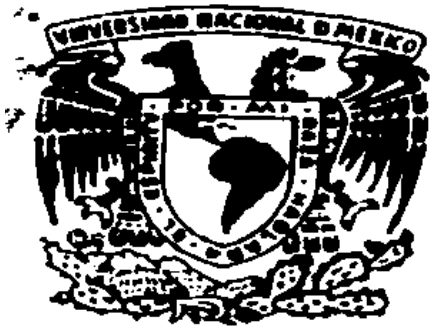

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

U.A.C.P. Y P. DEL C.C.H.
I. I. M. A. S.

DEDUCCION PARCIAL POR FACTORIZACION

T E S I S

QUE PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

PAULO MAXIMO GUTIERREZ GONZALEZ

DIRECTOR DE TESIS: DR. DAVID ROSENBLUETH

JUNIO 1997

Deducción Parcial por Factorización

Paulo Máximo Gutiérrez Gonzáles

Junio de 1997

Resumen

Un programa lógico de búsqueda exhaustiva provocará cálculos redundantes al tener que recorrer un espacio de búsqueda en su totalidad. Una subclase muy importante de los programas lógicos de búsqueda exhaustiva, y que en la actualidad es materia de investigación, la constituyen los programas lógicos que arrojan como una sola todas las respuestas a una meta de otro programa lógico.

En otro orden de ideas, en el área de la programación funcional y lógica, la Transformación de Programas o Deducción Parcial es una técnica de manipulación simbólica, utilizada para producir programas eficientes dado que se conoce parte de la entrada de un programa. Con la entrada conocida se efectua una evaluación parcial del programa original que conduce a un programa residual al menos tan eficiente como el original. De acuerdo al área de estudio de la Transformación de Programas, ésta debiera ocuparse de eliminar los cálculos redundantes que se presentan en un programa de búsqueda exhaustiva. Desafortunadamente, no existen en Transformación de Programas operaciones que combinadas realicen tal tarea.

Sin embargo, un artículo no publicado de Robinson describe, en combinación con Resolución, la regla de inferencia Factorización, y cuya referencia se omite en el artículo publicado, aunque tal artículo realmente combina Resolución y Factorización y las refiere unicamente como Resolución. El problema de la Factorización consiste en unificar dos átomos A_i y A_j, de ser éstos unificables, y remplazarlos por la resolvente.

Adicionalmente a su gran utilidad como regla de inferencia, Factorización puede ser utilizada como una operación más en Transformación de Programas, en tanto que puede ser utilizada para eliminar cálculos redundantes. Tales cálculos redundantes, una vez reunidos en el lugar indicado, son reducidos a uno solo factorizándolos.

El reunir varios cálculos redundantes significa manipular un predicado, con operaciones convencionales de Transformación de Programas, lo suficiente para obtener una cláusula cuyo cuerpo tenga dos o más átomos unificables, para a continuación factorizarlos.

El presente trabajo propone factorización como una operación más en Transformación de Programas y muestra su utilidad al conseguir programas más eficientes que calculan todas las respuestas.

Contenido

1 Introducción 2

2 Programación Lógica 5
2.1 Programas Lógicos 6
2.1.1 Forma Clausal de la Lógica 7
2.2 Modelos de Programas Lógicos 8
2.3 Semánticas Respuesta 13
2.4 Algoritmo de Unificación 14
2.5 Teorema de Unificación 15
2.6 Corrección de Resolución SLD 16
2.7 Completitud de Resolución SLD 19
2.8 Independencia de la Regla de Computación 20
2.9 Procedimientos de Refutación SLD 22

3 Factorización 26
3.1 Transformación de Programas 26
3.2 Factorización como parte de Transformación de Programas 28

4 Combinación de Factorización y el método de Tamaki 35
4.1 Método de Búsqueda Exhaustiva de Tamaki 35
4.2 Al adicionar Factorización al método de Tamaki se alcanza mayor eficiencia 44

5 Conclusiones 49

6 Trabajo a Futuro 50

7 Apéndice 51

1 Introducción

Pay Hayes [5] propone que computación es deducción controlada, por lo que un algoritmo se puede concebir como consistente de un componente lógico, que especifica el conocimiento a utilizarse en la resolución de problemas, y un componente de control, que determina las estrategias por las cuales dicho conocimiento es utilizado en la resolución de problemas [11].

El componente lógico de un algoritmo determina el significado de éste, mientras que el componente de control afecta el desempeño del algoritmo. Muchos algoritmos pueden mejorar su eficiencia mejorando su componente de control sin cambiar su lógica.

El conocimiento implicado en el componente lógico de un algoritmo puede ser utilizado para inferir aseveraciones o para deducir hechos. De esta forma se conciben los conceptos de procesamiento de abajo hacia arriba (bottom-up) para inferencia, y de procesamiento de arriba a abajo (top-down) para deducción, que son dos formas de implementar el control de algoritmos.

Si por otra parte, dado un algoritmo se conoce parte de su entrada, entonces existe un incremento en el conocimiento, por lo que se pueden deducir más hechos a partir de tal nuevo conocimiento. Es notable que este incremento en el conocimiento redunde en mayor eficiencia del componente de control del algoritmo.

En el ámbito de la programación lógica existe el área conocida como deducción parcial que se ocupa de manipular simbólicamente un programa lógico para obtener otro programa lógico más eficiente.

Una forma de obtener un programa lógico más eficiente a partir de uno inicial o fuente es eliminar computaciones *redundantes*. Sin embargo, aunque existen operaciones diversas en deducción parcial, ninguna de ellas elimina computaciones redundantes. Pero es posible, mediante la aplicación sucesiva de la operación desdoblar (unfold), obtener programas que hagan explícitas algunas computaciones redundantes.

En otro orden de ideas, un artículo no publicado de Robinson [9] expone la combinación de dos reglas de inferencia: *Resolución* y *Factorización*, combinación que es equivalente a la versión publicada [15] y que se conoce simplemente como *Resolución*.

De hecho, *para que el ámbito de aplicación de Resolución sea total (es decir, para que resolución se aplique a conjuntos de cláusulas arbitrarias), le es imprescindible Factorización*.

Aunque Factorización juega un papel muy importante conjuntamente con Resolución, no es su único ámbito de aplicación (un ejemplo sencillo lo constituye "la paradoja del barbero", ver [9] página 150).

Como se mencionó anteriormente, en algunos programas lógicos se pueden detectar, mediante aplicaciones sucesivas de la operación deshidratar, computaciones redundantes. Sin embargo, una vez evidenciadas tales redundancias, no existe ninguna operación en deducción parcial que las pueda eliminar.

Debido a que las computaciones redundantes quedan evidenciadas en la forma de dos o más átomos unificables en la misma cláusula, estos átomos quedan factorizados al unificarlos.

La Factorización por unificación encuentra natural aplicación en los programas lógicos con dos o más llamadas recursivas, los cuales se pueden volver completamente ineficientes al provocar, en distintos procesos de una corrida, computaciones idénticas correspondientes a idénticos niveles de recursión.

Otro campo de aplicación de la Factorización por unificación lo constituyen los programas lógicos que recorren un espacio de búsqueda complejo. En esta clase de programas se encuentran los programas lógicos que arrojan, como una sola, todas las respuestas a una meta de otro programa lógico[1].

Inherentemente, un programa lógico que calcula todas las respuestas a una meta lleva consigo computaciones redundantes al tener que recorrer todo un espacio de búsqueda, por lo que es posible hacerlo más eficiente.

EL OBJETIVO DEL PRESENTE TRABAJO DE TESIS ES PROPONER *FACTORIZACION* COMO *UNA OPERACION MAS EN DEDUCCION PARCIAL* MOSTRANDO SU ESPECIAL UTILIDAD AL CONSEGUIR PROGRAMAS MAS EFICIENTES QUE CALCULAN TODAS LAS RESPUESTAS POR LOS METODOS ESTABLECIDOS.

Aunque se da el método general, su sistematización es en su propio derecho materia de investigación aparte.

Debido a que este trabajo trata de ser autocontenido, el segundo capítulo expone los fundamentos y conceptos básicos de la programación lógica, el capítulo tres explica en qué consiste la regla de inferencia Factorización, algunas reglas de transformación de programas, y como es que Factorización se puede concebir

[1] Existen en la literatura métodos para transformar un programa lógico que arroja una respuesta a la vez, dada una meta, a otro programa lógico que arroja como una sola todas las respuestas a tal meta [1, 16, 18].

como una regla más en transformación de programas. Por último, el capítulo cuatro explica como se puede combinar Factorización con el método de Tamaki para obtener programas de búsqueda exhaustiva más eficientes.

2 Programación Lógica

La idea de que la lógica de primer orden, o al menos subconjuntos sustanciales de ella, puede ser utilizada como un lenguaje de programación fue revolucionaria ya que, hasta 1972, la lógica fue únicamente utilizada como un lenguaje de especificación o declarativo en las ciencias de la computación. Sin embargo, [16] mostró que la lógica tiene una interpretación procedural, lo que la hace muy efectiva como un lenguaje de programación, concibiéndose así la programación lógica (y al implementarse en 1972 el primer intérprete en ALGOL-W, se dio origen al acrónimo PROLOG de PROgramación en LOGica).

El lenguaje Prolog está basado en el subconjunto de la lógica referido como cláusulas de Horn. La mayoría de los sistemas de programación lógica disponibles actualmente son intérpretes o compiladores Prolog que utilizan una regla de computación simple, consistente en seleccionar siempre el átomo más a la izquierda en una meta (por tal característica, son conocidos como sistemas Prolog estándares), y basados en resolución.

Sin embargo, es necesario hallar reglas de computación más apropiadas, formas de programar en subconjuntos más grandes de la lógica (no únicamente el subconjunto clausal), y elaborar sistemas que no necesariamente estén basados en resolución.

Kowalski [9, 11], estableció que un algoritmo consta de dos componentes distintos, la lógica y el control:

1. *La lógica es la declaración de qué es el problema a resolver.*
2. *El control es la declaración de cómo ha de ser resuelto.*

El ideal de la programación lógica es que el programador solo tenga que especificar el componente lógico, y el control sea realizado únicamente por el sistema de programación lógica.

Estos componentes originan, por una parte, la semántica declarativa (componente lógico) y, por otra parte, la semántica procedural (control) de los programas lógicos.

Las interpretaciones y modelos de programas lógicos constituyen la semántica declarativa, y el concepto de sustitución respuesta correcta da significado a tal semántica declarativa en términos de la salida deseada a partir de un programa y una meta.

La contraparte procedural de sustitución respuesta correcta es la sustitución respuesta calculada, definida utilizando resolución SLD. Cada sustitución res-

puesta correcta es una instancia de una sustitución respuesta calculada, siendo esta última también correcta, por lo que la resolución SLD es correcta y completa.

Es un hecho notable que la completez es independiente de la regla de computación (que en los sistemas Prolog estándares consiste en siempre seleccionar el átomo más a la izquierda en una meta).

El material de este capítulo está basado en [14].

2.1 Programas Lógicos

Un *programa lógico* es un caso particular de una teoría de primer orden, por lo que se introducirá la sintaxis de las fórmulas bien formadas de esta última.

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.

El *lenguaje de primer orden* consiste de fórmulas bien formadas de la teoría.

Los *axiomas* son un subconjunto especificado de fórmulas bien formadas. Los axiomas y las reglas de inferencia se utilizan para derivar los teoremas de la teoría.

Un *alfabeto* consiste de siete clases de símbolos: variables, constantes, funciones, predicados, conectivos, cuantificadores, y símbolos de puntuación.

Los conectivos, cuantificadores, y símbolos de puntuación, son los mismos para cada alfabeto, y puede no haber constantes ni funciones en un alfabeto cualesquiera.

Las variables se denotan por las letras u, v, w, x, y, z, las constantes por a, b, c, las funciones de aridad mayor que cero normalmente por f, g, h, y los predicados de aridad mayor o igual a cero son normalmente denotados por p, q, y r (en todos los casos posiblemente con subíndices).

Los conectivos son $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, y los cuantificadores, $\exists$ y $\forall$. Los símbolos de puntuación son (,) y ,.

$\neg$, $\exists$, y $\forall$ tienen la misma y más alta precedencia, siguiéndoles $\vee$, a continuación del cual está $\wedge$, y por último, $\rightarrow$ y $\leftrightarrow$, con la misma y más baja precedencia.

El lenguaje de primer orden dado por un alfabeto, consiste del conjunto de todas las fórmulas construidas a partir de los símbolos del alfabeto, en donde:

Una *fórmula* (bien formada) es una fórmula atómica o átoma, y si F y G son fórmulas, también lo son $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$, $(\exists x F)$, y $(\forall x F)$, con x variable.

Si p es un predicado de aridad n y $t_1, \ldots, t_n$ son términos, $p(t_1, \ldots, t_n)$ es un *átomo*.

Un *término* es una variable, una constante, o una función de aridad mayor a cero con argumentos que son términos.

El *ámbito* de $\forall x$ $(\exists x)$ en $\forall x F$ $(\exists x F)$ es F.

Una ocurrencia de una variable en una fórmula se dice estado si sigue inmediatamente a un cuantificador o ocurre en el ámbito de un cuantificador que tiene la misma variable inmediatamente después de él. Cualquier otra ocurrencia de la variable es *libre*.

Una *fórmula cerrada* es una fórmula sin ocurrencias libres de ninguna variable.

Si se adiciona un cuantificador universal por cada variable que ocurre libre en una fórmula F, se obtiene la *cerradura* $\forall(F)$ *universal* de F. Similarmente, $\exists(F)$ denota la *cerradura existencial* de F, obtenida al añadir a la fórmula un cuantificador existencial por cada variable que ocurre libre en F.

2.1.1 Forma Clausal de la Lógica

Una *cláusula* es una fórmula de la forma $\forall x_1 \ldots \forall x_s(L_1 \vee \ldots \vee L_m)$, donde $x_1, \ldots, x_s$ son todas las variables que ocurren en $L_1 \vee \ldots \vee L_m$, y cada L_i es una literal. Una *literal* es un átomo (literal positiva) o la negación de un átomo (literal negativa).

En programación lógica, la cláusula $\forall x_1 \ldots \forall x_s(A_1 \vee \ldots \vee A_k \vee \neg B_1 \vee \ldots \vee \neg B_n)$ se denota por $A_1, \ldots, A_k \leftarrow B_1, \ldots, B_n$. Así, en *notación clausal*, se asume que todas las variables están universalmente cuantificadas, que las comas en el antecedente $B_1, \ldots, B_n$ denotan conjunciones, y que las comas en el consecuente $A_1, \ldots, A_k$ denotan disyunción, ya que, $\forall x_1 \ldots \forall x_s(A_1 \vee \ldots \vee A_k \vee \neg B_1 \vee \ldots \vee \neg B_n)$ es 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* contiene exactamente una literal positiva llamada la cabeza. Así, en $A \leftarrow B_1, \ldots, B_n$, A es la cabeza y $B_1, \ldots, B_n$ es el cuerpo de

la cláusula. Una cláusula unitaria es una cláusula de programa $A \leftarrow$ con cuerpo vacío.

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

En un programa lógico, el conjunto de todas las cláusulas con el mismo símbolo de predicado p en la cabeza, es referido como *la definición de* p.

Una meta es una cláusula de la forma $\leftarrow B_1, \ldots, B_n$, es decir, una cláusula con consecuente vacío, cada B_i $(i = 1, \ldots, n)$ es una *submeta* de la meta.

Si $y_1, \ldots, y_r$ son las variables de la meta $\leftarrow B_1, \ldots, B_n$, la notación clausal es una abreviación de $\forall y_1 \ldots \forall y_r(\neg B_1 \vee \ldots \vee \neg B_n)$ o, equivalentemente, $\neg \exists y_1 \ldots \exists y_r(B_1 \wedge \ldots \wedge B_n)$.

La *cláusula vacía* $\square$, es la cláusula con consecuente y antecedente vacíos. Tal cláusula es entendida como una *contradicción* o *refutación*.

Una *cláusula de Horn* es una cláusula de programa o una meta.

2.2 Modelos de Programas Lógicos

La semántica declarativa de un programa está dada por la semántica de fórmulas usual en lógica de primer orden.

Con objeto de poder discutir la veracidad o falsedad de una fórmula es necesario dar algun significado a cada símbolo en la fórmula. Los cuantificadores y conectivos tienen un significado fijo, pero el significado dado a las constantes, funciones, y predicados, puede variar.

Una *interpretación* consiste de algun dominio de discurso sobre el que las variables fluctuan, la asignación de cada constante a un elemento del dominio, la asignación de cada función a un mapeo en el dominio, y la asignación de cada predicado a una relacion en el dominio. Cada interpretación especifica de esta forma un significado para cada símbolo en la fórmula. Una interpretación en la que la fórmula expresa una oración cierta es referida como un *modelo de la fórmula*. Normalmente, la *interpretación pretendida* es la que da el significado principal de los símbolos y naturalmente se busca que sea un modelo.

Los *teoremas de una teoría* son las fórmulas que son *consecuencia lógica de los axiomas de la teoría*, es decir, aquellas que son ciertas en cada interpretación que es modelo de cada axioma de la teoría. En particular, cada teorema es cierto en la interpretación pretendida de la teoría.

Los sistemas de programación lógica usuales utilizan como única regla de inferencia la regla de resolución.

Supongamos que se desea demostrar que la fórmula $\exists y_1 \ldots \exists y_n (B_1 \wedge \ldots \wedge B_m)$ es una consecuencia lógica de un programa P. Los demostradores de teoremas por resolución lo hacen por refutación. Es decir, la negación de la fórmula a demostrar se añadiría a los axiomas y se deriva una contradicción. Al negar la fórmula a demostrar se obtiene la meta $\leftarrow B_1, \ldots, B_m$ y, manipulando tal meta de arriba hacia abajo, el sistema deriva metas sucesivas. Si eventualmente se deriva la cláusula vacía, se ha llegado a una contradicción y $\exists y_1 \ldots \exists y_n (B_1 \wedge \ldots \wedge B_m)$ constituye una consecuencia lógica de P.

