11
2ej.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

NORMALIZACION DE REGLAS Y MANEJO DE REGLAS RECURSIVAS EN UNA BASE DE DATOS DEDUCTIVA

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO EN COMPUTACION

P R E S E N T A N

ANASTACIA YADIRA AVALOS VAZQUEZ

EDUARDO MARIA DE URIARTE OCCELLI

DIRECTOR DE TESIS

M. en C. CRISTOBAL JUAREZ CASTELLANOS

MEXICO, D.F. MAYO DE 1992

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Tabla de Contenido

Tabla de Figuras

Capítulo 1

INTRODUCCION

Esta tesis forma parte de un proyecto de diseño y desarrollo de un sistema para la programación de sistemas expertos, el cual integra técnicas de la inteligencia artificial y de las bases de datos, aspectos que se tratan en la primera sección de este capítulo.

Con el fin de enmarcar el trabajo desarrollado en esta tesis, en la sección 1.2 se presentan los antecedentes del proyecto. Los objetivos trazados para este trabajo se plantean en la sección 1.3. Existen algunos trabajos relacionados con este proyecto, los cuales se describen brevemente en la sección 1.4. Finalmente, en la sección 1.5 se desglosa la estructura de la tesis.

1.1 Inteligencia Artificial y Bases de Datos

La **inteligencia artificial** es una área del conocimiento que se encarga de los métodos que le permiten a la computadora resolver tareas para cuya solución se requiere inteligencia cuando esta se lleva a cabo por un ser humano[Mins68].

Aunque la inteligencia artificial y las bases de datos son dos áreas del conocimiento que han recibido un gran impulso en forma aislada, recientemente se ha demostrado que las técnicas desarrolladas en ambas áreas son aplicables tanto en una como en otra. Una de las áreas de más desarrollo de la inteligencia artificial es la enfocada a sistemas expertos o sistemas basados en el conocimiento.

Un **sistema experto** es un conjunto de programas de computadora que trata problemas para cuya resolución se requiere el conocimiento de un experto en el área. Estos sistemas se diseñan para resolver problemas en áreas específicas de aplicación.

Una **base de datos** es una colección de datos interrelacionados almacenados más o menos permanentemente en una computadora de tal manera que:

- Los datos son compartidos por diferentes usuarios y programas de aplicación, pero existe un mecanismo común para la inserción, actualización, borrado y consulta de los datos.

- Tanto los usuarios finales como los programas de aplicación no necesitan conocer los detalles de las estructuras de almacenamiento.

Un modelo de datos es una colección de conceptos (matemáticamente) bien definidos que ayudan a considerar y expresar las propiedades estáticas y dinámicas de una aplicación determinada. Existen diferentes modelos de datos, los modelos clásicos son:

- **Modelo jerárquico:** La estructura lógica en la cual se sustenta el modelo jerárquico es el árbol. Un árbol se compone de un nodo raíz y varios nodos sucesores, ordenados jerárquicamente. Cada nodo representa una entidad (objeto) y las relaciones entre entidades son las conexiones entre nodos, estas conexiones no dependen de la información contenida en los nodos sino que son fijas y se definen al inicio.

- **Modelo de red:** El modelo de red, a diferencia del jerárquico, permite cualquier conexión entre entidades, es decir, se pueden representar relaciones de muchos a muchos.

- **Modelo relacional:** El modelo relacional se basa en la representación de los objetos y sus interrelaciones a través de relaciones. Una relación es un conjunto de n-adas (tuplos), donde un tuplo representa una entidad. Estas entidades se caracterizan por un conjunto de atributos. Al igual que el modelo de red, es posible representar relaciones de muchos a muchos.

El modelo relacional destaca sobre los otros debido a que la representación de la información a través de relaciones, es más comprensible y sencilla de manejar.

Desde el punto de vista de la inteligencia artificial el diseño de un sistema experto puede verse inicialmente como la construcción de una base de conocimiento para la representación del dominio de discurso. Para la construcción de una base de conocimientos se requiere una notación para representar este conocimiento denominada **esquema de representación**. Existen diferentes esquemas de representación tales como:

- **Redes Semánticas:** Representan el conocimiento en términos de una colección de objetos (nodos) y asociaciones binarias. Desde este punto de vista, una base de conocimientos es una colección de objetos y de interrelaciones definidas sobre ellos, las modificaciones a la base de conocimientos se efectúan por la inserción o borrado de objetos y la manipulación de las interrelaciones.

- **Esquemas Procedurales:** En estos esquemas se considera que una base de conocimiento está constituida por una colección de agentes activos o procesos. Un ventaja de estos esquemas de representación consiste en permitir la especificación de interacciones descritas entre hechos eliminando búsquedas innecesarias, sin embargo, una base de conocimiento procedural es difícil de comprender y modificar.

- **Esquemas Basados en "Frames":** Un "frame" es una estructura de datos compleja para representar una situación estereotipada. Un "frame" consiste de ranuras ("slots") para los objetos que juegan un papel ("role") en la situación estereotipada, así como relaciones entre estos "slots".

- **Esquemas Lógicos:** Una base de conocimiento bajo este punto de vista, consiste de una colección de fórmulas lógicas. Los esquemas lógicos disponen de reglas de inferencia, las cuales se pueden emplear para definir procedimientos de prueba. Por medio de tales procedimientos se puede extraer información, realizar la verificación de integridad semántica y solucionar problemas. La simplicidad de la notación empleada facilita la comprensión de las bases de conocimiento y además cuenta con una semántica formal.

Aunque no puede decirse que un esquema de representación sea mejor que otro, el esquema lógico tiene importantes ventajas para la construcción de sistemas expertos. Una de estas ventajas es el formalismo matemático en el que se fundamenta este esquema y otra es el hecho de que cuenta con reglas de inferencia, las cuales pueden utilizarse, como ya se dijo, para definir procedimientos de prueba.

De manera informal, se dice que una **base de datos deductiva** es una base de datos en la cual se deducen nuevos hechos a partir de hechos almacenados explícitamente. Las bases de datos deductivas son una extensión de las bases de datos relacionales por medio de la lógica de predicados.

En la construcción de sistemas expertos, se tratan dos variantes del conocimiento: hechos y reglas de inferencia. Desde este punto de vista se pueden clasificar a los sistemas expertos en cuatro clases [JuAr90] :

1.- Con un conjunto reducido de hechos y un conjunto reducido de reglas.

2.- Con un conjunto reducido de hechos y un conjunto grande de reglas.

3.- Con un conjunto grande de hechos y un conjunto reducido de reglas.

4.- Con un conjunto grande de hechos y un conjunto grande de reglas.

Como las bases de datos están diseñadas para manipular grandes volúmenes de información, se puede utilizar un sistema de base de datos para manejar los casos *3* y *4*, de esta forma, es posible usar una base de datos deductiva para la construcción de un sistema experto.

El conocimiento necesario para los sistemas expertos se puede representar en un sistema de base de datos de la siguiente forma:

- Los hechos se expresan como reglas con el antecedente vacío y se almacenan en relaciones de la base de datos denominadas relaciones base.

- Las reglas de inferencia describen como se pueden deducir conocimientos nuevos de hechos almacenados explícitamente.

1.2 Antecedentes del Proyecto

El proyecto que dio origen a este trabajo tuvo sus inicios en Julio de 1988 en el Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas (IIMAS) de la Universidad Nacional Autónoma de México, estando a cargo del investigador Cristóbal Juárez.

El objetivo que se planteó fue diseñar un *"Sistema de Bases de Datos Deductivas para la Programación de Sistemas Expertos"*, el cual tiene como ventajas el manejo de grandes volúmenes de información a través del uso de bases de datos, adicionando la capacidad de deducción mediante el uso de reglas de inferencia. Este sistema permite tanto generar bases de datos relacionales convencionales como bases de datos deductivas. La arquitectura propuesta para este sistema se muestra en la figura 1.1, la cual incluye los siguientes componentes: interfaz del usuario, lenguaje orientado a reglas, parser, monitor de transacciones, subsistema de inferencia, base de reglas, restricciones de integridad, manejador del catálogo, manejador de bases de datos relacional (RDBMS) y base de datos [JuAr90].

El **lenguaje orientado a reglas** ofrece las ventajas de manejar los esquemas lógicos de deducción y representación del conocimiento, además de utilizar las facilidades ofrecidas por un sistema de bases de datos relacional para el almacenamiento y manipulación de hechos.

El **subsistema de inferencia** es el núcleo del sistema, el cual es una extensión de un sistema manejador de bases de datos relacional para el soporte de deducción y se integra de la siguiente manera:

- El **intérprete** es el componente más importante del sistema de inferencia debido a que su función es deducir conocimiento a partir de los hechos almacenados en las relaciones base. Dada una consulta, el intérprete genera un árbol de derivación el cual representa una expresión del álgebra relacional, la cual se evalúa para responder a dicha consulta.

- El **módulo explicativo** como su nombre lo indica, explica y justifica como se llega a una conclusión y permite construir una interfaz más amigable para el usuario final, el cual no necesariamente tiene que conocer las reglas.

- El **mecanismo de actualización** está considerado para la modificación de reglas (relaciones base, reglas de deducción y restricciones de integridad) debido a que en la realidad, los sistemas sufren cambios continuamente.

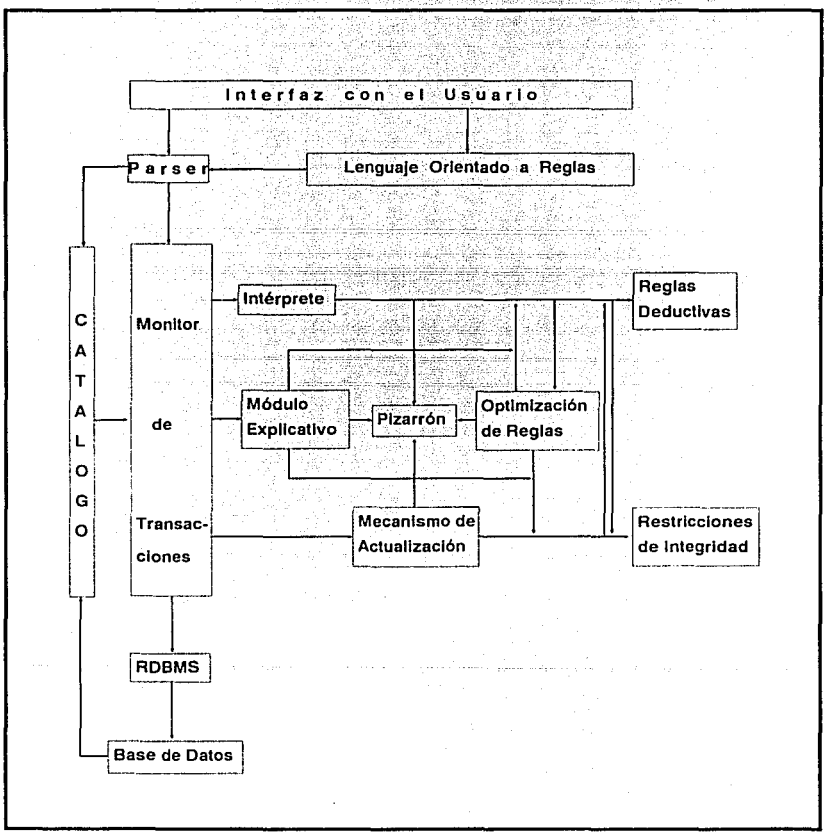


Figura 1.1: Arquitectura General de una Base de Datos Deductiva para la Programación de Sistemas Expertos.

- El **módulo de optimización** tiene como objetivo reducir los tiempos de respuesta debido a que se cuenta con un conjunto grande de hechos y un conjunto grande de reglas de deducción, lo que da como resultado un gran espacio de búsqueda durante los procesos deductivos. El principal punto de este módulo es mejorar el proceso de deducción, el cual contempla dos fases: la generación del árbol de derivación y la evaluación del mismo.

- El **pizarrón** es el área del sistema empleada para almacenar resultados intermedios. Estos resultados se emplean en el módulo explicativo para justificar las conclusiones.

El **monitor** transfiere el control al submódulo adecuado dependiendo de la operación que el sistema realiza.

El **catálogo** almacena información acerca de los objetos del sistema, es decir, funciones, relaciones, reglas y restricciones de integridad.

El primer prototipo para este sistema se desarrolló como tema de tesis en la Maestría en Ciencias de la Computación del IIMAS. Este primer prototipo, se limitó al desarrollo de un parser para realizar el análisis del lenguaje orientado a reglas, un intérprete el cual solo maneja un sistema de reglas normalizado y no recursivo y una interfaz con el manejador de la base de datos [Aran91].

1.3 Objetivos de la Tesis

El objetivo de esta tesis es desarrollar un segundo prototipo de una base de datos deductiva para la programación de sistemas expertos, el cual tendrá como base el primer prototipo creado y se le adicionarán las siguientes características:

1.- **Normalización de Reglas:** El sistema realizará un análisis para detectar si existen reglas a normalizar. En caso de que se trate de un sistema no normalizado, se realizarán las transformaciones necesarias en el sistema de reglas, de tal manera que el intérprete recibirá un sistema de reglas normalizado equivalente.

2.- **Detección de Reglas Recursivas:** Dado que puede existir un sistema de reglas que contenga tanto reglas no recursivas como reglas recursivas, es necesario realizar un proceso de detección de reglas recursivas. En el caso de que el sistema cuente con este último tipo de reglas, se llevarán a cabo las transformaciones necesarias al sistema, de manera que posteriormente se pueda realizar la evaluación de dichas reglas.

3.- **Evaluación de Reglas Recursivas:** Se implantará un procedimiento para la evaluación de las reglas recursivas a través de un algoritmo específico denominado *transformada delta*.

1.4 Trabajos Relacionados

Actualmente existen algunos sistemas que están relacionados con este trabajo.

El lenguaje lógico "LOLA - A Logic Language for Deductive Database and its Implementation" (Un Lenguaje Lógico para Bases de Datos Deductivas y su Implantación), se desarrolló en la Universidad Tecnológica de Munich [FrSS90]. Se diseñó como un lenguaje de consulta para un sistema de bases de datos deductivas. LOLA integra la programación lógica y el procesamiento relacional de consultas, donde la evaluación de estas consultas se efectúa de abajo hacia arriba comenzando por las relaciones base.

Una consulta se transforma a una expresión equivalente de un álgebra relacional extendida. Dado un conjunto de relaciones base, la expresión resultante calcula el conjunto de tuplos correspondientes que responden a la consulta. Las relaciones base pueden estar en memoria principal o en cualquier sistema de bases de datos relacional externo accesible por medio de SQL. Los componentes del sistema LOLA son:

- Una interfaz con el usuario que permite al sistema aceptar programas y consultas; después de que la consulta es compilada por medio de esta interfaz, se muestra la respuesta al usuario o se almacena como una relación.

- Un compilador que toma como entrada un programa o una consulta, y genera una expresión relacional equivalente.

- Un módulo de optimización que puede realizar dos tipos de optimizaciones: a nivel fuente o actuando sobre el código intermedio representado por una gráfica de operadores.

- Un sistema para ejecución ("run-time") compuesto por: LISP como lenguaje anfitrión, el álgebra relacional extendida denominada R-LISP, una base de datos en memoria principal y una interfaz con SQL para sistemas manejadores de bases de datos relacionales externos.

Un segundo trabajo relacionado es el denominado "DATALOG" el cual difiere de PROLOG en [Ullm88] :

- No permite símbolos de función como argumentos, es decir, DATALOG permite solo variables y símbolos de constantes como argumentos de un predicado.

- La interpretación de los programas de DATALOG utilizan el punto de vista del modelo teórico, mientras que PROLOG cuenta con una interpretación computacional.

El modelo de datos matemático que fundamenta a DATALOG es esencialmente el modelo relacional, aunque las relaciones no hacen referencia a sus argumentos por su nombre, sino por su posición.

Los sistemas descritos se orientan solo al desarrollo de bases de datos deductivas de propósito general, mientras que nuestro sistema pretende desarrollar una herramienta para la programación de sistemas expertos, la cual utilice una base de datos deductiva.

1.5 Organización de la Tesis

Este trabajo se organizó en seis capítulos. Los capítulos dos y tres ubican a este trabajo en los conceptos teóricos que lo fundamentan, mientras que los capítulos cuatro y cinco proporcionan los procedimientos de implantación para que finalmente en el capítulo seis se presente la evaluación de los resultados.

Debido a que una base de datos deductiva es una extensión de las bases de datos relacionales a través de la lógica de predicados, en el capítulo dos se presenta el modelo de datos relacional, los principales aspectos de la lógica proposicional, de la lógica de predicados y por último, se concluye con la definición formal de base de datos deductiva.

A través de las gráficas se pueden representar algunas propiedades de las reglas. En el tercer capítulo se presentan los principios fundamentales de la teoría de gráficas así como los diferentes tipos de gráficas dependiendo de sus características. Una gráfica se puede representar en forma matricial; esto es útil debido a que, a través de esta representación, es posible detectar la recursividad en un sistema de reglas.

La arquitectura del sistema se presenta en la primera sección del capítulo cuatro. Posteriormente se explican los aspectos conceptuales de la normalización de reglas no recursivas así como su evaluación. La gráfica de dependencias se utiliza para representar la dependencia entre los predicados de una regla; dicha gráfica se representa en forma matricial y se utiliza posteriormente en la detección de reglas recursivas (ciclos). Por último se presenta el método para la evaluación de reglas recursivas.

En la primera sección del capítulo cinco se presentan las estructuras utilizadas para la representación interna de las reglas, así como las tablas necesarias para almacenar los objetos del sistema. Posteriormente se describe la forma en que se implantó el sistema.

Por último, en el capítulo seis se hace una evaluación de los resultados obtenidos durante el desarrollo de este trabajo, así como las ampliaciones futuras al sistema.

Capítulo 2

LOGICA
Y
BASES DE DATOS DEDUCTIVAS

De manera informal se define una base de datos deductiva, como una extensión de las bases de datos relacionales a través del uso de la lógica de predicados en la cual, nuevos hechos se deducen de hechos almacenados explícitamente en la base de datos.

Debido a que las bases de datos deductivas son una extensión de las bases de datos relacionales, al inicio de este capítulo se explicarán los fundamentos del modelo relacional. Posteriormente, en la sección 2.2 y 2.3, se estudiará brevemente la lógica proposicional y la lógica de predicados y por último, en la sección 2.4, se definirá formalmente una base de datos deductiva.

2.1 Bases de Datos Relacionales

2.1.1 Modelo de Datos Relacional

Los objetos con los que trata el modelo relacional son relaciones, las cuales se utilizan para representar objetos del mundo real y sus interrelaciones, a esta característica se le denomina relativismo semántico. Este relativismo semántico se expresa utilizando el término **entidad** tanto para los objetos como para las interrelaciones entre ellos. Dichas interrelaciones pueden considerarse objetos abstractos.

Un conjunto E de entidades del mismo tipo se caracteriza por un conjunto de **atributos** $A_1, A_2, ..., A_n$, denotado como $E\ (A_1, A_2, ..., A_n)$, donde $A_i : E \rightarrow D_i$, $i = 1, ..., n$, es una función que mapea del conjunto de entidades E al conjunto de valores D_i, denominado el **dominio** del atributo A_i. Dada la entidad e que pertenece al conjunto de entidades E, $A_i\ (e)$ en D_i es el **valor del atributo** A_i de la entidad e.

Una **n-ada (tuplo)** es un conjunto de valores que representan una entidad. De este modo, diferentes entidades se representan por diferentes tuplos.

Una **relación** R es un conjunto de tuplos. Una relación de orden n definida sobre los dominios $D_1, D_2, ..., D_n$, es un subconjunto del producto cartesiano de $D_1, D_2, ..., D_n$, es decir:

$$R \subseteq D_1 \times D_2 \times \ldots \times D_{n-1} \times D_n$$

El **esquema** de una base de datos relacional es una descripción de la estructura de las relaciones que la conforman.

Las bases de datos relacionales están constituidas por dos tipos de relaciones: las **relaciones base** y las **relaciones virtuales**. Las primeras existen explícitamente en la base de datos y las segundas se obtienen aplicando operadores algebraicos sobre las relaciones base y se les conoce comúnmente como **vistas**.

2.1.2 Algebra Relacional

El álgebra relacional consiste de un conjunto de operaciones que toman una o dos relaciones como entrada y producen como resultado un nueva relación. Se compone de dos grupos de operadores.

- Los operadores tradicionales sobre conjuntos:

- **Unión:** La unión de dos relaciones compatibles a la unión[1] R y S es el conjunto de tuplos que pertenecen a R, a S o a ambas, esto es:

$$R \cup S = \{ x \mid x \in R \text{ o } x \in S \}$$

- **Intersección:** La intersección de dos relaciones compatibles a la unión R y S es el conjunto de tuplos que pertenecen a R y a S, es decir:

$$R \cap S = \{ x \mid x \in R \text{ y } x \in S \}$$

- **Diferencia:** La diferencia de dos relaciones compatibles a la unión R y S (en ese orden) es el conjunto de tuplos que pertenecen a R y no pertenecen a S, es decir:

$$R - S = \{ x \mid x \in R \text{ y } x \notin S \}$$

- **Producto Cartesiano:** Sean R y S dos relaciones de orden K_1 y K_2 respectivamente, el producto cartesiano $R \times S$ es el conjunto de tuplos (r,s) de orden $K_1 + K_2$, donde r pertenece a R y s pertenece a S, es decir:

$$R \times S = \{ (r,s) \mid r \in R \text{ y } s \in S \}$$

■ Los operadores especiales:

- **Proyección :** Sea $R(A_1, A_2, \ldots, A_n)$ una relación de orden n, $X \subseteq \{A_1, A_2, \ldots, A_n\}$ y $Y = \{A_1, A_2, \ldots, A_n\} \backslash X$[2]. Permutando los atributos de R podemos representar $R(A_1, A_2, \ldots, A_n)$ como $R(X,Y)$. El resultado de la operación de proyección de la relación R sobre los atributos X, representado como $R[X]$, se define como:

$$R[X] = \{ x \mid \exists y \text{ tal que } (x,y) \in R(X,Y) \}$$

(1) Se dice que dos relaciones son compatibles a la unión si el número de atributos y el dominio de los atributos correspondientes son los mismos.

(2) Dados dos conjuntos C1 y C2, C1\C2 denota el complemento de C2 en C1.

- **Restricción:** Sea $R(A_1, A_2, ..., A_n)$ una relación de orden n, y P una condición lógica definida sobre el producto cartesiano de los dominios de los atributos, el resultado de la restricción (selección) de la relación R con respecto a P se denota como $R/P/$ y se define como:

$$R[P] = \{ x \mid x \text{ está en } R \text{ y } P(x) \text{ es verdadera} \}$$

- **División:** Sean $R(X,Y)$ y $S(Z)$ dos relaciones donde X,Y y Z son conjuntos de atributos. Asumiendo que Y y Z contienen el mismo número de atributos y los dominios de los atributos correspondientes son iguales, el resultado de la división de $R(X,Y)$ entre $S(Z)$ expresado como $R/Y:Z/S$, es el subconjunto máximo de la proyección $R/X/$ tal que el producto cartesiano con $S(Z)$ está contenido en $R(X,Y)$, es decir:

$$R(X,Y) = R[Y:Z]S \times S(Z) \cup Q(X,Y)$$

donde $R/Y:Z/S$ es el cociente y $Q(X,Y)$ es el residuo.

- **"Join":** Sea $R(A_1, A_2, ..., A_n)$ y $S(B_1, B_2, ..., B_m)$ dos relaciones y sean los conjuntos de atributos $X \subseteq \{A_1, A_2, ..., A_n\}$ y $Y \subseteq \{B_1, B_2, ..., B_m\}$ los cuales tienen el mismo número de atributos y los dominios de los atributos correspondientes son los mismos; y sean $Z = \{A_1, A_2, ..., A_n\} \backslash X$ y $W = \{B_1, B_2, ..., B_m\} \backslash Y$, permutando el orden de los atributos de las relaciones R y S podemos representarlas como $R(Z,X)$ y $S(Y,W)$. El "join" natural de las relaciones R y S sobre los atributos X y Y, se denota como $R/X=Y/S$, y se define como:

$$R[X=Y]S = \{ (z,x,w) \mid (z,x) \in R, (y,w) \in S \text{ y } x=y \}$$

2.1.3 Lenguaje Relacional SQL

Aunque el álgebra relacional es un lenguaje formal simple, no es adecuado para usuarios casuales de bases de datos, de hecho resulta bastante impráctico para realizar consultas, por tal motivo se han diseñado diferentes lenguajes relacionales de consulta que resultan ser prácticos para los usuarios en general. Uno de estos lenguajes es **SQL** ("Structured Query Language") el cual es bastante conocido y aceptado como un estándar.

En una base de datos se cuenta con cuatro lenguajes: el **Lenguaje de Consulta (QL)**, el cual permite extraer información almacenada en la base de datos, el **Lenguaje de Definición de Datos (DDL)**, con el cual se puede definir los objetos, el **Lenguaje de Manipulación de Datos (DML)**, este lenguaje se utiliza para la inserción, actualización y borrado de la información en la base de datos, y el **Lenguaje de Control (CL)**, el cual no sera tratado por no ser utilizado en este trabajo. SQL no es solamente un lenguaje de consulta, ya que de hecho está constituido por los cuatro lenguajes mencionados.

2.1.3.1 Lenguaje de Consulta

El concepto fundamental de SQL se denomina **bloque de consulta** cuya forma básica es la siguiente:

```
SELECT <lista de atributos>
FROM   <lista de relaciones>
WHERE  <condición>
```

El resultado de la ejecución de un bloque de consulta es una relación cuya estructura y contenido se determina por el mismo bloque. Los atributos listados en la cláusula SELECT se seleccionan de las relaciones de la lista de relaciones de la cláusula FROM.

Las dos primeras cláusulas (SELECT y FROM) del bloque de consulta, definen la operación de proyección. La condición de la cláusula WHERE es una expresión lógica, la cual contiene atributos de las relaciones listadas en la cláusula FROM y determina qué tuplos de las relaciones listadas califican para la operación de proyección. La cláusula WHERE, permite especificar las operaciones de restricción, de "join" o ambas.

Es importante observar que en el bloque de consulta no se especifica el orden en que se realizan las operaciones.

2.1.3.2 Lenguaje de Definición de Datos (DDL)

El esquema de una base de datos se especifica por un conjunto de definiciones expresadas en un lenguaje denominado Lenguaje de Definición de Datos ("Data Definition Language").

Entre las instrucciones más importantes del Lenguaje de Definición de Datos se encuentran la de creación y borrado de relaciones y vistas.

