Lógica y bases de datos

Parte I: Programación lógica

Matilde Celma Giménez
Laura Mota Herranz
Juan Carlos Casamayor Ródenas

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia (España)

1.INTRODUCCIÓN

La programación lógica aparece a principio de la década de los setenta como resultado directo de los primeros trabajos en prueba automática de teoremas y en inteligencia artificial. En 1972, Kowalski y Colmerauer apuntan la idea de que la lógica puede utilizarse como lenguaje de programación apareciendo el acrónimo PROLOG (PROgramming in LOGic). En este mismo año aparece el primer intérprete de este lenguaje.

La idea de que la lógica de primer orden, o un importante subconjunto de ella, pudiera ser utilizada como lenguaje de programación fue revolucionaria, ya que, hasta 1972, se había utilizado exclusivamente como lenguaje de especificación o declarativo en informática. Sin embargo Kowalski mostró que la lógica tiene también una interpretación procedural que la hace realmente efectiva como lenguaje de programación.

El objetivo de estos apuntes es introducir al estudiante en el campo de la programación lógica, para ello en el primer apartado se presenta la lógica de primer orden, base fundamental de la programación lógica; en el segundo, se estudian las distintas clases de programas lógicos que existen, programas definidos, normales y generales.

2. Lógica de Primer Orden

La lógica tiene como objetivo realizar deducciones sobre alguna realidad a partir del conocimiento que se tiene de la misma; para ello, es necesario disponer de un lenguaje no ambiguo que permita expresar este conocimiento así como representar las deducciones obtenidas.

El estudio de este lenguaje exige estudiar su sintaxis y su semántica. El aspecto sintáctico permite definir el conjunto de fórmulas bien formadas admitidas por la gramática del lenguaje formal; por otra parte, el aspecto semántico determina el significado de dichas fórmulas.

2.1. sintaxis de los lenguajes de 1er orden

Un lenguaje de 1er orden L es un par (A,F) donde:

• A es el alfabeto de símbolos (distinto para diferentes lenguajes). Con estos símbolos construiremos las fórmulas del lenguaje.

• F es conjunto de fórmulas bien formadas. Las reglas de construcción de estas fórmulas permanecen constantes sea cual sea el alfabeto.

Alfabeto (A)

El alfabeto de un lenguaje de 1er orden está formado por los siguientes conjuntos de símbolos:

• Variables: representadas por las últimas letras del abecedario (x, y, z…)

• Constantes: representadas por las primeras letras del abecedario (a, b, c...)

• Funciones: se representan también por letras minúsculas (f, g, h,…). Cada uno de estos símbolos tiene asociado un entero mayor que cero que indica el número de argumentos de la función y que se denomina aridad. (f(.,.) representa una función de aridad 2).

• Predicados: Cada uno de estos símbolos tiene asociado un entero mayor que cero que indica el número de argumentos del predicado y que se denomina aridad. (p(.,.) representa un predicado de aridad 2).

• Signos de puntuación: signos utilizados para la construcción de los términos y las fórmulas: **(**,), **,**

• Conectivas lógicas: representan los operadores lógicos clásicos, como conjunción ($\wedge$), disyunción ($\vee$), negación ($\neg$), implicación ($\rightarrow$), coimplicación ($\leftrightarrow$).

• Cuantificadores: los dos cuantificadores usuales, el universal ($\forall$) y el existencial ($\exists$).

Evidentemente, el conjunto de símbolos puede ser infinito.

Las constantes individuales se incluyen para que en el lenguaje haya fórmulas que puedan interpretarse como enunciados acerca de cosas particulares.

Conjunto de fórmulas (F)

Primero es necesario definir el concepto de *término*. Los términos son aquellas expresiones del lenguaje formal que se interpretan como objetos, es decir las "cosas" a las que se aplican las funciones, las "cosas" que tienen propiedades, las "cosas" acerca de las cuales se realizan aseveraciones.

Un término se define como sigue:

- cada variable o constante del alfabeto A es un término, y
- si f es una función n-aria y $t_1, t_2,\ldots, t_n$ son términos, entonces $f(t_1, t_2,\ldots, t_n)$ es un término.

Un término es *base* si no contiene ningún símbolo de variable.

Ahora ya podemos definir las fórmulas bien formadas del lenguaje. Primero empezamos por las *fórmulas atómicas o átomos* que son las expresiones más sencillas del lenguaje que son interpretables como aseveraciones, como por ejemplo que cierto objeto verifica cierta propiedad.

Un átomo se define como sigue:

- Si p es un símbolo de predicado n-ario de A y $t_1, t_2,\ldots, t_n$ son términos, entonces $p(t_1, t_2,\ldots, t_n)$ es una fórmula atómica.

Se dice que $p(t_1, t_2,\ldots, t_n)$ es un *átomo base* (totalmente instanciado) si y sólo si $t_1, t_2,\ldots, t_n$ son términos base.

Veamos ahora, las reglas que definen una fórmula bien formada (fbf):

1. Todo átomo es una fbf.

2. Si F_1 y F_2 son fbfs entonces:
   - (F)
   - $\neg F_1$
   - $F_1 \wedge F_2$
   - $F_1 \vee F_2$
   - $F_1 \rightarrow F_2$
   - $F_1 \leftrightarrow F_2$

   también son fbfs.

3. Si F es una fbf y x una variable entonces:

- $\exists x\ F$
- $\forall x\ F$

también son fbfs.

4. Ninguna otra expresión es una fbf.

Ocurrencia de una variable

Cualquier aparición de una variable en una fórmula es una *ocurrencia* de esa variable. En una fórmula, una variable puede ocurrir de dos formas distintas: libre y ligada. Para ver estas dos formas previamente necesitamos definir el concepto de *alcance de un cuantificador*.

Alcance de un cuantificador

El alcance de $\forall x$ en la fórmula $\forall x\ F$ es F. Más en general, si $\forall x\ F$ aparece como subfórmula de una fbf G, se dice que el alcance del cuantificador universal en G es F. Igualmente, el alcance de $\exists x$ en la fórmula $\exists x\ F$ es F (si $\exists x\ F$ aparece como subfórmula de una fbf G, se dice que el alcance del cuantificador existencial en G es F).

Intuitivamente, el alcance de un cuantificador ($\forall x$ o $\exists x$) es la primera fórmula bien formada que hay a su derecha.

Ocurrencia ligada de una variable

Una ocurrencia de una variable en una fórmula es ligada si:

a) es una ocurrencia sobre la que actúa un cuantificador, o

b) es una ocurrencia dentro del alcance de un cuantificador que actúa sobre una ocurrencia del mismo símbolo de variable.

Ocurrencia libre de una variable

Cualquier ocurrencia de variable que no sea ligada es libre.

Fórmula cerrada

Una fórmula es cerrada si no posee ocurrencias libres de variables.

Fórmula abierta

Una fórmula es abierta si posee al menos una ocurrencia libre de variable.

2.2. SEMÁNTICA DE LOS LENGUAJES DE 1er ORDEN: INTERPRETACIÓN Y MODELO

Un lenguaje de primer orden nos permite "hablar" sobre un universo de discurso. En él:

• los términos denotan objetos (individuos) de ese universo de discurso;

• los predicados denotan propiedades sobre los objetos del universo de discurso; y

• las fórmulas bien formadas son enunciados o sentencias sobre el universo.

Al dotar de semántica a un lenguaje de 1er orden se persiguen los siguientes objetivos:

• dar significado a cada uno de los símbolos del alfabeto A; y

• poder conocer el valor de verdad de cualquier fórmula bien formada de F.

Para ello tenemos que construir un contexto donde se puedan evaluar las fórmulas del lenguaje. A este contexto se le denomina *interpretación*.

Interpretación

Una interpretación I de un lenguaje de 1er orden L=(A,F) es un triplete (D, K, E), donde:

• D es un conjunto no vacío, denominado *dominio de I*. En este dominio las variables toman valor y por otra parte define el rango de variación de los cuantificadores.

• K es un conjunto de aplicaciones que permiten asignar un elemento del dominio a todo término del lenguaje L; K define lo siguiente:

•• una aplicación del conjunto de constantes de A sobre D

•• la asignación de una función de D^n sobre D a cada símbolo de función n-ario.

• E es la asignación de una relación definida sobre D^n a cada símbolo de predicado. Si p es un predicado n-ario, entonces E(p) se le denomina *extensión de p en la interpretación I* ($E(p) \subseteq D^n$)**.**

Valor de verdad de una fórmula en una interpretación

Una vez se ha definido una interpretación de un cierto lenguaje de 1er orden, ya podemos preguntarnos cuáles son los valores que tomará una fórmula al evaluarla en esta interpretación. A este valor se le denomina *valor de verdad*.

- Valor de verdad de una fórmula cerrada en una interpretación

El valor de verdad de una fórmula cerrada G en una interpretación puede ser o *cierto en I*, o *falso en I* y se define recursivamente como sigue:

1. Si G es una fórmula atómica, $G = p(t_1,\ldots, t_n)$ entonces G es cierta en I si y sólo si:

$$<K(t_1), \ldots, K(t_n)> \in E(p)$$

y falsa en caso contrario.

2. Si G contiene alguna conectiva lógica, se evalúa de acuerdo a las siguientes tablas:

$\mathbf{F_1}$	$\mathbf{G = \neg F_1}$
cierto	falso
falso	cierto

$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{G = F_1 \wedge F_2}$	$\mathbf{G = F_1 \vee F_2}$	$\mathbf{G = F_1 \rightarrow F_2}$	$\mathbf{G = F_1 \leftrightarrow F_2}$
cierto	cierto	cierto	cierto	cierto	cierto
cierto	falso	falso	cierto	falso	falso
falso	cierto	falso	cierto	cierto	falso
falso	falso	falso	falso	cierto	cierto

3. Si G es una fórmula universalmente cuantificada de la forma $G = \forall x\ F_1$, entonces G es cierta en I si y sólo si para cualquier valor del dominio de la interpretación ($\forall\ d \in D$) la fórmula cerrada que resulta al sustituir todas las ocurrencias de x por d en F_1 ($F_1(x/d)$) es cierta en I.

4. Si G es una fórmula existencialmente cuantificada de la forma $G = \exists x\ F_1$, entonces G es cierta en I si y sólo si para algún valor del dominio de la interpretación ($\exists d \in D$) la fórmula cerrada que resulta al sustituir todas las ocurrencias de x en F_1 por d ($F_1(x/d)$) es cierta en I.

5. Si G es una fórmula de la forma G =(F) entonces G se evalúa al valor de verdad de F.

Si G es una fórmula cierta (resp. falsa) en I, esto se expresar de la siguiente manera:

$$\models_I G \ \text{(resp. } \not\models_I G)$$

Una fórmula cerrada G se dice que es *válida* si y sólo si para toda interpretación I se cumple que $\models_I G$. Esto se denota así:

$$\models \mathbf{G}$$

- Valor de verdad de una fórmula abierta en una interpretación I

Dada una fórmula abierta con *n* variables libres, $F(x_1, \ldots, x_n)$, se denomina *asignación de F,* ρ *,* a un conjunto de pares x_i/d_i donde x_i es una variable libre de F y d_i un elemento del dominio de la interpretación:

$$\rho=\{x_1/d_1,\ldots,x_n/d_n\}$$

Todas las ρ_j $(1 \leq j \leq m)$, tales que $\models_I F(x_1/d_{1j},\ldots,x_n/d_{nj})$ definen un conjunto de tuplas

$$\{<d_{11}, \ldots, d_{n1},$$
$$<d_{12} \ldots, d_{n2},$$
$$\ldots$$
$$<d_{1m} \ldots, d_{nm}\}$$

Entonces, la fórmula abierta $F(x_1,\ldots, x_n)$ se evalúa en una interpretación I de la forma siguiente:

a) si $\forall$ ρ_j posible, $\rho_j = \{x_1/d_{1j},\ldots,x_n/d_{nj}\}$ se cumple $\models_I F(x_1/d_{1j},\ldots,x_n/d_{nj})$, entonces se dice que *F es cierta en I.*

b) si $\neg\exists\rho_j$, $\rho_j = \{x_1/d_{1j},\ldots,x_n/d_{nj}\}$ tal que $\models_I F(x_1/d_{1j},\ldots,x_n/d_{nj})$, entonces se dice que *F es falsa en I.*

Por tanto, una fórmula abierta puede no ser ni cierta ni falsa en una interpretación.

