge \ldots \wedge L_k)\theta)$ es consecuencia lógica de $comp(P)$.

Veremos que estas definiciones generalizan las definiciones dadas anteriormente de respuesta y respuesta correcta. Para ello el primer resultado que vemos es el siguiente:

4.4.4 Proposición. *Sea P un programa normal. Entonces P es consecuencia lógica de $comp(P)$.*

Demostración:
Para probarlo tenemos que ver que todo modelo de $comp(P)$ es modelo de P. Sea $\mathbf{M}$ un modelo de $comp(P)$ y sea $p(t_1, \ldots, t_n) \leftarrow L_1, \ldots, L_m$ una cláusula de P. Sean $y_1, \ldots, y_n$ las variables que ocurren en la cláusula y supongamos una asignación σ de esas variables tal que

$$\mathbf{M} \models \sigma(L_1 \wedge \ldots \wedge L_m)$$

Consideremos la definición completada de P

$$\forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \leftrightarrow E_1 \vee \ldots \vee E_k)$$

y supongamos que E_i es

$$\exists y_1 \ldots \exists y_d ((x_1 = t_1) \wedge \ldots \wedge (x_n = t_n) \wedge L_1 \wedge \ldots \wedge L_m)$$

Si modificamos σ de forma que a cada x_i le hace corresponder el término $\sigma(t_i)$ tenemos

$$\mathbf{M} \models \sigma(L_1 \wedge \ldots \wedge L_m)$$

que junto con el axioma 6 de la teoría de igualdad hace que $\mathbf{M} \models E_i$. De ahí que $\mathbf{M} \models p(t_1, \ldots, t_n)$ como queríamos probar. □

Vamos a generalizar la definición de operador consecuencia inmediata a programas normales.

4.4.5 Definición. Sea J una preinterpretación de un programa normal P y sea I una interpretación basada en J. Entonces

$T_P^J(I) = \{A_{J,\sigma} : A \leftarrow L_1 \wedge \ldots \wedge L_n \in P, \quad V$ es una asignación de variables de J y $I \models \sigma(L_1 \wedge \ldots \wedge L_n)\}$

Si J es la preinterpretación de P, escribiremos T_P en lugar de T_P^J.

4.4.6 Nota. T_P^J no es en general monótona, como se ve si consideramos el siguiente programa P:

$$p \leftarrow \neg p$$

No obstante, si P es un programa definido T_P^J es monótona.

4.4.7 Proposición. *Sea P un programa normal, J una preinterpretacion de $\mathbf{L}_P$, e I una interpretación basada en J. Entonces I es modelo de P si y sólo si $T_P^J(I) \subseteq I$.*

Demostración:
I es modelo de P si para toda instancia cerrada $A \leftarrow L_1, \ldots, L_n$ de una cláusula de P tal que $I \models L_1 \wedge \ldots \wedge L_n$ tenemos $I \models A$. Pero esto es lo mismo que decir que si $A \in T_P^J(I)$ entonces $A \in I$, que es lo que queríamos probar. □

4.4.8 Proposición. *Sea P un programa normal, J una preinterpretacion de P, e I una interpretación basada en J. Supongamos que I junto con la relación identidad asignada a $=$ es un modelo de la teoría de igualdad. Entonces I junto con la relación identidad asignada a $=$, es un modelo de $comp(P)$ si y sólo si $T_P^J(I) = I$.*

Demostración:
Supongamos primero que $T_P^J(I) = I$. Puesto que asumimos que I, junto con la relación identidad asignada a $=$ es modelo de la teoría de igualdad, basta probar que esta interpretación es modelo de cada definición completada de $comp(P)$. Consideremos una definición del tipo

$$\forall x_1 \ldots \forall x_n \neg q(x_1, \ldots, x_n)$$

puesto que I es punto fijo, se tiene que esta interpretación es modelo de la fórmula. Consideremos ahora una definición completada de la forma

$$\forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \leftrightarrow E_1 \vee \ldots \vee E_k)$$

Puesto que $T_P^J(I) \subseteq I$ tenemos que la interpretación es modelo de la fórmula

$$\forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \leftarrow E_1 \vee \ldots \vee E_k)$$

Pero también tenemos que $T_P^J(I) \supseteq I$ de donde tenemos que esa interpretación también es modelo de la fórmula

$$\forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \rightarrow E_1 \vee \ldots \vee E_k)$$

Supongamos ahora que I junto con la relación identidad asignada a $=$ es modelo de la completación. Entonces usando el hecho de que la interpretación es modelo de las fórmulas del tipo

$$\forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \leftarrow E_1 \vee \ldots \vee E_k)$$

tenemos que $T_P^J(I) \subseteq I$. Análogamente, puesto que la interpretación es modelo de

$$\forall x_1 \ldots \forall x_n (p(x_1, \ldots, x_n) \rightarrow E_1 \vee \ldots \vee E_k)$$

tenemos que $T_P^J(I) \supseteq I$. □

4.4.9 Proposición. *Sea P un programa definido y $A \in B_P$. Entonces $A \in Mpf(T_P)$ si y sólo si $comp(P) \cup \{A\}$ tiene un modelo de Herbrand.*

Demostración:
($\Rightarrow$) Supongamos que $A \in Mpf(T_P)$. Por la proposición anterior $Mpf(T_P) \cup \{s = s : s \in U_P\}$ es modelo de Herbrand de $comp(P) \cup \{A\}$.
($\Leftarrow$) Supongamos ahora que $comp(P) \cup \{A\}$ tiene un modelo de Herbrand $\mathbf{M}$. Puesto que $\mathbf{M} \models comp(P)$, el símbolo de predicado $=$ tiene asignada la relación identidad en $\mathbf{M}$. Por tanto $\mathbf{M}$ tiene la forma $\mathbf{M} = I \cup \{s = s : s \in U_P\}$ para alguna interpretación de Herbrand I de P. Luego, por la proposición anterior $I = T_P(I)$ y $A \in I$. De ahí que $A \in I \subseteq Mpf(T_P)$. □

4.4.10 Proposición. *Sea P un programa definido y sean $A_1, \ldots, A_m$ átomos. Si $\forall(A_1 \wedge \ldots \wedge A_m)$ es consecuencia lógica de comp(P) entonces también lo es de P.*

Demostración:
Sean $x_1, \ldots, x_k$ las variables que ocurren en $A_1 \wedge \ldots \wedge A_m$. Vamos a probar que $\forall x_1 \ldots \forall x_k (A_1 \wedge \ldots \wedge A_m)$ es consecuencia lógica de P, esto es, que $P \cup \{\neg \forall x_1 \ldots \forall x_k (A_1 \wedge \ldots \wedge A_m)\}$ es insatisfacible, o lo que es lo mismo, que

$$S = P \cup \{\neg A'_1 \vee \ldots \vee \neg A'_m\}$$

es insatisfacible, donde A'_i es A_i con las variables $x_1, \ldots, x_k$ reemplazadas por constantes de Skolem apropiadas.

Puesto que S está en forma clausal podemos restringir nuestra atención a interpretaciones de Herbrand. Sea I una interpretación de Herbrand de S. Podemos considerar I como una interpretación de P[6]. Supongamos que I es modelo de P y consideremos la preinterpretación J obtenida de I ignorando la asignación que esta interpretación hace a los símbolos de predicado del lenguaje. Por la proposición 4.4.7 tenemos que $T^J_P(I) \subseteq I$. Puesto que T^J_P es monótona, por la proposición 1.3.6 existe un punto fijo I' de T^J_P tal que $I' \subseteq I$. Puesto que I' junto con la relación identidad asignada a $=$ es obviamente un modelo de la teoría de igualdad, por la proposición esta interpretación es modelo de *comp*(P). De ahí que $\neg A'_1 \vee \ldots \vee \neg A'_m$ sea falsa en esta interpretación y puesto que $I' \subseteq I$, tenemos que $\neg A'_1 \vee \ldots \vee \neg A'_m$ es falso en I. Luego S es insatisfacible. $\square$

Nótese que combinando las proposiciones 4.4.4 y 4.4.10 tenemos que la información positiva que podemos deducir de *comp*(P) es exactamente la misma que podemos deducir de P. Es decir, hemos obtenido el siguiente resultado:

4.4.11 Teorema. *Sea P un programa definido, G un objetivo definido y θ una respuesta para $P \cup \{G\}$. Entonces θ es una respuesta correcta para comp$(P) \cup \{G\}$ si y sólo si lo es para $P \cup \{G\}$.*

4.5 SLDNF-resolución

Acabamos de definir el concepto declarativo de la completación de un programa. Veamos ahora su contrapartida procedural, esto es, cómo podemos llevarlo a la práctica para poder deducir información negativa. La idea básica es usar la SLD-resolución aumentada con la regla de fallo finito, esto es la *SLDNF- resolución.*

[6] Nótese que I no es necesariamente una interpretación *de Herbrand* de P

4.5.1 Nota. Nuestra intención es probar la adecuación de la SLDNF-resolución una vez la hallamos definido formalmente. Para ello necesitamos alguna regla en el proceso de selección de los átomos que nos garantice esa adecuación. La selección se realizará del siguiente modo:

No hay restricción para la elección de un literal positivo, en cambio sólo podremos seleccionar literales negativos cerrados.

Llamaremos a esta condición *condición de seguridad* (safeness condition)[7].

4.5.2 Definición. Sea G el objetivo $\leftarrow L_1, \ldots, L_m, \ldots, L_p$ y C la cláusula $A \leftarrow M_1, \ldots M_q$. Se dice que el objetivo G' es *derivado* de G y C usando el umg θ si se verifican las siguientes condiciones:

1. L_m es un átomo de G, llamado el átomo *seleccionado*.
2. θ es un umg de L_m y A.
3. G' es el objetivo normal $\leftarrow (L_1, \ldots, L_{m-1}, M_1, \ldots, M_q, L_{m+1}, \ldots, L_p)\theta$.

4.5.3 Definición. Sea P un programa normal y G un objetivo normal. Una *SLD-refutación de rango 0* de $P \cup \{G\}$ consiste en una sucesión

$$G_0 = G, G_1, \ldots, G_n = \Box$$

de objetivos normales, una sucesión $C_1, \ldots, C_n$ de variantes de cláusula de programa de P y una sucesión de $\theta_1 \ldots, \theta_n$ de umg tales que cada G_{i+1} se deriva de G_i y C_{i+1} usando θ_{i+1}.

4.5.4 Definición. Sea P un programa normal y sea G un objetivo normal. Un *SLDNF-árbol finitamente fallido de rango 0* para $P \cup \{G\}$ es un árbol satisfaciendo las siguientes condiciones:

1. El árbol es finito y cada nodo del árbol es un objetivo normal distinto de la cláusula vacía.
2. El nodo raíz es G.
3. Los literales seleccionados son siempre positivos.

[7] Esta condición es muy fuerte. Lloyd comenta en [37] cómo debilitarla.

4. Sea $\leftarrow L_1, \ldots, L_m, \ldots, L_p$ un nodo del árbol que no sea hoja y supongamos que L_m es el átomo seleccionado. Entonces, para cada cláusula de programa (o variante) $A \leftarrow M_1, \ldots, M_q$ tal que L_m y A son unificables mediante el umg θ el nodo tiene un hijo

$$\leftarrow (L_1, \ldots, L_{m-1}, M_1, \ldots, M_q, L_{m+1}, \ldots, L_p)\theta$$

5. Sea $\leftarrow L_1, \ldots, L_m, \ldots, L_p$ un nodo del árbol que no sea hoja y supongamos que L_m es el átomo seleccionado. Entonces no hay ninguna cláusula en P (o variante) tal que su cabeza unifique con L_m.

4.5.5 Definición. Sea P un programa normal y G un objetivo normal. Una *SLD-refutación de rango k+1* de $P \cup \{G\}$ consiste en una sucesión $G_0 = G, G_1, \ldots, G_n = \Box$ de objetivos normales, una sucesión $C_1, \ldots, C_n$ compuesta por variantes de cláusula de programa de P y/o literales negativos cerrados y una sucesión de $\theta_1 \ldots, \theta_n$ de sustituciones tales que para cada i

- O bien cada G_{i+1} se deriva de G_i y C_{i+1} usando θ_{i+1},
- O bien G_i es $\leftarrow L_1, \ldots, L_m, \ldots, L_p$, el literal seleccionado L_m de G_i es un literal negativo cerrado $\neg A_m$ y existe un SLDNF-árbol finitamente fallido de rango k para $P \cup \{\leftarrow A_m\}$. En este caso G_{i+1} es $\leftarrow L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_p$, θ_{i+1} es la sustitución identidad y C_{i+1} es $\neg A_m$.

4.5.6 Definición. Sea P un programa normal y sea G un objetivo normal. Un *SLDNF-árbol finitamente fallido de rango k+1* para $P \cup \{G\}$ es un árbol satisfaciendo las siguientes condiciones:

1. El árbol es finito y cada nodo del árbol es un objetivo normal distinto de la cláusula vacía.
2. El nodo raíz es G.
3. Sea $\leftarrow L_1, \ldots, L_m, \ldots, L_p$ un nodo del árbol que no sea hoja y supongamos que L_m es el literal seleccionado. Entonces,
   - O bien L_m es un átomo y para cada cláusula de programa (o variante) $A \leftarrow M_1, \ldots, M_q$ tal que L_m y A son unificables mediante el umg θ el nodo tiene un hijo

$$\leftarrow (L_1, \ldots, L_{m-1}, M_1, \ldots, M_q, L_{m+1}, \ldots, L_p)\theta$$

- O bien L_m es un literal negativo cerrado $\neg A_m$ y existe un SLDNF-árbol finitamente fallido de rango k para $P \cup \{\leftarrow A_m\}$, en cuyo caso el único hijo es $\leftarrow L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_p$.

4. Sea $\leftarrow L_1, \ldots, L_m, \ldots, L_p$ un nodo del árbol que no sea hoja y supongamos que L_m es el literal seleccionado. Entonces

   - O bien L_m es un átomo y no hay ninguna cláusula en P (o variante) tal que su cabeza unifique con L_m.
   - O bien L_m es un literal negativo cerrado $\neg A_m$ y existe una SLDNF-refutación de rango k de $P \cup \leftarrow A_m$.