La forma fundamental para crear una relación base se efectúa por medio de la instrucción:

```
CREATE TABLE <nombre de la relación>
             (<atributo1> <tipo1> [NOT NULL],
              <atributo2> <tipo2> [NOT NULL],
                          :
                          :
              <atributon> <tipon> [NOT NULL])
```

donde [NOT NULL] indica que el atributo no puede asumir un valor nulo[3] y las expresiones enmarcadas por [] son opcionales.

(3)El valor nulo significa que el valor de un determinado atributo no se conoce o no existe.

Para la creación de una relación virtual (vista) es necesario hacer referencia a una o más relaciones a través de una consulta, la sintaxis de la instrucción es:

```
CREATE VIEW <nombre vista>
            [(<lista de atributos>)]
         AS <consulta>
```

Para borrar una relación base la sintaxis es:

```
DROP TABLE <nombre de la relación>
```

y para una vista:

```
DROP VIEW <nombre de la vista>
```

2.1.3.3 Lenguaje de Manipulación de Datos (DML)

El Lenguaje de Manipulación de Datos ("Data Manipulation Language") permite realizar operaciones en la base de datos tales como: inserción, actualización y borrado de información. Las instrucciones de SQL son las siguientes:

Para insertar:

```
INSERT INTO <nombre relación>
            [(<atributo1,..., atributon>)]
            {VALUES (<valor1>...<valorn>) |
             <consulta>}
```

Como se puede observar en la sintaxis de esta instrucción, existen diferentes formas para insertar un tuplo en una relación. Una de ellas, se realiza especificando los valores correspondientes a los atributos, en el orden en el cual se encuentran definidos en la relación, opcionalmente se pueden especificar los nombres de los atributos. Otra forma de insertar un tuplo, es estableciendo una consulta cuyos atributos seleccionados tengan la misma estructura que la relación en la cual se requiere insertar la información seleccionada.

Para borrar:

```
DELETE FROM <nombre relación>
    [WHERE <condición>]
```

En esta instrucción si no se especifica la condición de la cláusula WHERE, esto es, la condición que deben cumplir los tuplos que se desean borrar, entonces todos los tuplos de la relación serán borrados.

Para actualizar:

```
UPDATE <nombre relación>
SET {<atributo1> = <valor1>,...,
     <atributon> = <valorn>|
     (atributo1,...,atributon) = (subconsulta)}
[WHERE <condición>]
```

Como se observa en la sintaxis de la instrucción anterior, existen varias formas para actualizar información en una base de datos. En una de ellas, a cada atributo de la relación que se desea actualizar, se le asigna su valor correspondiente; otra forma, es listando los atributos a modificar y se establece una subconsulta la cual seleccionará los valores correspondientes a los atributos.

2.2 Lógica Proposicional

La lógica se encarga del estudio del razonamiento. Una de las formas más simples de la lógica es la lógica proposicional.

2.2.1 Lenguaje de la Lógica Proposicional

En la lógica proposicional nos interesan las sentencias que pueden ser verdaderas o falsas, pero no ambas. Formalmente, una sentencia declarativa se denomina una **proposición**. El valor "verdadero" o "falso" asignado a una proposición se denomina el **valor de verdad** de la proposición; se representa "verdadero" por **V** y "falso" por **F**.

Un símbolo de proposición se denomina **átomo** o **fórmula atómica**, esto es, una **proposición simple** es un átomo.

A partir de las proposiciones, se pueden construir **proposiciones compuestas** utilizando **conectivos lógicos**. En la lógica proposicional existen cinco conectivos lógicos: ¯ (negación), & (conjunción), V (disyunción), → (condicional) y ↔ (equivalencia).

En la lógica proposicional definimos **fórmulas bien formadas**, o simplemente **fórmulas**, en forma recursiva como:

1.- Un átomo es una fórmula.

2.- Si G es una fórmula, entonces (¯G) es una fórmula.

3.- Si G y H son fórmulas, entonces (G & H), (G V H), ($G \rightarrow H$), y ($G \leftrightarrow H$) son fórmulas.

4.- Todas las fórmulas se generan aplicando las reglas anteriores.

2.2.2 Interpretación de fórmulas

Los valores de verdad de cualquier fórmula pueden evaluarse en términos de los valores de verdad de los átomos que la conforman.

Sea G una fórmula proposicional, sean $A_1, A_2, \ldots, A_n$ todos los átomos que ocurren en la fórmula G, una **interpretación** de G es la asignación de valores de verdad a $A_1, \ldots, A_n$ en donde a cada A_i se le asigna **V** o **F**, mas no ambos.

Se dice que una fórmula G es **verdadera bajo una interpretación**, si y solo si, G evalúa a verdadero en la interpretación; en caso contrario, se dice que G es **falsa bajo la interpretación.**

2.2.3 Validez e Inconsistencia

Una fórmula se define como **válida** (tautología), si y solo si, es verdadera bajo todas sus interpretaciones. Se dice que una fórmula es **inválida** (contradicción), si y solo si, no es válida.

Una fórmula es **inconsistente** o **insatisfacible**, si y solo si, es falsa bajo todas sus interpretaciones. Se dice que una fórmula es **consistente** o **satisfacible**, si y solo si, no es inconsistente.

2.2.4 Formas Normales

Frecuentemente es necesario transformar una fórmula de una forma a otra y en especial a una *forma normal*. Esto se hace sustituyendo una fórmula dada por otra equivalente y repitiendo este proceso hasta obtener la forma deseada.

Dos fórmulas F y G se dice que son **equivalentes** (F es **equivalente** a G) denotada por $F \leftrightarrow G$, si y solo si, los valores de verdad de F y G son los mismos bajo cualquier interpretación de F y G.

Sean $F_1, F_2, ..., F_n$ fórmulas. $F_1 \vee F_2 \vee ... \vee F_n$ es verdadera si al menos alguna F_i, $1 \leq i \leq n$, es verdadera; en caso contrario, es falsa. $F_1 \vee F_2 \vee ... \vee F_n$ se denomina la **disyunción** de $F_1, ..., F_n$.

En forma similar, $F_1 \,\&\, F_2 \,\&\, ... \,\&\, F_n$ es verdadera si cada una de las F_i, $1 \leq i \leq n$, son verdaderas; en caso contrario, es falsa. $F_1 \,\&\, F_2 \,\&\, ... \,\&\, F_n$ se denomina la **conjunción** de $F_1, ..., F_n$.

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

Se dice que una fórmula F está en **forma normal conjuntiva**, si y solo si, F tiene la forma $F = F_1 \&...\& F_n$, para $n \geq 1$, y cada $F_1, ..., F_n$ es una disyunción de literales.

Se dice que una fórmula F está en **forma normal disyuntiva**, si y solo si, F tiene la forma $F = F_1 \vee...\vee F_n$, para $n \geq 1$ y cada $F_1, ..., F_n$ es una conjunción de literales.

Es posible transformar cualquier fórmula a una forma normal equivalente aplicando repetitivamente las leyes de D'Morgan, conmutativa, asociativa y distributiva hasta obtener la forma normal deseada.

2.2.5 Consecuencia Lógica

Tanto en matemáticas como en la vida cotidiana, frecuentemente tenemos que decidir cuando una afirmación sigue o es resultado de otras afirmaciones.

Dadas las fórmulas $F_1, F_2, ..., F_n$ y una fórmula G, se dice que G es una **consecuencia lógica** de $F_1, ..., F_n$, si y solo si, para cualquier interpretación I en la cual $F_1 \& F_2 \&...\& F_n$ es verdadera, G también es verdadera. $F_1, F_2, ..., F_n$ se denominan **axiomas** (**premisas** o **postulados**) de G.

Teorema de Deducción.- Dadas las fórmulas $F_1, ..., F_n$ y la fórmula G, se dice que G es consecuencia lógica de $F_1, ..., F_n$, si y solo si, la fórmula $((F_1 \&...\& F_n) \rightarrow G)$ es válida.

Teorema.- Dadas las fórmulas $F_1, ..., F_n$ y una fórmula G, se dice que G es una consecuencia lógica de $F_1, ..., F_n$, si y solo si, la fórmula $((F_1 \&...\& F_n \& \neg G)$ es inconsistente.

Si G es una consecuencia lógica de $F_1, ..., F_n$, la fórmula $((F_1 \&...\& F_n) \rightarrow G)$ se denomina un **teorema**, y G se denomina la **conclusión del teorema.**

2.3 Lógica de Predicados

Con átomos se construyen fórmulas, las cuales se utilizan para representar diversas ideas. En la lógica proposicional, dichos átomos se tratan como unidades simples que no cuentan con estructuras, por lo que existen muchas ideas complejas que no se pueden representar.

La **lógica de predicados** o de primer orden, permite expresar ideas más complejas que la lógica proposicional al agregar a esta última, tres nociones lógicas: términos, predicados y cuantificadores.

2.3.1 Lenguaje de la Lógica de Predicados

Los átomos se constituyen a partir de cuatro tipos de símbolos:

1.- Símbolos individuales o de constantes.

2.- Símbolos de variables.

3.- Símbolos de funciones.

4.- Símbolos de predicados.

Tanto un símbolo de función como un símbolo de predicado toman un número específico de argumentos. Si un símbolo de función f toma n argumentos, f se denomina un **símbolo de función de orden n.** En forma similar, si un símbolo de predicados P toma n argumentos, P se denomina un **símbolo de predicado de orden n.**

Una función es un mapeo que asocia a una lista de constantes una constante.

Se define un **término** en forma recursiva como:

1.- Un símbolo de constante es un término.

2.- Una variable es un término.

3.- Si f es un símbolo de función de orden n, y $t_1, ..., t_n$ son términos, entonces $f(t_1, ..., t_n)$ es un término.

4.- Todos los términos se generan aplicando las reglas anteriores.

Si P es un símbolo de predicado de orden n, y $t_1, ..., t_n$ son términos, entonces $P(t_1, ..., t_n)$ es un **átomo.**

La lógica de predicados cuenta con los mismos conectivos lógicos que la lógica proposicional para construir fórmulas. Por otro lado, ya que se han introducido variables, se usarán dos símbolos especiales ($\forall$ y $\exists$) para caracterizar a las variables. Los símbolos $\forall$ y $\exists$ se denominan **cuantificador universal** y **cuantificador existencial** respectivamente. Si x es una variable, entonces $(\forall x)$ se lee como "para toda x", mientras que $(\exists x)$ se lee como "existe una x".

Se define el **alcance** de un cuantificador que ocurre en una fórmula, como la subfórmula a la cual el cuantificador aplica.

Se dice que una ocurrencia de una variable en una fórmula es **ligada**, si y solo si, la ocurrencia está dentro del alcance de un cuantificador que esté empleando a la variable o bien, la ocurrencia de la variable está en el cuantificador. Una ocurrencia de una variable en una fórmula es **libre**, si y solo si, esta ocurrencia de la variable no está ligada.

Una variable es libre en una fórmula si al menos una ocurrencia de ella es libre en la fórmula. Una variable es ligada en una fórmula si al menos una ocurrencia de ella es ligada.

En la lógica de predicados, se definen **fórmulas bien formadas**, o simplemente **fórmulas**, en la siguiente forma recursiva:

1.- Un átomo es una fórmula.

2.- Si F y G son fórmulas, entonces $\neg(F)$, $(F \vee G)$, $(F \& G)$, $(F \rightarrow G)$ y $(F \leftrightarrow G)$ son fórmulas.

3.- Si F es una fórmula y x es una variable libre en F, entonces $(\forall x)F$ y $(\exists x)F$ son fórmulas.

4.- Las fórmulas se generan solo por un número finito de aplicaciones de las reglas 1, 2 y 3.

2.3.2 Interpretación de Fórmulas

Una **interpretación** de una fórmula F en la lógica de predicados consiste de un dominio no vacío D y una "asignación" de valores a cada símbolo de constante, símbolo de función y símbolo de predicado que ocurre en F de la siguiente forma:

1.- A cada símbolo de constante, se le asigna un elemento en D.

2.- A cada símbolo de función de orden n, se le asigna un mapeo de D^n a D.

3.- A cada símbolo de predicado de orden n, se le asigna un mapeo de D^n a **{V, F}**.

Dada una interpretación I de una fórmula F sobre un dominio D, la fórmula puede evaluar a **Verdadero** o **Falso**. Es importante señalar que cualquier fórmula que contenga variables libres no puede evaluarse.

Se dice que una fórmula G es **consistente** (satisfacible), si y solo si, existe una interpretación I tal que G evalúa a verdadero en I. Si una fórmula G es verdadera en una interpretación I, decimos que I es un **modelo** de G e I **satisface** a G.

Se dice que una fórmula G es **inconsistente** (insatisfacible), si y solo si, no existe una interpretación que satisfaga a G.

Se dice que una fórmula G es una **consecuencia lógica** de las fórmulas $F_1, F_2, ..., F_n$, si y solo si, para cada interpretación I en la cual $F_1 \& F_2 \& ... \& F_n$ es verdadera, G también es verdadera.

Las relaciones entre validez, inconsistencia y consecuencia lógica establecidas en los teoremas citados en la lógica proposicional, también son válidas en la lógica de predicados.

2.3.3 Forma Normal Prenex

En la lógica proposicional, se introdujeron dos formas normales: la forma normal conjuntiva y la forma normal disyuntiva. En la lógica de predicados existe la forma normal prenex. Esta forma se utiliza para simplificar los procedimientos de prueba de una fórmula.

Se dice que una fórmula F en la lógica de predicados está en **forma normal prenex**, si y solo si, F tiene la forma:

$$(Q_1 x_1) \ \ldots \ (Q_n x_n)\,(M)$$

donde cada $(Q_i x_i)$, $i = 1, \ldots, n$ es un cuantificador $(\forall x_i)$ o $(\exists x_i)$, y M es una fórmula que no contiene cuantificadores. $(Q_1 x_1) \ldots (Q_n x_n)$ se denomina el **prefijo** de la fórmula F y M se denomina la **matriz** de la fórmula F.

2.3.4 Forma Normal de Skolem

El procedimiento de deducción utilizado en este trabajo, se basa en un procedimiento de prueba por refutación, esto es, dado un conjunto de fórmulas S y una fórmula w se agrega la negación de la fórmula w al conjunto S y se demuestra que sea inconsistente. Estos procedimientos de refutación se aplican a la forma estándar de Skolem de una fórmula, por lo cual es importante que se defina en que consiste. Esta forma se basa en las siguientes ideas:

- Una fórmula en la lógica de predicados puede transformarse a una fórmula equivalente en forma normal prenex, donde la matriz no contiene cuantificadores y el prefijo es una secuencia de cuantificadores.

- La matriz, dado que no contiene cuantificadores, puede transformarse a una forma normal conjuntiva.

- Sin afectar la propiedad de inconsistencia, los cuantificadores existenciales en el prefijo pueden eliminarse utilizando funciones de Skolem.

Para las dos primeras transformaciones se aplican leyes de equivalencia sobre la fórmula.

Sea una fórmula F en forma normal prenex, donde la matriz M está en una forma normal conjuntiva. Supóngase que Q_r es un cuantificador existencial en el prefijo $(Q_1x_1)...(Q_nx_n)$, $1 \leq r \leq n$. Si no aparecen cuantificadores universales antes de Q_r, se escoge un símbolo de constante c diferente de los otros símbolos de constantes que ocurren en M, se sustituyen todas las x_r que aparecen en M por c, y se borra (Q_rx_r) del prefijo. Si $Q_{s_1}, ..., Q_{s_m}$ son todos los cuantificadores universales que aparecen antes de Q_r, para $1 \leq s_1 < s_2 < ... < r$, se escoge un nuevo símbolo de función $f(x_{s_1}, x_{s_2}, ..., x_{s_m})$ de orden m, f diferente de los otros símbolos de funciones que aparecen en la fórmula, se sustituyen todas las x_r en M por $f(x_{s_1}, x_{s_2}, ..., x_{s_m})$ y se borra (Q_rx_r) del prefijo. Después de aplicar todos los procesos anteriores a los cuantificadores existenciales en el prefijo, la fórmula que se obtiene está en la **forma estándar de Skolem** o simplemente **forma estándar**. Los símbolos de constantes y los símbolos de funciones utilizados para reemplazar las variables existenciales se denominan **funciones de Skolem.**

2.3.5 Forma Clausal

Se define una **cláusula** como una disyunción de literales.

La forma general de una cláusula es:

$$R_1 \vee \ldots \vee R_q \vee \neg P_1 \vee \ldots \vee \neg P_k$$

y es equivalente a

$$R_1 \vee R_2 \vee \ldots \vee R_q \leftarrow P_1 \wedge P_2 \wedge \ldots \wedge P_k.$$

Cuando $q = 1$ ó $q = 0$, la cláusula se denomina **cláusula de Horn**; en donde al lado izquierdo se le denomina **consecuente** y al derecho **antecedente.**

Una cláusula con r literales se denomina cláusula **r-literal**. Una cláusula 1-literal se denomina cláusula **unitaria**. Cuando una cláusula no tiene literales se denomina cláusula **vacía**.

Un conjunto S de cláusulas se considera como una conjunción de todas las cláusulas en S, donde cada variable en S está universalmente cuantificada en forma implícita.

Teorema. Sea S un conjunto de cláusulas que representan una forma estándar de una fórmula F, entonces F es inconsistente, si y solo si, S es inconsistente.

Se dice que una cláusula es de **rango restringido** si cualquier variable que ocurre en el lado izquierdo de una cláusula ocurre en el lado derecho de esta [NiYa78].

2.4 Bases de Datos Deductivas

Una base de datos deductiva se define como una base de datos en la cual, se pueden deducir nuevos hechos a partir de hechos almacenados explícitamente en la base de datos.

Una base de datos deductiva se constituye de [GaMN84] :

1.- Un conjunto finito de constantes.

2.- Un conjunto de cláusulas de primer orden que no contienen símbolos de funciones.

Los símbolos de funciones se excluyen para obtener respuestas explícitas y finitas a las consultas, por lo que los argumentos de los predicados que conforman una regla, sólo son símbolos de constantes o símbolos de variables.

Como se vio anteriormente la forma general de una cláusula es:

$$R_1 \vee R_2 \vee \ldots \vee R_q \leftarrow P_1 \& P_2 \& \ldots \& P_k$$

Existen diferentes tipos de cláusulas dependiendo del número de átomos del consecuente y del antecedente, esto es, en base a los valores de k y q, los tipos son[GaMN84] :

- **Tipo 1.-** $k = 0$ y $q = 1$. La cláusula tiene la forma

$$R(t_1, \ldots, t_m) \leftarrow .$$

Si todos los argumentos (t_i) de la cláusula son constantes, $c_{i_1}, \ldots, c_{i_m}$, se tiene:

$$R(c_{i_1}, \ldots, c_{i_m}) \leftarrow$$

que representa un hecho en la base de datos; en caso contrario, si al menos uno de los argumentos (t_i) es una variable, entonces la cláusula corresponde a una aserción general en la base de datos.

- **Tipo 2.-** $k = 1$ y $q = 0$. La cláusula tiene la forma

$$\leftarrow P(t_1, \ldots, t_m) .$$

Cuando todos los argumentos (t_i) de la cláusula son constantes, $c_{i_1}, \ldots, c_{i_m}$, se tiene:

$$\leftarrow P(c_{i_1}, \ldots, c_{i_m})$$

que representa un hecho negativo. En el caso de que al menos uno de los argumentos (t_i) sea una variable, entonces la cláusula puede considerarse como una restricción de integridad (cláusula tipo 3) o bien, que el valor no existe (valor nulo).

- **Tipo 3.-** $k > 1$ y $q = 0$. La cláusula tiene la forma

$$\leftarrow P_1 \ \& \ \ldots \ \& \ P_k .$$

Esta cláusula puede considerarse como una restricción de integridad, esto es, los datos que se inserten en una base de datos, deben satisfacer las leyes especificadas por las restricciones de integridad para poder ser aceptados en la base de datos.

- Tipo 4.- $k \geq 1$ y $q = 1$. La cláusula tiene la forma

$$R_1 \leftarrow P_1 \ \& \ P_2 \ \& \ \ldots \ \& \ P_k.$$

La cláusula puede considerarse como una restricción de integridad o como la definición del predicado R_1 en términos de los predicados $P_1, \ldots, P_k$ (dicha definición constituye una ley deductiva).

- Tipo 5.- $k = 0$ y $q > 1$. La cláusula tiene la forma

$$R_1 \ \vee \ R_2 \ \vee \ \ldots \ \vee \ R_n \leftarrow .$$

Si las x_i, $i = 1, \ldots, n$, son constantes, entonces se tiene una **aserción indefinida**, esto es, cualquier combinación de una o más de las R_i es verdadera, pero no se sabe cuales son las verdaderas.

- Tipo 6.- $k \geq 1$ y $q > 1$. La cláusula tiene la forma

$$R_1 \ \vee \ R_2 \ \vee \ \ldots \ \vee \ R_q \leftarrow P_1 \ \& \ P_2 \ \& \ \ldots \ \& \ P_k.$$

La cláusula puede interpretarse como una restricción de integridad o como la definición de datos indefinidos.

Por último se tiene la cláusula en donde $k = 0$ y $q = 0$ (cláusula vacía), dicha cláusula denota falsedad y no debe ser parte de la base de datos. Se dice que una cláusula es **definida** si el consecuente de la misma está constituida por un solo átomo.

Las bases de datos deductivas se dividen en dos tipos: las bases de datos deductivas definidas, en las que no se encuentran cláusulas de tipo 5 y tipo 6 y las bases de datos deductivas indefinidas en las que dichas cláusulas si existen. En este trabajo solo se considerarán las bases de datos deductivas definidas.

2.4.1 Bases de Datos Deductivas Definidas

2.4.1.1 Definición Formal

Una base de datos deductiva definida es una teoría particular de primer orden, la cual se fundamenta en la teoría de las bases de datos convencionales y en la adición de una nueva clase de axiomas. Formalmente, una base de datos deductiva definida se constituye de lo siguiente[GaMN84] :

1.- Una teoría cuyos axiomas propios son:

- Axiomas de particularización
  - **Axioma de cerradura de dominio.-** Establece que no existen mas elementos que aquellos que están en la base de datos.
  - **Axioma de nombre único.-** Indica que elementos con nombres distintos son diferentes.
  - **Axioma de Igualdad.-** Especifican las propiedades de igualdad
    - reflexividad
    - simetría
    - transitividad
  - **Axioma de completés.-** Se conforma de todos los hechos y leyes deductivas que involucran al predicado correspondiente.
- **Hechos elementales.-** Conjunto de fórmulas definidas por medio de cláusulas de la forma

$$R(c_1, \ldots, c_n) \leftarrow$$

■ **Leyes deductivas.-** Conjunto de cláusulas definidas de la forma

$$Q_1 \leftarrow P_1, \ldots, P_k$$

2.- Un conjunto de restricciones de integridad

Permiten representar las propiedades que los objetos deben cumplir para poder pertenecer a la base de datos.

Las relaciones en una base de datos deductiva se definen en conjunto por una serie de leyes deductivas y hechos elementales. Una base de datos deductiva se constituye por dos componentes: una base de datos extensional (BDE) y una base de datos intencional (BDI). La primera la constituyen los hechos elementales los cuales se almacenan en las relaciones base y la segunda se constituye por las leyes de deducción. Una **relación derivada** siempre está definida intencionalmente y adicionalmente puede estar definida en forma extensional.

Una relación derivada es equivalente a una vista cuando:

- No existen hechos elementales asociados a ésta.
- No aparecen leyes deductivas recursivas entre las leyes deductivas que implican esta relación.

2.4.1.2 Definición Operacional de Bases de Datos Deductivas Definidas

Considerando la complejidad combinatoria de los axiomas de particularización, sería bastante ineficiente implantar un sistema manejador de base de datos deductivas (definidas) utilizando la definición formal. La solución es similar a la que se utiliza para manejadores de base de datos convencionales.

Los axiomas de particularización en una base de datos deductiva definida pueden eliminarse de la siguiente manera:

- El axioma de cerradura de dominio se puede omitir utilizando cláusulas de rango restringido para las consultas, restricciones de integridad y leyes deductivas.

- Los axiomas de nombre único y de completés se excluyen interpretando la negación como falla.

En resumen, una base de datos deductiva definida que utiliza fórmulas de rango restringido, desde el punto de vista operacional se constituye como:

1.- Un conjunto de axiomas constituido por hechos elementales y leyes deductivas.

2.- Un conjunto de restricciones de integridad.

3.- Una metaregla: La negación como falla finita.

Capítulo 3

TEORIA DE GRAFICAS

A través de una gráfica se pueden representar en forma sencilla, ciertas propiedades de un sistema de reglas; en la primera sección se introducirán los fundamentos de la teoría de gráficas, en la sección 3.2 se analizarán las gráficas dirigidas mientras que en la sección 3.3 se estudiarán brevemente la gráficas dirigidas acíclicas.

Existen diferentes formas de representar una gráfica, una de ellas son las matrices como se describe en la sección 3.4. La teoría de gráficas servirá para la detección de recursividad en un sistema de reglas, esto se verá en el capítulo 4.

3.1 Fundamentos de la Teoría de Gráficas

Una **gráfica no dirigida** G consiste de un conjunto finito, no vacío, $V = V(G)$ de p vértices, y un conjunto $X = X(G)$ de q pares de vértices distintos, no ordenados que pertenecen a V. Cada par de vértices $x = (u,v)$ en el conjunto X es una arista de G y se dice que x une a u y v. Se escribe $x = uv$ para indicar que u y v son **vértices adyacentes**. El vértice u y la arista x son **incidentes** el uno con el otro, así como v y x.

Se dice que dos aristas x_1 y x_2 son **adyacentes** si estas inciden en un vértice común.

Una gráfica con p vértices y q aristas se denomina **gráfica** (p,q). A la gráfica (1,0) se le conoce como **gráfica trivial**.

Se dice que existe un **enlace** si una arista une a un vértice consigo mismo. Por definición una gráfica no permite tener enlaces.

Una **multigráfica** es aquella en la que no se permiten enlaces pero puede incluir más de una arista uniendo dos vértices, a estas aristas se les conoce como **aristas múltiples.**

Una **pseudográfica** es una gráfica que permite tanto enlaces como aristas múltiples uniendo dos vértices.

Una gráfica G está **etiquetada** cuando se asignan nombres distintos a cada uno de sus vértices.

Dos gráficas G y H son **isomorfas** si existe una correspondencia uno a uno entre sus conjuntos de vértices conservando la adyacencia, por lo tanto se puede decir que el isomorfismo es una relación de equivalencia en gráficas.

Una **subgráfica** de G es una gráfica que tiene todos sus vértices y aristas en G; si G_1 es una subgráfica de G entonces G es una **supergráfica** de G_1. Una **subgráfica de tensión** ("spanning subgraph") es una subgráfica que contiene a todos los vértices de G.