Modelo de un conjunto de fórmulas

Dada una interpretación I y una fórmula F, *I es modelo de F* si y sólo si $\models_I F$.

Dada una interpretación I y un conjunto de fórmulas CF, I es modelo de CF si y sólo si:

$$\forall F \in CF, \models_I F.$$

Se dice que un conjunto de fórmulas CF es:

- *satisfactible* si existe alguna interpretación que sea modelo de CF;
- *válido* si cualquier interpretación es modelo de CF;
- *insatisfactible* si ninguna interpretación es modelo de CF.

Consecuencia lógica de un conjunto de fórmulas

Sea CF un conjunto de fórmulas cerradas y sea G una fórmula. G es *consecuencia lógica* de CF si y sólo si todo modelo M de CF es modelo de G. Esto se denota de la siguiente manera:

$$CF \models G$$

Sea CF un conjunto de fórmulas se puede demostrar fácilmente, la siguiente equivalencia:

CF ⊨ G si y sólo si CF ∪ {¬ G} es insatisfactible

Uno de los objetivos más interesantes perseguidos por la lógica es el demostrar que una fórmula G es consecuencia lógica de un conjunto también de fórmulas CF (es decir *deducir G a partir de CF*), esto sin embargo significaría comprobar que todo modelo de CF también es modelo de G, lo que es un problema prácticamente imposible de abordar. Esta situación, ha llevado a desarrollar procedimientos automáticos de demostración que permitan afirmar que G es consecuencia lógica de CF sin explorar para ello todos los modelos de CF. La idea es encontrar un procedimiento sencillo que nos permita construir una argumentación paso a paso sabiendo que cada paso es válido y que nos permita concluir que una fórmula es consecuencia lógica de un conjunto de fórmulas. Para investigar este tipo de problemas vamos a introducir el concepto de sistema formal.

2.3. SISTEMA FORMAL DE DEDUCCIÓN EN LÓGICA DE 1ER ORDEN

Dado un lenguaje de 1er orden L, un sistema formal, S_F, se define por medio de:

- un conjunto de fórmulas bien formadas, llamadas *axiomas* (sería más adecuado hablar de *esquemas* de fórmulas bien formadas); y

- un conjunto finito de *reglas de inferencia* que permiten deducir una fórmula bien formada, tal como G, como consecuencia directa de un conjunto finito de fórmulas bien formadas como $G_1, G_2, \ldots$

Con un sistema formal se pueden construir deducciones por medio de sucesivas aplicaciones de las reglas de inferencia a partir de los axiomas, para ello hay que tener en cuenta que en un sistema formal los símbolos carecen de significado, y al manipularlos no se debe presuponer nada respecto a sus propiedades, salvo las que se especifiquen en el sistema formal.

Demostración en S_F

Una demostración en S_F es una sucesión finita $A_1, \ldots, A_n$ de fórmulas bien formadas de L, tal que $\forall i\ (1 \leq i \leq n)$ o:

- A_i es un axioma de S_F, o

- se deduce de los anteriores miembros de la sucesión aplicando alguna de las reglas de inferencia.

Teorema de S_F

Una fórmula G es un teorema de S_F si es el último miembro de una demostración en S_F. Cuando G es un teorema de S_F se representa:

$$|\!-\!-_{S_F} G$$

Podemos ahora relacionar el concepto de consecuencia lógica con el de deducción de la forma siguiente: "se dice que el sistema formal S_F es *adecuado* (correcto y completo), si se cumple lo siguiente:

G es *válida* ($|\!=$ G) si y sólo si $|\!-\!-_{S_F}$ G

Con este procedimiento es posible demostrar si una fórmula es válida, pero usualmente nuestro interés se centra en deducir "cosas" de un conocimiento dado que está representado por un conjunto de fórmulas por ello introducimos el concepto de teoría.

Teoría de 1er orden

Una teoría de 1er orden T, es un conjunto de fbfs de L. Una *deducción a partir de T en* S_F es una sucesión similar a la de una demostración, en la que además se pueden incluir elementos de T.

Teorema de una teoría

Una fórmula G es teorema de una teoría T en S_F si G es el último miembro de una deducción a partir de T en S_F. Cuando G es teorema de T en S_F se representa:

$$T |\!-\!-_{S_F} G$$

Si el sistema formal S_F es adecuado (**correcto y completo**), el concepto de consecuencia lógica de un conjunto de fórmulas es análogo a ser teorema de una teoría que tiene por axiomas ese conjunto de fórmulas. O sea, si CF un conjunto de fórmulas cerradas (o teoría) y G una fórmula cerrada, entonces:

CF $|\!=$ G si y sólo si CF $|\!-\!-_{S_F}$ G

3.PROGRAMACIÓN LÓGICA

Los sistemas de programación lógica en que estamos interesados utilizan como única regla de inferencia el **Principio de Resolución de Robinson** o alguna basada en este principio**.**

Antes de presentar esta regla veremos algunos conceptos previos.

Concepto de Literal

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.

Sea A un átomo y L un literal tal que L = ¬A, entonces se dice que A y L son *complementarios*.

Concepto de Cláusula

Una cláusula es una fórmula de la forma:

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

donde cada L_i es un literal y $x_1,\ldots, x_n$ son todas las variables que aparecen en $L_1 \vee \ldots \vee L_m$.

Ya que las cláusulas son muy usuales en programación lógica, es conveniente adoptar una notación especial. Sea C_1 la siguiente cláusula:

$$C_1: \ \forall x_1 \ldots \forall x_n (A_1 \vee \ldots \vee A_k \vee \neg B_1 \vee \ldots \vee \neg B_s)$$

donde $A_1, \ldots, A_k, B_1, \ldots, B_s$ son átomos y $x_1, \ldots, x_n$ todas las variables que ocurren en esos átomos, a partir de ahora la denotaremos por:

$$C_1: A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_s$$

donde se asume la cuantificación universal de todas las variables.

Comentario: hay que observar que el símbolo ← no pertenece al alfabeto definido en el apartado 2.1. Realmente la fórmula bien formada para representar la cláusula C_1 (asumiendo la cuantificación universal) sería:

$$B_1 \wedge \ldots \wedge B_s \rightarrow (A_1 \vee \ldots \vee A_k)$$

sin embargo, se suele invertir la flecha (aunque la semántica sigue siendo la misma) para aproximarse a la sintaxis del prolog.

Concepto de Sustitución

Una *sustitución* θ es un conjunto finito de la forma $\{v_1/t_1,\ldots, v_n/t_n\}$ donde cada v_i es una variable, cada t_i un término distinto de v_i y tal que las $v_1,\ldots, v_n$ son diferentes. Cada elemento v_i/t_i se denomina *enlace* de v_i. θ es una sustitución base si todos los t_i son términos base.

Sea E una expresión bien formada del lenguaje objeto (es decir E es un término, un literal o una fórmula) y sea $\theta = \{v_1/t_1,\ldots, v_n/t_n\}$ una sustitución entonces $E\theta$ es la expresión obtenida a partir de E al sustituir simultáneamente cada ocurrencia de la variable v_i por el término t_i. $E\theta$ se denomina *instancia de E por* θ.

Sean $\theta = \{u_1/s_1,\ldots, u_m/s_m\}$ y $\delta = \{v_1/t_1,\ldots, v_n/t_n\}$ dos sustituciones, la *composición* $\theta\delta$ de θ y δ es la sustitución obtenida a partir del siguiente conjunto:

$$\{u_1/s_1\delta ,\ldots, u_m/s_m\delta, v_1/t_1,\ldots, v_n/t_n\}$$

eliminando cualquier enlace $u_i/s_i\delta$ tal que $u_i = s_i\delta$ y cualquier par v_j/t_i tal que $v_j \in \{u_1,\ldots,u_m\}$.

La sustitución, ε, denotada por el conjunto vacío se denomina sustitución *identidad* ya que para cualquier expresión E se cumple: $E\varepsilon = E$.

Concepto de Unificador

Un *unificador* de un conjunto de expresiones $\{E_1,\ldots,E_n\}$ es una sustitución θ que unifica el conjunto es decir, que hace cualquier expresión del conjunto sintácticamente igual a cualquier otra expresión del conjunto: $E_1\theta = E_2\theta =\ldots = E_n\theta$

Se dice que dos expresiones E_1 y E_2 son unificables si existe un unificador de $\{E_1, E_2\}$

Concepto de Unificador Más General (mgu)

Un unificador θ de un conjunto de expresiones E, se dice que es un *unificador más general* de E si para cualquier otro unificador δ de E existe una sustitución γ tal que $\delta = \theta\gamma$.

3.1. Principio de Resolución de Robinson

El Principio de Resolución de Robinson es un método de demostración aplicable a cláusulas. La forma de demostrar que G es un teorema de la teoría definida por un conjunto de axiomas CF (o lo que es equivalente que es consecuencia lógica de CF), consiste en añadir la negación de G a CF y realizar inferencias hasta llegar a una contradicción (por este motivo se dice que es un método de refutación). En el contexto de las cláusulas, la contradicción viene representada por la cláusula vacía y la denotaremos por el símbolo $\square$.

La regla de inferencia se puede resumir de la forma siguiente: para cualquier par de cláusulas C_1 y C_2, si existe un literal L_1 en C_1 unificable con el complementario de un literal L_2 de C_2, entonces se aplica a C_1 y C_2 la sustitución que los unifica, se eliminan los literales L_1 y L_2 de las

cláusulas y se construye una disyunción con los restos de éstas. La nueva cláusula así obtenida, denominada *resolvente* de C_1 y C_2, es añadida al conjunto original de cláusulas. Si durante la aplicación del método se llega a tener dos cláusulas cuyo resolvente es la cláusula vacía se habrá alcanzado una contradicción.

El Principio de Resolución de Robinson tiene las propiedades de correctitud y completitud que se enuncian como sigue.

Sea C un conjunto de cláusulas:

• **Correcto:** "Si al aplicar a C el proceso de resolución cierto número de veces se deriva la cláusula vacía, entonces C es insatisfactible".

• **Completo:** "Si C es insatisfactible, la cláusula vacía se derivará al aplicar resolución un número finito de veces a C".

3.2. PROGRAMAS LÓGICOS

Un programa lógico general es un conjunto de fórmulas (llamadas sentencias de programa) con la forma siguiente:

$$\mathbf{A \leftarrow F}$$

donde A es un átomo (*cabeza*) y F es una fórmula bien formada (*cuerpo*). Todas las variables libres se suponen cuantificadas universalmente.

Un programa lógico debe ser visto como una teoría de primer orden donde los axiomas son las sentencias de programa y que tiene una única regla de inferencia basada, como veremos a continuación, en el Principio de Resolución de Robinson antes presentado.

Por ello, si se quiere demostrar que la fórmula G: $\exists x_1,\ldots,\exists x_r$ H es consecuencia lógica (o teorema) de un programa P, entonces se añade su negación ($\neg$G) a los axiomas y se intenta derivar la contradicción. Es importante destacar, que desde el punto de vista de la demostración de teoremas, el mayor interés reside precisamente en demostrar consecuencias lógicas, sin embargo desde el punto de vista de la programación estamos más interesados en los enlaces que se realizan para las variables $x_1,\ldots,x_r$, ya que estos enlaces representan precisamente la salida del programa.

A las fórmulas que se quieran demostrar a partir de un programa lógico las llamaremos *objetivos*.

Procedimientos de Resolución basados en Principio de Resolución de Robinson

Existen dos procedimientos de resolución basados en el Principio de Resolución de Robinson. Estos son:

• SLD: procedimiento de resolución Lineal con función de Selección para cláusulas Definidas. Este procedimiento es aplicable en programas definidos.

• SLDNF: procedimiento de resolución Lineal con función de Selección, aumentado con la Negación como Fallo. Este procedimiento es aplicable en programas normales.