4.5.7 Definición. Sea P un programa normal y G un objetivo normal. Una *SLDNF-refutación* de $P \cup \{G\}$ es una SLDNF- refutación de $P \cup \{G\}$ de rango k, para algún k.

4.5.8 Definición. Sea P un programa normal y G un objetivo normal. Un *SLDNF-árbol finitamente fallido* de $P \cup \{G\}$ es un SLDNF-árbol finitamente fallido de $P \cup \{G\}$ de rango k, para algún k.

4.5.9 Definición. Sea P un programa normal y G un objetivo normal. Una *respuesta computada* θ de $P \cup \{G\}$ es la sustitución obtenida restringiendo la composición de sustituciones $\theta_1 \ldots \theta_n$ a las variables de G, donde $\theta_1 \ldots \theta_n$ es la sucesión de sustituciones obtenida en una SLDNF-refutación de $P \cup \{G\}$.

4.5.10 Definición. Sea P un programa normal y G un objetivo normal. Una *SLDNF-derivación* de $P \cup \{G\}$ es una sucesión (finita o infinita) de objetivos normales $G = G_0, G_1, G_2 \ldots$, una sucesión $C_1, C_2 \ldots$ de variantes de cláusulas del programa P (llamadas *cláusulas de entrada* o literales negativos cerrados y una sucesión $\theta_1, \theta_2, \ldots$ de sustituciones satisfaciendo las siguientes condiciones:

1. Para cada i,

   - O bien G_{i+1} se deriva de G_i y la cláusula de entrada C_{i+1} usando θ_{i+1}.
   - O bien G_i es $\leftarrow L_1, \ldots, L_m, \ldots, L_p$ el literal seleccionado en L_m en G_i es un literal negativo cerrado $\neg A$ y existe un SLDNF-árbol finitamente fallido de $P \cup \{\leftarrow A_m\}$. En este caso G_{i+1} es

     $$\leftarrow L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_p$$

     θ_{i+1} es la sustitución identidad y C_{i+1} es $\neg A_m$.

2. Si la sucesión $G = G_0, G_1, G_2 \ldots$ es finita, entonces

   - O bien el último objetivo es la cláusula vacía
   - O bien el último objetivo es $\leftarrow L_1, \ldots, L_m, \ldots, L_p$, el literal seleccionado es L_m y es positivo y no hay ninguna cláusula de P (o variante) cuya cabeza unifique con L_m.
   - O bien el último objetivo es $\leftarrow L_1, \ldots, L_m, \ldots, L_p$, el literal seleccionado es L_m y es un literal negativo cerrado $\neg A_m$ y existe una SLDNF-refutación de $P \cup \{\leftarrow A_m\}$.

4.5.11 Definición. Sea P un programa normal y G un objetivo normal. Un *SLDNF-árbol* de $P \cup \{G\}$ es un árbol que satisface lo siguiente:

1. Cada nodo del árbol es un objetivo normal (puede ser la cláusula vacía).
2. El nodo raíz es G.
3. Sea $\leftarrow L_1, \ldots, L_m, \ldots, L_P$ $(p \geq 1)$ un nodo que no sea hoja en el que hemos seleccionado el literal L_m.

   (a) Si L_m es un átomo, entonces para cada cláusula de programa (o variante suya) $A \leftarrow M_1, \ldots, M_q$ tal que A y L_m sean unificables con el umg θ, el nodo tiene como hijo la cláusula

   $$\leftarrow (L_1, \ldots, L_{m-1}, M_1, \ldots, M_q, L_{m+1}, \ldots, L_p)\theta$$

   (b) Si L_m es un literal negativo cerrado $\neg A_m$ y existe un SLDNF-árbol finitamente fallido de $P \cup \{\leftarrow A_m\}$, entonces el único sucesor es

   $$\leftarrow (L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_p)\theta$$

4. Sea $\leftarrow L_1, \ldots, L_m, \ldots, L_P$ $(p \geq 1)$ un nodo hoja en el que hemos seleccionado el literal L_m. Entonces

   (a) O bien L_m es un átomo que no unifica con ninguna cabeza de cláusula (o variante de cláusula) de P.

   (b) O bien L_m es un literal negativo $\neg A$ y existe una SLDNF- refutación de $P \cup \{\leftarrow A_m\}$.

5. Los nodos que son cláusulas vacías no tienen sucesores.

4.5.12 Definición.

- Una SLDNF-derivación es *finita* si consta de una sucesión finita de objetivos, en otro caso se dice *infinita*.

- Una SLDNF-derivación *tiene éxito* si es finita y su último objetivo es la cláusula vacía.

- Una SLDNF-derivación *falla* si es finita y su última cláusula no es la cláusula vacía.

4.5.13 Definición. Sea P un objetivo normal y G un objetivo normal. Decimos que una computación de $P \cup \{G\}$ *es inviable*[8] si en algún punto de la computación alcanzamos un objetivo que sólo contenga literales negativos no cerrados.[9]

4.5.14 Ejemplo. Si G es el objetivo $\leftarrow \neg p(x)$ y P es un programa normal la computación de $P \cup \{G\}$ es inviable.

4.6 Adecuación y completitud de la SLDNF-resolución

Nuestro objetivo ahora es probar la adecuación de la SLDNF- resolución. Para ello necesitamos dos lemas previos debidos a Clark. Su demostración puede encontrarse en [10].

4.6.1 Lema. *Sean $p(s_1, \ldots, s_n)$ y $p(t_1, \ldots, t_n)$ dos átomos.*

1. *Si $p(s_1, \ldots, s_n)$ y $p(t_1, \ldots, t_n)$ no son unificables, entonces*

$$\neg\exists((s_1 = t_1) \wedge \ldots \wedge (s_n = t_n))$$

   es consecuencia lógica de la teoría de igualdad.

2. *Si $p(s_1, \ldots, s_n)$ y $p(t_1, \ldots, t_n)$ son unificables con el umg $\theta = \{x_1/r_1, \ldots, x_k/r_k\}$ dado por el algoritmo de unificación entonces*

$$\forall((s_1 = t_1) \wedge \ldots \wedge (s_n = t_n) \leftrightarrow (x_1 = r_1) \wedge \ldots \wedge (x_n = r_n))$$

   es consecuencia lógica de la teoría de igualdad.

[8] *Floundered* en el texto original

[9] En [37] podemos encontrar distintos resultados relacionados con la inviabilidad

4.6.2 Lema. *Sea P un programa normal y G un objetivo normal. Supongamos que el literal seleccionado en G es positivo.*

1. *Si no tiene objetivos derivados, entonces G es consecuencia lógica de $comp(P)$.*

2. *Si el conjunto $\{G_1, \ldots, G_r\}$ de objetivos derivados es no vacío entonces*

$$G \leftrightarrow G_1 \wedge \ldots \wedge G_r$$

es consecuencia lógica de $comp(P)$.

Teorema. 4.6.3 (Adecuación de la negación como regla de fallo)
Sea P un programa normal y G un objetivo normal. Si $P \cup \{G\}$ tiene un SLDNF-árbol finitamente fallido, entonces G es consecuencia lógica de la completación de P, $comp(P)$.

Demostración:
La demostración es por inducción sobre el rango k del SLDNF- árbol finitamente fallido de $P \cup \{G\}$. Sea G el objetivo $\leftarrow L_1, \ldots, L_n$.

($k = 0$) El resultado se tiene directamente del lema 4.6.2.

($k \to k + 1$) Supongamos ahora el resultado cierto para SLDNF-árboles finitamente fallidos de rango k y consideremos un SLDNF-árbol de $P \cup \{G\}$ de rango $k + 1$. La demostración de que en este caso también G es consecuencia lógica de $comp(P)$ la vamos a hacer por inducción sobre la profundidad de ese árbol, inducción que llamaremos secundaria.

($d = 1$) Supongamos primero que la profundidad del árbol es 1.

- Si el literal seleccionado es positivo, por el lema 4.6.2, primer apartado, se tiene el resultado.

- Si el literal seleccionado en G, L_i es el literal cerrado negativo $\neg A_i$, puesto que la profundidad es 1, existe una SLDNF-refutación de rango k de $P \cup \{\leftarrow A_i\}$[10]. Así, usando la proposición 4.4.4 y aplicando la hipótesis de inducción para árboles finitamente fallidos de profundidad $k - 1$ obtenemos que A_i es consecuencia lógica de $comp(P)$. Por último, puesto que A_i es cerrado, deducimos que $\neg\exists(L_1 \wedge \ldots \wedge L_n)$ es consecuencia lógica de $comp(P)$.

[10]Nótese que para un objetivo cuyo literal seleccionado sea positivo, el objetivo derivado es consecuencia lógica del objetivo dado y la cláusula de entrada.

$(d \to d+1)$ Supongamos ahora que el SLDNF-árbol de $P \cup \{G\}$ que estamos considerando tiene profundidad $d+1$.

- Si el literal seleccionado en G es positivo, el resultado se tiene por el segundo apartado del lema 4.6.2 y la hipótesis de inducción secundaria.
- Supongamos que el seleccionado en G es el literal cerrado negativo L_i. Aplicando la hipótesis de inducción secundaria obtenemos que

$$\neg\exists(L_1 \wedge \ldots \wedge L_{i-1} \wedge L_{i+1} \wedge \ldots \wedge L_n)$$

  es consecuencia lógica de $comp(P)$. De ahí que $\neg\exists(L_1 \wedge \ldots \wedge L_n)$ también sea consecuencia de $comp(P)$.

$\square$

4.6.4 Corolario. *Sea P un programa definido. Si $A \in F_P$ entonces $\neg A$ es consecuencia lógica de $comp(P)$*

Demostración:
Sea $A \in F_P$. Por el teorema 4.2.13 todo SLD-árbol favorable de $P \cup \{\leftarrow A\}$ falla de forma finita. En particular existe un SLD-árbol de $P \cup \{\leftarrow A\}$ que falla de forma finita. Por tanto, puesto que los SLD-árboles son casos particulares de SLDNF-árboles, tenemos que existe un SLDNF- árbol de $P \cup \{\leftarrow A\}$ que falla de forma finita. De ahí que $comp(P) \models \neg A$. $\square$

El siguiente resultado se debe esencialmente a Clark [10].

Teorema. 4.6.5 (Adecuación de la SLDNF-resolución) *Sea P un programa normal y G un objetivo normal. Entonces toda respuesta computada de $P \cup \{G\}$ es una respuesta correcta de $comp(P) \cup \{G\}$.*

Demostración:
Sea G el objetivo normal $\leftarrow L_1, \ldots, L_k$ y $\theta_1, \ldots, \theta_n$ la sucesión de sustituciones usada en una SLDNF-refutación de $P \cup \{G\}$. Tenemos que probar que $\forall((L_1 \wedge \ldots \wedge L_k)\theta_1, \ldots, \theta_n)$ es consecuencia lógica de $comp(P)$. Probamos el resultado por inducción sobre la longitud de la SLDNF-refutación.
$(n = 1)$ En este caso G es de la forma $\leftarrow L_1$. Podemos considerar dos posibilidades:

- L_1 *es positivo*. Entonces P tiene una cláusula unidad de la forma $A \leftarrow$ y $L_1\theta_1 = A\theta_1$. Puesto que $L_1\theta_1 \leftarrow$ es una instancia de una cláusula unidad de P, tenemos que $\forall(L_1\theta_1)$ es consecuencia lógica de P y de ahí que lo sea de $comp(P)$.

- L_1 *es negativo.* En este caso, L_1 es cerrado, θ_1 es la sustitución identidad y por el teorema 4.6.3 L_1 es consecuencia lógica de $comp(P)$.

$(n-1 \to n)$ Supongamos ahora que que el resultado se tiene para para respuestas computadas obtenidas con SLDNF- refutaciones de longitud $n-1$. Sea $\theta_1 \dots \theta_n$ la sucesión de sustituciones obtenida en una SLDNF-refutación de $P \cup \{G\}$ de longitud n. Sea L_m el literal seleccionado en G. De nuevo podemos considerar dos casos:

- L_1 *es positivo.* Sea $A \leftarrow M_1, \dots, M_q$ $(q \geq 0)$ la primera cláusula de entrada. Por hipótesis de inducción,

  $$\forall((L_1 \wedge \dots \wedge L_{m-1} \wedge M_1 \wedge \dots \wedge M_q \wedge L_{m+1} \wedge \dots \wedge L_k)\theta_1 \dots \theta_n)$$

  es consecuencia lógica de $comp(P)$. Por tanto, si $q > 0$, $\forall((M_1 \wedge \dots \wedge M_q)\theta_1, \dots, \theta_n)$ es consecuencia lógica de $comp(P)$ y en consecuencia $\forall(L_m\theta_1 \dots \theta_n)$, que es lo mismo que $\forall(A\theta_1 \dots \theta_n)$ es consecuencia logica de $comp(P)$. De ahí que $\forall((L_1 \wedge \dots \wedge L_k)\theta_1 \dots \theta_n)$ sea consecuencia lógica de $comp(P)$.

- L_1 *es negativo.* En este caso, L_m es cerrado, θ_1 es la sustitución identidad y por el teorema 4.6.3 L_m es consecuencia lógica de $comp(P)$. Usando la hipótesis de inducción se obtiene directamente que $\forall((L_1 \wedge \dots \wedge L_k)\theta_1 \dots \theta_n)$ es consecuencia lógica de $comp(P)$.

□

El siguiente resultado se debe a Jaffar, Lassez y Lloyd [29].

Teorema. 4.6.6 (Completitud de la negación como regla de fallo)
Sea P un programa definido y G un objetivo definido. Si G es consecuencia lógica de $comp(P)$, entonces cada SLD-árbol favorable es finitamente fallido.