La **eliminación de un vértice** v_i de una gráfica G da como resultado una subgráfica $G - v_i$ de G, esto es, una subgráfica que contiene todos los vértices de G excepto v_i y todas las aristas no incidentes con v_i. La **eliminación de una arista** x_j de G da como resultado una subgráfica de tensión $G - x_j$, esto es, una subgráfica que contiene todas la aristas de G excepto x_j.

Un **camino** ("walk") en una gráfica G es una secuencia alterna de vértices y aristas $v_0, x_1, v_1, \ldots, v_{n-1}, x_n, v_n$ que inicia y termina en vértices, en esta secuencia, cada arista es incidente con dos vértices: el que le precede y el que le sigue inmediatamente. Un camino que une a v_0 y a v_n, también puede denotarse por $v_0\ v_1\ v_2 \ldots v_n$. Se dice que un camino es **cerrado** si el vértice inicial es igual al vértice final, en caso contrario se trata de un **camino abierto**; en el caso de que todas las aristas sean distintas se trata de una **ruta** ("trail"); si todos los vértices son distintos se trata de una **trayectoria** ("path"). Cuando el camino es cerrado se dice que es un **ciclo** ("cycle"). La gráfica formada por un ciclo de n vértices se denota por C_n y la trayectoria con n vértices por P_n.

Dada la gráfica etiquetada G de la figura 3.1, un camino del vértice v_1 al vértice v_3 sería $v_1\ v_2\ v_5\ v_2\ v_3$, una ruta para ir del vértice v_1 al v_3 estaría definida por $v_1\ v_2\ v_5\ v_4\ v_2\ v_3$, una trayectoria de v_1 a v_4 sería $v_1\ v_2\ v_5\ v_4$ y un ciclo sería $v_2\ v_4\ v_5\ v_2$.

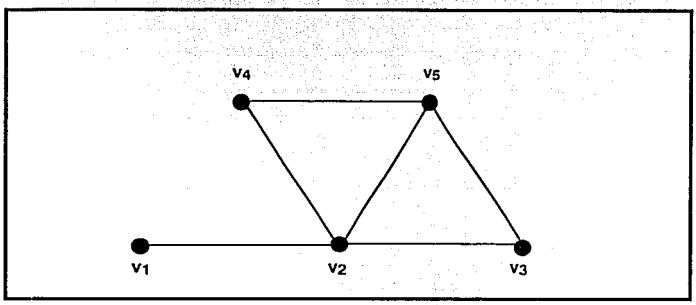


Figura 3.1: Gráfica No Dirigida.

Una gráfica es **conexa** si cada par de vértices están unidos por una trayectoria. Una subgráfica conexa máxima de G se denomina **componente conexo** o simplemente **componente** de G.

La **longitud** de un camino $v_0 v_1 v_2 ... v_n$ es n, esto es, el número de aristas que ocurren en él. El circuito ("girth") de una gráfica G, denotado $g(G)$, es la longitud del ciclo más corto en G; la circunferencia $c(G)$ es la longitud del ciclo más grande.

La **distancia** $d(u,v)$ entre dos vértices u y v en G, es la longitud de la trayectoria más corta que une a ambos vértices, en caso de que no exista alguna trayectoria, entonces $d(u,v) = \infty$.

El **grado de un vértice** v_i en una gráfica G, denotado por d_i o $d(v_i)$, es el número de aristas incidentes en v_i.

Teorema.- La suma de los grados de los vértices de una gráfica G es dos veces el número de sus aristas,

$$\sum_{v \in V(G)} d(v_i) = 2q$$

El **grado de una gráfica** G es la suma del grado de cada uno de sus vértices.

Se dice que el vértice v está **aislado** si el grado de v es igual a cero, denotado por $d(v) = 0$, si $d(v) = 1$ entonces v es un **punto final.**

Se dice que una gráfica es una **gráfica completa** K_p si cada uno de los pares de sus p vértices son adyacentes. La **gráfica complemento** de K_p es totalmente disconexa y regularmente es de grado cero.

Una **bigráfica** G es una gráfica cuyo conjunto de vértices $V(G)$ pueden particionarse en dos subconjuntos $V_1(G)$ y $V_2(G)$ tales que cada arista de G une un vértice de $V_1(G)$ con un vértice de $V_2(G)$. Si G contiene cada arista uniendo $V_1(G)$ y $V_2(G)$ entonces G es una **bigráfica completa**. Si $V_1(G)$ y $V_2(G)$ tienen m y n vértices respectivamente, se denota como $G = (K_{m,n}) = K_{m,n}$.

Se dice que una gráfica es **acíclica** si no tiene ciclos.

Un **árbol** es una gráfica conexa acíclica, la cual es muy utilizada, debido a que es un tipo de gráfica muy simple y permite representar diferentes relaciones del mundo real. Cualquier gráfica sin ciclos es un **bosque**, de este modo, los componentes de un bosque son árboles. Cada árbol no trivial tiene al menos dos vértices finales.

3.2 Gráficas Dirigidas

Una **gráfica dirigida** o **digráfica** D consiste de un conjunto finito, no vacío de vértices $V(G)$ junto con una colección $X(G)$ de pares ordenados de vértices distintos. Los elementos de $X(G)$ son aristas dirigidas o arcos.

Cualquier par de vértices (u,v) se denomina **arco** o **arista dirigida** y se denota como uv. El arco uv va de u hacia v y es incidente con u y v. Así, se dice que u es adyacente a v y que v es adyacente a u.

El **grado positivo** de un vértice v, denotado por $g^+(v)$, es el número de vértices adyacentes que salen de él hacia otro vértice, y el **grado negativo** de un vértice v, denotado por $g^-(v)$, es el número de vértices adyacentes que entran en él provenientes de otro vértice.

Una **gráfica orientada** es una digráfica que no contiene pares simétricos de aristas dirigidas.

Un **camino dirigido** en una digráfica es una secuencia alterna de vértices y arcos, $v_0, x_1, v_1, ..., x_n, v_n$ en la cual cada arco x_i es $v_{i-1}v_i$. La longitud de tal camino es n, esto es, el número de arcos que ocurren en él. Por otro lado, un **semicamino** es una secuencia alterna $v_0, x_1, v_1, ..., x_n, v_n$ de vértices y arcos, pero cada arco x_i puede ser $v_{i-1}v_i$ o v_iv_{i-1}. Si todos los vértices son distintos se trata de una **semitrayectoria**. Cuando el semicamino es cerrado se dice que es un **semiciclo**.

Un camino dirigido es **cerrado** si el primer vértice es igual al último. Un **camino de tensión** es aquel que contiene a todos los vértices de la digráfica. Una **trayectoria dirigida** es un camino dirigido en el cual todos los vértices son distintos. Un **ciclo dirigido** es un camino cerrado no trivial con todos los vértices distintos, excepto el primero y el último.

Si existe una trayectoria de u a v, se dice que v **es alcanzable desde** u, y la **distancia** de u a v, denotada por $d(u,v)$, es la longitud de su trayectoria más corta.

Existen tres tipos de digráficas conexas. Una digráfica es **fuertemente conexa**, si cada dos vértices son mutuamente alcanzables. Si al menos un vértice es alcanzable a partir de otro, entonces se trata de una digráfica **unilateralmente conexa**. La digráfica **débilmente conexa** es aquella en la que dos vértices cualesquiera están unidos por una semitrayectoria.

Un **componente fuertemente conexo** de una digráfica es una subgráfica fuerte máxima, un **componente unilateral** es una subgráfica unilateralmente conexa máxima y un **componente débil** es una subgráfica débilmente conexa máxima.

En la digráfica D de la figura 3.2, se muestra un árbol que representa una estructura familiar, en donde cada vértice representa a un miembro de la familia y cada arco de la digráfica simboliza el progenitor de dicho miembro.

3.3 Gráficas Dirigidas Acíclicas

Una **gráfica dirigida acíclica** o **digráfica acíclica** es una gráfica que no contiene ciclos dirigidos.

Teorema.- Una digráfica acíclica tiene al menos un vértice de grado positivo igual a cero y al menos un vértice de grado negativo igual a cero.

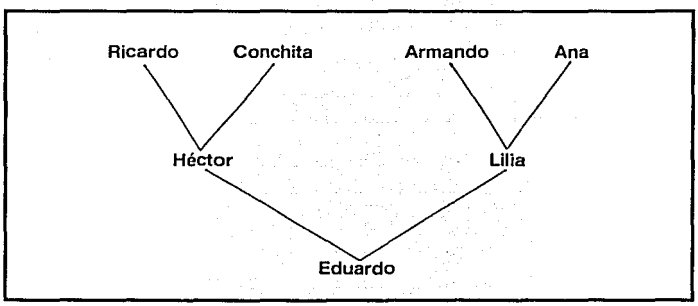


Figura 3.2: Digráfica.

Un **vértice fuente** de una digráfica D es un vértice que puede alcanzar a todos los demás; un **vértice sumidero** es un vértice a partir del cual no se puede alcanzar a todos los demás. Un **árbol saliente** (arborescencia) es una digráfica acíclica con un vértice fuente; un **árbol entrante** es una digráfica acíclica con un vértice sumidero.

Teorema.- Una digráfica débil es un árbol saliente, si y solo si, exactamente solo un vértice tiene grado negativo 0 y todos los demás tienen grado negativo 1.

Teorema.- Una digráfica débil es un árbol entrante, si y solo si, exactamente solo un vértice tiene grado positivo 0 y todos los demás tienen grado positivo 1.

Una **base de vértices** de una digráfica D es una colección mínima de vértices a partir de los cuales todos los vértices de D son alcanzables. Así, un conjunto S de vértices de una digráfica D es un vértice base, si y solo si, cada vértice de D es alcanzable desde un vértice de S y ningún vértice de S es alcanzable desde cualquier otro. Cada digráfica acíclica tiene una base de vértices única que consiste de todos los vértices de grado negativo igual a cero.

3.4 Representación de Gráficas

Una gráfica está totalmente determinada por sus adyacencias o bien, por sus incidencias. Esta información puede representarse por medio de matrices. Así, dada una gráfica debidamente etiquetada, existen diferentes matrices que permiten caracterizar e identificar algunas propiedades de la gráfica.

- **Matriz de Adyacencias**

La **matriz de adyacencias** $A = [a_{ij}]$ de una gráfica etiquetada G con p vértices, es una matriz de p x p en la cual $a_{ij} = 1$ si el vértice v_i es adyacente con el vértice v_j, en caso contrario $a_{ij} = 0$. De esta manera, existe una correspondencia uno a uno entre una gráfica etiquetada con p vértices y la matriz simétrica p x p con diagonal cero. Dado que esta matriz es simétrica es posible representarla como una matriz triangular inferior.

La matriz de adyacencias de una digráfica etiquetada D se define como $A = A(D) = [a_{ij}]$ donde $a_{ij} = 1$ si el arco v_iv_j está en D, en caso contrario $a_{ij} = 0$. $A(D)$ no es necesariamente simétrica.

Para la digráfica D mostrada en la figura 3.2, la matriz de adyacencias correspondiente es:

	Eduardo	Héctor	Lilia	Ricardo	Conchita	Armando	Ana
Eduardo	0	1	1	0	0	0	0
Héctor	0	0	0	1	1	0	0
Lilia	0	0	0	0	0	1	1
Ricardo	0	0	0	0	0	0	0
Conchita	0	0	0	0	0	0	0
Armando	0	0	0	0	0	0	0
Ana	0	0	0	0	0	0	0

Figura 3.3: Matriz de Adyacencias.

- **Matriz de Incidencias**

La matriz de incidencias asociada a una gráfica G de p vértices y q aristas en donde ambos están etiquetados, es una matriz de $p \times q$ que se define como $B = [b_{ij}]$, en donde $b_{ij} = 1$ si b_i y x_j son incidentes, en caso contrario $b_{ij} = 0$.

- **Matriz de Caminos**

La matriz de caminos $W = [w_{ij}]$ de una digráfica D con p vértices, es una matriz de p x p en la cual $w_{ij} = 1$ si existe un camino desde el vértice v_i hasta el vértice v_j, en caso contrario, $w_{ij} = 0$.

Es posible obtener la matriz de caminos W a partir de la matriz de adyacencias A. Una forma de obtener la matriz de caminos es a través del algoritmo de Warshall, el cual se describe a continuación [DöMü72] :

1 . - Sea $A = \{a_{ij}\}$ la matriz de adyacencias binaria asociada a una digráfica G, $W = \{w_{ij}\}$ la matriz de caminos a generar para una digráfica G y n el número de vértices de la digráfica.

2 . - Hacer W = A.

3 . - Hacer j = 1.

4 . - Hacer i = 1.

5 . - Si $w_{ij} = 1$, entonces $w_{ik} = (w_{ik}$ OR $w_{jk})$ para $k = 1, 2, ..., n$.

6 . - i = i + 1; si i ≤ n regresar al paso 5.

7 . - j = j + 1; si j ≤ n regresar al paso 4, en caso contrario, ir al paso 8.

8 . - Terminar.

Para la digráfica D de la figura 3.2, la matriz de caminos correspondiente es:

	Eduardo	Héctor	Lilia	Ricardo	Conchita	Armando	Ana
Eduardo	0	1	1	1	1	1	1
Héctor	0	0	0	1	1	0	0
Lilia	0	0	0	0	0	1	1
Ricardo	0	0	0	0	0	0	0
Conchita	0	0	0	0	0	0	0
Armando	0	0	0	0	0	0	0
Ana	0	0	0	0	0	0	0

Figura 3.4: Matriz de Caminos.

- **Matriz de Ciclos**

  Una matriz de ciclos $C = [c_{ij}]$ de G tiene un renglón para cada ciclo y una columna para cada arco, con $c_{ij} = 1$ si el i-ésimo ciclo contiene el arco x_j, en caso contrario $c_{ij} = 0$.

  Es importante señalar, que a diferencia de las matrices de adyacencia y de incidencia, las matrices de caminos y de ciclos no determinan a una gráfica G.

Capítulo 4

NORMALIZACION DE REGLAS Y MANEJO DE REGLAS RECURSIVAS

En la primera sección de este capítulo se presenta la arquitectura general del sistema particularizando en los módulos para este segundo prototipo.

Se dice que un sistema de reglas no está normalizado cuando varias reglas tienen el mismo predicado como consecuente. Por cuestiones técnicas, es muy importante normalizar el sistema, esto es, que cada una de las reglas tenga un predicado distinto como consecuente. Es así, que al realizar una consulta, se puede seleccionar una sola regla y expanderla en el árbol de derivación correspondiente para evaluar la consulta utilizando la base de datos extensional. La metodología para la normalización de reglas se analiza en la sección 4.2.

El proceso de deducción se puede realizar de diferentes formas por lo que en la sección 4.3 se presentan los métodos para llevar a cabo la deducción para reglas no recursivas. En la sección 4.4 se define la gráfica de dependencias, la cual sirve para representar las dependencias entre los predicados que conforman las reglas de un programa lógico.

De manera informal se dice que una regla es recursiva si dicha regla está definida en términos de sí misma en forma directa o indirecta. Dado que es posible encontrar reglas recursivas en un programa lógico, es importante contar con un buen algoritmo para la detección de ciclos así como para la evaluación de reglas recursivas, de lo contrario, el árbol de derivación que incluyera alguna regla recursiva crecería indefinidamente y no se obtendría ninguna solución. En las dos últimas secciones se presentan los algoritmos para la detección y evaluación de reglas recursivas.

4.1 Arquitectura General del Sistema

En la figura 4.1 se muestra la arquitectura del sistema objeto de esta tesis, la cual corresponde a un subconjunto de la arquitectura mostrada en el capítulo 1. Los principales componentes del sistema son: lenguaje orientado a reglas, parser, monitor de transacciones e intérprete.

Como se mencionó en el capítulo 1, los objetivos para este segundo prototipo se limitan a la normalización de reglas, a la detección de reglas recursivas así como a la evaluación de las mismas. Sin embargo, en esta sección también se describen los componentes desarrollados en el primer prototipo.

El lenguaje orientado a reglas consta de cuatro objetos: funciones, relaciones, reglas de deducción y restricciones, de los cuales para este trabajo sólo se consideran relaciones y reglas. El lenguaje utilizado permite generar una base de datos convencional, si el programa sólo incluye relaciones; y permite generar un sistema deductivo, si el programa incluye reglas deductivas además de relaciones.

El parser efectúa tanto el análisis léxico y el análisis sintáctico de un programa, como la generación de la representación interna de las reglas y las estructuras necesarias para la definición de la base de datos extensional. Además, el parser lleva a cabo la normalización del sistema de reglas así como la detección de recursividad.

El monitor de transacciones está compuesto por dos submódulos: el monitor y la interfaz con el manejador de bases de datos relacional (RDBMSI). El monitor transfiere el control al módulo correspondiente dependiendo de la operación que se realice. Durante la fase de deducción el control se transfiere al intérprete. La RDBMSI se emplea para crear las relaciones base, insertar, borrar y modificar la información de la base de datos, evaluar consultas y evaluar árboles de derivación.

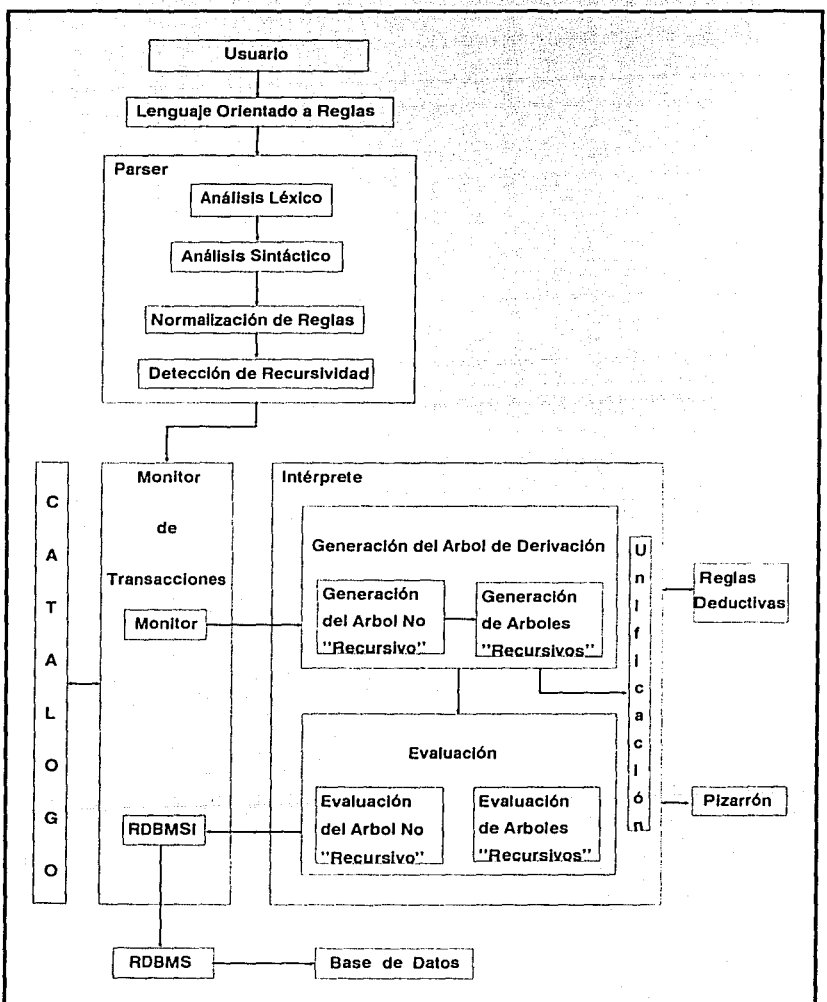


Figura 4.1: Arquitectura del Sistema

El intérprete es el núcleo del sistema, y constituye un extensión de un sistema manejador de base de datos relacional para soportar la deducción. El intérprete se encarga de deducir conocimientos de hechos almacenados en la base de datos. La función del intérprete es llevar a cabo la generación del árbol de derivación, la unificación(1) y la evaluación. El generador del árbol de derivación se compone del generador del árbol no "recursivo" y el generador de árboles "recursivos", mientras que el submódulo de evaluación está formado por la evaluación del árbol no "recursivo" y por la evaluación de árboles "recursivos".

El árbol se construye para una consulta empleando el conjunto de reglas deductivas; en el caso de existir reglas recursivas, se generan los árboles "recursivos" en forma independiente. El submódulo de unificación obtiene el unificador más general(2) para dos átomos si existe, en caso contrario termina con falla. El submódulo de evaluación de árboles "recursivos" utiliza un algoritmo iterativo, mientras que la evaluación de árboles no "recursivos" obtiene una expresión equivalente al árbol de derivación en SQL y efectúa la evaluación de la expresión obtenida.

El pizarrón es el área de trabajo que se utiliza para almacenar tanto resultados intermedios como el árbol de derivación.

4.2 Normalización de Reglas No Recursivas

Se dice que un sistema de reglas no se encuentra normalizado si existe más de una regla con el mismo predicado como consecuente. Por razones técnicas es conveniente **normalizar** el sistema de reglas, esto es, distinguir cada una de las reglas por predicados distintos en el consecuente. Existen dos métodos para la normalización de reglas a los cuales denominaremos: normalización por reducción y normalización por extensión.

(1) Una sustitución es un conjunto finito de la forma $\{t1/v1, ..., tn/vn\}$ $n \geq 0$, donde cada vi , $i = 1, ..., n$ es una variable y cada ti, $i = 1, ..., n$, es un término diferente de vi, tal que no existen dos elementos en el conjunto con la misma variable vj después de la diagonal.
Una sustitución θ se denomina un **unificador** para un conjunto $W = \{E1, .., Ek\}$, si y solo si, $E1\,\theta = E2\,\theta = ... = Ek\,\theta$.
Si existe un unificador para W se dice que el conjunto es unificable.

(2) Un unificador σ para un conjunto de expresiones $W = \{E1, ..., Ek\}$ es un unificador más general, si y solo si, para cada unificador θ de W, existe una sustitución λ tal que $\theta = \sigma \circ \lambda$.

En ambos métodos es necesario llevar a cabo un proceso de rectificación. Esta rectificación de reglas se requiere debido a que es posible tener reglas donde los términos del predicado del consecuente son símbolos de constantes, símbolos de funciones o bien variables repetidas. Se dice que un predicado está rectificado si para todas las reglas con predicados del consecuente iguales, se tiene la forma $P(x_1, ..., x_k)$ donde las variables $x_1, ..., x_k$ son distintas [Ullm88].

La rectificación consiste en introducir nuevas variables para cada uno de los argumentos del predicado del consecuente, e introducir en el cuerpo de la regla submetas preconstruidas, tales como predicados aritméticos, que satisfagan cualesquiera de las restricciones que el predicado del consecuente introduce por medio de constantes o por la repetición de variables.

Supóngase que se tiene una regla R con consecuente $P(y_1, ..., y_k)$ donde cada y_i puede ser un símbolo de función, un símbolo de constante o una variable que puede repetirse. Se reemplaza el consecuente de R por $P(x_1, ..., x_k)$, donde cada x es una nueva variable distinta a cualquier otra, y a R se le adicionan las submetas $x_i = y_i$, $1 \leq i \leq k$. Si y_i es una variable, se puede eliminar la submeta $x_i = y_i$ y sustituir las x_i que aparezcan en la regla por la y_i. Es importante notar que cuando se hace una sustitución para y_i, no se puede hacer después otra sustitución para la misma variable y_i. Por ejemplo, dadas las reglas

```
R1: P(a,x,y) ← R(x,y)

R2: P(x,y,x) ← R(y,x)
```

al rectificar las reglas se tiene

```
R1: P(u,v,w) ← R(x,y) & u=a & v=x & w=y

R2: P(u,v,w) ← R(y,x) & u=x & v=y & w=x
```

eliminando las submetas asociadas a variables se tiene

```
R1: P(u,v,w) ← R(v,w) & u=a

R2: P(u,v,w) ← R(v,u) & w=u
```

4.2.1 Normalización por Reducción

Para explicar el concepto de la normalización por reducción de reglas no recursivas, se empleará el siguiente ejemplo: sea el sistema de reglas que define una estructura familiar, esto es, el parentesco que existe entre los diferentes miembros de una familia.

Sistema de reglas NO normalizado:

```
R1: Padre(x,y)       ←
R2: Madre(x,y)       ←
R3: Esposo(x,y)      ←
R4: Progenitor(x,y)  ← Padre(x,y) V Madre(x,y)
R5: Abuela(x,y)      ← Madre(x,z) & Madre(z,y)
R6: Abuela(u,v)      ← Madre(u,w) & Padre(w,v)
R7: Abuelo(x,y)      ← Esposo(x,z) & Abuela(z,y)
```

El sistema cuenta con tres relaciones base y cuatro reglas de deducción. El significado que se les da a las reglas es similar en todas, por ejemplo:

`Padre(x,y)` se lee: *x es padre de y.*

En este método se realiza una rectificación de las reglas para después unificarlas. Posteriormente se forma una sola regla a través de la disyunción lógica de los antecedentes de las reglas a normalizar.

En el sistema de reglas anterior, existen dos reglas a normalizar: *R5* y *R6* cuyo predicado del consecuente es *Abuela*.

```
R5: Abuela(x,y) ← Madre(x,z) & Madre(z,y)
R6: Abuela(u,v) ← Madre(u,w) & Padre(w,v).
```

Al efectuar la normalización, se obtendrá una sola regla para *Abuela (R5)*, la cual se formará por la disyunción de los antecedentes de cada una de las reglas.

```
R5: Abuela(x,y) ← (Madre(x,z) & Madre(z,y)) V
                  (Madre(x,z) & Padre(z,y))
```

La regla R_6 se elimina del sistema de reglas por lo que el sistema queda como sigue:

Sistema de reglas normalizado:

```
R1: Padre(x,y)        ←
R2: Madre(x,y)        ←
R3: Esposo(x,y)       ←
R4: Progenitor(x,y)   ← Padre(x,y) V Madre(x,y)
R5: Abuela(x,y)       ← (Madre(x,z) & Madre(z,y)) V
                        (Madre(x,z) & Padre(z,y))
R7: Abuelo(x,y)       ← Esposo(x,z) & Abuela(z,y)
```

La necesidad de rectificar las reglas es muy clara en casos como los que se muestran a continuación:

- **Caso 1:** Dadas las siguientes reglas

```
R1: Q(x,y,a) ← P1(x,z) & P2(z,y)
R2: Q(u,v,b) ← P3(u,w) & P4(w,v)
```

R_1 y R_2 no unifican debido a que *a* y *b* son constantes. Rectificando las reglas se tiene

```
R1': Q(x,y,x1) ← P1(x,z) & P2(z,y) & x1=a
R2': Q(u,v,x2) ← P3(u,w) & P4(w,v) & x2=b
```

de este modo R_1 y R_2 unifican por lo cual ya puede llevarse a cabo la normalización.