Desde el punto de vista de demostradores de teoremas, el deseo interesa es demostrar consecuencias lógicas. Sin embargo, desde el punto de vista de programación interesan mucho más las asignaciones hechas a las variables $y_1, \ldots, y_n$, ya que constituyen la salida de la carrida del programa.

Una *interpretación de un lenguaje L de primer orden* consiste de:

1. Un conjunto D no vacío, llamado el *dominio de la interpretación*.
2. Para cada constante en L, la asignación de un elemento en D.
3. Para cada función de aridad n en L, la asignación de un mapeo de D^n a D.
4. Para cada predicado de aridad n en L, la asignación de un mapeo de D^n en {cierto, falso } o, equivalentemente, una relación en D^n.

Sea I una interpretación de un lenguaje L de primer orden. Una *asignación de variables, con respecto a I*, es una asignación a cada variable en L de un elemento en el dominio de I.

Sea I una interpretación de un lenguaje L de primer orden con dominio D, y sea A una asignación de variables. Una *asignación, con respecto a I y A*, de los términos en L se define como:

1. Cada variable es asignada de acuerdo a A.
2. Cada constante es asignada de acuerdo a I.
3. Si $t'_1, \ldots, t'_n$ son las asignaciones de términos de $t_1, \ldots, t_n$ y f' es la asignación de f, $f'(t'_1, \ldots, t'_n) \in D$ es la asignación de término de $f(t_1, \ldots, t_n)$.

Sea I una interpretación de un lenguaje L de primer orden con dominio D, y sea A una asignación de variables. A una fórmula en L se le puede dar un valor de verdad falso o cierto, con respecto a I y A, de acuerdo a:

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

2. Si la fórmula tiene la forma $\neg F, F \wedge G, F \vee G, F \rightarrow G$, o $F \leftrightarrow G$, su valor de verdad está dado por la tabla 1.

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
cierto	*cierto*	*falso*	*cierto*	*cierto*	*cierto*	*cierto*
cierto	*falso*	*falso*	*falso*	*cierto*	*falso*	*falso*
falso	*cierto*	*cierto*	*falso*	*cierto*	*cierto*	*falso*
falso	*falso*	*cierto*	*falso*	*falso*	*cierto*	*cierto*

Tabla 1: Tabla de verdad

3. Si la fórmula tiene la forma $\exists x F$, su valor de verdad es cierto si *existe* d en D tal que F tiene valor de verdad cierto, con respecto a I y $A(x/d)$, donde $A(x/d)$ es A excepto que a x le es asignado d; de lo contrario, su valor de verdad es falso.

4. Si la fórmula tiene la forma $\forall x F$, su valor de verdad es cierto si, para todo d en D, F tiene valor de verdad cierto, con respecto a I y $A(x/d)$: de lo contrario, su valor de verdad es falso.

El valor de verdad de una fórmula cerrada no depende de la asignación de variables. Consecuentemente, se puede hablar sin ambigüedad del valor de verdad de una fórmula cerrada con respecto a una interpretación I, y si tal valor de verdad es cierto, se dice que I es un *modelo* de la fórmula. Además, si S es un conjunto de fórmulas cerradas de un lenguaje L de primer orden e I una interpretación de L, se dice que I es un modelo para S si I es un modelo para cada fórmula de S. En particular, si $S = (F_1, \ldots, F_n)$ es un conjunto finito de fórmulas cerradas, I es un modelo para S si y solo si I es un modelo para $F_1 \wedge \ldots \wedge F_n$.

Los *axiomas* de una teoría de primer orden conforman un subconjunto de fórmulas cerradas en el lenguaje de la teoría. En programación lógica, los axiomas de una teoría de primer orden son las cláusulas de un programa.

Sea T una teoría de primer orden y L el lenguaje de T. Un *modelo para* T es una interpretación de L que es modelo para cada axioma de T.

Sea S un conjunto de fórmulas cerradas de un lenguaje L de primer orden. Se dice que S es *satisfactible* si L tiene una interpretación que es modelo para S.

S es **válido** si cada interpretación de *L* es un modelo para *S*. *S* es **no satisfactible** si no tiene modelos.

Sea *S* un conjunto de fórmulas cerradas, y *F* una fórmula cerrada de un lenguaje *L* de primer orden. Se dice que *F* es una **consecuencia lógica** de *S* si, para cada interpretación *I* de *L*, ser modelo *I* de *S* implica que *I* es modelo de *F*.

Notar que si $S = \{F_1, \ldots, F_n\}$ es un conjunto finito de fórmulas cerradas, *F* es consecuencia lógica de *S* si y solo si $F_1 \wedge \ldots \wedge F_n \rightarrow F$ es válida.

PROPOSICION 2.2.1 *Sea S un conjunto de fórmulas cerradas y F una fórmula cerrada de un lenguaje L de primer orden. F es consecuencia lógica de S si y solo si* $S \cup \{\neg F\}$ *no es satisfactible.*

Cuando se provee una meta *G* a un sistema Prolog, con un programa *P* cargado, se le esta pidiendo al sistema que demuestre que el conjunto de cláusulas $P \cup \{G\}$ no es satisfactible. Si *G* es la meta $\leftarrow B_1, \ldots, B_n$ con variables $y_1, \ldots, y_r$, la proposición anterior establece que demostrar que $P \cup \{G\}$ no es satisfactible es exactamente lo mismo que demostrar que $\exists y_1 \ldots \exists y_r (B_1 \wedge \ldots \wedge B_n)$ es consecuencia lógica de *P*.

Así, el problema básico es determinar la no satisfactibilidad o la satisfactibilidad de $P \cup \{G\}$, donde *P* es un programa y *G* es una meta. De acuerdo a las definiciones dato implica demostrar que cada interpretación de $P \cup \{G\}$ no es modelo. Evidentemente lo anterior es un gran problema. Sin embargo, existe una clase mucho más pequeña y conveniente de interpretaciones que las todas las que se necesita investigar para demostrar la no satisfactibilidad. Tales interpretaciones son conocidas como *interpretaciones de Herbrand*.

Un *término aterrizado* es un término sin variables, y un *átomo aterrizado* es un átomo sin variables.

Sea *L* un lenguaje de primer orden. El *universo* U_L *de Herbrand para L* es el conjunto de todos los términos aterrizados que pueden ser formados con las constantes y funciones en *L* (si *L* no tiene constantes se adiciona alguna para formar los terminos aterrizados).

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

Sea un lenguaje *L* de primer orden. Una interpretación para *L* es una *interpretación de Herbrand* si:

1. El dominio de la interpretación es el universo U_L de Herbrand.
2. Las constante en L se asignan a ellas mismas en U_L.
3. Si f es una función en L de aridad n, f se asigna al mapeo U_L^n en U_L dado por $(t_1, \ldots, t_n) \rightarrow f(t_1, \ldots, t_n)$.

Ya que no existe restricción en la asignación de los predicados en L, asignaciones distintas corresponden a distintas interpretaciones de Herbrand.

Ya que para todas las interpretaciones de Herbrand la asignación de constantes y funciones es fija, es posible asociar una interpretación de Herbrand con un subconjunto de la base de Herbrand. Tal subconjunto es el conjunto de todos los átomos atérrizados que son ciertos con respecto a la interpretación. Inversamente, dado un subconjunto arbitrario de la base de Herbrand, existe una interpretación de Herbrand correspondiente, definida al especificar qué predicados mapean a cierto precisamente cuando el predicado aplicado a sus argumentos está en el subconjunto dado.

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

Abreviando del lenguaje, muchas veces es conveniente referirse a una interpretación de un conjunto S de fórmulas, en lugar del lenguaje de primer orden del que provienen las fórmulas. Normalmente se asumirá que el lenguaje de primer orden está definido por las constantes, funciones y predicados que aparecen en S. Con tal suposición, se puede hacer referencia al universo U_S de Herbrand y a la base B_S de Herbrand de S y también referirse a las interpretaciones de Herbrand de S como subconjuntos de la base de Herbrand de S. En particular, el conjunto de fórmulas será muchas veces un programa P, de modo que se puede hacer referencia al universo U_P y a la base B_P de Herbrand de P.

PROPOSICION 2.2.2 *Sea S un conjunto de cláusulas que tiene un modelo. Entonces S tiene un modelo de Herbrand.*

PROPOSICION 2.2.3 *Sea S un conjunto de cláusulas. S es no satisfactible si y solo si no tiene modelos de Herbrand.*

Las dos proposiciones anteriores muestran que para demostrar la no satisfactibilidad de un conjunto de cláusulas es suficiente considerar solo interpretaciones de Herbrand.

2.3 Sustituciones Respuesta

Una *sustitución* θ es un conjunto finito $\{v_1/t_1, \ldots, v_n/t_n\}$, 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. Cada v_i/t_i es referido como una asignación para v_i. θ se conoce como una *sustitución aterrizada* si los t_i son todos términos aterrizados. θ es conocida como una *sustitución pura de variables* si los t_i son todos variables.

Si θ es vacía, es referida como la *sustitución identidad* ε.

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 sustitución y E una expresión. La *instancia* $E\theta$ de E por θ, es la expresión obtenida de E al reemplazar simultáneamente cada ocurrencia de la variable v_i en E por el término t_i $(i = 1, \ldots, n)$. $E\theta$ es llamada una *instancia aterrizada* de E si $E\theta$ es aterrizada.

Si $S = \{E_1, \ldots, E_n\}$ es un conjunto finito de expresiones y θ una sustitución, $S\theta$ denota el conjunto $\{E_1\theta, \ldots, E_n\theta\}$. Si $\theta = \varepsilon$, $E\theta = E\varepsilon = E$ para toda expresión E.

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

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

eliminando cualquier asignación $u_i/s_i\sigma$ para la que $u_i = s_i\sigma$ y eliminando cualquier asignación v_j/t_j para la que v_j *esté en* $\{u_1, \ldots, u_m\}$.

PROPOSICION 2.3.1 *Sean* θ, σ *y* γ *sustituciones.*

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

Sean E y F expresiones. Se dice que E y F son *variantes* si existen sustituciones θ y σ tales que $E = F\theta$ y $F = E\sigma$. Se dice que E es una *variante de* F o que F es una *variante de* E.

Sea E una expresión y V el conjunto de variables que ocurren en E. Una *sustitución renombre* para E es una sustitución pura de 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 $(V \setminus \{x_1, \ldots, x_n\}) \cap \{y_1, \ldots, y_n\} = \emptyset$.

PROPOSICION 2.3.2 *Sean E y F expresiones variantes. Existen sustituciones θ y σ tales que $E = F\theta$ y $F = E\sigma$, donde θ es una sustitución renombre para F y σ es una sustitución renombre para E.*

El interes se centrará en sustituciones que hacen a cada expresión de un conjunto de expresiones sintacticamente identica a las demás en el conjunto. Se dice que una de tales sustituciones unifica al conjunto de expresiones.

El concepto de unificación se remonta a Herbrand en 1930, y es redescubierto en 1960 por Prawitz, para posteriormente ser explotado por Robinson para su utilización en la regla de resolución.

Sea S un conjunto finito de expresiones simples. Una sustitución θ se dice un *unificador* para S si $S\theta$ se reduce a una expresión simple. Un unificador θ para S es un *unificador más general* (umg) para S, si para cada unificador σ de S, existe una sustitución γ tal que $\sigma = \theta\gamma$.

De la definición de un unificador más general se sigue que si θ y σ son ambos umg's de $\{E_1, \ldots, E_n\}$, $E_1\theta$ es una variante de $E_1\sigma$. La última proposición muestra que $E_1\sigma$ puede ser obtenido de $E_1\theta$ simplemente renombrando variables. Así, los umg's son únicos modulo el renombre.

El algoritmo de unificación toma un conjunto finito de expresiones simples como entrada y devuelve un umg si el conjunto es unificable. En caso contrario, reporta el hecho de que el conjunto no es unificable.

Sea S un conjunto finito de expresiones simples. Si se localiza la posición más a la izquierda en la que no todas las expresiones en S tienen el mismo símbolo y se extrae de cada expresión en S la subexpresión que inicia en tal posición, el conjunto de subexpresiones así obtenido es referido como el *conjunto discordante de* S.

En el siguiente algoritmo S denota un conjunto finito de expresiones simples.

2.4 Algoritmo de Unificación

1. Hacer $k = 0$ y $\sigma_0 = \epsilon$.
2. Si $S\sigma_k$ contiene un solo elemento, parar: σ_k es un umg de S. De lo contrario, hallar el conjunto discordante 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, hacer $\sigma_{k+1} = \sigma_k\{v/t\}$, incrementar k y volver a 2. De lo contrario, parar y reportar que S no es unificable.

2.5 Teorema de Unificación

TEOREMA 2.5.1 (Teorema de unificación) *Sea S un conjunto finito de expresiones simples. Si S es unificable, el algoritmo de unificación termina y devuelve un umg para S. Si S no es unificable, el algoritmo de unificación termina y reporta tal hecho.*

Sea P un programa y G una meta. Una sustitución respuesta para $P \cup \{G\}$ es una sustitución para variables de G. Tal sustitución no necesariamente contiene una asignación para cada variable en G. En particular, si G no tiene variables la única sustitución respuesta posible es la sustitución identidad.

Sea P un programa, G la meta $\leftarrow A_1, \ldots, A_k$ y θ una sustitución respuesta para $P \cup \{G\}$. Se dice que θ es una sustitución respuesta correcta para $P \cup \{G\}$ si $\forall((A_1 \wedge \ldots \wedge A_k)\theta)$ es una consecuencia lógica de P.

Evidentemente, θ es una sustitución respuesta correcta para $P \cup \{\leftarrow A_1, \ldots, A_k\}$ si y solo si $P \cup \{\neg\forall((A_1 \wedge \ldots \wedge A_k)\theta)\}$ es no satisfacible. Tal definición de sustitución respuesta correcta captura el significado intuitivo de una "respuesta correcta", proveyendo el significado declarativo de la salida devuelta a partir de un programa y una meta. La semántica procedural se preocupa en establecer la equivalencia entre tal concepto declarativo y el correspondiente procedural, el cual es definido por el procedimiento de refutación utilizado por el sistema.

Un sistema de programación lógica puede también devolver como respuesta "no". Se dice que la respuesta "no" es correcta si $P \cup \{G\}$ es satisfacible.

2.6 Correctes de Resolución SLD

Aunque existen muchos procedimientos de refutación basados en la regla de inferencia resolución que resultan ser refinamientos del procedimiento original publicado por Robinson [15], el procedimiento de refutación implementado en los sistemas Prolog estándares es referido como resolución lineal con función de selección (de donde se toman las siglas SL del inglés "Linear resolution with Selection function") para cláusulas definidas (la D de SLD es tomada del inglés "Definite clauses") [12, 3].

En el presente trabajo de tesis solo se considerarán procedimientos y refutaciones SLD.

Una *regla de computación* es una función de un conjunto de metas a un conjunto de átomos tal que el valor de la función para una meta es siempre un átomo, referido como el *átomo seleccionado* en la meta.

Sean $G_i \leftarrow A_1, \ldots, A_m, \ldots, A_k$, C_{i+1} $A \leftarrow B_1, \ldots, B_q$ y R una regla de computación. G_{i+1} es *derivada de* G_i y C_{i+1} *utilizando un umg* θ_{i+1} *vía* R si se cumple que:

1. A_m es el átomo seleccionado dado por la regla de computación R.
2. $A_m\theta_{i+1} = A\theta_{i+1}$, es decir, θ_{i+1} es un umg de A_m y A.
3. G_{i+1} es la meta $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta_{i+1}$.

Se dice que G_{i+1} es una *resolvente* de G_i y C_{i+1}.

Sea P un programa, G una meta, y R una regla de computación. Una *derivación SLD de* $P \cup \{G\}$ *vía* R consiste de una secuencia (finita o infinita) $G_0 = G, G_1, \ldots$ de metas, una secuencia $C_1, C_2, \ldots$ de variantes de cláusulas del programa P, y una secuencia $\theta_1, \theta_2, \ldots$ de umg's, tales que cada G_{i+1} es derivado de G_i y C_{i+1} utilizando θ_{i+1} vía R. Una derivación SLD puede ser percibida gráficamente como en la Figura 1.

Una *refutación SLD de* $P \cup \{G\}$ *vía* R es una derivación SLD finita de $P \cup \{G\}$ vía R que tiene a la cláusula vacía $\square$ como última meta en la derivación. Si $G_n = \square$ se dice que la refutación tiene longitud n.

Una *refutación SLD sin restricciones* es una refutación SLD, excepto que es eliminado el requerimiento de que las sustituciones θ_i sean unificadores más generales. Solo se requiere que éstas sean unificadores.

Las derivaciones SLD pueden ser *finitas o infinitas*. Una derivación SLD finita puede tener éxito o fallar. Una *derivación SLD exitosa* es la que finaliza con

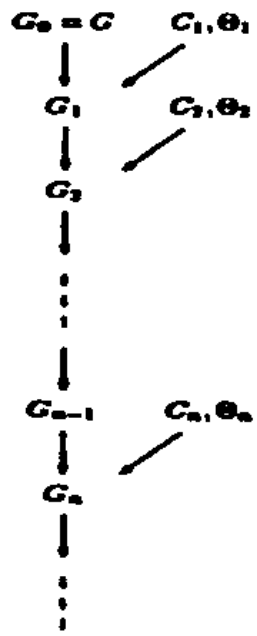

Figura 1: Una derivación SLD

la cláusula vacía. Es decir, una derivación SLD exitosa es una refutación. Una *derivación SLD fallida* es la que finaliza en una meta no vacía con la propiedad de que el átomo seleccionado en tal meta no unifica con la cabeza de ninguna cláusula de programa.

La figura 2 (sección 2.9) es un ejemplo de una derivación finita con ramas exitosas y ramas fallidas, y la figura 3 (sección 2.9) es un ejemplo de una derivación infinita también con ramas exitosas y ramas fallidas.

Sea P un programa. El *conjunto exitoso* de P es el conjunto de A's $\in B_P$ tales que $P \cup \{\leftarrow A\}$ tiene una refutación SLD (al utilizar alguna regla de computación que dependa de A).

