CENTRO DE INVESTIGACION CIENTIFICA Y DE EDUCACION SUPERIOR DE ENSENADA

MANEJO DE LA NEGACION EN PROGRAMACION LOGICA

TESIS

MAESTRIA EN CIENCIAS

SONIA FAVELA VARA

Ensenada Baja California

Diciembre de 1989

RESUMEN de la Tesis de Sonia Favela Vara presentada como requisito parcial para la obtención del grado de MAESTRO EN CIENCIAS COMPUTACIONALES. Ensenada, Baja California, México. Diciembre de 1989.

MANEJO DE LA NEGACION EN PROGRAMACION LOGICA

Resumen aprobado:

M. en C. José Alfredo Amor Montaño
Director de Tesis

Se analizan las distintas propuestas para atacar el problema de la negación en Programación Lógica.

Al principio, se da una presentación de lo que es Programación Lógica, su funcionamiento y resultados generales, para tener un conocimiento uniforme y una notación estandarizada para el estudio subsecuente.

La Programación Lógica es un sistema de deducción, basado en la lógica de primer orden, que puede ser automatizado en una computadora.

Un programa lógico se compone de cláusulas definidas o cláusulas de programa de las que no se puede deducir ningún conocimiento negativo como consecuencia lógica, por esto surge la necesidad de dar significado extra-lógico de tal manera que se puedan inferir negaciones.

Los significados que se exponen en el presente trabajo son la Suposición del Mundo Cerrado, la Completación de un programa, la Circunscripción, la Suposición del Mundo Cerrado Generalizada y la Negación por Fracaso. Se dan su definición, resultados de consistencia y las relaciones entre ellos.

También se da a conocer la Negación por Inconsistencia que aumenta tipos de fórmulas a los programas y así se deduce conocimiento negativo como consecuencia lógica del programa, utilizando sólo la lógica de primer orden.

La Negación por Fracaso es la regla que se utiliza en PROLOG por medio de la resolución-LSDNF, así es que se explica de una forma amplia, dando resultados de correctez y completez y algunas semánticas para ella.

TESIS DEFENDIDA POR: SONIA FAVELA VARA

Y APROBADA POR EL SIGUIENTE COMITE:

M.C. José Alfredo Amor Montaño, Director del Comité

M.C. Fernando Favela Vara, Miembro del Comité

M.C. Hugo Homero Hidalgo Silva, Miembro del Comité

M.C. Francisco Suárez Vidal, Miembro del Comité

Ing. Salvador Castañeda Avila, Jefe del Centro de Cómputo Electrónico

Dr. Héctor Echavarría Heras, Director Académico Interino

Diciembre 7, 1989

CENTRO DE INVESTIGACION CIENTIFICA Y DE
EDUCACION SUPERIOR DE ENSENADA

CENTRO DE COMPUTO

MANEJO DE LA NEGACION EN PROGRAMACION LOGICA

TESIS

QUE PARA CUBRIR PARCIALMENTE LOS REQUISITOS NECESARIOS PARA OBTENER EL GRADO DE MAESTRO EN CIENCIAS PRESENTA:

SONIA FAVELA VARA

ENSENADA, B.C., DICIEMBRE DE 1989.

A Ti por mí.

A mis papás por todo lo que me han dado, entre otras cosas: la vida.

A Fernando, Jesús, Elia y Alejandra por ser mis primeros amigos.

A tí Rogelio porque "yo me casé contigo".

Mi profundo agradecimiento a José Alfredo Amor por esas mil horas de trabajo y ayuda para que pudiera terminar esta tesis.

CONTENIDO

CONTENIDO (Continuación)

CONTENIDO (Continuación)

LISTA DE FIGURAS

MANEJO DE LA NEGACION EN PROGRAMACION LOGICA

I. INTRODUCCION.

La Programación Lógica surge a partir de concebir a la lógica de predicados como lenguaje de programación. Durante la evolución de la Programación Lógica, Gödel, Herbrand y Skolem aportaron procedimientos de prueba al cálculo de predicados y señalaron que éste tiene todo el poder expresivo de un lenguaje. Más tarde se mostró que cualquier fórmula bien formada del cálculo de predicados de primer orden puede ser convertida en un conjunto de cláusulas. Sin embargo, aunque el cálculo de predicados es un lenguaje formal en el que se pueden expresar una gran variedad de enunciados, fue diseñado como un formalismo matemático de razonamiento humano más que como un lenguaje de cómputo o una máquina de inferencia automática. Robinson[1965] introdujo el principio de resolución, que es la única regla de inferencia requerida para construir un sistema de inferencia correcto y completo para el cálculo de predicados de primer orden en forma de cláusulas. En 1969, Green afirmó que tal procedimiento puede ser visto como un sistema de pregunta-respuesta. En 1971, Kowalski y Kuehner mejoraron más el algoritmo de resolución original introduciendo la resolución lineal con función de selección (resolución-LS). Kowalski fue capaz de mostrar que el subconjunto de las cláusulas de Horn del cálculo de

predicados de primer orden puede ser implementado eficientemente por computadora. Más tarde, el subconjunto de las cláusulas de Horn del cálculo de predicados junto con el mecanismo de resolución, se convirtió propiamente en un lenguaje computacional y, quizás de forma más importante, en un mecanismo de deducción automática. Este concepto es entonces conocido como Programación Lógica; un ejemplo de un sistema de Programación Lógica es PROLOG (PROgramar en LOGica), diseñado por el grupo de investigación de Colmerauer en Marsella en 1972.

La diferencia fundamental entre la Programación Lógica y la programación convencional es que la primera requiere la descripción de la estructura lógica de los problemas y la segunda que se indique cómo trabaja la computadora para resolverlos.

Hay dos interpretaciones básicas de Programación Lógica: la interpretación declarativa, que ve a la programación lógica como el cálculo de predicados original, y la interpretación de procedimiento, que ve a ésta como un lenguaje de programación.

En la interpretación declarativa se tiene que los probadores de teoremas por resolución son sistemas de refutación, esto es, la negación de una fórmula a ser probada se suma a los axiomas y se deriva una contradicción.

La interpretación de procedimiento da un carácter operacional a los programas lógicos, consiste esencialmente en ver a las metas como conjuntos de llamadas a procedimientos,

cada una de las cuales es procesada llamando a un procedimiento apropiado. La concepción de una interpretación de procedimiento fue quizás el avance más importante en lógica computacional que dió credibilidad a la lógica como un lenguaje de programación.

Posiblemente la ventaja más significativa del uso de la lógica como lenguaje de programación es que el contenido descriptivo de un programa lógico no depende de ninguna suposición acerca del mecanismo de ejecución; esto hace posible desarrollar la lógica del programa y los atributos de funcionamiento en una forma más separada que la que puede lograrse usando los lenguajes orientados a la máquina.

La resolución-LSD (resolución-LS para cláusulas definidas) se aplica sólo a cláusulas de Horn con exactamente una meta, así que usando resolución-LSD nunca podemos deducir información negativa, de aquí la importancia del uso de la negación para agregar significado a los recursos del programador.

Nuestro objetivo al llevar a cabo esta investigación fue el de analizar las distintas propuestas para atacar el problema de la negación en Programación Lógica.

En este trabajo primero se da una descripción de lo que es Programación Lógica, explicando el funcionamiento de la resolución-LSD y dando los resultados principales de correctez y completez para este sistema.

Hay dos formas de obtener datos negativos, una es ampliar

nuestro método deductivo a fórmulas más generales que puedan incluir negación, y otro es dar significados no-lógicos o extralógicos a los programas suponiendo, por ejemplo, que si *A* no se puede deducir entonces *no A* es válido en el sistema.

Nosotros expondremos los significados extralógicos más conocidos que se han dado como son la Suposición del Mundo Cerrado, la Completación de un programa, la Circunscripción, la Suposición del Mundo Cerrado Generalizada. Después expondremos el funcionamiento de la Negación por Fracaso que se aplica a programas normales (programas que incluyen negación) y presentaremos su relación con las semánticas anteriores. Por último mostraremos la Negación por Inconsistencia que es totalmente compatible con la negación en lógica clásica y se aplica a conjuntos de fórmulas de tipo más general.

II. PROGRAMACION LOGICA.

II.1 PRELIMINARES.

Como se describió antes, la Programación Lógica significa programar en el subconjunto de las cláusulas de Horn del cálculo de predicados de primer orden, usando el principio de resolución. A continuación resumiremos las definiciones con las que trabajaremos, basándonos en la notación utilizada por Lloyd[1987].

II.1.1 TEORIAS DE PRIMER ORDEN.

La lógica de primer orden tiene dos aspectos: sintáctico y semántico. El aspecto sintáctico se ocupa de las fórmulas bien formadas admitidas por la gramática de un lenguaje formal, así como de la teoría de las pruebas. La semántica se interesa en el significado asignado a las fórmulas bien formadas y a los símbolos que contienen.

Una *teoría de primer orden* consiste de un alfabeto, un lenguaje de primer orden, un conjunto de axiomas y un conjunto de reglas de inferencia.

Un *alfabeto* consiste de siete clases de símbolos:

(a) constantes

(b) símbolos funcionales

(c) símbolos de predicados

(d) variables

(e) conectivos

(f) cuantificadores

(g) símbolos de puntuación.

Las clases (d) a (g) son las mismas para cada alfabeto, mientras que las clases (a) a (c) varían de alfabeto a alfabeto. Para cualquier alfabeto, solo las clases (a) y (b) pueden ser vacías. Los *conectivos* son $\neg$, $\wedge$, $\vee$, $\rightarrow$ y $\leftrightarrow$, los *cuantificadores* son $\exists$ y $\forall$, y los *símbolos de puntuación* son "(", ")" y ",". Para permitir fórmulas sin paréntesis, adoptaremos la siguiente jerarquía de precedencia, con la precedencia mayor hasta arriba:

$\neg$, $\forall$, $\exists$

$\vee$

$\wedge$

$\rightarrow$, $\leftrightarrow$

A continuación definiremos el lenguaje de primer orden dado por un alfabeto.

Una *expresión* es una sucesión finita de símbolos del lenguaje.

Un *término* es definido inductivamente como sigue:

1. Una variable es un término.
2. Una constante es un término.
3. Si f es una función n-aria y $t_1,\ldots,t_n$ son términos, entonces $f(t_1,\ldots,t_n)$ es un término.
4. Una expresión es un término solo si lo es con base en 1, 2, 3.

Una *fórmula (bien formada)* es definida inductivamente como sigue:

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, llamada *fórmula atómica* o *átomo*.
2. Si F y G son fórmulas, entonces lo son $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$ y $(F \leftrightarrow G)$.
3. Si F es una fórmula y x es una variable, entonces $(\forall xF)$ y $(\exists xF)$ son fórmulas.
4. Una expresión es una fórmula solo si lo es con base en 1, 2, 3.

Usualmente es conveniente escribir la fórmula $(F \rightarrow G)$ como $(G \leftarrow F)$.

El *lenguaje de primer orden* dado por un alfabeto consiste del conjunto de todas las fórmulas construídas utilizando los símbolos del alfabeto.

El significado informal de los cuantificadores y conectivos es como sigue:

$\neg$ es negación,

$\wedge$ es conjunción (y),

$\vee$ es disyunción inclusiva (o),

$\rightarrow$ es implicación,

$\leftrightarrow$ es equivalencia,

$\exists$ es el cuantificador existencial, $\exists x$ significa que hay un individuo en el universo del discurso,

$\forall$ es el cuantificador universal, $\forall x$ significa para todo individuo en el universo del discurso.

El *alcance* de $\forall x$ ($\exists x$) en $\forall x F$ ($\exists x F$) es F. Una *ocurrencia acotada* de una variable en una fórmula es una ocurrencia inmediatamente después de un cuantificador o una ocurrencia en el alcance de un cuantificador, que tiene la misma variable inmediatamente después del cuantificador. Cualquier otra ocurrencia de una variable es *libre*.

Una *fórmula cerrada* es una fórmula que no tiene ocurrencias libres de ninguna variable.

Si F es una fórmula, entonces $\forall$(F) denota la *cerradura universal* de F, que es la fórmula cerrada que se obtiene al aumentar un cuantificador universal para cada variable con ocurrencias libres en F. Análogamente, $\exists$(F) denota la *cerradura existencial* de F, que se obtiene aumentando un cuantificador existencial para cada variable con ocurrencias libres en F.

Una *literal* es un átomo o la negación de un átomo. Una *literal positiva* es un átomo. Una *literal negativa* es la negación de un átomo.

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 es una literal y $x_1, \ldots, x_s$ son todas las variables que ocurren en $L_1 \vee \ldots \vee L_m$.

Como las cláusulas son tan comunes en programación lógica, será conveniente adoptar una notación clausal

especial. Denotaremos la cláusula

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

donde $A_1, \ldots, A_k, B_1, \ldots, B_n$ son átomos y $x_1, \ldots, x_s$ son todas las variables que ocurren en estos átomos, por

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

Así, en la notación clausal, se asume que todas las variables están universalmente cuantificadas, las comas en el antecedente $B_1, \ldots, B_n$ denotan conjunción y las comas en el consecuente $A_1, \ldots, A_k$ denotan disyunción. Estas convenciones están justificadas porque

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

es lógicamente equivalente a

$$\forall x_1 \ldots \forall x_s (A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_n)$$

Una *cláusula de programa definido* es una cláusula de la forma

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

que contiene precisamente un átomo en su consecuente. A es llamado la *cabeza* y $B_1, \ldots, B_n$ es llamado el *cuerpo* de la cláusula del programa.

Una *cláusula unitaria* es una cláusula de la forma

$$A \leftarrow$$

es decir, una cláusula de programa definido con un cuerpo vacío.

Un *programa definido* es un conjunto finito de cláusulas de programa definido.

En un programa definido, el conjunto de todas las cláusulas del programa con el mismo símbolo de predicado p en

la cabeza se llama la *definición* de p.

Una *meta definida* es una cláusula de la forma

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

esto es, una cláusula que tiene un consecuente vacío. Cada B_i $(i = 1,\ldots,n)$ se llama una *submeta* de la meta.

Si $y_1,\ldots,y_r$ son las variables de la meta $\leftarrow B_1,\ldots,B_n$ entonces su notación clausal es una abreviatura para

$$\forall y_1 \ldots \forall y_r (\neg B_1 \vee \ldots \neg B_n)$$

o, equivalentemente,

$$\forall y_1 \ldots \forall y_r \neg (B_1 \wedge \ldots \wedge B_n)$$

o, equivalentemente,

$$\neg \exists y_1 \ldots \exists y_r (B_1 \wedge \ldots \wedge B_n)$$

La *cláusula vacía*, denotada $\square$, sería la cláusula con antecedente y consecuente vacíos. Esta cláusula será entendida como el paso siguiente a una contradicción.

Una *cláusula de Horn* es una cláusula que es una cláusula de programa definido o una meta definida.

II.1.2 INTERPRETACIONES Y MODELOS.

Una *pre-interpretación* de un lenguaje de primer orden L consiste de lo siguiente:

1. Un conjunto no vacío D, llamado el *dominio* de la pre-interpretación.
2. Para cada constante en L, la asignación de un elemento en D.
3. Para cada símbolo funcional n-ario en L, la

asignación de una función de D^n a D.

Una *interpretación* I de un lenguaje de primer orden L consiste de una pre-interpretación J de L, con dominio D, junto con lo siguiente: para cada símbolo de predicado n-ario en L, la asignación de una función de D^n en {verdadero,falso} (o, equivalentemente, una relación sobre D^n). Decimos que I está *basada sobre* J.

Sea J una pre-interpretación de un lenguaje de primer orden L. Una *asignación de variables (con respecto a J)* es una asignación para cada variable en L, de un elemento en el dominio de J.

Sea J una pre-interpretación de un lenguaje de primer orden L, con dominio D, y sea V una asignación de variables. La *asignación de términos (con respecto a J y V)* de los términos en L se define como sigue:

1. A cada variable se le da su asignación de acuerdo a V.
2. A cada constante se le da su asignación de acuerdo a J.
3. Si $t'_1,\ldots,t'_n$ son las asignaciones de términos de $t_1,\ldots,t_n$ y f' la asignación del símbolo funcional n-ario f con respecto a J, entonces $f'(t'_1,\ldots,t'_n) \in D$ es la asignación de términos de $f(t_1,\ldots t_n)$.

Sea J una pre-interpretación de un lenguaje de primer orden L, V una asignación de variables con respecto a J, y A un átomo, supóngase que A es $p(t_1,\ldots,t_n)$ y $d_1,\ldots,d_n$ en el

dominio de J son las asignaciones de términos de $t_1,\ldots,t_n$ con respecto a J y V entonces $A_{J,V} = p(d_1,\ldots,d_n)$ es la *J-instancia de A con respecto a V*.

Sea I una interpretación de un lenguaje de primer orden L, con dominio D, y sea V una asignación de variables. Entonces a una fórmula en L se le puede dar un *valor de verdad*, verdadero o falso, (*con respecto a I y V*) como sigue:

1. Si la fórmula es un átomo $p(t_1,\ldots,t_n)$ entonces el valor de verdad se obtiene calculando el valor de $p'(t'_1,\ldots,t'_n)$, donde p' es la función asignada a p por I y $t'_1,\ldots,t'_n$ son las asignaciones de término de $t_1,\ldots,t_n$ con respecto a I y V.

2. Si la fórmula tiene la forma $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$, o $F \leftrightarrow G$, entonces el valor de verdad de la fórmula está dado por la siguiente tabla, en donde verdadero se representa por V y falso por F:

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
V	V	F	V	V	V	V
V	F	F	F	V	F	F
F	V	V	F	V	V	F
F	F	V	F	F	V	V

3. Si la fórmula tiene la forma $\exists xF$, entonces el valor de verdad de la fórmula es verdadero si hay $d \in D$ tal que F tiene valor de verdad verdadero con respecto a I y $V(x/d)$, donde $V(x/d)$ es V excepto que se asigna d a x; de lo contrario, su valor de verdad es falso.

4. Si la fórmula tiene la forma $\forall xF$, entonces el valor de verdad de la fórmula es verdadero si, para toda $d \in D$, tenemos

que F tiene el valor de verdad verdadero con respecto a I y V(x/d); de lo contrario, su valor de verdad es falso.

Claramente el valor de verdad de una fórmula cerrada no depende de la asignación de variables, entonces podemos hablar del valor de verdad de una fórmula cerrada con respecto a una intepretación. Si el valor de verdad de una fórmula cerrada con respecto a una interpretación es verdadero (falso), decimos que la fórmula es verdadera (falsa) con respecto a la interpretación.

Sea I una interpretación de un lenguaje de primer orden L y sea F una fórmula cerrada de L, entonces I es un *modelo* de F si F es verdadero con respecto a I.

Los *axiomas* de una teoría de primer orden son un subconjunto fijo de fórmulas cerradas en el lenguaje de la teoría. Por ejemplo, las teorías de primer orden en las que estamos más interesados tienen cláusulas de programa como axiomas.

Sea T una teoría de primer orden y sea L el lenguaje de T. Un *modelo* de T es una interpretación para L que es un modelo para cada axioma de T. Si T tiene un modelo, decimos que T es *consistente*.

Sea S un conjunto de fórmulas cerradas de un lenguaje de primer orden L y sea I una interpretación de L, decimos que I es un *modelo* de S si I es un modelo de cada fórmula de S.

Sea S un conjunto de fórmulas cerradas y F una fórmula cerrada de un lenguaje de primer orden L, decimos que F es una

consecuencia lógica de S si, para cada interpretación I de L, si I es modelo de S entonces I es un modelo de F.

Teorema 1. Sea S un conjunto de fórmulas cerradas y F una fórmula de un lenguaje de primer orden L, entonces F es una consecuencia lógica de S si y solo si S ∪ {¬F} no tiene modelo.

Aplicando estas definiciones a programas, vemos que cuando damos una meta G al sistema, con el programa P cargado, queremos que el sistema nos muestre que cada interpretación de P ∪ {G} no es un modelo. Pero esto parece un problema gigantesco, sin embargo, veremos que hay una clase de interpretaciones mucho menor y más conveniente a investigar, para mostrar que P ∪ {G} no tiene modelo. Estas son las tan llamadas interpretaciones de Herbrand.

Sea L un lenguaje de primer orden. El *universo de Herbrand* U_L de L es el conjunto de todos los términos sin variables, que pueden ser formados utilizando las constantes y símbolos funcionales que aparecen en L. (En el caso de que L no tenga constantes, aumentamos una constante, digamos c, para formar los términos sin variables.)

Sea L un lenguaje de primer orden. La *base de Herbrand* B_L de L es el conjunto de todos los átomos sin variables que se pueden formar utilizando símbolos de predicados de L con términos sin variables del universo de Herbrand como argumentos.

Sea L un lenguaje de primer orden. La

pre-interpretación de Herbrand de L es la pre-interpretación dada por lo siguiente:

1. El dominio de la pre-interpretación es el universo de Herbrand U_L.
2. Las constantes en L son asignadas a sí mismas en U_L.
3. Si f en un símbolo funcional n-ario en L, entonces la función de $(U_L)^n$ en U_L definida por

$$(t_1,\ldots,t_n) \rightarrow f(t_1,\ldots,t_n)$$

es asignada a f.

Una *interpretación de Herbrand* de L es cualquier interpretación basada en la pre-intepretación de Herbrand de L.

Como para las interpretaciones de Herbrand, las asignaciones de constantes y símbolos funcionales son fijas, es posible identificar una interpretación de Herbrand con un subconjunto de la base de Herbrand, que sería el conjunto de todos los átomos sin variables que son verdaderos con respecto a la interpretación.

Sea L un lenguaje de primer orden y S un conjunto de fórmulas cerradas de L. Un *modelo de Herbrand* de S es una interpretación de Herbrand de L que es modelo de S.

Por lo general será conveniente referirse, abusando del lenguaje, a una interpretación de un conjunto S de fórmulas, en lugar de una interpretación del lenguaje de primer orden en el que las fórmulas están expresadas. Normalmente, asumimos que este lenguaje de primer orden está definido por las

constantes, símbolos funcionales y símbolos de predicados que aparecen en S. Con base en esto, nos referiremos al universo de Herbrand U_S y a la base de Herbrand B_S de S, además identificaremos a las interpretaciones de Herbrand de S con los subconjuntos de B_S.

Teorema 2. Sea S un conjunto de cláusulas y supongamos que S tiene un modelo, entonces S tiene un modelo de Herbrand.

Teorema 3. Sea S un conjunto de cláusulas, entonces S no tiene modelo si y solo si no tiene modelo de Herbrand.

Si S es un conjunto de fórmulas cerradas arbitrario, no es generalmente posible mostrar que S no tiene modelo restringiéndonos a los modelos de Herbrand.

Ejemplo 1. Sea $S = \{p(a), \exists x \neg p(x)\}$. Nótese que la segunda fórmula en S no es una cláusula. S tiene modelo, por ejemplo consideremos $D = \{0,1\}$, asignando 0 a a, y a p la función que manda 0 a verdadero y 1 a falso. Claramente esto da un modelo de S, sin embargo, S no tiene modelo de Herbrand ya que las únicas interpretaciones de Herbrand de S son $\emptyset$ y $\{p(a)\}$ que no son modelos de S.

II.1.3 UNIFICACION.

El propósito principal de un sistema de programación lógica es calcular las respuestas, estas respuestas se calculan por unificación con base en resolución.

Una *substitución* 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 es

un término distinto de v_i y las variables $v_1,\ldots,v_n$ son distintas. θ es una *substitución básica* si los t_i son términos sin variables. θ es una *substitución de puras variables* si los t_i son variables.

Una *expresión* es un término, una literal o una conjunción o disyunción de literales. Una *expresión simple* es un término o un átomo.

Sea $\theta = \{v_1/t_1,\ldots,v_n/t_n\}$ una substitución y E una expresión, entonces $E\theta$, la *instancia* de E por θ, es la expresión obtenida a partir de E al reemplazar cada ocurrencia de la variable v_i en E por el término t_i $(i = 1,\ldots,n)$. Si $E\theta$ no tiene variables, entonces $E\theta$ es una *instancia sin variables* de E.

Si $S = \{E_1,\ldots,E_n\}$ es un conjunto finito de expresiones y θ es una substitución, entonces $S\theta$ denota al conjunto $\{E_1\theta,\ldots,E_n\theta\}$.

Sean $\theta = \{u_1/s_1,\ldots,u_m/s_m\}$ y $\sigma = \{v_1/t_1,\ldots,v_n/t_n\}$ substituciones, entonces la *composición* $\theta\sigma$ de θ y σ es la substitución obtenida del conjunto

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

borrando cualquier elemento $u_i/s_i\sigma$ para el cual $u_i = s_i\sigma$, y cualquier elemento v_j/t_j para el cual $v_j \in \{u_1,\ldots,u_m\}$.