- **Caso 2:** Dadas las siguientes reglas

```
R1: Q(x,f(x)) ← P1(x,y)
R2: Q(u,u)    ← P2(u,u)
```

R_1 y R_2 no unifican debido a que se tiene el problema de ocurrencia de la variable en el símbolo de función que la sustituye. Rectificando las reglas se tiene

R_1': Q(x,z) ← P_1(x,y) & z=f(x)

R_2': Q(u,v) ← P_2(u,v) & v=u

así R_1 y R_2 unifican por lo cual ya puede llevarse a cabo la normalización.

■ **Caso 3:** Dadas las siguientes reglas

R_1: Q(x,a) ← P_1(x,y)

R_2: Q(u,u) ← P_2(u,u)

R_1 y R_2 si unifican pero al normalizar, la regla que se obtiene no es equivalente

R_1': Q(a,a) ← P_1(a,y) V P_2(a,a)

esto implica que es necesario rectificar las reglas R_1 y R_2 antes de normalizarlas

R_1': Q(x,z) ← P_1(x,y) & z=a

R_2': Q(u,v) ← P_2(u,u) & v=u

de tal forma que R_1' y R_2' unifican y al normalizar la regla resultante es equivalente.

Este método se resume en los siguientes pasos:

1.- Rectificar las reglas.

2.- Unificar los consecuentes de las reglas a normalizar.

3.- Realizar la sustitución de variables.

4.- Formar una sola regla a través de la disyunción lógica de los antecedentes de las reglas a normalizar.

4.1.2 Normalización por Extensión

Este segundo método de normalización, que es el que se utiliza en este trabajo, resulta más simple que el anterior. Aunque en este método también se requiere del proceso de rectificación de reglas, en este trabajo no se lleva a cabo.

Como se vio en el sistema de reglas no normalizado del ejemplo anterior, es necesario normalizar las reglas *R5* y *R6*, cuyo predicado del consecuente es *Abuela*. Bajo este segundo método de normalización, se crea la regla (*R8*), la cual tendrá como antecedente la disyunción de los dos predicados a normalizar, los cuales fueron renombrados previamente.

```
R5: Abuela_Materna(x,y) ← Madre(x,z) & Madre(z,y)

R6: Abuela_Paterna(x,y) ← Madre(x,z) & Padre(z,y)

R8: Abuela(u,v)         ← Abuela_Materna(u,v) V
                          Abuela_Paterna(u,v)
```

De esta manera el sistema de reglas normalizado, bajo este método, queda como sigue:

Sistema de reglas normalizado:

```
R1: Padre(x,y)          ←

R2: Madre(x,y)          ←

R3: Esposo(x,y)         ←

R4: Progenitor(x,y)     ← Padre(x,y) V Madre(x,y)

R5: Abuela_Materna(x,y) ← Madre(x,z) & Madre(z,y)

R6: Abuela_Paterna(x,y) ← Madre(x,z) & Padre(z,y)

R7: Abuelo(x,y)         ← Esposo(x,z) &
                          Abuela(z,y)

R8: Abuela(u,v)         ← Abuela_Materna(u,v) V
                          Abuela_Paterna(u,v)
```

Este proceso es más simple que el método de reducción debido a que no es necesario unificar porque se crea una nueva regla con nuevos términos y se renombran las reglas a normalizar.

Dado un conjunto de reglas que tienen el mismo predicado de orden *n* como consecuente, el método se resume en los siguientes pasos:

1.- Generar una nueva regla cuyo consecuente estará formado por el predicado que se encuentre en el consecuente del conjunto de reglas y por *n* variables nuevas.

2.- Renombrar cada uno de los predicados de los consecuentes del conjunto de reglas.

3.- Formar el antecedente de la regla generada como una disyunción de los predicados renombrados donde cada átomo tendrá las *n* variables que aparecen en el consecuente de la nueva regla.

4.3 Proceso de Deducción para Reglas No Recursivas Normalizadas

En las bases de datos deductivas existen dos métodos para realizar el proceso de deducción: el **método interpretativo** y el **método compilado**. En el primero se combina el uso de las leyes deductivas con la búsqueda en la base de datos extensional, mientras que en el método compilado se retrasa el acceso a la base de datos hasta que se han concluido las transformaciones. Ambos métodos serán descritos por medio de un ejemplo.

4.3.1 Método Interpretativo

Considérese la siguiente instancia de una base de datos.

Padre	
Papá	Hijo
Héctor	Armando
Héctor	Eduardo
Héctor	Gabriel

Madre	
Mamá	Hijo
Conchita	Héctor
Conchita	Sergio
Conchita	Bertha
Bertha	Patricia

Esposo	
Esposo	Esposa
Ricardo	Conchita

representada por las siguientes cláusulas:

```
C1: Padre(Héctor,Armando)      ←
C2: Padre(Héctor,Eduardo)      ←
C3: Padre(Héctor,Gabriel)      ←
C4: Madre(Conchita,Héctor)     ←
C5: Madre(Conchita,Sergio)     ←
C6: Madre(Conchita,Bertha)     ←
C7: Madre(Bertha,Patricia)     ←
C8: Esposo(Ricardo,Conchita)   ←
```

Sean las siguientes reglas de deducción del ejemplo anterior:

```
R1: Abuela(x,y) ← (Madre(x,z) & Madre(z,y)) V
                  (Madre(x,z) & Padre(z,y))
R2: Abuelo(x,y) ← Abuela(z,y) & Esposo(x,z)
```

Supóngase que se desea conocer de quién es abuelo *Ricardo*, lo cual se representa con la cláusula:

```
Q: ← Abuelo(Ricardo,y);
```

la cual unifica con R_2 obteniendo la sustitución $\{x / Ricardo\}$ para generar el resolvente:

```
C9: ← Abuela(z,y) & Esposo(Ricardo,z)
```

resolviendo primero para *Esposo(Ricardo,z)* el cual unifica con C_8, se obtiene:

```
C10: ← Abuela(Conchita,y)
```

para resolver esta cláusula se aplica R_1:

```
C11: ← (Madre(Conchita,z) & Madre(z,y)) V
       (Madre(Conchita,z) & Padre(z,y))
```

al unificar con C_4 se obtiene el primer valor para z, z = *Héctor*; dado que no existe una cláusula que unifique con *Madre(Héctor,y)*, solo queda por resolver:

```
C12: ← Padre(Héctor,y)
```

la cual unifica con C_1, para obtener y = *Armando*. De manera similar, unificando C_{12} con C_2 se obtiene un segundo valor para y, y = *Eduardo*, y unificando C_{12} con C_3 se obtiene y = *Gabriel*.

Nuevamente al tomar C_{11} para unificar con C_5 se obtiene otro valor para z, z = *Sergio*, generando el resolvente:

```
C13: ← Padre(Sergio,y)
```

el cual no unifica con ninguna cláusula, de igual manera al unificar C_{11} con C_6 se obtiene para z el valor z = *Bertha*, quedando:

```
C14: ← Madre(Bertha,y)
```

donde se obtiene un valor más para y, y = *Patricia*.

Resumiendo lo anterior, se puede concluir que los valores obtenidos para la variable y constituyen el resultado a la consulta formulada, de quién es abuelo *Ricardo*, por lo tanto la solución es:

```
y = Armando
y = Eduardo
y = Gabriel
y = Patricia.
```

4.3.2 Método Compilado

Existen dos variantes del método compilado para efectuar la deducción: pseudo-compilación y compilación. La técnica de **pseudo-compilación** consiste en compilar sólo un camino a la vez por donde es probable encontrar la solución, y en caso de no obtener la solución por ese camino se compila otro. La técnica de **compilación** consiste en compilar todos los caminos. Por ejemplo, considérese la misma base de datos del ejemplo anterior con la misma consulta.

```
Q: ← Abuelo(Ricardo,y);
```

Unificando Q con R_2 y sustituyendo $\{x / Ricardo\}$ se obtiene la expresión:

```
E1: ← Abuela(z,y) & Esposo(Ricardo,z)
```

para el predicado *Abuela(z,y)* se emplea R_1, y aplicando la sustitución $\{z / z_1\}$ se obtiene:

```
E2: ← ((Madre(z,z1) & Madre(z1,y)) V
       (Madre(z,z1) & Padre(z1,y))) &
        Esposo(Ricardo,z)
```

dado que la expresión E_2 sólo contiene relaciones extensionales, ésta se evalúa en la base de datos.

El árbol de derivación que representa esta expresión se muestra en la siguiente figura:

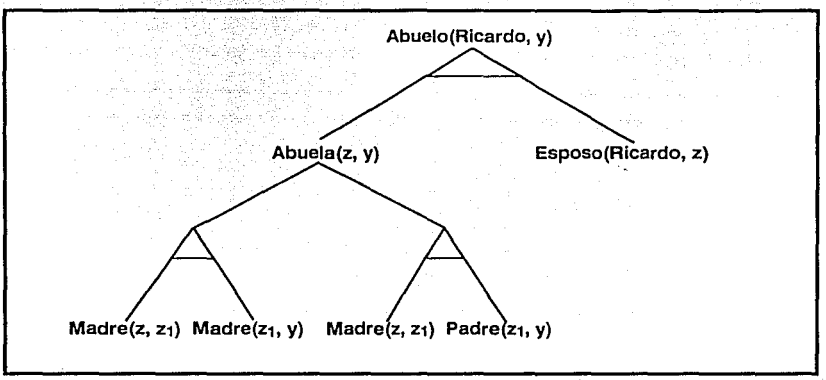


Figura 4.2: Arbol de Derivación.

4.4 Gráfica de Dependencias

Frecuentemente es necesario definir la forma en que los predicados dependen unos de otros en un programa lógico(3) [Lloy84]. Se dice que un predicado Q depende de un predicado P, si existe una regla en la cual Q aparece como consecuente y P aparece en el antecedente de dicha regla. Es posible representar la dependencia entre predicados a través de una **gráfica de dependencias**, cuyos vértices son los predicados del programa y los arcos representan las dependencias que existen entre los diferentes predicados [Ullm88].

Por ejemplo, para el siguiente sistema de reglas, en la figura 4.3 se muestra la gráfica de dependencias correspondiente:

Sistema de Reglas

R_1: Padre(x,y) ←

R_2: Madre(x,y) ←

(3) Dada una cláusula de Horn de la forma $Q \leftarrow P_1 \,\&\, \ldots \,\&\, P_n$, a cada uno de los P_i se le denomina una **submeta**. Una colección finita de cláusulas de Horn constituye un **programa lógico**.

```
R3: Hermano(x,y)          ←
R4: Esposo(x,y)           ←
R5: Progenitor(x,y)       ← Padre(x,y) V Madre(x,y)
R6: Ancestro(x,y)         ← Progenitor(x,y) V
                            Progenitor(x,z) &
                            Ancestro(z,y)
R7: Tío(x,y)              ← Hermano(x,z) &
                            Progenitor(z,y)
R8: Abuela_Materna(x,y)   ← Madre(x,z) & Madre(z,y)
R9: Abuela_Paterna(x,y)   ← Madre(x,z) & Padre(z,y)
R10: Abuelo(x,y)          ← Esposo(x,z) &
                            Abuela(z,y)
R11: Abuela(u,v)          ← Abuela_Materna(u,v) V
                            Abuela_Paterna(u,v)
```

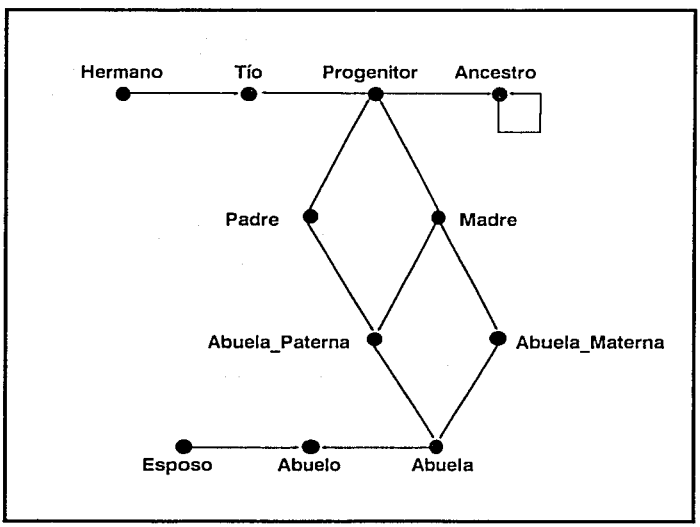


Figura 4.3: Gráfica de Dependencias.

4.5 Detección de Reglas Recursivas

Un programa lógico es **recursivo** si la gráfica de dependencias correspondiente a dicho programa tiene uno o más ciclos. Todos los predicados que están en uno o más ciclos se denominan **predicados recursivos**. Un programa lógico con una gráfica de dependencias acíclica es un programa no recursivo. Se dice que un predicado no es recursivo si, aún cuando esté en un programa recursivo, dicho predicado no es parte de algún ciclo en la gráfica de dependencias.

A cada predicado le corresponde una relación. Si el predicado está definido exclusivamente en forma extensional, entonces el predicado es **extensional puro**. Si el predicado está definido exclusivamente en forma intencional, entonces el predicado es **intencional puro**. Es posible encontrar predicados definidos tanto en forma extensional como intencional.

Los predicados extensionales puros son aquellos cuyos nodos no tienen arcos entrantes en la gráfica de dependencias, lo cual implica que no pueden ser recursivos.

Como se señaló en el capítulo 3, una gráfica puede representarse por medio de una matriz de adyacencias o por medio de una matriz de incidencias. En este trabajo se hará uso de la matriz de adyacencias para representar la gráfica de dependencias dado que la matriz de adyacencias siempre es cuadrada mientras que la matriz de incidencias no siempre lo es, y resulta más sencillo manipular una matriz cuadrada que una que no lo es. De este modo, las columnas y renglones de la matriz de adyacencias representan cada uno de los predicados que constituyen un sistema de reglas.

Para la gráfica de dependencias mostrada en la figura 4.3, la matriz de adyacencias asociada se presenta en la figura 4.4.

En un sistema de reglas se pueden tener reglas directamente recursivas y reglas indirectamente recursivas. Un predicado puede estar definido por varias reglas, algunas de las cuales pueden ser recursivas, por ejemplo:

```
R1: Ancestro(x,y) ← Progenitor(x,y)

R2: Ancestro(x,y) ← Progenitor(x,z) &
                    Ancestro(z,y)
```

	Padre	Madre	Hermano	Esposo	Progenitor	Ancestro	Tío	Abuelo_Materno	Abuelo_Paterno	Abuelo	Abuelo
Padre	0	0	0	0	1	0	0	0	1	0	0
Madre	0	0	0	0	1	0	0	1	1	0	0
Hermano	0	0	0	0	0	0	1	0	0	0	0
Esposo	0	0	0	0	0	0	0	0	0	1	0
Progenitor	0	0	0	0	0	1	1	0	0	0	0
Ancestro	0	0	0	0	0	1	0	0	0	0	0
Tío	0	0	0	0	0	0	0	0	0	0	0
Abuela_Materna	0	0	0	0	0	0	0	0	0	0	1
Abuela_Paterna	0	0	0	0	0	0	0	0	0	0	1
Abuelo	0	0	0	0	0	0	0	0	0	0	0
Abuela	0	0	0	0	0	0	0	0	0	1	0

Figura 4.4: Matriz de Adyacencias Correspondiente a la Gráfica de Dependencias de la Figura 4.3.

en este caso se procede a normalizar las reglas de la misma forma en que se normalizan las reglas no recursivas (sección 4.2):

```
R1: Anc_1(x,y)      ← Progenitor(x,y)

R2: Anc_2(x,y)      ← Progenitor(x,z) &
                      Ancestro(z,y)

R3: Ancestro(x,y) ← Anc_1(x,y) V Anc_2(x,y).
```

Este proceso de normalización se lleva a cabo al mismo tiempo que la normalización de reglas no recursivas.

4.5.1 Detección de Reglas Directamente Recursivas.

Una regla **directamente recursiva** es aquella en la cual el predicado del consecuente también aparece en el antecedente (en una digráfica equivale a un enlace). Un enlace, en la matriz de adyacencias, se representa por un 1 en la posición $a_{i,i}$ para la regla i.

En el caso de las reglas directamente recursivas, cada regla se divide en dos partes: la parte de condiciones iniciales y la parte recursiva. Es posible encontrar reglas que tengan más de una submeta recursiva, por ejemplo

```
R1: Ancestro(x,y) ← Progenitor(x,y)

R2: Ancestro(x,y) ← Ancestro(x,z) &
                    Ancestro(z,y).
```

Aunque el algoritmo de transformada delta soporta la evaluación de este tipo de reglas, en este trabajo no se considera su implantación, debido a que se requiere realizar varias transformaciones sobre la regla.

Para las reglas directamente recursivas, es necesario aplicar un proceso de normalización, el cual consiste en crear un nueva regla cuyo antecedente se formará con la submeta que contenga el predicado recursivo. Al consecuente de esta nueva regla se le asignará un nombre que no haya sido utilizado previamente en el sistema de reglas; en la regla directamente recursiva, se substituirá la submeta que contenía el predicado recursivo por el consecuente de la nueva regla. Por ejemplo, considérese el siguiente sistema de reglas:

Sistema de reglas

```
R1: Padre(x,y)      ←
R2: Madre(x,y)      ←
R3: Progenitor(x,y) ← Padre(x,y) V Madre(x,y)
R4: Ancestro(x,y)   ← Progenitor(x,y) V
                      (Progenitor(x,z) &
                       Ancestro(z,y))
```

la matriz de adyacencias correspondiente es:

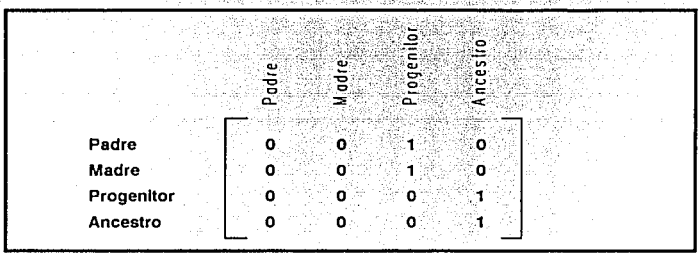


Figura 4.5: Matriz de Adyacencias Correspondiente al Sistema de Reglas que Contiene Reglas Directamente Recursivas.

Como se observa en la matriz (figura 4.5), el elemento $a_{4,4} = 1$, lo cual indica que la regla R_4 es directamente recursiva. Al normalizar esta regla se generará una nueva regla $A1$ (R_5) cuyo antecedente se formará por la segunda submeta de la disyunción lógica del antecedente de la regla *Ancestro* (parte recursiva), y en su lugar quedará la submeta $A1(x,y)$, misma que contendrá los mismos términos que el predicado del consecuente de *Ancestro*, de tal forma que ambas reglas quedan como sigue:

```
R4: Ancestro(x,y) ← Progenitor(x,y) V A1(x,y)

R5: A1(x,y)       ← Progenitor(x,z) &
                    Ancestro(z,y).
```

Es importante señalar que el predicado *Ancestro* sigue siendo recursivo a través de la regla $A1$ aunque ninguna de las reglas es directamente recursiva.

Este proceso se aplica a todas las reglas que sean directamente recursivas. El sistema de reglas normalizado equivalente del ejemplo anterior es:

Sistema de reglas normalizado equivalente:

```
R1: Padre(x,y)       ←

R2: Madre(x,y)       ←

R3: Progenitor(x,y)  ← Padre(x,y) V Madre(x,y)

R4: Ancestro(x,y)    ← Progenitor(x,y) V A1(x,y)

R5: A1(x,y)          ← Progenitor(x,z) &
                       Ancestro(z,y)
```

La gráfica de dependencias correspondiente es:

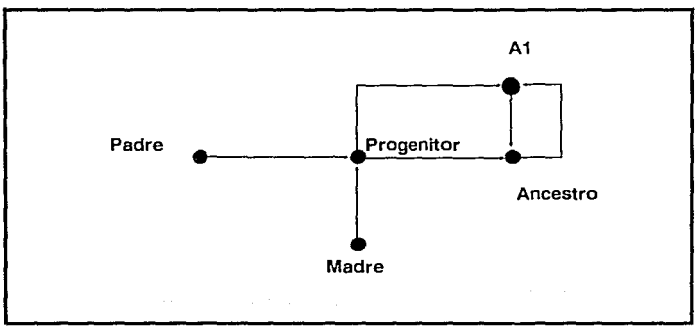


Figura 4.6: Gráfica de Dependencias Correspondiente al Sistema de Reglas Normalizado Equivalente.

y la matriz de adyacencias correspondiente a la gráfica de dependencias es:

	Padre	Madre	Progenitor	Ancestro	A1
Padre	0	0	1	0	0
Madre	0	0	1	0	0
Progenitor	0	0	0	1	1
Ancestro	0	0	0	0	1
A1	0	0	0	1	0

Figura 4.7: Matriz de Adyacencias Correspondiente al Sistema de Reglas Normalizado Equivalente.

4.5.2 Detección de Ciclos

Un conjunto de reglas pueden ser recursivas a través de ciclos que incluyen varios predicados, esto es, la gráfica de dependencias presenta ciclos multinodos.

Existe un algoritmo para la detección de ciclos[4] a partir de la matriz de adyacencias y la matriz de caminos asociadas a la gráfica de dependencias del sistema, el algoritmo es el siguiente [DöMü72] :

1. Sea A = $\{a_{ij}\}$ la matriz de adyacencias asociada a un sistema de reglas, W = $\{w_{ij}\}$ la matriz de caminos asociada a dicho sistema, *n* el orden de la matriz de adyacencias, *V[n]* el vector de marcas, *R[n]* el vector de bifurcación, *T[n+1]* el vector de ciclos, *b* el número de bifurcación, *p* el índice del menor vértice a probar y *m* el vértice actual.

2. Hacer i = 1.

3. Mientras i ≤ n

   3.1 Inicializar el vector R como R_k = i, donde (k = i ... n).

   3.2 Inicializar el vector V como V_k = 0, donde (k = i ... n).

   3.3 Hacer b = 1, T[1] = i, m = i.

   3.4 Mientras (b > 0)

      3.4.1 Realizar

         3.4.1.1 Hacer p = R[m].

         3.4.1.2 Si (p ≤ n), entonces

            3.4.1.2.1 Hacer j = R[m].

            3.4.1.2.2 Mientras (j ≤ n) & (vértice j no está en el ciclo)

               3.4.1.2.2.1 Incrementar j en 1.

(4) El algoritmo original se encuentra codificado en FORTRAN, mismo que fue reestructurado por Cristóbal Juárez C.

3.4.1.2.3 Si (j ≤ n), entonces

3.4.1.2.3.1 Hacer V[j] = 1,
R[m] = j + 1,
b = b + 1,
T[b] = j,
m = j.

mientras (j <> i) & (j ≤ n) & (p ≤ n)

3.4.2 Si (j = i), entonces

3.4.2.1 Almacenar el ciclo.

en caso contrario

3.4.2.2 Hacer R[m] = i.

3.4.3 Hacer V[m] = 0, b = b - 1.

3.4.4 Si (b = 0), entonces

3.4.4.1 Hacer m = 0.

en caso contrario

3.4.4.2 Hacer m = T[b].

3.5 Hacer i = i + 1.

4. Terminar.

4.6 Transformada Delta

Cuando un sistema de reglas contiene reglas recursivas, el árbol de derivación crece en forma infinita sin poder llegar a un punto terminal, lo cual ocasiona que se agote la memoria y la consulta termine en forma anormal.

Existen diversos algoritmos para evaluar reglas recursivas, uno de ellos es el algoritmo de transformada delta [Baye85], el cual consiste en evaluar las reglas recursivas en forma iterativa hasta que se presente el punto fijo, es decir, cuando las relaciones derivadas de las reglas recursivas ya no presenten cambios al aplicarse el algoritmo.

Sea la regla de deducción r, formada por las submetas $S_1, ..., S_n$ que contienen las variables $X_1, ..., X_m$. Para cada $S_i = p_i(A_{i1}, ..., A_{ik})$, donde p_i es un predicado ordinario, existe una relación R_i previamente calculada, donde las A_i son argumentos (variables o símbolos de constantes). Una **ecuación equivalente** al antecedente de la regla r, es una expresión del álgebra relacional tal que a partir de las relaciones $R_1, ..., R_n$, se puede computar una relación $R(X_1, ..., X_m)$ que contiene a todos y cada uno de los tuplos $(a_1, ..., a_m)$, de tal forma que al sustituir X_j por a_j, $1 \leq j \leq m$, todas las submetas $S_1, ..., S_n$ son verdaderas.

Dado el conjunto de relaciones $R_1, ..., R_k$ para los predicados correspondientes a la base de datos extensional, un **punto fijo** de las ecuaciones equivalentes (con respecto a $R_1, ..., R_k$), es una solución para las relaciones correspondientes a los predicados de la base de datos intencional de dichas ecuaciones.

Sean $P_1, ..., P_m$ las variables de las ecuaciones correspondientes a los predicados $p_1, ..., p_m$ de la base de datos intencional, y sean $R_1, ..., R_k$ las relaciones asignadas a los predicados de la base de datos extensional $r_1, ..., r_k$. Una solución o punto fijo, para las relaciones $R_1, ..., R_k$ asigna a $P_1, ..., P_m$ relaciones particulares $P_1^{(1)}, ..., P_m^{(1)}$ tales que las ecuaciones se satisfacen. Si $S_1 = P_1^{(1)}, ..., P_m^{(1)}$ y $S_2 = P_1^{(2)}, ..., P_m^{(2)}$ son dos soluciones para el conjunto de ecuaciones dado, se dice que $S_1 \leq S_2$ si la relación $P_i^{(1)}$ es un subconjunto de la relación $P_i^{(2)}$, $1 \leq i \leq m$.

Se puede resolver un conjunto de ecuaciones asumiendo inicialmente que todas las P_i están vacías y que las relaciones R_i han sido dadas. Utilizando los valores actuales de las relaciones de la base de datos intencional con los valores de las relaciones de la base de datos extensional, se obtienen nuevos valores para las relaciones de la base de datos intencional. Este proceso se repite hasta que no haya cambios en cada P_i [Ullm88].

El concepto básico del algoritmo de **transformada delta** para evaluar en forma eficiente las reglas recursivas, es que solo los cambios reales de una relación R, denotados por ΔR, se llevan de una iteración a otra; las iteraciones terminan cuando ya no ocurre ningún cambio (las relaciones ΔR permanecen vacías). El proceso de las transformaciones de una iteración a otra se basa en la monotonía de las operaciones tanto del OR lógico como del "join" natural.

Para explicar el algoritmo se utilizará la regla recursiva *Ancestro* del ejemplo anterior (figura 4.6):

```
Ancestro(x,y) ← A1(x,y) V Progenitor(x,y)
A1(x,y)       ← Ancestro(x,z) & Progenitor(z,y)
```

Las relaciones asociadas a los predicados de la base de datos intencional *Ancestro* y *A1* están inicialmente vacías. *Progenitor* se obtiene de la primera evaluación de *Padre* y *Madre*, que corresponde a las condiciones iniciales, asignando el resultado a *Ancestro*.

