Una conexión entre Grámaticas Formales y Bases de Datos Deductivas

Julio César Peralta Estrada Carlos Rodrigo Silva Sortibrand

Universidad Nacional Autónoma de México
Facultad de Ingenìeria
Tesis para obtener el Título de
Ingeniero en Computación

Director de Tesis: Dr. David A. Rosenblueth

Ciudad Universitaria, D. F. Noviembre, 1992

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Contenido

Introducción

Programación Lógica y Bases de Datos Deductivas

En años recientes, los temas de Programación Lógica y Bases de Datos Deductivas han cobrado gran importancia. Un factor importante fué:

- La selección, por el Proyecto Japonés de la Quinta Generación, de Prolog y el Modelo de Datos Relacional como base para el desarrollo de nuevas arquitecturas.

Como resultado, se han generado algunas implementaciones prototipo de Bases de Datos Deductivas en diversas universidades de Europa, E.U. y Japón [CGT90].

Las Bases de Datos Deductivas surgen como una extensión del modelo relacional. El término Bases de Datos Deductiva resalta la habilidad de utilizar el estilo de la Programación Lógica para expresar deducciones concernientes al contenido de una base de datos. Estos nuevos sistemas nos ofrecen las siguientes ventajas:

1. Provee de un ambiente bastante expresivo para modelar datos.
2. A través de un solo lenguaje se expresa la base de datos, consultas, restricciones de integridad, vistas y programas.
3. Los fundamentos teóricos de las Bases de Datos Deductivas se basan en la Programación Lógica.
4. Resuelve consultas para programas recursivos (comunmente llamadas consultas recursivas).

Tradicionalmente las consultas pasan a través de un optimizador de consultas relacional. Este optimizador produce un plan para manipular grandes cantidades de datos almacenados en la base de datos, buscando un camino para que la consulta pueda ser evaluada de manera eficiente, es decir revisa en que orden se realizan las operaciones sobre la base de datos; sin embargo esta estrategia no es suficiente para resolver las consultas recursivas, por lo que algunos investigadores se han enfocado al estudio de estrategias para procesamiento de consultas recursivas [IIA92].

Como resultado de las investigaciones acerca del procesamiento de consultas recursivas, se han desarrollado varios métodos [BR88, RLK86, BSH91, HA92, SZ87, BR87], de los que resalta el denominado Conjuntos Mágicos, por su eficiencia. Posiblemente este método es llamado así por el misterio que rodea al principio en que está basado.

Programación Lógica y Gramáticas Formales

Los conceptos de Programación Lógica han sido aplicados, desde 1972, al procesamiento de lenguaje tanto natural como formal, principalmente a través del lenguaje Prolog.

Esto sigue vigente y como ejemplo tenemos que una parte importante del Proyecto de Quinta Generación Japonés, se encuentra el procesamiento de lenguaje natural, y dado que Prolog fue seleccionado para el proyecto, gran parte de las investigaciones en lingüística computacional han sido implantadas en él, debido a la gran facilidad de Prolog para representar gramáticas y analizadores sintácticos. Existe también un proyecto de procesamiento en lenguaje natural que se está realizando en forma conjunta por los paises de la Comunidad Económica Europea, también desarrollado en Prolog, lo cual nos da una idea de la importancia de Prolog como un lenguaje para el procesamiento de lenguaje natural.

En procesamiento de lenguaje natural, comúnmente nos interesa manipular símbolos (palabras, fonemas, partes de la oración) y objetos estructurados (secuencias, grafos) construidos a partir de símbolos.

Prolog es un lenguaje de alto nivel en el que se pueden expresar operaciones sobre símbolos (representados por términos simples) y estructuras (representadas por términos compuestos), sin tener que preocuparse de como se representan estos conceptos en la máquina; también permite expresar operaciones complejas de recuperación (inferencia) sobre la información básica [GM89].

El concepto de recursión juega un papel fundamental en el procesamiento de lenguaje natural. Los objetos lingüísticos se describen por estructuras de datos recursivas, y las operaciones sobre éstas, se expresan como algoritmos recursivos. A diferencia de otros lenguajes de programación de alto nivel, Prolog no tiene tantas restricciones en la definición de predicados que se llaman a sí mismos (directa o indirectamente), y así se pueden traducir los algoritmos de manera directa [GM89].

Bases de Datos Deductivas y Gramáticas Formales

En las secciones siguientes veremos que tanto las Bases de Datos Deductivas y las Gramáticas Formales tienen como factor común a la Programación Lógica, sin embargo no se ha explotado esta característica; por un lado, el estudio de Gramáticas Formales es un área clásica de la Ciencias de la Computación, mientras que Bases

de Datos Deductivas es una área reciente, la cual puede aprovechar los resultados de la anterior. La hipótesis del presente trabajo es mostrar que existe una conexión entre ambas áreas, para esto se toma un algoritmo de análisis sintáctico (Analizador Sintáctico Arriba-Abajo con Cartas), de Gramáticas Formales, y un método de procesamiento de consultas recursivas (Conjuntos Mágicos), por parte de Bases de Datos Deductivas. Como ejemplo se utiliza un programa recursivo y una consulta, se evalúa dicho programa con Conjuntos Mágicos. Este mismo ejemplo es evaluado con el Analizador Sintáctico con Cartas encontrando características comunes.

Descripción del trabajo

El trabajo ha sido dividido en cuatro capítulos, en el primero se hace una revisión de los conceptos de lógica de primer orden y resolución, necesarios para entender la terminología del resto del trabajo. Posteriormente se hace una introducción a Gramáticas Formales en el capítulo segundo, haciendo énfasis en análisis sintáctico con cartas. En el capítulo tercero hacemos referencia a conceptos de Bases de Datos Deductivas, con especial interés en el procesamiento de consultas recursivas utilizando el método de Conjuntos Mágicos. En el cuarto capítulo se realiza la transformación del Analizador Sintáctico Arriba-Abajo con Cartas, obteniendose Conjuntos Mágicos. Por último se expresan las conclusiones del trabajo.

Agradecimientos

Agradecemos al Dr. David A. Rosenblueth la oportunidad de trabajar con él, lo cual nos ha motivado de manera extraordinaria. Al Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas (IIMAS) de la U.N.A.M. por todas las facilidades que nos ha brindado para desarrollar nuestro trabajo.

Capítulo 1

Programación Lógica

En este capítulo revisamos algunos conceptos de Programación Lógica para tener una plataforma común a través de la cual se pueda establecer una conexión entre Gramáticas Formales y Bases de Datos Deductivas.

1.1 Conceptos de Programación Lógica

A continuación se definen de manera formal conceptos generales para manejar el lenguaje de la Programación Lógica. Gran parte de los conceptos mostrados en el capítulo se encuentran en [LloS7].

Un *término* puede ser:

- Una variable.
- $f(t_1,\ldots,t_n) \quad n \geq 0$ donde f es un símbolo de función n-ario y $t_1,\ldots,t_n$ son términos. A los símbolos de función de aridad 0 se les llama constantes; en este caso se omiten los paréntesis.

Si p es un símbolo de predicado y $t_1,\ldots,t_n$ son términos, entonces $p(t_1,\ldots,t_n)$ es una *fórmula atómica* o simplemente *átomo*.

Si F y G son fórmulas atómicas, entonces $(\sim F), F \wedge G, F \vee G, F \rightarrow G, F \leftrightarrow G$ también son fórmulas.

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

Una *cláusula* es una fórmula de la forma:

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

donde cada L_i es una literal y $x_1, \ldots, x_s$ son todas las variables que ocurren en $L_1 \vee \ldots \vee L_m$ [Llo87].

Dado que las cláusulas son muy comunes en Programación Lógica, se acostumbra adoptar una *notación especial para cláusulas*.

A cada cláusula de la forma:

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

donde $A_1, \ldots, A_k, B_1, \ldots, B_n$ son átomos y $x_1, \ldots, x_s$ son todas las variables que ocurren en los átomos, comúnmente se abrevia:

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

Por lo que se ve, esta notación asume que todas las variables están universalmente cuantificadas. Las comas en el *antecedente* $B_1, \ldots, B_n$ denotan conjunción, y las comas en el *consecuente* $A_1, \ldots, A_k$ denotan disyunción [Kow74].

Una *cláusula definida* es una cláusula de la forma:

$$A \leftarrow B_1, \ldots, B_n \quad (n \geq 0)$$

que contiene exactamente un átomo (A) en el consecuente. A es llamado la *cabeza* de la cláusula y $B_1, \ldots, B_n$ es llamada el *cuerpo* de la cláusula. De manera informal se interpreta como: A es cierto si B_1 y ... y B_n son ciertos.

Una *cláusula unitaria* es de la forma:

$$A \leftarrow$$

es decir, una cláusula definida con el cuerpo vacío, una cláusula incondicional. Una cláusula unitaria afirma que una *n-ada* pertenece a una relación.

Un *programa lógico* se define como un conjunto finito de cláusulas definidas.

En un programa lógico, el conjunto de todas las cláusulas del programa con el mismo símbolo de predicado p en la cabeza se denomina *definición* de p.

Una *consulta* es una cláusula de la forma:

$$\leftarrow B_1, \ldots, B_n \quad (n \geq 1)$$

esto es, una cláusula que tiene el consecuente vacío. Cada átomo B_i $(i = 1, \ldots, n)$ es llamado una *submeta* de la consulta. Intuitivamente se interpreta como: No es el caso de que B_1 y ... y B_n.