Ejemplo 2. Sea $\theta = \{x/f(y),\ y/z\}$ y $\sigma = \{x/a,\ y/b,\ z/y\}$, entonces $\theta\sigma = \{x/f(b),\ z/y\}$.

La substitución dada por el conjunto vacío se llama la *substitución identidad*, que denotaremos por $\emptyset$. Nótese que

$E\emptyset = E$, para todas las expresiones E.

A continuación tenemos las propiedades de las substituciones.

Teorema 4. Sean θ, σ y γ substituciones, entonces:

1. $\theta\emptyset = \emptyset\theta = \theta$.
2. $(E\theta)\sigma = E(\theta\sigma)$, para toda expresión E.
3. $(\theta\sigma)\gamma = \theta(\sigma\gamma)$.

Ejemplo 3. Sea $\theta = \{x/f(y),\ y/z\}$ y $\sigma = \{x/a,\ z/b\}$, entonces $\theta\sigma = \{x/f(y),\ y/b,\ z/b\}$. Sea $E = p(x,y,g(z))$, entonces $E\theta = p(f(y),z,g(z))$ y $(E\theta)\sigma = p(f(y),b,g(b)) = E(\theta\sigma)$.

Sean E y F expresiones, se dice que E y F son *variantes* si hay substituciones θ y σ tales que $E = F\theta$ y $F = E\sigma$, también se dice que E es una variante de F o F es una variante de E.

Ejemplo 4. $p(f(x,y),g(z),a)$ es una variante de $p(f(y,x),g(u),a)$, sin embargo $p(x,x)$ no es una variante de $p(x,y)$.

Sea E una expresión y V el conjunto de variables que ocurren en E, una *substitución de renombre* de E es una substitución de puras variables $\{x_1/y_1,\ldots,x_n/y_n\}$ tal que $\{x_1,\ldots,x_n\} \subseteq V$, las y_i son distintas y $(V\setminus\{x_1,\ldots,x_n\}) \cap \{y_1,\ldots,y_n\} = \emptyset$.

Teorema 5. Sean E y F expresiones que son variantes, entonces existen substituciones θ y σ tales que $E = F\theta$ y $F = E\sigma$, donde θ es una substitución de renombre de F y σ es una substitución de renombre de E.

Estamos particularmente interesados en substituciones que

unifiquen un conjunto de expresiones, esto es, que hagan todas las expresiones de un conjunto sintácticamente iguales. El concepto de unificación fue definido por Herbrand en 1930, y fue incorporado por Robinson[1965] en su principio de resolución.

Sea S un conjunto finito de expresiones simples, una substitución θ es llamada un *unificador* de S si Sθ tiene un elemento. Un unificador θ de S es llamado un *unificador más general* (umg) de S si, para cada unificador σ de S, existe una substitución γ tal que $\sigma = \theta\gamma$.

Ejemplo 5.

{p(f(x),a), p(y,f(w))} no es unificable, porque los segundos argumentos no se pueden unificar.

{p(f(x),z), p(y,a)} es unificable, ya que σ = {y/f(a), x/a, z/a} es un unificador, un unificador más general es θ = {y/f(x), z/a}, nótese que $\sigma = \theta$(x/a).

Se sigue de la definición de un umg que si θ y σ son umg's de $\{E_1,\ldots,E_n\}$, entonces $E_1\theta$ es una variante de $E_1\sigma$, y entonces $E_1\sigma$ puede ser obtenido a partir de $E_1\theta$ renombrando variables.

Sea S un conjunto finito de expresiones simples, el *conjunto discorde* de S se define como sigue: localizar la posición del símbolo más a la izquierda en el que no todas las expresiones en S tienen el mismo símbolo y extraer de cada expresión en S la subexpresión que empieza en esa posición; el conjunto de dichas expresiones es el conjunto discorde.

Ejemplo 6. Sea

$S = \{p(f(x),h(y),a),\ p(f(x),z,a),\ p(f(x),h(y),b)\}$,

entonces el conjunto discorde es $\{h(y),\ z\}$.

Algoritmo de unificación:

Sea S un conjunto finito de expresiones simples.

1. Poner $k = 0$ y $\sigma_0 = \emptyset$.

2. Si $S\sigma_k$ tiene un elemento, entonces parar; σ_k es un umg de S; de otro modo, encontrar el conjunto discorde D_k de $S\sigma_k$.

3. Si existen v y t en D_k tales que v es una variable que no ocurre en t (verificación de ocurrencia), entonces poner $\sigma_{k+1} = \sigma_k\{v/t\}$, incrementar k e ir a 2; de otro modo, parar, S no es unificable.

Ejemplo 7.

Sea $S = \{p(f(a),g(x)),\ p(y,y)\}$.

1. $\sigma_0 = \emptyset$.

2. $D_0 = \{f(a),\ y\}$, $\sigma_1 = \{y/f(a)\}$,

   $S\sigma_1 = \{p(f(a),g(x)),\ p(f(a),f(a))\}$.

3. $D_1 = \{g(x),\ f(a)\}$, entonces S no es unificable.

Sea $S = \{p(x,x),\ p(y,f(y))\}$.

1. $\sigma_0 = \emptyset$.

2. $D_0 = \{x,y\}$, $\sigma_1 = \{x/y\}$, $S\sigma_1 = \{p(y,y),p(y,f(y))\}$.

3. $D_1 = \{y,f(y)\}$; como y ocurre en f(y), S no es unificable.

Sea $S = \{p(a,x,h(g(z))),\ p(z,h(y),h(y))\}$.

1. $\sigma_0 = \emptyset$.

2. $D_0 = \{a,z\}$, $\sigma_1 = \{z/a\}$,

$S\sigma_1$ = {p(a,x,h(g(a))), p(a,h(y),h(y))}.

3. D_1 = {x,h(y)}, σ_2 = {z/a, x/h(y)},

$S\sigma_2$ = {p(a,h(y),h(g(a))),p(a,h(y),h(y))}.

4. D_2 = {g(a),y}, σ_3 = {z/a, x/h(g(a)), y/g(a)},

$S\sigma_3$ = {p(a,h(g(a)),h(g(a)))}, entonces S es unificable y σ_3 es un umg.

Teorema 6 (Teorema de unificación). Sea S un conjunto finito de expresiones simples. Si S es unificable, entonces el algoritmo de unificación termina y da un umg de S. Si S no es unificable, entonces el algoritmo de unificación termina y reporta este hecho.

La mayoría de los sistemas de PROLOG usan esencialmente el algoritmo de unificación anterior, pero sin la verificación de ocurrencia que es muy costoso; desde un punto de vista teórico, esto es un desastre porque destruye la correctez de la resolución-LSD.

II.1.4 PUNTOS FIJOS.

Cada programa definido está asociado con un mapeo monotónico que juega un papel muy importante en la teoría. Esta sección introduce los conceptos requeridos y los resultados concernientes a los mapeos monotónicos y sus puntos fijos.

Sea S un conjunto, una *relación* R sobre S es un subconjunto de SxS.

Usualmente usaremos notación infija escribiendo (x,y) $\in$ R

como xRy.

Una relación R sobre un conjunto S es un *orden parcial* si las siguientes condiciones se satisfacen:

1. xRx, para toda $x \in S$.

2. xRy y yRx implica $x = y$, para toda $x, y \in S$.

3. xRy y yRz implica xRz, para toda $x, y, z \in S$.

Ejemplo 8. Sea S un conjunto y $\mathcal{P}(S)$ el conjunto de todos los subconjuntos de S, entonces la inclusión de conjuntos, $\subseteq$, es un orden parcial sobre $\mathcal{P}(S)$.

Adoptaremos la notación estándar y usaremos $\leq$ para denotar un orden parcial.

Sea S un conjunto con un orden parcial $\leq$, entonces $a \in S$ es una *cota superior* de un subconjunto X de S si $x \leq a$, para toda $x \in X$. Similarmente, $b \in S$ es una *cota inferior* de X si $b \leq x$, para toda $x \in X$.

Sea S un conjunto con un orden parcial $\leq$, entonces $a \in S$ es la *mínima cota superior o supremo* de un subconjunto X de S si a es una cota superior de X y, para todas las cotas superiores a' de X, tenemos $a \leq a'$. Similarmente, $b \in S$ es la *máxima cota inferior o ínfimo* de un subconjunto X de S si b es una cota inferior de X y, para toda cota inferior b' de X, tenemos $b' \leq b$.

El supremo de X es único, si existe, y se denota por sup(X). Similarmente, el ínfimo de X es único, si existe, y se denota por inf(X).

Un conjunto parcialmente ordenado L es una *red completa*

si sup(X) e inf(X) existe para todo subconjunto X de L.

Denotaremos a sup(L), el *elemento tope*, con $\top$, y a inf(L), el *elemento base* de la red completa L, con $\bot$.

Ejemplo 9. En el ejemplo anterior, $\mathcal{P}(S)$ bajo $\subseteq$ es una red completa, de hecho, el supremo de una colección de subconjuntos de S es su unión y el ínfimo es su intersección. El elemento tope es S y el elemento base es $\emptyset$.

Sea L una red completa y T:L $\rightarrow$ L un mapeo, T es *monótona* si $x \leq y$ implica $T(x) \leq T(y)$.

Sea L una red completa y $X \subseteq L$, X es *dirigido* si cada subconjunto finito de X tiene una cota superior en X.

Sea L una red completa y T:L $\rightarrow$ L un mapeo, T es *contínua* si $T(\sup(X)) = \sup(T(X))$, para cada subconjunto dirigido X de L.

Nuestro interés en estas definiciones se debe a que para un programa definido P, la colección de todas las interpretaciones de Herbrand, tomando la contención como orden, forma una red completa de una manera natural y hay un mapeo contínuo asociado con P definido sobre esta red.

Sea L una red completa y T:L $\rightarrow$ L un mapeo, $a \in L$ es el *menor punto fijo* de T si a es un punto fijo (esto es, $T(a) = a$) y para todos los puntos fijos b de T, tenemos $a \leq b$. El *mayor punto fijo* se define de manera similar.

El teorema a continuación se debe a Tarski[1955].

Teorema 7. Sea L una red completa y T:L $\rightarrow$ L monótona, entonces T tiene un mínimo punto fijo, mpf(T), y un máximo

punto fijo, MPF(T), además:

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

$MPF(T) = sup\{x \mid T(x) = x\} = sup\{x \mid x \leq T(x)\}$.

Sea L una red completa y $T: L \to L$ monótona, entonces definimos:

$T\uparrow 0 = \bot$

$T\uparrow\alpha = T(T\uparrow(\alpha-1))$, si α es un ordinal sucesor

$T\uparrow\alpha = sup\{T\uparrow\beta \mid \beta < \alpha\}$, si α es un ordinal límite

$T\downarrow 0 = \top$

$T\downarrow\alpha = T(T\downarrow(\alpha-1))$, si α es un ordinal sucesor

$T\downarrow\alpha = inf\{T\downarrow\beta \mid \beta < \alpha\}$, si α es un ordinal límite

Teorema 8. Sea L una red completa y $T: L \to L$ monótona, entonces, para cualquier ordinal α, $T\uparrow\alpha \leq mpf(T)$ y $T\downarrow\alpha \geq MPF(T)$; además existen ordinales β_1 y β_2 tales que $\gamma_1 \geq \beta_1$ implica $T\uparrow\gamma_1 = mpf(T)$ y $\gamma_2 \geq \beta_2$ implica $T\downarrow\gamma_2 = MPF(T)$.

El menor α tal que $T\uparrow\alpha = mpf(T)$ se llama el *ordinal cerradura* de T, el siguiente resultado muestra que si T es contínua, el ordinal cerradura de T es menor o igual a ω.

Teorema 9. Sea L una red completa y $T: L \to L$ contínua, entonces $mpf(T) = T\uparrow\omega = sup\{T\uparrow\alpha \mid \alpha < \omega\}$.

El teorema análogo para MPF(T) no es verdadero, el ejemplo 11 en la siguiente sección es un contraejemplo.

II.2 PROGRAMAS DEFINIDOS.

En esta parte describiremos la semántica declarativa y la

semántica de procedimiento de los programas definidos.

II.2.1 SEMANTICA DECLARATIVA.

La teoría de los programas definidos es más simple que la teoría de las clases de programas que permiten literales negativas en los cuerpos de las metas, debido a que los programas definidos no permiten negaciones y nos podemos evitar las dificultades teóricas y prácticas del manejo de submetas negadas.

Teorema 10 (La propiedad de la intersección de modelos).

Sea P un programa definido y $\langle M_i \rangle_{i \in I}$ un conjunto no vacío de modelos de Herbrand de P, entonces $\cap_{i \in I} M_i$ es un modelo de Herbrand de P.

Como cada programa definido P tiene a B_p como modelo de Herbrand, el conjunto de todos los modelos de Herbrand de P es no vacío, entonces la intersección de todos los modelos de Herbrand de P existe y además es a su vez un modelo de Herbrand de P llamado el *modelo mínimo de Herbrand* de P; denotaremos este modelo por M_p. Esto no necesariamente se cumple si P no es programa definido.

Teorema 11. Sea P un programa definido, entonces

$$M_p = \{A \in B_p \mid A \text{ es una consecuencia lógica de P}\}.$$

Sea P un programa definido, entonces $\mathcal{P}(B_p)$, que es el conjunto de todas las interpretaciones de Herbrand de P, es una red completa bajo el orden parcial de la inclusión de conjuntos $\subseteq$. El elemento tope o supremo de esta red es B_p y

el elemento base o ínfimo es Ø. El supremo de cualquier conjunto de interpretaciones de Herbrand es la interpretación de Herbrand que es la unión de todas las interpretaciones de Herbrand en el conjunto. El ínfimo es la intersección.

Sea P un programa definido, el mapeo $T_P: \mathcal{P}(B_P) \rightarrow \mathcal{P}(B_P)$ se define como sigue: sea I una interpretación de Herbrand, entonces $T_P(I) = \{A \in B_P \mid A \leftarrow A_1,\ldots,A_n$ es una instancia sin variables de una cláusula en P y $\{A_1,\ldots,A_n\} \subseteq I\}$.

T_P es monótona, y da la liga entre la semántica declarativa y la de procedimiento de P.

Ejemplo 10. Sea P:

$$p(f(x)) \leftarrow p(x)$$

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

Sean $I_1 = B_p$, $I_2 = T_P(I_1)$, $I_3 = \emptyset$, entonces

$$T_P(I_1) = \{q(a)\} \cup \{p(f(t)) \mid t \in U_p\},$$

$T_P(I_2) = \{q(a)\} \cup \{p(f(f(t))) \mid t \in U_P\}$, y $T_P(I_3) = \emptyset$.

Teorema 12. Sea P un programa definido, entonces el mapeo T_P es contínuo.

Las interpretaciones de Herbrand que son modelos de P pueden ser caracterizadas en términos de T_P.

Teorema 13. Sea P un programa definido e I una interpretación de Herbrand de P, entonces I es un modelo de P si y solo si $T_P(I) \subseteq I$.

El siguiente teorema da una caracterización del modelo mínimo de Herbrand de un programa definido, utilizando puntos fijos, y se debe a Kleene.

Teorema 14 (La caracterización del modelo mínimo de Herbrand con puntos fijos). Sea P un programa definido, entonces $M_P = mpf(T_P) = T_P\uparrow\omega = \bigcup_{n<\omega} T_P\uparrow n$.

Sin embargo, puede pasar que $MPF(T_P) \neq T_P\downarrow\omega = \bigcap_{n<\omega} T_P\downarrow n$

Ejemplo 11. Sea P

$$p(f(x)) \leftarrow p(x)$$

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

entonces $T_P\downarrow\omega = \{q(a)\}$, pero $MPF(T_P) = \emptyset = T_P\downarrow(\omega + 1)$.

Ahora veremos la definición de una respuesta correcta, este concepto es central en programación lógica.

Sea P un programa definido y G una meta definida, una *respuesta* para $P \cup \{G\}$ es una substitución de variables de G.

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

La definición anterior de respuesta correcta nos da una descripción de la salida deseada a partir de un programa definido y una meta, un sistema de programación lógica puede contestar con substituciones o con la respuesta "no". Decimos que la respuesta "no" es *correcta* si $P \cup \{G\}$ tiene modelo.

Quisiéramos que una respuesta θ fuera correcta si y solo si $\forall((A_1\wedge\ldots\wedge A_k)\theta)$ es verdadero con respecto al modelo mínimo del programa, sin embargo este resultado general no es verdadero.

Ejemplo 12. Sea P

$$p(a) \leftarrow,$$

sea G la meta $\leftarrow p(x)$ y θ la substitución identidad, entonces $M_P = \{p(a)\}$ y entonces $\forall x p(x)\theta$ es verdadero en M_P, sin embargo, θ no es una respuesta correcta porque $\forall x p(x)\theta$ no es una consecuencia lógica de P.

Teorema 15. Sea P un programa definido y G una meta definida $\leftarrow A_1,\ldots,A_k$, supongamos que θ es una respuesta para $P \cup \{G\}$ tal que $(A_1 \wedge \ldots \wedge A_k)\theta$ no tiene variables, entonces las siguientes afirmaciones son equivalentes:

1. θ es correcto.
2. $(A_1 \wedge \ldots \wedge A_k)\theta$ es verdadero con respecto a cada modelo de Herbrand de P.
3. $(A_1 \wedge \ldots \wedge A_k)\theta$ es verdadero con respecto al modelo mínimo de Herbrand de P.

II.2.2 CORRECTEZ DE LA RESOLUCION-LSD.

En esta sección se introduce la semántica de procedimiento de los programas definidos.

Hay muchos procedimientos de refutación basados en la regla de inferencia de resolución, que son refinamientos del procedimiento original de Robinson[1965], el procedimiento de refutación que veremos fue descrito primero por Kowalski y fue llamado resolución-LSD que quiere decir resolución lineal con función de selección para cláusulas definidas.

Sea G la meta $\leftarrow A_1,\ldots,A_m,\ldots,A_k$ y C la

cláusula de programa $A \leftarrow B_1,\ldots,B_q$, entonces G' es el *resolvente* de G y C usando un umg θ, si se dan las siguientes condiciones:

1. A_m es un átomo, llamado el átomo *seleccionado*, en G.
2. θ es un umg de A_m y A.
3. G' es la meta $\leftarrow (A_1,\ldots,A_{m-1},B_1,\ldots,B_q,A_{m+1},\ldots,A_k)\theta$.

Sea P un programa definido y G una meta definida, una *derivación-LSD* para $P \cup \{G\}$ consiste de una sucesión (finita o infinita) $G_0 = G, G_1,\ldots$ de metas, una sucesión $C_1, C_2,\ldots$ de variantes de cláusulas de programa de P y una sucesión $\theta_1, \theta_2,\ldots$ de umg's tales que cada G_{i+1} es resolvente a partir de G_i y C_{i+1} usando θ_{i+1}.

Cada C_i es una variante de una cláusula de programa de tal manera que C_i no tenga variables que ya hayan aparecido en la derivación hasta G_{i-1}. Cada variante de cláusula de programa $C_1, C_2,\ldots$ se llama una *cláusula de entrada* de la derivación.

Una *refutación-LSD* para $P \cup \{G\}$ es una derivación-LSD finita para $P \cup \{G\}$ que tiene a la cláusula vacía $\square$ como la última meta en la derivación. Si $G_n = \square$, la refutación tiene *longitud n*.

Las derivaciones-LSD pueden ser *finitas* o *infinitas*. Una derivación-LSD finita puede tener éxito o fracasar. Una derivación-LSD *tiene éxito* si termina en la cláusula vacía, en otras palabras, una derivación tiene éxito si es una refutación. Una derivación-LSD *fracasa* si termina en una

meta no vacía con la propiedad de que el átomo seleccionado en esta meta no unifica con la cabeza de ninguna cláusula de programa. Más adelante se verán ejemplos de derivaciones con éxito, con fracaso e infinitas.

Sea P un programa definido, el *conjunto éxito* de P es el conjunto de todas las $A \in B_P$ tales que $P \cup \{\leftarrow A\}$ tiene una refutación-LSD.

El conjunto éxito es la contraparte de procedimiento del modelo mínimo de Herbrand, similarmente tenemos la contraparte de procedimiento de una respuesta correcta.

Sea P un programa definido y G una meta definida, una *respuesta computada* θ para $P \cup \{G\}$ es la substitución obtenida al restringir la composición $\theta_1 \ldots \theta_n$ a las variables de G, donde $\theta_1, \ldots, \theta_n$ es la sucesión de los umg's usados en una refutación-LSD para $P \cup \{G\}$.

Teorema 16 (Correctez de la resolución-LSD).

Sea P un programa definido y G una meta definida, entonces cada respuesta computada para $P \cup \{G\}$ es una respuesta correcta para $P \cup \{G\}$.

Corolario 1. Sea P un programa definido y G una meta definida, supongamos que existe una refutación-LSD para $P \cup \{G\}$, entonces $P \cup \{G\}$ no tiene modelo.

Corolario 2. El conjunto éxito de un programa definido está contenido en su modelo mínimo de Herbrand.

Podemos mostrar que si $A \in B_P$ y $P \cup \{\leftarrow A\}$ tiene una refutación-LSD de longitud n, entonces $A \in T_P \uparrow n$, este

resultado se debe a Apt y van Emden[1982].

Si A es un átomo, [A] = {A'∈ B_P | A' = Aθ, para alguna substitución θ}.

Teorema 17. Sea P un programa definido y G una meta definida ← $A_1,\ldots,A_k$, supongamos que P ∪ {G} tiene una refutación-LSD de longitud n y $\theta_1,\ldots,\theta_n$ es la sucesión de umg's de la refutación-LSD, entonces tenemos que $\bigcup_{j=1}^{k} [A\theta_j\theta_1\ldots\theta_n] \subseteq T_P \uparrow n$.

Ahora veremos el problema de la verificación de ocurrencia, que es muy costoso y en la mayoría de los sistemas de PROLOG no se implementa debido a que es muy raro que se requiera; aunque esto es cierto, su omisión puede causar serias dificultades.

Ejemplo 13. Considérese el programa

m ← p(x,x)

p(x,f(x)),

dada la meta ← m, un sistema PROLOG sin la verificación de ocurrencia contestará "si" (es decir, Ø es una respuesta computada). Esta respuesta es incorrecta porque m no es una consecuencia lógica del programa, el problema surge porque, sin la verificación de ocurrencia, el algoritmo de unificación del sistema PROLOG unificará erróneamente a p(x,x) y p(y,f(y)).

Así vemos que la ausencia de la verificación de ocurrencia ha destruído uno de los principios en que se basa la programación lógica: la correctez de la resolución-LSD.

Ejemplo 14. Considérese el programa

$$m \leftarrow p(x,x)$$

$$p(x,f(x)) \leftarrow p(x,x),$$

esta vez un sistema PROLOG sin la verificación de ocurrencia se irá a un loop infinito en el algoritmo de unificación porque intentará utilizar una sustitución "circular" hecha en el segundo paso del cómputo.

II.2.3 COMPLETEZ DE LA RESOLUCION-LSD.

El siguiente resultado de completez se debe a Apt y van Emden[1982].

Teorema 18. El conjunto éxito de un programa definido es igual a su modelo mínimo de Herbrand.

El resultado que está a continuación fue probado primero por Hill en 1974.

Teorema 19. Sea P un programa definido y G una meta definida, supóngase que $P \cup \{G\}$ no tiene modelo, entonces existe una refutación-LSD para $P \cup \{G\}$.

Ahora analizaremos las respuestas correctas, para ver que cada respuesta correcta es una instancia de una respuesta computada.

Teorema 20. Sea P un programa definido y A un átomo, supóngase que $\forall(A)$ es una consecuencia lógica de P, entonces existe una refutación-LSD para $P \cup \{\leftarrow A\}$ con la substitución identidad como la respuesta computada.

El resultado principal de completez es el siguiente y se

debe a Clark.

Teorema 21 (Completez de la Resolución-LSD). Sea P un programa definido y G una meta definida, para cada respuesta correcta θ para P $\cup$ ⟨G⟩, existe una respuesta computada σ para P $\cup$ ⟨G⟩ y una substitución γ tal que $\theta = \sigma\gamma$.

Así, para cada respuesta correcta θ para P $\cup$ ⟨G⟩ hay una respuesta computada σ para P $\cup$ ⟨G⟩, más general o igual que θ.

II.2.4 INDEPENDENCIA DE LA REGLA DE COMPUTO.

Una *regla de cómputo* es una función de selección que va del conjunto de metas definidas al conjunto de átomos, tal que, el valor de la función para una meta es un átomo en esa meta, que se llama el átomo *seleccionado*.

Sea P un programa definido, G una meta definida y R una regla de cómputo, una *derivación-LSD* para P $\cup$ ⟨G⟩ *vía R* es una derivación-LSD para P $\cup$ ⟨G⟩ en la que se utiliza la regla de cómputo R para seleccionar átomos.