Para evaluar el sistema de reglas mostrado anteriormente, en la transformada delta se considera un esquema de iteraciones donde el operador de implicación (←) se sustituye por el operador de asignación (:=) y se asignan índices a las relaciones derivadas. Para los cálculos siguientes se denotará *Ancestro* por *A* y *Progenitor* por *P*, las variables no se emplearán y la operación "join" se denotará con el operador asterisco (*); así, se tienen las siguientes reglas:

(1) $A_{i+1} := A1_i \vee P$

$A1_{i+1} := A_i * P$

Como las operaciones utilizadas (OR lógico y "join" natural) son monótonas, $A1_i$ y A_i pueden denotarse como expresiones del tipo:

(2) $A_i = A_{i-1} \vee \Delta A_i$; $\Delta A_i = A_i - A_{i-1}$

$A1_i = A1_{i-1} \vee \Delta A1_i$; $\Delta A1_i = A1_i - A1_{i-1}$

Sustituyendo (2) en (1) se obtiene:

(3) $A_{i+1} := (A1_{i-1} \vee \Delta A1_i) \vee P = (A1_{i-1} \vee P) \vee \Delta A1_i$
$= A_i \vee \Delta A1_i$

$A1_{i+1} := (A_{i-1} \vee \Delta A_i) * P = A_{i-1} * P \vee \Delta A_i * P$
$= A1_i + \Delta A_i * P$

Ahora se define $AuxA_{i+1}$ y $AuxA1_{i+1}$ como:

(4) $AuxA_{i+1} = \Delta A1_i$

$AuxA1_{i+1} = \Delta A_i * P$

Por lo tanto se tiene

$$(5)\quad A_{i+1} = A_i \vee AuxA_{i+1} \quad ; \quad \Delta A_{i+1} \subseteq AuxA_{i+1}$$
$$A1_{i+1} = A1_i \vee AuxA1_{i+1} \quad ; \quad \Delta A1_{i+1} \subseteq AuxA1_{i+1}.$$

El conjunto de ecuaciones (5) nos permite calcular $AuxA_{i+1}$ y $AuxA1_{i+1}$ en forma iterativa, lo cual resulta más eficiente que evaluar la expresión (1). En resumen, de los pasos anteriores se tiene el siguiente esquema de iteraciones:

1: $AuxA_{i+1} := \Delta A1_i$

2: $AuxA1_{i+1} := \Delta A_i * P$

3: $\Delta A_{i+1} := AuxA_{i+1} - A_i$

4: $\Delta A1_{i+1} := AuxA1_{i+1} - A1_i$

5: $A_{i+1} := A_i \vee \Delta A_{i+1}$

6: $A1_{i+1} := A1_i \vee \Delta A1_{i+1}$

Cuando las relaciones ΔA_i y $\Delta A1_i$ están vacías, también $AuxA_{i+1}$, $AuxA1_{i+1}$, ΔA_{i+1} y $\Delta A1_{i+1}$ se vacían. De este modo $A_{i+1} = A_i$ y $A1_{i+1} = A1_i$, lo que significa que ya se alcanzó el punto fijo del esquema de iteraciones.

Cabe hacer notar que las ecuaciones (2) y (5) son independientes de la forma particular de la regla (1), de esta manera, (2) y (5) pueden considerarse como metaecuaciones que se establecen para cualquier sistema de reglas con operadores monótonos. Solamente la ecuación (4) depende de la forma de la regla pero esta puede derivarse fácilmente de reglas arbitrarias al sustituir (2) y expander la regla.

Se puede observar con facilidad los pares de asignaciones 1-2, 3-4 y 5-6 de tal forma que el algoritmo estructurado del esquema incremental de iteraciones es el siguiente:

1. Inicialización.

2. Mientras no se alcance el Punto fijo

   2.1 Generación de cambios.

   AuxA(i+1) := ΔA1(i)

AuxA1(i+1) := ΔA(i) * Progenitor

2.2 Almacenamiento de los cambios.

ΔA(i+1) := AuxA(i+1) - A(i)

ΔA1(i+1) := AuxA1(i+1) - A1(i)

2.3 Solución.

A(i+1) := A(i) V ΔA(i+1)

A1(i+1) := A1(i) V ΔA1(i+1)

3. Terminar.

Capítulo 5

IMPLANTACION DEL SISTEMA

En la primera sección de este capítulo se presentan las estructuras utilizadas para la representación interna de las reglas, así como las tablas necesarias para almacenar los objetos del sistema. Para este segundo prototipo se crearon algunas de las estructuras aunque la mayoría de estas fueron creadas desde el primer prototipo, siendo necesario modificar algunas de ellas.

En la sección 5.2 se explica el proceso de normalización de reglas no recursivas, mientras que los diferentes aspectos para el manejo de la recursividad se tratan en la sección 5.3. Posteriormente, en las secciones 5.4 y 5.5 se describe la generación del árbol de derivación y el proceso de evaluación de reglas recursivas.

5.1 Representación Interna del Sistema de Reglas

En esta sección se describen las tablas para almacenar los objetos del lenguaje, así como la representación interna de las reglas.

El sistema tiene, desde el punto de vista de la lógica, tres tipos de términos: símbolos de funciones, variables y símbolos de constantes, mientras que desde el punto de vista de las bases de datos deductivas cuenta con leyes de deducción y relaciones base. Para cada uno de estos elementos el sistema asocia una tabla.

En el sistema se identifican los objetos del lenguaje por medio de la asignación de identificadores, los cuales se forman por un código, el cual es un número entero positivo asignado de manera secuencial en la tabla a la que pertenece el objeto, esto es, distintos tipos de objetos pueden tener el mismo código pero se almacenan en tablas diferentes.

En este trabajo no se utilizan los símbolos de funciones, sin embargo la estructura fue considerada desde el primer prototipo para futuras ampliaciones al sistema. Los componentes de la tabla de funciones, como se muestra en la figura 5.1 son: el identificador asociado a la función (*fnsysname*), el nombre de la función (*fnusername*), el tipo (*fntype*) y el número de argumentos que contiene (*fpars*).

```
struct functions
{
 int fnsysname;              /* Código de la función */
 char fnusername[namesize];/* Nombre de la función */
 int fntype;                 /* Tipo de función      */
 int fpars;                  /* Número de argumentos */
}; /* Tabla de funciones */
```

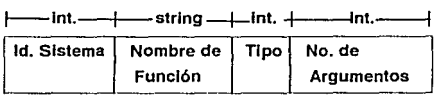


Figura 5.1: Representación Interna de las Funciones.

La representación interna de las variables consta de dos elementos: *varatrsysname*, que almacena el código asignado a la variable y *varatrusername*, que es un arreglo de caracteres donde se guarda el nombre de la variable:

```
struct varatr
{
 int varatrsysname;              /* Código de la
                                    variable      */
 char varatrusername[namesize]; /* Nombre de la
                                    variable o el
                                    atributo      */
}; /* Tabla de variables */
```

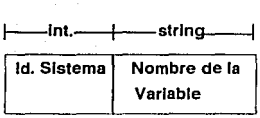
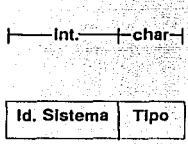


Figura 5.2: Representación Interna de la Tabla de Variables.

Para almacenar las constantes existe una tabla principal cuya estructura se muestra en la figura 5.3a. En esta tabla únicamente se almacena el código de la constante en *ctesysname* y el tipo de la constante en *ctype*. Debido a que el sistema manipula diferentes tipos de constantes, se requieren distintas longitudes para almacenarlas, por lo que se emplean tablas independientes para cada tipo. La estructura de cada tabla contiene el *código de la constante* y un campo de *valor*. El campo de *valor* en cada tabla tiene el tipo asociado a la tabla. En la figura 5.3b se muestra la tabla de constantes tipo caracter, en la figura 5.3c la tabla de constantes tipo entero y por último en la figura 5.3d la tabla de constantes tipo real.

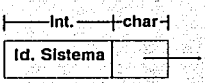
```
struct constants
{
  int ctesysname;          /* Código de la constante    */
  char ctetype;            /* Tipo de la constante      */
}; /* Tabla de constantes */
```



(a) Estructura de la Tabla General de Constantes.

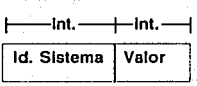
Figura 5.3: Tabla de Constantes.

```
struct ctestring
{
 int ctestrsysname; /* Código de la constante string */
 char *ctestrvalue; /* Valor */
}; /* Tabla de constantes tipo "string" */
```



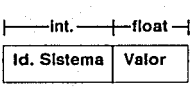
(b) Estructura de la Tabla para Constantes Tipo Caracter.

```
struct cteinteger
{
 int cteintsysname; /* Código de la constante entera */
 int cteintvalue;   /* Valor */
}; /* Tabla de constantes tipo entero */
```



(c) Estructura de la Tabla para Constantes Tipo Entero.

```
struct ctereal
{
 int cterealsysname; /* Código de la constante real */
 float cterealvalue; /* Valor */
}; /* Tabla de constantes tipo real */
```



(d) Estructura de la Tabla para Constantes Tipo Real.

Figura 5.3: Tabla de Constantes.

La tabla que almacena las reglas de deducción consta de seis componentes (figura 5.4): el código asignado a la regla (*rulesysid*), el nombre que el usuario asocia a la regla (*ruleusername*), el nombre del consecuente (*ruleconsname*), el apuntador a la estructura interna de la regla (*ruletree*), el número de argumentos del consecuente (*rpars*) y el apuntador a la estructura de ciclos de las reglas recursivas (*rulerecur*), el cual es nulo si el predicado del consecuente no pertenece a algún ciclo, en caso contrario apunta a la lista de ciclos.

```
struct rules
{
 int rulesysid;                /* Código de la regla      */
 char ruleusername[namesize];/* Nombre de la regla       */
 char ruleconsname[namesize];/* Nombre del consecuente */
 struct tree *ruletree;        /* Apuntador a la
                                  estructura de la regla */
 int rpars;                    /* Número de argumentos    */
 struct list *rulerecur;       /* Apuntador a la
                                  estructura de ciclos    */
}; /* Tabla de reglas */
```

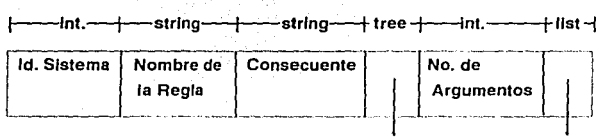


Figura 5.4: Representación Interna de las Reglas.

Como se describió en el capítulo 2 una regla consiste de un consecuente y un antecedente, para este trabajo el antecedente se representa en notación polaca como una lista de orden n, es decir, los operadores lógicos son de orden n, excepto el operador de negación el cual es de orden uno. Por lo anterior, la estructura empleada para representar reglas y submetas (átomos y operadores lógicos) consiste de tres elementos: un apuntador a un predicado, un operador lógico y un apuntador a la lista de operandos del operador lógico, como se muestra en la figura 5.5, donde *treemterm* apunta al multitérmino que representa un predicado, *treeop* es el código del operador lógico y *treelist* es el apuntador a la lista de operandos.

```
struct tree
{
 struct mterm *treemterm;  /* Apuntador a un predicado */
 int treeop;               /* Operador Lógico */
 struct listree *treelist; /* Operandos del operador
                              lógico */
}; /* Estructura de una regla */
```

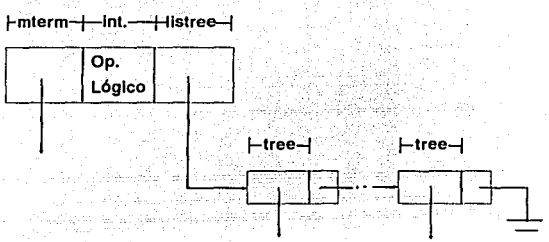


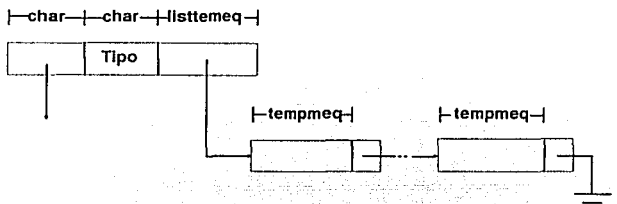
Figura 5.5: Estructura de una Regla.

La estructura que representa un multitérmino(1) consiste de un apuntador al nombre del multitérmino, el tipo y los argumentos del multitérmino (figura 5.6a). A través de *fsymb* se apunta al nombre, debido a que empleando la misma estructura de un multitérmino se representa un símbolo de función, un símbolo de predicado o un símbolo de constante; en *ftype* se almacena el tipo del multitérmino mientras que *args* apunta a los argumentos del multitérmino. Cada argumento se representa por medio de una multiecuación(2) especial denominada multiecuación temporal, cuya estructura se muestra en la figura 5.6b. La multiecuación temporal a través de S apunta a una lista de variables y M apunta a un multitérmino.

(1) Un **multitérmino** puede ser el vacío o tener la forma $f(E_1, ..., E_n)$ donde f es un símbolo de predicado o de función y cada E_i (i = 1, ..., n) está constituida por una pareja $<S_i,M_i>$, donde S_i es un conjunto de variables y M_i es un multitérmino.

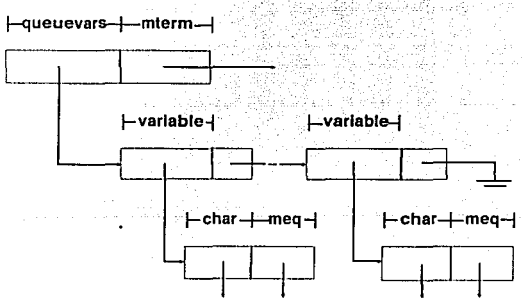
(2) Una **multiecuación** es la generalización de una ecuación, y tiene la forma $S = M$, donde S es un conjunto no vacío de variables y M es un multiconjunto de términos, los cuales no son variables. Un multiconjunto es una familia de elementos donde existe un ordenamiento entre ellos, pero es posible que se repitan elementos.

```
struct mterm
{
 char *fsymb;              /* Apuntador al nombre del
                              multitérmino            */
 char ftype;               /* Tipo de multitérmino    */
 struct listtemeq *args;   /* Lista de argumentos     */
}; /* Multitérmino */
```



(a) Estructura de los Multitérminos.

```
struct tempmeq
{
 struct queuevars *S;      /* Lista de variables */
 struct mterm *M;          /* Multitérmino       */
}; /* Multiecuación temporal */
```



(b) Representación Interna de las Multiecuaciones Temporales.

Figura 5.6: Representación Interna de los Multitérminos.

La estructura de ciclos (ver figura 5.7) está formada por todos los ciclos en los que participa un predicado. La lista que contiene la información de los diferentes ciclos consta de dos partes, *valuenode* que apunta a la información del ciclo y *next* que apunta al siguiente ciclo en el que participa dicho predicado. La estructura con la información del ciclo también está formada por dos partes: el número de reglas que participan en el ciclo (*tam*), que corresponde al tamaño del vector de ciclos y un apuntador al vector de ciclos (*vector*).

```
struct list
{
 char *valuenode;     /* Lista de nodos */
 struct list *next;   /* Siguiente nodo */
}; /* Lista general para el manejo de apuntadores */

struct ciclo
{
 int tam;      /* Tamaño del vector de ciclos */
 int *vector; /* Vector de ciclos             */
}; /* Estructura de ciclos recursivos */
```

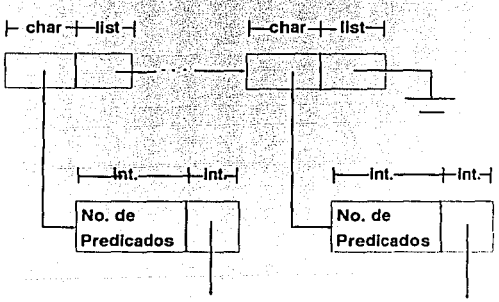


Figura 5.7: Estructura de Ciclos Recursivos.

Debido a que una relación R se puede representar como una regla cuyo antecedente está vacío ($R\leftarrow$), el esquema de las relaciones también se almacena en la tabla de reglas. Dichas relaciones se crean en la base de datos extensional, por lo que es necesario reconocer los atributos de cada relación. La estructura de la tabla de relaciones de la figura 5.8 contiene dos campos: el código de la relación (*rulesysid*) y un apuntador a la lista de atributos (*plsatr*).

```
struct atrofrel
{
 int rulesysid;        /* Código de la relación */
 struct list *plsatr; /* Lista de atributos    */
}; /* Tabla de relaciones */
```

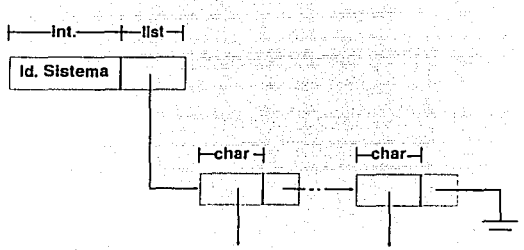


Figura 5.8: Tabla de Relaciones.

A diferencia de las variables, para los atributos de las relaciones es necesario definir el tipo, por lo que además de almacenarse en la tabla de variables es necesario almacenarlos en la tabla de atributos (figura 5.9). El código del atributo se almacena en *atrsysname*, el tipo del atributo ("string", caracter, real o entero) en *atrtype*, en *atrsize1* se almacena la longitud del atributo y en *atrsize2* se almacena una segunda longitud del atributo empleada para el tipo real.

```
struct atributtes
{
 int atrsysname;    /* Código del atributo    */
 int atrtype;       /* Tipo del atributo      */
 int atrsize1;      /* Tamaño 1 del atributo  */
 int atrsize2;      /* Tamaño 2 del atributo  */
}; /* Tabla de atributos */
```

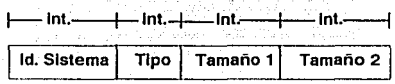


Figura 5.9: Tabla de Atributos.

Ejemplo: Considérese que se tiene la siguiente regla:

```
R1: Q(x,y) ← P1(x,f(z),z) AND P2(z,y,"A")
```

los nombres de los símbolos de predicados, variables, símbolos de funciones y símbolos de constantes que aparecen en la regla se almacenan en las tablas respectivas, la estructura que almacena la regla consiste de un apuntador al multitérmino que representa el consecuente *Q(x,y)*, el operador lógico AND y un apuntador a una lista de orden dos, en la cual el primer elemento de la lista representa al átomo *P1(x,f(z),z)* y el segundo elemento de la lista representa a *P2(x,y,"A")*. Como se observa en la figura 5.10 la estructura de multitérminos empleada permite almacenar de la misma forma los símbolos de predicados, los símbolos de funciones y los símbolos de constantes.

En la figura 5.11 se muestra la representación interna de un ciclo, en el que intervienen tres reglas recursivas en forma indirecta, el predicado *P1* depende del predicado *P5*, este depende del predicado *P3* y a su vez este último depende del primer predicado *P1*. Como se observa en la figura las tres reglas comparten el mismo vector de ciclos, de esta manera se evita la redundancia en la información.

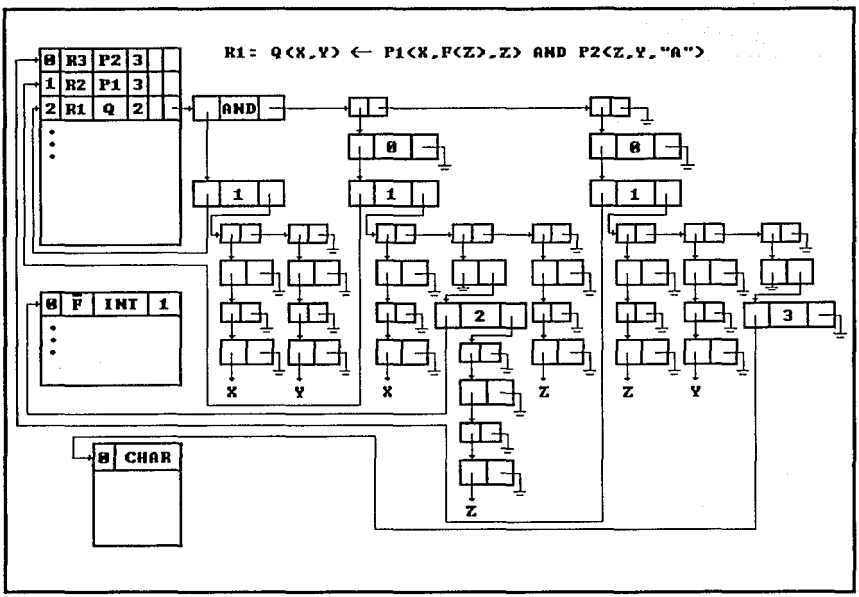


Figura 5.10: Representación Interna de la Regla R1.

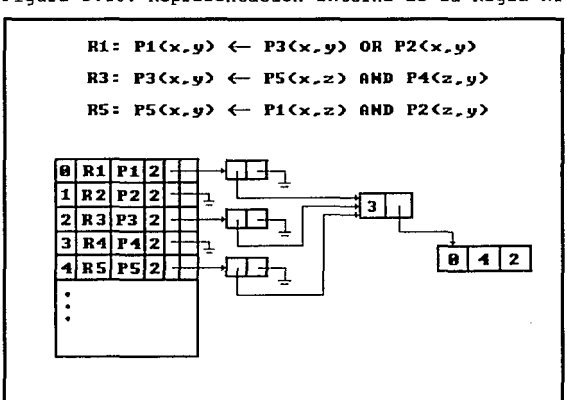


Figura 5.11: Representación Interna de un Ciclo.

5.2 Normalización de Reglas No Recursivas

El parser realiza un análisis léxico y sintáctico del programa lógico y genera las estructuras internas que representan el programa. Una de estas estructuras es la tabla de reglas. Como se mencionó en la sección 4.2, si existen dos o más reglas con el mismo predicado como consecuente, se dice que el sistema de reglas no está normalizado. Los predicados de los consecuentes que intervienen en el sistema, se almacenan una sola vez en la tabla de reglas; cualquier referencia a dichos predicados en el antecedente de una o varias reglas se hace a través de apuntadores. En el caso de que existan dos o más reglas con el mismo predicado como consecuente, las referencias que se hagan a estos predicados en los antecedentes de las reglas se harán a través de apuntadores a la primera regla en la tabla. Al normalizar el sistema de reglas, este solo contendrá reglas con predicados únicos como consecuente, por ejemplo, considérese el siguiente sistema de reglas no normalizado:

R_1: Q(x,y) ← P_1(x,y) V P_2(x,y)

R_2: P_1(x,y) ←

R_3: P_2(x,y) ← P_3(x,z) & P_2(z,y)

R_4: Q(u,v) ← P_3(u,w) & P_4(w,v)

R_5: P_3(x,y) ← P_1(x,y) V P_4(x,y)

R_6: P_4(x,y) ←

cuya representación interna simplificada se muestra en la figura 5.12. Como se puede observar, la estructura de la regla R_1 está formada por un operador OR y una lista de predicados P_1 y P_2, los cuales apuntan a la regla de donde son consecuentes dichos predicados. A su vez la regla R_4 está formada por un operador AND y una lista de predicados P_3 y P_4, a los que también se hace referencia a través de apuntadores. Aunque los predicados de los consecuentes de ambas reglas son iguales, ambos consecuentes son independientes. En caso de que existiera alguna regla que incluyera al predicado Q en el antecedente, el apuntador que hace referencia a dicho predicado estaría apuntando a la regla R_1 por ser la primera que aparece.

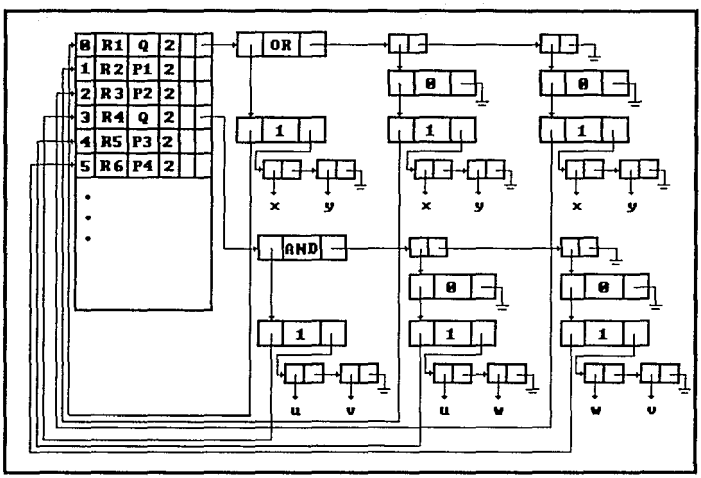


Figura 5.12: Representación Interna de una Regla No Normalizada.

Para realizar la normalización de las reglas es necesario llevar a cabo un proceso de ordenamiento sobre los predicados del consecuente de las reglas de tal manera que facilite la identificación de los predicados que sean iguales. Este proceso se lleva a cabo a través de un vector asociado a la tabla de reglas, el cual está formado por apuntadores a cada uno de los registros de la tabla (figura 5.13), debido a que resulta más eficiente reubicar los apuntadores que alterar la información en la tabla.

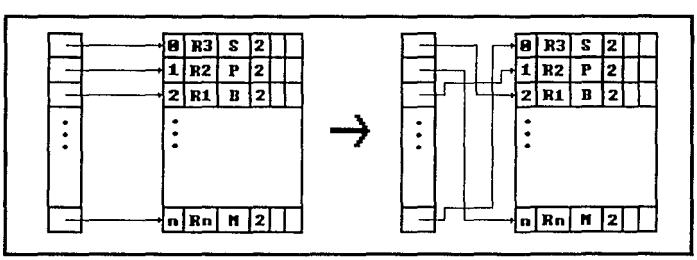


Figura 5.13: Vector de Ordenamiento Asociado a la Tabla de Reglas.

5.2.1 Algoritmo de Ordenamiento ("Heapsort")

El método de ordenamiento utilizado en este trabajo es el algoritmo de "*Heapsort*". Se eligió este método después de realizar un análisis de la complejidad de los principales métodos de ordenamiento (figura 5.14); durante este análisis se observó que los algoritmos basados en árboles de decisión son los más eficientes, destacando entre ellos el algoritmo de "*Heapsort*".