Sea P un programa, G una meta, y R una regla de computación. Una *sustitución respuesta* R*-calculada* θ para $P \cup \{G\}$ es la sustitución obtenida al restringir la composición $\theta_1 \ldots \theta_n$ a las variables de G, donde $\theta_1 \ldots \theta_n$ es la secuencia de mgu's utilizados en una refutación SLD de $P \cup \{G\}$ vía R.

TEOREMA 2.6.1 (Corrección de resolución SLD) *Sea P un programa, G una meta, y R una regla de computación. Cada sustitución respuesta R-calculada para $P \cup \{G\}$ es una sustitución respuesta correcta.*

COROLARIO 2.6.1 *Sea P un programa, G una meta, y R una regla de computación. Si existe una refutación SLD de $P \cup \{G\}$ via R, $P \cup \{G\}$ no es satisfactible.*

El teorema 2.6.1 muestra la equivalencia entre el concepto declarativo de "una respuesta correcta" y el correspondiente procedural, estableciendo así la corrección de la resolución SLD.

Aún más, [3] establece en forma precisa que el conjunto exitoso de un programa está contenido en su modelo mínimo de Herbrand. En el presente trabajo de tesis no se considera necesario definir el concepto de modelo mínimo de Herbrand, pero basta decir que éste restringe aún más el espacio de búsqueda de "respuestas correctas".

Cabe mencionar que el espacio de búsqueda es normalmente concebido como una estructura arborescente (figuras 2 y 3). Tales arboles son conocidos entonces como *arboles de búsqueda* o como *arboles OR* (Ya que dos hermanos cualesquiera forman parte, cada uno, de caminos alternativos para obtenner una derivación).

Los *arboles AND* en contraposición son para los que todos los hermanos deben conformar un todo sin excluir a ninguno, como las subpartes de una meta: formar parte del todo:

Sea P un programa lógico. Un *árbol de prueba* (o *árbol AND*) para P es un árbol en el que cada nodo esta etiquetado con un átomo, y se define inductivamente como sigue:

1. Un nodo etiquetado con un átomo con símbolos que solo ocurren en P, o con variables, es un árbol de prueba para P.
2. Sea T un árbol de prueba para P con una hoja x etiquetada con A', y

   $A \leftarrow B_1, \ldots, B_n$ $n \geq 0$.

   una variante de una cláusula en P sin variables en común con las etiquetas de T. Si A y A' unifican con unificador más general θ, entonces el árbol obtenido al adicionar como hijos de x los nodos etiquetados con $B_1, \ldots, B_n$ y con θ aplicada a cada etiqueta del árbol así resultante es también un árbol de prueba para P. Si $n = 0$, el nodo correspondiente a x y etiquetado con $A'\theta$ es una *hoja cerrada*.

Un árbol de prueba con todas sus hojas cerradas es un *árbol de prueba cerrado*.

---

[3] Es usual en computación utilizar la palabra correctez para referirse al término corrección aprobado por la academia de la lengua española.

2.7 Completez de Resolución SLD

LEMA 2.7.1 (Lema ung) *Sea P un programa y G una meta. Si $P \cup \{G\}$ tiene una refutación SLD sin restricciones, $P \cup \{G\}$ tiene una refutación SLD de la misma longitud. Además, si $\theta_1, \ldots, \theta_n$ son los unificadores de la refutación SLD sin restricciones y $\theta'_1, \ldots, \theta'_n$ son los ung's de la refutación SLD, existe una sustitución γ tal que $(\theta_1 \ldots \theta_n) = (\theta'_1 \ldots \theta'_n)\gamma$.*

LEMA 2.7.2 (Lema sustento) *Sea P un programa, G una meta, y θ una sustitución. Si existe una refutación SLD de $P \cup \{G\theta\}$, existe una refutación SLD de $P \cup \{G\}$ de la misma longitud. Además, si $\theta_1, \ldots, \theta_n$ son los ung's de la refutación SLD de $P \cup \{G\theta\}$ y $\theta'_1, \ldots, \theta'_n$ son los ung's de la refutación SLD de $P \cup \{G\}$, existe una sustitución γ tal que $\theta(\theta_1 \ldots \theta_n) = (\theta'_1 \ldots \theta'_n)\gamma$.*

El resultado siguiente es debido a [3].

TEOREMA 2.7.1 (Primer resultado de completez[3]) *El conjunto existoso de un programa es igual a su modelo mínimo de Herbrand.*

El resultado de completez siguiente fue primero demostrado por [6], ver también [3].

TEOREMA 2.7.2 *Sea P un programa y G una meta. Si $P \cup \{G\}$ no es satisfactible, existe una regla R de computación y una refutación SLD de $P \cup \{G\}$ vía R.*

Debido a que las sustituciones respuesta calculadas son siempre "más generales", no es posible demostrar la contraparte exacta del teorema de corrección de resolución SLD. Sin embargo, se puede demostrar que cada sustitución respuesta correcta es una instancia de una sustitución respuesta calculada.

LEMA 2.7.3 *Sea P un programa y A un átomo. Si $\forall(A)$ es una consecuencia lógica de P, existe una refutación SLD de $P \cup \{\leftarrow A\}$ con la sustitución identidad como la sustitución respuesta calculada.*

El resultado más fuerte de completez es debido a [4].

TEOREMA 2.7.3 (Completez de resolución SLD) *Sea P un programa y G una meta. Para cada sustitución respuesta correcta θ para $P \cup \{G\}$ existe una regla de computación R, una sustitución respuesta R-calculada σ para $P \cup \{G\}$, y una sustitución γ tales que $\theta = \sigma\gamma$.*

[3] Es usual en computación utilizar la palabra completez para referirse al término completeza aprobado por la academia de la lengua española.

2.8 Independencia de la Regla de Computación

El teorema 2.7.2 establece que si $P \cup \{G\}$ no es satisfactible existe una refutación de $P \cup \{G\}$ utilizando alguna regla R de computación. Sin embargo, hasta el momento, R no está bajo control. El resultado de la presente sección es que si $P \cup \{G\}$ no es satisfactible se puede siempre hallar una refutación utilizando cualquier regla de computación dada a priori. Tal "independencia" de la regla de computación tiene implicaciones importantes en los sistemas de programación lógica .

Si C es una cláusula de programa, C^+ denota la cabeza y C^- el cuerpo.

LEMA 2.8.1 (Switching lemma) *Sea P un programa, G una meta, y R una regla de computación. Supongase que $P \cup \{G\}$ tiene una refutación SLD $G_0 = G, G_1, \ldots, G_{i-1}, G_i, G_{i+1}, \ldots, G_n = \square$ con cláusulas de entrada $C_1, \ldots, C_n$, y umg's $\theta_1, \ldots, \theta_n$ vía R. Si además:*

1. *G_{i-1} es $\leftarrow A_1, \ldots, A_{i-1}, A_i, \ldots, A_{j-1}, A_j, \ldots, A_k$*
2. *G_i es $\leftarrow (A_1, \ldots, A_{i-1}, C_i^-, \ldots, A_{j-1}, A_j, \ldots, A_k)\theta_i$*
3. *G_{i+1} es $\leftarrow (A_1, \ldots, A_{i-1}, C_i^-, \ldots, A_{j-1}, C_{i+1}^-, \ldots, A_k)\theta_i\theta_{i+1}$*

existe una refutación SLD de $P \cup \{G\}$ utilizando la regla R' de computación que es la misma que R, excepto que A_j es seleccionado en G_{i-1} en lugar de A_i y A_i es seleccionado en G_i en lugar de A_j. Además si σ es la sustitución respuesta R-calculada para $P \cup \{G\}$ y σ' es la sustitución respuesta R'-calculada, $G\sigma$ es una variante de $G\sigma'$

TEOREMA 2.8.1 (Independencia de la regla de computación) *Sea P un programa, G una meta, y R una regla de computación. Supongase que existe una refutación SLD de $P \cup \{G\}$ vía R. Sea R' cualquier regla de computación. Entonces, existe una refutación SLD de $P \cup \{G\}$ vía R'. Además, si σ y σ' son las respectivas sustituciones respuesta calculadas, $G\sigma$ es una variante de $G\sigma'$.*

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

TEOREMA 2.8.2 *Sea P un programa y R una regla de computación. El conjunto $R-$exitoso de P es igual a su modelo mínimo de Herbrand.*

El siguiente resultado se debe a [6]. Ver también [3].

TEOREMA 2.8.3 *Sea P un programa, G una meta, y R una regla de computación. Si $P \cup \{G\}$ no es satisfactible, existe una refutación SLD de $P \cup \{G\}$ vía R.*

El teorema siguiente se debe a [4].

TEOREMA 2.8.4 (Completitud fuerte de resolución SLD) *Sea P un programa, G una meta, y R una regla de computación. Para cada sustitución respuesta correcta θ para $P \cup \{G\}$ existe una sustitución respuesta R – calculada σ para $P \cup \{G\}$ y una sustitución γ tales que $\theta = \sigma\gamma$.*

2.8 Procedimientos de Refutación SLD

El *espacio de búsqueda* de una refutación es referido como el *árbol SLD*.

El resultado de la sección anterior establece que basta con solo una regla de computación para construir el árbol SLD. Por lo que tal regla de computación puede ser fijada a priori para después construir el árbol SLD utilizandola. De esta forma, el tamaño del espacio de búsqueda es considerablemente reducido.

Sea P un programa, G una meta, y R una regla de computación. El *árbol SLD para* $P \cup \{G\}$ *vía* R se define como:

1. Cada nodo del árbol es una meta (posiblemente 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 supongase que A_m es el átomo seleccionado por R. Entonces, tal nodo tiene un descendiente para cada cláusula de entrada $A \leftarrow B_1, \ldots, B_q$ tal que A_m y A son unificables. El descendiente es $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$ donde θ es un umg de A_m y A.
4. Los nodos que son la cláusula vacía no tienen descendientes.

Cada rama del árbol SLD es una derivación de $P \cup \{G\}$. Las ramas correspondientes a derivaciones exitosas son referidas como *ramas exitosas*, las correspondientes a derivaciónes infinitas se conocen como *ramas infinitas*, y las ramas correspondientes a derivaciónes fallidas son las *ramas fallidas*.

Como ejemplo considérese la meta `:- p( x, b )` y el programa[4]:

```
1. p( x, z ) :- q( x, y ), p( y, z )
2. p( x, z ).
3. q( a, b ).
```

Siguiendo la regla de computación de los sistemas Prolog estándares (que consiste en siempre seleccionar el átomo más a la izquierda en la meta) se obtiene el árbol SLD de la figura 2; y siguiendo la regla de computación consistente en seleccionar siempre el átomo más a la derecha en la meta, se obtiene el árbol SLD de la figura 3 (en ambos casos los átomos seleccionados son los átomos subrayados).

El ejemplo ilustra el hecho de que la elección de la regla de computación repercute en el tamaño y estructura del árbol SLD correspondiente: El árbol de la

[4]El símbolo ← toma diversas formas en los sistemas estándares, una de cuyas formas es `:-`.

figura 2 es finito, mientras que el árbol de la figura 3 es infinito (habiendo en ambos casos dos ramas exitosas correspondientes a las respuestas {x/a} y {x/b}). Sin embargo, no importa la elección de la regla de computación si $P \cup \{G\}$ es insatisfactible, ya que en tal caso el árbol SLD correspondiente tendrá una rama exitosa (teorema 2.7.2).

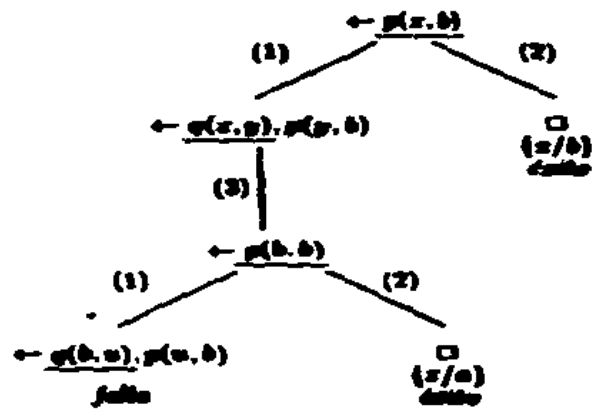

Figura 2: Un árbol SLD finito.

TEOREMA 2.8.1 *Sea P un programa, G una meta, y R una regla de computación. Si $P \cup \{G\}$ no es satisfactible el árbol SLD para $P \cup \{G\}$ vía R tiene al menos una rama exitosa.*

TEOREMA 2.8.2 *Sea P un programa, G una meta, y R una regla de computación. Cada sustitución respuesta correcta θ para $P \cup \{G\}$ es "despleguda" en el árbol SLD para $P \cup \{G\}$ vía R.*

θ "despleguda" significa que hay una rama exitosa tal que θ es una instancia de la sustitución respuesta calculada de la refutación correspondiente a esa rama.

Aunque dos árboles SLD puedan ser muy diferentes en tamaño y estructura, son esencialmente los mismos con respecto a sus ramas exitosas.

TEOREMA 2.8.3 *Sea P un programa y G una meta. Cada árbol SLD para $P \cup \{G\}$ tiene una infinidad de ramas exitosas o cada árbol SLD para $P \cup \{G\}$ tiene el mismo número finito de ramas exitosas.*

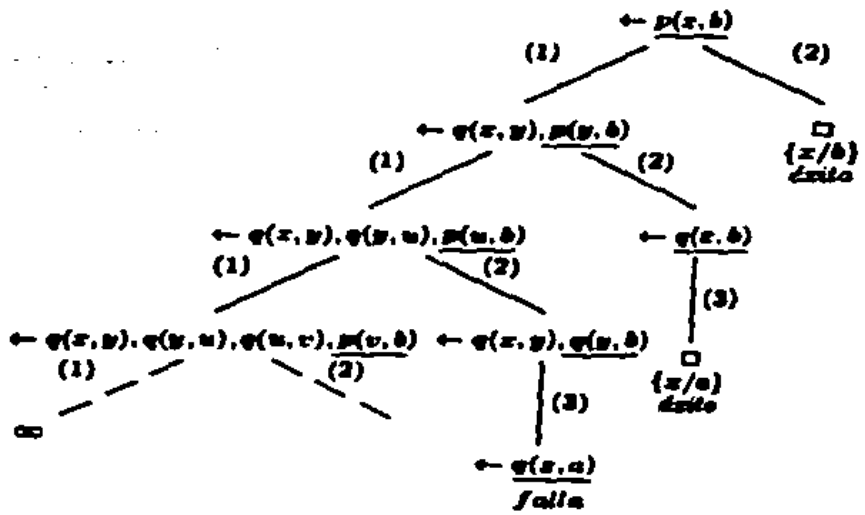

Figura 3: Un árbol SLD infinito.

Una *regla de búsqueda* es una estrategia de búsqueda en árboles SLD para hallar ramas exitosas. Un *procedimiento de refutación SLD* queda determinando por una regla de computación y una regla de búsqueda.

Los sistemas Prolog estándares utilizan como regla de computación aquella que siempre selecciona el átomo más a la izquierda en una meta, junto con *búsqueda a profundidad como regla de búsqueda*. La regla de búsqueda se implementa utilizando una pila de metas. Una instancia de tal pila representa la rama actualmente investigada. La computación se convierte así en una secuencia de entradas y salidas en la pila. Cuando el átomo seleccionado de la meta en el tope de la pila unifica con la cabeza de una cláusula de programa, la resolvente es metida a la pila. Cuando ya no existen más cláusulas de programa cuya cabeza unifique con el átomo seleccionado en la meta del tope de la pila, tal meta es sacada de la pila y se repite el proceso para el nuevo tope de la pila.

En un sistema cuya regla de búsqueda sea la búsqueda a profundidad, queda establecido el orden en que las cláusulas de un programa serán probadas. Los sistemas Prolog estándares utilizan el orden de las cláusulas en un programa como el orden en que éstas serán probadas, lo que es muy simple y eficiente de implementar, pero con la desventaja de que cada llamada a una definición prueba las cláusulas en la definición en exactamente el mismo orden.

Es desde luego deseable que la regla de búsqueda sea tal que cada rama exitosa en el árbol SLD sea eventualmente hallada. Sin embargo, una regla de búsqueda sin un componente de búsqueda a lo ancho (como lo es la búsqueda a profundidad) puede no hallar cada rama exitosa en un árbol SLD infinito.

La razón de que los sistemas Prolog estándares no implementen una regla de búsqueda con un componente de búsqueda a lo ancho, es que tal componente es menos compatible con una implementación eficiente.

Es natural preguntarse si los sistemas Prolog estándares son "completos". De acuerdo al teorema 2.9.1, si $P \cup \{G\}$ no es unidirectible, no importa que regla de computación sea utilizada, el árbol SLD correspondiente contendrá siempre una rama exitosa. Desgraciadamente, ninguno de los resultados de completez anteriores son aplicables a muchos de los sistemas Prolog actuales debido a consideraciones de eficiencia.

Para aclarar lo anterior considérense la meta `G :- p( a, c ).` y el programa *P* siguiente:

```
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 ).
```

Se puede demostrar que $P \cup \{G\}$ tiene una refutación, y aún más, que si omite cualquier cláusula de *P*, $P \cup \{G\}$ no contendrá ya refutaciones.

Es un hecho que los sistemas Prolog que utilizan búsqueda a profundidad y que prueban las cláusulas en un orden fijo (normalmente tal orden es de arriba a abajo) nunca hallarían en el ejemplo una refutación, sin importar la regla de computación utilizada ni el orden de las cláusulas en el programa. Lo anterior se debe a que si el orden conduce a probar la cláusula 3 antes que la cláusula 4, ésta última ya no nunca se probará, y viceversa, si el orden conduce a probar la cláusula 4 antes que la cláusula 3, la cláusula 3 ya no se probará. Sin embargo, son necesarias todas las cláusulas para hallar una refutación.