Teniendo en cuenta que si usamos cualquier procedimiento de resolución basado en el de Robinson, vamos a realizar refutaciones, es conveniente cambiar el aspecto de los objetivos. Veámoslo:

Sea P un programa lógico y G un cierto objetivo:

$$G: \exists x_1,\ldots, \exists x_n\, F(x_1,\ldots,x_n)$$

se desea probar que:

$$P \models \exists x_1,\ldots, \exists x_n\, F(x_1,\ldots,x_n)$$

Para aplicar el Principio de Resolución de Robinson hay que refutar, por tanto, hay que probar que:

$$P \cup \{\neg\exists x_1,\ldots, \exists x_n\, F(x_1,\ldots,x_n)\}$$

es insatisfactible. Cojamos esta cláusula añadida (el objetivo negado) y cambiémosle su forma:

$$\neg\exists x_1,\ldots, \exists x_n\, F(x_1,\ldots,x_n) \equiv \forall x_1,\ldots, \forall x_n\, \neg F(x_1,\ldots,x_n)$$

y transformado este resultado:

$\forall x_1,\ldots, \forall x_n\, \neg F(x_1,\ldots,x_n) \equiv \neg F(x_1,\ldots,x_n)$ (asumiendo la cuantificación universal de $x_1,\ldots,x_n$)

$\neg F(x_1,\ldots,x_n) \equiv \mathbf{\leftarrow F(x_1,\ldots,x_n)}$

Comentario: hay que observar que la expresión ← F no es una fórmula bien formada tal como se definió en el apartado 2.1. De nuevo para aproximarnos a la notación del prolog (que por otra parte resulta muy cómoda) se realiza las siguientes transformaciones y asunciones:

$\neg F \equiv \neg F \vee \text{falso} \equiv \text{falso} \leftarrow F$, expresión que representaremos por $\leftarrow F$

Esta será la forma que tomarán los objetivos lanzados a un programa lógico, ($\leftarrow F(x_1,\ldots,x_n)$), aunque lo que se desea es saber si es deducible $\exists x_1,\ldots, \exists x_n\, F(x_1,\ldots,x_n)$ y como ya se ha mencionado antes, el máximo interés reside en conocer que valores pueden tomar las variables $x_1,\ldots,x_n$ manera que al sustituir en F las variables por esos valores, la fórmula cerrada que resulta es consecuencia lógica del programa lógico.

3.2.1. Programas definidos

Un programa *definido* es un programa lógico cuyas sentencias son *sentencias definidas*

Sentencia Definidas

Tienen la siguiente forma:

$$\mathbf{A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n}$$

donde A, B_1, B_2, …, B_n son átomos (en un programa definido no aparecen literales negativos).

Objetivo Definido

Un objetivo definido tiene la siguiente forma:

$$\mathbf{\leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_m}$$

donde A_1, A_2, …, A_m son átomos.

Veamos unos conceptos previos necesarios para definir el procedimiento de resolución SLD:

Respuesta

Sea P un programa definido y G un objetivo definido. Una respuesta θ para $P \cup \{G\}$ es una *sustitución* de las variables de G.

Respuesta Correcta

Sea P un programa definido, $\leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n$ un objetivo definido y θ una respuesta para $P \cup \{\leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n\}$, entonces θ es una respuesta correcta para $P \cup \{\leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n\}$ si y sólo si $P \models \overline{\forall}((A_1 \wedge A_2 \wedge \ldots \wedge A_n)\,\theta)$.

($\overline{\forall}$ representa el cierre universal, es decir a cuantificación universal de todas las variables que aparezcan libres en la fórmula que precede.)

Ejemplos

• Sea P_1 el siguiente programa definido:

$p(x, y) \leftarrow q(y)$
$q(a) \leftarrow$

1) Objetivo definido: $\leftarrow p(z, z)$

Respuesta Correcta $\theta = \{z/a\}$ ya que $P_1 \models p(a, a)$

2) Objetivo definido: $\leftarrow q(a)$

Respuesta Correcta $\theta = \{\}$ ya que $P_1 \models q(a)$

3) Objetivo definido: $\leftarrow q(b)$

Respuesta Correcta = no existe ya que $P_1 \cup \{\leftarrow q(b)\}$ es satisfactible

- Sea P_2 el siguiente programa definido:

$p(a) \leftarrow$
$p(b) \leftarrow$
$q(a, b) \leftarrow$
$q(a, z) \leftarrow$

4) Objetivo Definido: $\leftarrow p(x) \wedge q(x, y)$

Respuesta correcta $\theta_1 = \{x/a, y/b\}$, ya que $P_2 \models p(a) \wedge (q(a,b)$
Respuesta correcta $\theta_2 = \{x/a, y/z\}$, ya que $P_2 \models \forall z(p(a) \wedge (q(a,z))$

Resolvente de un Objetivo Definido y una Sentencia Definida

Sea G el siguiente objetivo definido $G = \leftarrow A_1 \wedge \ldots \wedge A_k \wedge \ldots \wedge A_n$ y sea S una *variante* de una sentencia definida (obtenida a base de renombrar ciertas variables, ya que es necesario si aparecen repetidas en G y S):

$S = A \leftarrow B_1 \wedge \ldots \wedge B_q$

entonces *G' se deriva de G y S usando el mgu* θ si se cumplen las siguientes condiciones:

1. A_k es un átomo de G seleccionado por una *regla de computación*;

2. θ es el mgu de A_k y A $(\theta = mgu(A_k, A))$;

3. $G' = \leftarrow (A_1 \wedge \ldots \wedge A_{k-1} \wedge B_1 \wedge \ldots \wedge B_q \wedge A_{k+1} \wedge \ldots \wedge A_n)\theta$

a G´ se le denomina *resolvente* de G y S.

3.2.1.1. Derivación SLD

Sea P un programa definido y G un objetivo definido. Una derivación SLD para $P \cup \{G\}$ consiste en:

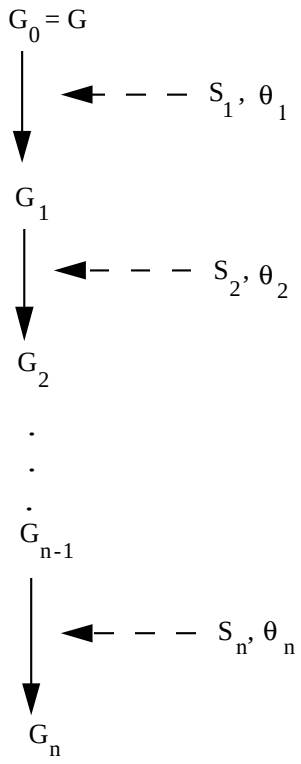
- una secuencia de objetivos $G_0, G_1, \ldots$ donde $G_0 = G$;

• una secuencia de variantes de sentencias de P: S_1, S_2, …; y

• una secuencia de mgu θ_1, θ_2, …

tal que G_{i+1} se deriva de G_i y S_{i+1} usando θ_{i+1} (G_{i+1} es el resolvente de G_i y S_{i+1}).

Esto se puede ver de una forma más gráfica como se muestra en la siguiente figura.



Tipos de Derivaciones SLD

Veamos cuales son las posibles derivaciones que se pueden dar:

- *Infinita*: en cualquier objetivo G_n el átomo seleccionado A_k siempre es unificable con alguna sentencia S_{n+1}.

- *Finita*: la derivación termina en un número finito de pasos (las secuencias son finitas). Supongamos que la longitud es n, entonces se da alguno de estos dos casos:

• Fracasa: el átomo seleccionado A_k en el último objetivo G_n no es unificable con ninguna sentencia del programa.

• Exito: el último objetivo $G_n = \square$.

Refutación SLD

Es una derivación SLD que tiene éxito. En este caso hemos podido demostrar que el conjunto de cláusulas resultado de unir la cláusula proveniente del objetivo (realmente, la negación del objetivo) y las sentencias del programa es insatisfactible.

Respuesta Computada θ para P ∪{G}

Una respuesta computada θ para P ∪ {G} es la composición de $\theta_1, \ldots, \theta_n$, restringida a las variables de G, donde $\theta_1, \ldots, \theta_n$ es la secuencia de mgu´s de una refutación para P ∪ {G}.

3.2.1.2.Independencia de la Regla de Computación

Esta propiedad asegura que nuestras computaciones no dependen de cómo seleccionamos los átomos en nuestros objetivos. Se formula de la siguiente forma:

"Dado un programa definido P y un objetivo G, tal que existe una refutación para P∪{G} usando un regla de computación C, entonces existirá también una refutación para P ∪ {G} usando cualquier otra regla de computación C´."

3.2.1.3.Propiedades del procedimiento de resolución SLD

• **Correcto**: toda respuesta computada para P ∪ {G} es una respuesta correcta para P ∪ {G}. Esto asegura que todo lo que computemos va a ser consecuencia lógica.

• **Completo**: para toda respuesta correcta θ para P ∪ {G} existe una respuesta computada α para P ∪ {G} y una sustitución β tal que:

$$\theta = \alpha\beta$$

Esto nos asegura que todo lo que es consecuencia lógica es posible computarlo.

Ejemplo

Sea P el siguiente programa definido:

$p(x, y) \leftarrow q(y)$
$q(a) \leftarrow$

y sea el siguiente objetivo $G = \leftarrow p(z_1, z_2)$

Tomemos una refutación SLD:

- secuencia de objetivos :
  $G_0 = \leftarrow p(z_1, z_2)$, $G_1 = \leftarrow q(y)$, $G_3 = \square$

- secuencia de sentencias:
  $S_1 = p(x, y) \leftarrow q(y)$, $S_2 = q(a) \leftarrow$

- secuencia de mgu:
  $\theta_1 = \{z_1/x, z_2/y\}$, $\theta_2 = \{y/a\}$

la respuesta computada de esta refutación es:

$$\theta = \theta_1.\theta_2 \equiv \{z_1/x, z_2/a\}$$

que también es un respuesta correcta.

Tomemos ahora otra respuesta correcta $\alpha = \{z_1/a, z_2/a\}$, es posible encontrar una sustitución $\beta = \{x/a\}$ tal que: $\alpha = \theta\beta$

3.2.1.4.Árbol SLD

Dado un programa definido P y un objetivo definido G, el conjunto de todas las derivaciones posibles para $P \cup \{G\}$ se puede representar en un árbol definido de la siguiente forma:

a) cada nodo es un objetivo;

b) el nodo raíz es G;

c) si $\leftarrow A_1 \wedge \ldots \wedge A_k \wedge \ldots \wedge A_n$ $(n \geq 1)$ es un nodo del árbol y si A_k es el átomo seleccionado por una regla de computación, entonces para cada sentencia S de P de la forma:

$$A \leftarrow B_1 \wedge \ldots \wedge B_q \quad \text{tal que } \theta = \text{mgu}(A, A_k)$$

el nodo tendrá un hijo con la forma:

$$\leftarrow (A_1 \wedge \ldots \wedge A_{k-1} \wedge B_1 \wedge \ldots \wedge B_q \wedge A_{k+1} \wedge \ldots \wedge A_n)\theta$$

d) los nodos que son la cláusula vacía ($\square$) no tienen hijos.

Árbol SLD fallado finitamente

Sea P un programa definido y G un objetivo definido. Un árbol SLD fallado finitamente para $P \cup \{G\}$ es un árbol finito y que no tiene ramas de éxito

Equivalencias entre conceptos

- derivación SLD ≡ rama del árbol SLD

- derivación SLD con éxito ≡ rama del árbol SLD que termina en □
  (refutación) (rama de éxito)

- derivación SLD fallada ≡ rama del árbol SLD finita que no termina □
  (rama fallada)

- derivación SLD infinita ≡ rama del árbol SLD infinita

Por tanto, podemos afirmar que:

Conjunto de refutaciones SLD ≡ Conjunto de ramas de éxito del árbol SLD

Construcción de los árboles SLD: Regla de Computación

Cada regla de computación da lugar a un árbol SLD distinto (diferente geometría). Aún así, debido a la propiedad del SLD de Independencia de la Regla de Computación que asegura que toda refutación se puede computar con cualquier regla, es posible afirmar que el *conjunto de ramas de éxito* para distintos árboles SLD (construidos con diferentes reglas de computación) *es siempre el mismo.*