Demostración:
Sea G el objetivo $\leftarrow A_1, \dots, A_q$. Supongamos que $P \cup \{G\}$ tiene un SLD-árbol favorable que no falla de forma finita. Vamos a probar que $comp(P) \cup \{\exists(A_1 \wedge \dots \wedge A_q)\}$ tiene un modelo.
Sea BR una rama no fallida en el SLD-árbol favorable de $P \cup \{G\}$. Supongamos que BR es $G_0 = G, G_1, \dots$ con los umg $\theta_1, \theta_2, \dots$ y las cláusulas de entrada $C_1, C_2, \dots$. El primer paso es usar BR para definir una preinterpretación J de P.
Supongamos que $\mathbf{L}$ es el lenguaje de primer orden subyacente en P. Evidentemente suponemos que $\mathbf{L}$ es lo bastante rico como para edmitir todas las

separaciones de variables necesarias en BR. Definimos la relación $*$ sobre el conjunto de los términos de $\mathbf{L}$ de la siguiente manera: Sean s y t téminos en $\mathbf{L}$. Entonces $s * t$ si existe $n \geq 1$ tal que $s\theta_1 \ldots \theta_n = t\theta_1 \ldots \theta_n$, esto es, si existe un $n \geq 1$ tal que $\theta_1 \ldots \theta_n$ unifica a s y t. Claramente se tiene que $*$ es relación de equivalencia. Vamos a definir el dominio D de nuestra preinterpretación J como el conjunto de todas las $*$-clases de equivalencia de términos de $\mathbf{L}$. Denotaremos la clase del término s como $[s]$.
Demos ahora una asignación a los símbolos de función de $\mathbf{L}$. Si c es una constante de $\mathbf{L}$ le asignamos la clase de equivalencia $[c]$. Si f es un símbolo de función n-ario, $(n \geq 1)$, le asignamos la aplicación de D^n en D definida como $([s_1],\ldots,[s_n]) \rightarrow [f(s_1,\ldots,s_n)]$. Trivialmente se comprueba que está bien definida. Esto completa la definición de J.
El siguiente paso es dar una asignación a los símbolos de predicado de forma que extendamos J a un modelo de $comp(P) \cup \{\exists(A_1 \wedge \ldots \wedge A_q)\}$. Primero definimos el conjunto I_0 como sigue:

$$I_0 = \{p([t_1],\ldots,[t_n]) : p(t_1,\ldots,t_n) \text{ aparezca en } BR\}$$

Vemos que $I_0 \subseteq T_P^J(I_0)$, donde T_P^J es la aplicación asociada a la preinterpretación J. Supongamos que $p([t_1],\ldots,[t_n]) \in I_0$ y que $p(t_1,\ldots,t_n)$ aparece en G_i, $i \in \omega$. Puesto que BR es favorable y no finita existe $j \in \omega$ tal que $p(s_1,\ldots,s_n) = p(t_1,\ldots,t_n)\theta_{i+1}\ldots\theta_{i+j}$ aparece en G_{i+j} y $p(s_1,\ldots,s_n)$ es el átomo seleccionado en G_{i+j}. Supongamos que C_{i+j+1} es $p(r_1,\ldots,r_n) \leftarrow B_1,\ldots,B_m$. Por definición de T_P^J tenemos que

$$p([r_1\theta_{i+j+1}],\ldots,[r_n\theta_{i+j+1}]) \in T_P^J(I_0)$$

Entonces, usando el hecho de que, para cada k, $\theta_1\ldots\theta_k$ puede ser tomada como idempotente, tenemos que:

$$\begin{array}{ll} & p([t_1],\ldots,[t_n]) \\ = & p([t_1\theta_{i+1}],\ldots,[t_n\theta_{i+1}]) \\ = & p([s_1],\ldots,[s_n]) \\ = & p([s_1\theta_{i+j+1}],\ldots,[s_n\theta_{i+j+1}]) \\ = & p([r_1\theta_{i+j+1}],\ldots,[r_n\theta_{i+j+1}]) \end{array}$$

Así que $p([t_1],\ldots,[t_n]) \in T_P^T(I_0)$, y por tanto $I_0 \subseteq T_P^J(I_0)$.
Aplicando ahora la proposición 1.3.6, existe I tal que $I_0 \subseteq I$ y $I = T_P^J(I)$. Esta interpretación I nos da la asignación buscada a los símbolos de predicado de $\mathbf{L}$. Asignamos también la relación identidad a $=$.
Esto completa la definición de la interpretación I, junto con la relación identidad asignada a $=$, para $comp(P) \cup \{\exists(A_1 \wedge \ldots \wedge A_q)\}$. Nótese que esta interpretación es un modelo de $\exists(A_1 \wedge \ldots \wedge A_q)\}$, puesto que $I_0 \subseteq I$. Nótese además que esta interpretación es claramente un modelo para la teoría de

igualdad. De ahí que, aplicando la proposición 4.4, tengamos que I junto con la relación identidad asignda a $=$ sea modelo de $comp(P) \cup \{\exists(A_1 \wedge \ldots \wedge A_q)\}$.
□

4.6.7 Nota. El modelo construido en esta demostración no es un modelo de Herbrand. De hecho el teorema no puede ser probado restringiendo nuestra atención a modelos de Herbrand[11].

4.6.8 Definición. Una regla de computación *segura* es una función del conjunto de objetivos normales, ninguno de los cuales consiste únicamente en literales negativos no cerrados en el conjunto de los literales tal que el valor de la función para tales objetivos sea o bien un literal positivo o bien un literal negativo cerrado llamado el *literal seleccionado* de ese objetivo.

4.6.9 Definición. Sea P un programa normal, G un objetivo normal y R una regla de computación segura. Adoptaremos las siguientes definiciones:

1. Una *SLDNF-derivación de* $P \cup \{G\}$ *vía* R es una SLDNF-derivación de $P \cup \{G\}$ en la que hemos usado la regla de computación R para seleccionar literales.

2. Un *SLDNF-árbol de* $P \cup \{G\}$ *vía* R es un SLDNF-árbol de $P \cup \{G\}$ en el que hemos usado la regla de computación R para seleccionar literales.

3. Una *SLDNF-refutación de* $P \cup \{G\}$ *vía* R es una SLDNF-refutación de $P \cup \{G\}$ en la que hemos usado la regla de computación R para seleccionar literales.

4. Una *respuesta R-computada* $P \cup \{G\}$ *vía* R es una respuesta computada de $P \cup \{G\}$ en la que hemos usado la regla de computación R para seleccionar literales.

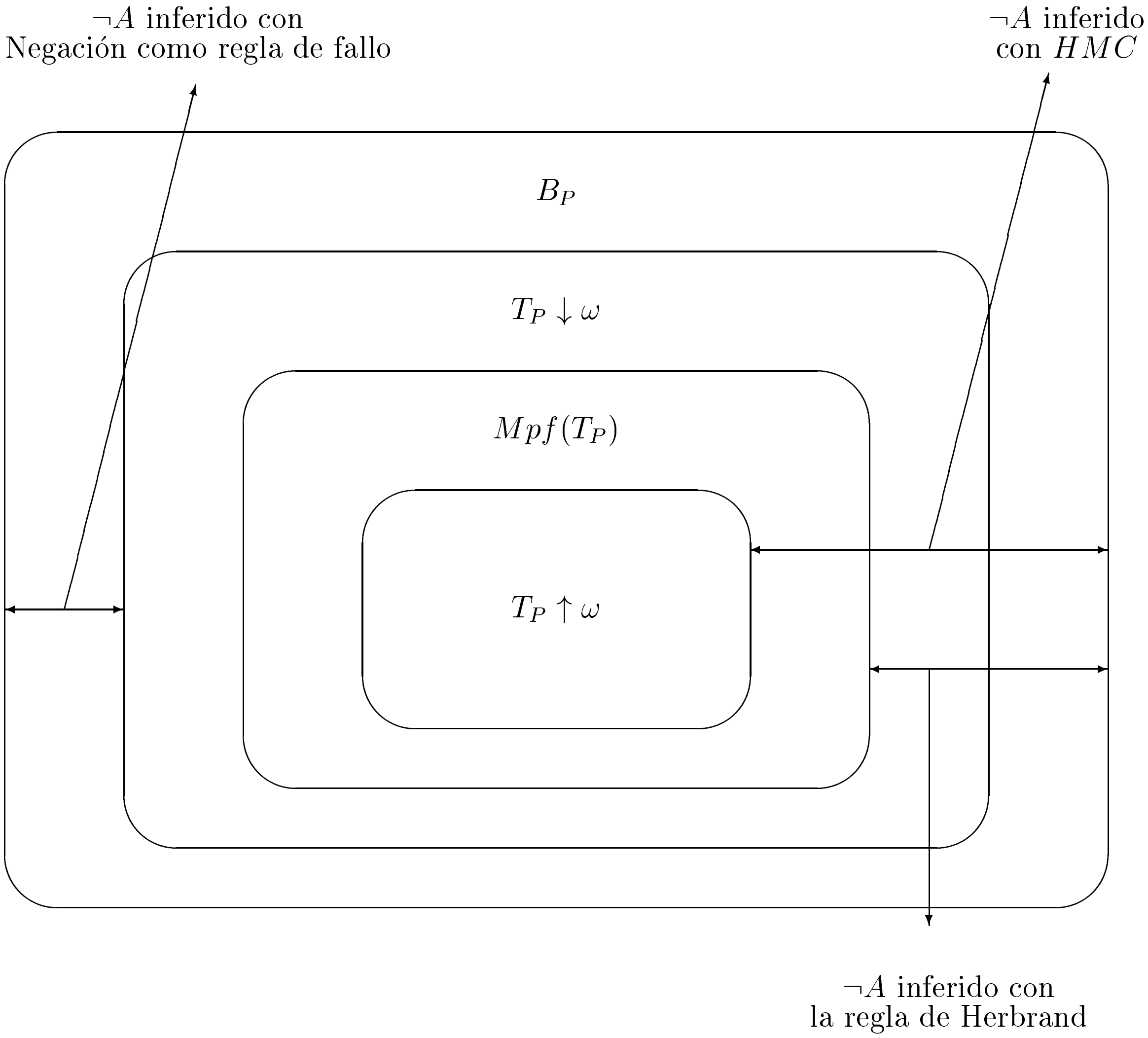
4.6.10 Nota. Existen numerosos estudios sobre cómo inferir información negativa, no obstante aún no está probada la completitud de forma general, lo que abre las puertas a una posterior investigación en este terreno.

4.7 La regla de Herbrand

Terminamos este capítulo comparando los distintos resultados que hemos presentado, pero antes veamos una nueva regla de inferencia.

[11] Ver Lloyd [37].

Definición. 4.7.1 (Regla de Herbrand) Sea P un programa definido y sea $A \in B_P$. Entonces si $comp(P) \cup A$ no tiene modelo de Herbrand, inferimos $\neg A$.



Tenemos ahora tres posibles reglas de inferencia para deducir información negativa. Si P es un programa definido se tiene que

- $\{A \in B_P : \neg A$ puede ser deducido con la negación como regla de fallo$\}= B_P \backslash T_P \downarrow \omega$
- $\{A \in B_P : \neg A$ puede ser deducido con la regla de Herbrad$\}=B_P \backslash Mpf(T_P)$
- $\{A \in B_P : \neg A$ puede ser deducido con la $HMC\}=B_P \backslash T_P \uparrow \omega$

Puesto que $T_P \uparrow \omega \subseteq Mpf(T_P) \subseteq T_P \downarrow \omega$, se tiene que la regla más fuerte es la HMC, seguida de la regla de Herbrand y por último, la negación como regla de fallo. Como en general $T_P \uparrow \omega$, $Mpf(T_P)$ y $T_P \downarrow \omega$ son distintos, las reglas son diferentes.

4.8 Resultados recientes

La noción de SLD-resolución presentada por Kowalski [30] en 1974 nos da un método que nos permite deducir información negativa, esto es, nos permite estudiar si un literal positivo es o no consecuencia lógica de un programa. Años más tarde, en 1979 Clark [11] propuso ampliar esta regla con la *negación de fallo finito* y crear así la $SLDNF$- resolución. La idea de la $SLDNF$-resolución es muy simple y ya ha sido presentada en esta memoria. Dado un átomo cerrado A,

$$\begin{array}{rcl} \neg A \text{ tiene éxito} & \Longleftrightarrow & A \text{ falla de forma finita} \\ \neg A \text{ falla de forma finita} & \Longleftrightarrow & A \text{ tiene éxito} \end{array}$$

El problema es que "tener éxito" y "fallar de forma finita" no son las únicas posibilidades de una derivación. Así, por ejemplo, dado el programa $P = \{A \leftarrow A\}$ y el objetivo normal G, $\leftarrow \neg A$, el $SLDNF$-árbol de $P \cup \{G\}$ ni tiene éxito ni falla de forma finita.

$$\begin{tikzcd} \leftarrow \neg A \arrow[dr] & \\ & \leftarrow A \arrow[d] \\ & \leftarrow A \arrow[d] \\ & \leftarrow A \\ & \vdots \end{tikzcd}$$

Este ejemplo pone de manifiesto la existencia de una tercera posibilidad: Existen SLD- árboles que ni tienen éxito ni fallan de forma finita *con las definiciones hasta ahora aceptadas.* Este problema ya es mencionado en 1991 en los trabajos de Apt y Bezem [4] y Apt y Pedreschi [7]. El problema fue tratado por Martelli y Tricomi en 1992 [39] quienes propusieron una revisión de

la definición original en la que los árboles subsidiarios utilizados en la resolución de literales negativos se construían "dentro" del árbol principal. Estos autores utilizaban fórmulas más complicadas que los objetivos normales y la construcción era bastante artificiosa. En el último número de *The Journal of Logic Programming* K.R. Apt y K. Doets [5] han publicado un artículo titulado *A new definition of SLDNF-resolución* en el que presentan una construcción alternativa.
En la solución que presentan, como en la definición original, los árboles subsidiarios para la resolución de literales negativos se construyen "aparte", pero no se toman como una única arista dentro del árbol principal, sino que *contruyen sus ramas en paralelo*, como un subárbol del árbol principal. Así si alguno de esto árboles subsidiarios es infinito, el ábol principal *se considera infinito*[12]
Estudiemos ahora la nueva definición presentada por Apt y Doets.

4.8.1 Una nueva definición

4.8.1 Definición. Sea C una cláusula, L un literal positivo de C y P_i una cláusula de programa[13] normal $A \leftarrow E$.

1. Diremos que C *se resuelve a* D *vía* θ *con respecto a* Σ, o de otra forma, que el par (θ, D) es un *resolvente* de C respecto a Σ, (con notación $C \overset{\theta}{\to} D(\Sigma)$) si:
   - *O bien* $\Sigma = (L, P_i)$, θ es un umg de L y A y D es la cláusula obtenida sustituyendo en C el literal L por E y aplicando θ.
   - *O bien* Σ es un literal negativo de C, $\theta = \varepsilon$ y D es la cláusula obtenida a partir de C eliminando la ocurrencia del literal Σ.
2. Diremos que una cláusula es *aplicable* a un átomo si alguna variante de su cabeza unifica con el átomo.