La *cláusula vacía* se denota por $\square$, es la cláusula con consecuente y antecedente vacío. Esta cláusula se entiende como una contradicción.

Una *cláusula de Horn* es una cláusula definida, una consulta o una cláusula vacía [Llo87].

Una *sustitución* θ es un conjunto finito de la forma:

$$\{V_1/t_1, \ldots, V_n/t_n\}$$

donde cada V_i es una variable, cada t_i, es un término distinto de V_i y las variables $V_1, \ldots, V_n$ son distintas.

Una *expresión* es un término, una literal, o la 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. Entonces $E\theta$, la *instancia* de E por θ, es la expresión que se obtiene al reemplazar simultáneamente cada ocurrencia de la variable V_i en E por el término t_i $(i = 1, \ldots, n)$.

A la sustitución $\{\}$ se le denomina *sustitución identidad* y se denota por ε. Nótese que Eε =E para toda expresión E.

Las propiedades elementales de las sustituciones son:
Sean θ, σ y γ sustituciones. Entonces

(a) $\theta\varepsilon = \varepsilon\theta = \theta$

(b) (Eθ)σ =E($\theta\sigma$), para toda expresión E.

(c) $(\theta\sigma)\gamma = \theta(\sigma\gamma)$.

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

Ejemplo: $p(f(X), a)$ y $p(Y, f(W))$ no unifican porque no existe ninguna sustitución θ tal que $\{a\}\theta$ y $\{f(W)\}\theta$ sean el mismo término.

Ejemplo: $p(f(X), Z)$ y $p(Y, a)$ sí unifican porque $\sigma = \{Y/f(a), X/a, Z/a\}$ es un unificador. El unificador más general es $\theta = \{Y/f(X), Z/a\}$. Nótese que $\sigma = \theta\{X/a\}$.

Se debe notar que utilizaremos mayúsculas para las variables, y minúsculas para símbolos de función y de predicado.

1.2 Resolución SLD

La resolución SLD es una regla de inferencia que se utiliza para programas lógicos, es una versión de la regla de inferencia denominada principio de resolución, descrito por J.A. Robinson en 1965. El principio de resolución se aplica a cláusulas de la forma: [Ns90]

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

Ya que las cláusulas definidas son un conjunto restringido de cláusulas (donde $m = 1$), su correspondiente forma restringida de resolución se presenta a continuación:

Sea G de la forma $\leftarrow A_1, \ldots, A_k$ y C de la forma $A \leftarrow B_1, \ldots, B_q$. Entonces G' es la *cláusula resolvente* de G y C utilizando el unificador más general θ si las siguientes condiciones se cumplen:

(a) A_m es un átomo en G.

(b) θ es el unificador más general de A_m y A.

(c) G' es la cláusula

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

La cláusula resolvente G' es una *consecuencia lógica* de G y C ($G, C \models G'$) [Llo87].

Ejemplo: Para el programa lógico:

(1) par(a,b) ←

(2) par(b,c) ←

(3) anc(X,Y) ← par(X,Y)

y la consulta ← anc(a,Y)

tenemos los siguientes pasos de resolución:

$\leftarrow$ anc(a,Y)
| (3) $\theta' = \{X/a\}$
$\leftarrow$ par(a,Y)
| (1) $\theta'' = \{Y/b\}$
□

La respuesta se obtiene al componer todas las sustituciones (θ' y θ'') aplicadas a la consulta.

La Programación Lógica puede verse como la combinación de la expresividad de las cláusulas definidas con la eficiencia de los probadores de teoremas o procedimiento de prueba, que emplean dichas cláusulas.

Un *probador de teoremas* está constituido por un sistema de reglas de inferencia y una estrategia de búsqueda.

Las *reglas de inferencia* especifican la forma de cada paso que constituye una prueba. Todos los posibles caminos de aplicar las reglas de inferencia, ya sea a un conjunto dado de cláusulas o las cláusulas derivadas de ellas determina el *espacio de búsqueda*, para el conjunto de cláusulas. Al especificar una *estrategia de búsqueda* sistemática para investigar cláusulas en el espacio de búsqueda, se determina un *procedimiento de prueba* [Kow79].

El lenguaje Prolog está basado en resolución SLD. Además existen otros sistemas de reglas de inferencia como por ejemplo el método llamado Conjuntos Mágicos.

Capítulo 2

Gramáticas Formales

En este capítulo se introducen los conceptos necesarios para poder estudiar las Gramáticas Formales desde el punto de vista de la Programación Lógica, teniendo especial interés en Gramáticas de Contexto Libre y Analizadores Sintácticos con Cartas.

2.1 Relación entre Programación Lógica y Gramáticas Formales

Uno de los proyectos de investigación que ha recibido mayor atención en la última década, es el denominado procesamiento de lenguaje natural, por lo que las investigaciones en la representación de lenguaje natural se han orientado básicamente a desarrollos en lingüística computacional, teniendo como principal herramienta a la Programación Lógica.

Cuando se escriben programas para procesar lenguaje natural, se necesita tener una definición segura (o una aproximación) del lenguaje natural, para saber cómo deberá comportarse el sistema bajo ciertas condiciones.

Un *lenguaje formal* se define como un conjunto de cuerdas, cuya pertenencia (al conjunto) queda determinada por reglas. El conjunto de cuerdas, en un lenguaje, puede ser infinito. Sin embargo, el número de reglas que definen a un lenguaje infinito, es finito.

Lo que se necesita son *sistemas matemáticos formales* que definan la pertenencia de los conjuntos infinitos, de cuerdas, y le asignen una estructura a cada miembro de este conjunto. A estos sistemas formales se les denomina *gramáticas* [GM89].

2.2 Definición de Gramática de Contexto Libre

Una *gramática de contexto libre*, G, queda definida por la cuarteta:

$$G = \langle NT, T, P, S \rangle$$

donde

NT: Símbolos no terminales

T: Símbolos terminales

P: Reglas o producciones de la forma $A \rightarrow w$ donde:

- $A \in NT$
- $w \in (T \cup NT)^*$

S: Símbolo inicial

además $T \cap NT = \emptyset$ y $S \in NT$ [HU79].

Ejemplo:

$$\begin{aligned} S &\rightarrow NP\ VP \\ VP &\rightarrow V\ NP \\ NP &\rightarrow \text{hombre} \\ NP &\rightarrow \text{Aristóteles} \\ V &\rightarrow \text{es} \end{aligned}$$

donde:

- $NT = \{S, NP, VP, V\}$
- $T = \{$Áristóteles,hombre,es$\}$
- $P = \{$

  $S \rightarrow NP\ VP,$
  $VP \rightarrow V\ NP,$
  $NP \rightarrow$ hombre,
  $NP \rightarrow$ Aristóteles,
  $V \rightarrow$ es
  $\}$

Una gramática tiene dos *funciones*:

- Definir los conjuntos de cuerdas en el lenguaje bajo consideración
- Asociar una o más estructuras a cada cuerda gramatical del lenguaje.

A estas estructuras se les representa con árboles, normalmente denominados *árboles de análisis sintáctico* [AHU74].
Para la cuerda, *Aristóteles es hombre*, se construye el árbol de la figura 2.1.

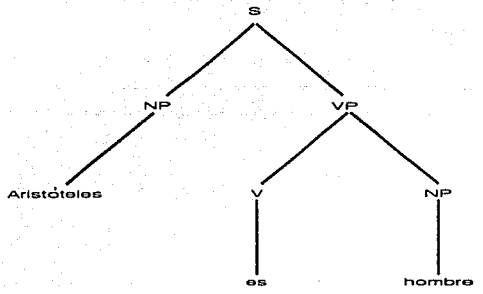

Figura 2.1: *Arbol de análisis sintáctico.*

Como las gramáticas están expresadas en un formalismo declarativo, sólo contienen información de qué objetos se combinan y cuáles son las propiedades del objeto resultante. Es por esto que su representación en Prolog resulta directa, por ser éste también de naturaleza declarativa.
Algunas restricciones notacionales en el formalismo de las gramáticas, pueden tener el efecto de limitar seriamente las clases de gramáticas que se pueden expresar; cambios mínimos en la notación pueden incrementar radicalmente el potencial matemático de los sistemas caracterizados [GMS9].
El tipo de regla sintáctica utilizada a lo largo de la siguiente exposición, es la *regla con estructura de frase*; ésto simplemente dice que el lado izquierdo (LHS) de la producción, puede componerse de los elementos del lado derecho (RHS), de esa misma producción [GM89].

Ejemplo:

$$S \rightarrow NP\ VP$$

donde:

$LHS = S$

$RHS = NP\ VP$

2.3 Analizadores Sintácticos

Es importante hacer diferencia entre el sistema de reglas de inferencia y la estrategia de búsqueda dentro de cualquier método de análisis. El primero construye el árbol de análisis (sistemas de reglas de inferencia); el segundo determina la forma en que se recorre este árbol (estrategia de búsqueda).

Los analizadores sintácticos abajo-arriba buscan construir el árbol de análisis sintáctico, a partir de las hojas; es decir, tratan de llegar al símbolo inicial sustituyendo la cuerda o partes de ella con el lado derecho de las producciones de la gramática.

Por otro lado, *los analizadores sintácticos arriba-abajo* trabajan igual que el sistema de reglas de inferencia de Prolog, es decir, van construyendo el árbol de análisis sintáctico de la raíz a las hojas, o del símbolo inicial a los símbolos terminales que constituyen la cuerda.

