CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
SECCIÓN DE COMPUTACIÓN

BASES DE DATOS DEDUCTIVAS CON EXTENSION A CONJUNTOS FINITOS

TESIS QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS
EN LA ESPECIALIDAD DE
INGENIERIA ELECTRICA

PRESENTA:

LIC. FERNANDO ZACARÍAS FLORES

DIRECTOR DE TESIS: **DR. SERGIO V. CHAPA VERGARA**

MARZO / 98

XM

RESUMEN

En el presente trabajo de tesis presentamos una propuesta que mejora la manera en que lenguajes como PROLOG manejan las Bases de Datos Deductivas y, concretamente el manejo de nuevas estructuras como lo son los conjuntos. Para esto, seleccionamos solo aquellas partes de nuestro interés del lenguaje SuRE [JOM95], [OJ96], esto por considerar a SuRE como un lenguaje robusto del cual partir. El hecho de elegir conjuntos como nuestro tema de interés, se debe a que este tipo de estructuras de datos permiten que el usuario pueda manipular de manera clara, transparente y sencilla muchos de los problemas que con estas estructuras se resuelven de manera directa. Además, otro punto relevante tratado en nuestro trabajo es, que se incorpora el uso de matching para aquellos casos en los que es suficiente con este algoritmo y no se requiere de la unificación. Esto es importante mencionar debido a que el algoritmo de unificación es más complejo, aunque no se deja de utilizar para los casos en que sí se necesita.

El trabajo medular de la presente se desarrollo a través de ejemplos, con la finalidad de hacer mas clara la ejemplificación de nuestros conceptos e ideas. En nuestra investigación también se proponen algunas variantes de como atacar los problemas de la semántica de los lenguajes orientados a el manejo de las Bases de Datos Deductivas, ya que esta es la base para poder mejorar en mucho la manipulación que de éstas se hace. También, mejoramos la forma en que el usuario plantea su solución del problema y, no solo eso, sino que además mejoramos el procesamiento de ejecución, debido a como resolvemos los diferentes problemas. Proponemos una extensión muy natural que consiste en generalizar aseveraciones de subconjuntos a aseveraciones parciales y trabajar en dominios reticulados mas generales. Estas ayudan a dar claridad y hacer formulaciones concisas a los problemas, involucrando operaciones de agregación y recursión en las consultas a Bases de Datos Deductivas.

Palabras Clave: Bases de Datos Deductivas, Programación lógica, LDL, CORAL, SuRE, conjuntos, agregación, matching, unificación, dominios reticulados.

Dedicatoria especial

A la memoria de mi madre

Filiberta Flores Rojas de Zacarías

Por los años de felicidad que me diste y por el cariño que siempre recibí de ti, TE QUIERO!.

INDICE GENERAL

INTRODUCCIÒN

Hoy día, las necesidades de contar con lenguajes más expresivos y poderosos como herramientas para el programador, ha motivado el desarrollo de diversos lenguajes. El objetivo principal es, proporcionar una mayor facilidad en el modelado y desarrollo de algoritmos de forma eficiente y rápida para el programador. Si a esto agregamos la necesidad de manejar grandes volúmenes de datos y un alto grado de interactividad sistema-usuario el problema se complica.

Actualmente, las **B**ases de **D**atos **D**eductivas (BDD) han tenido un fuerte desarrollo debido a las nuevas tendencias con respecto al diseño e implantación de estas, las cuales descansan en el modelo lógico que muestra ventajas sobre el modelo relacional de datos. Dentro de las BDD que han surgido destacan: CORAL, LOGRES, LDL, DECLARE, ECRC, COL, etc. [RU93].

Como se mencionó anteriormente las BDD se han seguido estudiando en los últimos años, principalmente con el uso del lenguaje de la lógica como la herramienta del modelado de datos. Es esta la razón que ha dado fuerza a los sistemas de bases de datos como CORAL, LDL1, etc. En el presente trabajo se presentan las investigaciones y análisis realizados referentes a las BDD. Las nuevas aportaciones de la logica aquí expuestas proporcionan a los sistemas de BDD un mejor manejo de los datos así como también incrementan su eficiencia y expresividad en el diseño de programas. El uso de la logica es mostrado utilizando ejemplos que nos permiten analizar las diferentes alternativas de solución a los diversos problemas con que se enfrenta el desarrollo de sistemas de BDD. Basados en el análisis e investigación de los mismos enfrentamos el paradigma de "extensión a conjuntos" en las BDD, que representa un avance de suma importancia debido a que al incluir estos en la programación nos permite el manejo de funciones estratificadas, dando a lenguajes como CORAL y LDL un mayor alcance, mejorando el desarrollo de algoritmos por parte del "usuario" (programador). Para esto se necesita de una semántica declarativa y una operacional, las cuales nos indicarían si nuestro programa es correcto y si se pueden obtener respuestas satisfactorias.

Un sistema de BDD que cuenta con el manejo de conjuntos de manera directa, sin complicaciones de expresividad nos permite explotar de una mejor manera los datos que conforman nuestras bases de datos, ya que uno de los principales problemas en el manejo de grandes volúmenes de datos es el acceso a éstos. Si nuestro sistema de BDD cuenta con la extensión a conjuntos, esto facilita y reduce el tiempo de acceso a los datos de manera significativa. Permitiéndonos incluso, el pensar en un procesamiento paralelo de nuestras aplicaciones, lo que extiende el poder y alcance de cualquier lenguaje. Este es un punto relevante surgido de nuestras investigaciones. Nuestra investigación y análisis parte del estudio de los sistemas existentes a la fecha, resaltando y proponiendo soluciones a los diversos problemas que se encuentran inmersos en algunos de los sistemas analizados. Todo esto, sin descuidar que nuestras propuestas nos hagan perder expresividad y funcionalidad en la resolución de los problemas.

Entre los sistemas relacionados con este trabajo tenemos :

LDL [TZ86], CORAL [RPDS88], SuRE [O95] y Col [KdMS90]. Cada uno de ellos muestran ventajas y desventajas de lo que proponemos hacer, en base a una breve discusion de cada uno de los sistemas, llegamos a la justificacion del uso de SuRE (este nombre se debe a los conceptos que en el se introducen, **Su**bconjuntos, **R**elaciones y **E**cuaciones) como la mejor alternativa a nuestro objetivo de implantar un lenguaje ampliado. En LDL1 se tiene la proposición "collect-all" que se expresa vía "set-grouping". LDL1 la incorpora mediante el modelo dominante de acuerdo al concepto de límite de la noción de minimización, el cual aunque interesante ha resultado significativamente complicado [NT89]. A respecto consideramos desde el enfoque semántico que en SuRE es más natural. Además, mientras SuRE tiene un estilo más funcional del set-grouping, LDL1 introduce las funciones de una manera oculta, como se detalla en [RU93].

Por otro lado, en LDL1 los manejadores de conjuntos (unión, intersección, etc.) se construyen vía predicados, lo que resta expresividad y claridad en el manejo de esto ; en SuRE, éstos son definidos por el programador de una forma clara y natural. Con lo que se refiere a la estrategia de búsqueda en LDL1 se tiene que es bottom-up y en SuRE se combina el manejo de tablas memo y la colección de resultados en caminos de búsqueda tipo

top-down. Una parte relevante y novedosa de SuRE es que su semántica operacional está basada en tablas de memorización e iteración de punto fijo, que es una nueva estrategia en el área de los lenguajes de programación lógica para la detección de ciclos existentes en un programa [O95]. En adición a lo anterior, podemos identificar varias clases de programas donde la semántica operacional puede ser simplificada.

Con lo que respecta a CORAL este es un sistema similar a LDL1 surgido en la Universidad de Wisconsin en 1988 cuyo objetivo inicial fue el desarrollo del algoritmo cuyo nombre es "Magic Templates" [RU93], el cual ofrece un potencial para soportar tuplas no base. Una característica importante de CORAL es el uso del algoritmo de "matching" (muy restringido), solo para los casos en que los términos son "ground"[1] . No obstante, el uso del "matching" puede ser utilizado como una alternativa a problemas en los cuales el uso de éste es suficiente para resolver un problema sin tener que gastar mas procesamiento que el necesario, como lo sería la utilización del algoritmo de unificación. En SuRE si se plantea la alternativa de "matching" en la solución de algunos problemas típicos que no requieren de un desgaste mayor de procesamiento vía el algoritmo de unificación. Por otro lado CORAL emplea estratificación, algo que es bueno e incluso SuRE ha implementado también esta característica.

En relación a COL, se puede observar que su noción sobre estratificación es más débil que la propuesta en este trabajo, es decir, la estratificación de programas S-M (set monotonic) está fuera de su alcance. Su aproximación acerca de la semántica declarativa esta basada sobre modelos minimales justificados que corresponden a la aproximación clásica, y además muchos programas que tienen una semántica declarativa en nuestra aproximación conllevarán a una semántica declarativa indefinida. En cuanto a la integración de la programación subconjunto-ecuacional SuRE cuenta con ella, mientras que COL adolece de esta, restandole poder, esto hace a SuRE un lenguaje más poderoso que nos permite la implantación de funciones de una forma más natural y sencilla. Esta característica se describe posteriormente mencionando las ventajas que con esto se logra.

---

[1] **Término ground** es aquel que solo cuenta con argumentos constantes

Como consecuencia de lo anterior, en nuestro trabajo de tesis elegimos trabajar sobre SuRE (ver [JOM95], [Jay92], [JP87], [JP89] y [OJ96]), como una buena alternativa a nuestro objetivo de canalizar nuestras ideas y resultados producto de nuestra investigación. Obviamente el contemplar a SuRE como nuestro lenguaje de trabajo es un tanto ambicioso, considerando que nuestro interés radica en analizar algunas nuevas alternativas que mejoren la manera en que lenguajes como Prolog manejan las BDD y, en particular el manejo de nuevas estructuras como los conjuntos, es que en nuestro trabajo seleccionamos solo aquellas partes de interés para el manejo de conjuntos y las operaciones que con ellos se pueden realizar. Es bien sabido que el contar con estructuras que nos permitan manejar conjuntos y sus operaciones nos amplian el alcance de nuestro lenguaje. Para esto, Proponemos un sublenguaje de SuRE con algunas variaciones a las expuestas en [O95], con la finalidad de mejorar algunas de las deficiencias existentes en Prolog. Así mismo, realizamos la definición formal de la Sintaxis y la Semántica que deben ser consideradas.

Como parte de nuestro trabajo, la definición sintáctica a diferencia de la referida en [O95] se pulió hasta llevarla a una gramática deterministica la cual nos proporciona beneficios tales como los referidos en el capítulo 4. Por lo que respecta a la semántica proponemos una transformación (reescritura) a Programas Normales Lógicos en el sentido técnico en que lo define J.Lloyd [Llo87], adaptándolos para que funcionen correcta y eficientemente en lenguajes como CORAL, implantando los aspectos relevantes al manejo de los conjuntos, ya que estos proporcionan un mayor poder a los sistemas de BDD en el manejo de los datos según revelan nuestras investigaciones.

Como parte de la transformación proponemos el uso del algoritmo Flattening el cual lo adaptamos para transformar programas funcionales a su forma relacional. Cabe mencionar que en nuestro trabajo de investigación sólo consideramos programas jerárquicos y estratificados, ya que estos son programas bien definidos. Como resultado de nuestras investigaciones se sugiere a SuRE, como una buena alternativa a la solución de muchos de los problemas a los que se enfrentan los sistemas de BDD. Además, se

presentan ejemplos típicos resueltos en algunos de los sistemas existentes (como CORAL) y su respectiva propuesta de solución empleada en SuRE, la cual en algunos casos no solo mejora la forma de expresar nuestros algoritmos, sino que también mejora la eficiencia en el procesamiento de ejecución debido a la forma de resolver cada uno de los diferentes problemas. Hemos investigado una extensión muy natural que consiste en generalizar aseveraciones de subconjuntos a aseveraciones parciales y trabajar en dominios reticulados mas generales. Se muestra en [JOM95] que las aseveraciones de orden parcial ayudan a dar claridad y formulaciones concisas a los problemas, involucrando operaciones de agregación y recursión en las consultas a bases de datos deductivas. Nuestra propuesta se fundamenta en un sublenguaje de SuRE, aunque con algunas modificaciones tales como; la selección cuidadosa y relevante de los elementos que permitan a este sublenguaje mantenerlo poderoso y expresivo tal como lo es SuRE.

Finalmente se trabaja la gramática empleada en SuRE para transformarla a una gramática deterministico la que proporciona algunas ventajas como las que se mencionan en el capítulo 4. El diseño del algoritmo flattening para la transformación a programas normales, es decir transformar cláusulas funcionales a cláusulas relacionales y, la transparencia de esta transformación para generar código que igual corre en lenguajes como CORAL y LDL1 los cuales cuentan con constructores de conjuntos y estratificación, o bien en lenguajes como Prolog que tiene un estilo de programación relacional y que carece tanto del manejo de conjuntos como de estratificación. Dando de esta manera una alternativa más en el desarrollo del paradigma de subset-equational.

En el capítulo 1 se define lo que son las Bases de Datos Deductivas, con la finalidad de hallar un lenguaje poderoso y expresivo que nos permita posteriormente trabajar en éste. Además se describen las características que dieron origen al surgimiento de éstas, de igual forma se hace una breve reseña histórica de las BDD a la fecha. En este también se plantean las características que se espera cumplan las BDD, para lo cual se expone Prolog como uno de los lenguajes que ha tenido una mayor proliferación a pesar de tener algunas deficiencias que en la actualidad son conocidos por la gente que lo emplea. Prolog es elegido para el análisis y el estudio de sus particularidades debido a su uso generalizado y que nos permite

exponer nuestras observaciones de una manera clara y concisa, para esto, se muestran ejemplos típicos en los cuales se resaltan las características en que Prolog como algunos otros sistemas un tanto restringidos y no claros adolecen, complicando la solución de este tipo de problemas.

Las necesidades que surgen de las BDD plantean el compromiso de extender el alcance de los lenguajes existentes abocados a la gestión de Bases de Datos Deductivas hacia nuevas alternativas de solución. El capítulo 2 gira en torno a esta necesidad, lo que nos condujo hacia la búsqueda de propuestas que nos permitan por un lado, que el usuario pueda plantear sus algoritmos de una manera clara y sencilla, es decir, que el lenguaje no sea complejo en su escritura. Por otro lado, se hace uso de algunos conceptos básicos y ejemplos que nos permiten analizar la forma de como son atacados estos problemas en algunos sistemas existentes, como lo es Datalog.