4.8.2 Definición. Una sucesión (finita o infinita) $C_0 \overset{\theta_1}{\to} \dots C_n \overset{\theta_{n+1}}{\to} C_{n+1} \dots$ de pasos de resolución es una *pseudoderivación* si para cada paso:

- La cláusula de entrada empleada no contiene variables que ocurran en el objetivo inicial ni en ninguna de las demás cláusulas de entrada de los pasos anteriores ("Separación de variables").

[12] Veremos la construcción formal más adelante

[13] Apt y Doets presentan su definición para programas generales que nosotros no hemos tratado en esta memoria, por eso, en la medida de lo posible, adaptaremos estos resultados a programas normales.

- El umg empleado es relevante ("Relevancia").

Intuitivamente, según esta construcción, una SLDNF-derivación va a ser una pseudoderivación en la que *la resolución de la cláusula S a la cláusula D respecto Σ, cuando Σ sea un literal negativo, esté justificada por un SLDNF-árbol finitamente fallido.* A continuación vamos a considerar unos sistemas de árboles que Apt y Doets llaman *árboles complejos*[14].

4.8.3 Definición. Un *árbol complejo* es una terna $\mathbf{T}$=($\mathcal{T}$, T, *subs*) donde:

- $\mathcal{T}$ es un conjunto de árboles.
- T es un elemento de $\mathcal{T}$ que llamaremos *árbol principal.*
- *subs* es una función parcial definida sobre el conjunto de los nodos de los árboles de $\mathbf{T}$, tal que a algunos de estos nodos les asigna un árbol ("susbsidiario") de $\mathbf{T}$.

4.8.4 Definición. Definiremos un *camino* en $\mathbf{T}$ como una sucesión de nodos $N_1, \ldots, N_i, \ldots$ de árboles de $\mathbf{T}$ tal que para todo i, N_{i+1} es

- O bien un hijo de N_i en el correspondiente árbol de $\mathbf{T}$
- O bien la raíz del árbol $subs(N_i)$.

Así podemos considerar un árbol complejo como un tipo especial de grafo dirigido, en el que hay dos tipos de aristas: las "usuales", que unen dos nodos de un mismo árbol y las que unen un nodo con la raíz de un árbol subsidiario. Un SLDNF-árbol va ser un árbol complejo de un tipo especial, construido como límite de ciertos árboles complejos finitos: los *pre-SLDNF- árboles.*
En lo que sigue fijaremos un programa P.

4.8.5 Definición. Un *pre-SLDNF-árbol* (relativo a P) es un árbol complejo cuyos nodos son objetivos (posiblemente marcados) o literales (posiblemente marcados). La función *subs* asigna a los nodos que sean literales negativos cerrados $\neg A$ un árbol en $\mathbf{T}$ de raíz A. Definimos por inducción la clase de los pre-SLDNF- árboles.

- Para cada objetivo C, el árbol complejo $(\{C\}, C, \emptyset)$ es un pre-SLDNF-árbol (un pre-SLDNF-árbol *inicial*).

[14] Apt y Doets comentan en su artículo que les dan este nombre "a falta de uno mejor" *(for lack of a better name)*

- Si **T** es un pre-SLDNF-árbol, entonces cualquier *extensión* de **T** es un pre-SLDNF-árbol.

Definimos una *extensión* de **T** como el árbol complejo resultante de aplicar los siguientes pasos a todo objetivo distinto de la cláusula vacía que sea una hoja no marcada en algún árbol de $\mathcal{T}$.
Sea C por tanto una tal cláusula.

1. Primero tomamos el literal seleccionado de la cláusula C. Si no existiera ninguno con la marca *"seleccionado"* marcamos uno. Sea L el literal seleccionado de C.
   - Si L es positivo.
     - Si C no tiene resolventes respecto a L y alguna cláusula de P, entonces marcamos la cláusula C como *fallo*.
     - Si C tiene resolventes, para cada cláusula P_i de P aplicable a L, elegimos una resolvente (θ, D) de C respecto a L y P_i y la añadimos como inmediato sucesor de C en T. Estas resolventes de eligen de tal modo que todas las ramas de **T** sean pseudoderivaciones.
   - $L = \neg A$ es un literal negativo.
     - Si A no es cerrado, marcamos C como *inviable*.
     - Si A es cerrado
       * Si no está definido $subs(C)$, entonces añadimos un nuevo árbol $T' = (\{A\}, \emptyset)$[15] a **T** y definimos $subs(C) = T'$.
       * Si $subs(C)$ está definido y tiene éxito, marcamos C con la etiqueta *"fallo"*
       * Si $subs(C)$ está definido y falla de forma finita, entonces añadimos la resolvente de C $(\varepsilon, C - \{L\})$ como el único sucesor inmediato de C en T.
2. Además, marcamos todas las cláusula vacías como *"éxito"*

Nótese que si **T** no tiene hojas sin marcar, **T** es trivialmente una extensión suya, con lo que el proceso se estaciona.

Cada pre-SLDNF-árbol no es más que un árbol complejo, por lo que tiene sentido hablar de *inclusión* entre pre-SLDNF-árboles y de *límite* de una sucesión de pre-SLDNF-árboles encajados.

[15]Representamos un árbol como un par ordenado $T = (\mathcal{N}, \mathcal{A})$ donde $\mathcal{N}$ es el conjunto de los nodos y $\mathcal{A}$ el conjunto de las aristas. En este caso el nuevo árbol que añadimos a **T** no tiene aristas y tiene un único nodo: A

4.8.6 Definición.

- Un *SLDNF-árbol* es el límite de una sucesión $\mathbf{T}_0, \ldots, \mathbf{T}_n, \ldots$ tal que $\mathbf{T_0}$ es un pre-SLDNF-árbol inicial y, para todo i, $\mathbf{T_{i+1}}$ es una extensión de $\mathbf{T}_i$.
- Un *SLDNF-árbol para C* es un SLDNF-árbol con C como raíz del árbol principal.
- Se dice que un (pre-)SLDNF-árbol *"tiene éxito"* si contiene hojas marcadas como *éxito*.
- Se dice que un (pre-)SLDNF-árbol *"falla de forma finita"* si es finito y todas sus hojas están marcadas como *"fallo"*.
- Se dice que un SLDNF-árbol es *finito* si no contiene *caminos infinitos* (cf. Definición 4.8.4).

Definamos ahora el concepto de SLDNF-derivación.

4.8.7 Definición. Una *(pre-)SLDNF-derivación* para C es una rama del árbol principal de $\mathbf{T}$ junto con todos los árboles de $\mathbf{T}$ cuyas raíces puedan ser alcanzadas desde los nodos de esta rama. Se dice que una SLDNF-derivación es *finita* si todos los caminos contenidos en ella (i.e., en la rama del árbol principal considerada y sus árboles subsidiarios) son finitos.

Finalmente, veamos la definición de *respuesta computada.*

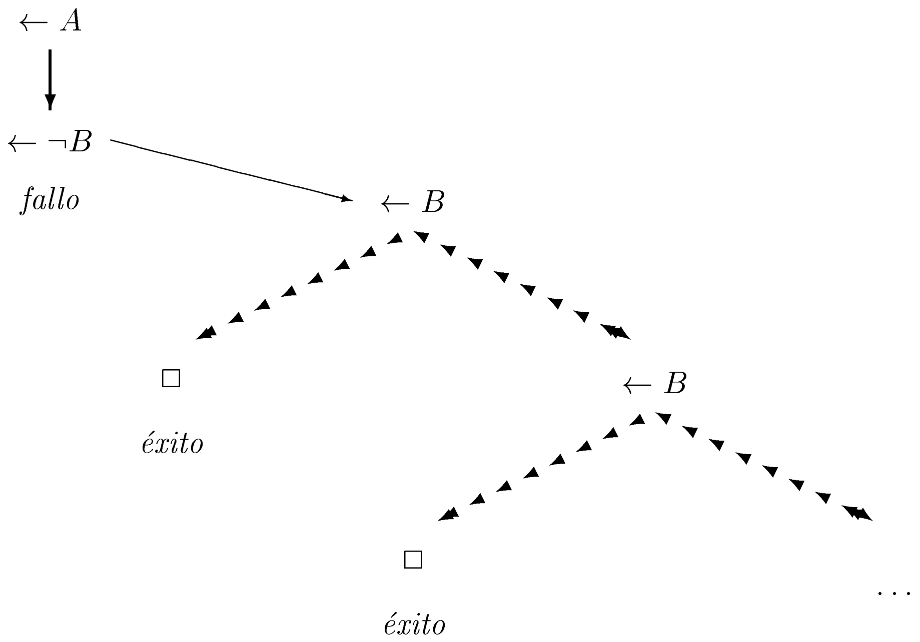
4.8.8 Definición. Consideremos una rama del árbol principal de un (pre-)SLDNF-árbol que termine con la cláusula vacía. Sean $\theta_1, \ldots, \theta_n$ las sustituciones obtenidas a lo largo de esa rama. Entonces a la restricción $(\theta_1 \ldots \theta_n) \restriction V(C)$ de la composición $\theta_1 \ldots \theta_n$ a las variables de C la llamaremos *respuesta computada* de C.

Los pre-SLDNF-árboles pueden crecer indefinidamente. No obstante, si el SLDNF-resultante tiene éxito o falla de forma finita, este hecho puede ser conocido tras un número finito de pasos.

4.8.9 Ejemplo. consideremos el siguiente programa P:

$$\begin{array}{l} A \leftarrow \neg B \\ B \leftarrow B \\ B \leftarrow \end{array}$$

Si consideramos el objetivo $\leftarrow A$, tenemos el siguiente SLDNF-árbol de $P \cup \{\leftarrow A\}$.



Aquí el árbol subsidiario con raíz $\leftarrow B$ crece indefinidamente. No obstante una extensión del árbol subsidiario inicial formado por el nodo $\leftarrow B$ tiene éxito, por lo que en la siguiente extensión el nodo $\leftarrow \neg B$ es marcado como *fallo*. En consecuencia el SLDNF-árbol se considera finitamente fallido aunque no sea finito.

El siguiente teorema es el principal resultado presentado por Apt y Doets en su artículo.

4.8.10 Teorema.

1. *Todo pre-SLDNF-árbol es finito.*
2. *Todo SLDNF-árbol es límite de una única sucesión de pre- SLDNF-árboles.*
3. *Si el SLDNF-árbol τ es límite de la sucesión de pre- SLDNF-árboles $\mathbf{T}_0, \ldots, \mathbf{T}_i, \ldots$, entonces para toda sustitución θ*
   (a) *τ tiene éxito y tiene a θ como respuesta computada si y sólo si algún $\mathbf{T}_i$ tiene éxito y tiene a θ como respuesta computada.*

(b) τ falla de forma finita si y sólo si algún $\mathbf{T}_i$ falla de forma finita.

Demostración:

1. Se tiene trivialmente por inducción.

2. La única forma de que dos extensiones de un pre-SLDNF-árbol incluido en un SLDNF-árbol dado sean distintas es por la selección de literales en los objetivos no vacíos. No obstante esta selección viene marcada por el SLDNF-árbol dado.

3. Vemos las equivalencias (I) y (II) con el mismo razonamiento:

($\Longrightarrow$) Una rama del árbol principal de τ finalizada con $\Box$ o un árbol de τ finitamente fallido no es más que un conjunto de nodos (posiblemente marcados). Cada uno de esos nodos (incluidos los marcados) pertenecen a algún $\mathbf{T}_i$. El mayor de esos $\mathbf{T}_i$ es el que buscamos.

($\Longleftarrow$) Se tiene trivialmente, ya que cada $\mathbf{T}_i$ está contenido en τ. $\Box$

Este resultado nos permite asociar a cada SLDNF-árbol con éxito o finitamente fallido τ un número natural que denominaremos *rango de τ y θ*, $rang(\tau, \theta)$, que es el menor i para el que se tiene la equivalencia (3), con $\theta = \varepsilon$ si τ es finitamente fallido.
Otro de los conceptos fundamentales en el estudio de la deducción negativa, el concepto de regla de selección, también está implícitamente definido. En la definición de SLDNF-árbol la regla de selección está "incorporada" en la construcción de una extensión. Claramete este proceso de selección puede ser separado de la construcción de una extensión, de forma que *una regla de selección es una función que en cada pre-SLDNF-árbol selecciona un literal en cada una de sus hojas distintas de la cláusula vacía no marcadas.*

4.8.2 Comparación con la definición de Lloyd

Las definiciones presentadas en la primeraparte del capítulo, que hasta el momento son las más aceptadas, fueron presentadas por Lloyd en la segunda edición de su libro *Foundations of Logic Programming* en 1987 [37]. En la segunda parte de su artículo Apt y Doets comparan esta definición clásica con su nueva definición.
Como hemos visto, Lloyd define las SLDNF-derivaciones finitamente fallidas y los SLDNF-árboles (para abreviar, SLDNF- objetos) por inducción sobre su rango, donde rango podemos definirlo informalmente como el número de niveles del árbol en los que seleccionamos literales negativos.