La figura 4 ilustra tal situación. Se puede notar que la rama más a la izquierda del árbol SLD de *P* es infinita (de hecho, ésta es infinita para cada regla de computación y cada orden fijo para probar las cláusulas), por lo que un sistema que utiliza búsqueda a profundidad nunca hallará la rama exitosa. En tal derivación se ha utilizado la regla de computación estándar (que consiste en seleccionar siempre el átomo más a la izquierda en la meta y probar las cláusulas en orden de arriba a abajo).

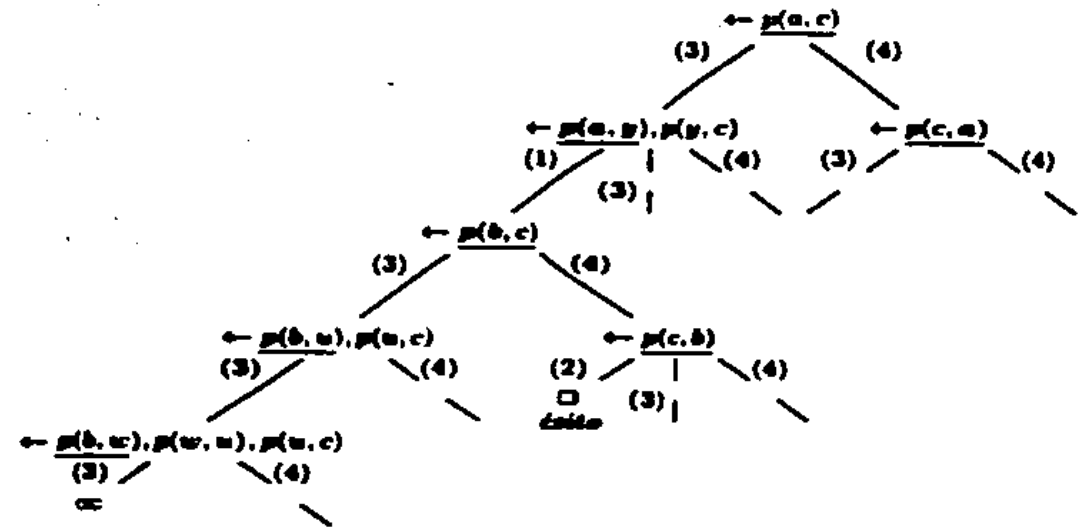

Figura 4: Arbol SLD que ilustra el problema de la búsqueda a profundidad.

3 Factorización

Un *conjunto S de cláusulas* es *inconsistente* si y solo si no es consistente.

Un *conjunto S de cláusulas* es *consistente* si y solo si todas sus cláusulas son ciertas en alguna interpretación de S.

Una *cláusula* es *cierta* en una interpretación de un conjunto S de cláusulas si y solo si cada instancia de la cláusula libre de variables, obtenida al reemplazar variables por términos en el universo de discurso de S, es cierta en la interpretación. En caso contrario, la cláusula es falsa en la interpretación.

Una *cláusula libre de variables* es *cierta* en una interpretación I si y solo si siempre que todas sus condiciones sean ciertas en I, al menos una de sus conclusiones es cierta en I.

Equivalentemente, una *cláusula libre de variables* es *cierta* en una interpretación I si y solo si al menos una de sus condiciones es falsa en I o al menos una de sus conclusiones es cierta en I. En caso contrario, la cláusula es falsa en I.

Un *sistema de reglas de inferencia* es *correcto* si cada conjunto de cláusulas que tiene una refutación, construida de acuerdo a las reglas del sistema, es in-

consistente.

Un *sistema de reglas de inferencia es completo* si cada conjunto inconsistente de cláusulas tiene una refutación, construida de acuerdo a las reglas del sistema.

Un *sistema de reglas de inferencia correcto y completo* es para el que la noción semántica de inconsistencia coincide con la noción sintáctica de refutabilidad.

Como única regla de inferencia Resolución es completa para demostrar la inconsistencia de conjuntos de cláusulas de Horn. Aún más, funciona también para muchos conjuntos de cláusulas no Horn.

Para que Resolución funcione para todos los conjuntos inconsistentes de cláusulas no Horn debe asistirse de Factorización.

La combinación de Factorización y Resolución es descrita por primera vez en un artículo no publicado de Robinson (ver [9] página 139), y es equivalente a la versión publicada de la regla de Resolución [15]. Por lo tanto, la demostración de completez en el artículo publicado establece la completez de la combinación Resolución y Factorización.

La razón de que la demostración de completez en [15] jamás apele a la regla de factorización es que en el artículo se habla en todo momento de conjuntos[5] de cláusulas. Es decir, en [15] no se consideran secuencias[6] de cláusulas.

Por otra parte, dado un programa P y una meta G, el árbol SLD correspondiente puede encontrar nodos que son instancias unos de otros. Es decir, el árbol SLD puede evidenciar redundancias en la computación.

En general, dado un programa P y una meta G, al efectuar evaluación parcial se pueden detectar redundancias en la computación.

Una observación interesante es que, adicionalmente a su gran utilidad como regla de inferencia, factorización puede ser utilizada como una regla más en evaluación parcial en tanto que se puede utilizar para eliminar redundancias en la computación.

[5] Por definición en un conjunto cualesquiera no se puede hablar de un elemento del conjunto mayor, menor, o igual, a otro elemento también en el conjunto. Es decir, en un conjunto no existe ningún tipo de orden. De tal propiedad se deriva también el hecho de que en un conjunto no se duplica ninguno de sus elementos ya que un elemento del conjunto es distinto de cualquier otro elemento del conjunto.

[6] Una secuencia es una colección de objetos que tiene asociada una relación de orden, por lo que en una secuencia sí pueden existir duplicados y dos permutaciones distintas de exactamente los mismos objetos constituyen dos secuencias distintas.

3.1 Transformación de Programas

La transformación de programas[7] [2] es una técnica de manipulación simbólica utilizada para producir programas eficientes dado que se conoce parte de la entrada de un programa. Es decir, con la entrada conocida se efectua una evaluación parcial del programa original para obtener un programa residual, el cual deberá ser al menos tan eficiente como el programa original con entrada restringida.

El *programa residual* es, a su vez, un programa lógico y *se obtiene mediante la aplicación de las operaciones de definición, desdoblar (unfold), doblar (fold), sustitución de metas, y eliminación de cláusulas*, al programa original con entrada parcialmente conocida.

La operación de *definición* consiste en introducir una nueva cláusula (referida como cláusula definición) para definir un nuevo predicado en términos de predicados ya existentes.

La operación *desdoblar* consiste en realizar un paso de resolución.

La operación *doblar* consiste en reemplazar una submeta en el cuerpo de una cláusula que es una instancia del cuerpo de una cláusula definición por la instancia correspondiente de la cabeza de la cláusula definición.

La operación doblar es la inversa de la operación desdoblar en el sentido de que si una cláusula F se obtiene doblando la cláusula C utilizando una cláusula definición D, al desdoblar F utilizando D se obtendrá una variante de C.

Dado un término, átomo, o cláusula f, $vars(f)$ es el conjunto de variables que ocurren en f.

Dada una cláusula C de la forma $H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n$, las *variables de liga* de $A_1, \ldots, A_m$ en C son las que ocurren en $A_1, \ldots, A_m$ y en $H, B_1, \ldots, B_n$.

Sea C una cláusula de un programa P en el cuerpo de la cual ocurre la submeta G_1, supongase también que para alguna submeta G_2, la fórmula

$$\forall x_1, \ldots, x_s (\exists y_1, \ldots, y_m G_1 \leftrightarrow \exists z_1, \ldots, z_n G_2)$$

es consecuencia lógica de P. En donde, $x_1, \ldots, x_s$ son las variables de liga de G_1 en C. $\{y_1, \ldots, y_m\} = vars(G_1) - \{x_1, \ldots, x_s\}$. $\{z_1, \ldots, z_n\} = vars(G_2) - \{x_1, \ldots, x_s\}$. Y $\{z_1, \ldots, z_n\} \cap vars(C) = \{\}$.

[7] Conocida también como evaluación parcial o deducción parcial

La operación de *sustitución de metas* consiste en reemplazar G_1 por G_2 en la cláusula C.

La operación de *eliminación de cláusulas* consiste en eliminar una cláusula C de un programa P para obtener un programa Q en el que C es consecuencia lógica.

La aplicación de las operaciones de transformación de programas preserva corrección. Es decir, todas las fórmulas que son consecuencia lógica del programa residual lo son del programa original.

Sin embargo, para preservar completitud en la aplicación de las operaciones de transformación de programas, es decir, para asegurar que todas las fórmulas que son consecuencia lógica del programa original lo sean también del programa residual, se debe restringir el uso de estas operaciones [17].

3.2 Factorización como parte de Transformación de Programas

La Transformación de programas tiene por objeto producir un programa residual al menos tan eficiente como el original. Esto se logra de varias formas: reduciendo el no *determinismo* inherente en muchos programas lógicos, evitando recalcular los valores de estructuras de datos intermedias, eliminando variables o estructuras de datos innecesarias, etc.

Una clase importante de programas lógicos son los programas lógicos recursivos. En esta clase se encuentran los programas lógicos que contienen dos o más llamadas recursivas. Por naturaleza, estos últimos provocan corridas ineficientes al ser fuentes de computaciones redundantes. Es decir, como regla general, al hacer dos o más llamadas recursivas eventualmente se producirán computaciones idénticas en distintas secciones de la corrida que corresponden a idénticos niveles de recursión.

Como ilustración considérese el problema de hallar el n-ésimo elemento de la sucesión de Fibonacci. La sucesión $\{a_n\}$ de Fibonacci es tal que $a_0 = 0$, $a_1 = 1$, y para $n > 1$, $a_n = a_{n-1} + a_{n-2}$.

El programa lógico siguiente se deriva de forma natural de la definición de $\{a_n\}$:

```
%fibon(N,F): F es el N-esimo elemento de la sucesion de Fibonacci.

fibon(0, 0).
fibon(1, 1).
fibon(N, F) :- N > 1
. Np is N - 1, fibon(Np, Fp)
. Npp is N - 2, fibon(Npp, Fpp)
. F is Fp + Fpp.
```

Así por ejemplo, para el átomo $fibon(5, F)$ se tiene el árbol de prueba de la figura 5[9].

En la figura 5 se puede observar que en distintas secciones de la corrida se calculan de forma redundante a_0, a_1, a_2, y a_3.

Para evitar calcular a_0, a_1, a_2, y a_3 más de una vez, se debe:

1. "Subir un nivel" el subárbol izquierdo y,
2. Eliminar átomos duplicados en el árbol resultante.

[9]En la figura 5, f representa el símbolo de predicado $fibon$.

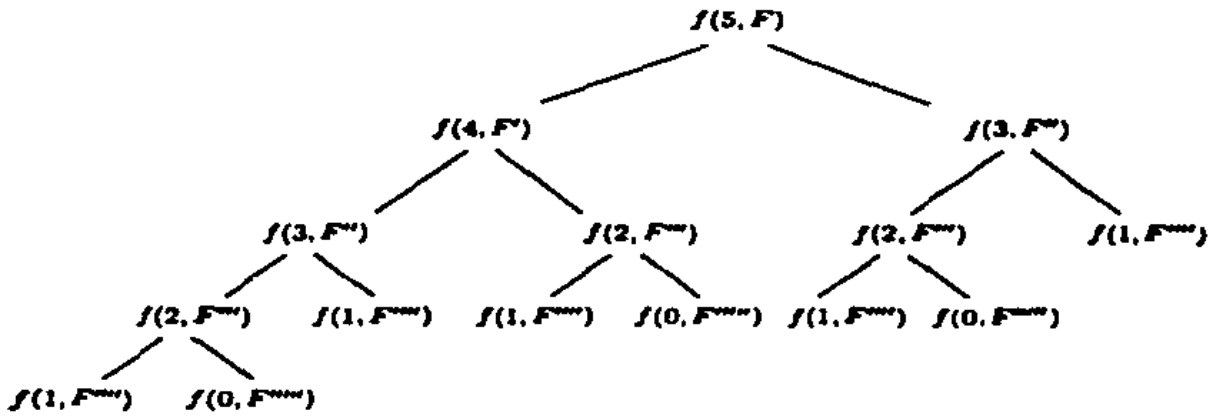

Figura 5: Arbol de prueba para $fibon(5, F)$ -Programa original.

El paso 1 se traduce en desdoblar la primera llamada recursiva a $fibon(N, F)$, resultando el programa:

```
%fibon(N,F): F es el N-esimo elemnato de la sucesion de Fibonacci.

fibon(0, 0).
fibon(1, 1).
fibon(2, 1).
fibon(N, F) :- N > 2, Np is N - 1
, N is Np - 1, fibon(N, G)
, Np is Np - 2, fibon(Np, Gp), Fp is G + Gp
, Npp is N - 2, fibon(Npp, Fpp)
, F is Fp + Fpp.
```

En donde, dado que Np es $N-1$ y Nf es $Np-1$, Nf es $(N-1)-1 = N-2$ que es justamente Npp. Es decir, la primera y tercera llamadas recursivas unifican.

Al aplicar la sustitución $\theta = \{Npp/Nf, Fpp/G, Np/N-1, Fp/G+Gp\}$, resulta el programa lógico:

```
%fibon(N,F): F es el N-esimo elemento de la sucesion de Fibonacci.
```

```
fibon(0, 0).
fibon(1, 1).
fibon(2, 1).
fibon(N, F) :- N > 2
, N is N - 2, fibon(N, G)
, Np is N - 3, fibon(Np, Gp)
, N is N - 2, fibon(N, G)
, F is G + Gp + G.
```

Por último, al eliminar los átomos duplicados (paso 2) se obtiene:

```
%fibon(N,F): F es el N-esimo elemento de la sucesion de Fibonacci.

fibon(0, 0).
fibon(1, 1).
fibon(2, 1).
fibon(N, F) :- N > 2
, N is N - 2, fibon(N, G)
, Np is N - 3, fibon(Np, Gp)
, F is G + Gp + G.
```

El árbol de prueba del programa residual correspondiente al átomo $fibon(5, F)$ es el mostrado en la figura 6.

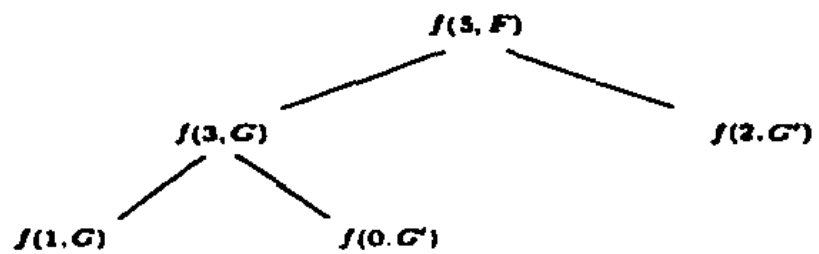

Figura 6: Arbol de prueba para $fibon(5, F)$ -Programa residual.

De este último resultado se observa una mejora notable, ya que para la meta $\leftarrow fibon(5, F)$, se han eliminado una cantidad significativa de cálculos redundantes.

Es más o menos obvio que cualquier programa lógico que consienga dos o más llamadas recursivas a un mismo predicado llevará consigo cálculos redundantes al provocar la ejecución simultánea de tales llamadas. Sin embargo, no siempre se tiene que aceptar irremediablemente semejante ineficiencia, en algunos casos es posible manipular el programa para obtener una versión más eficiente. Existen, de hecho, casos en los que se reducen dos o más llamadas recursivas a una sola.

Como el ejemplo anterior muestra, un primer paso en la manipulación de un programa lógico con dos o más llamadas recursivas es desdoblarlas cuantas veces sea necesario para obtener una cláusula cuyo cuerpo contiene dos átomos unificables.

En este punto el problema se torna en determinar cuando dos átomos A_i y A_j unifican. Es decir, el problema consiste en hallar una sustitución θ que al aplicarla a $\leftarrow A_1, \ldots, A_i, \ldots, A_j, \ldots, A_n$ hagan a A_i y A_j sintácticamente equivalentes.

Una vez hallada θ, $\leftarrow A_1, \ldots, A_i, \ldots, A_j, \ldots, A_n$ es reemplazada por $(\leftarrow A_1, \ldots, A_i, \ldots, A_n)\theta$.

Realmente el problema que se plantea la factorización es el enunciado en los dos párrafos anteriores, es decir, *el problema de la factorización es determinar cuando dos átomos A_i y A_j en $\leftarrow A_1, \ldots, A_i, \ldots, A_j, \ldots, A_n$ unifican*.

El hallar dos átomos A_i y A_j en $\leftarrow A_1, \ldots, A_i, \ldots, A_j, \ldots, A_n$ que unifiquen significa intentar unificar cada A_k, para $k \in \{1, \ldots, n\}$, con cada uno de los A_l, para $l \in \{1, \ldots, n\}$, lo cual sería lo suficientemente ineficiente como para considerar hacerlo al tiempo de ejecución[6]. La alternativa es intentar hallar A_i y A_j al tiempo de compilación sin importar mucho qué tan ineficiente resulte esta operación ya que sólo se llevará a cabo una vez.

Sin embargo, como el ejemplo anterior muestra, lo normal es que las redundancias en computaciones se presenten al tiempo de ejecución, por lo que al tiempo de compilación todo lo que se puede hacer es eliminar sólo una parte de ellas, una vez evidenciadas en el programa residual. Es decir, el compilador todo lo que puede hacer es remover las redundancias que se presentan en un programa que las contiene.

Desde luego, habrá programas para los que surjan todas las redundancias al transformarlo, y para esta clase de programas el problema está resuelto. Sin

[6] Este razonamiento muestra que la factorización es una operación cuadrática, ya que el intentar empatar dos elementos en un conjunto de n elementos significa comparar cada uno de los n con cada uno de los n, lo que se traduce en n^2 comparaciones.

embargo, pareciera que en general, según el grado de eficiencia deseada es la manipulación que se tiene que llevar a cabo en el programa.

La última afirmación no se cumple en general, ya que el manipular demás un programa puede, lejos de eliminar redundancias, generarlas. Del mismo modo, el uso no moderado de la factorización puede también generar redundancias.