De las características discutidas en los dos capítulos anteriores se desprende nuestra propuesta de semántica formal basada en el lenguaje SuRE. El objetivo del capítulo 3 es tratar con la construcción del lenguaje que cuente con el manejo de conjuntos y/o multiconjuntos en un sistema de BDD. Dado que pensar en SuRE es muy ambicioso, nos centramos en los conjuntos y más concretamente en el paradigma subconjunto-ecuacional; dando como resultado un lenguaje que gestione los datos de una BDD con mayor representación.

En el capítulo 4 se presenta el lenguaje prototipo que experimenta con las características de extensión a conjuntos y agregación, estas incrementan los beneficios del lenguaje y son deseables en todo sistema de BDD. Con el fin de tener una base sólida y robusta en el modelado del lenguaje se propone para su definición una **Gramàtica Libre de Contexto Determinística** y con el algoritmo flattening se pueden transformar las cláusulas funcionales y relacionales, tomando en cuenta que para respetar la semántica es necesario completar la teoría con otros axiomas. Siendo el objetivo la **transformación de programas de orden parcial** (en particular los referentes al manejo de conjuntos) **a programas normales** lo cual

es significativo y actual, en este trabajo se definen una serie de macros apoyados en algunos trabajos relacionados y sugerencias del **Dr. Osorio**.

Por último, se presentan algunas conclusiones fruto de nuestra investigación, las que esperamos sirvan para continuar con futuros trabajos relacionados con el presente, alcanzando metas mas ambiciosas y nuevas alternativas a las encontradas en los diferentes artículos a que se hace referencia aquí.

INTRODUCCIÓN A LAS BASES DE DATOS DEDUCTIVAS

El desarrollo de las bases de datos deductivas ha observado un avance significativo en la solución a las necesidades informativas de grandes volúmenes de datos, caracterizadas por la representación de los datos y la operación de éstos, se han podido extender las gestiones de los sistemas para satisfacer las necesidades informativas. El uso de la lógica con su enfoque sintáctico y semántico ha permitido, describir los datos y encontrar las relaciones existentes entre ambos, si además se tiene un manejo de conjuntos los beneficios se ven reflejados en las consultas en los sistemas de BDD, permitiendo realizar la verificación de las restricciones de integridad.

Nuestro interés radica en contar con un lenguaje poderoso y expresivo, que nos permita incorporar y experimentar alternativas de solución a los diversos problemas existentes en los sistemas de BDD. Aquí, presentamos algunas de las soluciones posibles a diversos problemas encontrados en sistemas existentes apoyados en ejemplos que ilustran los beneficios que se logran incorporando dichas soluciones. En seguida, se presentan algunos comentarios generales acerca de lo que son las BDD, sus características básicas y lo que esperamos que tenga cualquier sistema de BDD.

En la sección 1.1 se define y explica la evolución que las bases de datos deductivas han experimentado, las cuales son el tema de interés de este trabajo de tesis. La siguiente sección 1.2 de éste capítulo se aboca al tratamiento de las bases de datos deductivas mostrando los detalles relevantes de éstas. La parte sintáctica de las bases de datos deductivas se presenta en la sección 1.3, brindando algunos ejemplos que ilustran tal sintaxis. Y finalmente en la sección 1.4 se presenta Prolog como una buena alternativa de partida para la realización de nuestro análisis comparativo de las bases de datos deductivas.

1.1 HISTORIA DE LAS BASES DE DATOS DEDUCTIVAS.

La demostración automática de teoremas [Chang & Lee 73] es el origen de las bases de datos deductivas (BDD), para incluir posteriormente la metodología de programación lógica [Lloyd 84]. Los primeros en reconocer la conexión entre la demostración de teoremas y la deducción en sistemas de preguntas y respuestas fueron Green y Raphael [Green]. Usando el principio de resolución de Robinson plantean un sistema de resolución de consultas a una base de datos, en donde para dar la respuesta se requiere un sistema de "inferencia" que usa sistemáticamente un mecanismo de deducción.

El principio de resolución de Robinson [1965] usa sistemáticamente el concepto de unificación dado por Herbrand [1930], adicionando una estrategia de resolución de derivar una sentencia a partir de un par de ellas. De esta manera Robinson proporciona un algoritmo que calcula el unificador más general de dos términos del cual podemos derivar todos los términos unificados. Debido a su importancia práctica, se inició una fructífera investigación que dio origen a encontrar nuevos algoritmos eficientes y propuestas nuevas a estrategias de deducción; [Paterson 78], [Mastelli 82].

Tomando como base lo anterior a mediados de los setentas se desarrollaron los primeros sistemas de BDD fundamentados en particulares paradigmas de deducción que incluyen procedimientos de búsqueda, técnicas de indexación y optimización de las consultas.

Sistema de Chang	DEDUCE-1 & DEDUCE-2
Sistema de Kellog	DADM
Sistema de Minker	MRPPS

La programación en lógica ha sido un tema fundamental en la teoría y práctica de Ciencias de la Computación, motivada principalmente por la elegancia y cualidades precisas de la lógica matemática. La base fue, también, el trabajo original de Robinson y el uso del calculo de predicados de primer orden como lenguaje de programación. El tema empieza ha ser discutido y desarrollado ampliamente por L.R. Apt, M.H. Van Emden [Apt82] y R.A. Kowalsky [Kowlsky 74 y 79] y M.H. Van Hemden y Kowalsky [76] para conducir poco a

poco al lenguaje de programación PROLOG W.F. Clocksin y C.S. Mellish [1981], P. Soussel [1975].

Cabe mencionar que otro desarrollo importante fue la identificación en 1970 por E.F. Codd del calculo relacional de primer orden, con interpretación finita, como el formalismo natural para la descripción y manejo de las bases de datos, dando como consecuencia que en 1970 fue la década de la programación lógica y los sistemas de bases de datos relacionales, que en estas dos últimas décadas han consolidado una sólida tecnología

El desafío de BDD y sistemas basados en conocimiento es continuar la tecnología basada en herramientas de programación PROLOG con la eficiencia de la tecnología de bases de datos combinando los beneficios como mencionaremos mas tarde (ejemplo LDL)

Como otros desarrollos posteriores a los citados anteriormente podemos destacar la fuerza que han tenido las nuevas técnicas de gráficas de conexión para el desarrollo de la evaluación eficiente de consultas, como el propuesto por Henschen y Naqvi dentro del contexto de las bases de datos.

A mediados de la década de los 80´s [Bancilhen y Ramakishnan 1986] surge un fuerte desarrollo sobre el procesamiento recursivo de consultas. Este desarrollo dio nuevos resultados que han hecho que este procesamiento sea mas eficiente y claro.

Dentro de los proyectos más destacados tenemos el proyecto LDL [TZ86] de MCC en Austin, el proyecto NAIL-GLUE [MUG86] de Stanford y EKS-V1 [VBK90] los cuales trajeron consigo contribuciones importantes en la construcción de sistemas prototipos. No olvidando que los proyectos EKS-V1 y LDL representan los primeros esfuerzos realizados fuera de las universidades.

Nuestro objetivo principal es el de presentar el uso de los conjuntos y constructores de conjuntos, los cuales son usados comúnmente en las aplicaciones de procesamiento de datos. LDL se distingue por el tratamiento que hace del manejo de los conjuntos, a pesar de que

estos no son manejados de manera directa como se hace en nuestro trabajo. Además, en este lenguaje se trata el manejo de la negación, la que se basa en la estratificación y establece que no exista recursión a través de la negación. Este punto es de suma importancia para un sistema de BDD, es por esto que la incluimos en nuestro trabajo como una parte necesaria.

LDL ha sido valorado como una herramienta poderosa y flexible para la especificación de aplicaciones de conocimiento intencional además, éste ha sido enriquecido con la incorporación de atributos tales como: nombrados, herencia y nuevas primitivas. Incluye además capacidad de actualización sobre las relaciones derivadas.

CORAL es un sistema de base de datos deductivas que contiene un lenguaje declarativo, que proporciona una variedad amplia de métodos de evaluación permitiendo una combinación en su programación imperativa y declarativa. Coral admite reglas basadas en cláusulas de Horn, así como una semántica declarativa basada en el "modelo minimal" de Herbrand. Intuitivamente, esto significa que las reglas pueden ser entendidas como simples sentencias if-then en la lógica sin considerar el orden de evaluación. Lo relevante de CORAL es que los programas con negación o set-grouping son restringidos a ser modularmente estratificados de izquierda a derecha lo cual le da importancia en la recursión. Esto debido a que se resuelve el problema de recursión vía metas negativas como se describe en la sección 1.2.6.

Por otro lado, un programa en CORAL es una colección de módulos, y cualquier módulo puede ser entendido como una simple definición de una o más relaciones exportadas, o conjunto de hechos.

1.1.1 Prototipos y sistemas

A continuación, del artículo de Raghu y Jeffrey [RU93] se muestra una tabla (fig. 1.1) que resume las características relevantes de los sistemas de BDD más conocidos. Los primeros puntos importantes para nuestro trabajo son el manejo de la Recursión, Negación y Agregación loas cuales son tratados por varios de los sistemas que se observan en la tabla, todos coinciden en el empleo del manejo de negación estratificada, aunque en el caso de

Coral es modularmente estratificada, similar a Sure lo que le da mayor eficiencia en su ejecución. Además, la inclusión de agregación da a un lenguaje un alcance mayor, sobre todo si en este se incluyen nuevas herramientas tales como los conjuntos, que es nuestro caso. Lo relevante en nuestro caso con el manejo de conjuntos es que éstos son manejados de manera natural y transparente para el usuario, a diferencia de otros como LDL en el que el manejo de tales estructuras es de forma indirecta y compleja.

Por otro lado, en la segunda parte de la tabla sobresalen la optimización e interfaces con algunos otros sistemas. Como se puede observar la evolución de los sistemas está en la dirección del uso de lenguajes orientados a objetos. En nuestra propuesta la optimización la manejamos a través del uso de un simple "matching" en lugar del complejo algoritmo de "unificación". Cabe mencionar que el uso del matching es eficiente y de bajo costo para muchos de los casos, aunque no se descarta el uso del algoritmo de unificación para aquellos casos en que es necesario y que el matching queda limitado. Además, en SuRE se introduce el manejo de una tabla de memorización, la que se describe en la sección 3.2.3., esta tabla nos permite evitar ciclos infinitos en la ejecución de ciertas definiciones.

Sistema o Prototipo	Desarrollado en:	Cuenta con Recursión	Maneja Negación	Incluye Agregación
Aditi	U. Melbourne	General	Estratificada	Estratificada
Col	INRIA	-	Estratificada	Estratificada
ConceptBase	U. Aachen	General	Localmente Estratificada	-
Coral	U. Wisconsin	General	Modularmente Estratificada	Modularmente Estratificada
EKS-VI	ECRC	General	Estratificada	Super Conjunto de Estratificación
Logic Base	Universidad Simon Fraser	Lineal, no Lineal	Estratificada	-
DECLARE	MAD Intelligent Sys.	General	Localmente Estratificada	Super conjunto de Estratificación
Hy+	U. Toronto	Rutas de consulta	Estratificada	Estratificada
X4	U. Karlsruhe	General (solo Preds-binarios)	-	-
LDL LDL++	MCC	General	Estratificada localmente restringida	Estratificada localmente restringida
LOGRES	Politécnico de Milán	Lineal	Semántica inflacionaria	Estratificada
LOLA	U.T. Munich	General	Estratificada	Predicados calculados
Glue-Nail	U. Stanford	General	Bien fundada	Solo pegar
Startburst	IBM Almaden	General	Estratificada	Estratificada
XSB	Suny Stony Brook	General	Bien fundada	Modularmente Estratificada

Figura 1.1 a) Resumen de alcances en Prototipos y Sistemas más conocidos

Sistema o Prototipo	Actualiza-ciones	Restric-ciones	Optimizaciones	Almacena-miento	Interfases
Aditi	-	-	Magic Sets, SN, Join-Order Selection	EDB, IDB	Prolog
COL	-	-	-	Memorial Principal	ML
ConceptBase	Si	Si	Magic Sets, SN	EDB	C, Prolog
CORAL	Si	-	Magic Sets, SN, Context Factoring, Projection pushing	EDB, IDB	C, C++ Extensible
EKS-VI	Si	Si	Query_subquery left/right linear	EDB, IDB	Persistent Prolog
LogicBase	-	-	Chain_based Evaluation	EDB, IDB	C, C++, SQL
DECLARE	-	-	Magic Sets, SN Projection pushing	EDB	C, List
Hy+	-	-	-	Principal Memory	Prolog, LDL, Coral, Smalltalk
LDL LDL++	Si	-	Magic Sets, SN, left/right linear Projection pushing Bushy depth-first	EDB	C, C++, SQL
LOGRES	Si	Si	Algebraic, SN	EDB, IDB	INFORMIX
LOLA	-	Si	Magic Sets, SN, Projection pushing, Join-order selection	EDB	TransBase (SQL)
X4	-	Si	Ninguna, Eval. Top_down	EDB	Lisp
Glue-Nail	-	-	Magic Sets, SN, Rigth_linear Join-order selection	EDB	-
Startburst	-	-	Magic Sets, variante SN	EDB, IDB	Extensible
XSB	-	-	Memoing, Top_down	EDB, IDB	C, Prolog

Figura 1.1 b) Resumen de alcances en Prototipos y Sistemas más conocidos

Como se puede observar en las tablas anteriores casi todos los sistemas cuentan con manejo de recursión de forma general, lo cual, en el lenguaje SuRE y en nuestro trabajo se incluye ya que esta es importante para la resolución de cualquier problema. Por otro lado, la negación y agregación se maneja de forma estratificada en la mayoría de los lenguajes así como en SuRE, aunque en nuestra propuesta se proponen algunos cambios en la implementación de ésta. Los detalles de esta propuesta se describen en el capítulo 3 y 4.