Es importante notar que usar una regla de cómputo para seleccionar átomos en una derivación-LSD es una restricción, ya que si la misma meta ocurre en diferentes lugares entonces la regla de cómputo siempre seleccionará el mismo átomo de esa meta; por lo tanto, hay derivaciones-LSD que no son derivaciones-LSD vía R, para ninguna regla de cómputo R.

Sea P un programa definido, G una meta definida y R una regla de cómputo, una *refutación-LSD* para P $\cup$ ⟨G⟩ *vía R* es una refutación-LSD para P $\cup$ ⟨G⟩ en la que se utiliza la regla de

cómputo R para seleccionar átomos.

Sea P un programa definido, G una meta definida y R una regla de cómputo, una *respuesta computada-R* para P ∪ {G} es una respuesta computada para P ∪ {G} que proviene de una refutación-LSD para P ∪ {G} via R.

Teorema 22 (Independencia de la Regla de Cómputo). Sea P un programa definido y G una meta definida, supongamos que hay una refutación-LSD para P ∪ {G} con respuesta computada σ, entonces, para cada regla de cómputo R, existe una refutación-LSD para P ∪ {G} via R con respuesta computada-R σ' tal que $G\sigma'$ es una variante de $G\sigma$.

Sea P un programa definido y R una regla de cómputo, el *conjunto éxito-R* de P es el conjunto de todas las $A \in B_P$ tales que P ∪ {← A} tiene una refutación-LSD via R.

Teorema 23. Sea P un programa definido y R una regla de cómputo, entonces el conjunto éxito-R de P es igual a su modelo mínimo de Herbrand.

Teorema 24. Sea P un programa definido, G una meta definida y R una regla de cómputo, supóngase que P ∪ {G} no tiene modelo, entonces existe una refutación-LSD para P ∪ {G} via R.

Teorema 25 (Completez Fuerte de Resolución-LSD). Sea P un programa definido, G una meta definida y R una regla de cómputo, entonces para cada respuesta correcta θ para P ∪ {G}, existe una respuesta computada-R σ para P ∪ {G} y una substitución γ tal que $\theta = \sigma\gamma$.

II.2.5 PROCEDIMIENTOS DE REFUTACION-LSD.

Aquí consideraremos las estrategias que puede adoptar un sistema de programación lógica en su búsqueda de una refutación.

Sea P un programa definido y G una meta definida, un *árbol-LSD* para P ∪ ⟨G⟩ es un árbol que satisface lo siguiente:

1. Cada nodo del árbol es una meta definida (posiblemente la vacía).
2. El nodo raíz es G.
3. Sea $\leftarrow A_1,\ldots,A_m,\ldots,A_k$ $(k \geq 1)$ un nodo en el árbol y supóngase 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 con un 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 la cláusula vacía no tienen hijos.

Cada rama del árbol-LSD es una derivación para P ∪ ⟨G⟩, las ramas que corresponden a derivaciones con éxito se llaman *ramas éxito*, las ramas que corresponden a derivaciones infinitas se llaman *ramas infinitas* y las ramas que corresponden a derivaciones con fracaso se llaman *ramas fracaso*.

Sea P un programa definido, G una meta definida y R una regla de cómputo, el *árbol-LSD* para P ∪ ⟨G⟩ *vía R* es el árbol-LSD para P ∪ ⟨G⟩ en el que los átomos seleccionados son aquellos seleccionados por R.

Ejemplo 15. Considérese el programa

1. p(x,z) ← q(x,y),p(y,z)
2. p(x,x) ←
3. q(a,b) ←

A continuación mostramos dos árboles-LSD para este programa y la meta ← p(x,b). En la primera figura se utiliza la regla de cómputo de PROLOG estándar (selecciona el átomo de más a la izquierda), en la segunda figura la regla de cómputo utilizada es la que siempre selecciona el átomo de más a la derecha. Las cláusulas del programa utilizadas están indicadas con un número, los átomos seleccionados están subrayados y se muestran las ramas éxito, fracaso e infinitas. Nótese que el primer árbol es finito, mientras que el segundo es infinito, cada árbol tiene dos ramas éxito correspondientes a las respuestas {x/a} y {x/b}.

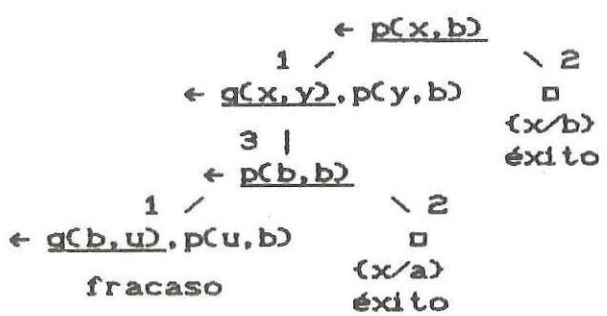


Figura 1. Arbol-LSD finito.

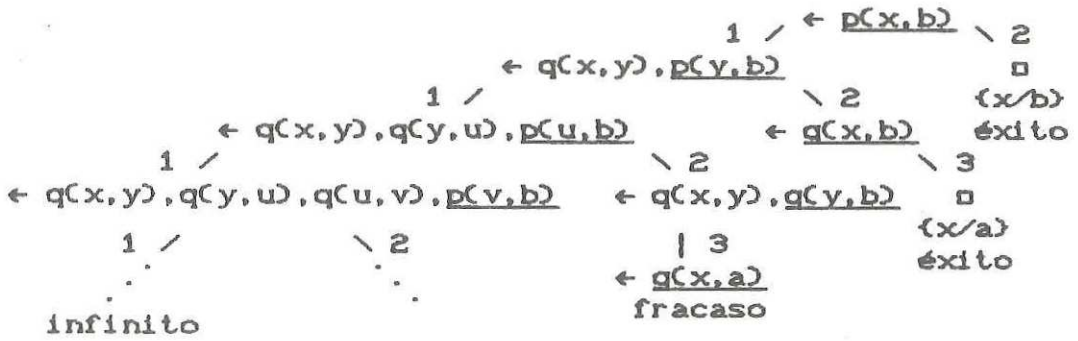


Figura 2. Arbol-LSD infinito.

Este ejemplo muestra que la elección de la regla de cómputo tiene una gran influencia en el tamaño y la estructura de los árboles-LSD correspondientes, sin embargo, sin importar qué regla de cómputo se elija, si P ∪ {G} no tiene modelo, entonces el árbol-LSD correspondiente tiene al menos una rama éxito.

Teorema 26. Sea P un programa definido, G una meta definida y R una regla de cómputo, supóngase que P ∪ {G} no tiene modelo, entonces el árbol-LSD para P ∪ {G} vía R tiene al menos una rama éxito.

Teorema 27. Sea P un programa definido, G una meta definida y R una regla de cómputo, entonces cada respuesta correcta θ para P ∪ {G} está desplegada en el árbol-LSD para P ∪ {G} vía R.

Desplegada significa que, dado θ, hay una rama éxito tal que θ es una instancia de la respuesta computada para la refutación correspondiente a esa rama.

Mientras que dos árboles-LSD pueden tener muy diferente tamaño y estructura, son esencialmente los mismos con respecto a las ramas éxito.

Teorema 28. Sea P un programa definido y G una meta definida entonces todos los árboles-LSD para P U {G} tienen un número infinito de ramas éxito, o, tienen el mismo número finito de ramas éxito.

Una *regla de búsqueda* es una estrategia para examinar árboles-LSD y así encontrar ramas éxito. Un *procedimiento de refutación-LSD* se especifica con una regla de cómputo junto con una regla de búsqueda.

Los sistemas de PROLOG estándar utilizan la regla de cómputo que siempre selecciona el átomo más a la izquierda en una meta junto con la regla de búsqueda primero-a-profundidad (depth-first). La regla de búsqueda se implementa usando una pila de metas, una instancia de la pila de metas representa la rama que está siendo investigada. El cómputo se vuelve esencialmente una sucesión mezclada de introducciones (push) y extracciones (pop) de la pila. Una introducción se lleva a cabo cuando el átomo seleccionado en la meta del tope de la pila se unifica satisfactoriamente con la cabeza de una cláusula de programa, el resolvente es introducido en la pila. Una extracción ocurre cuando no hay (más) cláusulas de programa con cuya cabeza se pueda unificar el átomo seleccionado en la meta del tope de la pila, esta meta es entonces extraída y se investiga la siguiente elección de la

cláusula cuya cabeza se unifique con el átomo seleccionado en el nuevo tope de la pila. Aunque las reglas de la búsqueda primero-a-profundidad tienen problemas innegables, se pueden implementar muy eficientemente. Esta técnica es enteramente consistente con el punto de vista de que PROLOG es principalmente un lenguaje de programación en lugar de un probador de teoremas.

Para un sistema cuya búsqueda es primero-a-profundidad, la regla de búsqueda se reduce a una *regla de ordenamiento*, esto es, una regla que especifica el orden en el que se probarán las cláusulas del programa. Los sistemas de PROLOG estándar utilizan el orden de las cláusulas en el programa como el orden fijo en el que son probadas. Esta implementación es muy fácil y eficiente, pero tiene la desventaja de que cada llamada a una definición prueba las cláusulas en la definición en exactamente el mismo orden.

Naturalmente, es preferible que la regla de búsqueda sea *equitativa (fair)*, esto es, tal que cada rama de éxito en el árbol-LSD será eventualmente encontrada. Para árboles-LSD infinitos, la regla de búsqueda que no tenga un componente primero-a-lo-ancho (breadth-first) no será equitativa en general. Sin embargo un componente primero-a-lo-ancho es menos compatible con una implementación eficiente.

De acuerdo a los resultados anteriores, si P U {G} no tiene modelo, no importa cuál sea la regla de cómputo, el árbol-LSD correspondiente siempre contiene una rama éxito,

pero un sistema de programación lógica con una regla de búsqueda primero-a-profundidad que use un orden fijo para probar las cláusulas de programa y una regla de cómputo arbitraria no siempre encuentra la rama éxito. En otras palabras, ninguno de los resultados de completez anteriores se aplica a la mayoría de los sistemas PROLOG actuales porque las consideraciones de eficiencia han forzado la implementación de reglas de búsqueda no equitativas.

Ejemplo 16. Sea P el siguiente programa

1. p(a,b) ←
2. p(c,b) ←
3. p(x,z) ← p(x,y), p(y,z)
4. p(x,y) ← p(y,x)

y G la meta ← p(a,c). P ∪ {G} tiene una refutación, y si se omite cualquier cláusula de P no la tendrá. Un sistema de programación lógica que use una búsqueda primero-a-profundidad con el orden para tratar las cláusulas del programa fijo, nunca encontrará una refutación debido a que las cláusulas 3 y 4 tienen cabezas generales y la que se coloque primero es la que siempre se utilizará, así el sistema no va a considerar nunca la otra.

La figura 3 a continuación ilustra el árbol-LSD correspondiente. Se utiliza la regla de cómputo estándar, que selecciona el átomo de más a la izquierda y prueba las cláusulas en el orden en que están en el programa. La rama de más a la izquierda del árbol-LSD es infinita y entonces una

búsqueda primero-a-profundidad nunca encontrará la rama éxito.

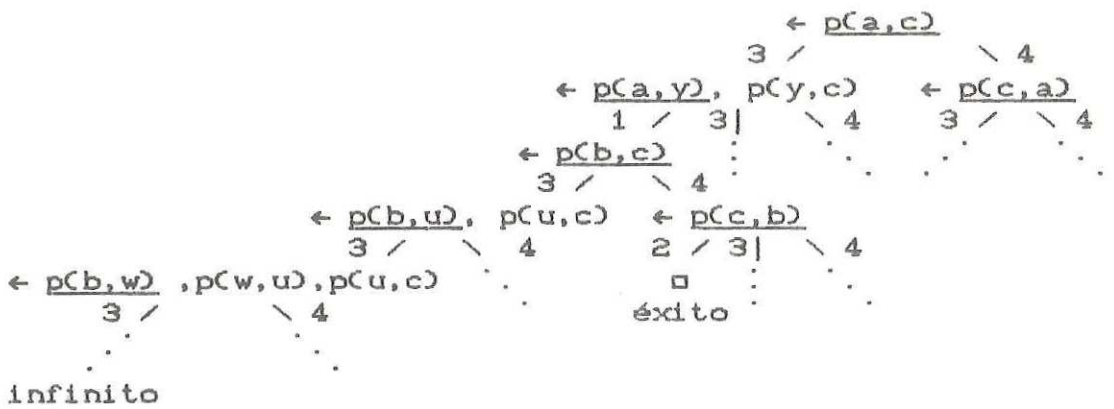


Figura 3. Arbol-LSD que ilustra el problema con búsqueda primero-a-profundidad.

II.2.6 CORTES.

En esta parte, discutiremos el corte, que es una facilidad de control controversial y muy usada, brindada por los sistemas PROLOG. Usualmente se escribe como "!" en programas, aunque algunos sistemas lo escriben como "/". Ha habido una discusión considerable sobre las ventajas y desventajas del corte y, en particular, cuando éste afecta la semántica de los programas en que aparece. El corte no afecta la semántica declarativa de los programas definidos, pero puede ocasionar una forma indeseable de incompletez en el procedimiento de refutación.

Primero, debemos ser precisos acerca de qué hace un corte. A través de esta discusión, restringimos nuestra

atención a sistemas que siempre seleccionan el átomo más a la izquierda en una meta. El corte es simplemente una anotación de programa no-lógica que comunica cierta información de control al sistema. Aunque se escribe como un átomo en el cuerpo de una cláusula, no es un átomo y no tiene significado lógico. La semántica declarativa de un programa con cortes es exactamente la semántica declarativa del programa sin los cortes, en otras palabras, los cortes no modifican de ninguna manera la lectura declarativa del programa.

La meta que causa que la cláusula conteniendo el corte sea activada se llama la meta *padre*, es decir, el átomo seleccionado en el padre se unifica con la cabeza de la cláusula cuyo cuerpo contiene al corte. Ahora, cuando se "selecciona" el corte simplemente "tiene éxito" inmediatamente, sin embargo, si después se regresa al corte, el sistema descontinúa la búsqueda en el subárbol cuya raíz es la meta padre. Por lo tanto, el corte ocasiona que el resto de este subárbol sea podado del árbol-LSD.

Ejemplo 17. Considérese el siguiente fragmento de un programa:

A ← B, C

⋮

B ← D, !, E

⋮

D ←

⋮

donde A, B, C, D y E son átomos. En la figura 4 se muestra parte del árbol-LSD de una llamada a este programa. El átomo seleccionado en la meta ← B, C origina la introducción del corte, el átomo D se selecciona y tiene éxito, entonces el corte tiene éxito, pero la submeta E fracasa y el sistema regresa al corte. El sistema descontinúa cualquier otra búsqueda en el subárbol que tiene la raíz ← B, C, y en su lugar, continúa la búsqueda con la siguiente elección para la meta ← A. Esto puede ser implementado si simplemente se extraen metas de la pila hasta que ← A esté en el tope.

Figura 4. El efecto del corte.

Así tenemos que un corte solamente poda al árbol-LSD. Si no hay respuesta en la parte podada (esto es, si no contiene una rama éxito), entonces tal uso del corte se llama

seguro. Sin embargo, si se poda una rama éxito por el corte, este uso se llama *inseguro*. Los usos seguros de corte son benéficos ya que mejoran la eficiencia sin perder respuestas. Los usos inseguros de corte son perjudiciales debido a que se pierden respuestas correctas.

El efecto dañino de los cortes es que ocasionan una forma de incompletez de respuestas correctas, en la implementación de la resolución-LSD. Con un uso inseguro del corte, un sistema puede contestar "no", cuando debe contestar "sí", dando una respuesta incorrecta.

II.3 FRACASO FINITO.

Los principales resultados de esta sección son varias caracterizaciones del conjunto con fracaso finito de un programa definido.

Sea P un programa definido, entonces F_P^d, el conjunto de átomos en B_P que *fracasan de una forma finita con profundidad d*, se define como sigue:

1. $A \in F_P^1$ si $A \notin T_P\downarrow 1$.
2. $A \in F_P^d$, para $d > 1$, si para cada cláusula $B \leftarrow B_1,\ldots,B_n$ en P y cada substitución θ tal que $A = B\theta$ y $B_1\theta,\ldots,B_n\theta$ no tienen variables, existe k tal que $1 \leq k \leq n$ y $B_k\theta \in F_P^{d-1}$.

Sea P un programa definido, entonces el *conjunto con fracaso finito* F_P de P se define como $F_P = \cup_{d\geq 1} F_P^d$.

Sea P un programa definido y G una meta definida, un árbol-LSD para P ∪ {G} tiene *fracaso finito* si es finito y no contiene ramas éxito.

Sea P un programa definido, el *conjunto con fracaso finito LSD* de P es el conjunto de todas las $A \in B_P$ para las cuales existe un árbol-LSD para P ∪ {← A} con fracaso finito.

Nótese que el fracaso finito LSD sólo garantiza la existencia de un árbol-LSD con fracaso finito, los otros pueden ser infinitos. A continuación veremos formas de selección de átomos que aseguran encontrar un árbol-LSD con fracaso finito, si existe.

Una derivación-LSD es *equitativa (fair)* si fracasa, o, para cada átomo B en la derivación, (alguna versión con más instancias de) B se selecciona en un número finito de pasos.

Nótese que hay derivaciones-LSD vía la regla de cómputo estándar que no son equitativas. Uno puede obtener equidad si, por ejemplo, se selecciona el átomo más a la izquierda a la derecha de los átomos (posiblemente ninguno) introducidos en el paso de derivación anterior, si hay tal átomo; si no, se selecciona el átomo más a la izquierda.

Un árbol-LSD es *equitativo* si cada rama del árbol es una derivación-LSD equitativa.

A continuación tenemos el resultado principal de esta sección.

Teorema 29. Sea P un programa definido y $A \in B_P$, entonces las siguientes afirmaciones son equivalentes:

1. $A \in F_P = \cup_{d \geq 1} F_P^d$
2. $A \notin T_P{\downarrow}\omega = \cup_{n<\omega} T_P{\downarrow}n$
3. A está en el conjunto de fracaso finito LSD.
4. Cada árbol-LSD para $P \cup \{\leftarrow A\}$ equitativo tiene fracaso finito.

Este teorema muestra que la resolución-LSD equitativa es una implementación válida y completa del fracaso finito.

II.4 PROGRAMAS NORMALES.

Esta sección presenta a los programas normales, que son programas tales que los cuerpos de sus cláusulas de programa, son conjunciones de literales.

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.

Un *programa normal* es un conjunto finito de cláusulas de programa.

Una *meta normal* es una cláusula de la forma

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

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

Cada programa definido es un programa normal pero no inversamente.

III. MANEJO DE LA NEGACION EN PROGRAMACION LOGICA.

La negación es un concepto difícil lógicamente. La respuesta *no*, no siempre significa lo mismo en diferentes contextos.

El uso de la negación en sistemas computarizados es más complicado por el hecho de que un tratamiento de la negación totalmente correcto lógicamente, causa una explosión combinatoria, así que tenemos que hacer "concesiones lógicas" debido a las limitaciones computacionales en la práctica. El reto es mejorar el funcionamiento sin perder el control lógico.

Como sólo la información positiva puede ser consecuencia lógica de un programa, se necesitan reglas especiales extralógicas para deducir la información negativa en Programación Lógica.

Dos casos especiales mencionados por Shepherdson[1988], donde la negación clásica puede ser implementada fácilmente son los siguientes:

1) Eliminación de la negación por renombre. Si no hay ocurrencias en el programa de $p(t_1,\ldots,t_n)$ para términos $t_1,\ldots,t_n$, entonces las literales negativas $\neg p(u_1,\ldots,u_n)$ pueden convertirse en positivas introduciendo un nuevo predicado $nop(x_1,\ldots,x_n)$ para $\neg p(x_1,\ldots,x_n)$. Este truco trabaja también cuando hay ocurrencias positivas de $p(t_1,\ldots,t_n)$ que no pueden unificarse con las negativas.

2) Programas definidos con preguntas normales. No hay problema en permitir preguntas con negación si el programa consiste sólo de cláusulas de Horn definidas. Como la resolución-LSD es completa para la negación clásica, no pueden tener éxito preguntas con una literal negativa porque no pueden ser consecuencia lógica del programa.

Se han dado varios significados de la negación en Programación Lógica, nosotros desarrollaremos los más conocidos que son: la Suposición del Mundo Cerrado, la Base de Datos Completada, la Circunscripción y la Suposición del Mundo Cerrado Generalizada. En el siguiente capítulo desarrollaremos la Negación por Fracaso y la Regla de Herbrand, explicando qué significado tienen, después expondremos la Negación por Inconsistencia que sí es compatible con la negación en Lógica Clásica.

III.1 SUPOSICION DEL MUNDO CERRADO.

Este concepto fue ideado por Reiter[1978].

La lógica de primer orden interpreta un programa literalmente, es decir, todos los conocimientos lógicos están representados explícitamente en el programa. Desafortunadamente, el número de hechos negativos acerca de un dominio dado en general excede al número de hechos positivos y así el requerimiento de que todos los hechos, positivos y negativos, se representen explícitamente puede ser imposible.

En el caso de programas que sólo tienen constantes hay una solución rápida para este problema: sólo representar explícitamente los hechos positivos e implícitamente los negativos, si su contraparte positiva no es consecuencia lógica del programa. Nótese sin embargo, que adoptando esta convención, estamos suponiendo que sabemos todo acerca de cada predicado del dominio; que no hay vacíos en nuestro conocimiento. La representación implícita de los hechos negativos supone conocimiento total acerca del dominio representado. Afortunadamente, en la mayoría de las aplicaciones, se garantiza esta suposición.

Nos referiremos a esto como la Suposición del Mundo Cerrado (SMC), su opuesto, la Suposición del Mundo Abierto, asume sólo la información dada en el programa y entonces requiere que todos los hechos, positivos y negativos, sean representados explícitamente, permitiendo vacíos en el conocimiento acerca del dominio.

La SMC puede ser axiomatizada aumentando al programa P el conjunto de axiomas ext(P) dado por:

ext(P) = {¬A|A es una literal positiva sin variables y P $\not\vdash$ A}.

Nos restringimos a los modelos de Herbrand y para axiomatizar esto aumentamos:

Los Axiomas de igualdad (AI):

$x = x$,

$x = y \rightarrow y = x$,

$x = y \wedge y = z \rightarrow x = z$,

$x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow [p(x_1,\ldots,x_n) \leftrightarrow p(y_1,\ldots,y_n)]$ para cada predicado p,

$x_1 = y_1 \wedge \ldots \wedge x_n = y_n \rightarrow [f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n)]$ para cada función f.

Los Axiomas de Libertad (AL):

$f(x_1,\ldots,x_n) \neq g(y_1,\ldots,y_m)$ para cada par f, g de funciones distintas,

$f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n) \rightarrow x_1 = y_1 \wedge \ldots \wedge x_n = y_n$ para cada función f,

$t(x) \neq x$ para cada término $t(x)$ diferente de x y en donde x ocurra.

El Axioma de Cerradura del Dominio (ACD):

$x = t_1 \vee x = t_2 \vee \ldots$ donde $t_1, t_2, \ldots$ son todos los términos sin variables.

Entonces tenemos que:

$$SMC(P) = P \cup ext(P) \cup AI \cup AL \cup ACD.$$

Nótese que si SMC(P) es consistente entonces determina un único modelo, porque AI ∪ AL ∪ ACD restringe el dominio a los modelos de Herbrand y P ∪ ext(P) determina los valores de verdad de todas las literales sin variables.

Consideremos un programa P para el cual SMC(P) es consistente. Sea A un átomo sin variables y supongamos que queremos inferir ¬A. Para usar SMC, tenemos que mostrar que A no es una consecuencia lógica de P. Desafortunadamente, debido a la indecidibilidad del problema de validez de la lógica de primer orden, no hay un algoritmo que tomando una A

arbitraria como entrada responda en un tiempo finito si A es o no una consecuencia lógica de P. Si A no es consecuencia lógica, puede quedarse en un loop. Así tenemos que la inexistencia de un procedimiento de prueba para SMC(P) se debe a que el conjunto de axiomas ext(P) no es recursivamente enumerable y por lo tanto, no es recursivo.

SMC es un ejemplo de una regla de inferencia no-monótona, es decir, en donde la suma de axiomas nuevos puede hacer decrecer el conjunto de teoremas que se tenían antes.

Ejemplo 18. Si $P = \{estudiante(Juan) \vee maestro(José)\}$, entonces SMC(P) es inconsistente, por lo tanto, todos los enunciados que contengan los predicados estudiante y maestro, y las constantes Juan y José, son teoremas de SMC(P). Si $P' = P \cup \{maestro(José)\}$, entonces $\neg maestro(José)$ no es teorema de SMC(P') , y $P \subseteq P'$ pero $SMC(P') \subsetneq SMC(P)$.