4 Combinación de Factorización y el método de Tamaki

El modelo streams o de flujos de ejecución de programas Prolog [7, 8, 13] concibe a un predicado como una función que mapea un conjunto de variables ligadas a otro conjunto de variables también ligadas vía un flujo o stream. Para cada conjunto de variables ligadas en un flujo de entrada, el predicado genera uno a la vez cero o más conjuntos de variables ligadas como resultado de computaciones exitosas.

4.1 Método de Búsqueda Exhaustiva de Tamaki

Si en el modelo de flujos de ejecución de programas Prolog para todos los conjuntos de variables ligadas de entrada sus salidas se mezclan en cascada en un único flujo de salida, tal flujo arrojará como una sola todas las salidas correspondientes a la entrada.

Tal esquema de ejecución aunque elegante, tiene el inconveniente de que se deben generar y mantener numerosos ambientes de ligas. Sin embargo, bajo este esquema existe una clase importante de programas más prácticos. En esta clase los predicados de un programa son referidos como predicados de modo fijo.

Un *predicado es de modo fijo* si en sus argumentos están completamente identificadas las entradas y las salidas y los datos que fluyen son términos aterrizados (libres de variables)[10].

Para un predicado de modo fijo el compilador puede determinar, para cada flujo, el conjunto de variables cuyas ligas deberán ser mantenidas por tal flujo y asignar cada variable a una entrada de una tupla fluyendo en el flujo.

Como ilustración considérese la cláusula

$$p((X,Y),\ (Z,V)) \leftarrow q((X),\ (Z,W)),\ r((Y,Z,W),\ (V)).^{11} \qquad (1)$$

Ya que p no es la composición de q y r, se necesitan interfaces que distribuyan y combinen los elementos de las tuplas que fluyen en los flujos (ver figura 7). La segunda interfaz i_2 debe sincronizar sus dos flujos de entrada. Es decir, i_2 debe procesar un triple (Y', Z, W') sólo si, tanto Y' como (Z, W'), provienen del

[10] Un predicado de esta naturaleza puede siempre ser llamado con entradas aterrizadas y devolver salidas aterrizadas.

[11] Donde (y) encierran sólo variables de entrada o sólo variables de salida.

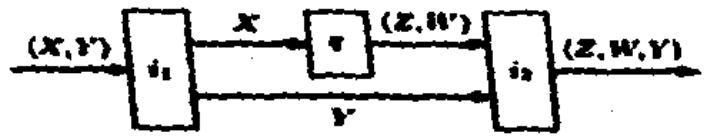

Figura 7: Interfaces de Flujos.

mismo (X,Y) que ingresó al flujo de entrada del bloque entero. Esto constituye un problema serio de sincronización cuya solución afectará el desempeño de la implementación.

Un problema similar se presenta al intentar paralelizar el AND. Considerar la cláusula

$$p(\langle X\rangle,\ \langle Y,Z\rangle) \leftarrow q(\langle X\rangle,\ \langle Y\rangle), r(\langle X\rangle,\ \langle Z\rangle) \qquad (2)$$

Ya que los dos subobjetivos del cuerpo no tienen dependencia de Entrada/Salida, se pueden combinar los dos flujos de Y y Z en un flujo de pares para ejecutar los dos subobjetivos de forma paralela. Sin embargo, nuevamente se presenta el problema de sincronizar los dos flujos de entrada de la operación combinada (una Y y una Z forman la pareja (Y,Z) sólo si son generados de la misma X).

La solución dada por Tamaki [16] a tal problema de sincronización se resume en[12]:

1. Restringir el cuerpo de una cláusula de forma tal que pueda ser recursivamente estructurado en bloques de una entrada y una salida.
2. Las entradas a una cláusula o a un bloque deben ser valores individuales, y la salida de una cláusula o bloque será un flujo.

[12] Aunque Tamaki propone su método para ser aplicado a un programa de modo fijo y obtener un programa con guardas en sus cláusulas de Horn (y que por tanto hace uso del operador comit |), en el presente trabajo de tesis se restringirá el método para obtener un programa no paralelo aprovechando que el commit puede ser reemplazado por un corte !.

3. Conectar dos bloques secuenciales mediante una interfaz. Tal interfaz descompondrá el flujo de salida del primer bloque para hacer una llamada al segundo bloque por cada elemento del flujo y pasará cada flujo de salida así resultante a un flujo único de salida. La interfaz es también responsable de dejar pasar las variables que son entrada al bloque entero pero que son utilizadas solo en el segundo bloque (como Y en el primer ejemplo). Por su naturaleza de operación, la interfaz encapsulará al segundo bloque.

Al aplicar estas ideas a la cláusula del primer ejemplo resulta la implementación mostrada en la figura 8 (en donde una línea delgada indica el flujo de un

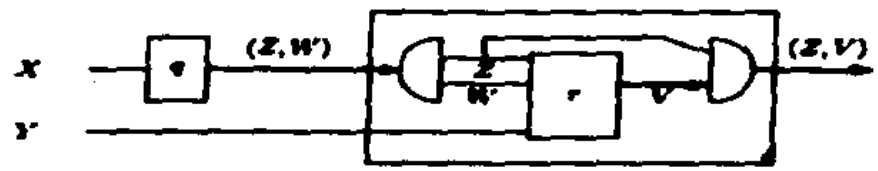

Figura 8: Interfaz Encapsuladora.

valor individual, una línea gruesa indica el flujo de varios valores, un semicírculo izquierdo descompone un flujo en valores individuales, y un semicírculo derecho compone valores individuales y flujos en un único flujo).

El problema de sincronización es simplificado por la decisión de ingresar a un bloque solo valores individuales.

Se ha mencionado que en un predicado de modo fijo los datos que fluyen son términos aterrizados. Por tanto, para cubrir este requerimiento, si una variable forma parte de la entrada de una submeta del cuerpo de un predicado de esta naturaleza (caso en el que a tal variable se le denomina consumidora), ésta debe ocurrir antes, de acuerdo al orden de ejecución, como salida en otra submeta o debe ser entrada a la cabeza (caso en el que a la variable se le refiere como productora). De acuerdo a este razonamiento, las variables que forman parte de la salida de la cabeza son consumidoras. Así por ejemplo, de acuerdo a la regla de computación de los sistemas Prolog estándares, la variable X del predicado q es consumidora tanto en la cláusula 1 como en la cláusula 2. En el mismo predicado, las variables Z y W', son productoras en la cláusula 1, y la variable Y' es productora en la cláusula 2. El predicado r tiene como consumidoras a Y', Z, y W', en la cláusula 1, y a X' en la cláusula 2. En la cláusula 1, V' es productora

para r, y en la cláusula 2. Z es productora para r. La cabeza p de la cláusula 1 tiene a X y Y' como productoras, y a Z y Y' como consumidoras, y la cabeza p de la cláusula 2, tiene a X como productora, y a Y' y Z como consumidoras.

Una restricción adicional es impuesta por la composición de valores individuales en un único flujo. Para poder componer varias variables en una única, cada una de éstas debe ocurrir sólo una vez. Lo anterior se traduce en que ninguna variable puede tener ocurrencias productoras en más de una submeta, incluyendo la cabeza.

Si un predicado p de modo fijo tiene n argumentos de entrada y m argumentos de salida, se dice que tiene aridad (n, m), y es de la forma

$$p((s_1, \ldots, s_n), (t_1, \ldots, t_m))$$

en donde $s_1, \ldots, s_n$ y $t_1, \ldots, t_m$ son términos. p también puede denotarse por $p((\sigma), (\tau))$, en donde $\sigma = (s_1, \ldots, s_n)$ y $\tau = (t_1, \ldots, t_m)$.

Siguiendo con la notación, una cláusula se denota por $H \leftarrow B$, donde H es la cabeza de la cláusula y B, el cuerpo de la cláusula, es una secuencia de bloques. Un bloque B es una meta o es de la forma $\{B_1, \ldots, B_n\}$ con $n \geq 0$, donde cada B_i es un bloque.

La estructura de bloques induce una clasificación de las variables de una cláusula. Una variable que debe ingresar a un bloque con un valor asignado es una variable de importación para tal bloque. Del mismo modo, una variable cuyo valor es determinado en un bloque para ser utilizado fuera de él, es una variable de exportación para el bloque. Por último, una variable cuyo valor debe ser determinado antes de ingresar a un bloque para ser utilizado fuera de él, es una variable de paso para el bloque (corresponde a la interfaz del bloque ser conducto de una variable de paso)

Si $Import(B)$, $Export(B)$, y $Paso(B)$, denotan los conjuntos de variables de importación, exportación, y de paso, respectivamente, del bloque B en

$$p((\sigma), (\tau)) \leftarrow B$$

entonces:

1. $Import(B) = \{todas\ las\ variables\ en\ \sigma\}$.
2. $Export(B) = \{todas\ las\ variables\ en\ \tau\}$.

Para el i ésimo componente de $B = \{B_1, \ldots, B_n\}$, se tiene:

1. $Import(B_i) = (Import(B) \cup Salida(B_1, \ldots, B_{i-1})) \cap Entrada(B_i)$.

2. $Export(B_i) = (Export(B) \cup Entrada(B_{i+1}, \ldots, B_n)) \cap Salida(B_i)$.

3. $Paso(B_i) = (Import(B) \cup Salida(B_1, \ldots, B_{i-1})) \cap (Export(B) \cup Entrada(B_{i+1}, \ldots, B_n))$.

En donde $Entrada(B_1, \ldots, B_n)$ ($Salida(B_1, \ldots, B_n)$) es el conjunto de variables que ocurren en los argumentos de entrada (argumentos de salida) de las metas en los bloques $B_1, \ldots, B_n$, con $n \geq 0$.

Para estructurar las cláusulas de un predicado p en bloques, se definen los predicados Q^p, R^B, e I_i^B. Q^p tiene aridad $n + 2$ si p tiene aridad (n, m) y es el predicado encargado de componer en un único flujo a todos los conjuntos de variables ligadas de salida que corresponden a un conjunto de variables ligadas de entrada. Para cada bloque B con una tupla de importación de tamaño n, R^B tiene aridad $n + 2$. Para cada bloque secuencial B con n componentes, se definen n predicados (las interfaces en el bloque) I_i^B ($1 \leq i \leq n$) de aridad $j+3$, donde j es el tamaño de la tupla de paso del i-ésimo componente del bloque.

Q^p se define para cada símbolo de predicado no primitivo con n argumentos de entrada y definido por $k > 0$ cláusulas con cuerpos $B_1, \ldots, B_k$:

$Q^p(x_1, \ldots, x_n, z_0, z_2) \leftarrow R_1^B(x_1, \ldots, x_n, z_0, z_1), R_2^B(x_1, \ldots, x_n, z_1, z_2), \ldots, R_k^B(x_1, \ldots, x_n, z_{k-1}, z_k)$.

Si σ y τ denotan las tuplas de importación y de exportación del bloque B, y si B es un bloque vacío, es decir, si B corresponde a una cláusula unitaria, se incluyen las cláusulas:

$R(\sigma, z_0, z_1) \leftarrow^!. \; z_0 = [(\tau) \mid z_1]$.
$R(\_, \ldots, \_, z_0, z_1) \leftarrow^!. \; z_0 = z_1$.

Si por el contrario B corresponde a una cláusula no unitaria, que tiene la forma $B = \{B_1, \ldots, B_n\}$ (para $n > 0$), y se deben incluir las cláusulas:

$R(\sigma, z_0, z_1) \leftarrow^!. \; R^{B_1}(\sigma_1, x, []), I_1(\pi_1, x, z_0, z_1)$.
$R(\_, \ldots, \_, z_0, z_1) \leftarrow^!. \; z_0 = z_1$.

$I_i(\pi_i, [(\tau_i) \mid x], z_0, z_2) \leftarrow^!. \; R^{B_{i+1}}(\sigma_{i+1}, y, []), I_{i+1}(\pi_{i+1}, y, z_0, z_1), I_i(\pi_i, x, z_1, z_2)$.

$I_i(\_, \ldots, \_, [], z_0, z_1) \leftarrow^!. \; z_0 = z_1$.

$I_i(\pi_i, [\_ \mid x], z_0, z_1) \leftarrow^!. \; I_i(\pi_i, x, z_0, z_1)$.

En donde $1 \leq i \leq n-1$, y

$I_n(\pi_n, [(\tau_n) \mid x], z_0, z_2) \leftarrow!, z_0 = [(\tau) \mid z_1], I_n(\pi_n, x, z_1, z_2)$.

$I_n(\_, \ldots, \_, [], z_0, z_1) \leftarrow!, z_0 = z_1$.

$I_n(\pi_n, [\_ \mid x], z_0, z_1) \leftarrow!, I_n(\pi_n, x, z_0, z_1)$.

En donde σ_i, τ_i, y π_i, son las tuplas de importación, exportación, y de paso, de B_i.

Como ilustración considerese el problema de hallar las permutaciones de los elementos de una lista, para lo que se define el predicado *permuta*:

```
% permuta(L,M): M es una lista permutacion de la lista L.

  permuta( [], [] ). % (1)
  permuta( L, [X|M] ) :-del( L, [X,M] ),permuta( M, M ). % (2)

% del([L1],[A,L2]):  L2 es la lista obtenida al eliminar A de L1[13].

  del( [N|T], [N,T] ). % (3)
  del( [N|T], [A,[N|T2]] ) :-del( T, [A,T2] ). % (4)
```

Al transformar la versión (1)-(4) de *permuta* desdoblando una sola vez la llamada recursiva en la clausula (2), o sea, al unificar *permuta*(M, N) con la cabeza de (1) y con la cabeza de (2), se obtiene:

```
permuta(L,[X]):-del(L,[X,[]]). %(5)
permuta(L,[X,Xp|Np]):-del(L,[X,Lp]),del(Lp,[Xp,Np]),permuta(Np,Np).
```

En (5), la submeta *del*(L, [N, []]) significa que al eliminar N de la lista L, se obtiene la lista vacía. Es decir, L es la lista [N], por lo que se puede reemplazar L por [N] en la cabeza, y eliminar así el cuerpo de la cláusula. Lo anterior no es más que realizar evaluación parcial de (5) utilizando (3) para obtener (6).

```
permuta([X],[X]). %(6)
```

Resultando el programa residual:

[13] Debido a que en las implementaciones estándares de Prolog no se pueden utilizar (y) para formar terminos se han utilizado [y]

```
permuta([],[]).
permuta([X],[X]).
permuta(L,[X,Xp|Mp]):-del(L,[X,Lp]),del(Lp,[Xp,Mp]),permuta(Mp,Mp).

% del([L1],[A,L2]): L2 es la lista obtenida al eliminar A de L1.

del( [X|T], [X,T] ).
del( [X|T], [A,[X|T2]] ) :-del( T, [A,T2] ).
```

En este último programa la clasificación de variables para la tercera cláusula (cuyo cuerpo es el bloque B) del predicado *permuta* queda de la forma siguiente:

$$\begin{array}{llll} Import(B) = \{L\} & Export(B) = \{[X, Xp \mid Np]\} & & \\ Import(B_1) = \{L\} & Export(B_1) = \{X, Lp\} & Pasc(B_1) = \{\} \\ Import(B_2) = \{Lp\} & Export(B_2) = \{Xp, Mp\} & Pasc(B_2) = \{X\} \\ Import(B_3) = \{Mp\} & Export(B_3) = \{Np\} & Pasc(B_3) = \{X, Xp\} \end{array}$$

Por lo que al transformarlo con el método de Tamaki se obtiene el programa:

```
% permutamak(L,LL,[]): LL es la lista de todas las permutaciones de la lista L.
permutamak(X1,Z0,Z3):-perm1(X1,Z0,Z1),perm2(X1,Z1,Z2),perm3(X1,Z2,Z3).

perm1([],Z0,Z1):-!,Z0=[[]|Z1].
perm1(_,Z0,Z1):-!,Z0=Z1.

perm2([X],Z0,Z1):-!,Z0=[[X]|Z1].
perm2(_,Z0,Z1):-!,Z0=Z1.

perm3(L,Z0,Z1):-!,deltamak(L,X,[]),i1(X,Z0,Z1).
perm3(_,Z0,Z1):-!,Z0=Z1.

i1([[X,Lp]|Y],Z0,Z2):-!,deltamak(Lp,YY,[]),i2(X,YY,Z0,Z1),i1(Y,Z1,Z2).
i1([],Z0,Z1):-!,Z0=Z1.
i1([_|Y],Z0,Z1):-!,i1(Y,Z0,Z1).

i2(X,[[Xp,Mp]|Y],Z0,Z2):-!,permutamak(Mp,YY,[]),i3(X,Xp,YY,Z0,Z1),i2(X,Y,Z1,Z2).
i2(_,[],Z0,Z1):-!,Z0=Z1.
i2(X,[_|Y],Z0,Z1):-!,i2(X,Y,Z0,Z1).

i3(X,Xp,[Np|Y],Z0,Z2):-!,Z0=[[X,Xp|Np]|Z1],i3(X,Xp,Y,Z1,Z2).
i3(_,_,[],Z0,Z1):-!,Z0=Z1.
i3(X,Xp,[_|Y],Z0,Z1):-!,i3(X,Xp,Y,Z0,Z1).

deltamak(X1,Z0,Z2):-del1(X1,Z0,Z1),del2(X1,Z1,Z2).
```

```
del1([H|T],Z0,Z1):-!,Z0=[[H,T]|Z1].
del1(_,Z0,Z1):-!,Z0=Z1.

del2([H|T],Z0,Z1):-!,deltamak(T,X,[]),i4(H,X,Z0,Z1).
del2(_,Z0,Z1):-!,Z0=Z1.

i4(H,[[L,T2]|X],Z0,Z2):-!,Z0=[[L,[H|T2]]|Z1],i4(H,X,Z1,Z2).
i4(_,[],Z0,Z1):-!,Z0=Z1.
i4(H,[_|X],Z0,Z1):-!,i4(H,X,Z0,Z1).
```