1.2 CARACTERISTICAS DE LAS BASES DE DATOS DEDUCTIVAS

En esta sección presentamos conceptos generales que dan relevancia a los sistemas de BDD y a su vez los hacen poderosos y diferentes de los sistemas de bases de datos relacionales, empezamos definiendo lo que son las BDD para pasar a el lenguaje de la lógica, el cual es empleado para la representación de los datos y reglas, éste lenguaje nos proporciona algunos beneficios importantes que deben ser parte fundamental de cualquier sistema de BDD. Posteriormente se mencionarán conceptos básicos de todo sistema de BDD resaltando aquellos que en nuestra investigación proporcionan mayores beneficios para el "usuario" y para su ejecución.

1.2.1 Bases de datos deductivas

Una base de datos deductiva es una base de datos en la cual nuevos hechos se pueden derivar de algunos otros que fueron introducidos explícitamente. Las BDD pueden ser determinadas. Las BDD determinadas se definen como un caso particular de la teoría de primer orden, junto con un conjunto de restricciones de integridad. Esta teoría se fundamenta de las bases de datos convencionales con la adición de una nueva clase de axiomas, las cuales se establecen para las leyes deductivas y para una re formulación que completa los axiomas.

Es por esto que en los sistemas de BDD su lenguaje de consulta y estructura de almacenamiento se diseñan basados en un "modelo lógico de datos", dando relevancia a las

bases de datos deductivas, pues con esta característica se ofrece una definición de datos rica en semántica como se describe en seguida.

Un modelo lógico de datos consiste de un lenguaje lógico para describir datos y sus relaciones así como una estrategia para la manipulación de éstos que hace posible atender consultas y realizar la validación de restricciones de integridad. El lenguaje matemático empleado para describir los datos en un modelo de base de datos deductiva es un lenguaje lógico de primer orden, cuyo fundamento teórico [Lloy87] ofrece las siguientes ventajas:

1. Ofrece una semántica bien fundamentada.
2. Expresa hechos, consultas y restricciones de integridad.
3. Puede resolver problemas de valores nulos y datos indefinidos.
4. El uso de una regla simple puede reemplazar muchos hechos explícitos. Esta característica proporciona un entorno expresivo y económico (disminuye el almacenamiento en disco), de los hechos para el modelado de datos.
5. El concepto de BDD generaliza el concepto de bases de datos relacionales.

Estos cinco puntos son los que nos conducen a tomar al lenguaje de la lógica como la base de desarrollo del lenguaje de una base de datos deductiva. En primer lugar, por contar con una semántica bien definida, segundo por su poder expresivo implícito y en tercer lugar, por el alcance que puede tener al contar con extensiones tales como el manejo de conjuntos a través de un lenguaje declarativo. Un lenguaje declarativo nos lleva a la conclusión de que una BDD es un conjunto de reglas no ordenadas (una característica que en Prolog no se cumple). Dada una base de datos, esta se puede particionar en un conjunto de hechos y un conjunto de reglas. El conjunto de hechos es llamado Base de Datos Extensional (BDE) y el conjunto de las reglas se denomina Base de Datos Intensional (BDI).

1.2.2 Que es un Hecho? (BDE)

Suponga que deseamos establecer que "Fernando es padre de kimberley" Este **"hecho"** consta de dos objetos llamados **fernando** y **kimberley** y una relación entre ambos llamada **Padre**, de aquí se observa que los "hechos" son normalmente representados por un

predicado (relación) con argumentos que son variables ("objetos" que pueden ser constantes). Por ejemplo, el hecho:

Padre (fernando, kimberley)

significa que Fernando es Padre de kimberley. Donde **Padre** es predicado representado extensionalmente, esto es, almacenando en la base de datos una relación de todos los hechos verdaderos para el predicado Padre. Entonces (fernando, kimberley) debe ser uno de los hechos en la relación almacenada o extensional.

1.2.3 Sintaxis de las Reglas. (BDI)

Las **reglas** son usadas cuando establecemos que un hecho depende de un grupo de hechos. Por tanto, una **regla** es un enunciado acerca de objetos y las relaciones existentes entre ellos. Para el manejo de estas reglas necesitamos de una notación del estilo de Prolog como sigue:

p :- q1, q2, ... , qn

Esta regla declarativa es: q1, q2, ..., qn implica p. Cada una de las proposiciones: p (la cabeza) y qi (las submetas del cuerpo) son fórmulas atómicas (o literales) consistiendo de un predicado que contiene términos, donde cada término puede ser constante, variable o símbolo de función aplicado a términos.

1.2.4 Recursión.

Prolog permite la declaración de predicados recursivos. Un predicado es recursivo si su definición involucra al predicado mismo, su forma general es la siguiente:

p(t) :- ..., p(t'), ...

La especificación de la BDI en muchos sistemas permiten a las reglas el uso de recursión general. Sin embargo, algunos las limitan a recursión lineal (define más adelante) o las restringen a formas relacionadas con el grafo de búsqueda, tales como la cerradura transitiva.

Ejemplo 1.1 Aquí se define la relación *"ser primos de la misma generación"*, que consiste en que se trate de la misma persona o bien que los padres de ambos sean primos de la misma generación.

```
primos_misma_generación(X,Y) :- igual(X,Y).
primos_misma_generación(X,Y) :- padre(U,X),
                                primos_misma_generación(U,V), padre(V,Y).
```

los predicados anteriores definen la noción de empezar en la misma generación recursivamente. No obstante, implementaciones de SQL no soportan tales definiciones, como este ejemplo (el cual es linealmente recursivo); por lo tanto, una de las extensiones importantes en los sistemas deductivos está en la habilidad de soportar programas de consultas con reglas recursivas.

La optimización de consultas recursivas ha sido un área de investigación y a menudo enfocada, a clases importantes de recursión como las definidas en esta sección. Se dice que un predicado p depende de un predicado q -no necesariamente distinto de p- si alguna regla con p en la cabeza tiene una submeta cuyo predicado es q o (recursivamente) depende de q y q depende de p, p y q son mutuamente recursivos.

Un programa es llamado **linealmente recursivo** si cada regla contiene al menos una submeta cuyo predicado es mutuamente recursivo con la cabeza del predicado; en otras palabras hay una sola ocurrencia del predicado recursivo en el cuerpo de la regla. Las reglas del ejemplo 1.1 son linealmente recursivas mientras que la regla del ejemplo 1.2 no lo es.

Ejemplo 1.2 La siguiente relación significa que *"X es un pariente antecesor de Y"* si existe por un lado, que X sea ancestro de Z y por otro que este Z sea un ancestro de Y.

```
ancestro(X,Y) :- ancestro(X,Z), ancestro(Z,Y)
```

La presencia de reglas recursivas abre una variedad de nuevas opciones y problemas (conjuntos mágicos, top-down, etc.) que deben ser resueltos..

Con respecto a la negación y agregación se puede decir que éstas constituyen extensiones a los programas basados en cláusulas de Horn. Dado que estas contribuyen a elevar el poder de expresividad (semánticamente) de los lenguajes de consulta, estas características se presentan en las secciones siguientes.

1.2.5 Negación.

Un lenguaje de consulta de base de datos puede enriquecerse permitiendo submetas negadas en el cuerpo de las reglas, no obstante, se pierde una importante propiedad en las mismas. A pesar de esto, en la vida diaria también existe un **no** para poder representar proposiciones negativas.

Ejemplo 1.3 si el expediente de Joe **no** esta, entonces **no** esta registrado.
si **no** juega futbol, entonces **no** tiene tacos

Las proposiciones negativas se pueden ver con negación como falla (NF), esto nos indica que si una proposición es verdadera y no la podemos concluir como tal, entonces se infiere como falsa. Esto no es correcto, ya que en la lógica clásica para inferir una cláusula que sea falsa debemos demostrar que la proposición es falsa, lo cual no se puede demostrar, por lo que se entra en una polémica en la elección de la semántica declarativa del programa, dicha semántica debe ser adecuada a la programación lógica con negación.

Una **interpretación** es una función que asigna a cada proposición **P** un valor único de falso o verdadero (0 o 1 respectivamente en tablas de verdad). En tablas de verdad, cada entrada de la tabla es una interpretación. Analizando la tabla 1.1 podemos observar que las tres primeras interpretaciones son verdaderas y la cuarta es falsa cuando p y q son falsas.

Como se observó en la tabla tenemos interpretaciones ciertas o falsas, pero solo nos interesan las que sean verdaderas para llegar al siguiente resultado. Si tenemos a **T** como un

conjunto de fórmulas propuestas, donde cada fórmula esta compuesta de proposiciones, entonces existe una interpretación verdadera **I** de **T** que **es un modelo sí y sólo si** todas las fórmulas de **T** son ciertas en **I**. Denotaremos M(T), como el conjunto de todos los modelos de **T**.

Retomando nuevamente la tabla 1.1, observamos que los modelos de T son: M(T) = {p-cierta, q-cierta}, {p-cierta, q-falsa} y {p-falsa, q-cierta}, todas estas son las interpretaciones en se hace cierta a la proposición.

Un **modelo minimal** es un modelo tal que ningún subconjunto es un modelo [Das92]. Es deseable que el "usuario" tuviera en mente el modelo minimal cuando escribe el programa lógico. En el siguiente ejemplo 1.4 se observa esto.

Ejemplo 1.4 Sea P el siguiente programa y su tabla de verdad.

$$p :- \neg q$$

Tabla 1.1 tabla de verdad del programa P

p	**q**	$\neg q \rightarrow p$
1	1	1
1	0	1
0	1	1
0	0	0

Los modelos M(P) son: {q}, {p}, {p, q}

Los modelos minimales serían: {q} y {p}, ya que estos son los conjuntos mas pequeños y que además tienen valor de cierto. Este tipo de problemas no los podemos resolver, como en la semántica del cálculo de proposiciones, tenemos que recurrir a otro tipo de semántica como la propuesta por Clark [Clark78] (Completion Comp(P)), la cual intenta tener una simple fórmula, donde las literales negadas de P sean una consecuencia lógica de Comp(P) y así considerarlas verdaderas. Tomando nuevamente el ejemplo 1.4, entonces Comp(P) será:

$$p \leftrightarrow \neg q \wedge \neg q$$

Tabla 1.2 Tabla de verad para Comp(P)

p	q	$p \rightarrow \neg q$	$p \leftrightarrow \neg q$	$p \leftrightarrow \neg q \wedge \neg q$
1	1	0	0	0
1	0	1	1	1
0	1	1	1	0
0	0	1	0	0

Como se puede observar el segundo renglón de la tabla 1.3 infiere **¬q.** Analicemos el siguiente ejemplo.

Ejemplo 1.5 Transiciones existentes entre distintos puntos.

tr(x,y) :- edge(x,y).
tr(x,y) :- edge(x,z), tr(z,y).
edge(1,2)
edge(2,3)
Mi = { edge(1,1), edge(1,2), edge(1,3), ... , edge(3,3), tr(1,1), tr(1,2), ... , tr(3,3) }
Modelo minimal = { edge(1,2), edge(2,3), tr(1,2), tr(2,3), tr(1,3) }
este es el modelo minimal, ya que no existe ningún subconjunto en este, tal que pueda ser un modelo.

El modelo minimal de un programa P se define como $M = \{ A / P \vdash A \}$, es decir, A es una consecuencia lógica de P. No obstante, en presencia de literales negadas un programa puede no tener un modelo minimal, por ejemplo la regla del ejemplo 1.6

Ejemplo 1.6 Aquí se muestra como se aplica la anterior definición

$$p(a) :- \neg p(b).$$

nuestro predicado cuenta con la literal a y b, lo que lo hace tener dos modelos minimales { p(a) } y { p(b) }. En estos casos, una alternativa es la siguiente.

El significado de un programa con negación es usualmente dado por algunos "intended models", es decir, dar un modelo que trate de inferir lo que el "usuario" quiere decir [RU93]. El reto es desarrollar algoritmos para escoger un modelo ("inteded model") donde:

1. - Las reglas signifiquen lo que el "usuario" desea.
2. - Permita resolver consultas eficientemente. En particular, es deseable que este trabaje bien con una transformación de **magics sets** (técnica que nos permite reescribir reglas para cada tipo de consulta, esto es, qué argumentos del predicado están ligados a constantes y cuales a variables), en el sentido de que se puedan modificar las reglas y las reglas resultantes al aplicar Comp(P), estas permitirían que sólo una parte relevante del problema seleccionado fuera calculado eficientemente. (Alternativamente, otras técnicas de evaluación eficientes deben ser desarrolladas). Se nota que depender de un "intended model" en general, resulta un tratamiento de la negación que difiere de la lógica clásica.
3. - Que la semántica declarativa sea adecuada.

En el ejemplo 1.6, escoger uno de los dos modelos minimales sobre el otro no puede justificarse en términos de la lógica clásica, ya que la regla es lógicamente equivalente a **p(a)** v **p(b),** por lo tanto, Clark propone una estrategia (negación como falla) que trate de inferir lo que el "usuario" desea. Desafortunadamente esta no es propia de los lenguajes de programación lógica, perdiendo el sentido de semántica declarativa y toma el sentido de semántica procedural como en otros lenguajes de programación. En los ejemplos 1.5 y 1.6 se observa esto es decir, no especifica que tipo de valores son aceptados en los argumentos y admite cuantificadores sin restringirlos, además al transformar el programa en Comp(P), resulta más difícil de interpretar para inferir la negación como falla. Por esto en nuestro trabajo se propone utilizar otro tipo de semántica basada en estratificación.

1.2.6 Estratificación (semántica por niveles).

Un programa es estratificado si no existe recursión a través de la negación [Das92]. Una clase importante de negación es la **estratificación,** la cual tratamos en nuestra investigación de trabajo de tesis. Es importante destacar que algunos sistemas como CORAL, COL, LDL,

y SuRE cuentan con estratificación (con diferentes enfoques) que los hace superiores a Prolog, ya que en ejemplos que presentamos posteriormente estos sólo son resueltos si se cuenta con algún tipo de estratificación.

Este tipo de semántica refina los programas por niveles, donde cada nivel contiene cláusulas que pueden ser recursivas o no, en dos formas: a) recursión sin negación, b) negación sin recursión

Esto indica que cada nivel debe aparecer sin recursión a través de la negación, para que se encuentre estratificado. [Das92].