Algoritmo	Promedio	Peor Caso	Espacio Utilizado
Selection	$n^2 / 4$	$n^2 / 4$	Mismo de los Datos.
Bubble_sort	$n^2 / 4$	$n^2 / 2$	Mismo de los Datos.
Merge_sort	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	n espacios extras.
Quick_sort	$\Theta(n \log_2 n)$	$n^2 / 2$	$\log_2 n$ espacios extras.
Heap_sort	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	Mismo de los Datos.
Radix_sort	$\Theta (n + m)$	$\Theta (n + m)$	Espacio extra para ligas.
Address_cal_sort	$\Theta (n)$	$\Theta (n^2)$	Espacio extra para ligas.

n = número de elementos a ordenar.

m = número de dígitos del elemento a ordenar.

Figura 5.14: Tabla Comparativa de la Complejidad de los Principales Métodos de Ordenamiento.

El algoritmo de "Heapsort" utiliza una estructura de datos denominada "Heap" la cual es un árbol binario que debe satisfacer la siguiente condición:

$$K_j \leq K_i; \text{ para } 2 \leq j \leq n \; ; \; i = \text{trunc}\left(\frac{j}{2}\right)$$

donde K_i corresponde al i-ésimo elemento del arreglo a ordenar y K_j corresponde al j-ésimo elemento del arreglo a ordenar, n es el número de elementos del arreglo, i es el índice correspondiente al elemento padre y j el índice correspondiente al elemento hijo en caso de que exista.

El árbol binario se almacena secuencialmente en el arreglo, siguiendo el orden de arriba hacia abajo y de izquierda a derecha, de tal manera que los índices que corresponden a los hijos tanto izquierdo como derecho de un elemento i, son $2i$ y $2i+1$ respectivamente.

Cuando los elementos a ser ordenados ya se encuentran bajo una estructura "heap", entonces se procede a construir una lista ordenada en forma decreciente, eliminando repetitivamente la raíz (que siempre contiene al elemento más grande), llenando el arreglo de salida de su última posición a la primera y reordenando los elementos que quedan en el "heap" de tal forma que se reestablezca la propiedad del "heap" y así nuevamente se tenga al elemento más grande como raíz.

Algoritmo de ordenamiento: Heapsort

1. Sea *n* el número de elementos a ordenar.
   Sea *t* el número de elementos del heap.
   Sea *m* una variable temporal.
   Sea *L* el arreglo de elementos a ordenar.

2. Construir el heap inicial (Construye_heap).

3. Hacer t = n.

4. Mientras t ≥ 2

   4.1 Hacer m = L[1].

   4.2 Construye_heap.

   4.3 Hacer L[t] = m.

   4.4 Decrementar t en 1.

5. Terminar.

Algoritmo para la construcción del heap: Construye_heap

1. Sea *R* la raíz del heap.
   Sea *K* el elemento a insertar.
   Sea *V* la posición libre en el heap.
   Sea *HM* la posición del hijo con el mayor valor.
   Sea *L* el arreglo con los elementos a ordenar.

2. Hacer V = R.

3. Mientras V no sea una hoja del heap

   3.1 Hacer HM = hijo más grande de V.

   3.2 Si K < HM, entonces

      3.2.1 Hacer L[V] = L[HM].

      3.2.2 Hacer V = HM.

      en caso contrario

      3.2.3 Hacer V igual a una hoja del heap.

4. Hacer L[V] = K

5. Terminar.

5.2.2 Proceso de Normalización

El proceso de normalización se inicia con el ordenamiento de la tabla de reglas y posteriormente se verifica si existen reglas que tengan el mismo predicado como consecuente. En el caso de que el sistema no esté normalizado se genera una lista, la cual contiene los grupos de reglas a normalizar y a su vez, cada grupo tiene asociada una lista que contiene las reglas que le pertenecen, esto es, que tienen el mismo predicado como consecuente.

Habiéndose detectado el número de grupos de reglas a normalizar, se genera dinámicamente una nueva tabla que contendrá a las reglas originales, así como las reglas creadas a través de la normalización.

Los consecuentes de las nuevas reglas a generar, serán los correspondientes a cada uno de los predicados de los consecuentes de los grupos de reglas a normalizar, mientras que los antecedentes de dichas reglas estarán formados por la disyunción lógica de los predicados de los consecuentes de las reglas pertenecientes a cada grupo, los cuales se renombran de tal modo que queden identificados en forma única dentro del sistema. Los términos de los predicados de las nuevas reglas son independientes de los términos de los predicados que les dieron origen. Al igual que los predicados que conforman los antecedentes de las reglas, a estos términos se hace referencia por su dirección y no por su nombre.

Por último, todos los apuntadores del sistema serán reubicados en la nueva tabla a la posición correspondiente.

La figura 5.15 muestra el conjunto de reglas normalizadas Q' y Q'' que resultaron de renombrar las reglas

R_1: $Q(x,y) \leftarrow P_1(x,y) \vee P_2(x,y)$

R_4: $Q(u,v) \leftarrow P_3(u,w) \ \& \ P_4(w,v)$

presentadas en la figura 5.12. La regla generada al normalizar R_1 y R_4 tiene a Q como predicado de su consecuente y el antecedente está formado por los predicados Q' y Q'' que corresponden a las reglas renombradas R_1 y R_4 respectivamente. El identificador del sistema de la nueva regla corresponde al último número de identificador de la tabla de reglas incrementado en uno. Este número es negativo, lo cual permite diferenciar las reglas nuevas de las originales.

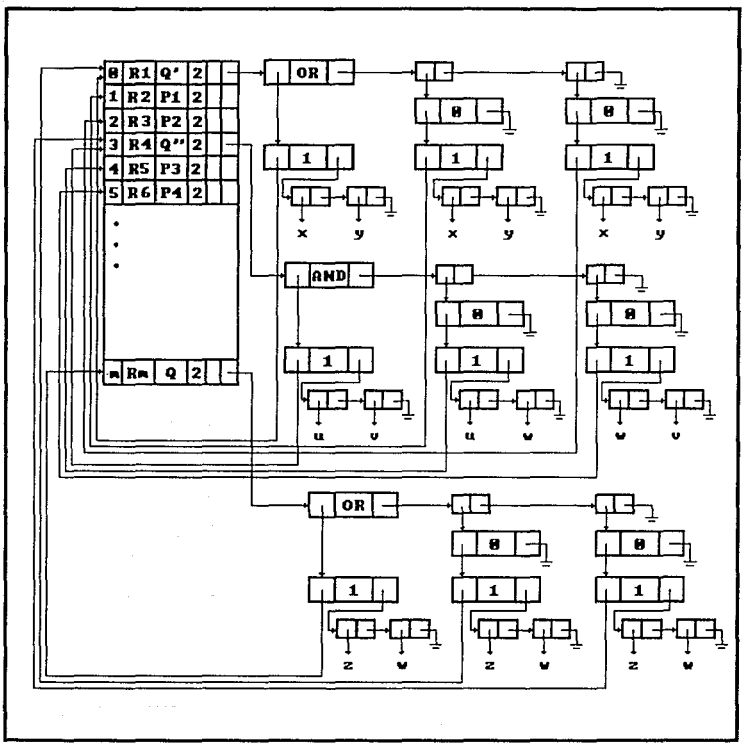


Figura 5.15: Representación Interna de una Regla Normalizada.

Algoritmo de Normalización

1. Sea *n* el número de reglas del sistema.
   Sea *m* el número de grupos de reglas a normalizar.
   Sea *N* la lista de reglas a normalizar.

2. Ordenar la tabla de reglas en forma alfabética ascendente de acuerdo al nombre del consecuente.

3. Generar la lista N con los grupos de reglas con consecuentes iguales.

4. Si N <> vacío

   4.1 Copiar la tabla de reglas a una nueva tabla de tamaño n + m.

   4.2 Mientras N <> fin

      4.2.1 Crear en la nueva tabla una regla con el consecuente igual al del grupo de reglas a normalizar.

      4.2.2 Formar el antecedente de la nueva regla con la disyunción de los consecuentes de las reglas a normalizar pertenecientes al grupo.

      4.2.3 Renombrar los consecuentes de las reglas normalizadas.

      4.2.4 Obtener el siguiente grupo de reglas de la lista N.

   4.3 Reubicar los apuntadores de la antigua tabla de reglas hacia la nueva.

5. Terminar.

5.3 Manejo de Reglas Recursivas

Una vez que el sistema de reglas se encuentra normalizado, es necesario detectar si existen reglas recursivas debido a que una regla recursiva requiere un método de evaluación diferente.

Como se mencionó en la sección 4.5 una regla puede ser recursiva en forma directa o indirecta. En el caso de tratarse de recursividad indirecta, es necesario determinar los nodos que participan en cada uno de los ciclos.

Para detectar si existen reglas recursivas, se genera la gráfica de dependencias correspondiente al sistema de reglas la cual se representa por medio de una matriz de adyacencias binaria, dicha matriz está constituida por bits debido a que se logra un ahorro notable en la utilización de la memoria. Para generar la matriz se analiza la dependencia entre los predicados de cada una de las reglas del sistema, obteniéndose la posición de cada predicado dentro de la matriz. Por ejemplo, si el predicado q depende del predicado p, entonces se coloca un "1" en la posición correspondiente a la columna asociada a q sobre el renglón asociado a p.

Para un sistema de n reglas se genera una matriz de tamaño $n \times n$, la cual se representa por una matriz de tamaño $n \times ((n/palabra) + r)$, donde *palabra* es el tamaño de la palabra del sistema que se esté utilizando, y $r = 1$ si el residuo de $n/palabra$ es diferente de cero, en caso contrario $r = 0$. Para este trabajo el tamaño de la palabra del sistema es de 16 bits.

Algoritmo de Normalización de Reglas Directamente Recursivas

1. Sea *R* la lista de las submetas directamente recursivas.
   Sea *n* el número de reglas del sistema.
   Sea *m* el número de elementos de la lista *R*.

2. Generar la matriz de adyacencias.

3. Si existen reglas directamente recursivas

   3.1 Simplificar la estructura de las reglas.

   3.2 Generar la lista R.

   3.3 Copiar la tabla de reglas a una nueva tabla de tamaño n + m.

   3.4 Mientras R <> fin

      3.4.1 Identificar la submeta que contenga el predicado directamente recursivo.

3.4.2 Generar una nueva regla cuyo antecedente esté formado por la submeta recursiva y el consecuente esté identificado en forma única.

3.4.3 Sustituir en la regla directamente recursiva, la submeta recursiva por el consecuente de la nueva regla.

3.4.4 Obtener el siguiente elemento de la lista R.

4. Terminar.

Con la finalidad de verificar si existe recursividad directa, es necesario generar una matriz de adyacencias previa a la definitiva, en el caso de que existan reglas directamente recursivas, se procede a simplificar la estructura del antecedente de dichas reglas.

La simplificación de la estructura de las reglas directamente recursivas consiste en eliminar los anidamientos innecesarios y en aplicar la ley de distributividad. La primera simplificación se realiza eliminando paréntesis que no se requieren en la estructura del antecedente, mientras que la aplicación de la distributividad solo se realiza distribuyendo el operador lógico AND sobre el OR.

Para llevar a cabo la normalización de reglas directamente recursivas se genera una lista de orden *m*, donde cada elemento de la lista dará origen a una nueva regla. Los *m* elementos de la lista están formados por un nodo que contiene dos apuntadores: el primero apunta a la submeta que contiene el predicado recursivo, la cual pertenece al antecedente de una regla directamente recursiva y el segundo apunta a esta regla recursiva. Por ejemplo, para la siguiente regla

```
R1: Ancestro(x,y) ← Progenitor(x,y) V
                    (Progenitor(x,z) &
                     Ancestro(z,y))
```

la submeta

```
(Progenitor(x,z) & Ancestro(z,y))
```

formaría parte de la lista de reglas directamente recursivas.

A partir de *m* se genera dinámicamente una nueva tabla que contendrá a las reglas originales, así como a las nuevas reglas creadas a partir de la normalización de las reglas directamente recursivas.

En la normalización de este tipo de reglas, se crea una regla para cada uno de los elementos de la lista de reglas directamente recursivas, cuyo antecedente lo constituye la submeta que contiene el predicado recursivo y al predicado del consecuente se le asigna un nombre único, mismo que reemplaza a la submeta recursiva en la regla correspondiente. Por ejemplo, dada la siguiente regla directamente recursiva:

```
R1: An(x,y) ← Pr(x,y) V (Pr(x,w) & An(w,y))
```

al realizar la normalización, la submeta recursiva *Pr(x,w) & An(w,y)* se sustituye por el predicado *A1(x,y)* y la submeta pasa a formar el antecedente de la nueva regla R_m que tiene como predicado del consecuente a *A1* como se muestra en la figura 5.16:

```
R1': An(x,y) ← Pr(x,y) V A1(x,y)
Rm : A1(u,v) ← Pr(u,w) & An(w,v)
```

Una vez que se ha finalizado la normalización es necesario generar la matriz de adyacencias definitiva debido a que el sistema de reglas original se modificó. A partir de esta nueva matriz de adyacencias se genera la matriz de caminos utilizando el algoritmo de Warshall, dicho algoritmo se describió en la sección 3.4 en su versión original, y se presenta a continuación en forma estructurada[3]:

Algoritmo de Warshall

1. Sea $A = \{a_{ij}\}$ la matriz de adyacencias.
   Sea $W = \{w_{ij}\}$ la matriz de caminos.
   Sea n el número de vértices de la gráfica de dependencias.

2. Hacer W = A.

3. Hacer j = 1.

4. Mientras j ≤ n

   4.1 Hacer i = 1.

   4.2 Mientras i ≤ n

      4.2.1 Si $w_{ij} = 1$, entonces

(3) El algoritmo fue estructurado por Cristóbal Juárez C.

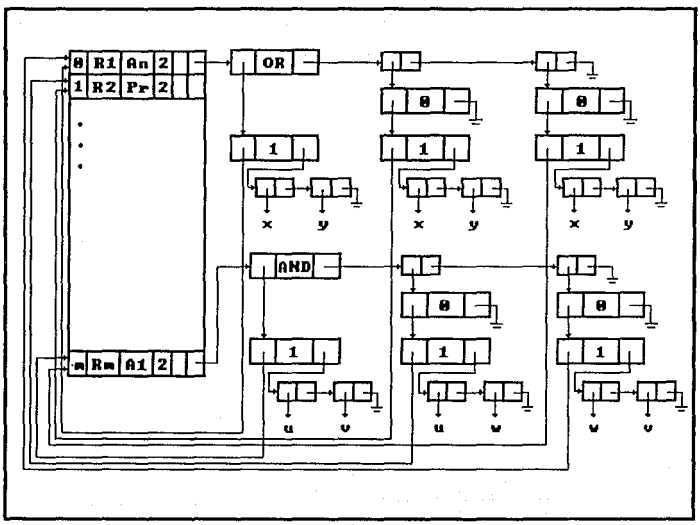


Figura 5.16: Representación Interna de una Regla Recursiva Normalizada.

4.2.1.1 Hacer k = 1.

4.2.1.2 Mientras k ≤ n

4.2.1.2.1 Hacer w_{ik} = (w_{ik} OR w_{jk}).

4.2.1.2.2 Incrementar k en 1.

4.2.2 Incrementar i en 1.

4.3 Incrementar j en 1.

5. Terminar.

Al igual que la matriz de adyacencias la matriz de caminos es binaria, del mismo orden y se representa por bits, lográndose así un gran ahorro de memoria.

Una vez que ya se tienen las matrices de adyacencias y de caminos, se realiza la detección de ciclos a través del algoritmo presentado en la sección 4.5.2. La estructura utilizada para almacenar los ciclos que se hayan detectado, está formada por un nodo que contiene el número de predicados que participan en el ciclo y un apuntador al vector de ciclos. Este vector es un arreglo de longitud variable generado en forma dinámica y contiene los identificadores de las reglas que participan en el ciclo. La estructura se accesa por cada una de las reglas que intervienen en un determinado ciclo a través de una lista propia de cada regla. Esta lista está formada por todos los ciclos en los que participa dicha regla.

Algoritmo para la Detección de la Recursividad

1. Normalizar las reglas directamente recursivas.
2. Generar la matriz de adyacencias.
3. Generar la matriz de caminos.
4. Detectar los ciclos.
5. Terminar.

5.4 Generación del Arbol de Derivación

En el proceso de generación del árbol de derivación se utiliza el método compilado de deducción.

Una consulta puede formarse por uno o más predicados conectados por los operadores lógicos AND, OR y NOT. La consulta se analiza por el parser transformándola a notación polaca y la representa en forma similar a la estructura de las reglas. Posteriormente el intérprete realiza la expansión del árbol analizando la estructura generada.

En la generación del árbol de derivación se emplean las mismas estructuras de datos utilizadas para representar a las reglas. El proceso de generación del árbol se inicia realizando una copia de la estructura de la regla a expander, esto tiene el propósito de no alterar la estructura original de la tabla de reglas. Para cada átomo de la regla se lleva a cabo un proceso de unificación con el consecuente de la regla correspondiente, en caso de que la unificación tenga éxito, se procede a realizar la sustitución de términos en la regla y finalmente, se mueve el apuntador que está dirigido a la tabla de reglas hacia la copia creada. De la misma forma se procede a expander cada uno de los átomos de la regla.

Durante la unificación del predicado a expander con el consecuente correspondiente, se puede tener éxito o fallar, en caso de falla se tienen que tomar ciertas acciones dependiendo de si el predicado es descendiente de un nodo AND, de un nodo OR o de un nodo NOT.

La propagación de las fallas se realiza de la siguiente manera:

- Si el nodo donde se produce la falla es descendiente de un nodo AND, esto significa que el nodo AND también falla.

- Si la falla ocurre en un nodo descendiente de un nodo OR entonces se elimina el nodo que falló, resultando alguno de los siguientes casos: si el nodo OR queda con un hijo, se elimina el nodo OR y el nodo restante se agrega al padre del nodo OR; si el nodo OR queda con un hijo y además el nodo OR es la raíz del árbol, entonces se elimina el nodo OR y el nodo restante queda como raíz del árbol; en caso de que el nodo OR quede con dos o más hijos, entonces no se realizan acciones adicionales.

- Si falla un nodo descendiente de un nodo NOT, entonces se interpreta que el nodo NOT evalúa a verdadero y por lo tanto se elimina el nodo NOT. Si el nodo NOT es descendiente de un nodo AND y el nodo AND queda con solo un hijo, entonces se elimina el nodo AND y el nodo restante se agrega al nodo padre del nodo AND.

Algoritmo para la Generación del Arbol de Derivación: Genera_árbol.

1. Sea *C* una lista de elementos que representan una meta o submeta.
   Sea *P* el primer predicado de la lista.
   Sea *R* la regla donde el consecuente podría unificar con *P*.

2. Mientras C <> fin

   2.1 Si P es un átomo, entonces

      2.1.1 Duplicar la regla R.

      2.1.2 Crear la estructura inicial para unificar.

      2.1.3 Unificar P con el consecuente de R.

      2.1.4 Si no falla, entonces

         2.1.4.1 Efectuar la sustitución de variables.

         2.1.4.2 Si P es recursivo, entonces

            2.1.4.2.1 Genera_árbol_recursivo.

            en caso contrario

            2.1.4.2.2 Expander el nodo.

            2.1.4.2.3 Genera_árbol.

            2.1.4.2.4 Eliminar P de la lista C.

         en caso contrario

         2.1.4.3 Propagar la falla.

      en caso contrario

      2.1.5 Genera_árbol.

      2.1.6 Eliminar P de la lista C.

3. Terminar.

Cuando el predicado a analizar pertenece a un ciclo, la generación del árbol de derivación se suspende para crear en forma independiente un árbol "recursivo" cuya raíz sea el nodo recursivo. En caso de que el predicado no sea recursivo, entonces se procede a expander el predicado en forma recursiva.

Es importante señalar que también es posible encontrar nodos recursivos durante la generación del árbol "recursivo"; si el nodo pertenece al ciclo que se está analizando, entonces se continúa con la expansión del árbol "recursivo", en caso contrario, se detiene la generación del árbol "recursivo" y se crea un nuevo árbol "recursivo" cuya raíz será el nodo recursivo encontrado.

Este proceso de generación de árboles "recursivos" se realizará tantas veces como ciclos independientes existan en el árbol de derivación y en los árboles "recursivos".

Durante la generación del árbol "recursivo" se verifica si el nodo a expander es igual a la raíz; si son iguales se obtiene el siguiente nodo a analizar, en caso contrario, se continúa la expansión del árbol "recursivo". Esto se hace para evitar que dicho árbol crezca en forma infinita.

Por ejemplo, dado el sistema de reglas:

```
R1: Padre(x,y)        ←
R2: Madre(x,y)        ←
R3: Progenitor(u,v)   ← Padre(u,v) V Madre(u,v)
R4: Ancestro(x,y)     ← Progenitor(x,y) V A1(x,y)
R5: A1(x,y)           ← Progenitor(x,z) &
                        Ancestro(z,y)
```

si se desea hacer una consulta sobre los ancestros de "Ana", el sistema genera un árbol de derivación (figura 5.17a) cuyo único nodo es la raíz del árbol *(Ancestro)*. Esto se debe a que la regla es recursiva por lo que la generación del árbol se detiene para crear el árbol "recursivo" correspondiente. Como se señaló anteriormente, el árbol "recursivo" generado (figura 5.17b) se simplifica eliminando anidamientos innecesarios y aplicando la distributividad del AND sobre el OR, de tal forma que el árbol resultante (figura 5.17c) queda con el

menor número de anidamientos posibles para ser evaluado. Después de simplificar el árbol "recursivo" se realiza un ordenamiento de las submetas que conforman a dicho árbol de tal manera que aparezcan primero las submetas que corresponden a las condiciones iniciales y posteriormente las submetas recursivas.

En la figura 5.17d se presenta la estructura de la lista de árboles "recursivos", que en este caso está formada por un solo elemento debido a que el árbol de derivación y el árbol "recursivo" no cuentan con ninguna otra regla recursiva que no esté incluida dentro del ciclo de *Ancestro*. El nodo de información de la lista indica que el árbol "recursivo" cuenta con dos submetas que constituyen las condiciones iniciales (Padre, Madre) y dos submetas que conforman la parte recursiva (A1: Ancestro y Padre, A1: Ancestro y Madre).

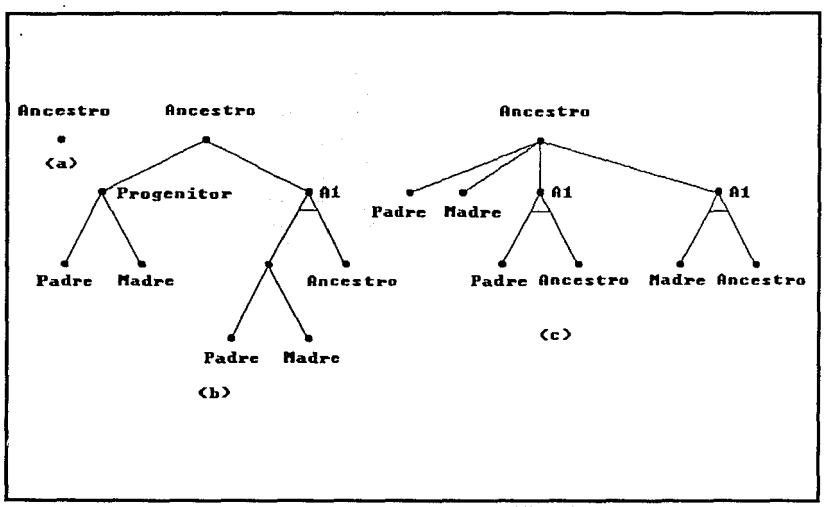


Figura 5.17: Generación del Arbol "Recursivo".
(a) Arbol de Derivación.
(b) Arbol de Derivación "Recursivo".
(c)Arbol de Derivación "Recursivo" Simplificado.

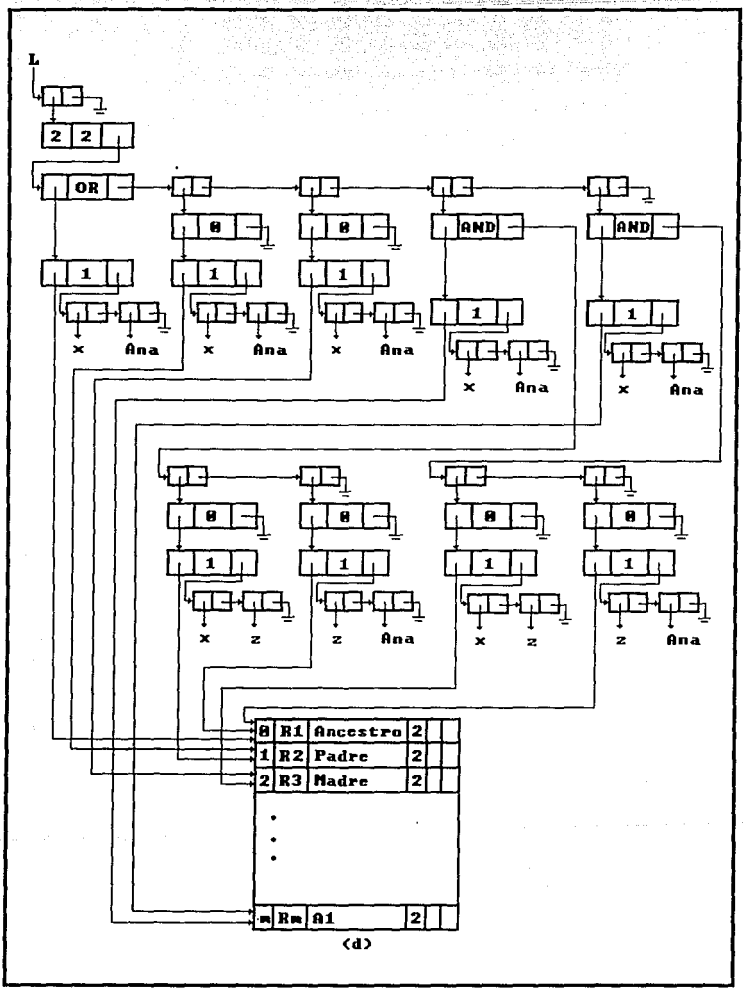


Figura 5.17: Generación del Arbol "Recursivo".
(d) Lista de Arboles "Recursivos".

Algoritmo para la generación del árbol recursivo:Genera_árbol_recursivo.

1. Sea *C* una lista de átomos y operadores que representan una meta o submeta.
   Sea *P* el primer predicado de la lista.
   Sea *R* la regla donde el consecuente podría unificar con *P*.
   Sea *S* la raíz del árbol "recursivo".