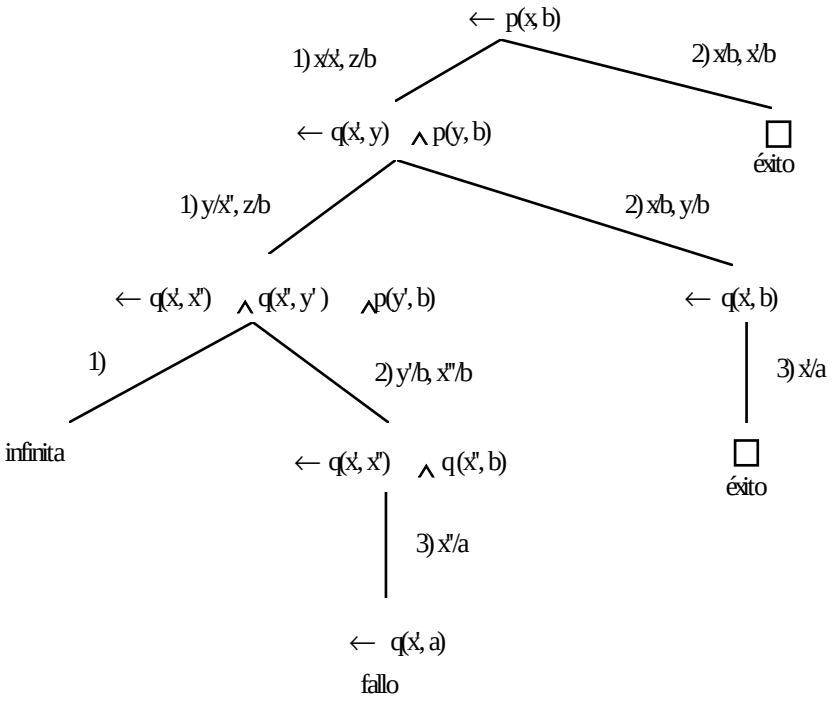
Ejemplo

Sea P el siguiente programa definido:

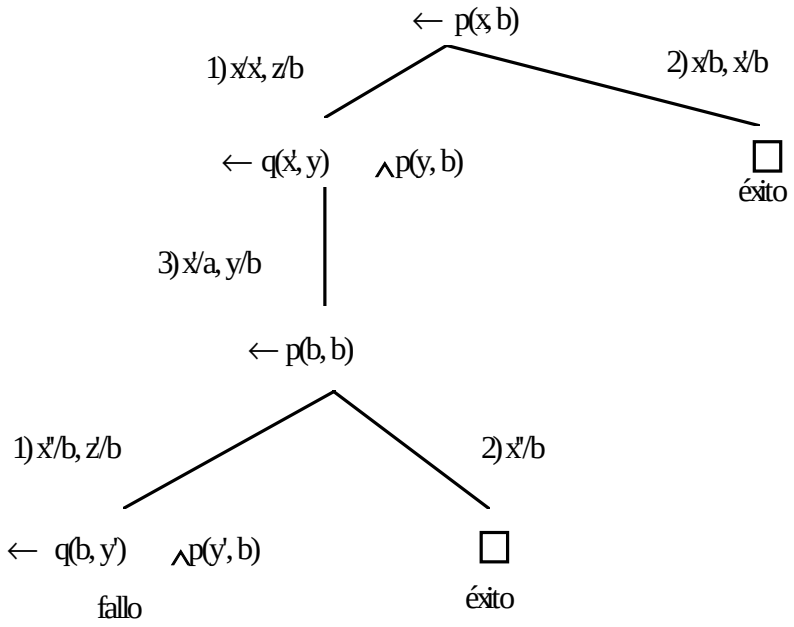
$p(x, z) \leftarrow q(x,y) \wedge p(y,z)$	1
$p(x, x) \leftarrow$	2
$q(a, b) \leftarrow$	3

y sea G el siguiente objetivo $\leftarrow p(x,b)$. Vamos a construir dos árboles con diferentes reglas de computación

Regla de Computación = 1er átomo de la derecha



Regla de Computación: 1er átomo de la izquierda



En los árboles se puede observar que el conjunto de ramas de éxito es el mismo en los dos.

Recorrido del árbol SLD: Regla de Búsqueda

La regla de búsqueda especifica la forma de recorrer el árbol SLD para encontrar las ramas de éxito. Hay dos formas fundamentales:

- e*n profundidad*: se pierde la completitud del procedimiento de resolución SLD; y
- e*n anchura*: se recorre por niveles el árbol (es muy costosa). Se mantiene la completitud.

Prolog

Los Prolog´s que se encuentran habitualmente en el mercado tienen las siguientes características

- Regla de computación: 1er átomo de la izquierda.

• Regla de búsqueda: en profundidad.

3.2.1.5.Semántica Declarativa de los Programas Definidos

Dado un programa definido P, sabemos como computar objetivos y cuál es la teoría de 1er orden que la representa. Pero, ¿qué significado tiene el programa P?. Para reponder a esa pegunta, nos planteamos encontrar un modelo de esa teoría que sea **"***equivalente***"** a la teoría.

Antes se introducen algunos conceptos previos.

Universo de Herbrand

Dado un lenguaje de 1er orden L, el *universo de Herbrand* U_L de L es el conjunto de todos los términos base.

Base de Herbrand

Dado un lenguaje de 1er orden L, la *base de Herbrand* B_L de L es el conjunto de todos los átomos base que se pueden formar con todos los símbolos de predicado de L y todos los términos de U_L.

Interpretación de Herbrand

Dado un lenguaje de 1er orden L, una *interpretación Herbrand I* es un triplete (U_L, K, E) siendo U_L el dominio de la interpretación, K la asignación a símbolos de constantes y a funciones definida de la siguiente forma:

• cada constante de L $\rightarrow$ ella misma

• a cada función n-aria f se la asigna la siguiente función:

$$\begin{array}{ccc} f(t_1, \ldots, t_n) & \rightarrow & f(t_1, \ldots, t_n) \\ \downarrow & & \downarrow \\ \in (U_L)^n & & \in U_L \end{array}$$

y, finalmente E es la asignación a cada símbolo de predicado n-ario de un relación definida sobre D^n.

Como puede observarse de la definición, las distintas interpretaciones de Herbrand que existen, se diferencian únicamente en las diferentes asignaciones E, o sea, en determinar qué átomos base van a ser ciertos y falsos en la interpretación. Por tanto una interpretación puede definirse como un subconjunto de B_L, y, conocer el valor de verdad de un átomo base en un interpretación de Herbrand se convierte en saber si pertenece o no al subconjunto de B_L equivalente a la interpretación.

Proposición

Sea C un conjunto de cláusulas, si C tiene un modelo, entonces C tiene un modelo de Herbrand.

Modelo Mínimo de Herbrand

Sea P un programa definido y sea $\{M_i\}_{i\in I}$ el conjunto no vacío de modelos de Herbrand de P. Entonces $M_P = \cap_{i\in I} M_i$ es también un modelo de P.

A M_P se le denomina *modelo mínimo de Herbrand*.

Este modelo de Herbrand será usualmente la semántica supuesta del programa lógico.

Ejemplo : Sea P el siguiente programa definido:

$p(a) \leftarrow$
$p(b) \leftarrow$
$r(x) \leftarrow p(x)$
$t(x, y) \leftarrow p(x) \wedge r(y)$
$s(c) \leftarrow$

y el siguiente lenguaje L = (A, F) donde

- constantes de A = {a, b, c}
- funciones de A = {}
- predicados de A = {p(.), r(.), t(.,.), s(.)}

entonces se puede afirmar que:

- el universo de Herbrand de L es:

  U_L = {a, b, c}

- La base de Herbrand de L es:

  B_L = {p(a), p(b), p(c), r(a), r(b), r(c), s(a), s(b), s(c), t(a, a), t(a, b), …}

- Cualquier subconjunto de B_L es una interpretación de Herbrand de P.

- Cualquier interpretación de Herbrand de P que contenga al conjunto:

  {p(a), p(b), s(c), r(a), r(b), t(a,a), t(a,b), t(b,a), t(b,b)}

  es un modelo de Herbrand de P

Teorema

Sea P un programa definido. Entonces $M_P \equiv \{A \in B_L : P \models A\}$

¿Cómo podríamos, dado un programa definido P, obtener su M_P?. Evidentemente la definición declarativa de M_P no nos dice mucho ya que para calcularlo deberíamos encontrar todos los modelos de Herbrand del programa y hallar su intersección, por ello vamos a dar una visión constructiva de M_P para lo cual necesitamos algunas definiciones previas.

(Normalmente, por comodidad consideraremos que las interpretaciones están asociadas a los programas y no al lenguaje subyacente a éstos. Utilizaremos por tanto U_P para referirnos al Universo de Herbrand de un programa, B_P a la base de Herbrand, etc.)

Conjunto de Interpretaciones de Herbrand

Dado un programa definido P, se define:

2= {conjunto de todas las interpretaciones de P (subconjuntos de B_P)}

que corresponde al conjunto de las partes de B_P.

Operador Consecuencia Lógica

Sea P un programa definido, el *operador consecuencia lógica* T_P es una aplicación con el siguiente dominio y codominio:

$$T_P : \quad 2 \rightarrow \quad 2$$

definida de la siguiente forma:

Sea I una interpretación de Herbrand (subconjunto de B_P), entonces:

$T_P(I)=\{A \in B_P | \ A \leftarrow A_1 \wedge \ldots \wedge A_n$ es una instancia base de una sentencia de P y
$\{A_1, \ldots, A_n\} \subseteq I\}$

El operador consecuencia lógica así definido sobre 2tiene las siguientes propiedades:

- Es monotónico, es decir para cualquier par de interpretaciones de Herbrand de P, I_1, I_2, se cumple:

$$\text{si } I_1 \subseteq I_2 \text{ entonces } T_P(I_1) \subseteq T_P(I_2)$$

- Existe una interpretación de Herbrand I que es el menor punto fijo de T_P**:**

$$T_P(I) = I$$

Teorema (Caracterización del Modelo Mínimo de Herbrand)

Sea P un programa definido. Entonces:

$$M_P = lfp(T_P)$$

donde $lfp(T_P)$ es el menor punto fijo del operador T_P.

Para computar el modelo mínimo de Herbrand, empezaremos aplicando el operador consecuencia lógica al conjunto vacío, y repetiremos su aplicación sobre la interpretación obtenida hasta llegar al punto fijo.

Ejemplo: (caso anterior)

$T_P \uparrow^1 = T_P(\varnothing) = \{p(a), p(b), s(c)\}$

$T_P \uparrow^2 = T_P(T_P \uparrow^1) = \{p(a), p(b), s(c), r(a), r(b)\}$

$T_P \uparrow^3 = T_P(T_P \uparrow^2) = \{p(a), p(b), s(c), r(a), r(b), t(a,a), t(a, b), t(b, a), t(b, b)\}$

$T_P \uparrow^4 = T_P \uparrow^3$

luego $T_P \uparrow^3$ es $lfp(T_P)$ y por tanto su M_P

Ejemplo: **Hijo(h, p/m)** **Desc(d, a)**

Programa Lógico

P ={ Hijo(maría, juan) ←,
Hijo(maría, dolores) ←,
Hijo(dolores, pedro) ←,
Hijo(dolores, mercedes) ←,
Hijo(juan, josé) ←,
Hijo(juan, isabel) ←,

Desc(x,y) ← Hijo(x,y)
Desc(x,y) ← Hijo(x,z) ∧ Desc(z,y)}

$T_P \uparrow^1$ = {Hijo(maría, juan), Hijo(maría, dolores), Hijo(dolores, pedro),
Hijo(dolores, mercedes), Hijo(juan, josé), Hijo(juan, isabel)}

$T_P \uparrow^2$ = {Hijo(maría, juan),Hijo(maría, dolores), Hijo(dolores, pedro),
Hijo(dolores, mercedes), Hijo(juan, josé), Hijo(juan, isabel),
Desc(maría, juan), Desc(maría, dolores), Desc(dolores, pedro),
Desc(dolores, mercedes), Desc(juan, josé), Desc(juan, isabel)}

$T_P \uparrow^3$ = {Hijo(maría, juan), Hijo(maría, dolores), Hijo(dolores, pedro),
Hijo(dolores, mercedes), Hijo(juan, josé), Hijo(juan, isabel),
Desc(maría, juan), Desc(maría, dolores), Desc(dolores, pedro),
Desc(dolores, mercedes), Desc(juan, josé), Desc(juan, isabel),

Desc(maría, pedro),Desc(maría, mercedes),Desc(maría, josé),
Desc(maría, isabel)}

$T_P \uparrow^4 = T_P \uparrow^3$

y por tanto lfp(P)= $T_P \uparrow^3 = M_P$.

3.2.2. NEGACIÓN

Una limitación que podemos destacar de los programas definidos, es su falta de expresividad para determinadas ocasiones en las que son necesarios literales negativos en el cuerpo de una sentencia de programa.

Ejemplo: Supongamos un programa que determina si dos conjuntos son diferentes:

diferente(x,y) ← miembro (z,x) ∧ ¬ miembro(z,y)

diferente(x,y)← ¬ miembro (z,x) ∧ miembro(z,y).