III.1.1 CONSISTENCIA DE LA SUPOSICION DEL MUNDO CERRADO.

Si P es consistente, no siempre es cierto que SMC(P) lo sea (véase el ejemplo anterior). A continuación presentamos algunos resultados para determinar la consistencia de SMC(P) con P dado.

Teorema 30. Si P consiste de cláusulas, Q es una negación de cláusula y el predicado de igualdad no aparece en P ni en Q, entonces:

$$SMC(P) \models Q \quad \text{si y solo si} \quad P \cup ext(P) \models Q.$$

Corolario 3. Si P consiste de cláusulas y la igualdad no

ocurre en él, entonces:

SMC(P) es consistente si y solo si P ∪ ext(P) es consistente.

Si hay conocimiento indefinido acerca de átomos sin variables, entonces SMC(P) es inconsistente, como se puede advertir en el siguiente resultado:

Teorema 31. Si P consiste de cláusulas y la igualdad no ocurre en él, entonces SMC(P) es consistente si y solo si para todos los átomos sin variables $A_1,\ldots,A_r$, $P \models A_1 \vee \ldots \vee A_r$ implica $P \models A_i$ para alguna $i = 1,\ldots,r$.

La información indefinida acerca de literales con variables no necesariamente implica la inconsistencia de la SMC.

Ejemplo 19. Sea P el siguiente programa:

$p(x) \leftarrow \neg q(x)$

$p(a)$

$q(b)$

entonces $p(a)$, $\neg p(b)$, $\neg q(a)$, $q(b)$ satisfacen la SMC y tenemos a $p(x) \vee q(x)$ como teorema sin que $p(x)$ ni $q(x)$ lo sean.

Teorema 32. SMC(P) es consistente si y solo si P tiene un modelo mínimo de Herbrand.

Los siguientes resultados son para cláusulas de Horn:

Teorema 33. Si P es un conjunto consistente de cláusulas de Horn entonces SMC(P) es consistente.

Como cada programa definido es consistente, tenemos:

Teorema 34. Si P es un programa definido, entonces SMC(P)

es consistente.

Las demostraciones de estos teoremas se pueden consultar en la publicación de Shepherdson[1988] y en la de Reiter[1978].

La SMC es muy natural para algunos programas correspondientes a bases de datos, pero se considera muy drástica para la mayoría de los programadores lógicos. Si se requiere que la SMC sea consistente para todas las extensiones con nuevos hechos, solamente se pueden tener programas relativos a bases de datos del tipo más simple. Debido a esto se han dado otros significados a la negación en Programación Lógica que expondremos a continuación.

III.2 BASE DE DATOS COMPLETADA.

La idea de este significado es asumir que las cláusulas que aparecen en un programa lógico P abarcan sólo la mitad *si* de un conjunto de definiciones *si y solo si* de los predicados del programa, la mitad *solo si* de cada definición es una ley de completación del predicado. La base de datos completada comp(P), dada implícitamente por las cláusulas de P, incluye este conjunto de definiciones junto con una teoría de igualdad que hace explícita la convención de que nombres diferentes denotan objetos diferentes.

La *definición* de un símbolo de predicado p en un programa normal P es el conjunto de todas las cláusulas del programa P

que tienen a p en su cabeza, es decir, el conjunto de las cláusulas que dicen algo sobre p.

A continuación daremos la definición de *comp*(P).

Sea $p(t_1,\ldots,t_n) \leftarrow L_1,\ldots,L_m$ una cláusula en un programa normal P. Requeriremos un nuevo símbolo de predicado =, que no aparezca en P, cuya interpretación será la relación de identidad. El primer paso es transformar 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 variables que no aparecen en la cláusula. Entonces, si $y_1,\ldots,y_d$ son las variables de la cláusula original, transformamos ésta en:

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

Ahora supongamos que esta transformación se hace para cada cláusula en la definición de p. Entonces obtenemos $k \geq 1$ fórmulas transformadas de la forma

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

donde cada E_i tiene la forma general

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

La *definición completada* de p es entonces la fórmula

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

Algunos símbolos de predicado en el programa pueden no estar en la cabeza de ninguna cláusula. Para cada uno de estos símbolos de predicado q, aumentamos explícitamente la cláusula $\forall x_1\ldots\forall x_n\neg q(x_1,\ldots,x_n)$. Esta es la definición de

q dada implícitamente por el programa. También llamamos a esta cláusula la *definición completada* de q.

Para crear las definiciones completadas intoducimos la igualdad, así que es esencial incluir algunos axiomas que la contengan. La teoría de la igualdad cuyos axiomas son AI y AL definidos antes, es suficiente para nuestros propósitos.

Sea P un programa normal. La *completación* de P, denotada por comp(P), es la colección de las definiciones completadas de los símbolos de predicado en P junto con la teoría de la igualdad.

Se discute con frecuencia que en el lenguaje cotidiano cuando uno escribe P uno realmente quiere decir comp(P), nótese que la manera de escribir P influye en la definición de comp(P).

Como Clark[1978] indicó, es apropiado considerar la completación de un programa normal, no el programa en sí, como el principal objeto de interés. Aunque el programador sólo le da el programa normal a un sistema de programación lógica, se entiende que el programa normal es completado por el sistema y el programador está programando con la completación. De acuerdo a esta noción, tenemos el concepto de una respuesta correcta.

Sea P un programa normal y G una meta normal, una *respuesta* para P ∪ {G} es una substitución de variables de G.

Sea P un programa normal, G una meta normal $\leftarrow L_1,\ldots,L_n$, y θ una respuesta para P ∪ {G}, decimos que θ es una *respuesta*

correcta para comp(P) ∪ {G} si $\forall((L_1 \wedge \ldots \wedge L_n)\theta)$ es una consecuencia lógica de comp(P).

Teorema 35. Si P es un programa normal entonces P es una consecuencia lógica de comp(P). En el sentido de que cada cláusula de P es consecuencia lógica de comp(P).

Ahora definiremos un mapeo T_P^J sobre la red de interpretaciones basadas en alguna pre-interpretación J, del lenguaje de P.

Sea J una pre-interpretación de un programa normal P e I una interpretación basada en J entonces $T_P^J(I) = \{A_{J,V} | A \leftarrow L_1 \wedge \ldots \wedge L_n \in P$, V es una asignación de variables con respecto a J, y $L_1 \wedge \ldots \wedge L_n$ es verdadero con respecto a I y V$\}$.

Cuando J es la pre-interpretación de Herbrand de P, escribimos T_P en lugar de T_P^J. Nótese que T_P^J no es monótona en general.

Ejemplo 20. Si P es el programa $p \leftarrow \neg p$, T_P no es monótona ya que $T_P(\emptyset) = \{p\}$ y $T_P(\{p\}) = \emptyset$, y $\emptyset \subseteq \{p\}$ pero $T_P(\emptyset) \not\subseteq T_P(\{p\})$.

Si P es un programa definido, entonces T_P^J es monótona.

Teorema 36. Sea P un programa normal, J una pre-interpretación de P e I una interpretación basada en J, entonces I es un modelo de P si y solo si $T_P^J(I) \subseteq I$.

Teorema 37. Sea P un programa normal, J una pre-interpretación de P e I una interpretación basada en J, supóngase que I junto con la relación identidad asignada a =

es un modelo de la teoría de la igualdad, entonces I junto con la relación identidad asignada a = es un modelo de comp(P) si y solo si $T_P^J(I) = I$.

Teorema 38. Sea P un programa definido y $A \in B_P$ entonces $A \in MPF(T_P)$ si y solo si comp(P) $\cup$ {A} tiene un modelo de Herbrand.

Teorema 39. Sea P un programa definido y $A_1,\ldots,A_m$ átomos, si $\forall(A_1 \wedge \ldots \wedge A_m)$ es una consecuencia lógica de comp(P) entonces también es una consecuencia lógica de P.

La información positiva que se puede deducir de comp(P) es la misma que la que se puede deducir de P, es decir, comp(P) no aumenta nueva información positiva. Para ser precisos, tenemos el siguiente resultado:

Teorema 40. Sea P un programa definido, G una meta definida, entonces θ es una respuesta correcta para comp(P) $\cup$ {G} si y solo si θ es una respuesta correcta para P $\cup$ {G}.

La demostraciones de estos teoremas se pueden encontrar en el libro de Lloyd[1987].

III.2.1 CONSISTENCIA DE comp(P).

Cada programa normal es consistente, pero la completación de un programa normal puede no serlo.

Ejemplo 21. Si tenemos P tal que P = {p $\leftarrow \neg$p}, P es consistente pero comp(P) = {p $\leftrightarrow \neg$p} no lo es.

Si el programa es definido tenemos que comp(P) es

consistente, pues los puntos fijos de T_P son modelos de comp(P), y para P definido siempre existe el mínimo punto fijo de T_P.

Una condición suficiente para asegurar que la completación de un programa normal sea consistente, es la que daremos a continuación, la idea es limitar el uso de la negación en reglas recursivas para mantener manejable a la teoría del modelo.

Una *función de nivel* de un programa normal es una función de su conjunto de símbolos de predicado a los enteros no-negativos. Nos referiremos al valor de un símbolo de predicado bajo esta función como el *nivel* de ese símbolo de predicado.

Un programa normal es *jerárquico* si tiene una función de nivel tal que, en cada cláusula de programa $p(t_1,\ldots,t_n) \leftarrow L_1,\ldots,L_m$, el nivel de cada símbolo de predicado que ocurre en el cuerpo es menor que el nivel de p.

Un programa normal es *estratificado* si existe una función de nivel tal que en cada cláusula del programa $p(t_1,\ldots,t_n) \leftarrow L_1,\ldots,L_m$, el nivel del símbolo de predicado de cada literal positiva sea menor o igual al de p y el nivel del símbolo de predicado de cada literal negativa en el cuerpo sea menor que el de p.

Teorema 41. Sea P un programa normal estratificado, entonces comp(P) tiene un modelo minimal normal de Herbrand. (Un modelo *normal* es uno para el que se asigna la relación

identidad a =, *minimal* significa que no hay un modelo normal de Herbrand estrictamente menor).

Este último teorema se debe a Apt, Blair y Walker[1988].

III.2.2 LA RELACION ENTRE LA SUPOSICION DEL MUNDO CERRADO Y LA BASE DE DATOS COMPLETADA.

La Suposición del Mundo Cerrado, SMC(P), y la Base de Datos Completada, comp(P), son similares superficialmente. Ambos son ejemplos de razonamiento por fracaso (default), asumiendo que si alguna información positiva no puede ser probada de un cierto modo a partir de P, entonces no es verdadera. Pero para SMC(P), la noción de prueba involucrada es aquella de la Lógica de Primer Orden, mientras que para comp(P), es usando una de las cláusulas del programa cuya cabeza se iguale con el átomo dado. A primera vista esto es una noción muy cercana de prueba, así que más átomos sin variables deberían ser falsos bajo comp(P) que bajo SMC(P), es decir, SMC(P) debe ser una consecuencia de comp(P). Pero esto no es tan simple porque comp(P) aumenta, en la mitad *solo si* de la definición completada de un predicado p, nuevos enunciados universales que pueden ser usados para probar cosas acerca de otros predicados además de p. También SMC(P) incluye el Axioma de Cerradura del Dominio (ACD) que restringe a los modelos de Herbrand, y esto no se incluye en comp(P), aunque esto se supone tácitamente a menudo. De hecho cuando SMC(P) y comp(P) son compatibles, SMC(P) implica comp(P)

porque SMC(P) tiene un solo modelo.

Para comprender la relación que existe entre SMC(P) y comp(P) se darán unos ejemplos y después unos resultados.

Ejemplo 22. Para el programa P: $p \leftarrow \neg p$,

SMC(P) es consistente pero comp(P) no.

Ejemplo 23. Para el programa P: $q \leftarrow \neg p$,

comp(P) es consistente pero SMC(P) no.

Ejemplo 24. Para el programa P: $p \leftarrow q$

$q \leftarrow \neg p$

$q \leftarrow q$,

comp(P) y SMC(P) son consistentes pero no compatibles, es decir, comp(P) ∪ SMC(P) es inconsistente.

Teorema 42. comp(P) ∪ SMC(P) es consistente si y solo si

1. comp(P) es consistente, y
2. para cualesquiera átomos sin variables $A_1,\ldots,A_r$, comp(P) ∪ ACD $\models (A_1 \vee \ldots \vee A_r)$ implica $P \models A_i$ para alguna $i = 1,\ldots,r$.

Teorema 43. Si P es un programa definido, entonces comp(P) ∪ SMC(P) es consistente.

También puede ser interesante considerar el efecto de aplicar la Suposición del Mundo Cerrado a la Base de Datos Completada, es decir, SMC(comp(P)); si consideramos el inverso comp(SMC(P)) no se gana nada ya que si SMC(P) es consistente tiene un solo modelo.

Teorema 44. SMC(comp(P)) ≤ SMC(P) ∪ comp(P).

Teorema 45. SMC(comp(P)) puede estar contenido

propiamente en SMC(P) ∪ comp(P), de hecho puede ser consistente mientras lo último no.

Teorema 46. SMC(comp(P)) es equivalente a SMC(P) ∪ comp(P) si y solo si para todos los átomos sin variables A,

$$comp(P) \models A \quad \text{implica} \quad P \models A,$$

es decir, cuando comp(P) no da nueva información acerca de átomos sin variables.

Teorema 47. Si SMC(P) ∪ comp(P) es consistente entonces SMC(comp(P)) es equivalente a él.

Corolario 4. Si P es un programa definido entonces SMC(P) ∪ comp(P) es lo mismo que SMC(comp(P)).

Todos estos resultados se pueden encontrar en la publicación de Shepherdson[1988].

III.3 CIRCUNSCRIPCION.

La Circunscripción fue introducida por McCarthy[1980]. Es una regla de suposición que puede ser usada junto con las reglas de inferencia de la lógica de primer orden y asume que los objetos satisfacen un predicado dado solo si tienen que hacerlo con base en una colección de hechos, circunscribiendo (limitando) así, el conjunto de dichos objetos. Los resultados de la circunscripción dependen del enunciado que se utilice para expresar los hechos.

El resultado de circunscribir un símbolo de predicado P

en un enunciado A es la fórmula que asegura que ningún símbolo de predicado cuya extensión incluya propiamente a la de P satisface A, es decir, minimiza a P en A.

La definición de circunscripción incluye un cuantificador de segundo orden, así es que se obtiene una fórmula de segundo orden. A continuación definiremos los conceptos que se requieren para entender la definición de circunscripción basándonos en la notación que utiliza Lifschitz[1985a].

III.3.1 FORMULAS DE SEGUNDO ORDEN.

Un *lenguaje de segundo orden* se define, como un lenguaje de primer orden, con constantes funcionales y constantes de predicado, cada una de alguna aridad n, $n \geq 0$, además de variables objeto se tienen variables funcionales n-arias y variables de predicado n-arias (las variables objeto y las constantes se identifican con variables y constantes funcionales de aridad 0). Las variables funcionales y de predicado pueden ser acotadas por cuantificadores. Un *enunciado* es una fórmula sin variables (funcionales o de predicado) libres.

Una *estructura* M para un lenguaje de segundo orden L consiste de un universo no vacío $|M|$, funciones de $|M|^n$ a $|M|$ representando a las constantes funcionales y subconjuntos de $|M|^n$ representando a las constantes de predicado. Para cualquier constante K, el objeto en M (función o conjunto) que la representa se denota por M[K]. La igualdad se interpreta

como identidad, las variables funcionales fluctúan sobre funciones arbitrarias de $|M|^n$ a $|M|$ y las variables de predicado sobre subconjuntos arbitrarios de $|M|^n$.

Un *modelo* de un enunciado A es cualquier estructura M tal que A es verdadero en M. A *implica* B si cada modelo de A es un modelo de B. A es *equivalente* a B si tienen los mismos modelos.

Un predicado n-ario es una expresión de la forma $\lambda x A(x)$, donde x es una n-upla de variables objeto y A(x) una fórmula. De forma usual, si U es $\lambda x A(x)$ y t es una n-upla de términos, entonces Ut se interpreta como A(t). Identificamos una constante de predicado P con el predicado $\lambda x P(x)$.

Si U, V son predicados n-arios entonces $U \leq V$ representa $\forall x(U(x) \rightarrow V(x))$, así $U \leq V$ expresa que la extensión de U es un subconjunto de la extensión de V (la extensión de un predicado P, es el conjunto de los elementos x tales que P(x)). Aplicamos esta notación a tuplas $U = U_1,\ldots,U_m$ y $V = V_1,\ldots,V_m$ de predicados, asumiendo que son similares (es decir, que U_i y V_i tienen la misma aridad): $U \leq V$ significa $U_1 \leq V_1 \wedge \ldots \wedge U_m \leq V_m$. Además $U = V$ representa $U \leq V \wedge V \leq U$ y $U < V$ es lo mismo que $U \leq V \wedge \neg(V \leq U)$, si $m = 1$ entonces $U < V$ significa simplemente que la extensión de U es un subconjunto propio de la extensión de V.

III.3.2 CIRCUNSCRIPCION PARALELA.

Primero consideraremos la circunscripción *paralela* cuando

no se especifican prioridades entre los predicados minimizados. Sea P una tupla de constantes de predicado, Z una tupla de constantes funcionales y/o de predicado diferentes de las de P y sea A(P,Z) un enunciado (si tenemos un conjunto de enunciados, tomamos su conjunción como A(P,Z)). La *circunscripción de P en A(P,Z) con variable Z* es el enunciado

$$A(P,Z) \wedge \neg\exists p\exists z(A(p,z) \wedge p < P)$$

donde p, z son tuplas de variables similares a P, Z y A(p,z) es el resultado de reemplazar todas las ocurrencias de P y Z en A por p y z. Denotamos a esta circunscripción por circ(A(P,Z); P; Z), si Z = Ø escribimos circ(A(P); P) y si P incluye a todas las constantes de predicado escribimos circ(A(P)).

Esta fórmula establece que los predicados de P tienen una extensión minimal bajo la suposición de que A(P,Z) se da y las extensiones de los predicados de Z pueden variar en el proceso de minimización.

II.3.3 SIGNIFICADO DE LA CIRCUNSCRIPCION EN LA TEORIA DE MODELOS.

Para dos estructuras M_1, M_2 escribimos $M_1 \leq^{P;Z} M_2$ si:

1. $|M_1| = |M_2|$
2. $M_1[K] = M_2[K]$ para toda K que no esté en P ni en Z
3. $M_1[P_i] \subseteq M_2[P_i]$ para toda $P_i \in P$.

Así, $M_1 \leq^{P;Z} M_2$ si M_1 y M_2 difieren sólo en cómo

interpretan las constantes en P y Z, y la extensión de cada P_i en M_1 es un subconjunto de su extensión en M_2.

M es $<^{P;Z}$-*minimal* en una clase S de estructuras si $M \in S$ y no hay estructura $M' \in S$ tal que $M' <^{P;Z} M$ ($M_1 <^{P;Z} M_2$ si $M_1 \leq^{P;Z} M_2$ pero no es cierto que $M_2 \leq^{P;Z} M_1$).

Los modelos de circunscripción pueden ser caracterizados como sigue:

Teorema 48. Una estructura M es un modelo de circ(A;P;Z) si y solo si M es minimal en la clase de los modelos de A con respecto a $<^{P;Z}$.

Esta equivalencia se sigue del hecho de que especificar valores de p, z para los que $A(p,z) \wedge p < P$ sea verdadero en M, es equivalente a especificar un modelo M' de A tal que $M' <^{P;Z} M$. Este es un resultado de correctez y completez para la circunscripción.

Ejemplo 25. A = P(c), c una constante objeto. La circunscripción de P en A afirma que la extensión de P es un conjunto minimal que satisface esta condición, en otras palabras, c es su único elemento:

$$circ(P(c);P) \leftrightarrow \forall x(P(x) \leftrightarrow x = c).$$

Ejemplo 26. A = $\neg$P(a), esto no nos da ninguna información "positiva" acerca de P y es verdadero cuando P es falso:

$$circ(\neg P(a);P) \leftrightarrow \forall x(\neg P(x)).$$

Ejemplo 27. A = (P(a) $\wedge$ P(b)), la circunscripción asegura que a y b son los únicos elementos de P:

$$circ(P(a) \wedge P(b);\ P) \leftrightarrow \forall x(P(x) \leftrightarrow x = a \vee x = b).$$

Ejemplo 28. A = (P(a) ∨ P(b)), en este caso hay dos valores minimales:

circ(P(a) ∨ P(b); P) ↔ ∀x(P(x) ↔ x = a) ∨ ∀x(P(x) ↔ x = b).

Ejemplo 29. A = (P(a) ∨ Q(b)), en este caso al circunscribir P y Q en A tenemos dos valores minimales:

$$circ(P(a) \vee Q(b); P, Q) \leftrightarrow (\forall x((P(x) \leftrightarrow x = a) \wedge \neg Q(x)) \vee (\forall x(\neg P(x) \wedge (Q(x) \leftrightarrow x = b))$$

III.3.4 CIRCUNSCRIPCION CON PRIORIDAD.

La circunscripción con prioridad que Lifschitz[1985a] llama circunscripción general, es definida por Gelfond, Przymusinska y Przymusinski[1989] como sigue:

La *circunscripción con prioridad* de A con prioridades $P_1 > \ldots > P_k$ y variables Z es denotada por $circ(A; P_1 > \ldots > P_k; Z)$ y se define como sigue:

$$circ(A; P_1 > \ldots > P_k; Z) = circ(A; P_1; P_2 + \ldots + P_k + Z) \wedge circ(A; P_2; P_3 + \ldots + P_k + Z) \wedge \ldots \wedge circ(A; P_k; Z),$$

donde A+B es el resultado de concatenar la tupla A con la B.

Ejemplo 30.

circ(∀x(p(x) ∨ q(x)); p, q) ↔ ∀x(p(x) ↔ ¬q(x)),

pues aquí se minimizan p y q sin prioridades, por lo tanto se tiene que si se da p(x) no se da q(x) y viceversa.

circ(∀x(p(x) ∨ q(x); p > q) ↔ ∀x(¬p(x) ∧ q(x)),

aquí hacemos la extensión de p tan pequeña como sea posible, aún si hace que la extensión de q crezca,

entonces p se hace igual a falso y q a verdadero.

Teorema 49. Una estructura M es un modelo de la circunscripción con prioridad circ(A;P_1>...>P_k;Z) si y solo si M es un modelo de A tal que para toda i ≤ k, M es $<^{P;Z'}$-minimal con P = P_i y Z' = P_{i+1}+...+P_k+Z.

En las aplicaciones es razonable asignar prioridades mayores a los predicados que representan excepciones de otros predicados.

III.3.5 RELACION CON SMC Y CONSISTENCIA.

Claramente hay una similitud entre la SMC y circunscribir todos los símbolos de predicado en una fórmula de predicado: en ambos casos "minimizamos" los predicados representados por los símbolos. Al mismo tiempo, estos dos procedimientos son técnicamente muy diferentes, la SMC extiende la base de datos con fórmulas muy simples: átomos negados sin variables, pero usa el concepto de consecuencia lógica para decidir qué fórmulas se pueden aumentar, por otro lado, la circunscripción es una traslación puramente sintáctica, pero su resultado se expresa por medio de una fórmula de lógica de segundo orden.

Los siguientes resultados se deben a Lifschitz[1985b], él considera un lenguaje de primer orden sin igualdad ni símbolos funcionales, con un número finito de constantes y símbolos de predicado.

Ejemplo 31. A = (p(a) ∧ q(b))

SMC(A) = (p(a) ∧ q(b), ¬p(b), ¬q(a)) ∪ AI ∪ AL ∪ ACD

$$circ(A) \leftrightarrow \forall x(p(x) \leftrightarrow x = a) \wedge \forall y(q(y) \leftrightarrow y = b)$$

Para derivar circ(A) de SMC(A) utilizamos que los objetos a y b son los únicos en el universo: $\forall x(x=a \vee x=b)$. En el caso general, se necesita el Axioma de Cerradura del Dominio (ACD).

Para derivar SMC(A) de circ(A) tenemos que asumir que los objetos denotados por a y b son diferentes. En el caso general necesitamos los Axiomas de Libertad (AL).

Estas dos suposiciones caracterizan la importante clase de las bases de datos cerradas E-saturadas estudiadas por Reiter. ACD es más fuerte: implica que los cuantificadores se pueden reemplazar por conjunciones y disyunciones finitas, así que cualquier fórmula es equivalente a una combinación proposicional de átomos.

Existe A para la que la equivalencia $SMC(A) \leftrightarrow circ(A)$ no puede ser probada aún asumiendo ACD y AL; esto es claro por el hecho de que SMC a veces es contradictoria, mientras que el resultado de circunscribir una conjunción de cláusulas no nos lleva a contradicción, de acuerdo con Etherington, Mercer y Reiter.