Si se realiza evaluación parcial y se eliminan cláusulas espurias se obtiene:

```
% permutamak(L,LL,[]): LL es la lista de todas las permutaciones de la lista L.
permutamak(X1,Z0,Z3):-perm1(X1,Z0,Z1),perm2(X1,Z1,Z2),perm3(X1,Z2,Z3).

perm1([],[[]|Z1],Z1):-!.
perm1(X,Z0,Z0).

perm2([X],[[X]|Z1],Z1):-!.
perm2(X,Z0,Z0).

perm3(L,Z0,Z1):-deltamak(L,X,[]),i1(X,Z0,Z1).

i1([[X,Lp]|Y],Z0,Z2):-deltamak(Lp,YY,[]),i2(X,YY,Z0,Z1),i1(Y,Z1,Z2).
i1([],Z0,Z0).

i2(X,[[Xp,Np]|Y],Z0,Z2):-permutamak(Np,YY,[]),i3(X,Xp,YY,Z0,Z1),i2(X,Y,Z1,Z2).
i2(_,[],Z0,Z0).

i3(X,Xp,[Np|Y],[[X,Xp|Np]|Z1],Z2):-i3(X,Xp,Y,Z1,Z2).
i3(_,_,[],Z0,Z0).

deltamak(X1,Z0,Z2):-del1(X1,Z0,Z1),del2(X1,Z1,Z2).

del1([H|T],[[H,T]|Z1],Z1):-!.
del1(X,Z0,Z0).

del2([H|T],Z0,Z1):-!,deltamak(T,X,[]),i4(H,X,Z0,Z1).
del2(X,Z0,Z0).

i4(H,[[L,T2]|X],[[L,[H|T2]]|Z1],Z2):-i4(H,X,Z1,Z2).
i4(_,[],Z0,Z0).
```

Y al introducir el predicado noteq para eliminar cortes[14]:

```
% permutamah(L,LL,[]): LL es la lista de todas las permutaciones de la lista L.
permutamah(X1,Z0,Z3):-perm1(X1,Z0,Z1),perm2(X1,Z1,Z2),perm3(X1,Z2,Z3).

perm1([],[[]|Z1],Z1).
perm1(X,Z0,Z0):- noteq(X,[]).

perm2([X],[[X]|Z1],Z1).
perm2(X,Z0,Z0):- noteq(X,[_]).

perm3(L,Z0,Z1):-deltamah(L,X,[]),i1(X,Z0,Z1).

i1([[X,Lp]|Y],Z0,Z3):-deltamah(Lp,YY,[]),i2(X,YY,Z0,Z1),i1(Y,Z1,Z3).
i1([],Z0,Z0).

i2(X,[[Xp,Mp]|Y],Z0,Z2):-permutamah(Mp,YY,[]),i3(X,Xp,YY,Z0,Z1),i2(X,Y,Z1,Z2).
i2(_,[],Z0,Z0).

i3(X,Xp,[Mp|Y],[[X,Xp|Mp]|Z1],Z2):-i3(X,Xp,Y,Z1,Z2).
i3(_,_,[],Z0,Z0).

deltamah(X1,Z0,Z2):-del1(X1,Z0,Z1),del2(X1,Z1,Z2).

del1([X|Y],[[X,T]|Z1],Z1).
del1(X,Z0,Z0):- noteq(X,[_|_]).

del2([X|T],Z0,Z1):-deltamah(T,X,[]),i4(X,X,Z0,Z1).
del2(X,Z0,Z0) :- noteq(X,[_|_]).

i4(X,[[L,T2]|X],[[L,[X|T2]]|Z1],Z2):-i4(X,X,Z1,Z2).
i4(_,[],Z0,Z0).

noteq(X,Y):-not(eq(X,Y)).
not(X):-X,!,fail.
not(X).
eq(X,X).
```

[14] La razón para eliminar los cortes es que el corte es una instrucción que sólo se necesita para transformar el programa original y una vez hecho esto se puede prescindir de él. Sin embargo, cada sistema se introducirá de nuevo para hacer más eficiente el programa final, ya que su uso se puede traducir en destartar ejecuciones innecesarias de algunas cláusulas.

4.2 Al adicionar Factorización al método de Tamaki se alcanza mayor eficiencia

La meta

```
?- permutamak([1,2,3,4],L,[]).
```

adicionada al programa resultante en la sección 4.1 dará como resultado:

```
L = [[1,2,3,4],[1,2,4,3],[1,3,2,4],[1,3,4,2],[1,4,2,3],[1,4,3,2]
,[2,1,3,4],[2,1,4,3],[2,3,1,4],[2,3,4,1],[2,4,1,3],[2,4,3,1]
,[3,1,2,4],[3,1,4,2],[3,2,1,4],[3,2,4,1],[3,4,1,2],[3,4,2,1]
,[4,1,2,3],[4,1,3,2],[4,2,1,3],[4,2,3,1],[4,3,1,2],[4,3,2,1]] ? ;

no
?-
```

Es decir, el método de Tamaki da como resultado un programa que arroja todas las respuestas de una pregunta.

Inherentemente todo método de búsqueda exhaustiva[15] lleva consigo computaciones redundantes al tener que recorrer el árbol de búsqueda para recolectar todas las respuestas.

Es precisamente en este punto que se puede vislumbrar una aplicación muy provechosa de la factorización. Es decir, un programa que conlleva computaciones redundantes puede hacerse más eficiente si le son detectadas algunas de tales redundancias para a continuación eliminarlas por factorización.

Con objeto de ilustrar las ideas anteriores considerese el programa resultante en la sección 4.1 al cual se le han renombrado sus variables para simplificar su manipulación:

```
permutamak(C, D, E) :-
        perm1(C, D, B),
        perm2(C, B, A),
        perm3(C, A, E).
perm1([], [[]|A], A).
perm1(A, B, B) :-
        noteq(A, []).
perm2([A], [[A]|B], B).
perm2(A, B, B) :-
        noteq(A, [_]).
```

[15]Cabe mencionar que aparte del método basado en el modelo de "flujos de ejecución" y dado por Tamaki, existe el dado por Ueda [18] que se basa en "continuaciones", y [1] expone una idea nueva a la que le llama "flujos estratificados".

```
perm3(B, C, D) :-
        deltemmb(B, A, []),
        i1(A, C, D).
i1([[D,C]|E], F, G) :-
        deltemmb(C, B, []),
        i2(D, B, F, A),
        i1(E, A, G).
i1([], A, A).
i2(F, [[D,C]|E], G, H) :-
        permtemmb(C, B, []),
        i3(F, D, B, G, A),
        i2(F, E, A, H).
i2(_, [], A, A).
i3(D, E, [B|C], [[D,E|B]|A], F) :-
        i3(D, E, C, A, F).
i3(_, _, [], A, A).
deltemmb(B, C, D) :-
        del1(B, C, A),
        del2(B, A, D).
del1([A|B], [[A,B]|C], C).
del1(A, B, B) :-
        notmq(A, [_|_]).
del2([B|C], D, E) :-
        deltemmb(C, A, []),
        i4(B, A, D, E).
del2(A, B, B) :-
        notmq(A, [_|_]).
i4(E, [[C,B]|D], [[C,[E|B]]|A], F) :-
        i4(E, D, A, F).
i4(_, [], A, A).
```

Al aplicar a este último programa las operaciones descritas en el apéndice se obtiene:

```
permtemmb(C, D, E) :-
        perm1(C, D, B),
        perm2(C, B, A),
        perm3(C, A, E).
perm1([], [[]|A], A) :- !.
perm1([_|_], A, A).
perm2([A], [[A]|B], B) :- !.
perm2(A, B, B).
i1([], A, A) :- !.
i1([[D,C]|E], F, G) :-
        deltemmb(C, B, []).
```

```
        i2(D, B, F, A),
        i1(E, A, G).
i2(_, [], A, A) :- !.
i2(F, [[D,C]|E], G, H) :-
        permutasi(C, B, []),
        i3(F, D, B, G, A),
        i2(F, E, A, H).
i3(_, _, [], A, A) :- !.
i3(D, E, [B|C], [[D,E|B]|A], F) :-
        i3(D, E, C, A, F).
deltasak(B, C, D) :-
        del1(B, C, A),
        del2(B, A, D).
del1([], A, A) :- !.
del1([A|B], [[A,B]|C], C).
del2([], B, B) :- !.
del2([B|C], D, E) :-
        deltasak(C, A, []),
        i4(B, A, D, E).
i4(_, [], A, A) :- !.
i4(E, [[C,B]|D], [[C,[E|B]]|A], F) :-
        i4(E, D, A, F).
perm3([], C, D) :- !,
        del2([], A, []),
        i1(A, C, D).
perm3([C], E, F) :- !,
        del2([], B, []),
        i4(C, B, A, []),
        i1([[C,[]]|A], E, F).
perm3([F,D], G, H) :- !,
        del2([], C, []),
        i4(D, C, B, []),
        i4(F, [[D,[]]|B], A, []),
        i1([[F,[D]]|A], G, H).
perm3([F1,E1,C1|D1], G1, H1) :-
        deltasak(D1, B1, []),
        i4(C1, B1, A1, []),
        i4(E1, A1, Z, []),
        i4(F1, Z, Y, []),
        permutasi([C1|D1], U, []),
        i3(F1, E1, U, G1, T),
        permutasi([E1|D1], S, []),
        i3(F1, C1, S, T, R),
        i2(F1, Z, R, Q),
```

```
i4(F1, A1, N, []).
i3(E1, F1, D, Q, L).
permutamb([F1|D1], K, []).
i3(E1, C1, K, L, J).
i2(E1, N, J, I).
i4(E1, B1, Q, []).
i4(F1, G, F, []).
i3(C1, F1, S, I, D).
i3(C1, E1, K, D, B).
i2(C1, F, B, A).
i1(Y, A, N1).
```

Y al adicionar la misma meta

```
?- permutamb([1,2,3,4],L,[]).
```

a este último programa, se obtendrá exactamente la misma salida de antes.

En general las transformaciones aplicadas preservan corrección. Es decir, para fines prácticos y en términos de las salidas producidas a partir de determinadas entradas, el programa original es equivalente al transformado.

Sin embargo, también para fines prácticos, existe una gran diferencia entre el desempeño de un programa y el otro.

La tabla 2 muestra los resultados obtenidos al calcular las permutaciones de 3, 4, 5, 6, 7, 8, y 9 elementos, utilizando la versión 3 del sistema *SICStus*.

SICStus 3	3	4	5	6	7	8	9
factimperm	0.002	0.007	0.018	0.187	0.838	11.025	77.72
imperm	0.004	0.010	0.047	0.300	2.079	17.784	188.78
find_all	0.003	0.014	0.068	0.422	3.376	33.480	*(Ejecución abortada)*
findall	0.005	0.010	0.087	0.429	3.180	32.640	*(Ejecución abortada)*

Tabla 2: Desempeños obtenidos al calcular permutaciones con programas diferentes.

En la tabla 2 se han incluido los desempeños obtenidos al calcular todas las permutaciones pasando como parámetro el predicado *permuta* al predicado *findall* que forma parte del sistema, así como pasando *permuta* al predicado *find_all* de usuario:

```
find_all(X, G, _) :- asserta(found(mark)), G,
asserta(found(X)), fail.
```

```
find_all(_, _, L) :- collect_found([], M), !, L = M.

collect_found(S, L) :- get_next(X), !, collect_found([X|S],L).
collect_found(L, L).

get_next(X) :- retract(found(X)), !, X \== mark.
```

La tabla muestra que el programa obtenido por el método de Tamaki es superior a hallar todas las respuestas utilizando *findall* o *find_all*. También se puede apreciar que el programa factorizado es aún superior a cualquiera de los otros.

La superioridad de *factmperm* es consecuencia de la eliminación de computaciones redundantes que, como antes se observó, son inherentes en todo programa de búsqueda exhaustiva.

5 Conclusiones

Se ha dado muestra de que se puede combinar Factorización con operaciones de Deducción parcial para obtener programas más eficientes. En este sentido es que se propone *Factorización como una operación más en Deducción parcial*.

Aunque se mostró en la sección 4.2 la utilidad de factorización al hacer más eficientes programas de búsqueda exhaustiva, su aplicación no se limita a estos últimos, sino que se aplica a todo programa que genere redundancias al tiempo de ejecución.

Claro que habrá programas que al tiempo de ejecución lleven a cabo cálculos redundantes y en los que tales cálculos no sean obvios con sólo observar el texto fuente, o aún más, tal vez los cálculos redundantes no estén completamente identificados ni aún conociendo la lógica del programa.

Como se apuntó en la sección 3.2, una posible solución sería implementar factorización en el sistema de inferencia. Sin embargo, como también se observó, tal "solución" produciría un sistema de inferencia ineficiente, pues el intentar factorizar dos átomos en una cláusula es una operación cuadrática.

La alternativa más viable es que dado un conocimiento a priori de la lógica del programa éste sea preprocesado de la forma descrita. Sin embargo, se debe tener presente que no se deben utilizar de forma no moderada las operaciones de Deducción parcial, pues el abuso en su utilización puede ser contraproducente.

El término medio es factorizar al tiempo de compilación no importando si puede o no resultar ineficiente esta operación, ya que sólo se ejecutará una vez. Sin embargo, como oportunamente se subrayó, tal sistematización es en su propio derecho materia de investigación aparte.

6 Trabajo a Futuro

La implementación del método propuesto sería en forma de una directiva al compilador, tal como `:-factorizar(predicado/aridad)`, y ésta debería comportarse de forma inteligente para no procesar en exceso al predicado de entrada con objeto de obtener una mejora real. De esta forma se puede, en principio, aplicar Factorización a cualquier predicado de cualquier programa lógico, el cual no debería ser modificado si no introduce redundancias computacionales.

Sin embargo, no tiene objeto aplicar Factorización a "ciegas", por lo que el método propuesto se puede adicionar, de forma natural, como una mejora al método de Tamaki. Esto significaría implementar el método de Tamaki por una parte, y Factorización por otra.

Debido a que el método de Tamaki recibe como entrada un programa en modo fijo, su implementación implica implementar el análisis de modos (Ueda [18] explica un método para hacer el análisis de modos), para a continuación implementar el método mismo.

Por otro lado, implementar Factorización significa codificar las operaciones de Transformación de Programas y el algoritmo de unificación. El algoritmo de unificación debe intentar unificar dos átomos del cuerpo de una cláusula previamente transformada. En caso de que no existan dos átomos unificables, se debe seguir procesando la cláusula, o hacer un retroceso y seguir un camino alternativo.

Aunque la implementación del análisis de modos, el método de Tamaki, las operaciones de Transformación de Programas, y el algoritmo de unificación, son en principio mecánicas, cada una podría tener complicaciones de entrada. Así por ejemplo, el método de Tamaki no es muy claro en una primera lectura, y el artículo contiene algunas notaciones confusas que habría que precisar.

Sin embargo, asumiendo que todas las implementaciones han sido realizadas, el problema fundamental es implementar un código que las utilice de forma inteligente para detectar y eliminar redundancias computacionales.

Este último problema se puede intentar resolver en dos etapas. Una primer etapa sería intentar sistematizarlo. La etapa siguiente sería considerablemente menos problemática y consistiría en implementar el método obtenido.

7 Apéndice

A continuación se describe el proceso por el que son detectadas y eliminadas las redundancias introducidas por *deltaweak* en el programa resultante de la sección 4.1.

Dado el programa fuente:

```
permtamnk(C, D, E) :-
        perm1(C, D, B),
        perm2(C, B, A),
        perm3(C, A, E).
perm1([], [[]|A], A).
perm1(A, B, B) :-
        noteq(A, []).
perm2([A], [[A]|B], B).
perm2(A, B, B) :-
        noteq(A, [_]).
perm3(B, C, D) :-
        deltamnk(B, A, []),
        i1(A, C, D).
i1([[D,C]|E], F, G) :-
        deltamnk(C, B, []),
        i2(D, B, F, A),
        i1(E, A, G).
i1([], A, A).
i2(F, [[D,C]|E], G, H) :-
        permtamnk(C, B, []),
        i3(F, D, B, G, A),
        i2(F, E, A, H).
i2(_, [], A, A).
i3(D, E, [B|C], [[D,E|B]|A], F) :-
        i3(D, E, C, A, F).
i3(_, _, [], A, A).
deltamnk(B, C, D) :-
        del1(B, C, A),
        del2(B, A, D).
del1([A|B], [[A,B]|C], C).
del1(A, B, B) :-
        noteq(A, [_|_]).
del2([B|C], D, E) :-
        deltamnk(C, A, []),
        i4(B, A, D, E).
del2(A, B, B) :-
```

```
        noteq(A, [_|_]).
i4(E, [[C,B]|D], [[C,[E|B]]|A], F) :-
        i4(E, D, A, F).
i4(_, [], A, A).
```

El objetivo es desdoblar la llamada a *deltamak* en el predicado *perm3* cuantas veces sea necesario para eliminar redundancias introducidas por tal llamada.

Así por ejemplo, al desdoblar la llamada a *deltamak* en *perm3* se obtendrán llamadas a *del1* y *del2*, y al desdoblar estas últimas resulta el nuevo predicado *perm3*:

```
perm3(B, C, D) :-
        noteq(B, [_|_]),
        del2(B, A, []),
        i1(A, C, D).
perm3([C|D], E, F) :-
        deltamak(D, B, []),
        i4(C, B, A, []),
        i1([[C,D]|A], E, F).
perm3([A|B], C, D) :-
        noteq([A|B], [_|_]),
        i1([[A,B]], C, D).
```

Repitiendo un par de veces más el paso anterior se obtiene el predicado *perm3*:

```
perm3(B, C, D) :-
        noteq(B, [_|_]),
        del2(B, A, []),
        i1(A, C, D).
perm3([A|B], C, D) :-
        noteq([A|B], [_|_]),
        i1([[A,B]], C, D).
perm3([C|D], E, F) :-
        noteq(D, [_|_]),
        del2(D, B, []),
        i4(C, B, A, []),
        i1([[C,D]|A], E, F).
perm3([D,B|C], E, F) :-
        noteq([B|C], [_|_]),
        i4(D, [[B,C]], A, []),
        i1([[D,[B|C]]|A], E, F).
perm3([F,D|E], G, H) :-
        noteq(E, [_|_]).
```

```
        del2(E, C, []).
        i4(D, C, B, []).
        i4(F, [[D,E]|B], A, []).
        i1([[F,[D|E]]|A], G, H).
perm3([H,G,E|F], I, J) :-
        deltamak(F, D, []).
        i4(E, D, C, []).
        i4(G, [[E,F]|C], B, []).
        i4(H, [[G,[E|F]]|B], A, []).
        i1([[H,[G,E|F]]|A], I, J).
perm3([F,E,C|D], G, H) :-
        noteq([C|D], [_|_]).
        i4(E, [[C,D]], B, []).
        i4(F, [[E,[C|D]]|B], A, []).
        i1([[F,[E,C|D]]|A], G, H).
```

Hasta aquí *perm3* ya obtiene los tres primeros elementos de la lista de entrada, y como ya no interesa detectar más redundancia ya no es necesario desdoblar a *deltamak*.