Así, en las SLDNF-derivaciones con éxito de rango 0 y árboles finitamente fallidos de rango 0 ningún literal negativo es seleccionado y la selección de un literal negativo cerrado $\neg A$ en un SLDNF-objeto de rango n+1 tiene éxito si existe un SLDNF-árbol con raíz $\leftarrow A$ finitamente fallido de rango n, y falla si existe una SLDNF-derivación de $\leftarrow A$ de rango n.
Ignorando pequeños detalles[16], hay tres diferencias fundamentales entre el tratamiento del preoblema de Lloyd y el de Apt y Doets.
La primera ya na sido mencionado y es el hecho de que para algunos programas y objetivos como $P = \{A \leftarrow A\}$ y $\neg A$ no exista ni SLDNF-refutaciones ni SLDNF-árboles finitamente fallidos según la definición de Lloyd.
La segunda es que según Lloyd la selección de un literal negativo cerrado $\neg A$ falla si existe un SLDNF-derivación para $\leftarrow A$, mientras que en el caso de Apt y Doets falla si existe un SLDNF-árbol con éxito para $\leftarrow A$. Reemplazando *SLDNF-derivación con éxito* por *SLDNF-árbol con éxito* se puede probar por inducción sobre el rango que si un SLDNF- objeto existe según la definición de Lloyd, entonces también existe segun la definición de Apt y Doets.
Por último el tratamiento de la *inviabilidad* es diferente. Para Lloyd un objetivo es inviable si consta únicamente de literales negativos no cerrados y la inviabilidad se deriva de no poder seleccionar ningún literal en ese objetivo. Para Apt y Doets la inviabilidad se deriva en el momento en que seleccionamos un literal negativo no cerrado en un objetivo. Evidentemente esta diferencia no es esencial y podemos adaptar fácilmente la definición de Lloyd, no obstante Apt y Dots destacan que su definición es más apropiada cuando se estudia SLDNF-resolución con una regla de selección fijada como puede ser en el caso de PROLOG, y su regla de *tomar el literal más a la izquierda.*

4.8.11 Nota. Distintos problemas relacionados con la SLDNF- resolución como puedan ser la terminación, la ausencia de inviabilidad o la selección de literales negativos no cerrados sólo serán resueltos una vez que tengamos una definición formal y clara de la misma. Los distintos trabajos publicados con un carácter eminentemente práctico carecían de este rigor tanto en las definiciones como en las demostraciones de los resultados. En su artículo Apt y Doets profundizan en este tema y dan una definición que perfila resultados conocidos aunque deja cabos sin atar, con lo que abre todo un campo para futuras investigaciones.

[16] Hay pequeñas diferencias en las presentación de la teoría entre Lloyd y la que hacen Apt y Doets. En la adaptación que hacemos de su artículo estas diferencias han sido subsanadas.

Apéndice A

Preliminares

El estudio de la Programación Lógica es esencialmente el estudio de ciertos conjuntos de fórmulas de un lenguaje de primer orden. Este primer apéndice presenta los principales resultados en Teoría de Conjuntos y Lógica de Primer Orden que necesitamos en la presente memoria.

A.1 Nociones básicas de Teoría de Conjuntos

A.1.1 Definición. Sea S un conjunto. Una *relación* R sobre S es un subconjunto del producto cartesiano $S \times S$.

A.1.2 Definición. Una relación R sobre un conjunto S es un *preorden* si se verifican las siguientes propiedades:

1. *Reflexiva:* Para todo $x \in S$, $(x, x) \in R$.
2. *Transitiva:* Si $(x, y) \in R$ y $(y, z) \in R$, entonces $(x, z) \in R$, para todo $\{x, y, z\} \subseteq S$.

A.1.3 Definición. Una relación R sobre un conjunto S es un *orden parcial* si se verifican las siguientes condiciones:

1. R es un preorden.
2. Si $(x, y) \in R$ y $(y, x) \in R$ entonces $x = y$, para todo $\{x, y\} \subseteq S$. *Propiedad Antisimétrica.*

A.1.4 Nota. A partir de ahora usaremos la notación usual xRy para indicar $(x, y) \in R$ y si no se indica lo contrario $\leq$ denotará la relación indicada por el contexto.

A.1.5 Definición. Sea S un conjunto con un orden parcial $\leq$ y X un subconjunto de S. Entonces se dice que $a \in S$ es una *cota superior* de X si para todo $x \in X$ se tiene $x \leq a$. Análogamente se dice que $b \in S$ es una *cota inferior* de X si para todo $x \in X$ se tiene $b \leq x$.

A.1.6 Definición. Sea S un conjunto con un orden parcial $\leq$ y sea $X \subseteq S$. Se dice que $a \in S$ es el *supremo* de X, $sup(X)$, si a es una cota superior y para cualquier otra cota superior a' se tiene que $a \leq a'$. De forma análoga se dice que $b \in S$ es el *ínfimo* de X, $inf(X)$, si b es una cota inferior de X y para cualquier otra cota inferior de X, b', se tiene que $b' \leq b$.

A.1.7 Nota. Es fácil ver que dado un conjunto S y $X \subseteq S$, si la relación $\leq$ es orden parcial y existen $sup(X)$ o $inf(X)$, estos son únicos. No obstante si la relación sólo es un preorden no se tiene la unicidad de supremos e ínfimos.

A.1.8 Definición. Sea A es un conjunto y $\leq$ un orden parcial en A. Decimos que el par $(A, \leq)$ es un *semirretículo superior* si para cualquier subconjunto finito de A, X, existe $sup(X)$.

A.1.9 Definición. Sea A es un conjunto y $\leq$ un orden parcial en A. Decimos que el par $(A, \leq)$ es un *retículo* si para cualquier subconjunto finito de A, X, existe $sup(X)$ e $inf(X)$.

A.1.10 Definición. Sea A es un conjunto y $\leq$ un orden parcial en A. Decimos que el par $(A, \leq)$ es un *retículo completo* si para cualquier subconjunto de A, X, existe $sup(X)$ e $inf(X)$.

A.1.11 Nota. Sea L un retículo completo. Usaremos el símbolo $\top$ para referirnos al supremo de L, y el símbolo $\perp$ para el ínfimo de L.

A.1.12 Ejemplo. Sea S un conjunto y $\mathcal{P}(S)$ el conjunto de las partes de S, en el que establecemos el orden parcial dado por la inclusión de conjuntos $\subseteq$. Es fácil verificar que el par $(\mathcal{P}(S), \subseteq)$ es un retículo completo en el que $\top = S$ y $\perp = \emptyset$.

A.2 Sintaxis de la lógica de primer orden

A.2.1 Definición. Un *lenguaje de primer orden* consta de un alfabeto y del conjunto de fórmulas que podemos definir sobre él[1]. El *alfabeto* de un lenguaje de primer orden **L** consta de

[1]Una fundamentación rigurosa de los conceptos de Lógica que aquí presentamos puede encontrarse, por ejemplo, en [20]

1. Un conjunto numerable de *variables*, V.
2. Las *conectivas lógicas*: $\neg$ (negación) y $\bigvee$ (disyunción).
3. El *cuantificador existencial*: $\exists$
4. Un conjunto numerable de símbolos de función, posiblemente vacío, SF.
5. Un conjunto numerable de símbolos de predicado, SP.
6. Unos *símbolos auxiliares*: "(", ")" y ",".[2]

A.2.2 Nota.

1. Los conjuntos anteriormente descritos son disjuntos.
2. El conjunto de los símbolos de función SF lleva asociado una función *aridad* que a cada símbolo f le asocia un entero no negativo $a(f)$, llamado la *aridad del símbolo de función f*.
3. Análogamente, el conjunto de los símbolos de predicado SP lleva asociado una función *aridad* que a cada símbolo p le asocia un entero no negativo $a(p)$, llamado la *aridad del símbolo de predicado p*
4. Los símbolos de predicado de aridad cero se denominan *símbolos proposicionales*.
5. Llamaremos *constantes* a los símbolos de función de aridad cero y denotaremos por SC al conjunto de las constantes.
6. Usaremos los siguientes símbolos para referirnos a los elementos de los distintos conjuntos. (Los usaremos frecuentemente con subíndices)
   - $\ldots, x, y, z$ serán variables.
   - $a, b, c, \ldots$ serán constantes.
   - $f, g, h, \ldots$ serán símbolos de función de aridad ≥ 1.
   - $p, q, r, \ldots$ serán símbolos de predicado.

A.2.3 Nota. En lo que sigue, nos referiremos siempre al lenguaje de primer orden **L** así definido.

[2]Además de los conjuntos de símbolos que aquí damos otros autores (Hogger [26] y Apt [3], por ejemplo) añaden las constantes booleanas TRUE y FALSE como símbolos proposicionales especiales, no obstante, la construcción de la Programación Lógica se puede hacer sin necesidad de estas constantes, como veremos.

A.2.4 Definición. Adoptaremos la siguiente definición inductiva de *término*:

1. Una variable es un término
2. Una constante es un término
3. Si f es un símbolo de función n-ario y $t_1, \ldots, t_n$ son téminos, entonces $f(t_1, \ldots, t_n)$ es un término.

Llamaremos $TERM_{\mathbf{L}}$ al conjunto de téminos del lenguaje **L** y simplemente $TERM$ cuando no exista ambigüedad.

A.2.5 Definición. Una *fórmula* se define del siguiente modo:

1. Si p es un símbolo de predicado n-ario y $t_1, \ldots, t_n$ son términos entonces $p(t_1, \ldots, t_n)$ es una fórmula, que llamaremos *fórmula atómica* o simplemente *átomo*.
2. Si F y G son fórmulas entonces $(\neg F)$ y $(F \vee G)$ son fórmulas.
3. Si F es una fórmula y x es una variable entonces $(\exists x F)$ también es una fórmula. Llamaremos $FORM_{\mathbf{L}}$ al conjunto de téminos del lenguaje **L** y simplemente $FORM$ cuando no exista ambigüedad.

A.2.6 Nota. Sean F y G dos fórmulas y x una variable. Adoptaremos las siguientes abreviaturas:

- Escribiremos $\forall x F$ en lugar de $\neg \exists x \neg F$
- $F \wedge G$ en lugar de $\neg((\neg F) \vee (\neg G))$
- $F \rightarrow G$ en lugar de $((\neg F) \vee G)$
- $F \leftrightarrow G$ en lugar de $(F \rightarrow G) \wedge (G \rightarrow F)$
- $(\vee_{i=1}^{n} F_i)$ en lugar de $(F_1 \vee (F_2 \vee \ldots \vee (F_{n-1} \vee F_n) \ldots))$
- $(\wedge_{i=1}^{n} F_i)$ en lugar de $(F_1 \wedge (F_2 \wedge \ldots \wedge (F_{n-1} \wedge F_n) \ldots))$

A.2.7 Nota. Para simplificar la notación adoptaremos los siguientes convenios:

1. Pueden eliminarse los paréntesis externos.

2. Las conectivas tienen una precedencia de asociación. De menor a mayor están ordenadas por: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.

3. Cuando una conectiva se asocia repetidamente, se asocia por la derecha.

A.2.8 Definición. Diremos que e es una *expresión* de **L** si es un término o una fórmula de **L**.

A.2.9 Nota. Si e es una expresión, denotaremos como $V(e)$ al conjunto de variables que ocurren en e.

A.2.10 Definición. El *alcance* de $\forall x$ (resp. $\exists x$) en la fórmula $\forall xF$ (resp. $\exists xF$) es F.

A.2.11 Definición. Definimos recursivamente el conjunto de las *variables libres* de un término t, $Libre(t)$, del siguiente modo:

$$Libre(t) = \begin{cases} \{x\} & \text{si } t = x \text{ es una variable} \\ Libre(t_1) \cup \ldots \cup Libre(t_n) & \text{si } t = f(t_1 \ldots t_n) \end{cases}$$

A.2.12 Definición.

1. El conjunto de las *variables libres* de una fórmula F, $Libre(F)$, se define recursivamente del siguiente modo:

$$Libre(F) = \begin{cases} Libre(t_1) \cup \ldots \cup Libre(t_n) & \text{si } t = p(t_1 \ldots t_n) \\ Libre(G) & \text{si } F = \neg G \\ Libre(G) \cup Libre(H) & \text{si } F = (G \vee H) \\ Libre(G) - \{x_i\} & \text{si } F = \exists x_i G \end{cases}$$

2. El conjunto de las *variables ligadas* de una fórmula F, $Ligada(F)$, se define recursivamente del siguiente modo:

$$Ligada(F) = \begin{cases} Ligada(t_1) \cup \ldots \cup Ligada(t_n) & \text{si } t = p(t_1 \ldots t_n) \\ Ligada(G) & \text{si } F = \neg G \\ Ligada(G) \cup Ligada(H) & \text{si } F = (G \vee H) \\ Ligada(G) \cup \{x_i\} & \text{si } F = \exists x_i G \end{cases}$$

A.2.13 Definición.

1. Un *término cerrado* es un término en el que no ocurre ninguna variable.

2. Una *fórmula cerrada* es una fórmula en la que no ocurre libre ninguna variable. Por tanto un *átomo cerrado* es un átomo en el que no ocurre ninguna variable.

3. Una *fórmula abierta* es una fórmula que no contiene cuantificadores.

A.2.14 Definición. Si $Libre(F) = \{x_1, \ldots, x_n\}$ entonces:

1. La clausura universal de F, $\forall(F)$ es

$$(\forall x_1) \ldots (\forall x_n) F$$

2. Análogamente, la clausura existencial de F, $\exists(F)$ es

$$(\exists x_1) \ldots (\exists x_n) F$$

A.2.1 Sustituciones

A.2.15 Definición. Dado un lenguaje de primer orden $\mathbf{L}$, una *sustitución* θ es una función del conjunto V de las variables en el conjunto $TERM$ de los términos

$$\theta : V \to TERM$$

Denotaremos por $SUST$ al conjunto de las sustituciones.

A.2.16 Definición. Adoptaremos las siguientes definiciones:

1. Definimos el *dominio* de una sustitución θ como el conjunto siguiente $Dom(\theta) = \{x \in V : \theta(x) \neq x\}$

2. El *codominio* de una sustitución θ como $Cod(\theta) = \{\theta(x) : x \in Dom(\theta)\}$

3. El *rango* de una sustitución θ como $Rang(\theta) = \bigcup_{x \in Dom(\theta)} V(\theta(x))$

A.2.17 Definición. Llamaremos *sustitución identidad* a la sustitución de dominio vacío y la denotaremos ϵ.

A.2.18 Definición. Una sustitución θ es *finita* si $D(\theta)$ es finito.

A.2.19 Nota. En lo que sigue sólo consideraremos sustituciones finitas y omitiremos el adjetivo.

A.2.20 Definición. Diremos que una sustitución θ es *básica* si para todo $x \in Dom(\theta)$, $\theta(x)$ es un término cerrado.

A.2.21 Nota. Sea θ una sustitución de dominio $Dom(\theta) = \{x_1, \ldots, x_n\}$. Notaremos θ de la siguiente forma

$$\{x_1/\theta(x_1), \ldots x_n/\theta(x_n)\}$$

A.2.22 Definición. Dada una sustitución θ, a cada par ordenado $(x, \theta(x))$ donde $x \in Dom(\theta)$ lo llamaremos *enlace* de la sustitución. Con la notación anterior, un enlace es cada una de las expresiones $x/\theta(x) \in \theta$.