Ambos tipos de analizadores sintácticos son independientes de la estrategia de búsqueda. Y en ambos, cuando se busca que símbolo no terminal reescribe lo encontrado, se puede registrar una cantidad muy grande de intentos fallidos y al mismo tiempo se observa la duplicación de los intentos antes de tener éxito. Esto se debe principalmente a que el analizador no recuerda lo que ha hecho con anterioridad [GMS89].

En resumen se puede ver que:

- Un analizador sintáctico mapea cuerdas en estructuras
- Los analizadores abajo-arriba trabajan desde la cuerda, hacia el símbolo inicial (de las hojas a la raíz del árbol)
- Los analizadores arriba-abajo trabajan desde el símbolo inicial hasta encontrar la cuerda deseada (de la raíz a las hojas del árbol)
- Estos analizadores sintácticos simples no almacenan resultados intermedios
- Almacenar resultados intermedios es la clave para un análisis eficiente.

Los pruebas hechas sobre este tipo de analizadores, muestran su ineficiencia debido a que no recuerdan los resultados intermedios (ya sean fallidos o exitosos); y en el peor de los casos su ejecución se degrada hasta:

- Una demanda de espacio de memoria proporcional a la longitud de la cuerda a analizar [GM89].
- Un tiempo de ejecución exponencial en la longitud [GM89].

2.4 Analizadores Sintácticos con Cartas

Para mejorar el desempeño de los analizadores sintácticos, se utiliza una tabla con la cual se recuerdan resultados intermedios, evitando duplicar el trabajo.

Una *Tabla de Subcuerdas (TS)* es un mecanismo que le permite a un analizador sintáctico mantener o recordar estructuras que ya encontró, de tal forma que pueda evitar volver a buscarlas.

Una TS muestra los subárboles posibles y que parte de la cuerda representan, para cuerdas que no pertenecen (por completo) al lenguaje de una gramática.

Para analizar la cuerda: *Aristóteles es hombre*, se numeran desde el inicio hasta el final los espacios entre los símbolos: ${}_0Aristóteles_1 es_2 hombre_3$. Además se utiliza una tabla que contiene entradas i, j $(0 \leq i < j \leq 3)$ que dicen qué símbolos no terminales de la gramática se reescriben [1] entre estos números.

Si la tabla se representa con un árbol, las aristas del mismo quedarán marcados con el nombre del símbolo terminal y el símbolo no terminal al que reescriben.

Es importante notar que a pesar de que no se tenga una forma de analizar la cuerda completa, de acuerdo con las reglas de la gramática, se puede proporcionar información parcial, no un árbol de toda la cuerda, sino un conjunto de árboles para las partes de la cuerda que el analizador sintáctico pueda encontrar. De aquí se deriva la solución de las TS como una estructura que contiene un conjunto de árboles indicando todos los posibles análisis (ya sean totales o parciales), que se le puede hacer a la cuerda.

En lugar de listar todas las posibles formas en que se analiza la cuerda, se indican las trayectorias (dentro de la tabla) que muestran los posibles análisis entre dos puntos de la cuerda.

Ejemplo: Para la gramática

$$\begin{aligned} S &\rightarrow NP\ VP \\ VP &\rightarrow V\ NP \\ NP &\rightarrow \text{hombre} \\ NP &\rightarrow \text{Aristóteles} \\ V &\rightarrow \text{es} \end{aligned}$$

y la cuerda *Aristóteles es hombre*, se tiene la Tabla de Subcuerdas de la figura 2.2.

[1] La producción $A \rightarrow BC$, comunmente se interpreta como el símbolo A se reescribe como BC, es decir, toda aparición de A se intercambia por BC.

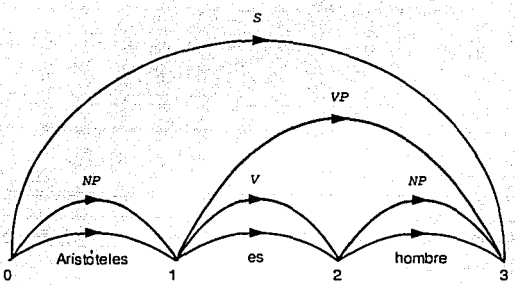

Figura 2.2: *Tabla de Subcuerdas.*

Las TS ahorran el trabajo de volver buscar, un símbolo no terminal, para un símbolo de la cuerda de entrada; pero no ahorran tiempo y trabajo para investigar hipótesis que con anterioridad fallaron.

Una forma de coordinar el trabajo y evitar intentos por duplicación, en el análisis, es tener una representación explícita de las distintas metas e hipótesis que adopta el analizador en un momento dado.

Como las hipótesis no se pueden representar con la estructura dada a las TS, entonces se le hacen dos modificaciones:

1. Se permitirán los ciclos que sean de una sola arista.
2. Pasar de las reglas simples en gramáticas, a *reglas marcadas* (con punto). De ésta forma, se puede ver cómo se va avanzando en la verificación de las hipótesis.

Ejemplo: Estos pueden ser nombres para las aristas de la *TS modificada*

$$S \rightarrow \cdot NP\ VP$$

$$S \rightarrow NP \cdot VP$$

$$S \rightarrow NP\ VP \cdot$$

Una TS más estas dos modificaciones normalmente se le denomina *carta* y a los analizadores que utilizan cartas se les llama *analizadores sintácticos con cartas.*

La extensión de TS a *Cartas* le permiten a un analizador sintáctico grabar información sobre metas que ha adoptado.

Las metas grabadas pueden no haber sido exitosas y todavía encontrarse bajo exploración.

Las aristas que representan hipótesis no confirmadas se les conoce como *aristas activas*; a los que representan hipótesis completamente confirmadas se les dice *aristas inactivas.*

Es evidente que las cartas con solo las aristas inactivas pueden representar lo mismo que una TS.

Una carta es un conjunto de tercias de la forma:

$$\langle inicio, fin, nombre \rightarrow encontrado \cdot por\ encontrar \rangle$$

Ejemplo:

$$\langle 0, 2, S \rightarrow NP \cdot VP \rangle$$
$$\langle 3, 5, NP \rightarrow Det\ N \cdot \rangle$$

Si la marca está al final de la producción, se trata de una *arista inactiva.*

Para poder construir la carta es necesario realizar una inicialización de la misma. Posteriormente se pueden aplicar dos reglas, una determina el sistema de reglas de inferencia (arriba-abajo o abajo-arriba), y la otra (regla fundamental) se aplica independientemente de las anteriores.

Regla Fundamental Si la carta contiene aristas del tipo $\langle i, j, A \rightarrow W_1 \cdot B\ W_2 \rangle$ y $\langle j, k, B \rightarrow W_3 \cdot \rangle$, donde A y B son símbolos no terminales y W_1, W_2, W_3 son secuencias (posiblemente vacías) de símbolos terminales o no terminales; entonces, adiciona la arista $\langle i, k, A \rightarrow W_1\ B \cdot W_2 \rangle$, a la carta.

Inicialización Es imposible aplicar la regla fundamental a una carta que no contiene aristas; se necesita por lo menos una arista activa y una inactiva. La tarea de asegurar que existan aristas inactivas, es la inicialización. Buscar los símbolos de la cuerda en el diccionario de la gramática, y encontrar que no terminales los reescriben. Los resultados de la búsqueda se colocan en la carta como aristas inactivas. Al hacer ésto estamos *construyendo la base* de la carta.

Regla abajo-arriba Si se adiciona una arista $\langle i, j, C \rightarrow W_1 \cdot \rangle$, a la carta (como producto de la inicialización o de aplicar la regla fundamental), entonces por cada regla de la gramática de la forma $B \rightarrow C\ W_2$, se agrega la arista $\langle i, i, B \rightarrow \cdot C\ W_2 \rangle$ a la carta.

Regla arriba-abajo Al inicializar[2], por cada regla $A \rightarrow W$, donde A es una categoría que puede expandir la carta, se agrega la arista $\langle 0, 0, A \rightarrow \cdot W \rangle$ a la carta. Si se adiciona una arista $\langle i, j, C \rightarrow W_1 \cdot B\ W_2 \rangle$ a la carta, entonces por cada regla $B \rightarrow W$, adiciona la arista $\langle j, j, B \rightarrow \cdot W \rangle$, a la carta.

La forma en que opera cada una de estas reglas se muestra en las siguientes figuras:

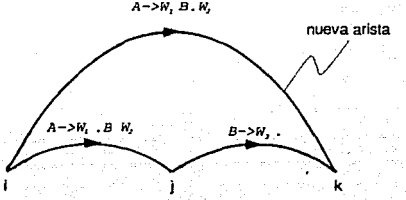

Figura 2.3: *Regla Fundamental.*

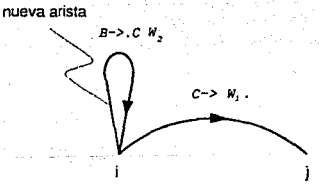

Figura 2.4: *Regla abajo-arriba.*

[2]Esta inicialización es propia de la regla arriba-abajo, ya que las aristas que se obtienen son aristas activas, a diferencia de la inicialización de la carta donde se obtienen aristas inactivas

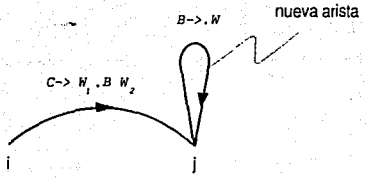

Figura 2.5: *Regla arriba-abajo.*

Un ejemplo de una carta generada utilizando la Regla Fundamental, la Regla Arriba-Abajo y la Inicialización, aplicadas a la gramática de la sección 2.4, y la cuerda *Aristóteles es hombre* se puede ver en la siguiente figura:

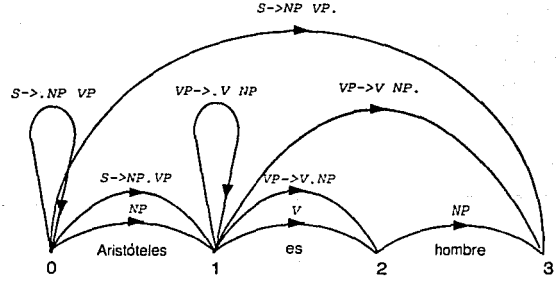

Figura 2.6: *Carta generada abajo-arriba*

Capítulo 3

Bases de Datos Deductivas

En este capítulo introducimos la terminología de las Bases de Datos Deductivas, dando una visión general de la evolución del modelo relacional hacia el modelo deductivo, es decir del cálculo relacional hacia la Programación Lógica. Poniendo especial interés en el procesamiento de consultas recursivas y la estrategia de Conjuntos Mágicos.