Sin embargo, todavía no se puede *factorizar* porque no se tienen varias submetas con el mismo símbolo de predicado y misma entrada.

Si ahora se desdobla la tercera submeta (átomo *i4*) de la cláusula *perm3* que llama a *deltamak* (la primera llamada a *i4* en tal cláusula no se desdobla porque no se desea que se instancie *D*), se obtiene:

```
perm3([H,G,E|F], I, J) :-
        deltamak(F, D, []).
        i4(E, D, C, []).
        i4(G, C, B, []).
        i4(H, [[G,[E|F]],[E,[G|F]]|B], A, []),
        i1([[H,[G,E|F]]|A], I, J).
```

La última llamada a *i4* tiene dos parejas en su segundo argumento, por lo que se desdoblará para cada argumento:

```
perm3([H,G,E|F], I, J) :-
        deltamak(F, D, []).
        i4(E, D, C, []).
        i4(G, C, B, []).
        i4(H, B, A, []).
        i1([[H,[G,E|F]],[G,[H,E|F]],[E,[H,G|F]]|A], I, J).
```

Se desdoblará *i1* para introducir una llamada más a *deltamak* el cual también se desdoblará:

```
perm3([K,J,H|I], L, M) :-
        deltamak(I, G, []),
        i4(H, G, F, []),
        i4(J, F, E, []),
        i4(K, E, D, []),
        deltamak([H|I], C, []),
        i4(J, C, B, []),
        i2(K, [[J,[H|I]]|B], L, A),
        i1([[J,[K,H|I]],[H,[K,J|I]]|D], A, M).
```

En este punto se sabe que la lista de entrada de la segunda llamada a *deltamak* tiene por lo menos un elemento por lo que se puede eliminar desdoblando tal llamada:

```
perm3([L,K,I|J], M, N) :-
        deltamak(J, M, []),
        i4(I, M, G, []),
        i4(K, G, F, []),
        i4(L, F, E, []),
        deltamak(J, D, []),
        i4(I, D, C, []),
        i4(K, [[I,J]|C], B, []),
        i2(L, [[K,[I|J]]|B], M, A),
        i1([[K,[L,I|J]],[I,[L,K|J]]|E], A, N).
```

deltamak ya no será desdoblada porque no se desea instanciar la lista, porque se correría el riesgo de borrarle elementos ya que no se sabe si es o no vacía (lo mismo ocurre con la penúltima llamada a *i4*). Sin embargo, la última llamada a *i4* si se puede desdoblar para a continuación desdoblar *i2* y así introducir una llamada a *permtamak*:

```
perm3([N,M,K|L], O, P) :-
        deltamak(L, J, []),
        i4(K, J, I, []),
        i4(M, I, H, []),
        i4(N, H, G, []),
        deltamak(L, F, []),
        i4(K, F, E, []),
        i4(M, E, D, []),
        permtamak([K|L], C, []),
        i3(N, M, C, O, B),
        i2(N, [[K,[M|L]]|D], B, A),
        i1([[M,[N,K|L]],[K,[N,M|L]]|G], A, P).
```

Si se vuelve a desdoblar *i2* se introduce una llamada más a *permtamak* para posteriormente ser factorizada:

```
perm3([P,O,N|N], Q, R) :-
        deltamak(N, L, []),
        i4(N, L, K, []),
        i4(O, K, J, []),
        i4(P, J, I, []),
        deltamak(N, M, []),
        i4(N, M, G, []),
        i4(O, G, F, []),
        permutamak([N|N], E, []),
        i3(P, O, E, Q, D),
        permutamak([O|N], C, []),
        i3(P, N, C, D, B),
        i2(P, F, B, A),
        i1([[O,[P,N|N]],[N,[P,O|N]]|I], A, R).
```

Se desdoblará *i1* para repetir el proceso con la pareja que aparece en su llamada. A continuación, por tanto se desdoblará dos veces *deltamak*:

```
perm3([T,S,Q|R], U, V) :-
        deltamak(R, P, []),
        i4(Q, P, O, []),
        i4(S, O, N, []),
        i4(T, N, M, []),
        deltamak(R, L, []),
        i4(Q, L, K, []),
        i4(S, K, J, []),
        permutamak([Q|R], I, []),
        i3(T, S, I, U, H),
        permutamak([S|R], G, []),
        i3(T, Q, G, H, F),
        i2(T, J, F, E),
        deltamak(R, D, []),
        i4(Q, D, C, []),
        i4(T, [[Q,R]|C], B, []),
        i2(S, [[T,[Q|R]]|B], E, A),
        i1([[Q,[T,S|R]]|M], A, V).
```

La última llamada a *deltamak* no se puede desdoblar porque tiene variables como argumentos, y lo mismo ocurre con la siguiente llamada a *i4*. La última llamada a *i4* sí se desdoblará y a continuación la última llamada a *i2* para introducir a *permdemak*:

```
perm3([V,U,S|T], W, X) :-
        deltamak(T, R, []),
        i4(S, R, Q, []),
```

```
        i4(U, Q, P, []).
        i4(V, P, O, []).
        deltamak(T, N, []).
        i4(S, N, M, []).
        i4(U, M, L, []).
        permutamak([S|T], K, []).
        i3(V, U, K, M, J).
        permutamak([U|T], I, []).
        i3(V, S, I, J, H).
        i2(V, L, H, G).
        deltamak(T, F, []).
        i4(S, F, E, []).
        i4(V, E, D, []).
        permutamak([S|T], C, []).
        i3(U, V, C, G, B).
        i2(U, [[S,[V|T]]|D], B, A).
        i1([[S,[V,U|T]]|O], A, Z).
```

Nuevamente se desdoblará la última llamada a i2 para introducir a *permutamak*:

```
perm3([X,W,U|V], Y, Z) :-
        deltamak(V, T, []).
        i4(U, T, S, []).
        i4(W, S, R, []).
        i4(X, R, Q, []).
        deltamak(V, P, []).
        i4(U, P, O, []).
        i4(W, O, N, []).
        permutamak([U|V], M, []).
        i3(X, W, M, Y, L).
        permutamak([W|V], K, []).
        i3(X, U, K, L, J).
        i2(X, M, J, I).
        deltamak(V, H, []).
        i4(U, H, G, []).
        i4(X, G, F, []).
        permutamak([U|V], E, []).
        i3(W, X, E, I, D).
        permutamak([X|V], C, []).
        i3(W, U, C, D, B).
        i2(W, F, B, A).
        i1([[U,[X,W|V]]|Q], A, Z).
```

Solo falta procesar la última pareja de i1, la última subunera. Al desdoblarla se introduce *deltamak*, y al desdoblar dos veces *deltamak*:

```
perm3([B1,A1,Y|Z], C1, D1) :-
        deltamak(Z, X, []),
        i4(Y, X, W, []),
        i4(A1, W, V, []),
        i4(B1, V, U, []),
        deltamak(Z, T, []),
        i4(Y, T, S, []),
        i4(A1, S, R, []),
        permutamak([Y|Z], Q, []),
        i3(B1, A1, Q, C1, P),
        permutamak([A1|Z], O, []),
        i3(B1, Y, O, P, N),
        i2(B1, R, M, N),
        deltamak(Z, L, []),
        i4(Y, L, K, []),
        i4(B1, K, J, []),
        permutamak([Y|Z], I, []),
        i3(A1, B1, I, M, N),
        permutamak([B1|Z], G, []),
        i3(A1, Y, G, M, F),
        i2(A1, J, F, E),
        deltamak(Z, D, []),
        i4(A1, D, C, []),
        i4(B1, [[A1,Z]|C], B, []),
        i2(Y, [[B1,[A1|Z]]|B], E, A),
        i1(U, A, D1).
```

Se desdoblará la última llamada a i4, luego la última llamada a i2 para introducir una llamada más a *permutamak*:

```
perm3([D1,C1,A1|B1], Z1, F1) :-
        deltamak(B1, Z, []),
        i4(A1, Z, Y, []),
        i4(C1, Y, X, []),
        i4(D1, X, W, []),
        deltamak(B1, V, []),
        i4(A1, V, U, []),
        i4(C1, U, T, []),
        permutamak([A1|B1], S, []),
        i3(D1, C1, S, Z1, R),
        permutamak([C1|B1], Q, []),
        i3(D1, A1, Q, R, P),
        i2(D1, T, P, O),
        deltamak(B1, N, []),
        i4(A1, N, M, []),
```

```
        i4(D1, M, L, []).
        permutamak([A1|B1], K, []).
        i3(C1, D1, K, O, J).
        permutamak([D1|B1], I, []).
        i3(C1, A1, I, J, N).
        i2(C1, L, N, G).
        deltamak(B1, F, []).
        i4(C1, F, E, []).
        i4(D1, E, D, []).
        permutamak([C1|B1], C, []).
        i3(A1, D1, C, G, B).
        i2(A1, [[C1,[D1|B1]]|D], B, A).
        i1(W, A, F1).
```

Por último se desdobla la última llamada a *i2* para introducir una última llamada a *permutamak*:

```
permak([F1,E1,C1|D1], G1, H1) :-
        deltamak(D1, B1, []).
        i4(C1, B1, A1, []).
        i4(E1, A1, Z, []).
        i4(F1, Z, Y, []).
        deltamak(D1, X, []).
        i4(C1, X, W, []).
        i4(E1, W, V, []).
        permutamak([C1|D1], U, []).
        i3(F1, E1, U, G1, T).
        permutamak([E1|D1], S, []).
        i3(F1, C1, S, T, R).
        i2(F1, V, R, Q).
        deltamak(D1, P, []).
        i4(C1, P, O, []).
        i4(F1, O, N, []).
        permutamak([C1|D1], M, []).
        i3(E1, F1, M, Q, L).
        permutamak([F1|D1], K, []).
        i3(E1, C1, K, L, J).
        i2(E1, M, J, I).
        deltamak(D1, H, []).
        i4(E1, H, G, []).
        i4(F1, G, F, []).
        permutamak([E1|D1], E, []).
        i3(C1, F1, E, I, D).
        permutamak([F1|D1], C, []).
        i3(C1, E1, C, D, B).
```

```
            i2(C1, F, B, A),
            i1(F, A, B1).
```

En esta última cláusula ya no es posible seguir aplicando transformaciones, por lo que se procuro de eliminar las redundancias detectadas a partir del programa resultado del proceso descrito:

```
    permutamak(C, D, E) :-
            perm1(C, D, B),
            perm2(C, B, A),
            perm3(C, A, E).
    perm1([], [[]|A], A).
    perm1(A, B, B) :-
            noteq(A, []).
    perm2([A], [[A]|B], B).
    perm2(A, B, B) :-
            noteq(A, [_]).
    i1([[D,C]|E], F, G) :-
            deltamak(C, E, [3).
            i2(D, E, F, A),
            i1(E, A, G).
    i1([], A, A).
    i2(F, [[D,C]|E], G, H) :-
            permutamak(C, E, [3).
            i3(F, D, E, G, A),
            i2(F, E, A, H).
    i2(_, [], A, A).
    i3(D, E, [D|C], [[D,E|B]|A], F) :-
            i2(D, E, C, A, F).
    i2(_, _, [], A, A).
    deltamak(B, C, D) :-
            del1(B, C, A),
            del2(B, A, D).
    del1([A|E], [[A,B]|C], C).
    del1(A, B, B) :-
            noteq(A, [_|_]).
    del2([B|C], D, E) :-
            deltamak(C, A, [3),
            i4(B, A, D, E).
    del2(A, B, B) :-
            noteq(A, [_|_]).
    i4(E, [[C,B]|D], [[C,[E|B]]|A], F) :-
            i4(E, D, A, F).
    i4(_, [], A, A).
    perm3(B, C, D) :-
```

```
        noteq(B, [_|_]).
        del2(B, A, []).
        i1(A, C, D).
perm3([A|B], C, D) :-
        noteq([A|B], [_|_]),
        i1([[A,B]], C, D).
perm3([C|D], E, F) :-
        noteq(D, [_|_]).
        del2(D, B, []).
        i4(C, B, A, []).
        i1([[C,D]|A], E, F).
perm3([D,B|C], E, F) :-
        noteq([B|C], [_|_]).
        i4(D, [B,C]], A, []).
        i1([[D,[B|C]]|A], E, F).
perm3([F,D|E], G, H) :-
        noteq(E, [_|_]).
        del2(E, C, []).
        i4(D, C, B, []).
        i4(F, [[D,E]|B], A, []).
        i1([[F,[D|E]]|A], G, H).
perm3([F,E,C|D], G, H) :-
        noteq([C|D], [_|_]).
        i4(E, [[C,D]], B, []).
        i4(F, [[E,[C|D]]|B], A, []).
        i1([[F,[E,C|D]]|A], G, H).
perm3([J,I,G|H], K, L) :-
        deltamak(H, F, []),
        i4(G, F, E, []).
        i4(I, E, D, []).
        i4(J, D, C, []).
        noteq([I,G|H], [_|_]).
        del2([I,G|H], B, []).
        i2(J, B, K, A).
        i1([[I,[J,G|H]],[C,[J,I|H]]|C], A, L).
perm3([I,H,F|G], J, K) :-
        deltamak(G, E, []),
        i4(F, E, D, []).
        i4(H, D, C, []).
        i4(I, C, B, []).
        noteq([H,F|G], [_|_]).
        i2(I, [[H,[F|G]]], J, A),
        i1([[H,[I,F|G]],[F,[I,H|G]]|B], A, K).
perm3([K,J,H|I], L, M) :-
```

```
        deltemmk(I, G, []).
        i4(H, G, F, []).
        i4(J, F, E, []).
        i4(K, E, D, []).
        notmq([H|I], [_|_]).
        del2([H|I], C, []).
        i4(J, C, B, []).
        i2(K, [[J,[H|I]]|B], L, A).
        i1([[J,[K,H|I]],[H,[K,J|I]]|D], A, M).
perm3([J,I,G|H], K, L) :-
        deltemmk(H, F, []).
        i4(G, F, E, []).
        i4(I, E, D, []).
        i4(J, D, C, []).
        notmq([G|H], [_|_]).
        i4(I, [[G,H]], B, []).
        i2(J, [[I,[G|H]]|B], K, A).
        i1([[I,[J,G|H]],[G,[J,I|H]]|C], A, L).
perm3([R,Q,G|P], S, T) :-
        deltemmk(P, H, []).
        i4(G, H, E, []).
        i4(Q, H, L, []).
        i4(R, L, E, []).
        deltemmk(P, J, []).
        i4(G, J, I, []).
        i4(Q, I, H, []).
        permtemmk([G|P], G, []).
        i3(R, Q, G, E, F).
        permtemmk([Q|P], E, []).
        i3(R, G, E, F, D).
        i2(R, H, D, C).
        notmq([R,G|P], [_|_]).
        del2([R,G|P], B, []).
        i2(Q, B, C, A).
        i1([[G,[R,Q|P]]|K], A, T).
perm3([Q,P,H|G], R, S) :-
        deltemmk(G, H, []).
        i4(H, M, L, []).
        i4(P, L, K, []).
        i4(Q, K, J, []).
        deltemmk(G, I, []).
        i4(H, I, H, []).
        i4(P, H, G, []).
        permtemmk([H|G], F, []).
```

```
        i3(Q, P, F, B, E),
        permtemak([P|O], D, []).
        i3(Q, M, D, E, C),
        i2(Q, G, C, B).
        noteq([Q,M|O], [_|_]),
        i2(P, [[Q,[M|O]]], B, A),
        i1([[M,[Q,P|O]]|J], A, E).
perm3([S,R,P|Q], T, U) :-
        deltemak(Q, O, []),
        i4(P, O, N, []),
        i4(R, N, M, []),
        i4(S, M, L, []),
        deltemak(Q, K, []),
        i4(P, K, J, []),
        i4(R, J, I, []),
        permtemak([P|Q], M, []),
        i3(S, R, M, T, G),
        permtemak([R|Q], F, []),
        i3(S, P, F, G, E),
        i2(E, I, E, D),
        noteq([P|Q], [_|_]),
        del2([P|Q], C, []),
        i4(S, C, B, []),
        i2(R, [[S,[P|Q]]|B], D, A),
        i1([[P,[S,R|Q]]|L], A, U).
perm3([R,Q,O|P], S, T) :-
        deltemak(P, N, []),
        i4(O, N, M, []),
        i4(Q, M, L, []),
        i4(R, L, K, []),
        deltemak(P, J, []),
        i4(O, J, I, []),
        i4(Q, I, N, []),
        permtemak([O|P], G, []),
        i3(R, Q, G, S, F),
        permtemak([O|P], E, []),
        i3(R, O, E, F, D),
        i2(R, N, D, C),
        noteq([O|P], [_|_]),
        i4(R, [[O,P]], B, []),
        i2(Q, [[R,[O|P]]|B], C, A),
        i1([[O,[R,Q|P]]|K], A, T).
perm3([Z,Y,W|X], A1, B1) :-
        deltemak(X, V, []),
```

```
        i4(W, V, U, []).
        i4(Y, U, T, []).
        i4(Z, T, S, []).
        delta4nk(X, R, []).
        i4(W, R, Q, []).
        i4(V, Q, P, []).
        permt4nk([W|X], O, []).
        i3(Z, Y, O, A1, N).
        permt4nk([Y|X], N, []).
        i3(Z, W, N, M, L).
        i2(Z, P, L, K).
        delta4nk(X, J, []).
        i4(W, J, I, []).
        i4(Z, I, N, []).
        permt4nk([W|X], O, []).
        i3(Y, Z, O, K, F).
        permt4nk([Z|X], M, []).
        i3(Y, W, K, F, D).
        i2(Y, M, D, C).
        notmq([Z,Y|X], [_|_]).
        del2([Z,Y|X], B, []).
        i2(W, B, C, A).
        i1(S, A, B1).
perm2([Y,Z,V|W], Z, A1) :-
        delta4nk(W, U, []).
        i4(V, U, T, []).
        i4(Z, T, S, []).
        i4(Y, S, R, []).
        delta4nk(W, Q, []).
        i4(V, Q, P, []).
        i4(Z, P, O, []).
        permt4nk([V|W], N, []).
        i3(Y, Z, N, Z, M).
        permt4nk([Z|W], L, []).
        i3(Y, V, L, M, K).
        i2(Y, O, K, J).
        delta4nk(W, I, []).
        i4(V, I, N, []).
        i4(V, N, G, []).
        permt4nk([V|W], F, []).
        i3(Z, Y, F, J, E).
        permt4nk([Y|W], D, []).
        i3(Z, V, D, E, C).
        i2(Z, G, C, B).
```

```
        noteq([Y,X|W], [_|_]),
        i2(V, [[Y,[X|W]]], B, A),
        i1(B, A, A1).
perm3([A1,Z,X|Y], B1, C1) :-
        deltemak(Y, W, []),
        i4(X, W, V, []),
        i4(Z, V, U, []),
        i4(A1, U, T, []),
        deltemak(Y, S, []),
        i4(X, S, R, []),
        i4(Z, R, Q, []),
        permtemak([X|Y], P, []),
        i3(A1, Z, P, B1, O),
        permtemak([Z|Y], N, []),
        i3(A1, X, N, O, M),
        i2(A1, Q, M, L),
        deltemak(Y, K, []),
        i4(X, K, J, []),
        i4(A1, J, I, []),
        permtemak([X|Y], N, []),
        i3(Z, A1, N, L, G),
        permtemak([A1|Y], F, []),
        i3(Z, X, F, G, E),
        i2(Z, I, E, D),
        noteq([Z|Y], [_|_]),
        del2([Z|Y], C, []),
        i4(A1, C, B, []),
        i2(X, [[A1,[Z|Y]]|B], D, A),
        i1(T, A, C1).
perm3([Z,Y,W|X], A1, B1) :-
        deltemak(X, V, []),
        i4(W, V, U, []),
        i4(Y, U, T, []),
        i4(Z, T, S, []),
        deltemak(X, R, []),
        i4(W, R, Q, []),
        i4(Y, Q, P, []),
        permtemak([W|X], O, []),
        i3(Z, Y, O, A1, N),
        permtemak([Y|X], M, []),
        i3(Z, W, M, N, L),
        i2(Z, P, L, K),
        deltemak(X, J, []),
        i4(W, J, I, []).
```

```
        14(Z, I, H, []).
        permutamk([V|Z], G, []).
        13(Y, Z, G, K, F).
        permutamk([Z|Z], E, []).
        13(Y, W, E, F, D).
        12(Y, H, D, C).
        noteq([Y|Z], [_|_]).
        14(Z, [[Y,X]], B, []).
        12(W, [[Z,[Y|Z]]|B], C, A).
        11(H, A, B1).
perm3([F1,E1,C1|D1], G1, H1) :-
        deltamak(D1, B1, []).
        14(C1, B1, A1, []).
        14(E1, A1, Z, []).
        14(F1, Z, Y, []).
        deltamak(D1, X, []).
        14(C1, X, W, []).
        14(E1, W, V, []).
        permutamk([C1|D1], W, []).
        13(F1, E1, U, G1, T).
        permutamk([E1|D1], S, []).
        13(F1, C1, S, T, R).
        12(F1, V, R, Q).
        deltamak(D1, P, []).
        14(C1, P, O, []).
        14(F1, O, N, []).
        permutamk([C1|D1], N, []).
        13(E1, F1, N, Q, L).
        permutamk([F1|D1], K, []).
        13(E1, C1, K, L, J).
        12(E1, N, J, I).
        deltamak(D1, M, []).
        14(E1, M, G, []).
        14(F1, G, F, []).
        permutamk([E1|D1], E, []).
        13(C1, F1, E, I, D).
        permutamk([F1|D1], C, []).
        13(C1, E1, C, D, B).
        12(C1, F, B, A).
        11(Y, A, H1).
```