A.2.23 Teorema. *Para cada sustitución θ existe una única aplicación*

$$\theta' : TERM \to TERM$$

definida por

$$\theta'(t) = \begin{cases} \theta(t) & si\ t\ es\ una\ variable \\ f(\theta'(t_1) \ldots \theta'(t_n)) & si\ t = f(t_1 \ldots t_n) \end{cases}$$

Haciendo un abuso de notación escribiremos $\theta(t)$ en lugar de $\theta'(t)$ y seguiremos llamando sustitución a esta nueva aplicación.[3]

A.2.24 Definición. Sean dos términos s y t y la sustitución $\theta = \{x/t\}$. El término resultante de sustituir en s la variable x por el término t es

$$s[x/t] = \theta(s)$$

A.2.25 Definición. Sean F una fórmula y t un término. La fórmula resultante de sustituir en F la variable x por el término t se define recursivamente por

$$F[x/t] = \begin{cases} p(t_1[x/t], \ldots, t_n[x/t]) & \text{si } F = p(t_1, \ldots, t_n) \\ \neg G[x/t] & \text{si } F = \neg G \\ G[x/t] \vee H[x/t] & \text{si } F = G \vee H \\ \exists y G[x/t] & \text{si } F = \exists y G \text{ y } x \neq y \\ \exists x G & \text{si } F = \exists x G \end{cases}$$

[3] Para poder dar la demostración de este resultado y otros análogos que citamos más adelante sería necesario establecer algunos resultados previos sobre el principio de inducción para términos y fórmulas y la libre generación de éstos, en los que no vamos a entrar. Estos resultados podemos encontrarlos en [20].

A.2.26 Definición. Una variable x de F es *sustituible* por el término t si se cumple una de las siguientes condiciones:

1. F es atómica.
2. $F = \neg G$ y x es sustituible por t en G.
3. $F = G \vee H$ y x es sustituible por t en G y H.
4. $F = \exists x G$
5. $F = \exists y G$, $x \neq y$, y no ocurre libre en t y x es sustituible por t en G.

A.2.27 Nota.

1. En lo sucesivo, al escribir $F[x/t]$ supondremos que x es sustituible por t en F.
2. Si una fórmula F contiene una variable libre x, escribiremos F como $F(x)$ y abreviaremos $F[x/t]$ como $F(t)$.

A.3 Semántica

A.3.1 Definición. Sea **L** un lenguaje de primer orden. Una *preinterpretación* de **L** es un par **M'**= (D, I) tal que:

1. D es un conjunto no vacío, llamado el *dominio* de la preinterpretación.
2. I es una aplicación definida sobre SF tal que:
   (a) Para cada constante c del lenguaje, $I(c) \in D$.
   (b) Para cada símbolo de función n-ario del lenguaje $I(f)$ es una aplicación parcial de D^n en D.
   $$I(f) : D^n \to D$$
   tal que $I(f)(I(t_1), \ldots, I(t_n)) = I(f(t_1, \ldots, t_n))$.

A.3.2 Definición. Sea **L** un lenguaje de primer orden. Una *interpretación* de **L** es un par **M**= (D, I) tal que:

1. **M** es una preinterpretación de **L**.

2. Además extendemos la aplicación I a los símbolos de predicado del siguiente modo: Para todo símbolo de predicado n- ario ($n > 0$)

$$I(p) \subseteq D^n$$

En este caso se dice que la interpretación **M** está basada en la interpretación **M'**.

A.3.3 Definición. Sea **L** un lenguaje de primer orden y **M**$= (D, I)$ una interpretación de **L**. Una *asignación* σ es una aplicación del conjunto de las variables de **L**, V, en el universo de la interpretación, D.

$$\sigma : V \to D$$

A.3.4 Teorema. *Sea* **M**$= (D, I)$ *una interpretación de* **L** *. Para cada asignación σ existe un única aplicación $\bar{\sigma} : TERM \to D$ definida por*

$$\bar{\sigma}(t) = \begin{cases} \sigma(t) & \textit{si } t \textit{ es una variable} \\ I(f)(\bar{\sigma}(t_1), \ldots, \bar{\sigma}(t_n)) & \textit{si } t = f(t_1, \ldots, t_n) \end{cases}$$

En lo sucesivo escribiremos $\sigma(t)$ en lugar de $\bar{\sigma}(t)$.

A.3.5 Definición. Sea **M**$= (D, I)$ una interpretación para **L**, $d \in D$ y σ una asignación. Se representa por $\sigma[x/d]$ la asignación σ' definida por

$$\sigma'(y) = \begin{cases} d & \text{si } y = x \\ \sigma(y) & \text{si } y \neq x \end{cases}$$

A.3.1 Consistencia y validez

A.3.6 Definición. Los elementos del conjunto $2 = \{1, 0\}$ se llaman *valores de verdad.* Diremos que 0 es el valor *falso* y que 1 es el valor *verdadero.*

A.3.7 Definición. Definimos las *funciones de verdad* del siguiente modo

1. La *función de verdad* de la negación es $H_\neg : 2 \to 2$ definida por

$$H_\neg(i) = 1 - i$$

2. La *función de verdad* de la disyunción es $H_\vee : 2 \times 2 \to 2$ definida por

$$H_\vee(i, j) = max\{i, j\}$$

3. Se define la función $H_\exists : \mathcal{P}(2) \to 2$ del siguiente modo:

$$H_\exists(X) = \begin{cases} 1, & \text{si } 1 \in X \\ 0, & \text{en caso contrario} \end{cases}$$

A.3.8 Teorema. *Sea* **M** *una interpretación de* **L**. *Para cada asignación* σ *existe una única aplicación* $\tilde{\sigma} : FORM \to 2$ *tal que*

$$\tilde{\sigma}(F) = \begin{cases} 1, & \textit{si } F = p(t_1, \dots, t_n) \\ & \ \textit{e } (I(t_1), \dots, I(t_n)) \in I(p) \\ 0, & \textit{si } F = p(t_1, \dots, t_n) \\ & \ \textit{e } (I(t_1), \dots, I(t_n)) \notin I(p) \\ H_\neg(G) & \textit{si } F = \neg G \\ H_\vee(G, H) & \textit{si } F = G \vee H \\ H_\exists(\{\sigma[x/m](G) : \ m \in D\}) & \textit{si } F = \exists x G \end{cases}$$

En lo sucesivo escribiremos $\sigma(F)$ *en lugar de* $\tilde{\sigma}(F)$.

A.3.9 Definición. Sea **L** un lenguaje de primer orden, F una fórmulas y **M** una interpretación de **L**. Diremos que:

1. F es *válida en* **M** *respecto de la asignación* σ, $\mathbf{M} \models_\sigma F$ si $\sigma(F) = 1$.

2. F es *válida en* **M**, $\mathbf{M} \models F$, si $\sigma(F) = 1$ para todas las asignaciones σ. En este caso se dice que **M** es un *modelo* de F y que **M** *satisface* la fórmula F.

3. F es *válida*, $\models F$, si es válida en todas las interpretaciones de **L**.

4. F es *consistente* o *satisfacible* si tiene algún modelo.

A.3.10 Nota. Las definiciones anteriores son las mismas en el caso en que hablemos de un conjunto de fórmulas Γ.

A.3.11 Definición. Sea **T** una teoría de primer orden y sea **L** el lenguaje de **T**. Un *modelo* de **T** es una interpretación de **L** que sea modelo para todo axioma de **T**. Si **T** tiene un modelo entonces decimos que es *consistente*.

A.3.12 Definición. Sea S un conjunto de fórmulas cerradas y F una fórmula cerrada. Diremos que F es *consecuencia lógica* de S, y lo denotaremos $S \models F$, si para toda interprtetación **M**, si **M** es modelo de S, entonces es modelo de F.

A.3.13 Definición. Dos fórmulas F y G son *lógicamente equivalentes*, $F \equiv G$, si para toda interpretación $\mathbf{M}$ se tiene que

$$\mathbf{M} \models F \text{ si y sólo si } \mathbf{M} \models G$$

A.3.14 Teorema. *Sea S un conjunto de fórmulas cerradas y F una fórmula cerrada. Entonces se tiene que F es consecuencia lógica de S si $S \cup \{F\}$ es insatisfacible.*

A.4 Formas prenexas

A.4.1 Definición. Una fórmula F está en *forma prenexa* si es de la forma

$$Q_1 x_1 \dots Q_n x_n G$$

donde para todo $i \in \{1, \dots n\}$, $Q_i \in \{\forall, \exists\}$, x_i son variables distintas y G es una fórmula abierta. Se dice que $Q_1 x_1 \dots Q_n x_n$ es el *prefijo* de F y que G es la *matriz* de F.

A.4.2 Teorema. *Para cada fórmula F existe una fórmula F' en forma prenexa equivalente a F.*

A.4.3 Nota. Existen algoritmos tales que dada cualquier fórmula F devuelven una fórmula F' en forma prenexa equivalente a la fórmula dada.

A.4.4 Definición. Un *literal* es un átomo o la negación de un átomo. Un *literal positivo* es un átomo. Un *literal negativo* es la negación de un átomo.

A.4.5 Definición.

- El *complementario* de un literal L es

$$\bar{L} = \begin{cases} \neg p(t_1, \dots, t_n) & \text{si } L \text{ es } p(t_1, \dots, t_n) \\ p(t_1, \dots, t_n) & \text{si } L \text{ es } \neg p(t_1, \dots, t_n) \end{cases}$$

- Dos literales L y L' son *complementarios* si

$$L' = \bar{L}$$

A.4.6 Definición.

- Una *cláusula conjuntiva* C es la conjunción de un conjunto finito de literales, i.e.,
$$C = \bigwedge_{i=1}^{n} L_i$$
- Una *cláusula disyuntiva* D es la disyunción de un conjunto finito de literales, i.e.,
$$D = \bigvee_{i=1}^{n} L_i$$

A.4.7 Definición.

- Una fórmula está en *forma normal conjuntiva, FNC*, si está en forma prenexa y su matriz es una conjunción de un conjunto de cláusulas disyuntivas, i.e.,
$$F = Q_1x_1 \dots Q_nx_n \bigwedge\nolimits_{i=1}^{n} (\bigvee\nolimits_{j=1}^{m_i} L_{i,j})$$
- Una fórmula está en *forma normal disyuntiva, FND*, si está en forma prenexa y su matriz es una disyunción de un conjunto de cláusulas conjuntivas, i.e.,
$$F = Q_1x_1 \dots Q_nx_n \bigvee\nolimits_{i=1}^{n} (\bigwedge\nolimits_{j=1}^{m_i} L_{i,j})$$

A.4.8 Definición.

1. F es una *forma normal conjuntiva* de G si F está en forma normal conjuntiva y $F \equiv G$.
2. F es una *forma normal disyuntiva* de G si F está en forma normal disyuntiva y $F \equiv G$.

A.4.9 Teorema. *Para cada fórmula F existen fórmulas G_1 y G_2 tales que G_1 es una forma normal conjuntiva de F y G_2 es una forma normal disyuntiva de F.*

A.4.10 Nota. Existe un algoritmo tal que dada cualquier fórmula F devuelve una forma normal conjuntiva (resp. disyuntiva) de F.

Apéndice B

Unificación

En este apéndice tratamos uno de los temas más importantes dentro de la fundamentación de la Programación Lógica y que tiene ramificaciones dentro de gran parte de las matemáticas actuales. La elaboración del apéndice se basa en recientes trabajos de Nipkow [40] y Lassez, Maher y Marriot [34], aunque también con aportaciones de Apt [3] y Lloyd [37]. Los conceptos que utilizo han sido definidos en el apéndice anterior.

B.1 La relación subsunción

B.1.1 Definición. Si $W \subseteq V$ entonces $\sigma \restriction_W$ es la restricción de σ a V, i.e.,

$$\sigma \restriction_W (x) = \begin{cases} \sigma(x) & \text{si } x \in W \\ x & \text{en otro caso} \end{cases}$$

B.1.2 Nota. Notaremos como $(s)\theta$ el término que se obtiene de aplicar a s la sustitución θ.

B.1.3 Definición. La *composición* de las sustituciones σ y τ la abreviaremos $\sigma\tau$. Así $(t)(\sigma\tau) = ((t)\sigma)\tau$. Diremos que una sustitución σ es *idempotente* si $\sigma\sigma = \sigma$. Nótese que en este caso $Dom(\sigma) \cap Rang(\sigma) = \emptyset$.

B.1.4 Definición. Decimos que el término t es una *instancia* del término s si existe una sustitución σ tal que $\sigma(s) = t$. En este caso también diremos que s es *más general* que t, (o bien que s es *menos particular* que t), y lo denotaremos $t \leq s$.

La elección de la orientación del símbolo $\leq$ vienen justificada por el hecho de que un término puede ser considerado como el conjunto de todas sus instancias, y desde este punto de vista t es un subconjunto de s.[1]

B.1.5 Proposición. *La relación $\leq$ definida en $TERM$ del siguiente modo:*

$$t \leq s \iff \textit{Existe una sustitución } \sigma \textit{ tal que } (s)\sigma = t$$

es un preorden en $TERM$, es decir, es reflexiva y transitiva en $TERM$.

B.1.6 Definición. Llamaremos *subsunción* a la relación en $TERM$ definida en la proposición anterior.

B.1.7 Definición. Definimos en $TERM$ la relación $\equiv$ del siguiente modo:

$$s \equiv t \iff (s \leq t) \wedge (t \leq s)$$

Si $s \equiv t$ diremos que s y t son *equivalentes*, o bien que s es una *variante* de t (y viceversa).

B.1.8 Definición. Se define en $TERM$ la relación $<$ de la siguiente manera:

$$s < t \iff s \leq t \quad y \quad t \not\leq s$$

Llamaremos a esa relación *subsunción estricta.*

La demostración del siguiente teorema así como la de los demás resultados de este apéndice pueden encontrarse en [28].