3.1 Introducción

A finales de los 70 y principios de los 80 se empezó a observar una fuerte interrelación entre la lógica formal y las bases de datos relacionales. Como resultado se ha desarrollado el modelo de Bases de Datos Deductivas que resalta la habilidad de utilizar la Programación Lógica, como lenguaje de consulta, para expresar deducciones con respecto al contenido de la Base de Datos.

La Programación Lógica y las Bases de Datos se han desarrollado de manera paralela. Por un lado Prolog, el lenguaje más popular de Programación Lógica, nace como simplificación de técnicas generales para demostración automática de teoremas, al volverla programable y eficiente.

De la misma forma, el Modelo de Datos Relacional nace como una simplificación de los modelos jerárquico y reticular, además de estar orientado a conjuntos [1], y poseer manipulación de datos no procedural. Uno de los principales objetivos de utilizar la Programación Lógica como lenguaje de consulta, es superar los obstáculos inherentes al modelo relacional. Entre los ejemplos más famosos, está el de la cerradura transitiva de una relación, que no puede ser expresada o consultada a través del cálculo relacional.

La investigación en Bases de Datos Deductivas ha tenido lugar principalmente en tres ámbitos. Primero, tener un desarrollo teórico firme. Las Bases de Datos Deductivas tienen una semántica formal contra la cual los algoritmos y sistemas pueden ser medidos. Segundo, el desarrollo de algoritmos de deducción rápidos y técnicas de op-

[1] Normalmente una consulta regresa un conjunto de respuestas

timización de consultas. Por último, la consistencia, que ha recibido menor atención; el desarrollo de métodos rápidos de revisión de consistencia se ha rezagado.

Como resultado de la investigación existen algunas implementaciones prototipo de Bases de Datos Deductivas:

- Proyecto LDL (Logic Data Language) de Microelectronics and Computer Technology Corp.
- Proyecto NAIL! (Not Another Implementation of Logic!) de la Universidad de Stanford.
- Sistema POSTGRES de la Universidad de Berkeley.
- Proyecto ALGRES del Politécnico de Milán.
- Proyecto PRISMA de la Universidad de Twente en Holanda.

Sin embargo, hasta donde sabemos, no hay productos comerciales de Bases de Datos Deductivas puras [CGT90].

3.2 Relación entre Programación Lógica y Bases de Datos

Para relacionar Bases de Datos y Programación Lógica se han considerado las siguientes características en común:

Bases de Datos.- Los sistemas convencionales basados en la Programación Lógica manejan bases de datos pequeñas, que se encuentran en memoria principal y a disposición de un solo usuario; éstas consisten en reglas de deducción y hechos.

Por el contrario, los sistemas de Bases de Datos, son enfocados a grandes volúmenes de información en memoria secundaria, compartidos y además ofrecen métodos eficientes para extraer, consultar o actualizar datos.

Consultas.- Una consulta denota el proceso a través del cual información relevante es extraída de la base de datos. En Programación Lógica, una consulta (o meta) es la respuesta dada por el encadenamiento de deducciones, que combina reglas y hechos; para probar o refutar la veracidad de la consulta.

En un sistema de Bases de Datos, una consulta (expresada a través de un lenguaje de manipulación de datos de propósito especial) es procesada para determinar la ruta más eficiente de acceso a los datos en memoria secundaria, para extraer la información relevante.

Restricciones.- Las restricciones especifican condiciones de integridad en la Base de Datos. El proceso a través del cual se conserva la correctez de la Base de Datos (para prevenir almacenar datos incorrectos) se le denomina *verificación de restricciones*. En Programación Lógica, las restricciones se expresan a través de reglas, que son utilizadas siempre que se modifica la Base de Datos.

En sistemas de Bases de Datos, únicamente se pueden expresar pocas restricciones utilizando el Lenguaje de Definición de Datos.

La Programación Lógica ofrece un gran poder para expresar consultas y restricciones, comparada con los lenguajes de manipulación y definición de datos en los sistemas de Bases de Datos. Además, la representación de consultas y restricciones es posible con un solo formalismo y su evaluación requiere el mismo mecanismo de inferencia, permitiendo un razonamiento más sofisticado acerca del contenido de la Base de Datos. Por otro lado, los sistemas de Programación Lógica no proveen de la tecnología para manejar gran volumen de información, de manera compartida y confiable.

La extensión de Programación Lógica y de Bases de Datos, consiste en construir nuevas clases de sistemas que tienen lugar en la intersección de los dos campos. Estos sistemas utilizan la Programación Lógica, para formular consultas y restricciones, y la tecnología de Bases de Datos para administrar eficientemente los datos [CGT90].

3.3 Conceptos de Bases de Datos Deductivas

Una Base de Datos Deductiva puede ser dividida en un *Base de Datos Extensional (EDB)* y un *Base de Datos Intensional (IDB)*. La EDB contiene todos las cláusulas unitarias sin variables (hechos), tal que $A \leftarrow$ es una cláusula de la Base de Datos Deductiva. Estas cláusulas están explícitamente como *n-adas* en la Base de Datos Deductiva. La IDB contiene todas las otras cláusulas definidas (reglas) que definen *n-adas* de la Base de Datos Deductiva de manera implícita [Sch91].

Además, las cláusulas de la EDB se agrupan dependiendo de su símbolo de predicado, y se introducen (como n-adas) en un sistemas de base de datos relacional.

Una IDB es un conjunto finito de reglas y la EDB es un conjunto finito de hechos, por tanto una Base de Datos Deductiva es un conjunto finito de relaciones.

Ejemplo: Dadas las tablas

PADRE

padre	*hijo*
juan	jorge
jorge	margarita
margarita	ana
juan	antonio
antonio	luis
maria	jorge
clara	luis
juana	pablo
maria	antonio
pablo	clara

PERSONA

nombre	*sexo*
pablo	masculino
juan	masculino
jorge	masculino
margarita	femenino
ana	femenino
antonio	masculino
luis	masculino
juana	femenino
maria	femenino
clara	femenino

Tabla 3.1: *Padre y Persona*

y las reglas:

papá(X,Y): se cumple sii X es papá de Y.
papá(X,Y) ← persona(X,masculino),padre(X,Y)

mamá(X,Y): se cumple sii X es mamá de Y.
mamá(X,Y) ← persona(X,femenino),padre(X,Y)

Decimos que las relaciones denotadas por los símbolos de predicado **padre** y **persona** pertenecen a la EDB; y tenemos la relaciones **papá** y **mamá** que pertenecen a la IDB.

A partir de las *n-adas* de la EDB y las reglas de la IDB podemos obtener las relaciones de la tabla 3.2.

Además podemos obtener otras relaciones de la EDB, a partir de las siguientes reglas de la IDB:

abuelo(X,Y): se cumple sii X es abuelo de Y.
abuelo(X,Z) ← padre(X,Y),padre(Y,Z)

hermano(X,Y): se cumple sii X es hermano de Y.
hermano(X,Y) ← padre(Z,X),padre(Z,Y),dif(X,Y)

PAPÁ

papá	*hijo*
juan	jorge
juan	antonio
jorge	margarita
antonio	luis
pablo	clara

MAMÁ

mamá	*hijo*
margarita	ana
clara	luis
maria	jorge
maria	antonio
juana	pablo

Tabla 3.2: *Papá y Mamá*

tío(X,Y): se cumple sii X es tío de Y.
tío(X,Y) ← persona(X,masculino),hermano(X,Z),padre(Z,Y)

Por tanto decimos que las relaciones abuelo, hermano y tío pertenecen a la IDB.

Un predicado que aparece en la IDB es un *predicado derivado* y además define una *relación derivada*. Un predicado que aparece en la EDB es un *predicado base* y define una *relación base*.

3.4 Estrategias de Evaluación de Consultas Recursivas

Las Bases de Datos Deductivas son una extensión de las bases de datos relacionales que permiten la definición de consultas recursivas. Por lo tanto, el problema de manejar consultas recursivas de manera correcta y eficiente es crucial para el desarrollo de las Bases de Datos Deductivas. Existen diferentes estrategias que solucionan algún aspecto del problema, a través de diferentes formalismos [BSH91, BR88].