Ejemplo 32. $A = (p(a) \vee p(b))$

$$SMC(P) = \{p(a) \vee p(b), \neg p(a), \neg p(b)\} \cup AI \cup AL \cup ACD$$

$$circ(P) \leftrightarrow ((\forall x(p(x) \leftrightarrow x = a) \vee \forall x(p(x) \leftrightarrow x = b))$$

Teorema 50. Si SMC(A) es consistente entonces

$$\models SMC(A) \leftrightarrow circ(A) \wedge AL \wedge ACD$$

El siguiente resultado se debe a Gelfond, Przymusinska y

Przymusinski[1989]:

Teorema 51. Si P es un programa definido entonces

$$circ(P) \models F \text{ si y solo si } SMC(P) \models F.$$

La SMC no es equivalente a la circunscripción, hay varias razones para ver a la circunscripción como una forma substancialmente más poderosa de razonamiento no-monótono:

1. La caracterización de los predicados minimizados dados por la circunscripción se expresa por medio de fórmulas universales, no sólo por medio de negaciones de átomos sin variables. Esto es esencial cuando ACD no es aplicable.

2. La circunscripción lleva a menudo a condiciones no-triviales sobre los predicados minimizados cuando la SMC es contradictoria.

3. La circunscripción nos permite minimizar un subconjunto de predicados, no necesariamente todos los representados en el lenguaje.

4. Si los objetivos de minimizar varios predicados entran en conflicto entre sí, la circunscripción da una oportunidad de asignar prioridades relativas a esos objetivos, utilizando circunscripción con prioridad.

5. En la circunscripción no perjudica tomar hechos irrelevantes en cuenta, pues si estos hechos no contienen el símbolo de predicado que se está circunscribiendo, aparecerán como conjunciones sin cambio y entonces las versiones originales de estos hechos pueden utilizarse.

6. Cuando se tiene un predicado P del cual se quiere

poder deducir nuevos hechos positivos al circunscribir A, que los que se deducen de A, basta con considerar a P dentro de Z, lo que no es posible con SMC.

Besnard, Moinard y Mercer[1989] demuestran que no siempre que se aplica la circunscripción a un enunciado A consistente, se obtiene un enunciado consistente, y dan restricciones para que con todo A consistente se obtenga la circunscripción de A consistente, aunque el resultado no es tan fuerte. Además dicen que en general, la consistencia y la completez para la circunscripción son incompatibles ya que la completez requiere que cualquier teoría consistente que no tenga un modelo minimal, tenga una circunscripción inconsistente.

III.4 SUPOSICION DEL MUNDO CERRADO GENERALIZADA.

Como vimos antes, la SMC(P) es inconsistente para cualquier programa P que implique información indefinida acerca de átomos sin variables, y es consistente solo si P tiene un modelo mínimo de Herbrand. Para encontrar una suposición más débil que sea útil y que sea consistente para todos los programas consistentes P, Minker sugirió reemplazar modelo mínimo por modelo minimal y formuló la Suposición del Mundo Cerrado Generalizada (SMCG).

Asumimos que P es un conjunto de cláusulas. La definición de SMCG(P) dice que un átomo sin variables A es falso en SMCG(P) si y solo si es falso en cada modelo minimal

de Herbrand de P, es decir, es análogo a SMC(P) solo que en lugar de tener ext(P) tenemos {¬A|A es falso en todos los modelos minimales de Herbrand de P}.

Consideramos un enunciado verdadero si es verdadero en todos los modelos minimales de P, falso si es falso en todos los modelos minimales de P, indefinido si es verdadero en un modelo minimal y falso en otro. Un evaluador de preguntas completo y correcto ideal, sería uno que haga tener éxito a una pregunta si y solo si es verdadera en todos los modelos minimales, fracasar si y solo si es falsa en todos los modelos minimales y no dar respuesta (es decir, un cómputo infinito) cuando la pregunta sea indefinida, es decir, verdadera en alguno pero no en todos los modelos minimales.

Los siguientes resultados se pueden consultar en Sheperdson[1988].

Teorema 52. Si P es un conjunto consistente de cláusulas entonces P tiene un modelo minimal de Herbrand.

Caracterización sintáctica de Minker de la SMCG:

Teorema 53. Un átomo sin variables A está en un modelo minimal de Herbrand de P si y solo si hay átomos sin variables $A_1,\ldots,A_r$, $r \geq 0$, tales que $P \models (A \vee A_1 \vee \ldots \vee A_r)$ pero $P \not\models (A_1 \vee \ldots \vee A_r)$.

Teorema 54.

1. Un modelo minimal de P es un modelo minimal de SMCG(P) e inversamente.
2. Si P es consistente entonces SMCG(P) es

consistente.

3. Si la cuantificación existencial de una matriz positiva es verdadera en todos los modelos minimales de P entonces es una consecuencia de P.
4. Si la cuantificación existencial de una matriz positiva es una consecuencia de SMCG(P) entonces es consecuencia de P.
5. SMCG(P) es una consecuencia de SMC(P) y si SMC(P) es consistente entonces coincide con SMCG(P).
6. Si Q es una consecuencia de SMCG(P) entonces es verdadera en todos los modelos minimales de P.

En 3 y 4, una matriz positiva es una fórmula construída a partir de átomos por el uso de $\wedge$, $\vee$ solamente, es decir, sin usar $\neg$.

El resultado 3 significa que para tratar de caracterizar los modelos minimales de P no importa aumentar a SMCG(P) los átomos sin variables que son verdaderos en todos los modelos minimales pues ya son consecuencia de P. El enunciado 4 dice que SMCG(P) no aumenta nueva información positiva.

Aunque por 1 todos los modelos minimales de P son modelos de SMCG(P), el inverso no es verdadero pues hay modelos de SMCG(P) que no son modelos minimales de P.

Ejemplo 33. Si P es $p \vee q$ entonces SMCG(P) no contiene átomos negados sin variables y {p, q} es un modelo de él.

El ejemplo anterior muestra que el inverso de 6 no es

cierto pues $(p \wedge \neg q) \vee (q \wedge \neg p)$ es verdadero en todos los modelos minimales de P, pero no es una consecuencia de SMCG(P). Por lo tanto, aún sin contar con la incompletez debida al procedimiento de cómputo utilizado para encontrar consecuencias de SMCG(P), SMCG(P) es una tentativa incompleta para caracterizar a los modelos minimales de P.

No hay una relación simple entre comp(P) y SMCG(P).

Ejemplo 34. Si $P = \{q \leftarrow \neg p\}$, el modelo $\{p, q\}$ de SMCG(P) no es un modelo de comp(P) = $\{q \leftrightarrow \neg p, \neg p\}$. Si $P = \{p \leftarrow p\}$, el modelo $\{p\}$ de comp(P) = $\{p \leftrightarrow p\}$, no es modelo de SMCG(P) = $\{\neg p\}$.

La relación de la SMCG con la circunscripción se expresa en el siguiente resultado que se puede consultar en la publicación de Gelfond, Przymusinska y Przymusinski[1989].

Teorema 55. Para todo P programa,

$$circ(P) \models F \text{ si y solo si } SMCG(P) \models F.$$

IV. NEGACION POR FRACASO.

Antes vimos lo que es una respuesta correcta para comp(P) ∪ {G} (ver II.2), ya que tenemos el concepto declarativo, veremos cómo implementarlo. La idea básica es usar resolución-LSD, aumentada con la Negación por Fracaso (resolución-LSDNF), en donde para mostrar que A es falsa hacemos una búsqueda exhaustiva de una prueba para A y si cada prueba posible fracasa se infiere ¬A. Si podemos mostrar que cada intento para probar A fracasa usando sólo el programa P, es en efecto una prueba de ¬A usando comp(P), entonces la Negación por Fracaso es justamente una regla de inferencia para deducciones de comp(P).

Nuestra primera tarea es dar una definición precisa de una refutación-LSDNF y un árbol-LSDNF con fracaso finito. Para esto, primero daremos las definiciones mutuamente recursivas de los conceptos de refutación-LSDNF de rango k y de árbol-LSDNF con fracaso finito de rango k.

En las definiciones a continuación, será necesario seleccionar literales de metas normales. Para seleccionar literales se requiere que si éstas son negativas, no tengan variables, si son positivas no hay restricción. Esta condición se llama la *condición de seguridad* en la selección de literales, y se usa para asegurar la correctez de la resolución-LSDNF. Si una regla de selección cumple la condición de seguridad, se llamará una *regla segura*.

Sea G la meta $\leftarrow L_1,\ldots,L_m,\ldots,L_p$ y sea C la cláusula $A \leftarrow M_1,\ldots,M_q$ entonces G' es *resolvente* de G y C usando θ si se dan las siguientes condiciones:

(a) L_m es un átomo en G, llamado el átomo *seleccionado*,

(b) θ es un umg de L_m y A, y

(c) G' es la meta normal

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

Sea P un programa normal y G una meta normal, una *refutación-LSDNF de rango 0* para $P \cup \{G\}$ consiste de una sucesión $G_0 = G, G_1,\ldots, G_n = \square$ de metas normales, una sucesión $C_1,\ldots,C_n$ de variantes de cláusulas de programa de P y una sucesión $\theta_1,\ldots,\theta_n$ de umg's tales que cada G_{i+1} es resolvente de G_i y C_{i+1} usando θ_{i+1}.

Sea P un programa normal y G una meta normal, un *árbol-LSDNF con fracaso finito de rango 0* para $P \cup \{G\}$ es un árbol que satisface lo siguiente:

(a) El árbol es finito y cada nodo del árbol es una meta normal no vacía.

(b) El nodo raíz es G.

(c) Sólo se seleccionan literales positivas en los nodos en el árbol.

(d) Sea $\leftarrow L_1,\ldots,L_m,\ldots L_p$ un nodo interno en el árbol y supóngase que L_m es un átomo y se selecciona, entonces, para cada (variante de) cláusula de programa $A \leftarrow M_1,\ldots,M_q$ tal que L_m y A son unificables con umg θ, este nodo tiene un hijo

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

(e) Sea $\leftarrow L_1,\ldots,L_m,\ldots,L_p$ un nodo hoja en el árbol y supóngase que L_m es un átomo y se selecciona, entonces no hay (variante de) cláusula de programa en P cuya cabeza unifique con L_m.

Sea P un programa normal y G una meta normal, una *refutación-LSDNF de rango k+1* para $P \cup \{G\}$ consiste de una sucesión $G_0 = G, G_1,\ldots, G_n = \square$ de metas normales, una sucesión $C_1,\ldots,C_n$ de variantes de cláusulas de programa de P o literales negativas sin variables, y una sucesión $\theta_1,\ldots,\theta_n$ de substituciones, tales que, para cada i:

(i) G_{i+1} es resolvente de G_i y C_{i+1} usando θ_{i+1}, o

(ii) G_i es $\leftarrow L_1,..,L_m,\ldots,L_p$, la literal seleccionada L_m en G_i es una literal negativa sin variables $\neg A_m$ y hay un árbol-LSDNF con fracaso finito 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 substitución identidad y C_{i+1} es $\neg A_m$.

Sea P un programa normal y G una meta normal, un *árbol-LSDNF con fracaso finito de rango k+1* para $P \cup \{G\}$ es un árbol que satisface lo siguiente:

(a) El árbol es finito y cada nodo del árbol es una meta normal no vacía.

(b) El nodo raíz es G.

(c) Sea $\leftarrow L_1,\ldots,L_m,\ldots,L_p$ un nodo interno en el árbol y supóngase que L_m se selecciona, entonces:

(i) L_m es un átomo y, para cada (variante de) cláusula de programa $A \leftarrow M_1,\ldots,M_q$ tal que L_m y A son unificables con 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

(ii) L_m es una literal negativa sin variables $\neg A_m$ y hay un árbol-LSDNF con fracaso finito de rango k para $P \cup \{\neg A_m\}$, en cuyo caso el único hijo es $\leftarrow L_1,\ldots,L_{m-1},L_{m+1},\ldots,L_p$.

(d) Sea $\leftarrow L_1,\ldots,L_m,\ldots,L_p$ un nodo hoja en el árbol y supóngase que L_m es seleccionada, entonces:

(i) L_m es un átomo y no hay (variante de) cláusula de programa en P cuya cabeza unifique con L_m, o

(ii) L_m es una literal negativa sin variables $\neg A_m$ y hay una refutación-LSDNF de rango k para $P \cup \{\neg A_m\}$.

Nótese que un(a) árbol-LSDNF con fracaso finito (refutación-LSDNF) de rango k es también un(a) árbol-LSDNF con fracaso finito (refutación-LSDNF) de rango n, para toda $n \geq k$.

Sea P un programa normal y G una meta normal, una *refutación-LSDNF* para $P \cup \{G\}$ es una refutación-LSDNF de rango k para $P \cup \{G\}$, para alguna k.

Sea P un programa normal y G una meta normal, una *respuesta computada* θ para $P \cup \{G\}$ es la substitución obtenida al restringir la composición $\theta_1 \ldots \theta_n$ a las variables de G, donde $\theta_1,\ldots,\theta_n$ es la sucesión de substituciones usada en una refutación-LSDNF para $P \cup \{G\}$.

Como sólo se seleccionan literales negativas sin variables entonces $L_i\theta$ no debe tener variables, para cada literal negativa L_i en G.

Ahora que hemos dado la definición de una respuesta computada, consideraremos el procedimiento que un sistema de programación lógica podría usar para calcular respuestas. La idea básica es usar resolución-LSD, aumentada con la regla de negación por fracaso. Cuando se selecciona una literal positiva, usamos esencialmente resolución-LSD para derivar una nueva meta. Sin embargo, cuando se selecciona una literal negativa sin variables, se entra al proceso que responde metas recursivamente para tratar de establecer la submeta negativa. Podemos ver a estas submetas negativas como lemas separados, que se deben establecer para calcular el resultado. Habiendo seleccionado una literal negativa sin variables ¬A en alguna meta, una tentativa es contruir un árbol-LSDNF con fracaso finito con raíz ← A antes de continuar con el cálculo restante. Si se construye este árbol-LSDNF con fracaso finito, entonces la submeta ¬A tiene éxito. Por otro lado, si se encuentra una refutación-LSDNF para ← A, entonces la submeta ¬A fracasa. Nótese que las substituciones solamente se llevan a cabo con llamadas con éxito de literales positivas, las llamadas negativas nunca las crean ya que sólo tienen éxito o fracasan. Así la negación por fracaso es solamente una prueba.

A continuación daremos las definiciones de

derivación-LSDNF y árbol-LSDNF.

Sea P un programa normal y G una meta normal, una *derivación-LSDNF* para P ∪ {G} consiste de una sucesión (finita o infinita) $G_0 = G, G_1, \ldots$ de metas normales, una sucesión $C_1, C_2, \ldots$ de variantes de cláusulas de programa (llamadas *cláusulas de entrada*) de P o literales negativas sin variables, y una sucesión $\theta_1, \theta_2, \ldots$ de substituciones que satisfacen lo siguiente:

(a) Para cada i:

(i) G_{i+1} es resolvente de G_i y una cláusula de entrada C_{i+1} usando θ_{i+1}, o

(ii) G_i es $\leftarrow L_1, \ldots, L_m, \ldots, L_p$, la literal seleccionada L_m en G_i es una literal negativa sin variables $\neg A_m$ y hay un árbol-LSDNF con fracaso finito para P ∪ $\{\leftarrow A_m\}$, 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 substitución identidad y C_{i+1} es $\neg A_m$.

(b) Si la sucesión $G_0, G_1, \ldots$ de metas es finita, entonces:

(i) la última meta es vacía, o

(ii) la última meta es $\leftarrow L_1, \ldots, L_m, \ldots, L_p$; L_m es un átomo, L_m es seleccionada y no hay (variantes de) cláusulas de programa en P cuya cabeza unifique con L_m, o

(iii) la última meta es $\leftarrow L_1, \ldots, L_m, \ldots, L_p$; L_m es una literal negativa sin variables $\neg A_m$, L_m es

seleccionada y hay una refutación-LSDNF para $P \cup \{\leftarrow A_m\}$.

Sea P un programa normal y G una meta normal, un *árbol-LSDNF* para $P \cup \{G\}$ es un árbol que satisface lo siguiente:

(a) Cada nodo del árbol es una meta normal (posiblemente vacía).

(b) El nodo raíz es G.

(c) Sea $\leftarrow L_1,\ldots,L_m,\ldots,L_p$ $(p \geq 1)$ un nodo interno en el árbol y supóngase que L_m es seleccionada, entonces:

(i) L_m es un átomo y, para cada (variante de) cláusula de programa $A \leftarrow M_1,\ldots,M_q$ tal que L_m y A son unificables con 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

(ii) L_m es una literal negativa sin variables $\neg A_m$ y hay un árbol-LSDNF con fracaso finito para $P \cup \{\leftarrow A_m\}$, en cuyo caso el único hijo es $\leftarrow L_1,\ldots,L_{m-1},L_{m+1},\ldots,L_p$.

(d) Sea $\leftarrow L_1,\ldots,L_m,\ldots,L_p$ $(p \geq 1)$ un nodo hoja en el árbol y supóngase que L_m es seleccionada, entonces:

(i) L_m es un átomo y no hay (variante de) cláusula de programa en P cuya cabeza unifique con L_m, o

(ii) L_m es una literal negativa sin variables $\neg A_m$ y hay una refutación-LSDNF para $P \cup \{\leftarrow A_m\}$.

(e) Los nodos que son la cláusula vacía no tienen hijos.

Los conceptos de derivación-LSDNF, refutación-LSDNF y

árbol-LSDNF generalizan los de derivación-LSD, refutación-LSD y árbol-LSD. Una derivación-LSDNF es *finita* si consiste de una sucesión finita de metas, de otro modo, es *infinita*. Una derivación-LSDNF *tiene éxito* si es finita y la última meta es la meta vacía. Una derivación-LSDNF *fracasa* si es finita y la última meta no es la meta vacía. Similarmente definimos ramas infinitas, éxito y fracaso de un árbol-LSDNF. Es claro que una derivación-LSDNF con éxito es una refutación-LSDNF y un árbol-LSDNF cuyas ramas sean ramas fracaso, es un árbol-LSDNF con fracaso finito.

Si una meta contiene sólo literales negativas con variables, entonces, debido a la condición de seguridad, no se puede seleccionar ninguna literal. Formalicemos esta noción. Un *cómputo* para P ∪ {G} es un intento de construir una derivación-LSDNF para P ∪ {G}.

Sea P un programa normal y G una meta normal, decimos que un cómputo para P ∪ {G} *tropieza* si en algún punto del cómputo se llega a una meta que contiene sólo literales negativas con variables.

Ejemplo 35. Si G es ← ¬p(x) y P es cualquier programa normal, entonces el cómputo para P ∪ {G} tropieza inmediatamente.

Hay otras formas en las que puede terminar un cómputo con la resolución-LSDNF además de las ya conocidas con la resolución-LSD que son éxito, fracaso o irse a infinito. Estas formas que dan resultados inconclusos son la que

llamamos antes tropezar y la que se llama *fin muerto* que sucede cuando la literal que se eligió es una literal negativa L sin variables que ni tiene éxito, ni fracasa, en cuyo caso la evaluación se congela porque no se recibe respuesta acerca de la evaluación de L.

Una condición muy fuerte para que no haya fin muerto es que cada literal sin variables tenga éxito o fracase.

Ahora daremos una condición bajo la que podemos estar seguros que la resolución-LSDNF nunca tropieza.

Sea P un programa normal y G una meta normal. Decimos que una cláusula de programa $A \leftarrow L_1,\ldots,L_n$ en P es *admitida* si cada variable que ocurre en la cláusula ocurre en la cabeza A o en una literal positiva del cuerpo $L_1,\ldots,L_n$. Decimos que una cláusula de programa es *permitida* si cada variable que ocurre en la cláusula ocurre en una literal positiva del cuerpo $L_1,\ldots,L_n$. Sea G, $\leftarrow L_1,\ldots,L_n$, decimos que G es *permitida* si cada variable que ocurre en G ocurre en una literal positiva del cuerpo $L_1,\ldots,L_n$. Decimos que P $\cup$ {G} es *permitido* si se tienen las siguientes condiciones:

(a) Cada cláusula en P es admitida.

(b) Cada cláusula en la definición de un símbolo de predicado que ocurre en una literal positiva en el cuerpo de G o en una literal positiva en el cuerpo de una cláusula en P es permitida.

(c) G es permitida.

Una meta permitida puede no tener éxito aunque alguna

instancia sin variables sí lo tenga.

Ejemplo 36. Sea P:

$$p(x) \leftarrow \neg r(x)$$

entonces la meta $\leftarrow p(a)$ tiene éxito pero $\leftarrow p(x)$ tropieza.

Una meta permitida puede fracasar aunque alguna instancia sin variables no.

Ejemplo 37. Sea P:

$$q(x) \leftarrow r(x)$$

$$p(a) \leftarrow p(a)$$

Con la regla de selección que toma la literal permitida de más a la izquierda, $\leftarrow \neg p(x), r(x)$ fracasa mientras que $\leftarrow \neg p(a), r(a)$ se queda en un loop.

Una meta permitida $\leftarrow L_1, L_2$ puede tener éxito aunque $\leftarrow L_1$ no tenga éxito bajo ninguna regla de selección:

Ejemplo 38. Sea P:

$$p(x) \leftarrow \neg s(x)$$

$$r(a)$$

$\leftarrow p(x), r(x)$ tiene éxito pues pasa a $\leftarrow \neg s(x), r(x)$ y luego a $\leftarrow \neg s(a)$ y a $\square$, pero $\leftarrow p(x)$ se pasa a $\leftarrow \neg s(x)$ que tropieza.

Nótese que una cláusula unitaria permitida no debe tener variables y cada cláusula permitida es admitida. Estas definiciones generalizan la definición de Clark[1978] de una pregunta permitida y el axioma de cobertura de Shepherdson[1984]. El siguiente resultado se debe a Lloyd y

Topor y a Shepherdson[1985].

Teorema 56. Sea P un programa normal y G una meta normal, supóngase que P ∪ {G} es permitido, entonces tenemos las siguientes propiedades:

(a) Ningún cómputo para P ∪ {G} tropieza.

(b) Cada respuesta computada para P ∪ {G} es una substitución básica para G.

Las condiciones anteriores se pueden verificar inmediatamente pero son muy restrictivas ya que prohiben cláusulas como p(x) ←.

IV.1 CORRECTEZ DE LA RESOLUCION-LSDNF.

El siguiente resultado es la correctez de la regla de la negación por fracaso. En preparación de la prueba de este resultado tenemos el siguiente lema debido a Clark[1978].

Lema 1. Sea P un programa normal y G una meta normal, supóngase que la literal seleccionada en G es positiva.

(a) Si no hay metas resolventes, entonces G es una consecuencia lógica de comp(P).

(b) Si el conjunto $\{G_1,\ldots,G_r\}$ de metas resolventes es no vacío, entonces $G \leftrightarrow G_1 \wedge \ldots \wedge G_r$ es una consecuencia lógica de comp(P).

El siguiente resultado se debe a Clark[1978].

Teorema 57 (Correctez de la regla de Negación por Fracaso). Sea P un programa normal y G una meta normal, si

P ∪ {G} tiene un árbol-LSDNF con fracaso finito, entonces G es una consecuencia lógica de comp(P).

Demostración.

La demostración es por inducción sobre el rango k del árbol-LSDNF con fracaso finito para P ∪ {G}.

Sea G la meta $\leftarrow L_1,\ldots,L_n$.

Supongamos primero que k = 0, entonces el resultado se sigue utilizando el lema 1.

Supongamos que el resultado se da para árboles-LSDNF con fracaso finito de rango k. Consideremos un árbol-LSDNF con fracaso finito de rango k+1 para P ∪ {G}. Llevaremos a cabo una inducción secundaria sobre la profundidad de este árbol para llegar al resultado:

Supongamos primero que la profundidad de este árbol es 1:

(i) Si la literal seleccionada en G es positiva entonces el resultado se sigue del lema 1(a).

(ii) Si la literal seleccionada L_i en G es la literal negativa sin variables $\neg A_i$, como la profundidad es 1, hay una refutación-LSDNF de rango k para P ∪ {$\leftarrow A_i$}. Nótese que para una meta cuya literal seleccionada es positiva, la meta derivada es una consecuencia lógica de la meta dada y la cláusula de entrada,

P es consecuencia lógica de comp(P) y se tiene la hipótesis de inducción sobre árboles-LSDNF con fracaso finito de rango k-1 en esta refutación, obtenemos que A_i es una consecuencia lógica de comp(P). Por lo tanto, $\neg\exists(L_1 \wedge \ldots \wedge L_n)$ es también una consecuencia lógica de comp(P), debido a que A_i no tiene variables.