B.1.9 Teorema. *No existe ninguna cadena descendente infinita de subsunción estricta $t_1 > t_2 > t_3 > \ldots$, i.e., la relación $<$ está bien fundamentada.*

B.1.10 Ejemplo. Sea s_1 el término $f(x, y, h(z))$. Una cadena descendente de subsunción estricta comenzando en s_1 podría ser

$$\begin{array}{lll}
Si\ \ \theta_1 = \{x/a\} & entonces & s_2 = (s_1)\theta_1 = f(a, y, h(z)) \\
Si\ \ \theta_2 = \{y/b\} & entonces & s_3 = (s_2)\theta_2 = f(a, b, h(z)) \\
Si\ \ \theta_3 = \{z/c\} & entonces & s_4 = (s_3)\theta_3 = f(a, b, h(c))
\end{array}$$

No existe ningún término estrictamente menor que s_4 por lo que hemos obtenido l cadena descendente *finita* $s_1 > s_2 > s_3 > s_4$.

[1] Intuitivamente, si consideramos $Conj(t) = \{t\theta : \theta \text{ sustitución }\}$ tenemos que

$$s \leq t \iff Conj(s) \subseteq Conj(t)$$

B.1.11 Definición. Una *permutación* es una aplicación $\xi : V \to V$ biyectiva[2].

B.1.12 Teorema. *Para cada permutación ξ existe una única aplicación*

$$\xi' : TERM \to TERM$$

definida por

$$(t)\xi' = \begin{cases} (t)\xi & \textit{si } t \textit{ es una variable} \\ f((t_1)\xi', \ldots, (t_n)\xi') & \textit{si } t = f(t_1 \ldots t_n) \end{cases}$$

Haciendo un abuso de notación escribiremos $(t)\xi$ en lugar de $(t)\xi'$

B.1.13 Lema. $s \equiv t \iff$ *Existe una permutación ξ tal que $(t)\xi = s$*

B.1.14 Lema. *La relación $\equiv$ es de equivalencia en $TERM$.*

B.1.15 Nota. Adoptaremos la siguiente notación respecto de la relación de equivalencia:

- Para cada término t, representaremos por $[t]$ su clase de equivalencia, i.e.,
$$[t] = \{t' \in TERM : t' \equiv t\}$$
- Representaremos por $\mathbf{T}$ el conjunto cociente de $TERM$ respecto de $\equiv$, i.e.,
$$\mathbf{T} = \{[t] : t \in TERM\}$$

B.1.16 Definición. Se define en $\mathbf{T}$ la relacion $\leq$ del siguiente modo:

$$[t] \leq [s] \iff t \leq s$$

B.1.17 Lema. *La relación $\leq$ es un orden parcial en $\mathbf{T}$.*

B.1.18 Nota. El mayor elemento de $\mathbf{T}$ es el conjunto de las variables.

Esto se tiene porque dado un término cualquiera t y una variable x siempre podemos encontrar la sustitución $\theta = \{x/t\}$, con lo que tendremos $t \leq x$.

[2]Una permutación es una sustitución, por lo que notaremos como $(x)\xi$ a la variable que resulta de aplicar la permutación ξ a la variable x.

B.1.19 Definición. Sea $\phi : TERM \times TERM \to V$ una biyección[3]. Definimos a partir de ϕ la siguiente aplicación $\sqcup_\phi : TERM \times TERM \to TERM$

$$\sqcup_\phi(s,t) = \begin{cases} f(s_1 \sqcup_\phi t_1, \ldots, s_n \sqcup_\phi t_n) & \text{si } s = f(s_1, \ldots, s_n) \text{ y } t = f(t_1, \ldots, t_n) \\ \phi(s,t) & \text{en otro caso} \end{cases}$$

A partir de ahora denotaremos $\sqcup_\phi(s,t)$ como $s \sqcup_\phi t$.

B.1.20 Definición. Dada la biyección $\phi : TERM \times TERM \to V$ y su correspondiente $\sqcup_\phi : TERM \times TERM \to TERM$ se define la siguiente aplicación $\tilde{\sqcup}_\phi : SUST \times SUST \to SUST$ como sigue:

$$(x)\tilde{\sqcup}_\phi(\sigma_1, \sigma_2) = (x)\sigma_1 \sqcup_\phi (x)\sigma_2$$

Denotaremos por $\sigma_1 \sqcup_\phi \sigma_2$ la sustitución $\tilde{\sqcup}_\phi(\sigma_1, \sigma_2)$

Obviamente, como cualquier otra sustitución, $\sigma_1 \sqcup_\phi \sigma_2$, se extiende de forma única al conjunto de los términos.

Puesto que la operación $\leq$ definida en $TERM$ es sólo un preorden, el supremo de un conjunto no está unívocamente determinado, así que sólo podemos hablar de la existencia o no de *un* supremo.

B.1.21 Proposición. *El término $t_1 \sqcup_\phi t_2$ es un supremo de t_1 y t_2 con la relación susbsunción.*

B.1.22 Lema. *Sean ϕ y ψ dos biyecciones cualesquiera de $TERM \times TERM$ en V, ϕ, $\psi : TERM \times TERM \to V$, entonces $s \sqcup_\phi t \equiv s \sqcup_\psi t$.*

B.1.23 Teorema. *En virtud del lema anterior podemos afirmar que* $\mathbf{T}$ *es un semirretículo superior con un operador supremo $\sqcup$ que llamaremos "la menor generalización".*

Combinando los resultados anteriores se tiene el siguiente teorema.

B.1.24 Teorema. *Dados dos términos cualesquiera s y t con una instancia común, esto es, que exista un término u tal que $u \leq s$ y $u \leq t$, entonces existe un ínfimo de s y t que denotaremos $s \sqcap t$.*

B.1.25 Nota. Representaremos por $\mathcal{T}$ al conjunto $\mathbf{T}$ al que hemos añadido un menor elemento $\perp$

[3]Esa biyección existe por se ambos conjuntos infinitos numerables.

B.1.26 Teorema. $(T, \leq)$ *es un retículo completo.*

B.1.27 Definición. En el conjunto de las sustituciones, $SUST$,establecemos la siguiente relación:

$$\theta \leq \theta' \quad \Longleftrightarrow \quad \text{existe una sustitución } \theta'' \text{ tal que } \theta'\theta'' = \theta$$

Si $\theta \leq \theta'$ se dice que θ es *menos particular* que θ'.

B.1.28 Lema. *La relación $\leq$ es un preorden en $SUST$, i.e., es reflexiva y transitiva en el conjunto de las sustituciones.*

B.1.29 Lema. *Si* **L** *tiene símbolos de función entonces*

$$\theta \leq \theta' \quad \Longleftrightarrow \quad \forall t \in TERM, \quad \theta(t) \leq \theta'(t)$$

.

B.1.30 Lema. *En el conjunto de las sustituciones, $SUST$, se define la siguiente relación:*

$$\theta \equiv \theta' \quad \Longleftrightarrow \quad (\theta \leq \theta') \wedge (\theta' \leq \theta)$$

Si $\theta \equiv \theta'$ decimos que θ y θ' son equivalentes

B.1.31 Lema. *$\theta \equiv \theta'$ si y sólo si existe una permutación ξ tal que $\theta\xi = \theta'$*

B.1.32 Lema. *La relación $\equiv$ es de equivalencia en el conjunto $SUST$ de las sustituciones.*

B.1.33 Lema. *Si* **L** *tiene símbolos de función entonces*

$$\theta \equiv \theta' \quad \Longleftrightarrow \quad \forall t \in TERM, \quad \theta(t) \equiv \theta'(t)$$

B.2 Unificación

El concepto de unificación, fundamental dentro de la Programación Lógica, ha tenido una historia controvertida. Podemos afirmar que el primer algoritmo de unificación fue dado por Herbrand [24] en 1930 para resolver ecuaciones. Posteriormente fue Robinson [44] quien retomó los estudios de Herbrand dando la primera definición de unificador de máxima generalidad como el unificador obtenido mediante el algoritmo presentado en el mismo trabajo. Más tarde fue el mismo Robinson en 1979 quien dio la siguiente definicion de umg: μ *es el umg* $\leftrightarrow$ *Para todo unificador* θ, $\theta = \mu\theta$. Una tercera definición, dada por Chang y Lee en 1973 [13] y adoptada por Lloyd en su trabajo de 1984 [36] fue aceptada como "oficial" durante algún tiempo y la encontramos en trabajos como los de Apt [3]:μ *es el umg* $\leftrightarrow$ *para todo unificador* θ *existe una susutitución* β *tal que* $\theta = \mu\beta$.[4] Más tarde, en 1986 Sterling y Shapiro dan una definición basada en el orden parcial establecido entre los términos. En esta memoria seguimos a Nipkow [40] en su trabajo de 1992.

B.2.1 Definición. Sean t_1 y t_2 dos términos.

1. Se dice que la sustitución θ es un *unificador* de t_1 y t_2 si $(t_1)\theta = (t_2)\theta($.
2. Representaremos por $\mathcal{U}(t_1, t_2)$al conjunto de los unificadores de t_1 y t_2
3. Se dice que t_1 y t_2 son *unificables* si $\mathcal{U}(t_1, t_2)\neq \emptyset$
4. θ es un *unificador de máxima generalidad* de t_1 y t_2 si
   - $\theta \in \mathcal{U}(t_1, t_2)$
   - Para toda $\theta' \in \mathcal{U}(t_1, t_2)$, $\theta' \leq \theta$, o dicho de otro modo ,existe un sustitución θ'' al que $\theta\theta'' = \theta'$.[5]
   - Se dice que un umg θ de dos átomos A yB es *relevante* si

     $$Dom(\theta) \cup Rang(\theta) \subseteq V(A) \cup V(B)$$

     esto es, si toda variable que ocurra en algún enlace de θ ocurre en A o en B.

Nótese que puesto que $\leq$ es sólo un preorden no podemos hablar del *el* supremo de un conjunto.

[4] Lloyd[37] define la composición de sustituciones de la siguiente manera: Dadas dos sustituciones $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ y $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$, la sustitución composición $\theta\sigma$ es la sustitución obtenida a partir de

$$\{u_1/s_1\sigma, \ldots, u_m/s_m\sigma, v_1/t_1, \ldots, v_n/t_n\}$$

eliminando los enlaces que

- Sean de la forma $u_i/s_i\sigma$ y $u_i = s_i\sigma$
- Sean de la forma v_j/t_j y $v_j \in \{u_1, \ldots, u_m\}$

Esta definición es equivalente a la que hemos dado.

[5] Esta definición es equivalente a la de Chang y Lee [13].

B.2.2 Lema. *Sean t_1 y t_2 dos términos y θ y θ' dos unificadores de máxima generalidad de t_1 y t_2. Entonces $\theta \equiv \theta'$ i.e., dados dos términos su unificador de máxima qeneralidad es único salvo equivalencias.*

B.2.3 Nota. Representaremos por $umg(t_1, t_2)$ al conjunto de unificadores de máxima generalidad de t_1 y t_2 y abreviaremos la expresión *"unificador de máxima generalidad"* como *umg*.

B.2.4 Ejemplo. Sean $t_1 = x$ y $t_2 = y$ dos téminos. La sustitución $\sigma = \{x/z, y/z\}$ es un unificador de x e y, pero según la definición anterior no es un *umg*, ya que dado el unificador $\sigma_1 = \{x/y\}$ (caso análogo con $\sigma_2 = \{y/x\}$) no podemos encontrar ninguna sustitución θ'' tal que $\{x/z, y/z\}\theta'' = \{x/y\}$. ($\sigma\{z/y\}$ es $\{x/y, z/y\}$ pero no σ_1).[6]

B.2.5 Nota. El concepto de unificador y de *umg* se extiende de forma natural a conjuntos de más de dos términos.

B.2.6 Lema. *Sean s_1 y s_2 términos tales que $V(s_1) \cap V(s_2) = \emptyset$. Entonces, si σ es un umg de s_1 y s_2 entonces σs_1 $(= \sigma s_2)$ es un ínfimo de s_1 y s_2.*

B.2.7 Nota. El lema anterior es falso si $V(s_1) \cap V(s_2) \neq \emptyset$. Así, si $s_1 = f(x, y)$ y $s_2 = f(y, x)$, un umg puede ser $\sigma = \{x/y\}$, pero $\sigma(s_1) = f(y, y)$ no es un ínfimo de s_1 y s_2. Este último sería, módulo permutación, $f(z_1, z_2)$.

B.3 Algoritmos de unificación

El problema de la unificación es mucho más amplio de lo que aquí abordamos. Podemos pensar que encontrar solución a la ecuación

$$2 + x = 5$$

no es más que encontrar la sustitución $\theta = \{x/3\}$. Este ejemplo pone de relieve que el problema de la Unificación subyace en cualquier problema que nos planteemos en Matemáticas.

Los algoritmos de unificación están muy estudiados. Desde el algoritmo de Herbrand de 1930 hasta el algoritmo lineal presentado por Martelli y Montanari [38] en 1976 ha corrido mucha tinta, aunque la investigación en el campo de la unificación no ha hecho más que empezar.

[6] He aquí un ejemplo de la controversia de la definición de *umg*. Existen definiciones para las que σ es en este caso *umg*.

B.3.1 Algoritmo de Herbrand

B.3.1 Definición. Sea S un conjunto de expresiones. El *conjunto discordante* de S, D_S se define como el resultado del siguiente "algoritmo": Comparamos los símbolos de **L** que aparecen en las expresiones de S empezando por la izquierda, hasta encontrar algún símbolo distinto.

- Si comparando posición a posición todos los símbolos que encontramos son iguales, entonces $D_S = \emptyset$
- Si al llegar a una posición determinada descubrimos símbolos distintos, el conjunto D_S tendrá como elementos aquellas subexpresiones[7] de las expresiones de S que contengan esos símbolos.

B.3.2 Ejemplo. Sean los siguientes conjuntos de expresiones:

- $S = \{p(f(x), a), p(g(x), a)\}$, $D_S = \{f(x), g(x)\}$
- $S = \{p(f(x), a), p(f(y), a)\}$, $D_S = \{x, y\}$
- $S = \{p(f(x), x), p(f(x), a)\}$, $D_S = \{x, a\}$
- $S = \{p(f(x), x), p(f(x), x)\}$, $D_S = \emptyset$

B.3.3 Nota. Sea $S = \{e_i : i \in I\}$ un conjunto de expresiones y θ una sustitución. Denotaremos como $S\theta$ al conjunto de expresiones siguiente: $S\theta = \{e_i\theta : i \in I\}$.