3.4.1 Características de las Estrategias

Optimización de consultas vs. Evaluación de consultas

Podemos distinguir las estrategias de acuerdo a su objetivo. En un primer tipo se incluyen estrategias que reescriben programas, es decir a partir de un programa inicial y una consulta el programa es reescrito con la finalidad de hacerlo más eficiente; éstas son llamadas *estrategias de optimización*. Como ejemplo de estas estrategias tenemos *Aho-Ullman, Counting y Reverse Counting, Conjuntos Mágicos, Generalización de Conjuntos Mágicos, Método de Alejandro y Kifer-Lozinski*. Debemos notar que estas estrategias no evalúan la consulta en la base de datos. Las estrategias que realizan

la evaluación en la base de datos son llamadas *estrategias de evaluación* y son estas las que obtienen la respuesta a la consulta. Como ejemplo de este segundo tipo tenemos *Henschen-Naqvi, Query/Subquery (QSQ), APEX, Prolog, Evaluación Naive y Evaluación Semi-Naive* [BR88].

Siempre es posible utilizar únicamente una estrategia de evaluación, sin embargo la fase de optimización restringe el espacio de búsqueda y en algunos casos lo hace operable.

Ejemplo: Considérese el siguiente conjunto de reglas y hechos:

(1) ancestro(X,Y) ← padre(X,Y)

(2) ancestro(X,Y) ← ancestro(X,Z),padre(Z,Y)

(3) padre(a,b) ←

(4) padre(b,c) ←

y la consulta ← ancestro(a,X) que queda expresada como:

(5) query(X) ← ancestro(a,X)

Para obtener la respuesta a la consulta, es necesario aplicar una estrategia de evaluación, en este caso utilizamos la estrategia de evaluación Naive. Esta estrategia obtiene todas las posibles consecuencias lógicas del programa y se evalua abajo-arriba. Las primeras consecuencias obtenidas son las cláusulas unitarias, posteriormente aplicando *instanciación* y *Modus Ponens*, se obtienen las siguientes consecuencias, hasta que obtenemos todas las consecuencias del programa. Para nuestro ejemplo tenemos los siguientes pasos de evaluación:

Paso 0: { }

No se tiene ninguna consecuencia lógica

Paso 1: { *padre(a,b), padre(b,c)* }

Se agregan las cláusulas unitarias (3 y 4).

Paso 2: { padre(a,b), padre(b,c), *ancestro(a,b), ancestro(b,c)* }

A partir de la regla (1) y padre(a,b) se obtiene *ancestro(a,b)*. De manera similar se obtiene *ancestro(b,c)*.

Paso 3: { padre(a,b), padre(b,c), ancestro(a,b), ancestro(b,c), *ancestro(a,c), query(b)* }

A partir de la regla (2) y de, ancestro(a,b) y padre(b,c), se obtiene *ancestro(a,c)*. Además de la regla (5) y ancestro(a,b) se obtiene *query(b)*.

Paso 4: { padre(a,b), padre(b,c), ancestro(a,b), ancestro(b,c), ancestro(a,c), query(b), *query(c)* }

A partir de la regla (5) y ancestro(a,c) se obtiene por último *query(c)*.

A partir de las consecuencias obtenidas en el Paso 4, no se generan nuevas consecuencias lógicas por lo que la evaluación termina.

De lo anterior podemos observar que si utilizamos la estrategia de evaluación Naive de manera directa, obtenemos como resultado:

{ ancestro(a,b), ancestro(a,c), ancestro(b,c), padre(a,b), padre(b,c), query(b), query(c) }

Si antes de evaluar con la estrategia Naive aplicamos la estrategia de optimización de Conjuntos Mágicos, obtenemos:

{ mag.ancestro(a), mag.query, ancestro(a,b), ancestro(a,c), padre(a,b), padre(b,c), query(b), query(c) }

En este ejemplo, obtenemos mayor número de consecuencias, por ser demasiado simple, si en lugar de tener solo dos cláusulas unitarias (3 y 4), tuviéramos un número n de cláusulas de la forma:

$\text{par}(a_1,a_2)\leftarrow$

$\text{par}(a_2,a_3)\leftarrow$

$\vdots$

$\text{par}(a_{n-1},a_n)\leftarrow$

Por un lado al aplicar únicamente la estrategia de evaluación Naive obtendríamos:

$$m = \frac{n(n+1)}{2}$$

donde m es el número de consecuencias lógicas con símbolo de predicado *ancestro*, por otro lado al utilizar la estrategia de Conjuntos Mágicos antes de evaluar, obtenemos para el mismo programa:

$$m = n - 1$$

En nuestro ejemplo vemos que en el segundo caso se reduce el espacio de búsqueda, al obtener solo los ancestros de a (que corresponde a la consulta). Mientras que al utilizar solo la estrategia de evaluación se obtienen los ancestros, tanto de a como de b.

La estrategia de optimización de Conjuntos Mágicos se revisa al final de este capítulo.

Interpretación vs. Compilación

Podemos decir que una estrategia compila o interpreta. Comúnmente se considera que una estrategia compila si reescribe el programa de entrada, de este tipo son las estrategias de optimización.

Arriba-abajo vs. Abajo-arriba

Considérese el siguiente conjunto de reglas:

```
ancestro(X,Y) ← padre(X,Y)
ancestro(X,Y) ← ancestro(X,Z),padre(Z,Y)
```

y la consulta:

```
query(X) ← ancestro(juan,X)
```

Podemos ver cada una de estas reglas como producciones de una gramática de contexto libre. Desde este punto de vista, los predicados de la EDB (padre en este ejemplo) sería un *símbolo terminal*, y los predicados derivados o de la IDB (ancestro en este ejemplo) corresponderían a los *símbolos no terminales*. Finalmente, siguiendo con la analogía, decimos que el *símbolo inicial* es query(X). Por supuesto sabemos que esta analogía no es cierta completamente, por dos razones:

(i) La presencia de variables y constantes en las literales.

(ii) La falta de orden entre las literales de la regla (por ejemplo en la segunda regla padre(X,Z), ancestro(Z,Y) y ancestro(Z,Y), padre(X,Z) representan la misma relación).

Por el momento nosotros podemos ignorar estas diferencias, y utilizar la analogía informalmente.

Ahora consideremos el lenguaje generado por esta gramática.

{ padre(juan,X),
padre(juan,X) padre(X,X1),
padre(juan,X) padre(X,X1) padre(X1,X2),
...}

Este lenguaje tiene dos propiedades interesantes:

(i) Consiste en cuerdas que utilizan predicados base, por tanto pueden ser evaluados directamente en la EDB.

(ii) Si evaluamos cada cuerda de este lenguaje a través de la base de datos y tomamos la conjunción de todos esos resultados, entonces tenemos la respuesta a la consulta.

Existe un problema, que el lenguaje es infinito, y tenemos que evaluar un número infinito de predicados base. Para evitar esta dificultad, se utilizan algunas *condiciones de terminación*. Un ejemplo de una condición de terminación es: al evaluar cada una de las cuerdas generadas ya no existe una instancia de las literales, por tanto las cuerdas generadas posteriormente tampoco tendrán una instancia, por lo que no es necesario evaluarlas. Para el caso de nuestro ejemplo podemos decir, que al evaluar la primera cuerda, obtenemos la primera generación de juan, con la segunda cuerda obtendríamos la segunda generación, y así sucesivamente. Si por ejemplo al evaluar la quinta cuerda ya no existe instancia de las literales de la cuerda, entonces decimos que se ha cumplido la condición de terminación y por lo tanto ya no evaluamos la sexta cuerda y posteriores (es decir, ya no existen más generaciones de juan).

De hecho, todos los métodos de evaluación de consultas hacen lo siguiente:

(i) Generan el lenguaje.

(ii) Mientras el lenguaje es generado, se evalúan todos los símbolos de la cuerda.

(iii) A cada paso se revisa la condición de terminación.

Por lo tanto existen esencialmente dos clases de métodos: los que generan abajo-arriba el lenguaje y los que lo generan arriba-abajo. Las estrategias abajo-arriba empiezan por los símbolos terminales (las relaciones base) y van generando los no terminales (relaciones derivadas) hasta que generan el símbolo inicial (la consulta). La estrategia arriba-abajo inicia por el símbolo inicial (la consulta) y se va expandiendo, al aplicar las reglas, hacia los no terminales (relaciones derivadas) [BR88].

3.5 Conjuntos Mágicos

La idea de optimización de *Conjuntos Mágicos* consiste en transmitir o pasar las ligas entre variables, en otras palabras, se realiza un encadenamiento a través de las variables instanciadas con la finalidad de reducir el espacio de búsqueda.

Este método está basado en la idea de *sideway information passing* [Ull89a]. De manera intuitiva esta idea nos dice que, teniendo una cierta regla y una submeta en el cuerpo de la regla con algun(os) argumento(s) ligado(s), al resolver la submeta, se instancian las variables, y así reducimos el dominio de las relaciones representadas por las siguientes submetas. Así, al evaluar estas submetas ya tendrán instanciadas algunas de sus variables. Podemos decir que es una composición de relaciones [Ull89a].

El método de Conjuntos Mágicos compila, es evaluable abajo-arriba y corresponde a la fase de optimización, es decir, no evalúa directamente sobre la base de datos, sino únicamente reescribe las reglas para que su evaluación sea más eficiente o en algunos casos sea posible.

El método de Conjuntos Mágicos agrega predicados de restricción en el programa. Estos *predicados*, llamados *mágicos*, restringen el dominio de las variables. Durante la evaluación abajo-arriba, las variables solo toman algunos valores de todos los posibles. Normalmente esto hace al programa más eficientemente ejecutable.