2. Mientras C <> fin

   2.1 Si P es un átomo, entonces

      2.1.1 Si P <> S, entonces

         2.1.1.1 Duplicar la regla R.

         2.1.1.2 Crear la estructura inicial para unificar.

         2.1.1.3 Unificar P con el consecuente de R.

         2.1.1.4 Si no falla, entonces

            2.1.1.4.1 Efectuar la sustitución de variables.

            2.1.1.4.2 Si P es recursivo y no pertenece al ciclo actual, entonces

               2.1.1.4.2.1 Generar un nuevo árbol "recursivo".

               en caso contrario

               2.1.1.4.2.2 Expander el nodo.

               2.1.1.4.2.3 Genera_árbol_recursivo.

               2.1.1.4.2.4 Eliminar P de la lista C.

            en caso contrario

            2.1.1.4.3 Propagar la falla.

en caso contrario

2.1.1.5 Eliminar P de la lista C.

en caso contrario

2.1.2 Genera_árbol_recursivo.

2.1.3 Eliminar P de la lista C.

3. Terminar.

Una vez que los árboles "recursivos" han sido generados, se simplifica su estructura eliminando anidamientos innecesarios y aplicando distributividad (AND sobre el OR), este es el mismo proceso que se utiliza para simplificar la estructura del antecedente de las reglas directamente recursivas.

Posterior a la simplificación, se procede a dividir cada uno de los árboles "recursivos" en su parte no recursiva (condiciones iniciales) y en su parte recursiva.

Estas operaciones se realizan con el objeto de preparar a los árboles "recursivos" para la aplicación del algoritmo de transformada delta. Además es necesario crear una relación en la base de datos, asociada a la raíz de cada árbol "recursivo", donde se almacenarán los resultados de la evaluación de estos árboles "recursivos".

Tanto para el árbol de derivación como para los árboles "recursivos", los nodos recursivos que dan origen a nuevos árboles "recursivos", son considerados relaciones base (temporales) cuya información será el resultado de la evaluación del árbol "recursivo" asociado a cada nodo. La evaluación de los árboles "recursivos" se realiza del último árbol generado al primero, de tal manera, que al evaluar cada uno de los árboles "recursivos" la información ya existe.

5.5 Evaluación de Reglas Recursivas

Para la evaluación de reglas recursivas es necesario crear algunas relaciones temporales, las cuales serán utilizadas por el algoritmo de transformada delta para almacenar los resultados intermedios de cada iteración. Con fines de eficiencia, las instrucciones que utiliza dicho algoritmo se generan dinámicamente y se almacenan en una estructura. Esto se realiza antes de iniciar su procesamiento con la finalidad de crear una sola vez dichas instrucciones debido a que el algoritmo es iterativo y por lo tanto las instrucciones son las mismas. Como se verá posteriormente en el algoritmo de evaluación de reglas recursivas, para cada árbol que aparece en la lista de árboles "recursivos" se aplica el algoritmo de transformada delta, el cual se describe a continuación.

Algoritmo de Transformada Delta[4]

1. Sea *Destino* la relación que contendrá el resultado de la evaluación del árbol "recursivo".
   Sean *Delta_D1 y Delta_Destino* las relaciones con los cambios reales de la evaluación.
   Sean *D1, Aux_D1, Aux_Destino y Temporal* relaciones auxiliares.
   Sean $ENR_1, ..., ENR_n$ los *n* elementos correspondientes a las relaciones base de la parte no recursiva del árbol "recursivo".
   Sean $ER_1, ..., ER_m$ los *m* elementos correspondientes a las relaciones base de la parte recursiva del árbol "recursivo".

2. Destino = Delta_Destino = $ENR_1 + ENR_2 + ... + ENR_n$.

3. Mientras (Delta_Destino <> vacío) o (Delta_D1 <> vacío)

   3.1 Aux_Destino = Delta_D1.

   3.2 Aux_D1 = (Delta_Destino * ER_1) + (Delta_Destino * ER_2) + ... + (Delta_Destino * ER_m).

   3.3 Delta_Destino = Aux_Destino - Destino.

   3.4 Temporal = Destino.

(4) El algoritmo original fue modificado con la finalidad de no crear relaciones intermedias que podían resultar muy grandes al momento de generar la información correspondiente.

3.5 Destino = Temporal + Delta_Destino.

3.6 Delta_D1 = Aux_D1 - D1.

3.7 Temporal = D1.

3.8 D1 = Temporal + Delta_D1.

4. Terminar.

La operación (*) utilizada en el algoritmo corresponde al "join" natural, el (-) corresponde a la operación "minus" y el (+) corresponde a la operación "union" en el lenguaje SQL.

Algoritmo de Evaluación de Reglas Recursivas

1. Sea *L* la lista de árboles "recursivos".
   Sea *R* la estructura con los nombres de las relaciones temporales.
   Sea *B* una estructura de "buffers" con las instrucciones del algoritmo de transformada delta.

2. Mientras L <> fin

   2.1 Asignar los nombres de las relaciones temporales a R.

   2.2 Crear las relaciones de la estructura R.

   2.3 Si no existe error, entonces

      2.3.1 Generar las instrucciones para el algoritmo de transformada delta y asignarlas a la estructura B.

      2.3.2 Si no existe error, entonces

         2.3.2.1 Aplicar Transformada Delta.

         2.3.2.2 Borrar las instrucciones de la estructura B.

      2.3.3 Borrar las relaciones de la estructura R.

   2.4 Obtener el siguiente elemento de la lista L.

3. Terminar.

Como se mencionó anteriormente, el proceso de evaluación se aplica a cada uno de los árboles "recursivos" y los resultados de dichas evaluaciones se almacenan en la relación correspondiente a cada árbol "recursivo", el resultado final se obtiene realizando una consulta sobre la raíz del árbol de derivación.

Al término de cada evaluación se borran de la base de datos todas las relaciones temporales utilizadas en el algoritmo de transformada delta.

Se ejemplificará la operación del algoritmo de transformada delta con el siguiente sistema de reglas:

```
R1: Progenitor(Fernando,Yadira)  ←
R2: Progenitor(M. Elena,Yadira)  ←
R3: Progenitor(Héctor,Eduardo)   ←
R4: Progenitor(Lilia,Eduardo)    ←
R5: Progenitor(Manuel,Fernando)  ←
R6: Progenitor(Narcisa,Fernando) ←
R7: Progenitor(Andrés,M. Elena)  ←
R8: Progenitor(María,M. Elena)   ←
R9: Ancestro(Ancestro,Descendiente) ←
    Progenitor(Ancestro,Descendiente) V
    A1(Ancestro,Descendiente)
R10: A1(Ancestro,Descendiente) ←
     Progenitor(Ancestro,W) &
     Ancestro(W,Descendiente)
```

en donde los hechos asociados a progenitor están almacenados en la relación base *Progenitor*. Por otro lado se asume lo siguiente:

1.- Las relaciones temporales requeridas ya han sido creadas, se encuentran inicialmente vacías y tienen la siguiente equivalencia de acuerdo al algoritmo de transformada delta:

- Destino = Ancestro
- D1 = A1
- Delta_Destino = Delta_Ancestro

- Delta_D1 = Delta_A1
- Aux_Destino = Aux_Ancestro
- Aux_D1 = Aux_A1

2.- Las instrucciones del algoritmo de transformada delta fueron previamente construidas en base a los nombres de las relaciones temporales mencionadas en el punto anterior.

La evaluación de la consulta de los ancestros de *Yadira*, la cual se representa como ← *Ancestro(x,Yadira)*, se inicia con la evaluación de las condiciones iniciales de la regla *Ancestro* (Paso2) y se muestra en la figura 5.18, en este caso la parte no recursiva del árbol solo está formada por la relación base *Progenitor*.

Progenitor	
Progenitor	Hijo
Fernando	Yadira
M. Elena	Yadira
Héctor	Eduardo
Lilia	Eduardo
Manuel	Fernando
Narcisa	Fernando
Andrés	M. Elena
María	M. Elena

Ancestro	
Ancestro	Descendiente
Fernando	Yadira
M. Elena	Yadira

Delta_Ancestro	
Ancestro	Descendiente
Fernando	Yadira
M. Elena	Yadira

Figura 5.18: Evaluación de Condiciones Iniciales del Algoritmo de Transformada Delta.

Debido a que la relación *Delta_Ancestro* no está vacía, se inicia la primera iteración del ciclo, de tal forma que en la figura 5.19 se presenta el estado de las relaciones según lo marca cada uno de los pasos del ciclo iterativo.

Aux_Ancestro	
Ancestro	Descendiente

(a) Transformada Delta: Paso 1
Aux_Ancestro = Delta_A1

Aux_A1	
Ancestro	Descendiente
Manuel	Yadira
Narcisa	Yadira
Andrés	Yadira
María	Yadira

(b) Transformada Delta: Paso 2
Aux_A1 = Delta_Ancestro * Progenitor

Delta_Ancestro	
Ancestro	Descendiente

(c) Transformada Delta: Paso 3
Delta_Ancestro = Aux_Ancestro - Ancestro

Temporal	
Ancestro	Descendiente
Fernando	Yadira
M. Elena	Yadira

(d) Transformada Delta: Paso 4
Temporal = Ancestro

Figura 5.19: Primera Iteración del Algoritmo de Transformada Delta.

Ancestro	Descendiente
Fernando	Yadira
M. Elena	Yadira

(e) Transformada Delta: Paso 5
Ancestro = Temporal +
Delta_Ancestro

Ancestro	Descendiente
Manuel	Yadira
Narcisa	Yadira
Andrés	Yadira
María	Yadira

(f) Transformada Delta:
Paso 6
Delta_A1 = Aux_A1 - A1

Ancestro	Descendiente

(g) Transformada Delta: Paso 7
Temporal = A1

Ancestro	Descendiente
Manuel	Yadira
Narcisa	Yadira
Andrés	Yadira
María	Yadira

(h) Transformada Delta:
Paso 8
A1 = Temporal +
Delta_A1

Figura 5.19: Primera Iteración del Algoritmo de Transformada Delta.

Durante la primera iteración, la relación *Delta_Ancestro* permite generar la nueva información de los progenitores de *Fernando* y *M. Elena* almacenando los cambios reales en la relación *Delta_A1*; a su vez esta relación genera en la siguiente iteración nueva información almacenándola en la relación *Delta_Ancestro*. El proceso iterativo continúa en forma similar hasta que se presenta el punto fijo, el cual ocurre cuando las relaciones *Delta_Ancestro* y *Delta_A1* se encuentran vacías, quedando el resultado de la consulta en la relación *Ancestro*(figura 5.20).

Ancestro	
Ancestro	**Descendiente**
Fernando	Yadira
M. Elena	Yadira
Manuel	Yadira
Narcisa	Yadira
Andrés	Yadira
María	Yadira

Figura 5.20: Estado Final de la Evaluación de la Regla Ancestro a través del Algoritmo de Transformada Delta.

Capítulo 6

EVALUACION DE RESULTADOS

En la primera sección de este último capítulo se presentan algunas de las ventajas de las bases de datos deductivas con respecto a las bases de datos relacionales.

En la sección 6.2 se presenta un resumen del estado en el que se encontraba el proyecto al inicio de este trabajo. Posteriormente en las secciones 6.3 y 6.4 se describen brevemente las características del software utilizado, así como las estrategias que se siguieron durante la implantación del sistema. En la sección 6.5 se describen los módulos que faltan por desarrollar para cubrir todos los aspectos planteados en el proyecto.

Por último, se presentan las conclusiones que se obtuvieron durante el desarrollo de este trabajo.

6.1 Bases de Datos Deductivas Versus Bases de Datos Relacionales

El esquema tradicional de las bases de datos ha permitido un gran desarrollo en la tecnología de la información, pero también resulta muy limitado en sus aplicaciones. El tipo de operaciones de las bases de datos relacionales se limita a la inserción, borrado, modificación y extracción de tuplos, siendo el "join" una de las operaciones más complejas que se pueden efectuar.

El DML cuenta con capacidades intrínsecas para accesar la base de datos eficientemente, pero el poder expresivo del DML es muy restringido. Es posible incrementar las capacidades de una aplicación combinando la eficiencia del DML para manipular grandes volúmenes de información, con el poder expresivo de un lenguaje anfitrión. Algunas aplicaciones donde es necesario

combinar estas capacidades son: diseño de bases de datos de CAD, bases de datos gráficas y bases de conocimiento para sistemas expertos. Estas aplicaciones se caracterizan por requerir un tiempo de respuesta mínimo y una constante modificación a la información, así como por la necesidad de incluir operaciones más poderosas.

Los sistemas de bases de datos deductivas dejan el esquema tradicional para proporcionar una nueva capacidad de inferencia y de facilidad en el manejo de la información. La fusión de los lenguajes de las bases de datos relacionales con el de la lógica de predicados, le da mayor expresividad al lenguaje utilizado en las bases de datos deductivas.

Muchos de los problemas a analizar requieren de procedimientos de resolución más complicados que los tratados en forma tradicional. Unos de estos procedimientos, considerados dentro de las bases de datos deductivas, son los basados en la recursividad. En general, el poder de la recursividad radica en la posibilidad de definir un conjunto infinito de objetos mediante una proposición finita. Como los procedimientos recursivos introducen la posibilidad de cálculos que no terminan, es necesario tener presente el problema de la terminación y el problema de la profundidad final de la recursividad. La terminación de un proceso recursivo, se maneja en este trabajo bajo el concepto de punto fijo. Por otro lado, la profundidad final no solo debe ser finita sino también pequeña, debido a que se requiere cierta cantidad de memoria para alojar el procedimiento recursivo.

Lo anterior se resuelve en este trabajo, escribiendo en la base de datos los resultados de cada iteración al aplicar el algoritmo de transformada delta. Aunque las operaciones de entrada/salida son más lentas que los accesos a memoria principal, esto no implica que los tiempos de respuesta sean mayores debido a que estos dependen de los algoritmos utilizados, en este caso, el algoritmo de transformada delta es muy eficiente y además permite manejar grandes volúmenes de información. El tiempo de respuesta puede ser mejorado si se utiliza memoria "cache".

6.2 Antecedentes del Proyecto

De acuerdo a la arquitectura general del sistema presentada en el capítulo 1, al inicio de este trabajo se contaba con un primer prototipo que cubría los siguientes aspectos:

La interfaz del usuario es muy primitiva debido a que se desarrolló únicamente con fines de prueba.

El lenguaje orientado a reglas se diseñó para la programación de sistemas expertos, con la característica de permitir tanto la generación de una base de datos relacional convencional, como la generación de un sistema deductivo. El lenguaje de programación que se utiliza consiste de cuatro tipos de objetos sintácticos: funciones, relaciones, reglas de deducción y restricciones de integridad, donde las funciones, las reglas de deducción y las restricciones de integridad son opcionales. Para las funciones y las restricciones de integridad, el prototipo cuenta con las estructuras necesarias para representarlas aunque no las manipula.

El parser además de realizar el análisis léxico y sintáctico del programa lógico de entrada, genera las estructuras para la representación interna de las reglas y las estructuras necesarias para la definición de la base de datos extensional.

El monitor de transacciones cuenta con dos submódulos: el monitor y la interfaz con el manejador de la base de datos relacional.

El subsistema de inferencia sólo cuenta con el intérprete, el cual se encarga de deducir conocimientos de hechos almacenados explícitamente en la base de datos. Los componentes que conforman al intérprete son: el generador del árbol de inferencia, el submódulo de unificación y el submódulo de evaluación para un sistema de reglas que se encuentre normalizado y que no sea recursivo.

Además el sistema cuenta con: una base de reglas, un manejador del catálogo, un manejador de base de datos relacional (RDBMS) y una base de datos.

6.3 Características del Software Utilizado

Los elementos de software requeridos para desarrollar la base de datos deductiva son: un manejador de bases de datos relacional y un lenguaje de propósito general. El primero no se desarrolló debido a que actualmente existen manejadores muy eficientes en cuanto a la manipulación de grandes volúmenes de información. En lo referente al lenguaje de programación se buscó un lenguaje que permitiera el manejo de memoria dinámica y que contara con estructuras de control para realizar una programación estructurada. Los criterios para la selección del software fueron: eficiencia, portabilidad y estandarización.

El lenguaje de programación seleccionado es C debido a que es un lenguaje intermedio entre alto y bajo nivel, cuenta con estructuras de control y además es portable.

El manejador relacional de bases de datos seleccionado debe corresponder a los estándares, ser portable, eficiente y apegarse al máximo al álgebra relacional. SQL es el estándar más aceptado para bases de datos relacionales. Se escogió ORACLE debido a que utiliza SQL y es portable. Además, se consideró que ORACLE cuenta con PRO*C, que le permite tener como lenguaje anfitrión a C.

6.4 Estrategias de Implantación

Las estructuras de datos utilizadas en los módulos de normalización de reglas y manejo de recursividad, desarrollados en este trabajo, son uniformes con las estructuras utilizadas en el primer prototipo de este sistema. Con fines de eficiencia en el manejo de memoria, las estructuras utilizadas se generan y liberan en forma dinámica.

La elaboración de este trabajo siguió una técnica de desarrollo incremental manteniendo la organización del sistema en cuatro módulos: lenguaje orientado a reglas, parser, monitor e intérprete. Con la finalidad de mantener esta modularidad se evitó el utilizar variables globales, de tal manera que la sustitución o modificación de cualquiera de los módulos, no repercute en el resto del sistema.

Con fines de portabilidad hacia otros sistemas, tanto en el aspecto de hardware como de software, en este trabajo se evitó el uso de funciones no estándares del lenguaje de programación.

Aunque el objetivo en este trabajo no es el optimizar los tiempos de respuesta del sistema, se buscó de implantar los algoritmos más eficientes en cuanto a tiempos de procesamiento y manejo de memoria.

6.5 Ampliaciones al Sistema

El objetivo de este trabajo es el desarrollar un prototipo que permita manipular reglas no normalizadas y reglas recursivas. Este segundo prototipo se encuentra aún limitado debido a que todavía faltan por desarrollar algunos módulos de acuerdo a la arquitectura general del sistema mostrada en el capítulo 1, dichos módulos son: módulo explicativo, mecanismo de actualización, optimización de reglas y el manejo de restricciones de integridad semántica.

El intérprete se encuentra incompleto debido a que aún no se implanta el manejo de funciones, lo cual se resolverá en trabajos posteriores. Actualmente la normalización de reglas y la detección de reglas recursivas se realiza cada vez que se carga el sistema. En versiones posteriores se pretende realizar la normalización y la detección de reglas recursivas dentro del parser, posterior al análisis léxico y sintáctico, de tal forma que al generar las estructuras internas del sistema estos dos aspectos ya se encuentren contemplados y puedan ser almacenados.

El sistema emplea tanto instrucciones del lenguaje de definición de datos como instrucciones del lenguaje de manipulación de datos, lo cual degrada el sistema. Esto es ineficiente dentro del sistema debido a que se generan vistas para representar nodos intermedios y la raíz del árbol de deducción; posteriormente se genera una expresión de consulta sobre la raíz, por lo que es necesaria la implantación de un método de evaluación que no genere vistas para los nodos intermedios.

Actualmente durante la evaluación de reglas recursivas, para cada consulta se generan los árboles "recursivos" asociados, las relaciones base que contendrán los resultados y las relaciones auxiliares para el proceso de evaluación a través de la transformada delta. En versiones futuras se pretende realizar estas operaciones una sola vez, de tal forma que cuando se realice una consulta que involucre reglas recursivas, todas las estructuras necesarias para su evaluación ya se encuentren generadas, logrando así una mayor eficiencia en el tiempo de procesamiento de una consulta.

En lo referente a la interfaz del usuario, se pretende desarrollar un módulo más amigable y poderoso que proporcione mayores facilidades para accesar al sistema.

Debido a que en una base de datos deductiva los hechos se encuentran almacenados en disco y los accesos a disco son más lentos que los accesos a memoria principal, se recomienda utilizar índices. Esto implica tanto la modificación del lenguaje deductivo como el desarrollo de un módulo para la creación y borrado de índices.

6.6 Conclusiones

Al reunir bajo un mismo concepto el formalismo de los esquemas lógicos con las características de las bases de datos relacionales, se obtiene un sistema de base de datos deductiva que permite manejar grandes volúmenes de información eficientemente, haciendo una representación del conocimiento a través de un esquema lógico.

La capacidad de evaluación de reglas recursivas, le da a una base de datos deductiva una mayor versatilidad en los procesos de deducción con respecto a las bases de datos convencionales, ya que el manejo de información no se limita a los hechos almacenados explícitamente en la base de datos, sino que además de expander el universo de información a través de reglas deductivas, se pueden utilizar procesos iterativos para deducir información de reglas que estén definidas en términos de si mismas.

Aunque faltan por implantar algunos módulos, este segundo prototipo constituye por si mismo una base de datos deductiva y conforma una buena base para el desarrollo de sistemas expertos debido a que cuenta con los elementos necesarios para formar una base de conocimientos, que desde el punto de vista de la inteligencia artificial, es la primera aproximación para el diseño de un sistema experto.

Finalmente se puede decir que este segundo prototipo, que constituirá un sistema de base de datos deductiva para la programación de sistemas expertos, se encuentra disponible para su utilización aún con las limitaciones descritas en este capítulo. Debido a que se utilizaron funciones estándares, este sistema es portable a cualquier equipo que soporte el lenguaje de programación C y el manejador de base de datos relacional ORACLE.

Apéndice A

DESCRIPCION EN BNF DE LA GRAMATICA DEL LENGUAJE

<DDB> ::= [<Functions>] <Relations> [<Rules>][<Constraints>]

<Functions> ::= **FUNCTIONS_DEF** <Functions_def>
<Functions_def> ::= <Function_def> | <Function_def> <Functions_def>
<Function_def> ::= [<fn_type>] <fn_declarator> <fn_body>
<fn_declarator> ::= <fn_name> ([<parameters>])
<parameters> ::= <identifier> | <identifier> , <parameters>
<fn_body> ::= <declars_list> {<fn_statements>}
<declars_list> ::= <declaration> | <declaration> ; <declars_list>
<fn_statements> ::= [<declars_list>] <statement_list>

<Relations> ::= **RELATIONS_DEF** <Relations_def>
<Relations_def> ::= <Relation_def> | <Relation_def> <Relations_def>
<Relation_def> ::= **RELATION** <Relation_name> (<Atributes>)
[**DOC** (<Comment>)]
<Relation_name> ::= <Identifier>
<Atributes> ::= <Atribute> | <Atribute> ; <Atributes>
<Atribute> ::= <Atr_name> : <Atr_type> [:<Atr_size1>]
[:<Atr_size2>] [**DOC** (<Comment>)]
<Atr_name> ::= <Identifier>
<Identifier> ::= <letter> | <letter> <String>
<Atr_type> ::= **STRING** | **CHAR** | **INTEGER** | **REAL** | **LONG** | **DOUBLE**
<Atr_size1> ::= <Number>
<Atr_size2> ::= <Number>

<String> ::= <letter> | <digit> | - | _ | <letter> <String> |
<digit> <String> | - <String> | <String>
<Comment> ::= <print_char> | <print_char> <Comment>
<letter> ::= a|b|...|y|z|A|B|...|Y|Z
<digit> ::= 0|1|2|...|8|9

<Rules> ::= **RULES_DEFINITION** <Rules_def>
<Rules_def> ::= <Rule_def> | <Rule_def> <Rules_def>
<Rule_def> ::= **RULE** <Rule_name> : <Consequent> ← <Antecedent>
[**DOC** (<Comment>)]
<Rule_name> ::= <String>

<Consequent> ::= <Predicate>
<Predicate> ::= <User_Pred> | <System_Pred>
<User_Pred> ::= <Predicate_name> (<Terms_list>)
<Predicate_name> ::= <Identifier>
<Terms_list> ::= <Term> | <Term> , <Terms_list>
<Term> ::= <Constant> | <Variable> | <Function>
<Variable> ::= <letter> | <letter> <String>
<Constant> ::= "<String>" | <Numeric_cte>
<Numeric_cte> ::= <digit> | <digit> <Numeric_cte>
<Function>::= <User_Function> | <System_Function>
<User_Function> ::= <Function_name> (<Params_list>)
<Function_name> ::= <Identifier>
<Params_list> ::= <Variable> | <Function>
<System_Function> ::= <plus> | <minus> | <multiplication> |
<division> | <module>
<plus> ::= **PLUS**(<Term> , <Term>) | <Term> + <Term>
<minus> ::= **MINUS**(<Term> , <Term>) | <Term> - <Term>
<multiplication> ::= **MUL**(<Term> , <Term>) | <Term> * <Term>
<division> ::= **DIV**(<Term> , <Term>) | <Term> / <Term>
<module> ::= **MOD**(<Term> , <Term>) | <Term> % <Term>
<System_Pred> ::= <less> | <less_equal> | <greather> |
<greather_equal> | <equal>
<less> ::= **LT**(<Term> , <Term>) | <Term> < <Term>
<less_equal> ::= **LEQ**(<Term> , <Term>) | <Term> ≤ <Term>
<greather> ::= **GT**(<Term> , <Term>) | <Term> > <Term>

<greather_equal> ::= **GEQ**(<Term>, <Term>) | <Term> ≥<Term>
<equal> ::= **EQ**(<Term> ,<Term>) | <Term> = <Term>

<Antecedent> ::= <Literals> | <Lit_Ant>
<Literals> ::= <Literal> | <Literal> <OR> <Literal> |
<Literal> <AND> <Literal>
<Lit_Ant> ::= <Ant> | <Literal> <OR> <Ant> | <Ant> <OR>
<Literal> | <Literal> <AND> <Ant> |
<Ant> <AND> <Literal>
<Ant> ::= <NOT> (<Antecedent>) | <Antecedent> | (<Antecedent>)
<Literal> ::= <Predicate> | <NOT> <Predicate> | (<Literal>)
<AND> ::= **AND**
<OR> ::= **OR**
<NOT> ::= **NOT**

<Constraints> ::= **CONSTRAINTS_DEF** <Const_definitions>
<Const_definitions> ::= <Const_definition> |
<Const_definition> <Const_definitions>
<Const_definition>::= **CONSTRAINT** <Const_name>
:[<Consequent>] ← <Antecedent>
[**DOC** (<Comment>)]
<Const_name> ::= <String>

Apéndice B

PALABRAS RESERVADAS DEL SISTEMA

B.1 Palabras Claves

FUNCTIONS_DEF
RELATIONS_DEF
RELATION
CONSTRAINTS_DEF
CONSTRAINT
RULES_DEF
RULE
STRING
CHAR
INTEGER
REAL
LONG
DOUBLE
DOC
←

B.2 Funciones Aritméticas

PLUS
MINUS
MUL
DIV
MOD

B.3 Predicados Aritméticos

<
≤
>
≥
=
LT
LEQ
GT
GEQ
EQ

B.4 Operadores Lógicos

AND
OR
NOT

Apéndice C

MENSAJES DE ERROR

C.1 Mensajes de Error del Parser

NAC-0001. Símbolo indefinido.
El nombre del identificador no fue declarado, o bien se encontró algún símbolo no aceptado en las palabras claves del lenguaje (Apéndice B). Este error se pudo generar al ocurrir un error previo en alguna definición.