La introducción de literales negativos en el cuerpo de las cláusulas sin embargo, no es trivial.

Con la regla de inferencia que hemos presentado en el punto anterior (procedimiento de resolución SLD) nunca podremos deducir información negativa. Para ser más precisos, si P es un programa definido y A un átomo de la Base de Herbrand, nunca podremos probar que ¬A es consecuencia lógica de P. La razón es que P ∪ {A} siempre es satisfactible teniendo, entre otros, a B_P como modelo.

Ejemplo: Sea P el siguiente programa definido:

estudiante(juan) ←
estudiante(pepe) ←
estudiante(ana) ←
profesor(maría) ←

y sea G= ¬ *estudiante(maría)* un objetivo. Como hemos comentado antes, G no es consecuencia lógica de P. Sin embargo tampoco ¬ G es consecuencia lógica de P.

Hipótesis del Mundo Cerrado

Lo que se puede hacer en situaciones como la del ejemplo anterior, es invocar una regla especial de inferencia: **"Si un átomo base A no es consecuencia lógica del programa, entonces inferimos ¬ A".** Esta regla de inferencia fue introducida por Reiter y se conoce como la "**Hipótesis**

del Mundo Cerrado" (HMC). Con esta regla de inferencia, podemos inferir ¬*estudiante(maría)* sobre la base de que *estudiante(maría)* no es consecuencia lógica de P.

La HMC es un ejemplo de regla de inferencia no-monotónica. Una regla de inferencia es no-monotónica si la inserción de un nuevo axioma puede disminuir el conjunto de teoremas que se podían probar antes. En el ejemplo anterior, si añadimos la cláusula *estudiante(maría)*←*,* ya no podremos utilizar la para probar ¬*estudiante(maría)*.

Sea P un programa para el cual la HMC es aplicable, y sea A un átomo de la Base de Herbrand B_P y supongamos que queremos inferir ¬A. Para poder utilizar la HMC debemos probar que A no es consecuencia lógica de P. Desafortunadamente, debido al problema de indecibilidad de lo lógica de primer orden, no hay algoritmo que, dado un átomo cualquiera responda en un tiempo finito si A es o no consecuencia lógica de P; si no lo es puede entrar en un bucle infinito.

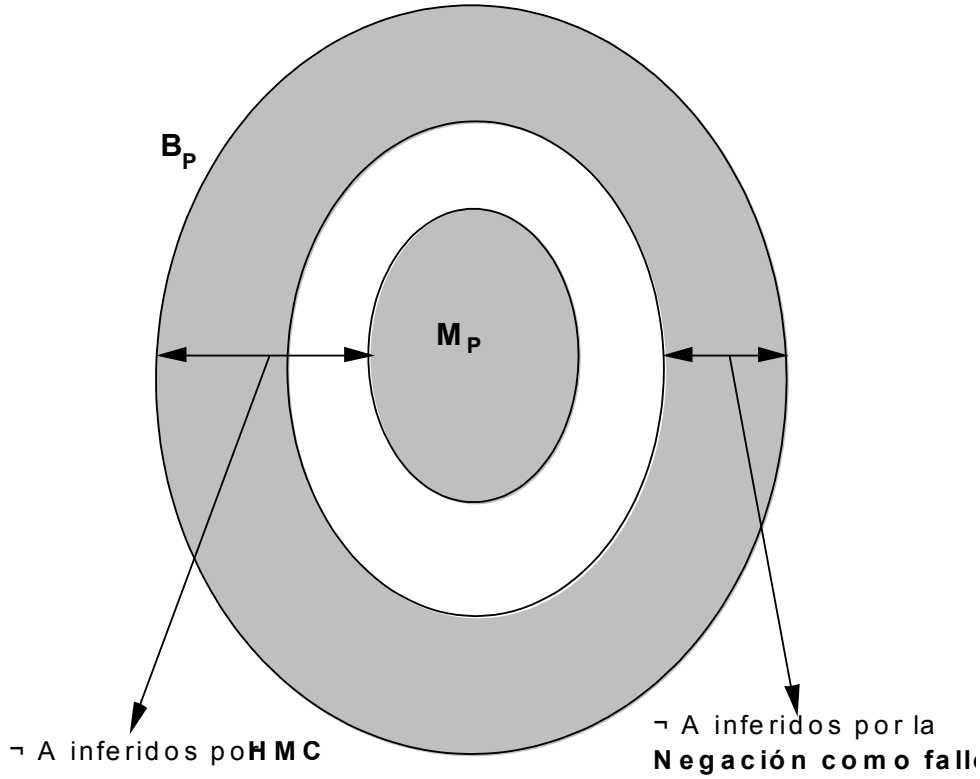
Negación como fallo

Por los problemas arriba mencionados, en la práctica, la aplicación de la HMC se limita a aquellos átomos cuyo prueba falla finitamente. Esta restricción, define otra regla de inferencia que se denomina **"Negación como fallo (NF)"** y que consiste en **"inferir ¬A si y sólo si el intento de inferir A falla finitamente"**, si esto ocurre A pertenece al conjunto de fallo finito SLD de P que, intuitivamente se define como sigue.

Para un programa definido P, el **conjunto de fallo finito SLD de P**, **F_P** es el conjunto de todos los átomos $A \in B_P$ para los que existe un árbol SLD fallado finitamente para $P \cup \{\leftarrow A\}$, es decir un árbol SLD finito y sin ramas de éxito.

Comparación Negación como Fallo e Hipótesis de Mundo Cerrado

Ya que el conjunto de fallo finito es en general un subconjunto del conjunto de éxito de un programa P (conjunto de átomos que son consecuencia lógica de P), es fácil apreciar que la negación como fallo es una regla de inferencia menos poderosa que la hipótesis del mundo cerrado.



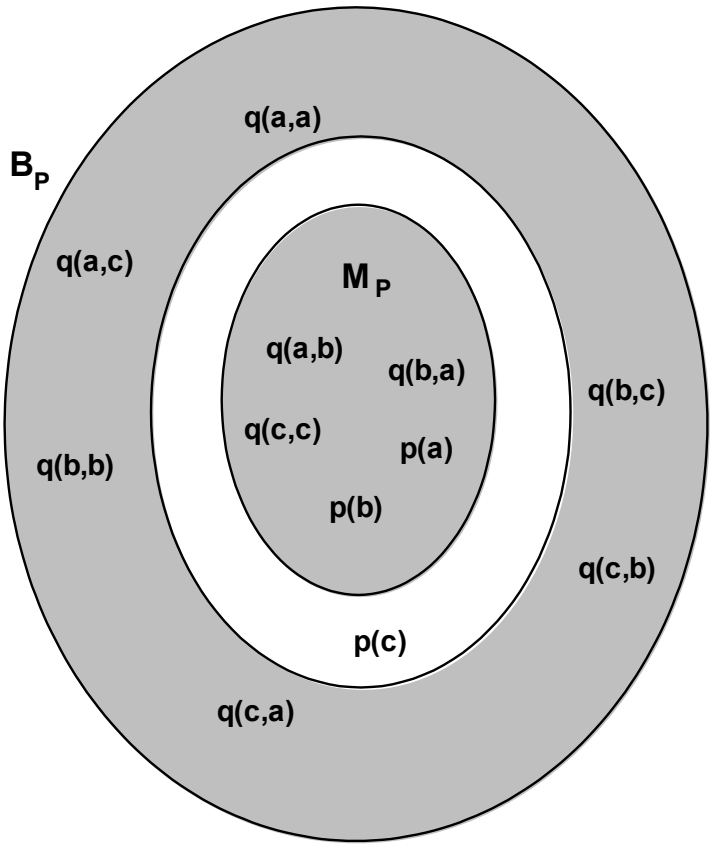
La Hipótesis del Mundo Cerrado de Reiter infiere cualquier átomo negado que no sea consecuencia lógica de P (no es implementable) y sin embargo la Negación como Fallo infiere, comparativamente, menos.

Ejemplo

Sea P la siguiente programa definido:

$p(x) \leftarrow q(x, y) \wedge p(y)$
$q(a, b) \leftarrow$
$q(b, a) \leftarrow$
$q(c, c) \leftarrow$
$p(b) \leftarrow$

entonces su B_P se puede dividir como se ve en el diagrama:



si asignamos a la negación una semántica por Fallo Finito, existe un átomo, p(c), sobre el cual no se puede decir ni que es consecuencia lógica ni que su negación lo sea.

Sin embargo, aunque ahora podríamos tener objetivos con literales negativos en un programa que serían tratados por la regla de la Negación como fallo, todavía no podemos afirmar que un literal ¬A es consecuencia lógica de P, siendo P un programa. El problema sigue siendo el mismo, ya que $P \cup \{A\}$ tiene siempre, entre otros, a la Base de Herbrand como modelo y es por tanto satisfactible. La razón de este comportamiento es que, las sentencias de programa sólo contienen la definición de los predicados en un sentido (←). Para poder deducir información negativa de un programa debemos pues *completar* estas definiciones es decir, el conjunto de sentencias que tienen

el mismo símbolo de predicado en la cabeza, se debe interpretar como la definición completa de dicho predicado en P.

Ejemplo: En el programa de los estudiantes deberíamos añadir una sentencia que exprese que sólo son estudiantes *juan, pepe*, y *ana*:

$$\text{estudiante}(x) \rightarrow (x=\text{juan}) \vee (x=\text{pepe}) \vee (x=\text{ana})$$

de manera que añadiendo también los axiomas de la igualdad necesarios, ya podríamos deducir que $\neg$*estudiante (maría)* es consecuencia lógica de $P \cup \{\text{estudiante}(x) \rightarrow (x=\text{juan}) \vee (x=\text{pepe}) \vee (x=\text{ana})\}$.

Compleción de un Programa

La *compleción* de un programa se obtiene añadiendo a P los axiomas de compleción para cada predicado del lenguaje subyacente junto con la teoría de la igualdad (estos últimos son necesarios al aparecer el predicado igualdad en los axiomas anteriores).

Dado un programa P llamamos compleción del programa **comp(P)** al programa P completado según se muestra a continuación.

Para cada símbolo de predicado del lenguaje L existirá un axioma de compleción que definirá lo que es falso en ese predicado. Para definir estos axiomas distinguiremos dos casos:

1) Existen sentencias en el programa P cuyo átomo de cabeza es el predicado *p*

Sea la siguiente una de esas sentencias:

$$p(t_1, \ldots, t_n) \leftarrow F$$

se transforma de la siguiente forma:

$$p(x_1, \ldots, x_n) \leftarrow \exists y_1,\ldots, \exists y_m\, (=(x_1, t_1) \wedge \ldots \wedge =(x_n, t_n) \wedge F)$$

donde $y_1,\ldots, y_m$ son las variables libres en F.

Supongamos que existen k ($k \geq 1$) sentencias con el predicado *p* en la cabeza. Se transforman todas ellas de la forma anterior, con lo cual tenemos:

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

donde E_i tiene la forma general

$$E_i = \exists y_1,\ldots, \exists y_m(=(x_1, t_{i1}) \wedge \ldots \wedge (x_n, t_{in}) \wedge F_i)$$

El axioma de compleción para p es el siguiente:

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

b) <u>En P no existe ninguna sentencia con el predicado *p* en la cabeza</u>

Intuitivamente esto quiere decir que en p no hay nada cierto. Esto se representa con el siguiente axioma de compleción:

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

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 es una respuesta correcta de $P \cup \{G\}$.

Este resultado establece la equivalencia entre P y comp(P) a nivel de consecuencias lógicas positivas:

$$A \in B_P \; (P \models A \leftrightarrow comp(P) \models A)$$

El trabajar con la compleción de P da sentido a los átomos negativos que se pueden inferir con la regla de la negación como fallo: "**Si $A \in B_P$ pertenece al conjunto de fallo finito de P ($A \in F_P$) entonces $comp(P) \models \neg A$**"

3.2.3. Programas Normales

Los programas normales incrementan la capacidad expresiva de los definidos al permitir literales negativos en el cuerpo de las sentencias de programa.

Programa Normal

Es un programa cuyas sentencias de programa son sentencias normales:

Sentencia Normal

Una sentencia normal tiene la siguiente forma:

$$A \leftarrow L_1 \wedge \ldots \wedge L_n$$

donde A es una átomo y $L_1, \ldots, L_n$ es una conjunción de literales.

Objetivo Normal

Un objetivo normal tiene la forma $\leftarrow L_1 \wedge \ldots \wedge L_m$ donde $L_1, \ldots, L_m$ es una conjunción de literales.