3.5.1 Definiciones

Sea P un programa lógico, y Q una consulta sobre P. Si definimos Q como una secuencia de literales negativas, por ejemplo: $\leftarrow$ anc(a,X), ésta queda expresada por la cláusula query(X) $\leftarrow$ anc(a,X). Y se agrega al programa para especificar la consulta.

Sea $p(t_1, t_2, \ldots, t_n)$ un predicado n-ario. Un *adorno* de p es una secuencia a de longitud n de letras b o f [Ull89a]; por tanto bbf es un adorno para un predicado ternario, y $fbbf$ es un adorno para un predicado de aridad 4. Un adorno a se puede interpretar como sigue: la *i-ésima* variable de p es ligada o no ligada si el *i-ésimo* elemento de a es b (bound o ligado) o f (free o no ligado), respectivamente. Sea $p(t_1, t_2, \ldots, t_n)$ una literal, un adorno $a_1 a_2 \ldots a_n$ de una literal es un adorno de p tal que:

1. si t_i es una constante entonces a_i es b.
2. si $t_i = t_j$ entonces $a_i = a_j$.

[BR88].

Considere una regla r de P. Teniendo un adorno en la cabeza de la regla, un argumento de una submeta en r se dice que es *distinguido* si cumple alguna de las siguientes condiciones:

- Si el argumento es constante.

- Si el argumento es ligado.
- Si aparece en un predicado EDB que tiene un argumento distinguido.

Por lo tanto, un argumento es *distinguido* si el rango de los posibles valores que puede tomar es restringido por algunas constantes que aparecen en el mismo predicado, o por algunas variables, que toman un rango restringido de valores. Desde esta definición, si una variable es distinguida en un predicado EDB entonces todas las variables de ese predicado son distinguidas. Una ocurrencia de un predicado EDB con todas las variables distinguidas es un *predicado distinguido* [CGT90].

Ejemplo: Consideremos el siguiente programa y la consulta que se indica por r_0 :

anc(X,Y): se cumple sii X es el ancestro de Y.
padre(X,Y): se cumple sii X es el padre de Y.

r_0 : query(X) ← anc(a,X).
r_1 : anc(X,Y) ← padre(X,Y).
r_2 : anc(X,Y) ← anc(X,Z),padre(Z,Y).

En la regla r_0, al ser a una constante es distinguida en anc, mientras que X no lo es, porque se encuentra en un predicado IDB (quedando el adorno de anc como bf). Consideremos ahora la cabeza de la regla r_1 con los mismos adornos bf (estos adornos se pasan de r_0 a r_1 y r_2). Entonces, X es distinguido en padre, porque corresponde a una variable ligada en la cabeza; Z es también distinguida ya que padre es un predicado EDB y tiene un argumento distinguido. Por lo tanto, esta es una ocurrencia distinguida del predicado EDB padre.

Revisando la regla r_2, que también tiene el adorno bf, X es distinguido en anc porque este corresponde a una variable ligada en la cabeza; anc es un predicado IDB, por tanto no transmite la distinción a otros argumentos. Entonces no hay ningún argumento distinguido en el predicado padre, y se dice que esta no es una ocurrencia distinguida del predicado EDB padre [CGT90].

La formalización del concepto de argumentos distinguidos se da con la utilización de sideway information passing [BR87], que indica la propagación de ligas en las relaciones.

El programa adornado correspondiente al ejemplo anterior es:

r_0 : query.f(X) ← anc.bf(a,X)
r_1 : anc.bf(X,Y) ← padre(X,Y)
r_2 : anc.bf(X,Y) ← anc.bf(X,Z),padre(Z,Y)

A partir de un programa adornado, se aplica el algoritmo de conjuntos mágicos.

3.5.2 Algoritmo de Conjuntos Mágicos

A continuación se muestra el algoritmo de Conjuntos Mágicos en su forma original [BR87].

Se debe notar que por cada regla del programa original se genera una regla modificada y por cada predicado derivado del cuerpo de la regla se genera una regla mágica, por lo que el algoritmo se describe en dos partes:

Entrada: Una regla adornada.

Salida: Conjunto de reglas mágicas.

1. Para cada predicado derivado L de la regla adornada.
   (a) Borra todos los predicados derivados de la derecha de L.
   (b) Reemplaza el nombre del predicado por mag.L.a y borra todas las variables no distinguidas (donde a es el adorno del predicado L).
   (c) Borra todos los predicados base no distinguidos.
   (d) Borra todas la variables no distinguidas en la cabeza de la regla y reemplaza el nombre del predicado p por mag.p.
   (e) Intercambia los dos predicados mágicos.
   (f) Agrega la regla obtenida al conjunto de reglas mágicas.
2. Para obtener la regla modificada:
   (a) Para cada regla con símbolo p en la cabeza agrega a la derecha el predicado mag.p con únicamente los argumentos distinguidos en la ocurrencia de p.
3. Por último, se le agrega el *hecho mágico* con respecto a la pregunta (cuyo símbolo de predicado es p): mag.p, solo con los argumentos distinguidos.

3.6 Programa ancestro

A continuación se muestra el ejemplo del programa ancestro. Este ejemplo ha sido utilizado por varios autores para mostrar diferentes métodos de procesamiento de consultas recursivas [BR88, CGT90, Ull89b, HA92, RLK86, Loz92].

A partir del programa adornado de la sección anterior:

r_0: query.f(X) ← anc.bf(X,a)
r_1: anc.bf(X,Y) ← par(X,Y)
r_2: anc.bf(X,Y) ← anc.bf(X,Z),par(Z,Y)

si aplicamos el algoritmo a la regla r_0 tenemos:

1. Escoge un predicado derivado L: L = anc.bf(a,X).
2. Quita todos los predicados a la derecha de L.
3. Reemplaza el nombre del predicado por mag.anc.bf y borra las variables no distinguidas: q.f(X) ← mag.anc.bf(a).
4. Borra todos los predicados base no distinguidos.
5. Borra todas las variables no distinguidas del lado izquierdo y reemplaza el símbolo del predicado por mag.query.f:

   mag.query.f ← mag.anc.bf(a)
6. Intercambia los 2 predicados mágicos:
   mag.anc.bf(a) ← mag.query.f. obteniendo la regla mágica.

Para la regla modificada tenemos:

query.f(X) ← mag.query.f, anc(a,X)

Aplicando el algoritmo a las demás reglas y agregando el hecho mágico (rm_0) de acuerdo a la consulta tenemos:

rm_0: mag.query.f
rm_1: mag.anc.bf(a) ← mag.query.f
rm_2: query.f(X) ← mag.query.f, anc.bf(a,X)
rm_3: anc.bf(X,Y) ← mag.anc.bf(X), par(X,Y)
rm_4: mag.anc.bf(X) ← mag.anc.bf(X)
rm_5: anc.bf(X,Y) ← mag.anc.bf(X), anc.bf(X,Z), par(Z,Y)

Este programa puede simplificarse al resolver las reglas rm_1 y rm_2, con la regla rm_0, y suprimiendo la regla rm_4, que es redundante. En el siguiente capítulo presentamos una simplificación en la notación del algoritmo de Conjuntos Mágicos; con la idea de poder realizar la conexión con el Analizador Sintáctico con Cartas. Existen otros métodos semejantes a Conjuntos Mágicos como: Generalización de Conjuntos Mágicos [BR87], Método de Alejandro [RLK86] y OLDT [TS86, War92].

Capítulo 4

Conexión entre Conjuntos Mágicos y Cartas

El objetivo de este capítulo es establecer una relación entre Conjuntos Mágicos y Analizadores Sintácticos con Cartas. Algunos indicios que se tienen, para establecer la conexión, son:

- Las producciones de las gramáticas contexto libre son muy parecidas a las reglas de las bases de datos deductivas [AD89].
- Los Analizadores Sintácticos Arriba-Abajo con Cartas y la estrategia de optimización de Conjuntos Mágicos emplean métodos de programación dinámica [AHU74].

Para establecer la conexión es necesario realizar algunas *adaptaciones* a ambos métodos.

4.1 Conjuntos Mágicos

En el capítulo anterior se revisó el algoritmo de Conjuntos Mágicos como originalmente fue propuesto [BR88]. Sin embargo podemos hacer las siguientes simplificaciones:

- Se puede evitar el uso de adornos al hacer notar que los predicados son de *modo fijo*, es decir que los argumentos son únicamente de entrada o de salida, en cada predicado.
- De igual manera podemos tener predicados binarios de modo fijo, los cuales tienen una lista con los argumentos de entrada como primer término, y otra lista cuyos elementos son los argumentos de salida del predicado original como segundo término.

Estas simplificaciones se realizan con la finalidad de tener un *programa cadena*. Un programa cadena solo contiene cláusulas de la forma:

$$p_0(t_0, t_n) \leftarrow p_1(t_0, t_1), p_2(t_1, t_2), \ldots, p_n(t_{n-1}, t_n) \quad (n > 0)$$

donde las submetas p_j tienen la forma:

$$p_j(t_e, t_s)$$

siendo predicados binarios y de modo fijo, con t_e el término de entrada y t_s el término de salida, a cláusulas de esta forma se les llama *cláusulas cadena*. En los programas cadena las clausulas unitarias no contienen variables [BKBR90].