Ejemplo 1.7 Nuevamente presentamos las cláusulas para hallar los ancestros de una persona

anc(X,Y) :- par(X,Y).

anc(X,Y) :- par(X,Z), anc(Z,Y).

nocyc(X,Y) :- anc(X,Y), ¬ anc(Y,X) .

es estratificado porque la definición del predicado **nocyc** depende de la definición de **anc**, pero la definición de **anc** no depende de la definición de **nocyc,** es decir, en la definición tanto de nocyc como de anc no hay recursión negada. La última regla es aplicada solamente cuando todos los hechos de **anc** son conocidos. En nuestra implantación si se permiten programas estratificados, lo que hace de nuestro lenguaje más eficiente en la resolución de este tipo de problemas.

Un programa es **localmente estratificado** para una base de datos si: cuando se sustituyen constantes por variables en todas las formas posibles en las reglas instanciadas resultantes, no existe alguna recursión a través de negación [Prz88].

Ejemplo 1.8 Cláusulas que determinan si un número es par

par(0).

par (X) :- predecesor (X, Y), ¬par (Y)),

predecesor (1, 0).
predecesor (2, 1).
predecesor (3, 2).

El programa no se encuentra estratificado ya que existe la recursión de par a través de la negación.

La semántica estratificada es una alternativa para manejar cláusulas negativas, dichas cláusulas no dependerían de la consecuencia lógica, y si de los modelos mínimos, garantizando que si un programa se encuentra estratificado se asegura que existe un modelo mínimo.

1.2.7 Set Grouping.

El Set Grouping se define para generar un conjunto anidado de valores, en LDL (Lenguaje Lógico de Datos) fue originalmente propuesto para generar un multiconjunto anidado de valores. Esta característica es de vital importancia, sobre todo porque es bien conocido que en muchos de los problemas que se presentan en los sistemas de BDD, pueden ser resueltos de una mejor manera si el sistema de BDD soporta conjuntos, los cuales le permiten al "usuario" no distraer su atención del problema real (como sucede en lenguajes como LDL, en el que los conjuntos son manejados de forma oculta), buscando la forma de resolver problemas que de ser manejados através de conjuntos sería transparente y natural su solución.

El que un sistema de BDD soporte el manejo de conjuntos de una forma natural, facilita aún mas al "usuario" la implementación del problema sin distracciones innecesarias.

El siguiente ejemplo 1.9 ilustra la creación de multiconjuntos y el uso de un agrupamiento o constructor de agregación típico del lenguaje CORAL < >:

Ejemplo 1.9 En CORAL el manejo de conjuntos es utilizando operadores tales como member, el cual es ilustrado en seguida.

```
module declse_eg2.
export okteam(f), team(f), engineer(f), pilot(f), doctor(f).

engineer(joe).
engineer(jack).
pilot(amy).
pilot(jack).
doctor(jack).
doctor(paula).

team({jack}).
team({amy, joe}).
team({amy, joe, paula}).
team({amy, joe, paula, jack}).
```

un buen equipo debe contener un doctor, un pilot y un engineer, no necesitan ser necesariamente gente diferente; sin embargo, el tamaño del equipo debe ser a lo más de 3 personas.

```
okteam(S) :- team(S), count(S, C), C<=3, member(S, X), member(S, Y),
             member(S, Z), engineer(X), pilot(Y), doctor(Z).
end_module.
```

Este programa ilustra el uso de un importante operador multiconjuntos, el predicado member. Cuando el predicado member es llamado el primer argumento es ligado a un multiconjunto y el segundo argumento a una variable, esto sucede repetidamente con el segundo argumento ligado a cada uno de los elementos del multiconjunto. Esto indica que podemos tomar un multiconjunto y expresar condiciones que involucran elementos de este conjunto..

En LDL, okteam puede ser definido por la regla siguiente:

```
okteam(S) :- team(X, Y, Z), engineer(X), pilot(Y), doctor(Z).
```

la ausencia de la literal count es un detalle técnico; la diferencia importante es que en el argumento de team puede ser especificado un conjunto usando una plantilla conteniendo variables. Esto es posible en LDL gracias a que hace uso de un empatamiento sobre conjuntos, mientras que en CORAL esto no es soportado.

SuRE al igual que LDL soporta la manipulación de conjuntos mediante variables, debido a que también utiliza el "matching" para conjuntos. Para el caso del lenguaje SuRE la manipulación de conjuntos es transparente y sencilla. Veamos algunos ejemplos que nos muestren su manejo.

Ejemplo 1.10 Sea la siguiente cláusula para definir la intersección de dos conjuntos:

intersect ({X_}, {X_}) ⊇ {X}

cuando los patrones "conjunto" ocurren en el lado izquierdo de una cláusula, todos los matching a través de estos patrones son usados en la instanciación correspondiente al lado derecho de la expresión y cada conjunto resultado es tomado para formar el super conjunto, que es la unión de todos los conjuntos resultado, es decir, el super conjunto de las aseveraciones. Este mecanismo de collect-all detecta cuando no hay matching y entonces obtiene el conjunto ∅, que es el mecanismo de vacío como falla (emptiness as failure).

Ejemplo 1.11 La unión de dos conjuntos se define como

unión(X1, X2) ⊇ X1

unión(X1, X2) ⊇ X2

gracias al mecanismo de collect-all se permite incluir la unión de X1 ∪ X2.

Ejemplo 1.12 La diferencia de dos conjuntos se define como sigue

diff(S, ∅) ⊇ S

diff({X_}, {Y_}) ⊇ if member(X, {Y\S}) then ∅ else {X}.

Los tres ejemplos anteriores tienen un significado "prometido" ("intended") y son consistentes de acuerdo a la programación lógica.

1.3 PROLOG COMO LENGUAJE DE CONSULTA DE BDD

Empezaremos nuestra descripción de Prolog abordando lo referente a su semántica declarativa. Una cláusula en este tipo de programación son las llamadas cláusulas de Horn con una literal positiva en la cabeza de la cláusula y una variedad de literales o literales negadas en el cuerpo de la misma, entonces cada cláusula del programa se simboliza así:

$$A :- B1, ..., Bn$$

donde cada cláusula representa una proposición o una relación entre proposiciones, para que se puedan inferir otras a partir de ellas, además cada literal se ha ido adaptando para que sean de acuerdo a la semántica declarativa del cálculo de predicados. En Prolog le son agregados símbolos, predicados y constructores, que no tienen que ver con la sintaxis de la semántica declarativa en programación lógica, y la justificación de este tipo de características es para obtener una mejor ejecución, incrementar legibilidad y para representar el conocimiento, a estas características les llamaremos extra lógicas, pues se desvían de la semántica declarativa.

Prolog es un lenguaje que ha mejorado como lenguaje de programación, aunque ha descuidado los problemas de robustez e incompletéz, así como la facilidad de controlar la información explícitamente (el **assert** que incrementa reglas o con el **cut** se interrumpe la secuencia normal del programa). En este sentido se espera que Prolog mejore.

Otro de los problemas que hay en Prolog es el operador not para la negación como falla y el constructor setof que sirve para recolectar todas las soluciones de una consulta, por lo que se necesita mejorar la eficiencia para que las implementaciones en Prolog sean completas y robustas.

1.3.1 Características extra lógicas de Prolog

La estrategia de evaluación de Prolog es "depth first search" (búsqueda del primero en profundidad) la cual puede conducir a ciclos infinitos, tanto para programas positivos como en la ausencia de símbolos de función o aritméticos. De los sistemas analizados, se observa que una evaluación en profundidad es incompleta, ya que solo se explora una rama y nunca

se toma otra alternativa. Debido a esto, en SuRE se emplea una evaluación "breadth first search" (búsqueda a lo ancho" combinada con un estilo funcional utilizando tablas de memorización y cálculo repetido hacia un punto fijo lo cual le da esa completéz esperada. Este tipo de evaluación particular de SuRE se discute con mas detalle y ejemplos en el capítulo 3 y 4. Además, siempre es importante tener en cuenta, la completéz y terminación del método de evaluación.

El operador **cut** es un predicado que reduce el espacio de búsqueda, ya que poda las ramas del árbol de búsqueda, evitando de esta forma que sean consideradas otras soluciones alternativas. Veamos la siguiente cláusula

$$A :- L_1, \ldots, L_i, !, L_{i+1}, \ldots, L_n$$

Para encontrar la solución, el cut (!) poda todas las soluciones alternativas de la conjunción $L_1, \ldots, L_i$, pero no afecta la conjunción $L_{i+1}, \ldots, L_n$

Ejemplo 1.13 Sea P el siguiente programa

R1. p(X, Y) :- q(X, Y), not r(Y).
R2. p(X, Y) :- s(X), r(Y).
F1. q(a, b).
F2. q(a, c).
F3. r(c).
F4. s(a).
F5. t(a).

y la meta G: ? p(X, Y), !, t(X).

el árbol de búsqueda en Prolog para P ∪ {G} se muestra en la fig. 1.2. La rama más a la izquierda es la derivación de éxito.

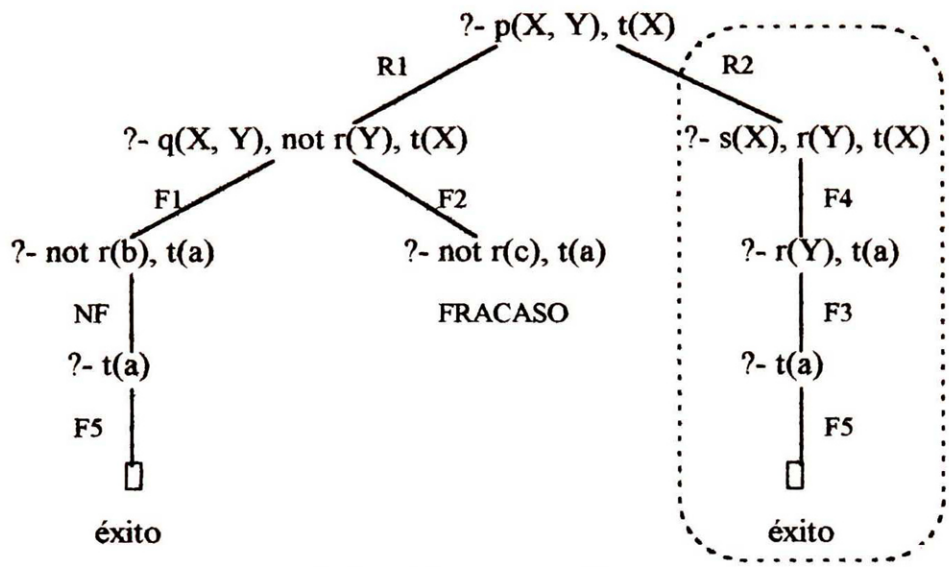

Fig. 1.2 árbol de búsqueda en Prolog

lo que el predicado cut hace en este ejemplo es podar el árbol quitando la rama más a la derecha (la encerrada por una linea punteada), por lo tanto la solución alternativa de p(X, Y) es podada del árbol de búsqueda.

La negación en Prolog esta basada en la negación como falla, y se encuentra implementada por el operador **not**, para un átomo la negación estaría representada como:

Ejemplo 1.14 Sea P el siguiente programa

p(a).

q(b).

la meta

? ¬p(X), q(X).

El par de metas son iguales, excepto que el orden de las literales se encuentra cambiado con la estrategia de más a la izquierda, la primera meta tiene un éxito y regresa X = a, pero en la segunda meta ocurre una falla finita en el árbol de búsqueda, entonces el programa no genera ninguna respuesta. El hecho de que no genere respuesta, o que la respuesta sea errónea se debe a que la variable no está instanciada cuando se realiza la negación.

En el siguiente capítulo abordaremos la necesidad de extender las cláusulas de Horn, a un lenguaje de BDD que le permita al "usuario" contar con herramientas para el desarrollo

de aplicaciones de una manera mas clara, natural y sencilla. Evitando con estas, la distracción que un "usuario" enfrenta al trabajar con lenguajes de BDD que no cuente con tales características.

Por otro lado, hemos mostrado en este capítulo la necesidad de corregir los problemas que originalmente se cometieron en los lenguajes pioneros de los sistemas de BDD. Esto nos permitirá avanzar de manera substancial en el diseño y forma de atacar los diversos problemas vistos anteriormente.

¿POR QUÉ ES NECESARIA LA EXTENSIÓN?

Como se vio en el capítulo anterior, lenguajes como Prolog tienen aún muchas características erróneas. Si bien Prolog ha sido afortunado al ser de los más utilizados en el área de las bases de datos deductivas y la programación lógica, Prolog rompe con la idea de un lenguaje de programación lógica, especialmente por tener un inadecuado e incompleto sistema de demostración (por ser sensitivo al orden en que se definen las reglas, el operador cut, evaluación Depth first search, predicados especiales, etc., descritos en el capítulo anterior) y algunas otras características que evitan que programas en Prolog funcionen correctamente, como uno lo esperaría. Además, esto conduce al "usuario" a dedicar un mayor tiempo al desarrollo de sus aplicaciones.

Por lo anterior, es que surgen nuevas ideas, las cuales dan origen a desarrollos de lenguajes tales como DATALOG. Comenzamos nuestro análisis tomando a DATALOG por ser el primero en incorporar nuevos conceptos relacionados con los sistemas de bases de datos (como los mostrados en la figura 2.3). Y esto, con la finalidad de proporcionar al "usuario" nuevas herramientas, más completas y permitiéndole no distraerse en restricciones impuestas por el sistema de BDD, obteniendo de esta manera aplicaciones claras, naturales y eficientes en su implementación y evaluación.

En la sección 2.1 se plantea la necesidad de extender los lenguajes basados en cláusulas de Horn dando las justificaciones para dicha extensión y presentando a Datalog como uno de los primeros sistemas (aunque sólo a nivel experimental y teórico) que proporcionan este avance en la extensión necesaria para contar con sistemas de BDD que incluyen el manejo de conjuntos; En la sección 2.2 se presentan algunos casos que son de nuestro interés para mostrar características de extensión a los lenguajes de sistemas de BDD; En seguida, en la sección 2.3 nos abocamos a la forma en que se incorporan estas extensiones a lenguajes como SuRE, LDL y COL.