Procedimiento de Resolución SLDNF

SLDNF = SLD + Negación como Fallo

La idea de este procedimiento de resolución es que dentro de una derivación se podrá inferir un literal negado si éste está en el conjunto de fallo finito (o sea, existe un árbol fallado finitamente con este átomo como raíz).

Utilizando el procedimiento de resolución SLDNF podemos ampliar la expresividad de nuestros programas y objetivos introduciendo la negación.

Vamos a ver las definiciones análogas a las del procedimiento de resolución SLD.

Respuesta

Sea P un programa normal y G un objetivo normal. Una respuesta θ para $P \cup \{G\}$ es una *sustitución* de las variables de G.

Respuesta Correcta

Sea P un programa normal, $\leftarrow L_1 \wedge \ldots \wedge L_n$ un objetivo normal y θ una respuesta para $P \cup \{\leftarrow L_1 \wedge \ldots \wedge L_n\}$, entonces θ es una respuesta correcta para $P \cup \{\leftarrow L_1 \wedge \ldots \wedge L_n\}$ si y sólo si $comp(P) \models \overline{\forall} ((L_1 \wedge \ldots \wedge L_n)\theta)$

3.2.4. Derivación SLDNF

Sea P un programa normal y sea G un objetivo normal. Una derivación SLDNF de $P \cup \{G\}$ consiste en:

- una secuencia de objetivos normales $G_0, G_1, \ldots$ donde $G_0 = G$;
- una secuencia de variantes de sentencias de P, $S_1, S_2, \ldots$; y
- una secuencia de mgu $\theta_1, \theta_2, \ldots$

que satisfacen lo siguiente:

1) para cada i:

1.1) si $G_i = \leftarrow L_1 \wedge \ldots \wedge L_k \wedge \ldots \wedge L_m$ y el literal seleccionado L_k es positivo, entonces G_{i+1} se deriva de G_i y S_{i+1} usando θ_{i+1}

1.2) si $G_i = \leftarrow L_1 \wedge \ldots \wedge L_k \wedge \ldots \wedge L_m$ y el literal seleccionado $L_k = \neg A_k$ (átomo base) y existe un *árbol SLDNF fallado finitamente* para $P \cup \{\leftarrow A_k\}$, entonces:

- $G_{i+1} = \leftarrow L_1 \wedge \ldots \wedge L_{k-1} \wedge L_{k+1} \wedge \ldots \wedge L_m$;
- $\theta_{i+1} = \varepsilon$ (identidad); y
- $S_{i+1} = \neg A_k$

2) si la secuencia de objetivos es finita $G_0, \ldots, G_n$, entonces se cumple una de estas condiciones:

2.1) el último objetivo es $G_n = \square$. A esta derivación se la denomina *refutación SLDNF*.

2.2) el último objetivo es $G_n = \leftarrow L_1 \wedge \ldots \wedge L_k \wedge \ldots \wedge L_m$, se selecciona el literal L_k positivo y no unifica con ninguna sentencia de P. A esta derivación SLDNF se la denomina *derivación fallada*.

2.3) el último objetivo es $G_n = \leftarrow L_1 \wedge \ldots \wedge L_k \wedge \ldots \wedge L_m$, se selecciona el literal $L_k = \neg A_k$ (A_k es base) y existe una refutación SLDNF para $P \cup \{\leftarrow A_k\}$. A esta derivación SLDNF se la denomina *derivación fallada*.

Respuesta Computada

Sea P un programa normal y sea G un objetivo normal. Una respuesta computada θ para $P \cup \{G\}$ es:

$$\theta = \theta_1\theta_2 \ldots \theta_n \text{ restringida a las variables de G}$$

donde $\theta_1\theta_2 \ldots \theta_n$ es la secuencia de sustituciones usadas en una refutación SLDNF para $P \cup \{G\}$.

Árbol SLDNF

El conjunto de derivaciones SLDNF que parten de un mismo objetivo se pueden representar mediante un árbol, de forma análoga a como hacíamos en el procedimiento de resolución SLD.

Sea P un programa normal y sea G un objetivo normal. Un árbol SLDNF para $P \cup \{G\}$ satisface las siguientes condiciones:

a) cada nodo es un objetivo normal

b) el nodo raíz es G

c) sea $\leftarrow L_1 \wedge \ldots \wedge L_k \wedge \ldots \wedge L_m$ ($m \geq 1$) un nodo no terminal. Entonces se cumple una de estas afirmaciones:

c.1) el literal seleccionado $L_k = A_k$, entonces por cada variante de sentencia de P, $A \leftarrow M_1 \wedge \ldots \wedge M_q$ tal que existe un mgu $\theta = mgu(A, A_k)$, existe un hijo:

$$\leftarrow (L_1 \wedge \ldots \wedge L_{k-1} \wedge M_1 \wedge \ldots \wedge M_q \wedge L_{k+1} \wedge \ldots \wedge L_m)\theta$$

c.2) el literal seleccionado $L_k = \neg A_k$ (átomo base) y existe un árbol SLDNF fallado finitamente para $P \cup \{\leftarrow A_k\}$, entonces tiene un único hijo

$$\leftarrow L_1 \wedge \ldots \wedge L_{k-1} \wedge L_{k+1} \wedge \ldots \wedge L_m$$

d) sea $\leftarrow L_1 \wedge \ldots \wedge L_k \wedge \ldots \wedge L_m$ ($m \geq 1$) un nodo terminal. Entonces se cumplen una de estas afirmaciones:

d.1) el literal seleccionado L_k es positivo y no unifica con ninguna sentencia de P.

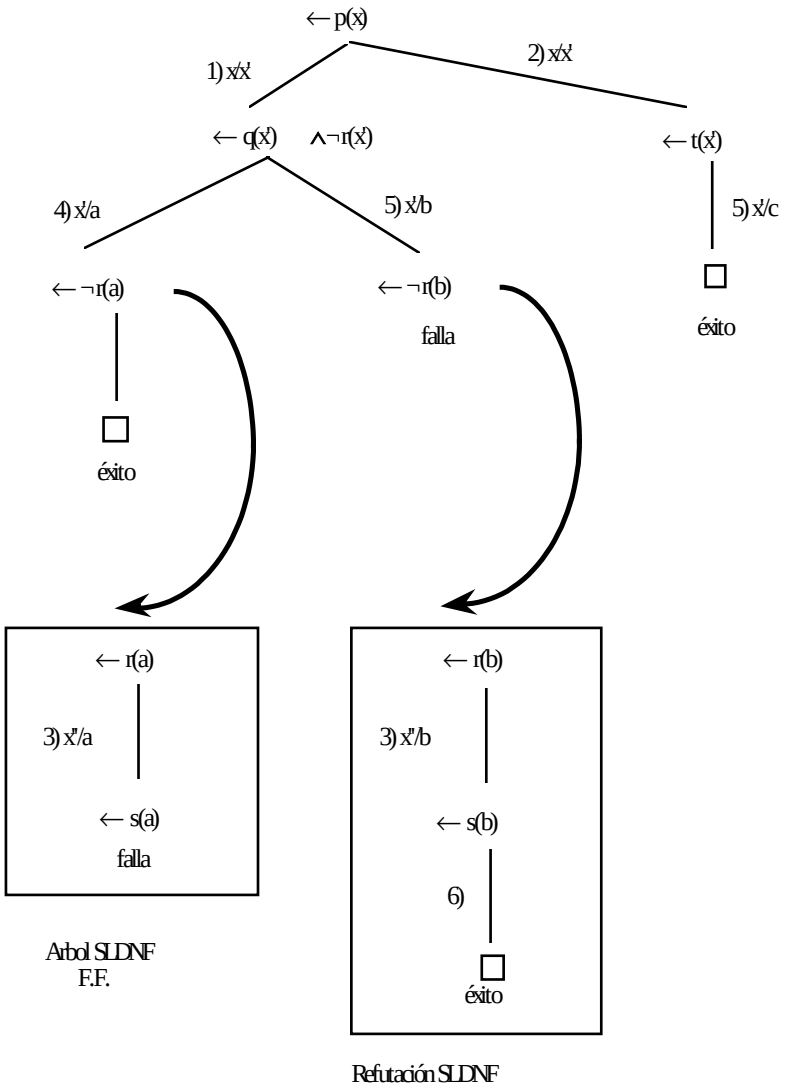
d.2) el literal seleccionado $L_k = \neg A_k$ (átomo base) y existe una refutación SLDNF para $P \cup \{\leftarrow A_k\}$

e) los nodos que son $\square$ no tienen hijos.

Ejemplo: Sea el siguiente programa P:

1 $p(x) \leftarrow q(x) \wedge \neg r(x)$ 2 $p(x) \leftarrow t(x)$
3 $r(x) \leftarrow s(x)$ 4 $q(a) \leftarrow$
5 $q(b) \leftarrow$ 6 $s(b) \leftarrow$ 7 $t(c) \leftarrow$

y sea el objetivo normal $\leftarrow p(x)$. El árbol SLDNF con regla de computación "el 1º de la izquierda" es el que se muestra a continuación:



Limitaciones de la Regla de Computación

En una derivación SLDNF no se puede seleccionar en ningún objetivo un literal negativo no base.

3.2.4.1.Regla de Computación Segura

Una regla de computación es *segura* si y sólo si:

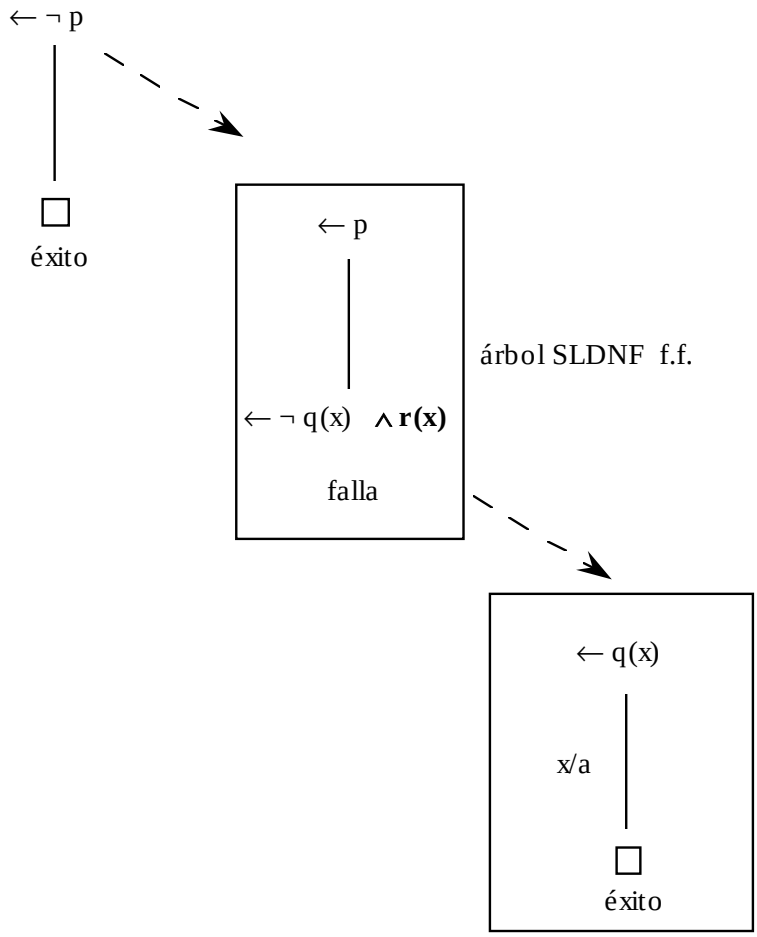
a) nunca selecciona un literal negativo no base; y

b) si en una derivación se produce un objetivo de la forma $\leftarrow L_1 \wedge \ldots \wedge L_m$, tal que $\forall i\ L_i$ es un literal negativo no base entonces la computación se detiene. A esta situación se la denomina *tropiezo*.

Ejemplo: Sea P el siguiente programa normal:

$p \leftarrow \neg q(x) \wedge r(x)$
$q(a) \leftarrow$
$r(b) \leftarrow$

y sea G el siguiente objetivo normal $\leftarrow \neg p$. Supongamos que se utiliza una regla de computación no-segura que selecciona el 1º de la izquierda.



y sin embargo comp(P) $|\neq \neg p$. O sea que utilizando una regla de computación no-segura se pierda la corrección del procedimiento de resolución SLDNF.