Ahora supongamos que el árbol-LSDNF de fracaso finito para P ∪ {G} tiene profundidad d+1.

(i) Si la literal seleccionada en G es positiva entonces el resultado se sigue del lema 1(b) y la segunda hipótesis de inducción.

(ii) Si la literal seleccionada en G es la literal negativa sin variables L_i, por la segunda hipótesis de inducción tenemos que $\neg\exists(L_1 \wedge \ldots \wedge L_{i-1} \wedge L_{i+1} \wedge \ldots \wedge L_n)$ es una consecuencia lógica de comp(P), por lo tanto $\neg\exists(L_1 \wedge \ldots \wedge L_n)$ es también una consecuencia de comp(P). ■

Corolario 5. Sea P un programa definido, si $A \in F_P$ entonces ¬A es una consecuencia lógica de comp(P).

Ahora tenemos la correctez de la resolución-LSDNF, este resultado se debe esencialmente a Clark[1978].

Teorema 58 (Correctez de la resolución-LSDNF). Sea P un programa normal y G una meta normal, entonces cada respuesta computada para $P \cup \{G\}$ es una respuesta correcta para $comp(P) \cup \{G\}$.

Demostración.

Sea G una meta normal $\leftarrow L_1,\ldots,L_k$ y $\theta_1,\ldots,\theta_n$ la sucesión de substituciones usada en una refutación-LSDNF para $P \cup \{G\}$. Tenemos que demostrar que $\forall((L_1 \wedge \ldots \wedge L_k)\theta_1 \ldots \theta_n)$ es una consecuencia lógica de comp(P). El resultado se prueba por inducción sobre la longitud de la refutación-LSDNF.

Supongamos primero que n = 1. Esto significa que G tiene la forma $\leftarrow L_1$. Consideraremos dos casos:

(a) L_1 positiva.

Entonces P tiene una cláusula unitaria de la forma $A \leftarrow$ y $L_1\theta_1 = A\theta_1$. Como $L_1\theta_1 \leftarrow$ es una instancia de una cláusula unitaria de P, se sigue que $\forall(L_1\theta_1)$ es una consecuencia lógica de P y, entonces, de comp(P).

(b) L_1 negativa.

En este caso, L_1 no tiene variables, θ_1 es la substitución identidad y el teorema anterior demuestra que L_1 es una consecuencia lógica de comp(P).

Ahora supongamos que el resultado se da para respuestas computadas que provienen de refutaciones-LSDNF

de longitud n-1. Supongamos que $\theta_1,\ldots,\theta_n$ es la sucesión de substituciones usada en la refutación-LSDNF para $P \cup \{G\}$ de longitud n. Sea L_m la literal seleccionada de G. Otra vez consideraremos dos casos:

(a) L_m positiva.

Sea $A \leftarrow M_1,\ldots,M_q$ $(q \geq 0)$ la primera cláusula de entrada. Por hipótesis de inducción

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

es una consecuencia lógica de comp(P). Además, si $q > 0$, $\forall((M_1 \wedge \ldots \wedge M_q)\theta_1 \ldots \theta_n)$ es una consecuencia lógica de comp(P). Consecuentemente, $\forall(L_m\theta_1 \ldots \theta_n)$ que es lo mismo que $\forall(A\theta_1 \ldots \theta_n)$, es una consecuencia lógica de comp(P). Si $q = 0$, es inmediato que $\forall(L_m\theta_1 \ldots \theta_n)$ es consecuencia lógica de comp(P). Por lo tanto tenemos que $\forall((L_1 \wedge \ldots \wedge L_k)\theta_1 \ldots \theta_n)$ es una consecuencia lógica de comp(P).

(b) L_m negativa.

En este caso, L_m no tiene variables, θ_1 es la substitución identidad y el teorema anterior demuestra que L_m es una consecuencia lógica de comp(P). Usando la hipótesis de inducción tenemos que $\forall((L_1 \wedge \ldots \wedge L_k)\theta_1 \ldots \theta_n)$ es una consecuencia lógica de comp(P). ■

Finalmente, veamos el problema de debilitar la condición

de seguridad sobre la selección de literales. Primero mostraremos que si se retira la condición de seguridad, entonces no tenemos la correctez para la regla de negación por fracaso.

Ejemplo 39. Sea P el programa normal siguiente:

$$p \leftarrow \neg q(x)$$

$$q(c) \leftarrow$$

Si no tenemos la condición de seguridad se puede seleccionar la literal $\neg q(x)$ y obtener un "árbol-LSDNF con fracaso finito" para $P \cup \{\leftarrow p\}$. La submeta $\neg q(x)$ fracasa porque hay una refutación de $\leftarrow q(x)$ en la que x se substituye por c. Sin embargo, $\neg p$ no es una consecuencia lógica de comp(P).

Es posible debilitar la condición de seguridad un poco y todavía obtener resultados. Considérese la siguiente condición de seguridad debilitada. Se permite proceder con submetas negativas con variables. Si la submeta negativa tiene éxito, entonces se procede como antes. Sin embargo, si la submeta negativa fracasa, se realiza un chequeo para asegurarse que no se hicieron substituciones a ninguna variable en la meta del nivel tope de la refutación correspondiente. Si no se hizo tal substitución, entonces la submeta negativa fracasa y se procede como antes. Pero, si se hizo una substitución, entonces se selecciona una literal diferente y se retrasa la selección de la submeta negativa, esperando a que después se acoten más de sus variables.

Alternativamente, se puede generar un error de control y parar el programa.

El punto clave aquí (en el caso de submeta negativa con variables) es que la refutación que causa que fracase la submeta negativa con variables debe probar algo de la forma ∀(A) en lugar de solo ∃(A). Para esta condición de seguridad debilitada, se siguen dando los dos teoremas anteriores. El único cambio a sus pruebas es en la prueba del teorema de la correctez de la regla de negación por fracaso, en el lugar donde remarcamos el uso del hecho de que A es sin variables.

La manera más simple de implementar la condición de seguridad en un sistema PROLOG es retrasar la selección de las submetas negativas hasta que todas las variables que aparezcan en la submeta hayan sido acotadas. Desafortunadamente, la mayoría de los sistemas PROLOG no tienen un mecanismo para retrasar la selección de las submetas y entonces no se dispone de esta solución. Más aún todavía, la mayoría de los sistemas PROLOG no verifican que las submetas negativas no tengan variables cuando son llamadas, esto puede llevar a un funcionamiento peor.

Dahl[1980] sugiere dos métodos para no evaluar una literal negativa con variables libres en PROLOG, uno es usar corrutinas para ordenar la meta y otro es utilizando conjuntos cuando se tiene un universo finito.

Ejemplo 40. Considérese el programa siguiente:

p(a) ←

q(b) ←

y la meta normal ← ¬p(x),q(x). Si este programa y meta corren en un sistema PROLOG que utilice la regla de cómputo estándar y que no cheque que las submetas negativas no tengan variables al ser llamadas, entonces dará la respuesta "no". De otro modo, se retrasa la primera submeta, se resuelve la segunda submeta y después la primera para dar la respuesta correcta {x/b}. Por supuesto, el problema con esta meta particular puede ser solucionado con un sistema PROLOG estándar reordenando las submetas en la meta. Sin embargo, esto no es el asunto.

Otra solución para el manejo de literales negativas con variables, es lo que propone Kunen[1987a]. Uno puede construir un intérprete PROLOG modificado que maneje lo que él llama conjuntos elementales, es decir, combinaciones booleanas finitas de conjuntos de la forma $\{(t_1\sigma,\ldots,t_n\sigma) \mid \sigma \text{ substitución}\} \subseteq B_P^n$ con $t_1,\ldots,t_n$ términos. Este intérprete podría diferir del PROLOG estándar en la forma en que trata una meta o submeta negativa, ¬L, donde L tiene variables sin instanciar. El intérprete podría llamarse a sí mismo recursivamente y calcular el conjunto de respuesta completo para L como la unión de una cadena de conjuntos elementales. Si la cadena es finita, entonces la respuesta para ¬L sería el complemento de la unión. Si la cadena es infinita, entonces el intérprete no para. Este intérprete

podría calcular los conjuntos de respuesta completa para cualquier programa jerárquico aunque podría carecer de la propiedad de completez.

Ejemplo 41. Sea el programa

$$A(c) \leftarrow$$

y la meta $\leftarrow \neg A(x)$ entonces la respuesta sería $\{x|x = c\}^c = \{x|x \neq c\}$.

Ahora discutiremos el efecto que pueden tener los cortes en un programa normal, sobre los resultados de correctez. Antes, mostramos que la existencia de un corte en un programa definido no afecta la correctez, pero puede introducir una forma de incompletez en la implementación de la resolución-LSD de respuesta correcta. Sin embargo, para programas normales, es posible que un corte afecte la correctez.

Ejemplo 42. Considérese el programa de subconjunto:

subconjunto(x,y) ← ¬p(x,y)

p(x,y) ← miembro(z,x), ¬miembro(z,y)

miembro(x,x.y) ← !

miembro(x,y.z) ← miembro(x,z)

en donde los conjuntos se representan por listas. La meta ← subconjunto([1,2,3],[1]) tiene éxito con este programa. La razón es que el uso inseguro del corte en la definición de miembro produce un árbol con fracaso finito para ← p([1,2,3],[1]) construído incorrectamente. Por lo que la submeta negativa ¬p([1,2,3],[1]) tiene éxito incorrectamente.

IV.2 COMPLETEZ DE LA RESOLUCION-LSDNF.

En esta sección, probaremos los resultados de completez de la regla de negación por fracaso para programas definidos y la resolución-LSDNF para programas jerárquicos.

El siguiente resultado se debe a Jaffar, Lassez y Lloyd[1983]. La demostración que mostramos a continuación es más sencilla que la original y se debe a Wolfram, Maher y Lassez.

Teorema 59 (Completez de la regla de Negación por Fracaso). Sea P un programa definido y G una meta definida, si G es una consecuencia lógica de comp(P) entonces cada árbol-LSD equitativo para P ∪ {G} fracasa de forma finita.

Demostración.

Sea G la meta $\leftarrow A_1,\ldots,A_q$. Supóngase que P ∪ {G} tiene un árbol-LSD equitativo que no fracasa de forma finita. Probaremos que comp(P) ∪ $\{\exists(A_1 \wedge \ldots \wedge A_q)\}$ tiene un modelo.

Sea RA una rama que no fracasa en el árbol-LSD equitativo para P ∪ {G}. Supóngase que RA es $G_0=G, G_1,\ldots$ con umg's $\theta_1, \theta_2,\ldots$ y cláusulas de entrada $C_1, C_2,\ldots$. El primer paso es usar RA para definir una pre-interpretación J de P.

Supóngase que L es el lenguaje de primer orden para P. Se define una relación $*$ sobre el conjunto de todos los términos en L como sigue. Sean s y t términos en L,

entonces $s*t$ si hay $n \geq 1$ tal que $s\theta_1 \ldots \theta_n = t\theta_1 \ldots \theta_n$, esto es, $\theta_1 \ldots \theta_n$ unifica a s y t. $*$ es una relación de equivalencia. Se define el dominio D de la pre-interpretación J como el conjunto de todas las clases de equivalencia de $*$ de los términos en L. Si s es un término en L, se denota a la clase de equivalencia que contiene a s por [s].

Después se dan las asignaciones a las constantes y símbolos funcionales en L. Si c es una constante en L, se asigna [c] a c. Si f es un símbolo funcional n-ario en L, se asigna a f el mapeo de D^n a D definido por $([s_1],\ldots,[s_n]) \rightarrow [f(s_1,\ldots,s_n)]$. Esto termina la definición de J.

La siguiente tarea es dar las asignaciones a los símbolos de predicado para extender J a una interpretación de $comp(P) \cup \{\exists(A_1 \wedge \ldots \wedge A_n)\}$. Primero se define el conjunto I_0 como sigue:

$$I_0 = \{p([t_1],\ldots,[t_n]) \mid p(t_1,\ldots,t_n) \text{ aparece en RA}\}.$$

Se mostrará que $I_0 \subseteq T_P^J(I_0)$, donde T_P^J es el mapeo asociado a la pre-interpretación J. Supóngase que $p([t_1],\ldots,[t_n]) \in I_0$, donde $p(t_1,\ldots,t_n)$ aparece en alguna G_i, $i \in \omega$. Como RA es equitativo y no fracasa, hay una $j \in \omega$ tal que

$$p(s_1,\ldots,s_n) = p(t_1,\ldots,t_n)\theta_{i+1}\ldots\theta_{i+j}$$

aparece en la meta G_{i+j} y $p(s_1,\ldots,s_n)$ es el átomo seleccionado en G_{i+j}. Supóngase que C_{i+j+1} es

$p(r_1,\ldots,r_n) \leftarrow B_1,\ldots,B_m$. Por la definición de T_P^J se sigue que $p([r_1\theta_{i+j+1}],\ldots,[r_n\theta_{i+j+1}]) \in T_P^J(I_0)$. Se tiene:

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

así que $p([t_1],\ldots,[t_n]) \in T_P^J(I_0)$. Entonces $I_0 \subseteq T_P^J(I_0)$.

Ahora, por el teorema 7 (página 23) hay una I tal que $I_0 \subseteq I$ e $I = T_P^J(I)$. I da las asignaciones a los símbolos de predicado en L. Se asigna a = la relación identidad sobre D.

Esto termina la definición de la interpretación I, junto con la relación identidad asignada a =, de $comp(P) \cup \{\exists(A_1 \wedge \ldots \wedge A_q)\}$. Nótese que esta interpretación es claramente un modelo para la teoría de la igualdad. Entonces por el teorema 37 (página 56), I junto con la relación de identidad asignada a = es un modelo de $comp(P) \cup \{\exists(A_1 \wedge \ldots \wedge A_n)\}$. ■

Corolario 6. Sea P un programa definido y $A \in B_P$, si $\neg A$ es una consecuencia lógica de comp(P) entonces $A \in F_P$.

Ahora analizaremos la completez de la resolución-LSDNF.

Ejemplo 43. Considérese el programa:

$p(x) \leftarrow$

$q(a) \leftarrow$

r(b) ←

y la meta ← p(x), ¬q(x). Claramente, x/b es una respuesta correcta, sin embargo, esta respuesta nunca puede ser computada, ni ninguna versión más general de ella.

Este ejemplo ilustra claramente uno de los problemas para obtener un resultado de completez para la resolución-LSDNF. La resolución-LSD da respuestas más generales. En el ejemplo anterior, ésta regresará la substitución identidad Ø para la submeta p(x). Lo que quisiéramos para la regla de negación por fracaso es instanciar x con la substitución x/b y así calcular la respuesta correcta. Sin embargo, la negación por fracaso solamente es una prueba y no puede hacer substituciones. A menos que se presente una meta que sea la raíz de un árbol-LSD con fracaso finito, no hay manera de instanciar más la meta para obtener dicho árbol. En el ejemplo anterior, ← q(x) no es la raíz de un árbol-LSD con fracaso finito y la negación por fracaso no tiene manera de encontrar la substitución apropiada x/b.

El siguiente ejemplo ilustra otro problema para obtener un resultado de completez para la resolución-LSDNF.

Ejemplo 44. Considérese el programa normal P

r ← p

r ← ¬p

p ← p

entonces, la substitución identidad Ø es una respuesta

correcta para comp(P) ∪ {← r}, pero ∅ no puede ser computada.

Estos ejemplos muestran que para obtener un resultado de completez, será necesario imponer más restricciones. Ahora mostraremos que para programas jerárquicos hay un resultado de completez. Sin embargo, este resultado no es muy útil porque la condición de jerarquía prohibe cualquier recursión, aunque todavía caracteriza a una base de datos convencional ya que: se pueden tener definiciones de relaciones por un conjunto de instancias, o por reglas generales, o por mezcla de ambos; los componentes de una relación son estructuras de datos generales, no solo cadenas y números; la notación clausal cubre los papeles de lenguaje de descripción de datos, lenguaje de preguntas y lenguaje de programación; la recuperación de información no sólo busca en un archivo sino que incluye realmente una deducción computacional. Esta condición es muy apropiada cuando se habla acerca de derivar consecuencias de la base de datos completada, ya que para obtenerla a partir de la base de datos original se convierten las cláusulas *si* en *si y solo si*, y esto sólo ocurre cuando se define alguna nueva relación en términos de relaciones ya definidas, que es precisamente la situación de jerarquía. Para el enunciado de este resultado, necesitamos generalizar el concepto de una regla de cómputo.

Una *regla de cómputo segura* es una función del conjunto de metas normales, ninguna de las cuales consiste solamente de

literales negativas con variables, al conjunto de literales tal que el valor de la función para una meta es una literal en dicha meta ya sea positiva, o negativa sin variables, llamada la literal *seleccionada*.

Sea P un programa normal, G una meta normal, y R una regla de cómputo segura entónces:

Una *derivación-LSDNF* para P ∪ {G} *vía* R es una derivación-LSDNF para P ∪ {G} en la que se utiliza la regla de cómputo R para seleccionar literales.

Un *árbol-LSDNF* para P ∪ {G} *vía* R es un árbol-LSDNF para P ∪ {G} en el que se utiliza la regla de cómputo R para seleccionar literales.

Una *refutación-LSDNF* para P ∪ {G} *vía* R es una refutación-LSDNF para P ∪ {G} en la que se utiliza la regla de cómputo R para seleccionar literales.

Una *respuesta computada-R* para P ∪ {G} es una respuesta computada para P ∪ {G} que proviene de una refutación-LSDNF para P ∪ {G} vía R.

Se tiene el siguiente resultado que se puede consultar en Shepherdson[1984].

Teorema 60. Una meta G no puede tener éxito bajo una regla de selección y fracasar bajo otra.

Ahora podemos dar el resultado de completez para programas jerárquicos. Algunas versiones de este resultado se deben a Clark[1978] y a Shepherdson[1985].

Teorema 01 (Completez de la resolución-LSDNF para programas jerárquicos). Sea P un programa normal jerárquico, G una meta normal, y R una regla de cómputo segura. Supóngase que P ∪ {G} es permitido, entonces se tienen las siguientes propiedades:

(a) El árbol-LSDNF para P ∪ {G} vía R existe y es finito.

(b) Si θ es una respuesta correcta para comp(P) ∪ {G} y θ es una substitución básica para G, entonces θ es una respuesta computada-R para P ∪ {G}.

Shepherdson[1988] define varios tipos de completez:

(a) θ-completez para G, si para todas las substituciones θ, comp(P) ⊨ ¬Gθ implica que G tiene éxito con respuesta que incluya a θ (es decir, respuesta σ tal que existe substitución α con $\theta = \sigma\alpha$).

(b) ∃-completez para G, si

comp(P) ⊨ ∃¬G implica que G tiene éxito.

(c) ¬-completez para G, si

comp(P) ⊨ ¬∃¬G implica que G fracasa.

Además, da el siguiente resultado:

Teorema 02. Para programas definidos la resolución-LSDNF es:

(a) θ-completa para todas las metas G tales que P ∪ {G} es permitido pero no para toda G permitida.

(b) ∃-completa para todas las metas sin variables pero no para todas las metas G tales que P ∪ {G} es

permitido.

(c) ¬-completa para todas las metas sin variables puramente negativas pero no para todas las metas sin variables.

Ejemplo 45. Sea P el programa:

$$p(x) \leftarrow$$

y la meta $\leftarrow p(x), \neg q(x)$.

Aquí se tiene que la meta es permitida y tiene como respuesta correcta a $\emptyset$ pero la resolución-LSDNF no da respuesta ya que llega a $\neg q(x)$, donde tropieza.

Ejemplo 46. Sea P el programa:

$$p(a) \leftarrow$$

$$p(f(a)) \leftarrow p(f(a))$$

y la meta $\leftarrow p(x), \neg p(f(x))$.

Aquí se tiene que $P \cup \{\leftarrow p(x), \neg p(f(x))\}$ es permitido y $\exists x(p(x) \wedge \neg p(f(x)))$ es verdadero ya que $comp(P) \models p(a) \vee p(f(a))$, pero la resolución-LSDNF se queda en un loop.

Ejemplo 47. Sea P el programa:

$$p \leftarrow p$$

y la meta $\leftarrow p, \neg p$.

Aquí se tiene que $p, \neg p$ no tiene variables y es falsa, pero la resolución-LSDNF se queda en un loop.

La completez de la regla de negación por fracaso y la resolución-LSDNF son de tal importancia que es una prioridad urgente encontrar resultados de completez más generales. Los

resultados de completez más interesantes podrían ser para clases de programas estratificados, que incluyen estrictamente a la clase de programas jerárquicos.

IV.3 RELACION CON LA SUPOSICION DEL MUNDO CERRADO.

Los siguientes resultados se deben a Shepherdson[1984].

Se puede ver que la resolución-LSDNF es un subsistema de SMC(P) así como de comp(P), aunque no es obvio inmediatamente. Los pasos positivos en la resolución-LSDNF son ciertamente inferidos del programa P, entonces también de SMC(P), pero como la negación se toma como fracaso para probar puede parecer a primera vista como si SMC(P) fuera un subsistema de la resolución-LSDNF. Sin embargo la resolución-LSDNF necesita para fracasar no sólo falta de éxito sino un árbol de evaluación finito que explore todos los pasos posibles y mostrar que terminan en fracaso.

Aunque SMC(P) y comp(P) pueden ser uno consistente mientras el otro no, hay dos casos importantes en los que son compatibles: primero, si P es definido, y segundo, si hay una resolución-LSDNF que sea completa para las metas sin variables, y si SMC(P) es consistente. En este caso SMC(P) y comp(P) no sólo son compatibles sino efectivamente equivalentes.

El hecho de que comp(P) y su consistencia dependan de la manera en cómo se escriben las cláusulas del programa, y pueda

tener efectos no deseados en el uso de definiciones recursivas, sugiere que cuando SMC(P) y comp(P) son compatibles, la SMC(P) debería ser una justificación más aceptable de la regla de la negación por fracaso que comp(P), aunque primero se debe verificar la consistencia.

Teorema 63. Sea P un programa normal y $\leftarrow L_1,\ldots,L_k$ una meta normal, entonces cada respuesta computada θ para $P \cup \{\leftarrow L_1,\ldots,L_k\}$ es tal que $SMC(P) \models \forall((L_1 \wedge \ldots \wedge L_k)\theta)$.

La razón de por qué SMC(P) y comp(P) trabajan igualmente bien aquí puede ser explicado brevemente. Los pasos positivos en la resolución-LSDNF son pasos de resolución que son consecuencias válidas solo de P. Los pasos negativos básicos (es decir, los que no dependen de otros), asignan el valor falso a $\neg A$ cuando la literal sin variables A tiene éxito, en cuyo caso A es implicado por P, es decir $\neg(\neg A)$, y asigna el valor verdadero a $\neg A$ cuando fracasa. Este último paso es el único que necesita más que P para su justificación. Si A fracasa entonces A o alguna otra literal positiva B obtenida al aplicar pasos de unificación fracasa al unificarse con una (instancia de) cabeza de cláusula de programa de P. La mitad *solo si* de comp(P) permite afirmar $\neg A$, SMC(P) también permite afirmar $\neg A$ ya que si A fracasa entonces $P \not\models A$, por lo tanto $\neg A \in ext(P)$.

Teorema 64. Si para cada literal sin variables hay alguna regla de selección bajo la cual ésta fracasa o tiene éxito bajo la resolución-LSDNF, entonces comp(P) es consistente y si

SMC(P) es consistente, comp(P) ∪ SMC(P) es consistente.

Corolario 7. Si SMC(P) es consistente y ← L tiene éxito con L un átomo sin variables, entonces P ⊨ L.

No podemos ver el uso de la resolución-LSDNF como equivalente a la completación del programa o usar la SMC(P), sino sólo como un intento incompleto para producir ambas, que obviamente se acercará más a su parte común que a alguna de ellas. Además tenemos que los resultados de la resolución-LSDNF no dependen sólo del contenido lógico del programa, sino también de la forma en que está escrito.

IV.4 RELACION CON LOS RESULTADOS ANTERIORES.

Resumiremos los resultados principales para programas definidos. Primero necesitamos más definiciones. La *regla de Herbrand* es como sigue: si comp(P) ∪ {A} no tiene modelo de Herbrand, o bien, si todo modelo de Herbrand de comp(P) es modelo de ¬A, entonces inferir ¬A.

Ahora tenemos tres reglas posibles para inferir información negativa: la SMC(P), la regla de Herbrand y la regla de negación por fracaso.

Como el operador T_P es monótono para un programa P definido, no sólo tiene un mínimo punto fijo mpf(T_P), sino también un máximo punto fijo MPF(T_P), que es la unión de todos los puntos fijos (ver teorema 7, página 23). Por el teorema 38 (página 57) éste es el máximo modelo de Herbrand de

comp(P). Así la regla de Herbrand permite inferir ¬A, si A pertenece al complemento (con respecto a la base de Herbrand B_P) de MPF(T_P).