2.1 EXTENSIÓN A CLÁUSULAS DE HORN

La programación lógica ofrece gran expresividad para consultas, superior al ofrecido por la definición de datos y lenguajes de manipulación de bases de datos relacionales. No obstante, los sistemas de programación lógica no cuentan con la capacidad para manejar grandes volúmenes de datos, compartirlos, etc., por lo que requieren ser ampliados incorporando algunas características de las bases de datos relacionales, así como extensiones que den un mayor poder a este tipo de lenguajes.

La necesidad de manejar grandes volúmenes de datos, compartirlos, etc. y la extensión natural de la programación lógica dan origen a la construcción de nuevos lenguajes utilizando la lógica como el medio para realizar las consultas. A estos les llamamos Sistemas de Bases de Datos Deductivas (SBDD).

En la tabla siguiente (tabla 2.1) mostramos la correspondencia entre conceptos similares que se manejan en las Bases de Datos Relacionales y la Programación Lógica. Esto con la finalidad de comprender las ventajas que presenta la programación lógica.

Tabla 2.1 Correspondencia entre conceptos similares en la programación lógica con las Bases de Datos Relacionales

BASES DE DATOS	*PROGRAMACIÓN LÓGICA*
Relación	**Predicado**
Atributo	**Argumento de predicado**
Tupla	**Hecho**
Vista	**Regla**
Pregunta	**Meta**
Coacción	**Meta (regresando un valor de verdad**

De la correspondencia mostrada en la tabla 2.1 podemos observar que Prolog opera sobre una base de hechos, lo cual, como vimos en el capítulo anterior, puede representarse empleando el lenguaje de la lógica, proporcionando un mayor número de beneficios (como los mostrados en la sección 1.2.1). Algunas de las características que hacían a Prolog poderoso en la década de los setentas, hoy en día son inconvenientes, esto es debido a el desarrollo que se ha venido dando en los últimos años, es decir, la extensión al manejo de conjuntos, la no dependencia del orden de las reglas, etc.

2.1.1 Justificación de interés en la extensión.

Los elementos que presentamos en seguida nos sirven para justificar la extensión de los lenguajes de BDD tales como Prolog.

1) **Procesamiento de un hecho a la vez**. Mientras que esperaríamos que el resultado de las consultas sobre la bases de datos sea un conjunto de hechos. Prolog regresa un solo hecho a la vez. La incorporación de conjuntos de manera práctica es implementado en LDL, aunque estos son manejados de forma indirecta, lo que los hace poco atractivos para el "usuario" ya que les resta claridad, expresividad y una forma sencilla en su uso. Por otro lado, la implementación de conjuntos en SuRE es de manera natural y directa, al igual que como se utilizan en las matemáticas haciéndolo un lenguaje mas poderoso.

2) **Procedural y Sensibilidad en el orden**. El procesamiento en Prolog es afectado por el orden de las reglas o hechos en la base de datos, y por el orden de los predicados en la cuerpo de las reglas. Esta característica es importante, pues el usuario además de plantear la solución a su problema, no debe descuidar el orden en que declara sus hechos y reglas, ya que de no hacerlo provoca que su aplicación no se ejecute correctamente. Por tal motivo, en lenguajes como CORAL, LDL1, SuRE, etc. consideran la solución a este problema dentro de su arquitectura. En el siguiente apartado (2.1.2) se presenta un ejemplo al respecto.

3) **Predicados especiales**. Existen predicados especiales de Prolog que controlan la ejecución del programa (por ejemplo, para entrada y salida de datos, rastreo, y afectación

del backtracking). En las implementaciones de los nuevos sistemas como los mencionados arriba, contemplan la no existencia de predicados especiales, ya que éstos rompen con la semántica declarativa en programación lógica.

Algunos otros elementos de Prolog en cuanto a la semántica declarativa son:

El predicado "**setof**" recolecta todas las soluciones de una consulta y esto lo logra buscando todas las ramas de éxito en el árbol de búsqueda y guardando los éxitos, ya que al realizar el backtracking se pierden por no existir conexión entre las ramas.

El predicado **var**(List), indica que la variable List deje de ser instanciada, en algún momento en que se efectúa la prueba del teorema. Otro componente es el predicado **cut** que indica que las elecciones sean ignoradas. Ambos predicados controlan la información de como debe llevarse a cabo la consulta o prueba del teorema.

Por lo que respecta a la semántica operacional, para poder decidir cual cláusula se aplica y ver que la conclusión sea válida se utiliza el **principio de resolución**, que trata de relacionar de alguna manera dos cláusulas y generar una nueva cláusula que sea la consecuencia de ellas. Esto se agrava cuando las cláusulas contienen variables, ya que al tratar de relacionar las cláusulas se tienen que ***"unificar"*** de acuerdo a su estructura, para poder deducir que dos cláusulas son idénticas.

Estas razones son las que motivaron la búsqueda de alternativas a Prolog como un lenguaje de BDD y programación lógica; una primera alternativa lo fue *Datalog*, esto a principios de los ochentas. Surge de la necesidad inmediata de probar los múltiples beneficios que brinda la extensión a las cláusulas de Horn.

2.1.2 Datalog

Datalog es un lenguaje diseñado para consultar BDD. Este es, **no procedural, orientado a conjuntos, sin sensibilidad al orden, sin predicados especiales** y **sin**

símbolos de función. Sintácticamente, Datalog es muy similar a Prolog puro, aunque con mejoras substanciales que dieron origen a los avances que hoy en día se tienen, como una muestra de estos beneficios presentamos el siguiente ejemplo.

Ejemplo 2.1 En seguida se presentan dos versiones de las reglas que determinan los ancestros de X con la finalidad de mostrar la sensibilidad al orden en Prolog..

a) **ancestor1**

ancestor(X, Y) :-parent(X, Y).

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

b) **ancestor2**

ancestor(X,Y) :- ancestor(Z,Y), parent(X, Z).

ancestor(X, Y) :- parent(X, Y)

Meta: ? = ancestor(X, Y).

Ambos programas, ancestor1 y ancestor2 son sintácticamente correctos en Prolog y Datalog. Como Datalog no es sensitivo al orden de las reglas, produce la respuesta correcta esperada (el conjunto de todos los pares ancestor-decendent). Mientras que en Prolog, la versión de ancestor1 es la única que da resultados correctos, ya que en la versión de ancestor2 Prolog entra en un loop infinito.

Datalog es el inicio de nuevos cambios en el desarrollo de lenguajes que pretendan dar al usuario una herramienta más poderosa. El procedimiento que da paso a Datalog se describe en seguida (figura 2.2).

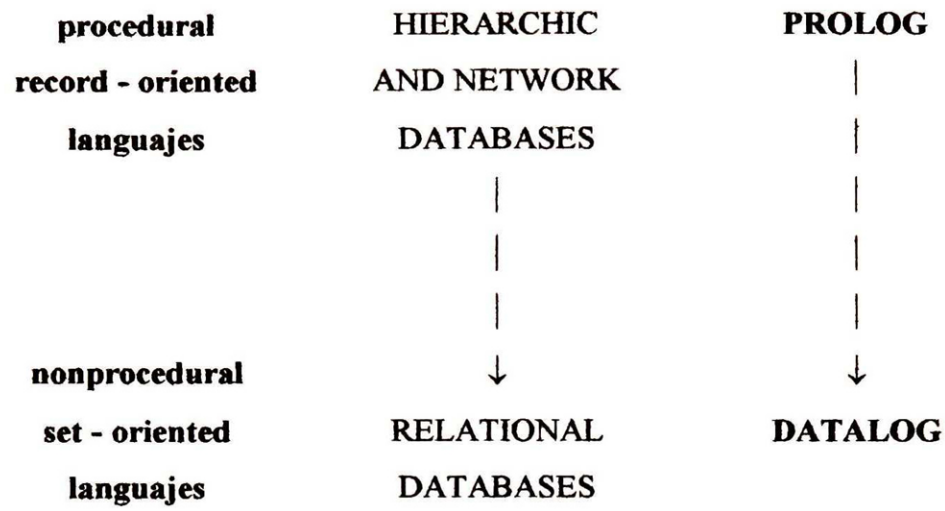

Figura 2.2 Datalog como una extensión de Prolog

Como podemos observar en la figura anterior, un paso importante cuando surge Datalog es el referente a contar con lenguajes no procedurales y que permitan nuevas estructuras de datos como los conjuntos, dando con esto mayor poder a la manipulación de datos, el que se muestra en ejemplos posteriores.

Por otro lado, algunas de las características contenidas en Datalog limitan su alcance como un lenguaje de propósito general, es decir Datalog es considerado una buena abstracción, debido principalmente a que en éste se propone la inclusión de nuevas estructuras de datos como los conjuntos, además de que ilustra el uso de la programación lógica como un lenguaje poderoso para el manejo de BDD. En seguida (figura 2.3), resumimos las correspondencias existentes entre Datalog y Prolog.

PROLOG	*DATALOG*
Depth-First search	(usualmente) Breadth-First search
un hecho a la vez	orientado a conjuntos
Sensitivo al orden	No sensitivo al orden
Predicados especiales	No predicados especiales
Símbolos de función	No símbolos de función

Figura 2.3 Comparación de Prolog y Datalog

Consideramos que Datalog fue un buen esfuerzo al proponer un cambio en la implementación de lenguajes lógicos, dando paso a nuevos desarrollos motivados por los resultados obtenidos. Como eco a estas nuevas alternativas surgen sistemas como LDL, CORAL, SuRE y otros, que son ejemplo del avance que las extensiones en sistemas de BDD se han dado, mejorando el poder expresivo y la claridad de éstos..

Debido a las necesidades que surgen en las aplicaciones de las bases de datos deductivas es que se requiere extender los lenguajes, una de las estructuras que en nuestro caso particular elegimos lo son los conjuntos, debido a que éstos nos brindan para algunos problemas una mejor manipulación y solución de los problemas, haciendo transparente y natural su resolución.

2.2 CASOS DE INTERÉS

En esta sección presentamos algunos ejemplos que motivan la extensión de los lenguajes de BDD. Iniciamos esta presentación exponiendo dos primeros casos, uno llamado los controles de una compañia y una versión de la distancia más corta.

Ejemplo 2.2 Los controles de una compañia.

El problema consiste en saber si la compañia X controla a Y, sí y solo si la suma promedio de las acciones de Y y X son arriba del 50%. Un valor de verdad indica que esto es cierto y en caso contrario nos da un valor falso.

Así cada cláusula nos indica lo siguiente:

controla (X, Y) ≥ mayor (suma (propiedades (X, Y)), 50)

propiedades (X, Y) ≥ {N} :- acciones(X, Y, N).

propiedades (X, Y) ≥ {N} :- acciones(Z, Y, N), controla(X, Z) = cierto.

La función controla(X, Y), retorna el valor de cierto si la compañia X controla a Y y falso en caso contrario. Dependiendo del valor de "suma" y "mayor".

La relación acciones(X, Y, N) significa que la compañia X tiene un N por ciento de las acciones de la compañia Y.

La recursividad de "propiedades" existe de acuerdo a que la compañia X puede tener directamente acciones en la compañia Y y viceversa. Además, Y es propiedad de X si las acciones de Y son a través de un intermediario Z, que son controladas por X.

Aquí se ilustra la recursión sobre agregación, ya que para que la compañia X controle a Y, se necesita que la suma de X en Y de todas las compañias Z, sean controladas por X y excedan del 50%.

Como se puede observar el uso de conjuntos simplifica y da claridad a la solución de problemas similares. Además, junto con el uso de conjuntos se relaciona la agregación, que es otra característica importante para muchos de los problemas tratados en los SBDD.

En seguida pasaremos a analizar algunos otros casos que también, al igual que los anteriores han servido para motivar la extensión de los lenguajes lógicos para el manejo de bases de datos deductivas.

Ejemplo 2.3. La distancia más corta.

distancia_corta(X, Y) ≤ C :- edge (X, Y, C).
distancia_corta(X, Y) ≤ C + distancia_corta(Z, Y) :- edge (X, Z, C).

la relación edge(X, Y, C) significa que hay un arco directo de X a Y con distancia C (no negativo). El operador + es monotónico, y el programa esta bien definido.

En este ejemplo podemos observar que el manejo de agregación también es una característica importante. Y por lo tanto, se debe contemplar en el diseño de los lenguajes

enfocados al manejo de SBDD, ya que en muchos problemas similares a este que nos ayudan a esclarecer los problemas que se presentan.

2.3 CONJUNTOS FINITOS

Otra característica importante que debe considerarse es el manejo de conjuntos o multi conjuntos como una operación natural y clara dentro de todo lenguaje. Ambos, como una extensión a los lenguajes para la manipulación de bases de datos deductivas o programación lógica, esto los hace poderosos y expresivos. Al respecto, pudieran existir diversas propuestas; las cuales pretenden dar al usuario la mejor forma para representar y manipular este tipo de estructuras de datos. En los últimos años se ha tratado de definir una semántica para que las bases de datos deductivas y programación lógica usen dichas estructuras, en seguida se abordan algunos conceptos al respecto, en los cuales se muestra como se expresa la manipulación de conjuntos en diversos lenguajes.

Set grouping y el mecanismo Collect-all.

Como se ha descrito anteriormente, implantar en los diferentes sistemas de BDD el manejo de conjuntos es debido a que éstos han agregado la noción de **collect-all**. En seguida, veremos brevemente como estas aproximaciones a collect-all han sido incorporadas a los lenguajes COL, LDL y SuRE.

COL	LDL	SuRE
f (a) ∋ b	f_1 (a, b)	f (a) ⊇ {b}
f (a) ∋ c	f_1 (a, c)	f (a) ⊇ {c}
	f (a, < X >) :- f_1 (a, X)	

Sea P el siguiente programa que maneja conjuntos, por medio del mecanismo collect-all:

f (a, {X}) :- f1 (a, X)

En LDL el collect-all es llamado "set-grouping", en todos los casos el resultado que se obtiene es: f (a) = {b, c} y solo en LDL es f (a, {b, c}).

La forma semántica declarativa del set y collect-all es añadir un constructor set a la lógica a nivel sintáctico, pero con una justificación de que se comporta como un conjunto finito [Jan94].

Se ve que el concepto de agregación es manejado a través de conjuntos. Donde agregación es una función parcial que mapea a un conjunto de valores, es decir, el máximo o el mínimo del conjunto, la cardinalidad del conjunto, la suma de los miembros del conjunto, etc. En [JOM95] se proporciona una forma elegante de incorporar agregación y su semántica correspondiente a la generalización para collect-all.