ALGORITMO DE UNIFICACION
Consideremos un contador k

1. *Pongamos $k = 0$ y $\sigma_0 = \epsilon$*
2. *Si $S\sigma_k$ es un conjunto unitario entonces paramos; σ_k es un umg de S. En otro caso hallar el conjunto discordante D_k de $S\sigma_k$.*
3. *Si existe v y t en D_k tal que v es una variable que no ocurre en t entonces hacemos $\sigma_{k+1} = \sigma_k\{v/t\}$, aumentamos en uno el contador k, $k = k + 1$, y volvemos al paso 2. En otro caso paramos y concluimos que S no es unificable.*

La demostración del siguiente teorema puede encontrarse en [37].

[7] No hemos definido *"subexpresión"* en esta memoria, como tampoco otros conceptos propios de la lógica. Esta y otras definiciones auxiliares pueden encontrarse en [20].

Teorema. B.3.4 (Teorema de Unificación) *Sea S un conjunto de expresiones. Si S es unificable el algoritmo de unificación presentado termina y da como salida un umg de S. Si S no es unificable, el algoritmo termina y da como salida que S no es unificable.*

B.3.5 Nota. El algoritmo que aquí presentamos es muy ineficiente, de tipo exponencial en el peor de los casos. Como ya hemos comentado, Martelli y Montanari[38] presentaron un algoritmo lineal en 1976.

Como hemos apuntado, la teoría de unificación es muy extensa y está muy ramificada. Para más información pueden consultarse los trabajos de T. Nipkow[40] y Lassez, Maher y Marriot[34].

Indice

$\mu-$recursión, 56
árbol complejo, 89
árbol principal, 89
átomo, 98
átomo seleccionado, 47
ínfimo, 96
átomo cerrado, 100

dominio de una sustitución , 100
SLD-derivación con éxito, 39
SLD-derivación no restringida , 38

alcance, 99
aplicación continua, 28
aplicación monótona, 28
aridad, 97
asignación, 103

base de Herbrand, 21

cabeza de la cláusula, 19
camino, 89
cláusula de entrada, 37
cláusula de programa definido, 19
cláusula aplicable, 88
cláusula básica, 23
cláusula conjuntiva, 106
cláusula de Horn, 20
cláusula de programa, 70
cláusula disyuntiva, 106
cláusula unidad, 19
cláusula unidad negativa, 19
cláusula unidad positiva, 19
cláusula vacía, $\Box$, 20
cláusulas de entrada, 78
codominio de una sustitución , 100
completación de un programa, 72
composición de sustituciones, 107
composición, 56
condición de seguridad, 76
conjunto de éxito, 39
conjunto de fallo finito, 66
conjunto de R-éxito, 50
conjunto dirigido, 28
conjunto discordante, 114
consecuencia lógica, 104
constante, 97
cota inferior, 96
cota superior, 96
cuerpo de la cláusula, 19

definición, 70
definición completda, 71
definición de un símbplo de predicado, 20
dominio, 102

enlace, 101
enumeración de $U_{\mathbf{L}}$, 58
equivalencia lógica, 105
expresión, 99

fórmula, 98
fórmula abierta, 100
fórmula atómica, 98
fórmula cerrada, 100
favorable, 68
forma de Skolem, 25
forma normal conjuntiva, FNC, 106
forma normal disyuntiva, FND, 106
forma prenexa, 105
funciones básicas, 56

funciones de verdad, 103
funciones primitivas recursivas, 56
funciones recursivas, 57
función de Skolem, 25
fórmula consistente, 104
fórmula satisfacible, 104
fórmula válida, 104

HMC, Regla de Inferencia, 65

instancia, 107
instancia básica, 23
interpretación, 102
interpretación de Herbrand, 21
inviabilidad, 80

lenguaje de primer orden, 96
literal, 105
literal negativo, 105
literal positivo, 105
literal seleccionado, 85
literales complementarios, 105

matriz, 105
mayor punto fijo, Mpf, 28
menor generalización, 110
menor modelo de Herbrand, 23
menor punto fijo, mpf, 28
modelo, 104

objetivo definido, 20
objetivo derivado, 37, 76
objetivo normal, 70
operador consecuencia inmediata, 32
orden parcial, 95
ordinal recursivo, 34

permutación, 109
peso de un término, 59
pre-SLDNF- derivación, 91
pre-SLDNF-árbol, 89
pre-SLDNF-árbol inicial, 89
prefijo, 105
preinterpretación, 102
preinterpretación de Herbrand, 21
preorden, 95
procedimiento de SLD-refutación, 52
programa definido, 19
programa general, 70
programa normal, 70
programa sucesor, 58
pseudoderivación, 88

rama con éxito, 51
rama fallida, 51
rama infinita, 51
rango, 93
rango de una sustitución, 100
recursión, 56
regla de búsqueda, 52
regla de computación, 47
regla de computación segura, 85
regla de selección, 93
relación, 95
relevante, 112
resolvente, 37
respuesta, 35, 72
respuesta computada, 39, 78, 91
respuesta correcta, 35, 72
respuesta R-computada, 47, 85
retículo, 96
retículo completo, 96

símbolo proposicional, 97
semirretículo superior, 96
separación de variables, 37
SLD- conjunto de fallo finito, 67
SLD-árbol, 51
SLD-árbol finitamente fallido de rango k, 76
SLD-derivación, 37
SLD-derivación fallida, 39
SLD-derivación vía R, 47
SLD-refutación, 37
SLD-refutacion vía R, 47
SLD-refutación de rango k, 76
SLD-resolución, 36

SLD-árbol finitamente fallido, 67
SLD-árbol vía R, 52
SLDNF- derivación vía R, 85
SLDNF- refutación vía R, 85
SLDNF-árbol, 79, 91
SLDNF-derivación, 78, 91
SLDNF-refutación, 78
SLDNF-resolución, 75
SLDNF-árbol finitamente fallido, 78
SLDNF-árbol vía R, 85
subobjetivo, 20
subsunción, 108
subsunción estricta, 108
suc, relación binaria, 58
supremo, 96
sustitución, 100
sustitución idempotente, 107
sustitución identidad, 100
sustitución menos particular, 111
sustituciones equivalentes, 111
sustitución básica, 23, 101

término, 98
término más general, 107
término menos particular, 107
Términos equivalentes, 108
términos unificables, 112
teoría de la igualdad, 71
término cerrado, 99

unificador, 112
universo de Herbrand, 20

valores de verdad, 103
variable libre, 99
variable ligada, 99
variable sustituible, 102
variante, 108

Referencias

[1] ALONSO, J.A. *"Programación Lógica"* Notas del curso 93-94. Univ. de Sevilla. Trabajo no publicado. 1994

[2] ANDREKA, H. y I.NEMETI *The Generalizated Completeness of Horn Predicate Logic as a Programming Language* Acta Cybernetica, 4, 1 (1978) 3-10

[3] APT, K.R. *Logic Programming. Handbook of Theoretical Computer Science* Elsevier Science Publishers B.V., 1990

[4] APT, K.R. y M. BEZEM *Acyclic Programs* New Generation Comput. 29(3) 1991, pp. 335-363.

[5] APT, K.R. y K. DOETS *A new definition of SLDNF- resolution* J. Logic Programming 1994, 18 pp. 177-190.

[6] APT, K.R. y M.H. VAN EMDEN *Contributions to the Theory of Logic Programming* J. ACM 29, 3 (July 1982) 841-862

[7] APT, K.R. y D. PEDRESCHI *Proving Termination of General Prolog Programs* en *Proceedings of the International Conference on Theoretical Aspects of Computer Software, Lecture Notes in Computer Science 526, 265-289* T. Ito yA. Meyer (eds.) Springer-Verlag. Berlín 1991.

[8] BLAIR H.A. *The recursion-theoretic complexity of predicate logic as a programming language* Inform and Control 54 (1-2) (1982) 25-47

[9] BLAIR H.A. *Decidability in the Herbrand Base* Manuscript presented at the Workshop of Foundations of Deductive Databases and Logic Programming. Washington DC, 1986.

[10] CLARK, K.L. *Negation as Failure* in *Logic and Data Bases* Gallaire, H. and Minker, J. (eds), Plenum Press, New York, 1978, 293-322

[11] CLARK, K.L. *Predicate Logic as Computational Formalism* Research Report DOC 79/59, Department of Computing, Imperial College, London, 1979

[12] COLMERAUER A. ET ALS. *Un Systeme de Communication Homme-Machine en Francais* Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille. 1973.

[13] CHANG, C.L. y R.C.T. LEE *Symbolic Logic and Mechanical Theorem Proving* Academic Press New York 1973.

[14] CHURCH, A. y S.C. KLEENE *Formal definitions in the theory of ordinal numbers* Fund. Math. 28 (1937) 11-21

[15] DAS, S.K. *Deductive Databases and Logic Programming* Addison-Wesley Publishing Company, 1992.

[16] DAVIS M. *The Prehistory and Early History of Automated Deduction* en *Automation of Reasoning. Classical Papers of Computational Logic 1957-1966* Ed. Jörgsiekmann and Graham Wrightson. Springer Verlag 1983

[17] DAVIS M. *Eliminating the irrelevant from mechanical proofs.* Proc. Symp. Applied Math. XV (1963) 15- 30.

[18] DAVIS, M. y H. PUTNAM *A Computing Procedure for Quantification Theory* J.ACM 7 (1960) 201-215

[19] DELAHAYE,J.P. *Outils logiques pour l'Intelligence Artificielle* Ed. Eyrolles, 1988.

[20] EBBINGHAUS H.D. ET ALS. *Mathematical Logic* Springer-Verlag New York Inc. 1984.

[21] FERNÁNDEZ MARGARIT, A. *"Algebra III"* Notas del curso 91-92. Univ. de Sevilla.

[22] GENESERETH, M.R. y N.J. NILSSON *Logical Foundations of Artificial Intelligence* Morgan Kaufmann Publishers, Inc. 1987

[23] GILMORE, P.C. *A Proof Method for Quantification Theory* IBM J. Res. Develop. 4 (1960) 28-35

[24] HERBRAND, J. *"Investigations in Proof Theory"* en *From Frege to Gödel: A Source Book in Mathematical Logic 1879- 1931*, van Heijenoort, J. (ed.), Harvard University Press, Cambridge, Mass., 1967, 525-581.

[25] HILL, R. *LUSH-resolution and its completeness* DCL Memo 78, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1974

[26] HOGGER, C.R. *Essentials of Logic Programming* Clarendon Press Oxford, 1990

[27] HRBACEK, K. y T. JECH *Introduction to set theory* Marcel Dekker Inc., 1984.

[28] HUET, G. *Résolution d'equations dans les langages d'ordre* $1, 2, \ldots, \omega$ Tesis doctoral. Univ. París VII, 1976.

[29] JAFFAR, J., J.-L. LASSEZ y J.W. LLOYD *Completeness of the Negation as Failure Rule*, IJCAI-83, Karlsruhe 1986, 500-506.

[30] KOWALSKI R.A. *Predicate Logic as a Programming Language* en Proceedings IFIP'74, North Holland, Amsterdam 1974, 569-574.

[31] KOWALSKI, R.A. *The Relation Between Logic Programming and Logic Specification* in *Mathematical Logic and Programming Languages* Hoare, C.A.R. and J.C. Shepherdson (eds.) Prentice-Hall Englewood Cliffs, N.J. 1985,11-27

[32] KOWALSKI, R.A. y C.J. HOGGER *Logic Programming* en *Encyclopedia of Artificial Intelligence, Vol 1, pags. 544- 558* John Wiley and Sons, 1990.

[33] LASSEZ, J.L. y M.J.MAHER *Closures and Fairness in the Semantics of Programming Logic* Theoretical Computer Science 29 (1984), 167-184

[34] LASSEZ, J.L., M.J. MAHER y A. MARRIOT*Unification Revisited* en *Minker ed. Foundations of Deductive Databases and Logic Programming* MorganKaufmann, Los Alttos, CA. 1988

[35] LOVELAND D.W. *Automated theorem proving: A quarter–century review* Contemporary Mathematics Vol. 29 American Mathematical Society 1985.

[36] LLOYD J.W.*Foundatios of Logic Programming* Springer, Berlin, 1984

[37] LLOYD J.W.*Foundatios of Logic Programming* Springer, Berlin, 2nd ed., 1987

[38] MARTELLI, A. y U. MONTANARI *Unification in Linear Time and Space: A Structured Presentation* Nota Interna B76-16, Instituto di Elaborazione della Informazione, Pisa, 1976

[39] MARTELLI, M. y C.A. TRICOMI *A new SLD-Tree* Inform. Process. Lett. 43(2) 57-62 (1992)

[40] NIPKOW T.*Ecuational Reasoning* Research Report July, 9, 1992

[41] ODIFREDDI, P. *Classical Recursion Theory* Elsevier Science Publishers B.V., 1989.

[42] PRAWITZ, D. *An Improved Proof Procedure* Theoria 26 (1960) 102-139

[43] REITER, R. *On Closed World Data Bases* in Logic and Data Bases Gallaire, H. and Minker, J. (eds), Plenum Press, New York, 1978, 55-76

[44] ROBINSON, J.A. *A Machine-oriented Logic Based on the Resolution Principle* J. ACM 12, 1 (Jan, 1965), 23-41

[45] SEBELIK, J. y P.STEPANECK *Horn Clause Programs for Recursive Functions* in Logic Programming Clark, K.L. and Tarnlund S.A. (eds.) Academic Press New York, 1982, 324,340

[46] SHEPHERDSON, J.C. *Undecidability of Horn Clause Logic and Pure Prolog* unpublished manuscript, 1985.

[47] SONENBERG, E.A. y R.W. TOPOR *Computation in the Herbrand Universe* unpublished manuscript, 1986.

[48] TARNLUND, S.A *Horn Clause Computability* BIT 17,2 (1977), 215-226

[49] TARSKI, A. *A Lattice-theoretical Fixpoint Theorem and its Applications* Pacific J. Math. 5 (1955) 285-309

[50] VAN EMDEN, M. H. y R.A. KOWALSKI *The Semantics of Predicate Logic as a Programming Language* J. ACM 23,4 (Oct, 1976) 733-742.