El algoritmo de conjuntos mágicos genera, a partir de una clausula cadena, el siguiente conjunto de reglas mágicas:

$mag.p_1(t_0) \leftarrow mag.p_0(t_0)$
$mag.p_2(t_1) \leftarrow mag.p_0(t_0), p_1(t_0, t_1)$
$mag.p_3(t_2) \leftarrow mag.p_0(t_0), p_1(t_0, t_1), p_2(t_1, t_2)$
$\vdots$
$mag.p_n(t_{n-1}) \leftarrow mag.p_0(t_0), p_1(t_0, t_1), p_2(t_1, t_2), \ldots, p_{n-1}(t_{n-2}, t_{n-1})$

Y su correspondiente regla modificada:

$p_0(t_0, t_n) \leftarrow mag.p_0(t_0), p_1(t_0, t_1), \ldots, p_n(t_{n-1}, t_n)$

Hay que notar que por cada predicado en el antecedente se genera una regla mágica, sin embargo en el caso de un predicado base, no se genera la regla mágica.

Por último hay que agregar un *hecho mágico* (al programa), con respecto a la consulta.

Para la consulta:

$\leftarrow q(t_e, t_s)$

se agrega el hecho:

$mag.q(t_e) \leftarrow$

4.2 Ejemplo de ancestro con Conjuntos Mágicos

Al utilizar las simplificaciones mencionadas en el punto anterior, el ejemplo de ancestro queda de la siguiente forma:

anc(X,Y) ← par(X,Y)
anc(X,Y) ← anc(X,Z),par(Z,Y)
par(a,b) ←
par(b,c) ←

y la consulta

← anc(a,X)

Utilizando la transformación de Conjuntos Mágicos :

anc(X,Y) ← mag.anc(X), par(X,Y)
mag.anc(X) ← mag.anc(X)
anc(X,Y) ← mag.anc(X), anc(X,Z), par(Z,Y)
par(a,b) ←
par(b,c) ←
mag.anc(a) ←

Evaluando abajo-arriba el programa transformado se obtienen las siguientes consecuencias (en cursivas se escriben los átomos generados en cada paso de evaluación):

Paso 0: { }

Paso 1: { *mag.anc(a), par(a,b), par(b,c)* }

Paso 2: { mag.anc(a), par(a,b), par(b,c), *anc(a,b)* }

Paso 3: { mag.anc(a), par(a,b), par(b,c), anc(a,b), *anc(a,c)* }

Las respuestas obtenidas a la consulta son las siguientes:

{ anc(a,b),anc(a,c) }

Hay que notar que al introducir los predicados mágicos se restringe la búsqueda de la respuesta. Los evaluadores abajo-arriba sin optimización obtienen todas las posibles respuestas y de éstas seleccionan las que coinciden con la consulta. El método de Conjuntos Mágicos restringe el espacio de búsqueda, con respecto a la consulta y además termina.

4.3 Analizadores Sintácticos con Cartas

Escribiendo el programa del Analizador Sintáctico con Cartas, para el evaluador abajo-arriba tenemos:

Inicialización con el símbolo inicial

edge(Str,Str,S,Cats) ← inicial(S),rule(S,Cats),cad(Str)

Inicialización de la carta

edge(E_k,E_f,Cat,[ ]) ← word(Cat,E_k,E_f)

Regla fundamental

edge(X,Z,Cat,Cats) ← edge(X,Y,Cat,[Cat1|Cats]),edge(Y,Z,Cat1,[ ])

Regla arriba-abajo

edge(Y,Y,Cat,Cats) ← edge(X,Y,Cat1,[Cat|Cats1]),rule(Cat,Cats)

Cada arista (edge) va a contener la parte de la producción que se va aplicar y el LHS de la producción, dos apuntadores a los símbolos de la cuerda que se está analizando. La subcuerda entre estos apuntadores corresponde a la parte de la producción que ha sido aplicada [Ros92].

4.4 Ejemplo de ancestro para Cartas

Reescribiendo el programa de ancestro para aplicar el Analizador Sintáctico Arriba-Abajo con Cartas:

$rule(C, [A_1, A_2, \ldots, A_n])$:se cumple sii

$$C(X_0, X_n) \leftarrow A_1(X_0, X_1), A_2(X_1, X_2), \ldots, A_n(X_{n-1}, X_n)$$

rule(anc,[par])←
rule(anc,[anc,par])←

word(C,X,Y): se cumple sii (X,Y) ∈ C

word(par,a,b)←
word(par,b,c)←

La especificación de la pregunta:

← anc(a,X)

En notación de Cartas queda determinado por:

inicial(anc)←
cad(a)←

Al utilizar el evaluador abajo-arriba se obtienen las siguientes consecuencias:

Paso 6: { cad(a), inicial(anc), rule(anc,[anc,par]), rule(anc,[par]), word(par,a,b), word(par,b,c), edge(a,a,anc,[anc,par]), edge(a,a,anc,[par]), edge(a,b,anc,[ ]), edge(a,b,anc,[par]), edge(a,b,par,[ ]), edge(a,c,anc,[ ]), edge(a,c,anc,[par]), edge(b,c,par,[ ]) }

Las respuesta obtenidas a la consulta son:

{ edge(a,b,anc,[ ]), edge(a,c,anc,[ ]) }

El significado intuitivo de las respuestas a la consulta (desde el punto de vista de análisis sintáctico) es que los pares {(a,b),(a,c)} pertenecen a la relación anc. Notamos además que cartas, al igual que conjuntos mágicos no computa todos los pares que pertenecen a la relación anc, sino solo los relacionados con la consulta (en este caso sólo los ancestro de a), en otras palabras restringe el espacio de búsqueda.

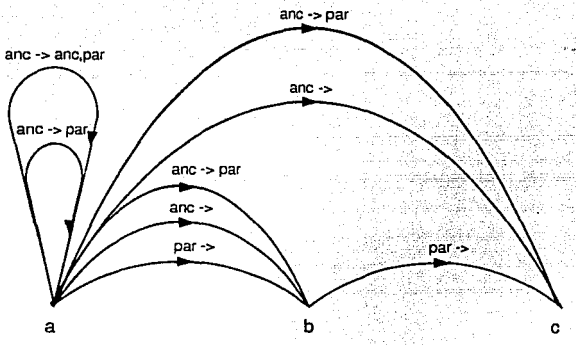

Figura 4.1: *Carta correspondiente al Paso 6.*

4.5 Especialización de Cartas a Conjuntos Mágicos

Comparando las consecuencias lógicas de ambos métodos, para el ejemplo de ancestro, podemos notar lo siguiente:

- Los predicados mágicos corresponden a las aristas activas vacías.
- Las aristas inactivas corresponden a las consecuencias del programa.
- Las cláusulas unitarias del programa corresponden la base de la carta.

Las semejanzas anteriores se aprecian a través de las consecuencias del programa, es decir, después de evaluarse en ambos métodos. Sin embargo, desde el punto de vista de ejecución, ambos métodos son diferentes.

Por un lado podemos ver que el Método de Conjuntos Mágicos, compila el programa, es decir lo reescribe, para posteriormente evaluar el programa reescrito (utilizando un evaluador abajo-arriba). Por otro lado el Analizador Sintáctico arriba-abajo con Cartas interpreta el programa (expresado a través de los meta-predicados rule y word), es decir se evalúan tanto el analizador como el programa.

Esto indica que para establecer la conexión entre ambos métodos, es necesario especializar el Analizador Sintáctico arriba-abajo con Cartas al programa objeto utilizando transformación de programas [Str87], utilizando pasos que conservan la correctez del programa.

Por lo tanto, a partir del Analizador Sintáctico Arriba-Abajo con Cartas y el programa ancestro, utilizando transformación de programas, se llega al programa de Conjuntos Mágicos, de la siguiente forma:

Analizador Sintáctico Arriba-Abajo con Cartas:

edge(X,Y,Cat,[Cat_1,Cat_2,...,Cat_n]): se cumple sii [1]

$$Cat \supseteq \{(X,Y)\} \circ Cat_1 \circ Cat_2 \circ \cdots \circ Cat_n$$

r_1: edge(X,X,Cat,Cats) ← inicial(Cat),rule(Cat,Cats),cad(X)
r_2: edge(X,Y,Cat,[ ]) ← word(Cat,X,Y)
r_3: edge(X,Z,Cat,Cats) ← edge(X,Y,Cat,[Cat1|Cats]),edge(Y,Z,Cat1,[ ])
r_4: edge(Y,Y,Cat,Cats) ← edge(X,Y,A,[Cat|B]),rule(Cat,Cats)

Consulta:

r_5: inicial(anc) ←
r_6: cad(a) ←

Programa ancestro:

rule(Cat,[Cat_1,Cat_2,...,C_n]): se cumple sii

$$Cat \supseteq Cat_1 \circ Cat_2 \circ \ldots \circ Cat_n$$

r_7: rule(anc,[anc,par]) ←
r_8: rule(anc,[par]) ←

word(Cat,X,Y): se cumple sii (X,Y) ∈ Cat

r_9: word(par,a,b) ←
r_{10}: word(par,b,c) ←

[1] Para las relaciones binarias R y S la *composición de las relaciones* R con S, que *se denota* $R \circ S$ queda determinada por:

$$R \circ S = \{(x,z) \mid \exists\, y \;\&\; (x,y) \in R \;\&\; (y,z) \in S\}$$

Resolviendo r_1 con r_5:

$\theta_1 = \{\text{Cat/anc}\}; t_1 = \theta_1 r_1$

t_1: edge(X,X,anc,Cats) ← rule(anc,Cats),cad(X)

Resolviendo t_1 con r_6:

$\theta_2 = \{\text{X/a}\}; t_2 = \theta_2 t_1$

t_2: edge(a,a,anc,Cats) ← rule(anc,Cats)