El problema para manejar esta semántica ocurre al definir cláusulas recursivas, ya que se espera que tenga varios modelos y un criterio para seleccionar el modelo "prometido" ("intended"[1]). Una noción para seleccionarlo es por medio de los modelos mínimos estudiados en [OJ93].

No hay un modelo como se definió en el capítulo anterior aceptable para la semántica de collect-all basada en consecuencia lógica [BN87, BRSS92], sólo es a través del modelo M(T) definido en la sección 1.2.5, donde se espera que un programa tenga varios modelos y se proporciona un criterio para obtener un modelo "intended", el cual se explica con detalle a través de ejemplos en el siguiente capítulo. Esta idea se originó por la introducción de estratificación [ABW88]. Veamos un ejemplo

Ejemplo 2.3

para $f \supseteq \{1\}$ algunos de los modelos son:

$M_1 = \{ f = \{1\}, f \supseteq \varnothing, f \supseteq \{1\} \}$

$M_2 = \{ f = \{1, \{1\}\}, f \supseteq \varnothing, f \supseteq \{1\}, f = \{1, \{1\}\} \}$

el modelo mínimo es el conjunto de modelos mínimos que están en M_1.

---

[1] intended model es un modelo que trata de inferir lo que el usuario quiere decir [RU93]

Set grouping vía Negación como falla.

Una idea para dar una semántica de collect-all, es trasladar un programa en SuRE a un "programa normal" en el sentido de Lloyd [Llo87], una vez hecho esto se traslada a negación como falla. El éxito de la traslación de programas SuRE a normales, es debido a que el programa trasladado es un programa normal estratificado, aunque esto no suceda en todos los casos, por ejemplo para programas tales como simple estratificación, puede que no lo sean, esto se debe a que la semántica basada en negación como falla es complicada de manejar por lo que se consideran dos alternativas.

1) Una es cambiar de la traslación a la obtención de programas estratificados con una semántica declarativa equivalente.
2) la otra es utilizar la misma traslación pero considerar estrategias mas generales para negación como falla, esto se puede con la semántica "Well founded" de simple estratificación que se apoya en modelos mínimos, y se basa en el modelo "intended".

En el siguiente capítulo se describen las características del lenguaje SuRE. En él se muestran ejemplos en los que se observa el beneficio que proporciona el que un lenguaje cuente con el manejo de conjuntos de una forma natural y transparente para el programador.

CARACTERÍSTICAS DEL LENGUAJE AMPLIADO

En el uso de la programación lógica se ha visto que en su forma más simple muestra una programación demasiado cruda, ya que en ocasiones una función debe ser convertida en una o varias relaciones dependiendo del concepto de la función. Por lo que se requiere de una semántica declarativa que mantenga su naturaleza lógica, para tener una programación mas fácil de entender y poder interpretar programas relacionales a través de conjuntos para que sean más claros en su escritura.

Debido a todo esto es que se encontró que SuRE es el lenguaje que más se acerca a lo que pretendemos, pues introduce conceptos diferentes a los que se manejan en los sistemas Prolog, ya que en el paradigma se introduce la programación de tipo subset-equational, que maneja aserciones con subconjuntos y ecuaciones, lo que evita el uso de características extra lógicas en el lenguaje de programación lógica. En la sección 3.1 se describen los puntos centrales sobre los cuales gira nuestra propuesta. Posteriormente, en la sección 3.2 se describen los dos paradigmas empleados en la extensión de lenguajes lógicos, en los cuales se explica la razón del porque el uso del "matching" (empatamiento) para algunos problemas es lo más adecuado, ya que simplifica la solución de éstos. Otra característica importante del lenguaje SuRE se refiere al paradigma subconjunto-relacional, que motivó su estudio y desarrollo en este trabajo, éste es tratado en la sección 3.3. En la sección 3.4 se presenta de forma completa la propuesta de integración de subconjuntos, relaciones y ecuaciones. La cerradura transitiva, que es considerada de suma importancia se trata en la sección 3.5. Por último, en la sección 3.6 se abordan algunos ejemplos que clarifican lo relacionado con programas de orden parcial.

3.1 Características principales en las que se centra esta propuesta

Nuestra propuesta gira en torno de SuRE que es un lenguaje que cuenta con una semántica declarativa que mantiene su naturaleza lógica, para que se tenga una programación sencilla y fácil de entender, además de poder interpretar programas relacionales a través de conjuntos que faciliten su comprensión. Así mismo, se pretende no

caer en implementaciones no correctas e incompletas para obtener una respuesta satisfactoria al realizar la semántica operacional. SuRE se compone de dos paradigmas

Subset-equational y Subset-relational

3.2 Programación Subconjunto-Ecuacional

Un programa en este paradigma puede tomar alguna de las dos formas siguientes:

$$f\,(terms) = expresión$$
$$f\,(terms) \supseteq expresión$$

donde *terms* corresponde a los datos del lenguaje, que son construidos a partir de átomos, variables, y constructores, mientras que las expresiones contienen adicionalmente funciones definidas por el usuario. Adicional a los términos de primer orden de Prolog, el lenguaje también provee términos de conjuntos y "matching" de conjuntos; aumentando el alcance del lenguaje y su expresividad, permitiendo declaraciones de subconjuntos con una capacidad "collect-all".

En este paradigma se estudian los subset-assertions y las equational-assertion, las cuales son tratadas como macros para subset-assertions, que abarcan programas de tipo jerárquico, simple subset-equational, estratificación simple subset-equational y estratificación subset-monotonic.

Los programas de tipo simple subset-equational, no manejan recursión alguna. Mientras los programas de estratificación simple manejan recursión en términos de otra función a un mismo nivel. Sin embargo los programas de tipo estratificación subset-monotonic manejan recursión de función en términos de otra función recursiva al mismo nivel pero de una función subset-monotonic, donde ya se habla de un orden con respecto a otra función.

3.3 Programación Subconjunto-Relacional

Aquí al igual que en el punto anterior se maneja lo mismo que en subset-equational sólo que le incorpora relaciones y negación. Además, existe un último nivel en el que se considera la agregación dando origen al lenguaje subset-logic SuRE.

Para no abocarnos a la consecuencia lógica, se optó por el modelo canónico donde el mecanismo de inferencia se le confía al modelo "prometido" (o intended) del programa. Antes de ver algunos ejemplos que muestran el modelo "prometido" damos algunos conceptos de interés.

Subset-assertions es el super conjunto aserción obtenido de **f(t)** $\supseteq$ **e**, donde **f** es la función definida por el usuario, **t** representa una tupla de términos y **e** es una expresión, que está compuesta de funciones definidas por el usuario. Este super conjunto es lo que resulta de todas las posibles instancias o aserciones, donde cada aserción es obtenida del valor de la función **f** aplicada a sus términos **t** que satisface a la expresión **e**. Si existen múltiples subset-assertions para f(t), entonces la operación collect-all colecta todos los elementos para obtener el super conjunto resultado.

Equational-assertions no tienen un significado especial, debido a que éstos sólo son trasladados a subset-assertion, con el único propósito de tener una mejor legibilidad en los programas. Por otro lado, los Relational-assertions p(t) son interpretados usualmente excepto que actúan sobre universos más complejos, ya que utilizan a los constructores de listas y conjuntos. Para la representación de conjuntos se utilizan los constructores {x\t} y $\varnothing$. Los conjuntos {__} son normalizados porque no contienen elementos repetidos. La manipulación que se hace de los conjuntos es a través de "matchings"

3.4 "Matching" de Conjuntos.

Porque los argumentos de las funciones son términos base, la aplicación de funciones requiere de un matching más que de un algoritmo de unificación. El algoritmo de matching es una forma sencilla y eficiente de resolver conjuntos, pues nos reduce la complejidad de

sus operaciones así como también se mejora su ejecución. La notación {x | S} se refiere a un conjunto en el cual x es un elemento y S es el resto del conjunto. Así, la forma de comparar conjuntos se reduce a el matching del patrón {x | S} si es tratado como la unión de {x} ∪ S. Donde solo las propiedades conmutativa y asociativa de la unión (∪) son usadas en el matching.

Por tanto, el termino { x | t } coincide con el conjunto S tal que x ∈ S y t = S-{x}, es decir, el conjunto S sin el elemento x. Así tenemos que, el conjunto representado por { _ | _} no contiene elementos duplicados, además este constructor cuenta con la siguiente propiedad

$$\{t1 \mid \{t2 \mid s\}\} = \{t2 \mid \{t1 \mid s\}\}$$

reflejando el hecho de que el orden en los conjuntos es irrelevante, pues como se puede observar podemos tener primero el elemento t1 y después el elemento t2 o viceversa es decir, primero al elemento t2 y luego el elemento t1.

Si hacemos un matching del conjunto {X\T} al conjunto {a, b, c}, se obtienen tres diferentes substituciones.

{X ← a, T ← {b, c}}, {X ← b, T ← {a, c}}, y {X ← c, T ← {a, b}}.

las cuales se descomponen recursivamente en pequeños conjuntos obteniendo un nùmero finito de "matchings" para el conjunto {X\T}.

Veamos los siguientes ejemplos que nos muestran el uso del matching entre conjuntos.

Ejemplo 3.1 Sea el siguiente subset-assertion del programa intersección de dos conjuntos.

$$\text{intersect}(\{X \mid \_\}, \{X \mid \_\}) \supseteq \{X\}$$

cuando los patrones de "conjunto" ocurren en el lado izquierdo de una subset-assertion, todos los matchings a través de estos patrones son usados en la instanciación correspondiente al lado derecho de la expresión y cada conjunto resultante es tomado para formar el super conjunto, que es la unión de todos los conjuntos resultado, es decir, el super conjunto de las aserciones. Este mecanismo de collect-all detecta cuando no hay matchings y

entonces obtiene el conjunto ∅, que es el mecanismo de vacío como falla (emptiness as failure).

Hagamos la siguiente consulta para el ejemplo 3.1 anterior

? intersect ({a,b,c}, {c,d,a})

y nos da como respuesta {a,c}

Ahora si la consulta es ? intersect ({a}, {b})

la respuesta es el conjunto vacío ∅.

Ejemplo 3.2 La unión de dos conjuntos la podemos definir como

unión (X1, X2) ⊇ X1

unión (X1, X2) ⊇ X2

esto es debido al mecanismo de collect-all, que nos permite incluir la unión de dos conjuntos

Ejemplo 3.3. El producto cruz lo podemos definir como

crossproduct ({X | _ }, {Y | _ }) ⊇ { [X | Y]}

Ejemplo 3.4 Podemos definir las permutaciones posibles de los elementos de un conjunto de la forma siguiente.

perms (ϕ) ⊇ { [ ] }

perms ({ X | T }) ⊇ distr (X, perms (T))

distr (X, { L \ _ }) ⊇ { [X | L] }

Frecuentemente las definiciones pueden ser declaradas de una manera compacta y no recursiva usando declaraciones subconjunto y términos subconjunto, porque muchas de las operaciones sobre conjuntos son realizadas en el proceso de matching. En las funciones crossproduct, intersect y distr, no hay necesidad de indicar explícitamente el caso cuando sus argumentos son en particular el conjunto vacío, ya que el resultado es el conjunto vacío.

Como se puede observar en los ejemplos anteriores, la definición de intersección de dos conjuntos (ejemplo 3.2 intersect) en SuRE es directa y no se requiere de una definición recursiva, la que sería necesaria si el lenguaje no contara con la extensión a conjuntos.

Con lo que respecta a la definición de permutaciones (ejemplo 3.5 perms), este es un ejemplo de lo que pueden ser las cláusulas de subconjuntos recursivos, estas son significativas y naturales. Una explicación detallada de este ejemplo se describe en [Jay92]..

La definición de perms muestra el uso de expresiones anidadas en el cuerpo de una cláusula subconjunto. Básicamente estas expresiones son ejecutadas en orden del mas interno y el que esté más a la izquierda, y la estrategia de default es calcular el conjunto resultante mediante una llamada interna antes de ejecutar una llamada externa.

Todos los ejemplos anteriores tienen un significado "prometido" ("intended") y son consistentes de acuerdo a la programación lógica. La definición de la cerradura transitiva es un interesante uso de las subset-assertions

3.5 Programas estratificados Subconjunto-ecuacional.

Con la finalidad de mostrar como el paradigma de programación subconjunto-ecuacional puede ser fácilmente extendido para soportar incluso funciones "closure", es decir, funciones que se definan sobre conjuntos con cerradura transitiva emplearemos algunos ejemplos. En general, diremos que las funciones "closure" son usadas en muchas áreas, especialmente en compiladores para el análisis del flujo de datos. En nuestro trabajo en particular el tratamiento de estas funciones es mediante tablas de memorización, Estas sirven para dar una semántica declarativa simple que corresponda a la semántica operacional correcta Debido a esto, introducimos la clase de programas estratificados subconjunto ecuacional. Un ejemplo en que se ilustra lo anterior sobre funciones closure es el mostrado en el ejemplo 3.3 en el que se quiere hallar el conjunto de nodos que pueden ser alcanzados en un grafo, representado como un conjunto de pares ordenados, iniciando desde un nodo dado.

Ejemplo 3.5 El siguiente ejemplo corresponde a los programas de simple estratificación subset-equational. El problema consiste en determinar que puntos son alcanzables a partir de un punto dado.

reach ({X \ _ }) ⊇ { X }

reach ({X \ _ }) ⊇ reach (edge (X))

edge (1) ⊇ {2}

edge (2) ⊇ {1}

en la semántica clásica se agregaría

reach ({1}) ⊇ {1}

pero esto no es una consecuencia lógica que reach({1}) tenga como respuesta {1, 2, {1} }, es decir, {1,2}, pero si tiene un significado "prometido". El problema en la lógica clásica es la inhabilidad de la definición de cerradura transitiva "reach".

Veamos ahora algunos ejemplos en los que existen subset-assertions condicionales

reach (X) ⊇ { X }

reach (X) ⊇ reach (Y) :- edge (X, Y)

la condición es indicada en la relación :- edge (X, Y), que nos indica que reach(Y) es verdadera sí y sólo si ocurre el "edge" de X a Y.