Si una lista a tiene por lo menos un elemento, la submeta *notEq*(a,[_|_]) fallará, por lo que si tal submeta es parte del cuerpo de alguna cláusula, dicha cláusula fallará también. Esta observación conduce a una espectacular

reducción de cláusulas en el último programa, pues el predicado *perm3* tiene muchas cláusulas es esta situación:

```
permtemak(C, D, E) :-
        perm1(C, D, B),
        perm2(C, B, A),
        perm3(C, A, E).
perm1([], [C|A], A).
perm1(A, B, B) :-
        noteq(A, []).
perm2([A], [[A]|B], B).
perm2(A, B, B) :-
        noteq(A, [_]).
i1([[D,C]|E], F, G) :-
        deltemak(C, B, []),
        i2(D, B, F, A),
        i1(E, A, G).
i1([], A, A).
i2(F, [[D,C]|E], G, H) :-
        permtemak(C, B, []),
        i3(F, D, B, G, A),
        i2(F, E, A, H).
i2(_, [], A, A).
i3(D, E, [B|C], [[D,E|B]|A], F) :-
        i3(D, E, C, A, F).
i3(_, _, [], A, A).
deltemak(B, C, D) :-
        del1(B, C, A),
        del2(B, A, D).
del1([A|B], [[A,B]|C], C).
del1(A, B, B) :-
        noteq(A, [_|_]).
del2([B|C], D, E) :-
        deltemak(C, A, []),
        i4(B, A, D, E).
del2(A, B, B) :-
        noteq(A, [_|_]).
i4(E, [[C,B]|D], [[C,[E|B]]|A], F) :-
        i4(E, D, A, F).
i4(_, [], A, A).
perm3(B, C, D) :-
        noteq(B, [_|_]),
        del2(B, A, []),
        i1(A, C, D).
```

```
perm3([C|D], E, F) :-
        noteq(D, [_|_]).
        del2(D, B, []).
        i4(C, B, A, []).
        i1([[C,D]|A], E, F).
perm3([F,D|E], G, H) :-
        noteq(E, [_|_]).
        del2(E, C, []).
        i4(D, C, B, []).
        i4(F, [[D,E]|B], A, []).
        i1([[F,[D|E]]|A], G, H).
perm3([F1,E1,C1|D1], G1, H1) :-
        deltamak(D1, B1, []).
        i4(C1, B1, A1, []).
        i4(E1, A1, Z, []).
        i4(F1, Z, Y, []).
        deltamak(D1, X, []).
        i4(C1, X, W, []).
        i4(E1, W, V, []).
        permutamak([C1|D1], U, []).
        i3(F1, E1, U, G1, T).
        permutamak([E1|D1], S, []).
        i3(F1, C1, S, T, R).
        i2(F1, V, R, Q).
        deltamak(D1, P, []).
        i4(C1, P, O, []).
        i4(F1, O, N, []).
        permutamak([C1|D1], M, []).
        i3(E1, F1, M, Q, L).
        permutamak([F1|D1], K, []).
        i3(E1, C1, K, L, J).
        i2(E1, N, J, I).
        deltamak(D1, H, []).
        i4(E1, H, G, []).
        i4(F1, G, F, []).
        permutamak([E1|D1], E, []).
        i3(C1, F1, E, I, D).
        permutamak([F1|D1], C, []).
        i3(C1, E1, C, D, B).
        i2(C1, F, B, A).
        i1(Y, A, H1).
```

En la última cláusula (*perm3*) se pueden factorizar las cuatro llamadas a *deltamak*: las dos llamadas *permutamak*([*C1*|*D1*], *U*, []) y *permutamak*([*C1*|*D1*], *M*, [])

a *permtamak*; las dos llamadas *permtamak*([*E1*|*D1*], *S*, []) y *permtamak*([*E1*|*D1*], *E*, []) a *permtamak*; y las dos llamadas *permtamak*([*F1*|*D1*], *K*, []) y *permtamak*([*F1*|*D1*], *C*, []) a *permtamak*, resultando:

```
para3([F1,E1,C1|D1], G1, H1) :-
        deltamak(D1, B1, []),
        i4(C1, B1, A1, []),
        i4(E1, A1, Z, []),
        i4(F1, Z, Y, []),
        i4(C1, B1, W, []),
        i4(E1, W, V, []),
        permtamak([C1|D1], U, []),
        i3(F1, E1, U, G1, T),
        permtamak([E1|D1], S, []),
        i3(F1, C1, S, T, R),
        i2(F1, V, R, Q),
        i4(C1, B1, O, []),
        i4(F1, O, N, []),
        i3(E1, F1, U, Q, L),
        permtamak([F1|D1], K, []),
        i3(E1, C1, K, L, J),
        i2(E1, N, J, I),
        i4(E1, B1, G, []),
        i4(F1, G, F, []),
        i3(C1, F1, S, I, D),
        i3(C1, E1, K, D, B),
        i2(C1, F, B, A),
        i1(Y, A, H1).
```

También se pueden factorizar algunas llamadas a *i4*:

```
para3([F1,E1,C1|D1], G1, H1) :-
        deltamak(D1, B1, []),
        i4(C1, B1, A1, []),
        i4(E1, A1, Z, []),
        i4(F1, Z, Y, []),
        permtamak([C1|D1], U, []),
        i3(F1, E1, U, G1, T),
        permtamak([E1|D1], S, []),
        i3(F1, C1, S, T, R),
        i2(F1, Z, R, Q),
        i4(F1, A1, N, []),
        i3(E1, F1, U, Q, L),
        permtamak([F1|D1], K, []),
        i3(E1, C1, K, L, J),
```

```
        i2(E1, H, J, I),
        i4(E1, B1, G, []),
        i4(F1, G, F, []),
        i3(C1, F1, B, I, D),
        i3(C1, E1, K, D, B),
        i2(C1, F, B, A),
        i1(Y, A, H1).
```

Por último, si se elimina el predicado noteq introduciendo cortes y reordenando cláusulas, se tiene la versión final:

```
permutamk(C, D, E) :-
        perm1(C, D, B),
        perm2(C, B, A),
        perm3(C, A, E).
perm1([], [[]|A], A) :- !.
perm1([_|_], A, A).
perm2([A], [[A]|B], B) :- !.
perm2(A, B, B).
i1([], A, A) :- !.
i1([[D,C]|E], F, G) :-
        deltamnk(C, B, []),
        i2(D, B, F, A),
        i1(E, A, G).
i2(_, [], A, A) :- !.
i2(F, [[D,C]|E], G, H) :-
        permutamnk(C, B, []),
        i3(F, D, B, G, A),
        i2(F, E, A, H).
i3(_, _, [], A, A) :- !.
i3(D, E, [B|C], [[D,E|B]|A], F) :-
        i3(D, E, C, A, F).
deltamnk(B, C, D) :-
        del1(B, C, A),
        del2(B, A, D).
del1([], A, A) :- !.
del1([A|B], [[A,B]|C], C).
del2([], B, B) :- !.
del2([B|C], D, E) :-
        deltamnk(C, A, []),
        i4(B, A, D, E).
i4(_, [], A, A) :- !.
i4(E, [[C,B]|D], [[C,[E|B]]|A], F) :-
        i4(E, D, A, F).
perm3([], C, D) :- !.
```

```
        del2([], A, []).
        i1(A, C, D).
perm3([C], E, F) :- !.
        del2([], B, []).
        i4(C, B, A, []).
        i1([[C,[]]|A], E, F).
perm3([F,D], G, H) :- !.
        del2([], C, []).
        i4(D, C, B, []).
        i4(F, [[D,[]]|B], A, []).
        i1([[F,[D]]|A], G, H).
perm3([F1,E1,C1|D1], G1, H1) :-
        deltemmk(D1, B1, []).
        i4(C1, B1, A1, []).
        i4(E1, A1, Z, []).
        i4(F1, Z, Y, []).
        permtemmk([C1|D1], U, []).
        i3(F1, E1, U, G1, T).
        permtemmk([E1|D1], S, []).
        i3(F1, C1, S, T, R).
        i2(F1, Z, E, Q).
        i4(F1, A1, E, []).
        i3(E1, F1, U, Q, L).
        permtemmk([F1|D1], K, []).
        i3(E1, C1, K, L, J).
        i2(E1, E, J, I).
        i4(E1, B1, G, []).
        i4(F1, G, F, []).
        i3(C1, F1, S, I, D).
        i3(C1, E1, K, D, H).
        i2(C1, F, H, A).
        i1(Y, A, H1).
```

Referencias

[1] Akira, O. and Yuji, M., *Parallel Programming with Layered Streams*, ICOT Research Center, Institute for New Generation Computer Technology. IEEE 1987.

[2] Alberto Pettorossi, and Maurizio Proietti, *Rules and Strategies for Transforming Functional and Logic Programs*, ACM Computing Surveys, 28, 2 (June 1996), 360-414.

[3] Apt, K. R. and Van Emden, M. H., *Contributions to the Theory of Logic Programming*, JACM, 29, 3 (July 1982), 841-862.

[4] Clark, K. L., *Predicate Logic as a Computational Formalism*, Research Report 79/59, Department of Computing, Imperial College.

[5] Hayes, P. J., *Computation and deduction*. Proc. 2nd MFCS Symp., Czechoslovak Acad. of Sciences, 1973, pp. 105-118.

[6] Hill, R., *LUSH-Resolution and its Completeness*, DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

[7] Jones, N. D. and Mycroft, A. *Stepwise Development of Operational and Denotational Semantics for Prolog*. In *Proc. 1984 International Symposium on Logic Programming*, IEEE Computer Society, 1984, pp. 281-288.

[8] Kahn, K. *A partial evaluator of Lisp written in Prolog*. In *Proc. the First Logic Programming Conference*, Marseille, 1982.

[9] Kowalski, R. A., *Logic for Problem Solving*, Elsevier North Holland, New York, 1979.

[10] Kowalski, R. A., *Predicate Logic as a Programming Language*. IFIP 74, 569-574.

[11] Kowalski, R. A., *Algorithm = Logic + Control*, CACM, 22, 7 (Jul. 1979), 424-436.

[12] Kowalski, R. A. and Kuehner, D., *Linear Resolution with Selection Function*, Artificial Intelligence, 2(1971), 227-260.

[13] Lindstrom, G. and Panangaden, P. *Stream-based Execution of Logic Programming*. In *Proc. 1984 International Symposium on Logic Programming*, IEEE Computer Society, 1984, pp. 168-176.

[14] Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, Berlin Heidelberg New York Tokio, 1984.

Referencias

[1] Akira, O. and Yuji, M., *Parallel Programming with Layered Streams*, ICOT Research Center, Institute for New Generation Computer Technology. IEEE 1987.

[2] Alberto Pettorossi, and Maurizio Proietti, *Rules and Strategies for Transforming Functional and Logic Programs*, ACM Computing Surveys, 28, 2 (June 1996), 360-414.

[3] Apt, K. R. and Van Emden, M. H., *Contributions to the Theory of Logic Programming*, JACM, 29, 3 (July 1982), 841-862.

[4] Clark, K. L., *Predicate Logic as a Computational Formalism*, Research Report 79/59, Department of Computing, Imperial College.

[5] Hayes, P. J., *Computation and deduction*. Proc. 2nd MFCS Symp., Czechoslovak Acad. of Sciences, 1973, pp. 105-118.

[6] Hill, R., *LUSH-Resolution and its Completeness*, DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

[7] Jones, N. D. and Mycroft, A. *Stepwise Development of Operational and Denotational Semantics for Prolog*. In *Proc. 1984 International Symposium on Logic Programming*, IEEE Computer Society, 1984, pp. 281-288.

[8] Kahn, K. *A partial evaluator of Lisp written in Prolog*. In *Proc. the First Logic Programming Conference*, Marseille, 1982.

[9] Kowalski, R. A., *Logic for Problem Solving*, Elsevier North Holland, New York, 1979.

[10] Kowalski, R. A., *Predicate Logic as a Programming Language*. IFIP 74, 569-574.

[11] Kowalski, R. A., *Algorithm = Logic + Control*, CACM, 22, 7 (Jul. 1979), 424-436.

[12] Kowalski, R. A. and Kuehner, D., *Linear Resolution with Selection Function*, Artificial Intelligence, 2(1971), 227-260.

[13] Lindstrom, G. and Panangaden, P. *Stream-based Execution of Logic Programming*. In *Proc. 1984 International Symposium on Logic Programming*, IEEE Computer Society, 1984, pp. 168-176.

[14] Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, Berlin Heidelberg New York Tokio, 1984.

[15] Robinson, J. A., *A Machine-oriented Logic Based on the Resolution Principle*, JACM, 12, 1 (Jan. 1965), 23–41.

[16] Tamaki, H. *Stream-based Compilation of Ground I/O Prolog into Committed-choice Languages. Proc. the Fourth International Conference on Logic Programming*, J-L. Lassez (ed.), The MIT Press, pp. 376–393, 1987.

[17] Tamaki, H. and Sato, T. 1984. *Unfold/fold transformation of logic programs.* In *Proceedings of the Second International Conference on Logic Programming* (Uppsala, Sweden),127–138.

[18] Ueda, K., *Making Exhaustive Search Programs Deterministic*, New Generation Computing, 5(1987), 29–44. OHMSHA, LTD, and Springer-Verlag.