El conjunto de todos los átomos sin variables que son falsos bajo la Suposición del Mundo Cerrado, es decir, tales que $P \not\models A$, es el complemento de mpf(T_P) = $T_P\uparrow\omega$ (ver teorema 2, página 16 y teorema 14, página 27).

Además tenemos que el conjunto de las A tales que comp(P) $\models \neg A$, es el conjunto de todos los átomos que fracasan bajo reglas de cómputo equitativas, que a su vez coincide con el conjunto de fracaso finito LSD, es decir, el conjunto de átomos que fracasan bajo alguna regla de cómputo y también con el complemento de $T_P\downarrow\omega$ (ver teorema 29, página 45 y corolario 5, página 86).

Así tenemos los siguientes resultados (ver figura 5):

$\{A \in B_P \mid \neg A$ puede ser inferida bajo la regla de Herbrand$\} = B_P - MPF(T_P)$

$\{A \in B_P \mid \neg A$ puede ser inferida bajo la SMC$\} = B_P - T_P\uparrow\omega$

$\{A \in B_P \mid \neg A$ puede ser inferida bajo la regla de negación por fracaso$\} = B_P - T_P\downarrow\omega$

Como $T_P\uparrow\omega \subseteq MPF(T_P) \subseteq T_P\downarrow\omega$, se sigue que la SMC es la regla más poderosa, seguida por la regla de Herbrand, seguida por la regla de negación por fracaso. Como $T_P\uparrow\omega$, MPF(T_P) y $T_P\downarrow\omega$ son generalmente distintos, entonces las reglas son distintas (ver ejemplo 11, página 27).

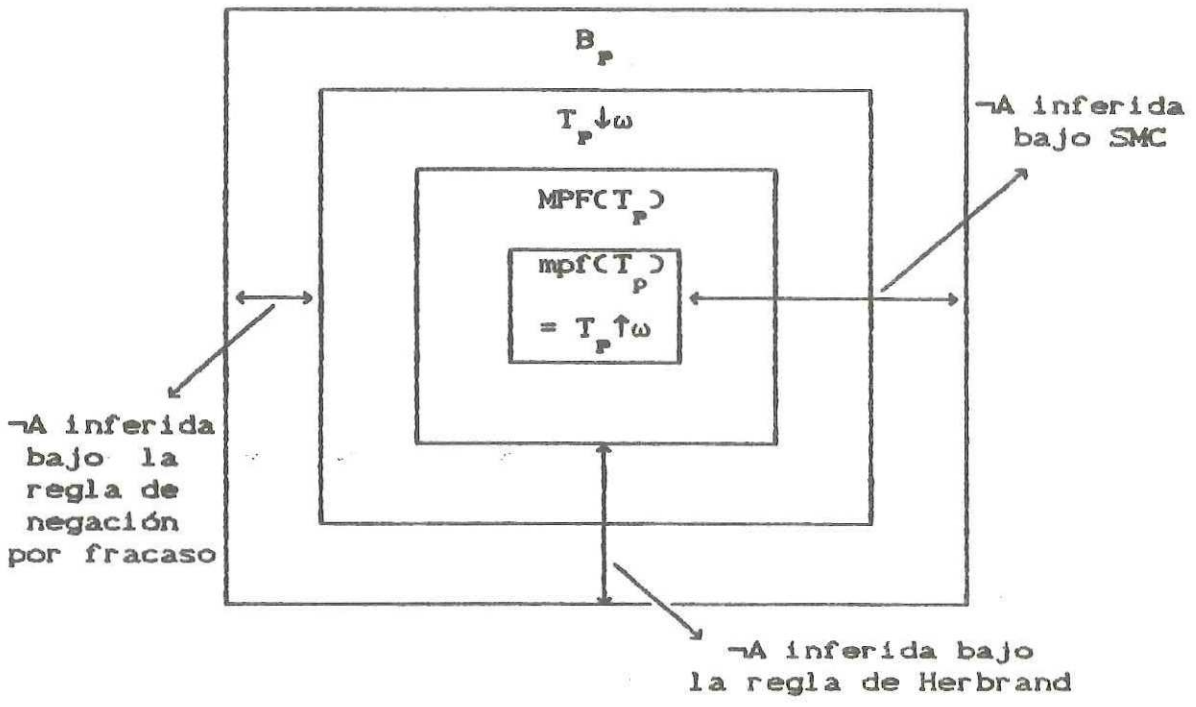


Figura 5. Relación entre las diferentes reglas.

Combinando el teorema 29 (página 45) y los corolarios 5 (página 88) y 6 (página 93) tenemos:

Teorema 65. Sea P un programa definido y $A \in B_P$ entonces los siguientes resultados son equivalentes:

(a) $A \in F_P$.

(b) $A \notin T_P\downarrow\omega$.

(c) A está en el conjunto de fracaso finito LSD.

(d) Cada árbol-LSD equitativo para P ∪ (← A) fracasa de forma finita.

(e) ¬A es una consecuencia lógica de comp(P).

Combinando los teoremas 57 (página 84) y 59 (página 93) tenemos:

Teorema 66. Sea P un programa definido y G una meta definida entonces G es una consecuencia lógica de comp(P) si y solo si P ∪ {G} tiene un árbol-LSD con fracaso finito.

Para enfatizar la diferencia entre modelos (arbitrarios) y modelos de Herbrand de comp(P) y entre $T_P{\downarrow}\omega$ y $MPF(T_P)$. Sea $A \in B_P$, entonces tenemos las siguientes propiedades:

(a) $A \in MPF(T_P)$ si y solo si comp(P) ∪ {A} tiene un modelo de Herbrand.

(b) $A \in T_P{\downarrow}\omega$ si y solo si comp(P) ∪ {A} tiene un modelo.

Jaffar y Stuckey sugieren que todos los programas "decentes" deben satisfacer $T_P{\downarrow}\omega = MPF(T_P)$ y los llaman canónicos. Otra condición natural es que comp(P) debe tener exactamente un modelo de Herbrand, es decir, que $mpf(T_P) = MPF(T_P)$. El programa del ejemplo 11 (página 27) tiene esta propiedad pero no es canónico.

Ejemplo 48. Sea P el programa:

$$p \leftarrow p$$

es canónico pero $mpf(T_P) = \emptyset \neq \{p\} = MPF(T_P)$.

Un programa que satisface estas dos condiciones, es decir, $T_P{\uparrow}\omega = T_P{\downarrow}\omega$ se llama determinado. Esto significa que la negación por fracaso es completa con respecto a SMC(P) así como con respecto a comp(P). Esto quiere decir no sólo que hay un único modelo de Herbrand sino que la intersección de cada modelo con la base de Herbrand está determinada, en otras

palabras, para cada átomo sin variables A se tiene que comp(P) $\models$ A o comp(P) $\models \neg$A.

IV.5 OTRAS SEMANTICAS PARA LA NEGACION POR FRACASO.

Kunen[1987b] propone otra semántica para la negación por fracaso en la que utiliza la lógica tri-valuada.

Hay tres tipos de respuesta para una meta $\leftarrow$ M:

(a) *no*, si $\exists$M no es consecuencia lógica de P

(b) *una o más substituciones* σ, si $\forall$Mσ es consecuencia de P, si M no tiene variables, la respuesta es *sí*

(c) *no claro*, que significa que el intérprete no para o que si para imprime un mensaje de error en lugar de una solución.

Los problemas con la negación hacen claro que la semántica no puede ser precisamente la semántica estándar de lógica de predicados de primer orden, pero es importante que sea declarativa, es decir, expresada en términos de nociones lógicas. El usuario deberá entender la semántica sólo por *lógica*, no por un entendimiento detallado de la implementación del intérprete. Esto es realmente el hecho clave que separa a la programación lógica de formas de programación más convencionales.

El requerimiento básico de cualquier intérprete de PROLOG debe ser la *correctez*, esto es, si se regresa una respuesta σ

para una pregunta ← M, entonces ∀Mσ es consecuencia lógica de P, mientras que si se regresa *no* entonces ∃M no es consecuencia lógica de P. Vemos la respuesta de la máquina de *no claro* como una verificación incompleta de si ∃M es consecuencia lógica de P o no, así, si un intérprete siempre regresa *no claro* es trivialmente correcto (e inútil).

Se toma la aproximación de la lógica tri-valuada, así, tenemos tres valores de verdad primitivos: v, f e i (verdadero, falso e indefinido) con las tablas de verdad dadas por Kleene, teniendo así la descripción de cómputos que fracasan al dar una respuesta.

Las tablas de verdad de Kleene pueden resumirse como sigue. Si ϕ es una combinación booleana de los átomos v, f e i, su valor de verdad es v si y solo si todas las maneras posibles de poner v ó f en las ocurrencias de i llevan al valor v calculado en lógica ordinaria bi-valuada; ϕ tiene valor f si y solo si $\neg\phi$ tiene valor v, y ϕ tiene valor i en cualquier otro caso.

Estos valores de verdad pueden ser extendidos en la manera obvia a lógica de predicados, pensando en los cuantificadores como conjunciones o disyunciones infinitas. Así, $\exists x\phi(x)$ tiene valor de verdad v si y solo si hay al menos un elemento en el dominio de discurso para el cual ϕ es v, tiene valor f si y solo si es f para todos los elementos y es i en cualquier otro caso. De forma similar para cuantificadores universales.

Kunen[1987b] describe dos definiciones equivalentes de la semántica formal, una basada en puntos fijos y otra vía modelos tri-valuados de comp(P).

Los modelos bi-valuados son un caso especial de los modelos tri-valuados, entonces la semántica es más restrictiva que la semántica comp(P) de Clark.

Kunen[1987b] da ejemplos de intérpretes que son correctos pero no completos para esta semántica.

Apt, Blair y Walker[1988] consideran los modelos mínimos que son *soportados*, en el sentido de que cada objeto en ese modelo es un hecho en el programa, o es la conclusión de una instancia sin variables de una regla cuyo cuerpo es verdadero en el modelo.

Se dice que una interpretación I de un programa P es *soportada* si para cada $A \in I$ hay una cláusula $A_1 \leftarrow L_1,\ldots,L_m$ en P y una substitución θ tal que $I \models L_1\theta,\ldots,L_m\theta$, $A = A_1\theta$, y cada $L_i\theta$ no tiene variables. Así, I es soportada si y solo si para cada $A \in I$ hay una cláusula en el conjunto de todas las instancias sin variables de cláusulas en P con cabeza A cuyo cuerpo es verdadero en I.

Un uso seguro de la negación es el siguiente: cuando se usa negación, uno se debe referir a una relación ya conocida. Más específicamente, primero se deben definir algunas relaciones (quizás recursivamente) en términos de sí mismas sin el uso de negación, después se pueden definir algunas nuevas relaciones en términos de sí mismas sin el uso de

negación y en términos de las anteriores posiblemente con el uso de la negación. Este proceso puede ser iterativo. Así se tienen los programas estratificados.

Lo que se quiere es un modelo minimal y soportado.

Se tiene que para P programa, I es un modelo de P si y solo si $T_P(I) \subseteq I$, y además, I es soportada si y solo si $I \subseteq T_P(I)$.

Se define M_P para programas estratificados en base a T_P y la estratificación como sigue. Sea $P = P_1 \cup \ldots \cup P_n$ tal que una cláusula está en P_i si y solo si el símbolo de predicado de su cabeza tiene nivel i, entonces se tiene:

$$M_1 = T_{P_1} \uparrow \omega(\emptyset)$$

$$M_2 = T_{P_2} \uparrow \omega(M_1),$$

$$\vdots$$

$$M_n = T_{P_n} \uparrow \omega(M_{n-1})$$

$$M_P = M_n$$

($T_P \uparrow \omega(M)$ se define como antes sólo que en lugar de empezar con $\emptyset$ se empieza con M a hacer la iteración).

Se tiene que M_P es punto fijo de T_P y por lo tanto modelo mínimo y soportado de P, además es independiente de la estratificación de P.

Hay átomos sin variables que son verdaderos en M_P pero no son consecuencia lógica de P. Un programa estratificado P tiene un único modelo propuesto, llamado M_P, la semántica propuesta por Apt, Blair y Walker[1988] es reducir la noción de consecuencia lógica de P a la de verdadero en M_P.

Es importante ver que la visión de negación representada por comp(P) y M_P no coincide.

Ejemplo 49. Sea P el programa siguiente:

$$p \leftarrow p$$

$$q \leftarrow \neg p$$

entonces se tiene:

comp(P) = $\{p \leftrightarrow p,\ q \leftrightarrow \neg p\}$ equivalente a $\{q \leftrightarrow \neg p\}$, por lo tanto comp(P) $\models \neg p$, mientras que $M_P = \{q\}$ y entonces $M_P \models \neg p$.

Se tiene una teoría completa ya que si $A \in B_P$, A es verdadero si está en M_P y es falso si no está en M_P, además no se puede tener A falso y verdadero.

Van Gelder[1988] presenta una nueva semántica, la semántica de árbol hermético (tight), para programas normales. Define la negación en programas normales como fracaso finito, limitando las pruebas a derivaciones herméticas que define como aquéllas que se expresan por árboles en los que ningún nodo tiene un ancestro idéntico, consecuentemente muchas metas que no fracasan en forma finita en otras formulaciones lo hacen en ésta, así, la regla de negación por fracaso se fortalece pero al costo de una ejecución de programa más cuidadosa (y cara).

Van Gelder[1988] muestra que los programas normales con la propiedad de tamaño de término acotado que asegura que ningún árbol de derivación hermética tiene un camino infinito, y la propiedad de libertad de negación recursiva que asegura

que dos metas no pueden estar cada una esperando que la otra fracase, son tales que bajo la semántica de árboles herméticos cada elemento de B_p es verdadero o falso. Tener la hermeticidad evita pruebas redundantes mientras mantiene la completez.

Przymusinski[1988] define lo que es modelo perfecto de un programa y dice que esta clase de modelos da una semántica correcta.

Los modelos perfectos son un subconjunto de los minimales y para programas definidos coinciden con los minimales.

Cada programa estratificado tiene un único modelo perfecto que coincide con el modelo M_p propuesto por Apt, Blair y Walker[1988], y con el modelo propuesto por Van Gelder[1988], además se tiene el siguiente resultado:

Teorema 67. Sea P estratificado y $P_1, \ldots, P_n$ tales que $p \in P_i$ si y solo si el nivel de p es i, entonces un modelo de P es perfecto si y solo si es un modelo de la circunscripción con prioridad circ(P; $P_0 > \ldots > P_n$).

No siempre se tiene un modelo perfecto, para P normal.

Przymusinski[1988] cree que la noción de modelo perfecto es conceptualmente más simple y más natural que la circunscripción con prioridad.

IV.6 IMPLEMENTACION DE LA NEGACION POR FRACASO.

La implementación de la negación por fracaso que se da en

PROLOG es la siguiente:

```
no(X) ← X,!,fail.
no(X) ←.
```

que tiene el inconveniente de que no verifica si las literales negativas que se ejecutan no tienen variables.

Una solución para hacer esta verificación sería la siguiente:

```
no(X) ← numerovars(X,0,N),N = 0,X,!,fail.
no(X) ← numerovars(X,0,N),N ≠ 0,escribe('literales
        negativas con variables'),!,fail.
no(X) ←.
```

donde numerovars(X,0,N) indexa las variables de X empezando en 0 y terminando en N-1, por lo tanto, N es el número de variables que tiene X.

Cuando se utiliza la negación por fracaso suponemos que sabemos todo acerca de cada predicado del dominio. Minker y Perlis[1985] dan una solución cuando se tienen datos indefinidos, que es la circunscripción protegida. Se quiere usar SMC excepto para aquellos datos que se sabe que no están completos, esto es, proteger los datos de la circunscripción.

Primero trabajan con programas que tienen sólo átomos sin variables. Se desean programas que contengan una forma de valor nulo con el que no se sabe cuándo un átomo particular es verdadero o falso, dicho dato desconocido se conoce como *excepcional* y al proceso de representar esta situación como *protección*. Para calcular la negación de un átomo, uno toma

el dominio entero, quita aquellas constantes que satisfacen el átomo positivo y también las constantes protegidas, entonces las constantes restantes son las que se asume que satisfacen la negación del átomo.

Después consideran programas definidos permitiendo excepciones (datos protegidos) pero sin símbolos funcionales. Para que su solución sea equivalente a la circunscripción protegida se "prepara" al programa aumentando protección extra que la circunscripción toma implícitamente.

Por último dan una implementación en PROLOG que es la siguiente, donde $\neg_E$ es la nueva negación relativa a la suposición del mundo cerrado protegida y E indica la excepción o dato protegido:

$\neg_E P \leftarrow P, !, fail.$

$\neg_E P \leftarrow EP, !, fail.$

$\neg_E P \leftarrow .$

V. NEGACION POR INCONSISTENCIA.

Este tipo de negación fue concebido por Gabbay y Sergot[1986]. Es introducido en PROLOG, reemplazando la noción de negación por fracaso.

Para este tipo de negación se requieren dos conjuntos a partir de los cuales se van a hacer las deducciones. El primero (P), es un conjunto de cláusulas de las que se van a deducir las metas de la forma en que se hace en Programación Lógica. El segundo (N), es un conjunto de resultados no deseados, es decir, un conjunto de metas que no se pueden deducir de nuestro programa.

La noción de inconsistencia es en la que se basa este método. Un par de conjuntos, de datos y de metas, de la forma (P,N) es inconsistente si alguna meta $n \in N$ se deduce de P, es decir, (P,N) es inconsistente porque queremos que $n \in N$ no tenga éxito y al mismo tiempo P es tal que n tiene éxito. El significado computacional de $\neg G$ se define vía la noción de inconsistencia como sigue: $\neg G$ se deduce de (P,N) si y solo si, (P+G,N) es inconsistente (P+G se define como $P \cup \{G\}$). La idea lógica correspondiente es el esquema de reducción al absurdo: si de $P \cup \{G\}$, N se deduce una inconsistencia entonces de P y N se deduce $\neg G$.

A continuación desarrollaremos este método.

V.1 NEGACION POR INCONSISTENCIA, SIN CUANTIFICADORES.

Definimos lo que consideraremos como cláusulas y metas, para la Negación por Inconsistencia.

Una *meta* es definida inductivamente como sigue:

1. Cualquier átomo sin variables es una meta.
2. Si $A_1,\ldots,A_n$ son metas, entonces $A_1 \wedge \ldots \wedge A_n$ y $\neg A_1$ son metas.

Una *cláusula* es definida inductivamente como sigue:

1. Cualquier átomo sin variables es una cláusula.
2. Si A es una meta y q un átomo sin variables entonces $q \leftarrow A$ es una cláusula.

Como se podrá notar el lenguaje utilizado en PROLOG para negación por inconsistencia no es el mismo que el usual, ya que cuando tenemos PROLOG para negación por fracaso no se admiten negaciones anidadas y aquí sí.

Un programa tiene la forma (P,N) con P un conjunto de cláusulas y N un conjunto de metas. El significado intuitivo de (P,N) es que P es la información positiva y N la negativa.

A continuación tenemos las reglas computacionales para una meta G a partir del programa (P,N).

Primero nótese que cualquier meta de la forma $\neg A$ puede ser escrita como $\neg(\wedge q_i \wedge \wedge \neg B_j)$, con q_i átomo.

(P,N)?G = 1, significa que la meta G se deduce de (P,N) como sigue:

(a) Para G atómica, (P,N)?G = 1, si y solo si

(a1) $G \in P$ o

(a2) para alguna cláusula $(G \leftarrow A) \in P$, $(P,N)?A = 1$.

Decimos que la meta G fracasa de una forma finita si $G \notin P$ y para cada $(G \leftarrow A) \in P$, A fracasa de una forma finita.

(b) $(P,N)?G_1 \wedge G_2 \wedge \ldots \wedge G_n = 1$ si y solo si para cada i, $(P,N)?G_i = 1$. Decimos que la meta fracasa de una forma finita si para alguna i, $(P,N)?G_i$ fracasa de una forma finita.

(c) $(P,N)?\neg(\wedge q_i \wedge \wedge \neg B_j) = 1$ si y solo si para alguna $A \in N \cup \{B_j\}$, $(P \cup \{q_i\}, N \cup \{B_j\})?A = 1$. Decimos que la meta original fracasa de una forma finita si la meta presente fracasa de una forma finita.

A continuación tenemos el resultado de correctez para la Negación por Inconsistencia, sin cuantificadores:

Teorema 68. Sea (P,N) un programa y G una meta. Sea $C(P,N)$ la siguiente fórmula : $C(P,N) = \bigwedge_{x \in P} x \wedge \bigwedge_{y \in N} \neg y$, entonces $C(P,N) \models G$ si $(P,N)?G = 1$, es decir, si G tiene éxito, entonces G es consecuencia lógica de los datos (P,N).

La completez para la Negación por Inconsistencia, sin cuantificadores es la siguiente:

Teorema 69. Si $C(P,N) \models G$ entonces $(P,N)?\neg\neg G = 1$.

Nótese que no se toma G sino $\neg\neg G$ para que en la deducción tomemos en cuenta al conjunto N y no solo a P.

V.2 NEGACION POR INCONSISTENCIA CON CUANTIFICADORES.

La definición de cláusulas y metas con cuantificadores es similar a la que tenemos sin cuantificadores excepto porque aquí sí se admiten las variables.

Cuando preguntemos por ¬Ax vamos a suponer que ¬Ax es igual a ∃x¬Ax.

A continuación definiremos lo que es la base de datos durante el cómputo; al principio del cómputo la base de datos es (P,N), pero durante el cómputo se aumentan Q y M.

Una *base de datos* es una cuádrupla (P,Q,N,M) donde P, Q son conjuntos de cláusulas (datos positivos) y N, M son conjuntos de metas (datos negativos). El significado lógico del éxito de una meta G a partir de (P,Q,N,M) es

$$\models \exists\, [(\forall \wedge P) \wedge \wedge Q \wedge (\forall \wedge \neg N) \wedge \wedge \neg M \rightarrow G].$$

El hecho de que una cláusula esté en Q o M indica que todas las substituciones θ realizadas durante el cómputo deben ser compatibles. En la práctica, siempre que llevemos a cabo una substitución θ para igualar cabezas, debemos reemplazar Q por Qθ y M por Mθ, donde Qθ, Mθ son el resultado de aplicar θ a cada variable en cada cláusula de Q y M. Es necesario, entonces, que al principio del cómputo tengamos variables completamente nuevas para la meta, y variables diferentes en P y N.

Definiremos la noción de tener éxito conjuntamente a partir de varias bases de datos, es decir,

$\langle (P_i, Q_i, N_i, M_i)?G_i = 1$ tiene éxito conjuntamente$\rangle$. La palabra conjuntamente significa "bajo la misma substitución θ de las variables de Q_i, M_i". La razón de que nos interese el tener éxito conjuntamente es que la base de datos cambia todo el tiempo, y un simple cómputo en la base puede dividirse en varias preguntas a partir de diferentes bases de datos, pero la substitución θ debe permanecer igual porque debe satisfacer la pregunta original.

Definimos la noción de $\langle (P_i, Q_i, N_i, M_i)?G_i = 1$ *tienen éxito conjuntamente*$\rangle$ dando las reglas de cómputo. Asumimos que las reglas en P_i y N_i se escriben con diferentes variables de las de Q_i y M_i. Las variables de P_i y N_i son todas diferentes. Q_i y M_i pueden tener variables en común.

(a) Reglas de cómputo para átomos:

(a1) Supongamos que para alguna j, G_j es atómica y hay una substitución θ y un átomo $H \in P_j \cup Q_j$ tal que $H\theta = G_j\theta$. Nótese que suponemos que P_j y Q_j tienen diferentes variables. En este caso las metas originales tienen éxito conjuntamente si tenemos lo siguiente:

$\langle (P_i, Q_i\theta, N_i, M_i\theta)?G_i\theta = 1$ tienen éxito conjuntamente para $i \neq j\rangle$.

(a2) Para G_i y θ como antes y para alguna $(H \leftarrow C) \in P_j \cup Q_j$ tenemos $H\theta = G_j\theta$ y $\langle (P_i, Q_i\theta, N_i, M_i\theta)?G_i\theta = 1$, $i \neq j$, y $(P_j, Q_j\theta, N_j, M_j\theta)?C\theta = 1$ tienen éxito

conjuntamente⟩.