Veamos un ejemplo más en el que se muestra el uso de la tabla de memorización para detectar los loops infinitos y no caer en ellos.

Ejemplo 3.6

Regla #1	$g(X) \supseteq \{10\}$
Regla #2	$g(X) \supseteq h(X)$
Regla #3	$h(X) \supseteq \{20\}$
Regla #4	$h(X) \supseteq p(g(X))$
Regla #5	$p(\{X \setminus \_\}) \supseteq \{X, 30\}$

En este ejemplo, la derivación de la consulta g(100) se muestra en la figura 3.1

Secuencia de Metas	Sustitución	Tabla de Memorización	# Deriv.
g (100) = Ans		ϕ	
h (100) = S1	Ans ← {10} ∪ S1	{g (100) = {10} ∪ S1}	1
g(100)=T1, {p(T1)=S2}	S1 ← {20} ∪ S2	{g (100) = {10, 20}∪ S2,ciclo	2
		h (100) = {20}∪ S2}	
{p ({10, 20}∪ S2) = S2}	T1 ← {10, 20}∪ S2	{g (100) = {10, 20}∪ S2,ciclo	3
		h (100) = {20}∪ S2}	
[]	S2 ← {10, 20, 30}	{g (100) = {10, 20, 30}, h (100) = {10, 20, 30}}	4

Figura 3.1 Derivación de g(100).

En esta tabla se nota como el último paso de la derivación tiene un aspecto funcional [{p({10, 20} ∪ S2) = S2}con la respuesta mínima calculada S2 = {10, 20, 30}. Esta es determinada como sigue. Previamente asumimos que S2 = ϕ. Entonces, en la primera derivación tenemos que h(100) = S1 debido a que g(X) ⊇ h(X) significa que g(X) ⊇ S1 :- h(X) = S1 por lo que la respuesta parcial es: Ans ← {10} ∪ S1. En seguida por la regla #4 tenemos que h (X) ⊇ Z :- g(X) = T1, p(T1) = S2 y por la regla #3 obtenemos que S1 ← {20} ∪ S2, luego de aquí si sustituimos el valor de S1 en la respuesta parcial anterior obtenemos que Ans ← {10} ∪ {{20} ∪ S2} [o bien, Ans ← {10, 20} ∪ S2], cuyos

resultados parciales son memorizados en la tabla. En la derivación #3 tenemos que h(X, Z) :- g(X,Y), p(Y,Z) por la regla #5, por lo que T1 ← {10, 20}∪ S2. En la última derivación obtenemos que S2 ← {10, 20, 30}con este resultado determinamos el ciclo que se genera en las memorizaciones de las derivaciones #2 y #3 lo que nos lleva a la determinación de la respuesta final. Como se puede notar el uso de las tablas de memorización es una herramienta adecuada en la detección de ciclos que pueden generarse como consecuencia de derivaciones intermedias en la obtención de una respuesta.

En general, los programas estratificados subconjunto-ecuacional consisten de funciones closure y funciones no-closure particionadas en varios niveles, tales que todas las funciones closure de un nivel dado sean definidas en términos de otras usando funciones monotónicas-subconjunto, pero pueden ser definidas en términos de cualquier otra función (closure o no-closure) desde un nivel mas bajo.

3.6 Programas Subconjunto-relacional

La programación Subconjunto-relacional es un paradigma de la programación con sentencias conteniendo subconjuntos y relaciones. Las formas generales de programas simples subconjunto-relacional son las siguientes.

f (terms) $\supseteq$ S

f (terms) $\supseteq$ S :- p_1 (terms), ..., p_n(terms)

p (terms)

p (terms) :- p_1 (terms), ..., p_n (terms)

Note que estas definiciones pueden contener conjuntos como parte de *terms*. El significado declarativo de una cláusula subconjunto es que, para todas sus instancias básicas, la función f sobre los términos base contiene un conjunto base S si la condición en el cuerpo, p_1 (terms), ..., p_n (terms), es cierta. Para incorporar la capacidad del collect-all, en el resultado de una consulta tal como en f (ground-terms) = term se hace coleccionando todas las soluciones de la expresión f(ground-terms) y entonces se unifica el resultado con el

término (term). Una consulta general puede tener la forma p(terms). La semántica operacional usa el matching de conjuntos con operaciones de conjuntos y, si las cláusulas contienen conjuntos, se realiza unificación de conjuntos con operaciones relacionales. Los argumentos de una función definida por operaciones de subconjuntos deben ser la base, y cada derivación debe ser terminada (además satisfecha o finitamente fallada) obteniendo un resultado para la meta ecuacional mostrada anteriormente. Mas aún, deben ser solo un número finito de derivaciones, simplemente porque todos los conjuntos en este contexto son finitos. Note que estas condiciones son análogas a las necesarias para la correctitud de la negación como falla en la programación relacional [L87]. Un par de ejemplos nos servirán para ilustrar este paradigma.

3.6.1 Setof. Los programas simples subconjunto-relacional pueden ser usados para simular la característica setof de Prolog de una manera declarativa.

Ejemplo 3.7 El apendizar dos listas puede definirse como sigue.

append ([ ], X, X)

append ([H, T], Y, {H | Z]) :- append (T, Y, Z)

la meta setof ([X | Y], append (X, Y, [1, 2, 3]), Answer) para Prolog definiendo las diferentes particiones de la lista [1, 2, 3] puede ser expresada por la siguiente declaración y consulta de subconjuntos.

parts (list) $\supseteq$ { [X | Y] } :- append (x, y, list)

? parts ([1, 2, 3]) = answer.

3.6.2 Set-terms en relaciones. El uso de conjuntos en relaciones hace posible algunas definiciones interesantes, tal es el caso de la siguiente definición de permutaciones de los elementos de un conjunto.

Ejemplo 3.8 En seguida definimos permutaciones usando conjuntos

set-to-list (ϕ, [ ])

set-to-list ({x | s}, [x | t]) :- set-to-list (s, t)

permutations (set) $\supseteq$ {list} :- set-to-list (set, list)

Como el argumento en permutaciones se asume que es la base, el primer argumento en el llamado de *set-to-list* en el cuerpo de *permutations* también será la base. Porque el matching de un conjunto de n-elementos contra la segunda declaración para set-to-list admite n diferentes matchings, cada uno es considerado separadamente reduciendo recursivamente el cuerpo set-to-list. De esta manera, todas las permutaciones a un nivel superior en el llamado a la función *permutations* será calculado. Aquí es importante recalcar que en la manipulación de conjuntos en muchas aplicaciones practicas reduce la unificación de conjuntos a el uso de un simple matching entre éstos. El uso de un matching simplifica el cálculo operacional, que es sumamente importante en el manejo de Bases de Datos Deductivas. Además de esta característica importante, la reformulación de una relación como una función evaluada sobre conjuntos, no solo especifica el modo de la información declarativamente, si no también gana flexibilidad de operación sobre el conjunto resultante ya sea por una meta de pertenencia o por una meta ecuacional.

En el contexto de las Bases de Datos Deductivas las cláusulas de subconjuntos nos permiten hacer formulaciones claras y concisas sobre problemas que involucran operaciones de agregación y recursión en consultas a Bases de Datos. Una operación de *agregación* es una función que mapea un conjunto sobre un valor, es decir, el máximo o el mínimo en el conjunto, la cardinalidad de éste, la suma de todos sus elementos, etc. Estas características son de interés en la literatura de la Bases de Datos Deductivas recientemente [JM95]. En nuestro estudio generalizado sobre las diferentes aseveraciones de la necesidad de contar con el manejo de subconjuntos, operaciones de agregación y el concepto de funciones monótonas nos lleva a la conclusión de que todas estas operaciones son expresadas naturalmente en términos de funciones más que de predicados, por esto nuestra propuesta se fundamenta en una implantación funcional sobre conjuntos. Por ejemplo, retomando nuestro caso (ejemplo 3.2) de unión anterior, la unión de dos conjuntos en nuestra propuesta puede ser definida como.

union(X1,X2) $\supseteq$ X1

$$\text{union}(X1,X2) \supseteq X2$$

en esta definición podemos observar que por la existencia del collect all, inferimos que union(X1,X2) = X1 $\cup$ X2. Además, la ejecución para hallar este resultado puede ser fácilmente realizado vía iteración sobre los dos conjuntos, lo cual refuerza nuestra propuesta de que para problemas prácticos basta con el uso del algoritmo de matching y así evitar el proceso de unificación que es más complejo.

La negación como falla puede también ser fácilmente simulada para el collect all, como se muestra en [JP89]. Además, las declaraciones que incluyen conjuntos se combinan de una manera natural con declaraciones ecuacionales y relacionales, esto hace posible contar con una base para la extensión de la programación lógica.

3.7 Integración de Subconjuntos, Relaciones y Ecuaciones

En éste punto exponemos los puntos clave que dan origen a nuestra propuesta de integrar en un lenguaje el manejo de subconjuntos, relaciones y ecuaciones de una manera fácil, clara y sencilla. Por tal motivo, iniciamos nuestra exposición mostrando lo que entendemos por una meta ecuacional la cual tiene la forma

$$f(\text{terms}) = \text{term}$$

donde f es una función que puede ser definida ya sea con aseveraciones ecuacionales o de subconjuntos, y los argumentos de f deberán estar disponibles en el momento de su llamado. Es decir, las metas ecuacionales serán usadas sólo para reducción y no como limitante. Una meta subconjunto es de la forma

$$f(\text{terms}) \supseteq \{\text{term}\}$$

donde, una vez más, f obedece a las mismas restricciones que en una meta ecuacional. Una meta subconjunto especifica la enumeración de un elemento a la ves del conjunto definido por f, y especifica una forma limitada de evaluación ociosa, con el beneficio además de que los conjuntos intermedios son evitados, esta capacidad tiene que ser implementada en programas simples subconjuntos-ecuacionales, ésta fue usada en el contexto limitado cuando

se conocía que una función se distribuye sobre la unión en una cierta posición del argumento [JP87, JN88, J90].

En seguida utilizaremos algunos ejemplos que nos sirvan para ilustrar el marco de extensión.

3.8 Cerradura Transitiva.

Se presentan dos ejemplos, con los que se propone la integración de la programación subconjunto-ecuacional estratificada y la programación subconjunto-relacional estratificada. Para esto es deseable reescribir la definición de reach de la sección 3.2 en el que el grafo es especificado por una relación edge más que un conjunto de pares ordenados. De esta forma el grafo no se incluirá en la tabla memo y así minimizaremos el costo de la búsqueda en la tabla de memorización.

Ejemplo 3.10

reach (v) $\supseteq$ adjacent (v)

reach (v) $\supseteq$ allreach (adjacent (v))

allreach ({x, | _}) $\supseteq$ reach (x)

adjacent (v) $\supseteq$ {v} :- edge (v, w)

edge (1, 2). edge (2, 3). edge (3, 4). edge (4, 1).

Las anteriores definiciones se estratifican en tres niveles, edge en el nivel 1, adjacent en el nivel 2 y reach y allreach en el nivel 3. Las funciones closure (en las que se contempla el ciclo) reach y allreach, deben ser declaradas en la línea de anotaciones para poder ser permutados. Es bien conocido que para lograr una cerradura transitiva eficiente en Prolog se requiere del uso de assert y retract [O88]. Debe ser mencionado que las expresiones anidadas en el cuerpo de una expresión subconjunto (o una ecuacional) son operadas mediante el proceso de flattening para reflejar el orden de reducción más profundo. Por ejemplo, la segunda expresión para reach es sintácticamente simple y al aplicarle el proceso de flattening obtenemos lo siguiente expresión.

reach (v) $\supseteq$ {s2} :- adjacent (v) = s1, allreach (s1) = s2.

La compilación en un programa de diagrama de flujo para la definición de reach puede ser [AU77].

out (b) $\supseteq$ diff (in (b), kill (b))

out (b) $\supseteq$ gen (b)

in (b) $\supseteq$ allout (pred (b))

allout ({p | _}) $\supseteq$ out (p)

diff (s1, s2) $\supseteq$ {x} :- member (x, s1), not member (x, s2)

member (x, {x | _})

donde kill (b), gen (b), y pred (b) son funciones predefinidas y evaluadas en conjuntos especificando la información relevante para un flujograma de un programa dado, y un block básico b. Este ejemplo muestra que las subset-assertions son capaces de formular problemas en el análisis de flujo de datos en una manera directa y natural.

3.9 Programas de Orden Parcial

En seguida se presentan dos ejemplos en los que se exponen programas de orden parcial. Las cláusulas de orden - parcial tienen la forma

$$f(terms) \geq expresión$$

$$f(terms) \leq expresión$$

donde cada variable en expresión también se presenta en f(terms). Por simplicidad de presentación en este punto de nuestro trabajo de tesis, exponemos éste concepto a través de los ejemplos siguientes.

Ejemplo 3.11 Las siguientes cláusulas determinan la distancia más corta.

La formulación del problema de la distancia más corta es es una de las ilustraciones más elegantes y suficientes de cláusulas de orden parcial.

short (X, Y) $\leq$ C :- edge (X, Y, C).

short (X, Y) ≤ C + short (Z, Y) :- edge (X, Z, C).

Esta definición es muy similar a la siguiente definición de reach

reach (X) ≥ {X}.

reach (X) ≥ reach (Y) :- edge (X, Y).

excepto que la operación de agregación aquí es ≤. La relación edge (X, Y) significa que hay un eje directo de X a Y con distancia C la cual es no negativa. La distancia de default entre dos nodos cualesquiera es max_int (el elemento máximo de los menores). El operador + es monotónico con respecto al ordenamiento numérico y, entonces el programa es bien definido. La lógica del problema de la distancia más corta es claramente especificada en el programa anterior. (note que el dominio resultado de la función short es totalmente ordenado. Este conocimiento puede ser utilizado para lograr una implementación muy eficiente para éste problema, asemejando el algoritmo de Dijkstra).

Ejemplo 3.12 Control de una compañia

El problema del control de una compañia a venido a ser el ejemplo estándar para la agregación monotónica.

controls (X, Y) ≥ gt (sum (holdings (X, Y)), 50)

holdings (X, Y) ≥ {N} :- shares (X, Y, N).

holdings (X, Y) ≥ {N} :- shares (Z, Y, N), controls (X, Z) = true.