3.2.4.2.Propiedades del SLDNF

- **Correcto**: sea P un programa normal y sea G un objetivo normal. Si θ es una respuesta computada para $P \cup \{G\}$, entonces θ es una respuesta correcta para $comp(P) \cup \{G\}$.

- **No completo**: dado un programa normal P y un objetivo normal G existen respuestas correctas θ que no pueden ser computadas por el SLDNF.

Ejemplo:

Sea P el siguiente programa normal

$r \leftarrow p$
$r \leftarrow \neg p$
$p \leftarrow p$

y sea G el siguiente objetivo normal $\leftarrow r$. Se puede ver fácilmente que comp(P) $|= r$, pero todo árbol SLDNF para $P \cup \{\leftarrow r\}$ es infinito sin ramas de éxito.

3.2.4.3.Clases de Programas Normales

La incompletitud del SLDNF está provocada por:

- *Tropiezo:* se detiene la computación.

- *Negación + Recursión:* la existencia de ramas infinitas provoca el no poder deducir consecuencias lógicas.

Vamos a definir subclases de programas normales que por su forma eviten estos problemas. En concreto presentaremos los programas permitidos, jerárquicos y estratificados.

3.2.4.4.Programas Permitidos (no tropiezo)

Antes de definir esta clase de programas normales es necesario dar unas definiciones previas.

Sentencia admisible

Una sentencia de programa normal $A \leftarrow L_1 \wedge \ldots \wedge L_n$ es admisible si cada variable de la sentencia o aparece en la cabeza A o en algún literal positivo del cuerpo.

Sentencia permitida

Una sentencia de programa normal $A \leftarrow L_1 \wedge \ldots \wedge L_n$ es permitida si cada variable de la sentencia aparece en un literal positivo del cuerpo.

Objetivo permitido

Un objetivo normal $\leftarrow L_1 \wedge \ldots \wedge L_n$ es un objetivo permitido si cada una de sus variables aparece en un literal positivo.

Programa Permitido

Un programa normal P es permitido si cada sentencia de P es permitida.

Programa con un objetivo $P \cup \{G\}$ permitido

Un programa y un objetivo, $P \cup \{G\}$, es permitido si se cumplen las siguientes condiciones:

i) cada sentencia de P es admisible.

ii) si un símbolo de predicado *p* aparece en un literal positivo en G o en el cuerpo de alguna una sentencia de P, entonces toda sentencia de P en la que *p* aparece en su cabeza debe ser permitida.

iii) G es permitido.

Ejemplo

Sean P_1 y P_2 dos programas normales

$P_1 \equiv$ $p(x) \leftarrow q(x) \wedge \neg r(x)$ $q(x) \leftarrow t(x, y)$ $t(a, a) \leftarrow$ $t(a, b) \leftarrow$	$P_2 \equiv$ $p(x) \leftarrow q(x) \wedge \neg r(x)$ $q(x) \leftarrow t(x, y)$ $s(x) \leftarrow$ $t(a, a) \leftarrow$ $t(a, b) \leftarrow$

P_1 es un programa permitido, por tanto con cualquier objetivo permitido, el programa y el objetivo será permitido.

Sin embargo, P_2 no es permitido y es posible que aunque el objetivo sea permitido, el programa y el objetivo no lo sea. (p. e. $\leftarrow s(x) \wedge \neg p(x)$).

Teorema

Sea P un programa normal y sea G un objetivo normal. Si $P \cup \{G\}$ es permitido entonces:

a) ninguna computación de $P \cup \{G\}$ tropieza; y

b) cada respuesta computada de $P \cup \{G\}$ es una sustitución base de todas las variables del objetivo G.

3.2.4.5.Programas Jerárquicos y Estratificados (Negación + Recursión)

El otro problema que puede provocar la incompletitud del procedimiento de resolución SLDNF es la presencia de Recursión en el programa en términos de Negación. En un intento de evitar este problema definiremos dos clases de programas: Programas Jerárquicos y Estratificados. Para ello es necesario previamente introducir un concepto nuevo:

Aplicación de Nivel

Una aplicación de nivel AN es una aplicación definida como sigue:

$$AN : \text{símbolos de predicados} \rightarrow \text{Naturales}$$

de forma que para cada predicado p, le asigna un natural (o sea, AN(p) = n) que se denomina *nivel del predicado p*.

Programa Jerárquico

Un programa normal P es jerárquico si existe una aplicación de nivel AN tal que dada cualquier sentencia de P de la forma:

$$p(x_1, \ldots, x_m) \leftarrow L_1 \wedge \ldots \wedge L_n$$

se cumple que $\forall i$, $AN(p) > AN(p_i)$, donde $p_1, \ldots, p_n$ son los símbolos de predicados de $L_1, \ldots, L_n$. Esto quiere decir que se prohibe la presencia de recursión en el programa.

Programa Estratificado

Un programa normal P es estratificado si existe una aplicación de nivel AN, tal que para cualquier sentencia de P, de la forma:

$$p(x_1, \ldots, x_m) \leftarrow L_1 \wedge \ldots \wedge L_n$$

se cumple que:

i) AN(p) (AN(p_i) para todo $L_i = p_i(t_1, \ldots, t_l)$
ii) $AN(p) > AN(p_j)$ para todo $L_i = \neg p_i(t_1, \ldots, t_l)$

Esto quiere decir intuitivamente que en esta clase de programas no se puede darse recursión en términos de negación.

Ejemplo

$P_1 \equiv$ $p(x) \leftarrow q(x) \wedge \neg r(x)$ $q(x) \leftarrow \neg t(x)$ $t(a) \leftarrow$ $r(b) \leftarrow$	$P_2 \equiv$ $p(x) \leftarrow \neg r(x) \wedge p(x)$ $r(x) \leftarrow q(x)$ $q(a) \leftarrow$ $q(b) \leftarrow$
Jerárquico	Estratificado

3.2.4.6.Teorema (Completitud del SLDNF para Programas Jerárquicos)

Sea P un programa jerárquico, sea G un objetivo normal y sea C una regla de computación segura. Si $P \cup \{G\}$ es permitido entonces:

"toda respuesta correcta θ para $comp(P) \cup \{G\}$ y sustitución base de todas las variables de G es una respuesta C-computada para $P \cup \{G\}$".

3.2.4.7.Semántica Declarativa de los Programas Normales

En los programas definidos sabemos que M_P nos da la semántica adecuada (para la información positiva) y además el operador T_P define este modelo constructivamente.

¿Se puede aplicar lo mismo en programas normales?. La respuesta en este caso es no. En programas normales surgen (debido a la negación) tres problemas nuevos:

1) Inconsistencia de comp(P): debido a la presencia de recursión en términos de negación es posible que la compleción de un programa sea inconsistente (es decir, no tenga modelo).

2) T_P es no-monotónico: debido también a la negación el operador consecuencia inmediata ya no es monotónico, con lo cual no se puede aplicar la Teoría del Punto Fijo para localizar este modelo mínimo.

3) Existencia de varios modelos mínimos: al introducir la negación en el programa perdemos la unicidad de modelo mínimo.

Veamos unos ejemplos de estos tres problemas.

Ejemplo:

Sean P_1, P_2 y P_3 tres programas normales:

$P_1 \equiv p(x) \leftarrow \neg p(x)$ $P_2 \equiv p(x) \leftarrow \neg q(x)$ $P_3 \equiv p(a) \leftarrow \neg q(a)$

$q(a) \leftarrow$ $q(a) \leftarrow$ $q(b) \leftarrow$
$q(b) \leftarrow$ $r(b) \leftarrow$

Es fácil ver que comp(P_1) es inconsistente, o sea, no tiene ningún modelo.

También es sencillo observar que en los dos programas T_P es no-monotónico. Veámoslo en P_2:

$$T_{P(\varnothing)} = T_P\uparrow^1 = \{p(a), p(b), q(a), r(b)\}$$
$$T_P\uparrow^2 = \{q(a), p(b), r(b)\}$$

De P_3 se pueden encontrar dos modelos mínimos:

$M_1 = \{p(a), q(b)\}$ $M_2 = \{q(a), q(b)\}$

por tanto para programas normales no existe un modelo mínimo M_P, tal y como se definió para programas definidos

$$M_P = \cap_{i \in I} M_i$$

ya que se puede observar que $M_1 \cap M_2$ no es modelo de P_3.

Teorema

Sea P un programa normal, si P es estratificado entonces comp(P) tiene un *modelo minimal de Herbrand* (Modelo Estándar o Perfecto MS_P)

Por tanto sabemos que la compleción de los programa estratificados es consistente.

3.2.4.8.Semántica Declarativa por Modelo Estándar en programas Jerárquicos

Sea P un programa normal, si la negación aparecen en alguna sentencia de P entonces no existe un único modelo de Herbrand minimal.

Veámoslo en un ejemplo:

$P \equiv$ $p(x) \leftarrow q(x, y) \wedge \neg t(y)$
$q(a, b) \leftarrow$
$q(b, a) \leftarrow$
$t(b) \leftarrow$

es posible encontrar dos modelos mínimos, que son:

$M_1 = \{q(a, b), q(b, a), t(b), p(b)\}$

$M_2 = \{q(a, b), q(b, a), t(b), t(a)\}$

Vamos a analizar cada uno de estos modelos, observando las posibles instancias de la primera sentencia del programa:

$p(a) \leftarrow q(a, a) \wedge \neg t(a)$ 1
$p(a) \leftarrow q(a, b) \wedge \neg t(b)$ 2
$p(b) \leftarrow q(b, a) \wedge \neg t(a)$ 3
$p(b) \leftarrow q(b, b) \wedge \neg t(b)$ 4

claramente la diferencia de los dos modelos viene provocada por la instancia 3. Veamos que sentido le da cada modelo a esta instancia:

• Modelo M_1: este modelo afirma que p(b) es cierto, lo cual corresponde con lo que se podría denominar "Semántica razonable", ya que nada induce suponer que t(a) sea cierto.

• Modelo M_2: este modelo afirma que t(a) es cierto, por tanto p(b) no puede serlo, y, sin embargo, no existe nada en el programa que justifique esta afirmación.

El modelo M_1 coincide con el Modelo Estándar o Perfecto, que proporciona la semántica razonable de programas normales:

$M_1 = \{q(a, b), q(b, a), t(b), p(b)\} \equiv MS_P$

Ahora nos queda un problema pendiente: ¿cmo podemos caracterizar el Modelo Estándar de un programa P, MS_P?. O sea, tenemos que definir este modelo de una forma constructiva, tal y como hacíamos con el Modelo Mínimo en programas definidos.

Caracterización de MS_P mediante el operador TP en Programas Jerárquicos

En los programas jerárquicos (en general en normales) esta caracterización no es posible realizarla directamente, ya que el operador T_P no es monotónico. Veámoslo en la siguiente programa P:

$p(x) \leftarrow q(x, y) \wedge \neg t(y)$
$t(y) \leftarrow s(y)$
$q(a, b) \leftarrow$
$q(b, a) \leftarrow$
$s(b) \leftarrow$

Las sucesivas aplicaciones del operador T_P son:

$T_P\uparrow^1 = \{q(a, b), q(b, a), s(b)\}$

$T_P\uparrow^2 = \{q(a, b), q(b, a), s(b), t(b), \mathbf{p(a)}, p(b)\}$

$T_P\uparrow^3 = \{q(a, b), q(b, a), s(b), t(b), p(b)\}$

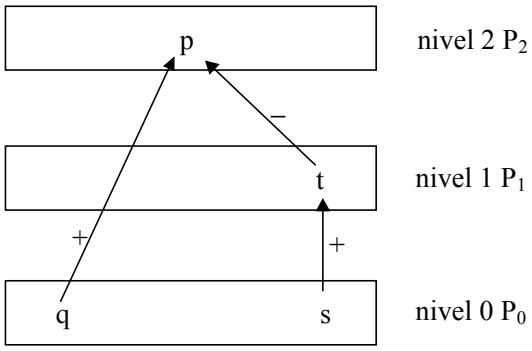
$$T_P\uparrow^3 \quad \subset \quad T_P\uparrow$$

Con lo cual no alcanzamos el punto fijo del operador T_P. Viendo las aplicaciones del operador y si nos fijamos en $T_P\uparrow^2$, el problema consiste en que en esta aplicación, **p(a)** aparece "demasiado pronto", dicho intuitivamente. O sea, se empieza a deducir hechos de **p** cuando todavía no se han deducido los hechos de **t** que pueden impedir que aparezcan estos primeros.