NAC-0002. Se espera identificador de la función.
El identificador que se esperaba correspondía al nombre de alguna función. Pudo haber ocurrido un error previo en la definición de la función.

NAC-0003. La función ya había sido definida.
Dos funciones distintas tienen asignado el mismo nombre. Verificar los nombres de las funciones.

NAC-0004. Se espera paréntesis "(".
El compilador encontró un expresión sin un paréntesis que abre en el lugar adecuado. Verificar la sintaxis donde ocurrió el error.

NAC-0005. Se espera paréntesis ")".
El compilador encontró un expresión sin un paréntesis que cierra donde se esperaba. Posiblemente el error fue ocasionado por un error previo.

NAC-0006. Se espera el caracter "{".
El compilador encontró una expresión sin una llave que inicie un bloque en el lugar apropiado. Verificar la sintaxis de inicio del bloque donde ocurrió el error.

NAC-0007. Se espera el caracter "}".
El compilador encontró una expresión sin una llave que cierre un bloque en el lugar adecuado. El cuerpo de una función es delimitado por el símbolo de llave. Verificar que cada llave que abre tenga asociada una llave que cierre.

NAC-0008. Un programa debe incluir la definición de relaciones.
Es un error fatal debido a que un programa debe contener las definiciones de las reglas. Pudo ocurrir debido a que estas se encuentran fuera del orden correcto, o bien se pudo haber omitido la palabra clave RELATION_DEF, la cual indica inicio de la definición de relaciones.

NAC-0009. La definición de una relación debe iniciar con la palabra clave RELATION.
Dentro de la definición de relaciones se encontró posiblemente la declaración de una relación sin estar encabezada de la palabra reservada RELATION.

NAC-0010. La relación ya está definida.
Se encontraron dos relaciones con el mismo identificador. Verificar los nombres de las relaciones.

NAC-0011. Tipo de atributo inválido.
El tipo asociado a un atributo de una relación no es un tipo válido en el lenguaje. Los tipos válidos de los atributos son: STRING, CHAR, INTEGER, REAL, LONG o DOUBLE.

NAC-0012. Se espera el nombre del atributo.
En la definición de una relación no se encontró el identificador de un atributo. Posiblemente ocurrió debido a un error previo.

NAC-0013. Falta el caracter ":".
El compilador esperaba dentro de una expresión el separador dos puntos y encontró otro símbolo. Verificar la sintaxis de la expresión.

NAC-0014. Debe aparecer una constante numérica.
El compilador esperaba recibir un símbolo constante y encontró otro símbolo. Posiblemente fue ocasionado por un error previo.

NAC-0015. Falta el caracter ";".
El compilador encontró un expresión sin el separador punto y coma en el lugar adecuado. Verificar la sintaxis de la expresión.

NAC-0016. La definición de una regla comienza con la palabra clave RULE.
Dentro de la definición de reglas se encontró posiblemente la declaración de una regla sin estar precedida por la palabra clave RULE. Posiblemente es causa de algún error anterior.

NAC-0017. Debe aparecer el nombre de la regla.
El compilador esperaba un identificador como nombre de la regla y encontró algún otro símbolo.

NAC-0018. Debe aparecer el nombre del consecuente de la regla.
Se esperaba un identificador como nombre del consecuente de la regla. Posiblemente es ocasionado por un error previo.

NAC-0019. Se espera un identificador.
Se esperaba un identificador en la posición del error y se encontró otro símbolo. Un identificador debe comenzar con una letra, el símbolo - o el símbolo _.

NAC-0020. La función no se ha definido.
Se hace referencia a una función la cual no fue definida en el bloque de funciones. Verificar los nombres de las funciones definidas.

NAC-0021. Debe aparecer el operador lógico ←.
Se detectó fin en la definición del consecuente de la regla y no se encontró el símbolo lógico ←. Verificar la sintaxis de la cláusula, en la cual debe aparecer el consecuente, el operador lógico ← y el antecedente.

NAC-0022. La instrucción debería tener en esta posición un predicado del sistema.
El símbolo encontrado no corresponde al identificador de algún predicado del sistema sino que pertenece a otro símbolo. Verificar la sintaxis de la expresión.

NAC-0023. El número de paréntesis que abren es distinto a los paréntesis que cierran.
Al terminar la definición de relaciones, reglas o restricciones el compilador detectó la carencia de algunos paréntesis. Verificar que cada paréntesis que abre tenga asociado un paréntesis que cierre.

NAC-0024. Error en la sintaxis de la expresión.
El compilador detectó un error fatal y no es lógico el segmento de programa que encontró. Posiblemente un error previo no se recuperó, o bien existe alguna incoherencia, por ejemplo: dos operadores sin operandos que los separe, una expresión incorrecta, paréntesis desbalanceados, etc.

NAC-0025. Una restricción debe iniciar con la palabra clave CONSTRAINT.
El compilador no encontró la palabra reservada CONSTRAINT durante la definición de restricciones. Una restricción de integridad se declara encabezada con la palabra clave CONSTRAINT, el nombre con que el usuario identifica la regla, el consecuente, en caso de existir, el operador lógico ← y el antecedente.

NAC-0026. Se espera el nombre asociado a la restricción.
El compilador espera un identificador que corresponda al nombre de la restricción de integridad. Posiblemente fue causado por un error previo. Verificar la sintaxis para declarar las restricciones de integridad.

C.2 Mensajes de Error del Monitor

NAC-1000. La tabla de relaciones no se encuentra.
El intérprete no encontró el archivo que contiene la representación interna de las relaciones. El nombre del archivo donde se almacena la tabla de relaciones, es el nombre del programa con la extensión T05.

NAC-1001. La tabla de reglas no se encuentra.
El intérprete no encontró el archivo que contiene la representación interna de las reglas. El nombre del archivo donde se almacena la representación interna de las reglas, es el nombre del programa con la extensión T08.

NAC-1002. La tabla de predicados del sistema no se encuentra.
El intérprete no encontró el archivo que contiene los predicados del sistema. El sistema almacena los predicados del sistema en el archivo NACSYSPR.DAT. Verificar la instalación del sistema.

NAC-1003. El archivo con el nombre dado no se encuentra.
El sistema no puede abrir el archivo por que no aparece almacenado en el lugar donde es buscado. Verificar el nombre del archivo que indicó al sistema o si está posicionado en el lugar correcto.

NAC-1004. Durante el proceso de unificación ocurrió una falla.
Durante el proceso de efectuar el "merge" de dos multiecuaciones se encontró inconsistencia en el proceso deductivo. Posiblemente la consulta introducida al sistema no es correcta.

NAC-1005. Durante la unificación dos términos no fueron consistentes.
Durante el proceso de efectuar el "merge" de dos multiecuaciones se encontró que dos términos no son consistentes y por lo tanto no pueden ser unificados. Verificar la consulta.

NAC-1006. El proceso de compactación falló.
El sistema encontró una incongruencia al efectuar la transformación de compactación de dos términos durante el proceso deductivo. Posiblemente la consulta no es correcta.

NAC-1007. La evaluación de un predicado aritmético falló.
Las expresiones que aparecen como operandos de un predicado aritmético pueden no ser correctas. Puede ocurrir como consecuencia de algún error previo.

NAC-1008. El proceso de unificación falló.
No fue posible obtener el unificador más general para un par de términos durante el proceso deductivo. Verificar que los términos a unificar sean consistentes. También verificar la sintaxis de la consulta.

NAC-1009. La preparación de la consulta por medio de PRO*C falló.
No se puede preparar una consulta dinámica posiblemente por un error de sintaxis o por que no se han creado previamente las relaciones en la base de datos. Verificar la sintaxis de la consulta.

YEC-2001. Memoria insuficiente.
Al hacer un requerimiento de memoria al sistema, esta se encontraba agotada por lo que era insuficiente para poder llevar a cabo el proceso. Se requiere incrementar la memoria principal del sistema que se esté utilizando.

YEC-2002. Una cadena de caracteres no pudo ser convertida a entero.
Al realizar el proceso de conversión de una cadena de caracteres ("string") a entero, esta cadena incluía algunos caracteres que no eran dígitos por lo que estos caracteres no pudieron ser convertidos a enteros. Puede ocurrir como consecuencia de algún error previo.

Apéndice D

ARCHIVOS DEL SISTEMA

D.1 Programas Fuentes

- **CREATETR.C:** Submódulo del intérprete que contiene la generación del árbol de derivación, la generación de la lista de árboles "recursivos" y la evaluación de las reglas.

- **NACMONI.C:** Monitor del sistema que transfiere el control al módulo correspondiente de acuerdo a la operación a realizar.

- **NACORAC.C:** Submódulo que maneja la interfaz con el precompilador para C de ORACLE.

- **NACPARSE.C:** Submódulo con el parser que realiza el análisis léxico y sintáctico de los programas y de las consultas.

- **NACSQL.C:** Submódulo para transformar los árboles de derivación a expresiones equivalentes en SQL.

- **NACTAB.C:** Submódulo para cargar las tablas del lenguaje.

- **NACTREE.C:** Submódulo para la unificación y sustitución de reglas, además de incluir algunas funciones para la generación del árbol de derivación.

- **NORMGRAL.C:** Submódulo del intérprete el cual se encarga de la normalización del sistema de reglas así como la detección de reglas recursivas.

D.2 Archivos de Definición de Datos del Sistema

- **DDBFU.H:** Declaración de todas las funciones del sistema.
- **LISTFU.H:** Declaración de funciones para el módulo de listas.
- **MONIFU.H:** Declaración de funciones para el módulo del monitor.
- **NACANTEC.H:** Definiciones externas de las variables.
- **NACDEFTO.H:** Constantes del sistema.
- **NACLIST.H:** Variables y estructuras para el manejo de listas.
- **NACMONI.H:** Declaración de datos para el módulo del monitor.
- **NACPARSE.H:** Estructuras del parser.
- **NACQUERY.H:** Definición de datos para el módulo de consulta.
- **NACSE.H:** "Includes" del sistema sin referencias externas.
- **NACSEANT.H:** "Includes" del sistema con referencias externas.
- **NACTAB.H:** Declaración de datos para el módulo de tablas del sistema.
- **NACUNIF.H:** Estructuras para la unificación.
- **NACVARSP.H:** Variables globales del sistema.
- **NORCONST.H:** Definición de constantes para los módulos de normalización de reglas, detección de reglas recursivas y evaluación de reglas recursivas.

- **NORDECFU.H:** Declaración de funciones para los módulos de normalización de reglas, detección de reglas recursivas y evaluación de reglas recursivas.

- **NORDEF.H:** Definición de datos para los módulos de normalización de reglas, detección de reglas recursivas y evaluación de reglas recursivas.

- **NORSTRUC.H:** Definición de las estructuras para los módulos de normalización de reglas, detección de reglas recursivas y evaluación de reglas recursivas.

- **ORACFU.H:** Declaración de funciones para el módulo de interfaz de C con ORACLE.

- **PARSEFU.H:** Declaración de funciones para el módulo del parser.

- **QUERYFU.H:** Declaración de funciones para el módulo de consultas.

- **QUERYGEN.H:** Definición de constantes para el módulo de consulta.

- **SQLFU.H:** Declaración de funciones para el módulo de SQL.

- **TABFU.H:** Declaración de funciones para el módulo de tablas del sistema.

- **TREEFU.H:** Declaración de funciones para el módulo de generación del árbol de derivación.

- **UNIFYFU.H:** Declaración de funciones para el módulo de unificación.

D.3 Archivos de Definición de Datos del Sistema Operativo MS-DOS

- **STDIO.H:** Librerías estándares de entrada/salida de C.
- **STDLIB.H:** Librerías adicionales de C.
- **CTYPE.H:** Librerías adicionales para manipular caracteres.
- **STRING.H:** Definición de funciones para el manejo de cadenas.

D.4 Archivos de Definición de Datos Utilizados por el Precompilador PRO*C

- **SQLCA.H:** Definiciones para la manipulación de errores.
- **SQLDA.H:** Definición del descriptor para consultas dinámicas.

D.5 Archivos de Trabajo

- **NACAUTOM.DAT:** Tabla de transiciones del reconocedor del lenguaje.
- **NACSYSPR.DAT:** Predicados del sistema.
- **NACPREFI.TAB:** Representación interna de la consulta.
- **NACERROR.ERR:** Mensajes de error generados durante la compilación.
- **NACERROR.MSG:** Mensajes de error del parser.

- **NACFUNCT.C:** Texto fuente de las funciones.
- **NACTES.TAB:** Constantes que aparecen en la consulta.
- **NACVARS.TAB:** Variables que aparecen en la consulta.
- **NACMERR.MSG:** Mensajes de error del monitor.

D.6 Archivos para Almacenar la Versión Compilada

- **<nombre_programa>.T01:** Tabla de funciones.
- **<nombre_programa>.T02:** Tabla de reglas.
- **<nombre_programa>.T03:** Tabla de atributos.
- **<nombre_programa>.T04:** Tabla de restricciones de integridad.
- **<nombre_programa>.T05:** Tabla de relaciones.
- **<nombre_programa>.T06:** Tabla de variables.
- **<nombre_programa>.T07:** Tabla de constantes.
- **<nombre_programa>.T08:** Representación de las reglas en notación polaca.

Apéndice E

ESTRUCTURAS DE DATOS DEL SISTEMA

E.1 Tablas para Almacenar los Objetos del Lenguaje

```
struct atributtes
{
 int atrsysname;    /* Identificador del atributo          */
 int atrtype;       /* Tipo del atributo                   */
 int atrsize1;      /* Tamaño parte entera del atributo    */
 int atrsize2;      /* Tamaño parte decimal del atributo   */
}; /* Tabla de Atributos */

struct atrofrel
{
 int rulesysid;          /* Identificador de la relación */
 struct list *plstatr;   /* Lista de atributos de la
                            relación                      */
}; /* Tabla de Relaciones */

struct ciclo
{
 int tam;        /* No. de reglas en el vector de ciclos */
 int *vector;    /* Vector con los predicados del ciclo  */
}; /* Reglas que Conforman el Ciclo */

struct constants
{
 int ctesysname;        /* Identificador de la constante */
 char ctetype;          /* Tipo de constante             */
}; /* Tabla General de Constantes */
```

```
struct constraints
{
 int constsysname;              /* Identificador de la
                                   restricción          */
 char constusername[namesize]; /* Nombre de la
                                   restricción          */
 char constconsname[namesize]; /* Nombre del
                                   consecuente          */
 struct tree *constmterm;       /* Estructura de la
                                   restricción          */
 int cpars;                     /* Número de parámetros */
};/* Tabla de Restricciones de Integridad Semántica */

struct cteinteger
{
 int cteintsysname;    /* Identificador de la constante */
 int cteintvalue;      /* Valor entero de la constante  */
}; /* Tabla de Constantes tipo Entero */

struct ctereal
{
 int cterealsysname;   /* Identificador de la constante */
 float cterealvalue;   /* Valor real de la constante    */
}; /* Tabla de Constantes tipo Real */

struct ctestring
{
 int  ctestrsysname; /* Identificador de la constante   */
 char *ctestrvalue; /* Valor de la cadena de caracteres */
}; /* Tabla de Constantes tipo "string" */

struct functions
{
 int fnsysname;              /* Identificador de la
                                función              */
 char fnusername[namesize]; /* Nombre de la función */
 int fntype;                /* Tipo de la función   */
 int fpars;                 /* Número de parámetros */
}; /* Tabla de Funciones */

struct listree
{
 struct tree *valuetree; /* Apuntador al nodo hijo       */
 struct listree *next;   /* Apuntador al siguiente hijo */
}; /* Lista de Hijos de un Nodo del Arbol de Deducción */
```

```
struct rules
{
 int rulesysid;                /* Identificador de la
                                  regla                  */
 char ruleusername[namesize]; /* Nombre de la regla      */
 char ruleconsname[namesize]; /* Nombre del consecuente */
 struct tree *ruletree;        /* Estructura de la regla */
 int rpars;                    /* Número de parámetros   */
 struct list *rulerecur;       /* Estructura de ciclos   */
}; /* Tabla de Reglas */

struct syspr
{
 int syspredid;                 /* Identificador del
                                   predicado             */
 char syspredusername[namesize];/* Nombre del predicado */
 struct rules *syspredrule;     /* Estructura predicado */
}; /* Predicados del Sistema */

struct tree
{
 struct mterm *treemterm;  /* Apuntador al multitérmino */
 int treeop;               /* Tipo de Operación         */
 struct listree *treelist; /* Lista de predicados       */
}; /* Estructura para la Representación de una Regla */

struct varatr
{
 int varatrsysname;             /* Identificador de la
                                   variable              */
 char varatrusername[namesize];/* Nombre de la variable */
}; /* Tabla de Variables */
```

E.2 Estructuras de Datos para el Parser

```
struct aritvarpred
{
 struct rules *vpred;                    /* Regla    */
 struct varatr *vpvar;                   /* Variable */
}; /* Variables de un Predicado Aritmético */
```

```
struct automata
{
 char charstate;     /* Caracter del estado actual       */
 int  nextstate;     /* Número del estado de transición */
 struct automata *otherchar; /* Siguiente caracter del
                                estado                    */
}; /* Autómata */

struct lstprvar
{
 struct aritvarpred *valstpred; /* Valor del predicado
                                   aritmético             */
 struct lstprvar *next;         /* Siguiente variable     */
}; /* Lista de variables de un predicado aritmético */

struct stack
{
 char sysidtype;      /* Identificador del tipo de
                         elemento                         */
 int sysidnum;        /* Número de elementos              */
 struct list *sterm; /* Lista de términos del elemento */
}; /* Estructura del "Stack" */

struct terminfo
{
 char termid;         /* Identificador del tipo de
                         término                          */
 int termnum;         /* Número de identificación  */
 struct list *terms; /* Lista de términos          */
}; /* Información de un Término */

struct varperule
{
 int varuleini; /* Identificador de la primera variable */
 int varulefin; /* Identificador de la última variable  */
}; /* Variables que Aparecen en una Regla */
```

E.3 Estructuras de Datos para la Normalización de Reglas

```
struct lisnor
{
 struct listree *treelist; /* Estructura de una regla */
 struct lisnor *sigregla;  /* Siguiente estructura    */
}; /* Lista de Reglas a Normalizar */

struct lisvar
{
 char *variable;        /* Apuntador a la variable */
 struct lisvar *sigvar; /* Siguiente variable      */
}; /* Lista de Variables */
```

E.4 Estructuras de Datos para la Detección de Reglas Recursivas

```
struct inforec
{
 struct rules *regla;/* Apuntador a la regla              */
 struct tree *arbol; /* Apuntador al subárbol recursivo */
}; /* Arbol que Contiene el Término Recursivo */

struct recur
{
 struct inforec *info; /* Información de la regla
                          recursiva                      */
 struct recur *next;   /* Siguiente regla recursiva */
}; /* Lista de Reglas Directamente Recursivas */
```

E.5 Estructuras de Datos para la Unificación

```
struct listmeq
{
 struct meq *valmeq;     /* Apuntador a la multiecuación */
 struct listmeq *next;   /* Siguiente multiecuación      */
}; /* Lista de Multiecuaciones */

struct listtemeq
{
 struct tempmeq *valtemeq; /* Apuntador a la multiecuación
                              temporal                     */
 struct listtemeq *next;   /* Siguiente multiecuación
                              temporal                     */
}; /* Lista de Multiecuaciones Temporales */

struct listvars
{
 struct variable *valvars; /* Apuntador a la variable */
 struct listvars *next;    /* Siguiente variable      */
}; /* Lista de Variables */

struct meq
{
 int counter;          /* Contador                      */
 int varnumber;        /* Número de variables del lado
                          izquierdo                     */
 struct listvars *S;   /* Lista de variables            */
 struct mterm *M;      /* Multitérmino                  */
}; /* Multiecuación */

struct mterm
{
 char *fsymb;              /* Nombre del multitérmino     */
 char ftype;               /* Tipo de multitérmino        */
 struct listtemeq *args;   /* Argumentos del multitérmino */
}; /* Multitérmino */

struct queuevars
{
 struct variable *valqvar; /* Valor              */
 struct queuevars *next;   /* Siguiente variable */
}; /* Cola de Variables */
```

```
struct system
{
 struct listmeq *T;     /* Parte T */
 struct upart *U;       /* Parte U */
}; /* Sistema R para el Proceso de Unificación */

struct tempmeq
{
 struct queuevars *S;  /* Lista de variables */
 struct mterm *M;      /* Multitérmino       */
}; /* Multiecuación Temporal */

struct upart
{
 int meqnumber;                    /* Número de
                                      multiecuaciones     */
 struct listmeq *zerocountermeq; /* Multiecuaciones con
                                      contador cero       */
 struct listmeq *equations;      /* Multiecuaciones con
                                      contador distinto de
                                      cero                */
}; /* Parte U */

struct variable
{
 char *namevar;      /* Nombre de la variable          */
 struct meq *M;      /* Multiecuación donde aparece la
                        variable                        */
}; /* Variables */

struct vars_in_mterm
{
 char *vars_tvaratr;             /* Valor               */
 struct variable *vars_variable; /* Estructura de la
                                    variable            */
 struct vars_in_mterm *next;     /* Siguiente variable */
}; /* Variables que Aparecen en el Multitérmino */
```

E.6 Estructuras de Datos para la Sustitución

```
struct varpred
{
 struct listvars *P;      /* Variables del predicado    */
 struct listvars *Q;      /* Variables del consecuente */
}; /* Lista de Variables para la Sustitución */
```

E.7 Estructuras de Datos para la Evaluación del Arbol de Deducción

```
struct buftd
{
 char *asigna;        /* Operación de asignación          */
 char *join;          /* Operación de "join"              */
 char *dif1;          /* Primera operación de diferencia */
 char *asigna1;       /* Primera operación de asignación */
 char *union1;        /* Primera operación de unión       */
 char *dif2;          /* Segunda operación de diferencia */
 char *asigna2;       /* Segunda operación de asignación */
 char *union2;        /* Segunda operación de unión       */
}; /* Estructura de Buffers del Algoritmo de Transformada
      Delta */

struct info_lsrec
{
 int numterest;          /* Número de términos estáticos */
 int numterdin;          /* Número de términos dinámicos */
 struct tree *arbolrec; /* Arbol "recursivo"             */
}; /* Información del Arbol "Recursivo" */

struct listviews
{
 char *pnameview;          /* Nombre de la vista */
 struct listviews *next;  /* Siguiente vista    */
}; /* Lista de Vistas Creadas en la Base de Datos */
```

```
struct reltd
{
 char *dl;              /* Nombre asociado a dl              */
 char *delta_dl;        /* Nombre asociado a delta_dl        */
 char *delta_destino;/* Nombre asociado a delta_destino */
 char *aux_destino;   /* Nombre asociado a aux_destino    */
 char *aux_dl;          /* Nombre asociado a aux_dl          */
 char *t_temporal;    /* Nombre asociado a t_temporal     */
}; /* Estructura de Relaciones Temporales de Transformada
      Delta */
```

Bibliografía

[Alag86] Alagic, Suad.
Relational Database Technology.
U.S.A.: Springer-Verlag, 1986. 259 pp.

[Baas88] Baase, Sara.
Computer Algorithms Introduction to Design.
U.S.A.: Addisson-Wesley, 2nd ed., 1988.

[Baye85] Bayer, R.
Database Technology for Expert Systems.
Proc. GI-Kongress Wissensbasierte Systeme 1985.
Springer Informatik Fachberichte 112, 1985.

[CeGT90] Ceri, S., Gottlob, G. & Tanca, L.
Surveys in Computer Science. Logic Programming and Databases.
Germany: Springer-Verlag, 1990.

[ChLe73] Chang, Chin-Liang & Lee, Richard Char-Tung.
Symbolic Logic and Mechanical Theorem Proving.
U.S.A.: Academic Press,1973. 331 pp.

[DöMü72] Dörfler, W. & Mühlbacher, J.
Graphentheorie Für Informatiker.
Germany: Walter de Gruyter, 1972.

[GaMi78] Gallaire, H., Minker, J.
Logic and Data Bases.
U.S.A.: Plenum Press, 1978.

[GaMN84] Gallaire, H., Minker, J. & Nicolas, J.-M.
Logic and Databases: A Deductive Approach.
ACM Computing Surveys, Vol. 16, No. 2
(June 1984) pp. 153-185

[Hara69] Harary, Frank.
Graph Theory.
U.S.A.: Addison-Wesley, 1969. 274 pp.

[KoSi91] Korth, Henry F. & Silberschatz, Abraham.
Database System Concepts.
Singapore: McGraw-Hill International, 2nd ed., 1991.

[Lloy84] Lloyd, J.W.
Fundations of Logic Programming.
U.S.A.: Springer-Verlag, 1984.

[TrSo84] Tremblay, J.-P., Sorenson, P.G.
An Introduction to Data Structures with Applications.
Singapore: McGraw-Hill International, 2nd ed., 1984.

[Ullm88] Ullman, Jeffrey D.
Principles of Database Knowledge-Base Systems Computer.
U.S.A.: Science Press, Inc., 1988.

[Wirt76] Wirth, Niklaus.
Algorithms + Data Structures = Programs.
U.S.A.: Prentice Hall, Inc., 1976.

Referencias

[Aran91] Aranda, Norma A.
Implantación de un Lenguaje Orientado a Reglas para la Programación de Sistemas Expertos.
Tesis de Maestría en Ciencias de la Computación;
México: IIMAS-UNAM, 1991. 116 pp.

[Baye85] Bayer, R.
Database Technology for Expert Systems.
Proc. GI-Kongress Wissensbasierte Systeme 1985.
Springer Informatik Fachberichte 112, 1985.

[DöMü72] Dörfler, W. & Mühlbacher, J.
Graphentheorie Für Informatiker.
Germany: Walter de Gruyter, 1972.

[FrSS90] Freitag, B., Schütz, H. & Specht, G.
LOLA-A Logic Language for Deductive Databases and its Implementation.
Technische Universitäc München, TUM-I9043, (Nov. 1990).

[GaMN84] Gallaire, H., Minker, J. & Nicolas, J.-M.
Logic and Databases: A Deductive Approach.
ACM Computing Surveys, Vol. 16, No. 2,
(June 1984). pp. 153-185.

[JuAr90] Juárez, C. & Aranda, N.A.
A Deductive Database System for Expert Systems Programming.
México: Comunicaciones Técnicas IIMAS-UNAM.
(Serie Naranja No. 558), 1990. 16 pp.

[Lloy84] Lloyd, J.W.
Fundations of Logic Programming.
U.S.A.: Springer-Verlag, 1984.

[Mins68] Minsky, M.L.
Semantic Information Processing.
U.S.A.: MIT Press, 1968.

[NiYa78] Nicolas, J.-M. & Yazdanian, K.
Integrity Checking in Deductive Data Bases.
In [GaMi78], U.S.A.: Plenum Press, 1978.