Resolviendo t_2 con r_7:

$\theta_3 = \{\text{Cats/[anc,par]}\}; t_3 = \theta_3 t_2$

t_3: edge(a,a,anc,[anc,par]) ←

Resolviendo t_2 con r_8:

$\theta_4 = \{\text{Cats/[par]}\}; t_4 = \theta_4 t_2$

t_4: edge(a,a,anc,[par]) ←

Aplicando θ_5 a r_3:

$\theta_5 = \{\text{X/Y,Cat/anc,Cat1/par,Cats/[ ]}\}; t_5 = \theta_5 r_3$

t_5: edge(Y,Z,anc,[ ]) ← edge(Y,Y,anc,[par]), edge(Y,Z,par,[ ])

Si tenemos:

r_3: edge(X,Z,Cat,Cats) ← <u>edge(X,Y,Cat,[Cat1|Cats])</u>, edge(Y,Z,Cat1,[ ])

r'_3: edge(A,D,C,Cs) ← edge(A,B,C,[C1|Cs]), edge(B,D,C1,[ ])

Resolviendo r_3 con r'_3 con respecto al átomo subrayado de r_3 y la cabeza de r'_3:

$\theta_6 = \{\text{A/X,D/Y,C/Cat,Cs/[Cat1|Cats]}\}$

t_6: edge(X,Z,Cat,Cats) ← edge(X,B,Cat,[C1|[Cat1|Cats]]), edge(B,Y,C1,[ ]), edge(Y,Z,Cat1,[ ])

Aplicando θ_7 a t_6:

θ_7 = {B/X,Cat/anc,C1/anc,Cat1/par,Cats/[ ]}; $t_7 = \theta_7 t_6$

t_7: edge(X,Z,anc,[ ]) ← edge(X,X,anc,[anc,par]), edge(X,Y,anc,[ ]), edge(Y,Z,par,[ ])

El programa resultante es:

t_3 : edge(a,a,anc,[anc,par]) .

t_4 : edge(a,a,anc,[par]) .

t_5 : edge(Y,Z,anc,[ ]) ← edge(Y,Y,anc,[par]), edge(Y,Z,par,[ ])

t_7 : edge(X,Z,anc,[ ]) ← edge(X,X,anc,[anc,par]), edge(X,Y,anc,[ ]), edge(Y,Z,par,[ ])

Definiendo:

par(X,Y) ↔ edge(X,Y,par,[ ])

anc(X,Y) ↔ edge(X,Y,anc,[ ])

mag.anc(X) ↔ edge(X,X,anc,z)

Tenemos:

m_1 : magic.anc(a).

m_2 : magic.anc(a).

m_3 : anc(Y,Z) ← magic.anc(Y), par(Y,Z)

m_4 : anc(X,Z) ← magic.anc(X), anc(X,Y), par(Y,Z)

Que es exactamente el mismo programa ancestro transformado por el método de Conjuntos Mágicos.

Por tanto, el método de Conjuntos Mágicos se puede ver como una versión compilada del Analizador Sintáctico con Cartas arriba-abajo.

La demostración anterior establece esta relación para el programa de ancestro. El caso de programas de modo fijo se estudia en [RPS92].

Conclusiones

La conexión establecida entre Conjuntos Mágicos y el Analizador Sintáctico Arriba-Abajo con Cartas, demostrada para el programa ancestro en el capítulo anterior, es válida tomando en cuenta varios aspectos:

- El método de Conjuntos Mágicos toma como entrada un programa y una consulta, el programa debe de cumplir las siguientes condiciones:

  a. Que sea de modo fijo

  b. Que las cláusulas unitarias no contengan variables.

  Por su parte, los Analizadores Sintácticos Arriba-Abajo con Cartas toman como entrada una gramática de contexto libre y una cuerda.

  Por lo tanto, para poder establecer la conexión en este aspecto, el programa de entrada de Conjuntos Mágicos debe ser expresado como gramática de contexto libre. Como los programas cadena si pueden ser expresados como gramáticas, el dominio de la conexión entre ambos métodos son los programas cadena.

- Desde el punto de vista de ejecución, Conjuntos Mágicos toma el programa de entrada, aplica la transformación y obtiene un programa resultante el cual es evaluado. Por lo tanto, decimos que Conjuntos Mágicos compila. En contraparte, el Analizador Sintáctico Arriba-Abajo con Cartas, a partir de evaluar la gramática y el analizador genera la cuerda (con respecto a la consulta), es decir las Cartas se ejecutan directamente, por lo que decimos que el Analizador Sintáctico Arriba-Abajo con Cartas interpreta. El analizador es el meta-intérprete y la gramática es el programa objeto; al especializar el meta-intérprete al programa (utilizando transformación de programas), se obtiene una versión compilada del programa y con las definiciones:

  $p(X,Y) \leftrightarrow edge(X,Y,p,[\,])$.

  $mag.p(X) \leftrightarrow edge(X,X,p,z)$.

  es igual al programa resultante de Conjuntos Mágicos.

A pesar de que la conexión establecida se restringe solamente a programas cadena, existe la posibilidad de ampliar el dominio de programas en la conexión, haciendo uso de la transformación de programas de modo fijo a programas composicionales [2], propuesta en [Ros91]. Además, debemos notar que a partir del Analizador Sintáctico Arriba-Abajo con Cartas y el programa ancestro obtenemos el mismo programa que Conjuntos Mágicos, utilizando transformación de programas. Esto sugiere la posibilidad de obtener un algoritmo que compile el analizador sintáctico y el programa objeto, antes de ser evaluado.

Se puede explotar la conexión establecida observando: Por un lado en análisis sintáctico se tiene un método alternativo al analizador arriba-abajo, el Analizador Sintáctico Abajo-Arriba con Cartas; uno presenta ventajas sobre otro dependiendo de la gramática. Por otro lado, en procesamiento de consultas recursivas se sugiere tener un método análogo al Analizador Sintáctico Abajo-Arriba con Cartas, como método alternativo a Conjuntos Mágicos; el cual se utilizaría dependiendo del programa. Esta posibilidad se estudia en [RPS92].

[2]Los programas cadena son un subconjunto propio de los programas composicionales. Los programas composicionales pueden tener cláusulas unitarias con variables

Bibliografía

[AD89] Harvey Abramson and Verónica Dahl. *Logic Grammars.* Springer-Verlag, 1989.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[BKBR90] Catriel Beeri, Paris Kanellakis, Fracois Bancilhon, and Raghu Ramakrishnan. Bounds on the propagation of selection into logic programs. *Journal of Computer and System Sciences*, 41(2):157–180, 1990.

[BR87] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. Sixth ACM Symposium on Principle of Database Systems*, pages 269–283, 1987.

[BR88] Francois Bancilhon and Raghu Ramakrishnan. An amateurs introduction to recursive query processing strategies. Technical Report 772, Computer Sciences Departament, University of Wisconsin, June 1988.

[BSH91] D.A. Bell, J. Shao, and M.E.C. Hull. A pipelined strategy for processing recursive queries in parallel. *Data & Knowledge Engineering*, (6):367–391, 1991.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

[GM89] Gerald Gazdar and Chris Mellish. *Natural Language Processing in Prolog. An Introduction to Computational Linguistics.* Addison-Wesley, 1989.

[HA92] Maurice A.W. Houtsma and Peter M.G. Apears. Algebraic optimization of recursive queries. *Data & Knowledge Engineering*, (7):299–325, 1992.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *Proc. of IFIP-74*, pages 569–574, 1974.

[Kow79] Robert Kowalski. *Logic for Problem-Solving.* Elsevier North-Holland, 1979.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[Loz92] Eliezer L. Lozinskii. Inference by generating in deductive databases. *Data & Knowledge Engineering*, (7):327–357, 1992.

[Ns90] Ulf Nilsson and Jan Malsuzyński. *Logic, Programming and Prolog.* John Wiley & Sons, 1990.

[RLK86] J. Rohmer, R. Lescoeur, and J.M. Kerisit. The alexander method-a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, (4):273–285, 1986.

[Ros91] David A. Rosenblueth. Fixed-mode logic programs as state-oriented programs. Technical Report Preimpreso No. 2, IIMAS, UNAM, 1991.

[Ros92] David A. Rosenblueth. Chart parsers as proof procedures for fixed-mode logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1125–1132, Tokyo, Japan, 1992.

[RPS92] David A. Rosenblueth, Julio C. Peralta, and Carlos Silva S. A grammatical reconstruction of the magic-sets method and an alternative method based on a parser. December 1992.

[Sch91] Helmut Schmidt. *Metalevel Control for Deductive Database Systems*, volume 479 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Str87] Paul A. Strooper. A transformation system for logic programs. Master's thesis, University of Waterloo, 1987.

[SZS7] Domenico Saccà and Carlo Zaniolo. Implementation of recursive queries for a data language based on pure horn logic. In *Proceedings of the Fourth International Logic Programming Conference*, pages 104–135. MIT Press, 1987.

[TS86] Hisao Tamaki and Taisuke Sato. Old resolution with tabulation. In *Proceedings of the Third International Logic Programming Conference*, pages 84–98. Springer-Verlag, 1986.

[Ull89a] Jeffrey D. Ullman. Bottom-up beats top-down for datalog. In *Proc. Eigth ACM Symposium on Principle of Database Systems*, pages 140–149, 1989.

[Ull89b] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems*, volume 2. Computer Science Press, 1989.

[War92] David S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):94–111, 1992.