En programas estratificados se puede demostrar que si el operador T_P se aplica sucesivamente a los niveles de P, entonces T_P es monotónico y, por tanto, tiene punto fijo. Además:

$MS_P \equiv lfp(T_P)$ (aplicado por niveles)

Aplicación de nivel



La aplicación del operador T_P se realiza sucesivamente en cada uno de los niveles del programa jerárquico (una vez en cada nivel, ya que el programa es jerárquico). De forma que el punto fijo del nivel *i* sirve de interpretación inicio para el nivel *i+1*. Veámoslo en el ejemplo:

$T_{P_0}\uparrow^0 (\varnothing) = \{q(a, b), q(b, a), s(b)\}$

$T_{P_0}\uparrow^1 = T_{P_0}\uparrow^0 \qquad lfp(T_{P_0})$

$T_{P_1}\uparrow^0 (lfp(T_{P_0})) = \{q(a, b), q(b, a), s(b)\} \cup \{t(b)\}$

$T_{P_1}\uparrow^1 = T_{P_1}\uparrow^0 \qquad lfp(T_{P_1})$

$T_{P_2}\uparrow^0 (lfp(T_{P_1})) = \{q(a,b), q(b,a), s(b), t(b)\} \cup \{p(b)\}$

$T_{P_2}\uparrow^1 = T_{P_2}\uparrow^0 \qquad lfp(T_{P_2})$

De forma que el modelo estándar del programa coincide con el punto fijo del nivel máximo del programa.

$MS_P = lfp(T_{P_{\text{nivel máximo}}}) = \{q(a,b), q(b,a), s(b), t(b), p(b)\}$

Teorema

Sea P un programa jerárquico. Entonces se cumple que:

$$MS_P = \{A \in B_P \mid comp(P) \models A\}$$

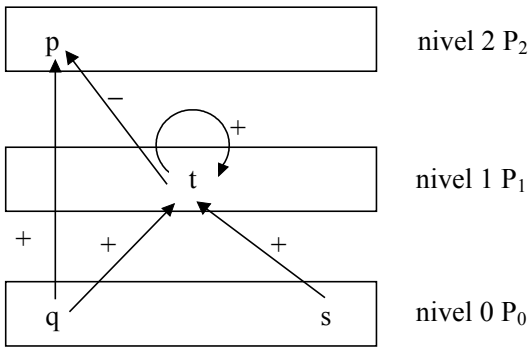
Semántica Declarativa por Modelo Estándar en Programas Estratificados

En los programas estratificados tampoco es posible la caracterización del modelo estándar por punto fijo del operador consecuencia inmediata T_P, ya que también está presente la Negación. Pero además la presencia de Recursión podría hacernos pensar que este problema se agrava.

Sea P el programa estratificado siguiente:

$p(x) \leftarrow q(x, y) \wedge \neg t(y)$
$t(y) \leftarrow q(y, z) \wedge t(z)$
$t(y) \leftarrow s(y)$
$q(d,e) \leftarrow$
$q(d, a) \leftarrow$
$q(a, d) \leftarrow$
$s(e) \leftarrow$

Aplicación de nivel



La aplicación del operador T_P se realiza sucesivamente en cada uno de los niveles (en estos programas posiblemente varias veces en cada nivel, debido a la presencia de recursión), de forma análoga a los programas jerárquicos. Veámoslo:

$T_{P_0}\uparrow^0(\varnothing) = \{q(b,c), q(d,e), q(a,d), q(d,a), s(e)\}$
$T_{P_0}\uparrow^1 = T_{P_0}\uparrow^0$ $\quad lfp(T_{P_0})$

$T_{P_1}\uparrow^0 (lfp(T_{P_0})) = \{q(b,c), q(d,e), q(a,d), q(d,a), s(e)\} \cup \{t(e)\}$
$T_{P_1}\uparrow^1 = \{q(b,c),q(d,e),q(a,d),q(d,a),s(e),t(e)\} \cup \{t(d)\}$
$T_{P_1}\uparrow^2 = \{q(b,c),q(d,e),q(a,d),q(d,a),s(e),t(e),t(d)\} \cup \{t(a)\}$
$T_{P_1}\uparrow^3 = T_{P_1}\uparrow^2$ $\quad lfp(T_{P_1})$

$T_{P_2}\uparrow^0(lfp(T_{P_1}))=\{q(b,c),q(d,e),q(a,d),q(d,a),s(e), t(e), t(d), t(a)\} \cup \{p(b)\}$
$T_{P_2}\uparrow^1 = T_{P_2}\uparrow^0$ $\quad lfp(T_{P_2})$

De forma que el modelo estándar de los programas estratificados también coincide con el punto fijo del nivel máximo.

$MS_P = lfp(T)=\{q(b,c), q(d,e), q(a,d), q(d,a), s(e), t(e), t(d), t(a), p(b)\}$

En programas estratificados no se puede enunciar un teorema análogo al torema anterior, esto quiere decir que puede existir un programa estratificado P tal que:

a) $A \in MS_P$ y $comp(P) \not\models A$
b) $CL_P = \{A \in B_P \mid comp(P) \models A\}$, CL_P no es modelo de comp(P)

Ejemplo

Sea P el siguiente programa estratificado:

$p(x) \leftarrow q(x, y) \wedge \neg r(y)$
$r(x) \leftarrow r(x)$
$q(a,b) \leftarrow$
$q(b,a) \leftarrow$

$MS_P = \{q(a,b), q(b,a), p(a), p(b)\}$
$CL_P = \{q(a,b), q(b,a)\}$

Se puede observar que CL_P no es modelo de comp(P) y además que p(a) y p(b) no son consecuencias lógicas de comp(P).

3.2.5. Programas Generales

Vamos a definir en este apartado los programas que tienen la forma más general. Veremos que como la semántica procedural a utilizar es la que proporciona el procedimiento de resolución SLDNF, será necesario transformar estos programas a otros normales "equivalentes".

Veamos algunos conceptos previos:

Sentencia General

Una sentencia general tiene la forma siguiente:

$$A \leftarrow F$$

donde A es un átomo y F es cualquier fbf.

Programa General

Un programa general es un programa cuyas sentencias son generales.

Objetivo General

Un objetivo general tiene la forma siguiente:

$$\leftarrow F$$

donde F es cualquier fórmula bien formada.

La semántica procedural para los Programas Generales la va a seguir dando el procedimiento de resolución SLDNF (aplicable a programa normales), es necesario por tanto realizar una transformación :

$$\text{Prog. General} \cup \text{Obj. General} \xRightarrow{\text{Transf.}} \text{Prog. Normal} \cup \text{Obj.Normal}$$

esta transformación debe poseer "**buenas propiedades**". Es decir, que lo que se derivaba del programa general se derive del normal y solamente eso.

3.2.5.1.Algoritmo de Lloyd

Sea P un programa general y sea G un objetivo general con la forma siguiente:

$$G= \leftarrow F(x_1, \ldots, x_n)$$

donde $x_1, \ldots, x_n$ son las variables libres de G.

Primero vamos a normalizar el objetivo construyendo el siguiente objetivo normal:

$$G' = \leftarrow \text{respuesta}(x_1, \ldots, x_n)$$

donde *respuesta* es un símbolo de predicado que no aparece en el programa original P.

Ahora construyamos el programa general intermedio P", añadiéndole la siguiente sentencia:

$$P'' = \{\text{respuesta}(x_1, \ldots, x_n) \leftarrow F\} \cup P$$

siendo posible demostrar que:

θ es respuesta correcta para $\text{comp}(P) \cup \{G\}$

si y sólo si

θ es respuesta correcta para $\text{comp}(P'') \cup \{\leftarrow \text{respuesta}(x_1, \ldots, x_n)\}$

Ya hemos, por tanto, normalizado el objetivo. Ahora sólo nos resta normalizar el programa general P" a un programa normal P', donde sea posible utilizar el procedimiento de resolución SLDNF.

Vamos a ver un conjunto de transformaciones que a partir de un conjunto de sentencias de programa generales permiten obtener un conjunto de sentencias de programa normales:

1. Sustituir $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg (V \wedge W) \wedge W_{i+1} \wedge \ldots \wedge W_n$

   por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg V \wedge W_{i+1} \wedge \ldots \wedge W_n$
   y por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg W \wedge W_{i+1} \wedge \ldots \wedge W_n$

2. Sustituir $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \forall x_1 \ldots \forall x_k\, W \wedge W_{i+1} \wedge \ldots \wedge W_n$

   por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg\exists x_1 \ldots \exists x_k\, (\neg W) \wedge W_{i+1} \wedge \ldots \wedge W_n$

3. Sustituir $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg \forall x_1 \ldots \forall x_k\, W \wedge W_{i+1} \wedge \ldots \wedge W_n$

   por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \exists x_1 \ldots \exists x_k\, (\neg W) \wedge W_{i+1} \wedge \ldots \wedge W_n$

4. Sustituir $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (V \leftarrow W) \wedge W_{i+1} \wedge \ldots \wedge W_n$

   por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge V \wedge W_{i+1} \wedge \ldots \wedge W_n$
   y por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg W \wedge W_{i+1} \wedge \ldots \wedge W_n$

5. Sustituir $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg (V \leftarrow W) \wedge W_{i+1} \wedge \ldots \wedge W_n$

   por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg V \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_n$

6. Sustituir $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge (V \vee W) \wedge W_{i+1} \wedge \ldots \wedge W_n$

por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge V \wedge W_{i+1} \wedge \ldots \wedge W_n$

y por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_n$

7. Sustituir $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg(V \vee W) \wedge W_{i+1} \wedge \ldots \wedge W_n$

por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg V \wedge \neg W \wedge W_{i+1} \wedge \ldots \wedge W_n$

8. Sustituir $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg (\neg W) \wedge W_{i+1} \wedge \ldots \wedge W_n$

por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_n$

9. Sustituir $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \exists x_1 \ldots \exists x_k\, W \wedge W_{i+1} \wedge \ldots \wedge W_n$

por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge \ldots \wedge W_n$

10.Sustituir $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg \exists x_1 \ldots \exists x_k\, W \wedge W_{i+1} \wedge \ldots \wedge W_n$

por $A \leftarrow W_1 \wedge \ldots \wedge W_{i-1} \wedge \neg p(y_1, \ldots, y_m) \wedge W_{i+1} \wedge \ldots \wedge W_n$

y por $p(y_1, \ldots, y_m) \leftarrow \exists x_1 \ldots \exists x_k\, W$

donde $y_1, \ldots, y_m$ son las variables libres de $\exists x_1 \ldots \exists x_k\, W$ y p es un nuevo símbolo de predicado.

Dado un programa general P, el algoritmo de Lloyd aplicado a P *termina en un número finito* de pasos dando como resultado un programa normal P'.

Equivalencia entre P y P'

Sea F una fórmula que sólo contiene símbolos de predicados de P. Entonces se cumple que:

$$comp(P) \models F \quad \Leftrightarrow \quad comp(P') \models F$$

O sea, queda asegurada la equivalencia entre el programa original P y el normalizado P'.

Ejemplo

Sea P el siguiente programa general:

$p(x) \leftarrow \forall y\, \forall z\, (q(x,y,z) \rightarrow r(x,y))$

$q(x,y,z) \leftarrow (t(x,y) \wedge v(z))$

$r(a,b) \leftarrow$

$r(a,c) \leftarrow$

$t(c,a) \leftarrow$

$v(d) \leftarrow$

Pasos de normalización

2. $p(x) \leftarrow \neg\, \exists y\, \exists z\, \neg(q(x,y,z) \rightarrow r(x,y))$

10. $p(x) \leftarrow \neg\, aux(x)$
$aux(x) \leftarrow \exists y\, \exists z\, \neg(q(x,y,z) \rightarrow r(x,y))$

9. $aux(x) \leftarrow \neg\, (q(x,y,z) \rightarrow r(x,y))$

5. $aux(x) \leftarrow q(x,y,z) \wedge \neg r(x,y)$

Con lo que finalmente obtendremos la siguiente programa normal P':

$p(x) \leftarrow \neg\, aux(x)$
$aux(x) \leftarrow q(x,y,z) \wedge \neg r(x,y)$
$q(x,y,z) \leftarrow t(x,y) \wedge v(z)$
$r(a,b) \leftarrow$
$r(a,c) \leftarrow$
$t(c,a) \leftarrow$
$v(d) \leftarrow$