(b) Regla de cómputo para conjunciones: Para $G_j = G_{j,1} \wedge \ldots \wedge G_{j,k}$, las metas originales tienen éxito conjuntamente si $\langle (P_i, Q_i, N_i, M_i) ? G_i = 1$ para $i \neq j$ y para $m = 1, \ldots, k$, $\langle (P_j, Q_j, N_j, M_j) ? G_{j,m} = 1$ tienen éxito conjuntamente⟩

(c) Regla de cómputo para negación: Para $G_j = \neg(\bigwedge_{m=1}^{k} H_{j,m} \wedge \bigwedge_{n=1}^{k'} \neg A_{j,n})$ las metas originales tienen éxito conjuntamente si $\langle (P_i, Q_i, N_i, M_i) ? G_i = 1$ para $i \neq j$, y para alguna $B \in M_j \cup N_j \cup \{A_{j,n} \mid n=1, \ldots, k'\}$, tenemos que $\langle (P_j, Q_j \cup \{H_{j,m} \mid m=1 \ldots k\}, N_j, M_j \cup \{A_{j,n} \mid n=1 \ldots k'\}) ? B = 1$ tienen éxito conjuntamente⟩, donde $H_{j,m}$ son átomos y $A_{j,n}$ son metas.

(d) La lista vacía de metas, ⟨⟩, siempre tiene éxito conjuntamente.

El resultado que se tiene acerca de la correctez es el siguiente:

Teorema 70. Si $\langle (P_i, Q_i, N_i, M_i) ? G_i = 1$ tienen éxito conjuntamente⟩ entonces existe una substitución θ tal que para cada i

$$(\forall \wedge P_i) \wedge \wedge Q_i\theta \wedge (\forall \wedge \neg N_i) \wedge \wedge \neg M_i\theta \models G_i\theta.$$

V.3 NEGACION POR FRACASO COMO CASO ESPECIAL DE LA NEGACION POR INCONSISTENCIA.

Para distinguir si la deducción de una meta G a partir de una base de datos P se lleva a cabo por medio de la negación por fracaso o por medio de la negación por inconsistencia, utilizaremos la siguiente notación:

P(?F)G para la negación por fracaso y

P(?I)G para la negación por inconsistencia.

Usaremos la notación P?G = 1 si el cómputo de G a partir de P tiene éxito de una forma finita; y P?G = 0, si el cómputo de G a partir de P fracasa de una forma finita. "?" puede ser (?F) o (?I).

Para poder tener a la negación por fracaso como un caso de la negación por inconsistencia, primero definiremos lo que es un árbol de cómputo con éxito para la negación por fracaso y la negación por inconsistencia.

Definimos la noción de *árbol de cómputo con éxito* para la pregunta P(?F)G = X (X es 0 o 1), como sigue:

(a) Un árbol etiquetado tiene la forma (T,≤,0,V), donde (T,≤,0) es un árbol finito con nodo tope 0 [es decir, ∀t(t≤0)]. V(t) es una función de etiquetado que da valores de la forma [P(t)(?F)G(t) = X(t)], con P(t) una base de datos, G(t) una meta, y X(t) el valor 0 o 1.

(b) Un árbol etiquetado (T,≤,0,V) es un árbol con éxito

para la pregunta P(?F)G = X si y solo si se tienen las siguientes condiciones:

(b1) V(O) = [P(?F)G = X].

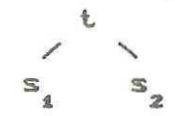
(b2) Sea t cualquier nodo en el árbol tal que

V(t) = [P(t)(?F)G(t) = X(t)], y supongamos que

$G(t) = G_1(t) \wedge G_2(t)$.

Entonces se da lo siguiente:

(b2.1) X(t) = 1. En este caso t tiene exactamente dos puntos inmediatamente precedentes en el árbol, s_1 y s_2, como sigue:



tal que $V(s_i) = [P(t)(?F)G_i(t) = 1]$ para i=1,2.

(b2.2) X(t) = 0. En este caso t tiene un punto inmediatamente precedente en el árbol, como sigue:

y $V(s) = [P(t)(?F)G_i(t) = 0]$ con i, 1 o 2.

(b3) Sea t un nodo tal que

$V(t) = [P(t)(?F)\neg G(t) = X(t)]$.

Entonces t tiene exactamente un punto imediatamente precedente s en el árbol como sigue:

y $V(s) = [P(t)(?F)G(t) = 1 - X(t)]$.

(b4) Supongamos que t es un nodo con

V(t) = [P(t)(?F)q = X(t)], para q átomo.

Entonces se da lo siguiente:

(b4.1) X(t) = 1 y t es un punto final del árbol. En este caso q ∈ P(t).

(b4.2) X(t) = 0 y t es un punto final del árbol. En este caso q no es cabeza de ninguna cláusula de P(t).

(b4.3) X(t) = 0 y t no es un punto final del árbol. Entonces para alguna m ≥ 1, existen exactamente m puntos inmediatamente precedentes s_i de t en el árbol como sigue:

y hay exactamente m cláusulas en P(t) con cabezas q de la forma $q \leftarrow B_i$, $i = 1,\ldots,m$, de tal manera que para $i = 1,\ldots,m$

$V(s_i) = [P(t)(?F)B_i = 0]$.

(b4.4) X(t) = 1 y t no es un punto final del árbol. Entonces t tiene exactamente un punto inmediatamente precedente s en el árbol como sigue

y V(s) = [P(t)(?F)B = 1] para alguna B tal que $q \leftarrow B \in P(t)$.

Definiremos los *árboles de cómputo para la negación por*

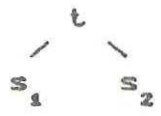
inconsistencia. La situación básica del cómputo es (P,N)(?I)G = X, donde P es un conjunto de cláusulas, N es un conjunto de metas, G es una meta, y X es 1 o 0. Un árbol con éxito para una pregunta de la forma anterior es un árbol etiquetado (T,≤,0,V) donde para cada t V(t) = [(P(t),N(t))(?I)G(t) = X(t)], y se tiene lo siguiente:

(1) V(0) = [(P,N)(?I)G = X].

(2) Sea t un nodo en el árbol tal que

V(t) = [(P(t),N(t))(?I)G(t) = X(t)] y supongamos que $G(t) = G_1(t) \wedge G_2(t)$. Entonces se da lo siguiente:

(2.1) X(t) = 1. En este caso t tiene exactamente dos predecesores inmediatos como sigue:



tal que $V(s_i) = [(P(t),N(t))(?I)G_i(t) = 1]$ para i=1,2.

(2.2) X(t) = 0. En este caso t tiene un punto inmediatamente predecesor s en el árbol como sigue

y $V(s) = [(P(t),N(t))(?I)G_i(t) = 0]$, con i, 1 o 2.

(3) Sea t un nodo tal que

V(t) = [(P(t),N(t))(?I)¬G(t) = X(t)]. Supongamos además que $N(t) = \langle N_1,\ldots,N_m\rangle$ (si m = 0 entonces N(t) es vacío), y $G(t) = \bigwedge_{i=1}^{k'} q_i \wedge \bigwedge_{j=1}^{k} \neg B_j$.

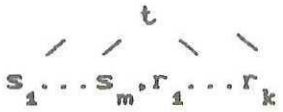
Con q_i átomos y si k' es 0 o k es 0, se entiende que

las q_i o las $\neg B_j$ no aparecen en G(t). Por supuesto asumimos que k' + k > 0. Entonces se da un caso de los siguientes:

(3.1) m + k = 0, es decir, N(t) = vacío, G(t) = Λq_i.

En este caso tenemos X(t) = 0.

(3.2) X(t) = 0. En este caso t tiene exactamente m + k puntos inmediatamente precedentes, como sigue:



y se da lo siguiente:

Sea P = P(t) ∪ {q_i}, N = N(t) ∪ {B_j}, entonces V(s_i) = [(P,N)(?I)N_i = 0], i = 1,...,m

V(r_j) = [(P,N)(?I)B_j = 0], j = 1,...k.

(3.3) X(t) = 1. En este caso t tiene un punto inmediatamente precedente como sigue:

con V(s) = [(P,N)(?I)C = 1], donde P, N son como antes y C ∈ N.

(4) Supongamos que t es un nodo tal que

V(t) = [(P(t),N(t))(?I)q = X(t)], con q átomo.

Entonces se da un caso de los siguientes:

(4.1) X(t) = 1, y t es un punto final del árbol.

Entonces q ∈ P(t).

(4.2) X(t) = 0, y t no es un punto final del árbol.

En este caso q no es cabeza de ninguna cláusula en P(t).

(4.3) X(t) = 0, y t no es un punto final del árbol. Entonces para alguna m ≥ 1, hay exactamente m puntos inmediatamente precedentes s_i a t en el árbol como sigue:

y hay exactamente m cláusulas en P(t) con cabeza q de la forma $q \leftarrow B_i$, i = 1,...,m, tal que para cada i,

$V(s_i)$ = [(P(t),N(t))(?I)B_i = 0].

(4.4) X(t) = 1, y t no es un punto final del árbol. Entonces t tiene un punto inmediatamente precedente s en el árbol como sigue:

y V(s) = [(P(t),N(t))(?I)B = 1] para alguna (q ← B) ∈ P(t).

En base a estas definiciones tenemos los siguientes resultados con respecto a la relación que existe entre la negación por inconsistencia y la negación por fracaso.

Teorema 71. Sea P un conjunto de cláusulas y G una meta sin negaciones anidadas, es decir, de la forma $\wedge a_i \wedge \wedge \neg b_j$ con a_i, b_j átomos; y las cláusulas de la forma $q \leftarrow \wedge c_i \wedge \wedge \neg d_j$ con c_i, d_j, q átomos. Sea N = {q|q átomo y P(?F)q= 0}. Entonces para cualquier G sin negaciones anidadas, se tiene:

P(?F)G = 1 si y solo si (P,N)(?I)G = 1.

Teorema 72. Sea P cualquier base de datos. Sea L = {y|P(?F)y = 0}. Supongamos que P es tal que toda meta

tiene éxito o fracasa. Entonces para toda G:

$$P(?F)G = 1 \quad \text{si y solo si} \quad (P,L)(?I)G = 1.$$

Debido a estos resultados, tenemos a la negación por fracaso como un caso particular de la negación por inconsistencia, de hecho si nuestro sistema de prueba de teoremas es completo, es decir, toda meta tiene éxito o fracasa, entonces la negación por fracaso es realmente la negación clásica ya que es igual a la negación por inconsistencia.

V.4 EJEMPLOS.

A continuación expondremos algunos ejemplos para que se vea más claramente la mecánica de la negación por inconsistencia.

Ejemplo 50. Considérese la base de datos siguiente:

$$A \vee B, \quad R \leftarrow A, \quad R \leftarrow B.$$

Ciertamente R debe deducirse de esta base de datos, esta base de datos no se puede representar en PROLOG con negación por fracaso. Para poder utilizar negación por inconsistencia podríamos definir a P y N como sigue:

$$P = \langle R \leftarrow A,\ R \leftarrow B\rangle \quad y \quad N = \langle \neg A \wedge \neg B\rangle.$$

Entonces si preguntamos R, lo tenemos que deducir directamente de P, lo cual no es posible, si preguntamos por $\neg\neg R$, entonces tenemos:

$$P' = \langle R \leftarrow A,\ R \leftarrow B,\ \neg R\rangle \quad y \quad N = \langle \neg A \wedge \neg B\rangle,$$

con la pregunta ?(¬A ∧ ¬B). No podemos formalmente escribir negaciones de átomos en los datos P, entonces se entiende ¬R ∈ P' como "aumenta R a N". Así, debemos checar las metas ¬A ∧ ¬B, y R de la base de datos (P_2,N_2), con $P_2 = P$, y $N_2 = N \cup \{R\}$. Queremos checar ¬A ∧ ¬B, y R como metas, porque si tiene éxito un elemento de N_2 entonces tenemos inconsistencia. Así la meta original ¬¬R tiene éxito. Si ninguno tiene éxito, entonces ¬¬R fracasa.

(a) La meta R fracasa por la misma razón que antes, ya que no se utiliza N_2.

(b) Para checar la meta ¬A ∧ ¬B, checamos dos cómputos:

(b1) Sumar A a la base de datos P, y checar N_2 como metas.

(b2) Sumar B a la base de datos P, y checar N_2 como metas.

Como los dos casos son simétricos trataremos solo uno. Checaremos $P_{2a} = \{R \leftarrow A, R \leftarrow B, A\}$, $N_2 = \{\neg A \wedge \neg B, R\}$. El sistema no es consistente, porque $R \in N_2$ tiene éxito a partir de P_{2a}.

Así tenemos que ¬¬R tiene éxito a partir de (P,N). Esto significa que aunque no podemos derivar R de la base de datos, podemos derivar ¬¬R.

Ejemplo 51. Ahora veamos un ejemplo con cuantificadores, tomando (P,∅,N,∅) como sigue: $P = \{A(x) \leftarrow \neg(A(a) \wedge \neg A(x))\}$ y $N = \{A(a) \wedge A(b)\}$. La meta será A(y). Q = M es vacío.

Paso 1. Unificar para tener la nueva submeta:

¬(A(a)∧¬A(y)). Esta submeta tiene éxito si la siguiente base de datos es inconsistente: (P',Q',N',M'), con P' = {A(x) ← ¬(A(a)∧¬A(x))}, Q' = {A(a)}, N' = {A(a)∧A(b)}, y M' = {A(y)}.

Paso 2. Para checar la consistencia, tratamos cada elemento de N' y M' como una meta. Ahora supongamos que tratamos A(y) como meta. A(y) se unifica con A(a) ∈ Q', es decir, la meta original tiene éxito con y = a.

V.5 VENTAJAS Y DESVENTAJAS.

(1) La negación por inconsistencia permite (efectivamente) poner hechos negativos y reglas negativas en la base de datos, con un número arbitrario de negaciones anidadas. Esto es una gran ventaja sobre las otras negaciones.

(2) La negación por inconsistencia siempre es lógicamente válida. Esto significa que no depende de las instancias, ni del orden de las cláusulas o de la base de datos. Es más, el significado de la negación por inconsistencia no cambia con el crecimiento de la base de datos, lo que no sucede con la negación por fracaso.

(3) Cuando una meta de la forma ¬G(x) tiene éxito, con ¬ como negación por inconsistencia, el cómputo satisfactorio produce una substitución θ para x tal que ¬G(x)θ tiene éxito.

(4) La negación por inconsistencia contiene a la negación

por fracaso como un caso especial en el caso en que la negación por fracaso funcione bien lógicamente.

(5) Una desventaja de la negación por inconsistencia es que no es eficiente computacionalmente como la negación por fracaso.

Se revisó la bibliografía posterior a la publicación de Gabbay y Sergot[1986] y solamente Shepherdson[1988] la menciona al decir que no va a tratar a la Negación por Inconsistencia en su trabajo.

Gabbay y Sergot[1986] prometen un segundo artículo con respecto a este tema, en donde tratarán la implementación de una parte de la Negación por Inconsistencia, pero hasta ahora no ha aparecido esta publicación.

Nos parece que este método se debería implementar ya que hace lo que ningún otro: aumenta efectivamente información negativa al programa y es compatible con la negación lógica. Además incluye a la Negación por Fracaso.

VI. CONCLUSIONES.

Después de haber hecho un estudio general del problema de la negación en Programación Lógica nos damos cuenta de que lo más importante es que sepamos qué queremos decir con *no*.

Tenemos dos formas de dar su significado a *no*, la primera consiste en dar suposiciones extra-lógicas que permitan deducir conocimientos negados aunque originalmente no sea posible; la segunda es utilizar la lógica de primer orden usual, es decir, deducir el conocimiento que sea consecuencia lógica de nuestro programa, como los programas normales tienen como modelo a la base de Herbrand entonces ningún conocimiento negativo es consecuencia lógica de ellos, por lo tanto, si queremos inferir negaciones de nuestros programas tenemos que ampliar los tipos de fórmulas posibles que se puedan incluir en ellos.

La primera forma de dar significado a la negación incluye a la Suposición del Mundo Cerrado, la Completación de un Programa, la Circunscripción, la Suposición del Mundo Cerrado Generalizada, la Regla de Herbrand y la regla de Negación por Fracaso.

La Suposición del Mundo Cerrado supone que sabemos todo, que no hay ambigüedades, es decir, conocimiento indefinido acerca de átomos sin variables y es tal que no existe un procedimiento de prueba general para ella. Cuando aplicamos la resolución-LSDNF a programas definidos, y tenemos éxito

entonces lo que deducimos es inferido de la Suposición del Mundo Cerrado aplicada al programa en cuestión. Esta suposición extra-lógica pide mucho y siempre que se tenga un programa que no tenga un modelo mínimo, sino solamente modelos minimales, la Suposición del Mundo Cerrado va a ser inconsistente y entonces no nos va a decir mucho ya que todo será válido.

La Completación de un programa está muy relacionada con la forma en que se escribe el programa, depende de la elección de las cabezas, no sólo del significado lógico. Cuando se tienen definiciones de símbolos de predicados mutuamente recursivas no se entiende su significado, ya que la Completación de un programa más bien nos dice que nuestros *si* son *si* de definiciones, es decir, en realidad son *si y solo si* aunque no se escriban como tales y es ilógico escribir definiciones mutuamente recursivas. La Completación de un programa está muy vinculada con la resolución-LSDNF, ya que esta última es correcta con respecto a la Completación aunque no siempre sea completa. Para programas estratificados siempre se va a tener a la Completación consistente ya que aquí la definición de cada símbolo de predicado sólo utiliza negativamente lo que ya está totalmente definido y afirmativamente lo que se está definiendo.

La Circunscripción es una forma más adaptable a las necesidades de una determinada aplicación ya que no exige el conocimiento total de lo que es verdadero en el sistema, si

circunscribimos todos los predicados obtenemos la Suposición del Mundo Cerrado y si tenemos un programa estratificado y hacemos la circunscripción con prioridad en base a la función de nivel entonces obtenemos la Completación. El problema de la Circunscripción es que obtenemos un enunciado de segundo orden. Es más fácil trabajar con los modelos ya que deben de minimizar los símbolos de predicados que queramos y con el orden indicado en el caso de la circunscripción con prioridad.

La Suposición del Mundo Cerrado Generalizada mejora a la Suposición del Mundo Cerrado, ya que no es inconsistente en el caso de no tener modelos mínimos, sólo que no siempre se obtiene respuesta cuando tenemos conflictos entre los modelos minimales.

La regla de la Negación por Fracaso es más computacional que lógica ya que se basa en el fracaso finito, aunque tiene la ventaja de que es muy fácil de implementar y de entender.

Todas estas suposiciones extra-lógicas son razonamientos no-monótonos, lo que complica su estudio.

En el segundo aspecto, es decir, cuando aumentamos tipos de fórmulas dentro del programa provocando que al utilizar la lógica de primer orden usual obtengamos conocimiento negado, se encuentra la Negación por Inconsistencia en donde se admiten fórmulas con negaciones anidadas además de tener un conjunto de metas que no son válidas, es decir, cláusulas que son disyunciones de literales negativas. Hay que tener claro que hay una gran diferencia entre esta negación y las

anteriores, ya que aquí no estoy agregando conocimientos y en las anteriores sí. En esta forma el razonamiento es monótono.

Dentro de todas estas formas de entender la negación, la que se ha implementado es la regla de la Negación por Fracaso por medio de la resolución-LSDNF y se ha planteado la implementación de la Negación por Inconsistencia.

En la resolución-LSDNF el orden de los predicados en metas y procedimientos influye en la eficiencia pero no tiene significado lógico, además sólo la literal distinguida de cada cláusula, la cabeza, es un candidato de unificación con la submeta, aquí se ve muy claro que la forma de escribir el programa influye en la regla de la Negación por Fracaso. La resolución-LSDNF y en general la regla de Negación por Fracaso tiene la desventaja de no dar respuestas a las literales negativas, ya que solamente admite literales negativas sin variables checando si tienen éxito o fracasan en el sistema, otra desventaja es la de que sólo se elije un árbol de deducción y entonces el sistema no es completo. Su gran ventaja es que es muy fácil de implementar y es lo que se utiliza en PROLOG.

La Negación por Inconsistencia sí da valores a variables de literales negativas, aunque su implementación es más complicada y costosa. Incluye a la Negación por Fracaso.

El rumbo que ha tomado la investigación en este campo está muy relacionado con la Negación por Fracaso ya que se ha

estudiado su significado lógico, se ha tratado de mejorar dando respuestas a literales negativas con variables, se ha buscado la manera de tener la condición de seguridad, es decir, que no se evalúen literales negativas con variables, se han investigado formas de cortar los loops que pudieran dar un procedimiento de evaluación de preguntas más completo y con un significado lógico más claro, se han estudiado los programas estratificados que son los programas para los que la Completación es consistente.

En resumen, hay que estar conscientes de lo que se está diciendo al escribir un programa lógico y asegurarnos que el sistema funciona de acuerdo a nuestra concepción del problema que se está resolviéndo. No es que un tipo de négación sea mejor que otro, sino que cada uno se adecúa a diferentes necesidades y lo que es más importante de entender es que la negación en Programación Lógica por lo general aumenta significado al programa lógico.

LITERATURA CITADA

Apt K.R., H.A. Blair y A. Walker, 1988. *Towards a Theory of Declarative Knowledge*, en Foundations of Deductive Databases and Logic Programming, J. Minker (ed.), Morgan Kaufmann Publishers, Inc., Los Altos CA, p. 89-148.

Apt K.R. y M.H. Van Emden, 1982. *Contributions to the Theory of Logic Programming*, J. ACM, Vol. 29, No. 3, p. 841-862.

Besnard P., Y. Moinard y R.E. Mercer, 1989. *The importance of Open and Recursive Circumscription*, Artificial Intelligence 39, p. 251-262.

Clark K.L., 1978. *Negation as Failure*, en Logic and Data Bases, H. Gallaire y J. Minker (eds.), Plenum Press, New York N.Y., p. 293-322.

Dahl V., 1980. *Two Solutions for the Negation Problem*, Logic Programming Workshop, Julio, p. 61-72.

Gabbay D.M. y M.J. Sergot, 1986. *Negation as Inconsistency*, J. Logic Programming, Vol. 3, No. 1, p. 1-35.

Gelfond M., H. Przymusinska y T. Przymusinski, 1989. *On the Relationship between Circumscription and Negation as Failure*, Artificial Intelligence 38, p. 75-94.

Jaffar J., J-L Lassez y J. Lloyd, 1983. *Completeness of the Negation as Failure Rule*, IJCAI-83, Karlsruhe, p. 500-506.

Kowalski R.A., 1979. *Algorithm = Logic + Control*, Comm. ACM, Vol. 22, No. 7, p. 424-436.

Kunen K., 1987a. *Answer Sets and Negation as Failure*, Logic Programming, Proceedings of the Fourth International Conference, J. Minker (ed.), p. 219-228.

Kunen K., 1987b. *Negation in Logic Programming*, J. Logic Programming, Vol. 4, No. 4, p. 289-308.

Lifschitz V., 1985a. *Computing Circumscription*, Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles CA, p. 121-127.

Lifschitz V., 1985b. *Closed-World Databases and Circumscription*, Artificial Intelligence 27, p. 229-235.

Lloyd, J.W., 1987. *Foundations of Logic Programming*, 2a. ed., Springer-Verlag, Alemania.

McCarthy J., 1980. *Circumscription - A Form of Non-Monotonic Reasoning*, Artificial Intelligence 13, p. 27-39.

Minker J. y D. Perlis, 1985. *Computing Protected Circumscription*, J. Logic Programming, Vol. 2, No. 4, p. 235-249.

Przymusinski T.C., 1988. *On the Declarative Semantics of Deductive Databases and Logic Programs*, en Foundations of Deductive Databases and Logic Programming, J. Minker (ed.), Morgan Kaufmann Publishers, Inc., Los Altos CA, p. 193-216.

Reiter R., 1978. *On Closed World Data Bases*, en Logic and Data Bases, H. Gallaire y J. Minker (eds.), Plenum Press, New York N.Y., p. 293-322.

Robinson J.A., 1965. *A Machine-oriented Logic Based on the Resolution Principle*, J. ACM, Vol. 12, No. 1, p. 23-41.

Shepherdson J.C., 1984. *Negation as Failure: A comparison of Clark's Completed Data Base and Reiter's Closed World Assumption*, J. Logic Programming, Vol. 1, No. 1, p. 51-79.

Shepherdson J.C., 1985. *Negation as Failure II*, J. Logic Programming, Vol. 2, No. 3, p. 185-202.

Shepherdson J.C., 1988. *Negation in Logic Programming*, en Foundations of Deductive Databases and Logic Programming, J. Minker (ed.), Morgan Kaufmann Publishers, Inc., Los Altos CA, p. 19-88.

Tarski A., 1955. *A Lattice-theoretical Fixpoint Theorem and its Applications*, Pacific J. Math. 5, p. 285-309.

Van Gelder A., 1988. *Negation as Failure Using Tight Derivations for General Logic Programs*, en Foundations of Deductive Databases and Logic Programming, J. Minker (ed.), Morgan Kaufmann Publishers, Inc., Los Altos CA, p. 149-176.