éste ejemplo ilustra el uso de una operación de agregación inductiva llamada sum. La función controls (X, Y) regresa el valor de cierto si la compañia X controla a la compañia Y, y falso en otro caso. La relación shares (X, Y, N) significa que la compañia X tiene N% de las acciones de la compañia Y. Las posesiones cíclicas son posibles, es decir, la compañia X puede tener acciones en la compañia Y y viceversa. Aquí observamos recursión sobre agregación, es decir, una compañia X controla a una compañia Y si la suma de las acciones de X en Y junto con las acciones en Y de todas las compañias Z controladas por X exceden el 50%. En éste ejemplo talvés sería más conveniente el uso de multi conjuntos que el de conjuntos.

Ejemplo 3.13 El problema del algoritmo min-max sobre un grafo bipartido

v (X, C) :- f (X, C).

v (X, C) :- $p_{=}$ (X, C).

v (X, C) :- $q_{=}$ (X, C).

$p_{\leq}$ (X, C) :- edgea (X, Y), v (Y,C).

$q_{\geq}$ (X, C) :- edgeb (X, Y), v (Y,C).

f (e, 3).	f (f, 2).	f (k, 2).	f (m, 1).	
edgea (g, f).	edgea (g, e).	edgea (h, e).	edgea (h, i).	edgea (h, j).
edgeb (i, g).	edgeb (i, h).	edgeb (i, m).	edgeb (j, m).	edgeb (j, k).

en este ejemplo se ilustra una interacción de $\geq$ y $\leq$ tal que nuestro programa no es de costo monotónico. Si usamos la semántica bien definida considerando programas normales estratificados ésta define un modelo total donde v(h)=2 y v(i)=2. Pensamos que éste es el intended model (modelo esperado) del programa. Van Gelder comenta que la conclusión puede ser establecida por una simple semántica de negación como fallo, sin necesidad de involucrarse en una semántica bien fundada. La semántica operacional definida en [JOM95] calcula también el intended model de su programa con un procedimiento que no involucra negación como fallo.

En el capitulo siguiente presentamos nuestra propuesta, la que resultó de una investigación de análisis y estudio sobre los diversos puntos que dan una superioridad a nuestro planteamiento respecto de los diferentes sistemas de bases de datos deductivas actuales. Así mismo se presenta una implementación del trabajo desarrollado mostrando las características técnicas importantes para la futura extensión del sistema.

MODELADO DEL LENGUAJE

En este capítulo se presentan los puntos sobresalientes de nuestro trabajo utilizando para su exposición ejemplos sencillos pero útiles en la explicación de los detalles que conforman nuestra contribución a los sistemas de bases de datos deductivas con extensión a conjuntos finitos. Cabe mencionar que como parte de nuestro trabajo se realizó una selección cuidadosa y exhaustiva de los elementos que conforman nuestra propuesta, cuidando que estos enriquezcan el lenguaje para bases de datos además, lograr que este sea más expresivo y claro en la solución de problemas a través de algoritmos sencillos y claros. Así mismo, se trata la extensión necesaria de los lenguajes para el manejo de conjuntos, tratando con esto de proporcionar al programador mejores herramientas para la solución de algunos problemas en los que el uso de conjuntos es natural. el que un lenguaje incluya el manejo de conjuntos brinda operaciones poderosas y claras respecto de las operaciones de agregación

En primera instancia, en las dos primeras secciones 4.1 y 4.2 se presentan nuestras aportaciones que son necesarias en la definición del lenguaje. En primera instancia, la propuesta de las clases de cláusulas del lenguaje que modelamos para el manejo de bases de datos deductivas con extensión a conjuntos finitos, el cual surge de SuRE. Aunque nuestro trabajo se apoya en SuRE, se hizo una selección cuidadosa de los elementos que conformarían nuestro lenguaje y con los que se deseaba trabajar para alcanzar nuestros objetivos; éstos tienen como propósito fundamental eliminar las características extra lógicos (como los descritos en 1.3.1) que se introducen en algunos sistemas de bases de datos deductivos, esto es necesario para evitar que se pierda la semántica del lenguaje, como es en el caso de Prolog. Los elementos básicos de nuestro lenguaje tienen algunas características que le dan un mejor desempeño y claridad en el manejo de conjuntos, permitiéndonos de esta manera experimentar para obtener un lenguaje reducido donde se manipulen los conceptos primordiales que se reflejan en SuRE. Aquí se abarca el paradigma subset-equational para los tipos de programas simples y de estratificación simple.

Nuestra segunda propuesta necesaria es definir la gramática libre de contexto la cual es importante en la definición del lenguaje. Sobre este tema trabajamos puliendo la gramática

presentada en [JOM95] hasta alcanzar que ésta fuera determinística, lo que nos brinda algunos beneficios tales como: facilitar su evaluación e implantación.

En la siguiente sección presentamos nuestras principales contribuciones resultado de nuestro trabajo de investigación. En la sección 4.3 definimos un algoritmo para la transformación de programas normales funcionales a su forma relacional, haciendo uso de un ***algoritmo "flattening" recursivo*** presentándolo a través de ejemplos en los que se manifiesten las características sobresalientes alcanzadas. El algoritmo que se diseño también esta pensado en aprovechar el que se cuenta con una gramática determinística y que por tal motivo las estructuras de datos empleadas para su implantación se definieron de manera sencilla y clara para nuestro caso de estudio.

Por último, nuestra última contribución obtenida como resultado de nuestro trabajo, es la transcripción de programas SuRE estratificados a programas estratificados normales, apoyados en la negación como falla. Esta transformación es de vital importancia para la solución de algunos de los problemas que se presentan en el área de las bases de datos deductivas, ya que el programa que se obtiene de la transcripción tiene una semántica "bien definida" y estudiada lo que hace que nuestro lenguaje se enriquezca. Se presenta la semántica requerida para los tipos de programas simples subset-equational y de estratificación simple apoyándonos en todo momento de casos que ejemplifiquen nuestra propuesta y contribución lograda. En algunos otros lenguajes la solución que se presenta para estos casos es resuelto con una mayor complejidad y en otros son considerados como ejemplos sin solución [RPDS88], esto es relevante en nuestra propuesta. Por fines prácticos algunos de nuestros ejemplos obtenidos se han probado en CORAL, debido a que este lenguaje cuenta con estratificación, lo que garantiza que los resultados obtenidos son correctos.

4.1 CLASES DE CLÁUSULAS DEL LENGUAJE MODELADO

En nuestro trabajo de investigación nos enfocamos al problema de programación lógica y funcional en el cual los conjuntos son nuestros elementos de interés y de primera clase, ya

que estos no son simulados por listas sino tratados como tales. Los beneficios que obtenemos al concentrarnos en la programación lógica y funcional son:

- Las operaciones sobre conjuntos pueden realizarse de manera no recursiva, gracias a la iteración implícita que sobre los conjuntos nos provee el "matching". No se descarta la posibilidad de un algoritmo recursivo aún cuando éste sea más complejo.
- La búsqueda no determinística puede ser especificada sin el uso de cortes ("cuts").
- Es posible una ejecución eficiente con ecuaciones.
- Es posible la generalización de Agregación de manera inmediata.

Nota: Estos beneficios se sustentan en [JOM95].

Es importante hacer notar que las operaciones de agregación y set-grouping son poderosas y han sido muy utilizadas en las bases de datos deductivas [RU93]. Además, uno de nuestros principales propósitos en este trabajo es el de proveer las bases para la programación sobre conjuntos.

Nuestra propuesta en el presente trabajo se basa en el lenguaje SuRE, el cual esta compuesto de los paradigmas siguientes:

1.- Subset - equational.

2.- Subset - relational.

de los cuales nos abocamos sólo a el primero de éstos.

Subset - equational

En este paradigma se introduce un nuevo concepto que es **subset-assertions**, el que se describe como:

$$f(\text{términos}) \supseteq \text{expresiones}.$$

El resultado de este concepto es obtenido cuando el conjunto de patrones ocurren en el lado izquierdo de la subset-assertions, y todos los "matchings" que están en los patrones son utilizados en la instancia que corresponde al lado derecho de la expresión, la unión de todos estos subconjuntos forma el resultado de la subset-assertions. Donde subset-assertions es el subconjunto de aserciones que satisfacen la expresión por medio del valor que resulta de la

función f, donde sus términos se encuentran instanciados y aparecen en el conjunto de instancias de expresiones.

Para la obtención del super conjunto aserción, empleamos el mecanismo de **collect-all** que consiste en la unión de los subconjuntos aserción que se van proporcionando, de acuerdo a la evaluación de f que satisfacen a la expresión.

para ilustrar esto veamos como es un query en el lenguaje:

? expresión.

donde la expresión es una instancia.

La obtención de cada instancia de la igualdad, en el paradigma subset-equational se realiza utilizando reducción en profundidad y un "matching" asociativo conmutativo [JP87]. Este tipo de "matching" se debe a que cada aserción se va agregando al subconjunto de aserciones por medio del constructor unión para conjuntos. Al mecanismo utilizado para obtener la aserción resultado de subset-assertions se le llama collect-all, que consiste en la unión de todos los subconjuntos que se obtienen con el constructor unión de cada aserción.

La igualdad se obtiene del query, ya que su significado es el término instanciado que hace verdadera la expresión. Esto origina que se complete el programa, es decir, cada término que satisface la igualdad se va agregando al subconjunto de aserciones como se describió anteriormente.

En seguida veremos un ejemplo en que se ilustran las partes mas sobresaliente del paradigma subset-equational que son completar un programa y las restricciones del "matching".

Completar un programa.

Se completa el programa para derivar la aserción de igualdad del subset-assertions. En seguida se mencionan las hipótesis que se originaron al completar el programa y se reflejan los siguientes resultados.

i) Hipótesis Collect-all. Si un valor de la expresión e es tal que e $\supseteq$ s1, ..., e $\supseteq$ sn, y determina que no existe otro subconjunto para e que complete el programa, entonces la hipótesis de collect-all infiere e = $U_{i=1,n}s_i$.Es decir es el único resultado de la subset-assertions.

ii) Hipótesis Emptiness-as-failure. Si se obtiene el siguiente resultado es que pasaron dos cosas:

ii.1) El valor del conjunto expresión fue vacío, y entonces proporciona al conjunto $\varnothing$ como respuesta.

ii.2) No hubo un "matching" para la función en el lado izquierdo de la aserción, originando el conjunto $\varnothing$.

Ejemplo 4.1 Sea el programa P con el que se determina quien es el padre de X.

padre (bob) = mark
padre (ann) = mark
padre (mark) = joe
madre (bob) = mary
madre (ann) = mary
madre (mark) = jane

relación (X) $\supseteq$ { padre (X) }
relación (X) $\supseteq$ { madre (X) }

la aserción para relación (X) = { padre (X) } $\cup$ { madre (X) }

Sea el query que cubre la hipótesis ii.1)

? relación (mary).

relación(mary) = { padre(mary) } $\cup$ { madre(mary) } = $\varnothing \cup \varnothing$

Sea el query que satisface que P sea completo.

? relación(bob)

relación(bob) = { padre(bob) } ∪ { madre(bob) }

= { mark } ∪ { mary }

= { mark, mary }

Restricciones para el "matching".

La otra parte sobresaliente del paradigma subset-equational lo son las restricciones para el "matching". El "matching" consiste en dos términos t1 (instanciado o no instanciado) y t2 (instanciado), donde existe una substitución θ, tal que:

$$t1\ \theta = t2$$

La restricción del "matching" es la no utilizar la propiedad de idempotent, la cual considera la identidad de dos conjuntos cuando existen elementos repetidos en el conjunto, esto es, el conjunto {a, a} es idéntico al conjunto {a}, ya que si existen elementos repetidos al efectuar el "matching" se puede tener un ciclo infinito. Veamos un ejemplo.

Ejemplo 4.2

se requiere del "matching" { H\T } con { 1,2,3 }

{ H ←1, T ← { 1,2,3 } }

en el ejemplo anterior podemos ver que esto no es válido, ya que H contiene el 1 y T contiene también el 1, por lo tanto no se hace uso de la propiedad de idempotent, que es la restricción del "matching".

En los ejemplos anteriores se ha mostrado como se obtiene el subset-assertions, por lo que ahora veremos los dos tipos de programas que se manejan en el paradigma subset-equational.

1.- Programa con simple subset-equational. Está enfocado a la obtención de completar un programa y el mecanismo de collect-all utilizado para obtener el subset-assertions.

2.- Programas con simple estratificación subset-equational. Estos son utilizados para funciones recursivas y se basan en la estratificación de las cláusulas de Horn con negación por falla, esto con el fin de evitar problemas de recursión con negación y conjuntos con cerradura transitiva.

En seguida, en la sección 4.2 presentamos la gramática propuesta para nuestro lenguaje. Esta tiene algunas características importantes como lo es el ser general, determinística y completa.

4.2 DEFINICIÓN DE LA GRAMÁTICA LIBRE DE CONTEXTO

Nuestra segunda aportación resultado de nuestro trabajo de investigación es la propuesta de una gramática libre de contexto ***determinística*** que le da algunas características importantes que facilitan su evaluación e implantación. En [JM95] se proponen una serie de producciones que definen la gramática del lenguaje, este conjunto de producciones son no deterministicas, lo que nos hace mas compleja su evaluación, por tal motivo como parte de nuestro trabajo dichas reglas fueron trabajadas hasta obtener un conjunto de producciones deterministicas que son mostradas en seguida.

#1	término	→ átomo | variable | constante(términos) | conjunto | lista
#2	términos	→ termino | termino, términos
#3	átomo	→ dígito | true | false | " | constante | nil | phi
#4	conjunto	→ { termino \ conjunto }
#5	lista	→ [termino | lista]

Un programa está compuesto de cláusulas, donde cada cláusula tiene la forma siguiente.

directiva. cláusula

directiva → "closure none"

cláusula: constante(términos) ⊇ término |

constante(términos) = término.

El contar con una gramática libre de contexto determinística es algo que debe ser tomado en cuenta a conciencia ya que esta debe permitir al usuario una forma clara y sencilla para definir sus cláusulas y programa en general, evitando así distraerlo en la solución de su problema teniendo que cuidar detalles del lenguaje mismo. Además, la evaluación Top-Down de esta gramática no tiene el problema de caer en ciclos infinitos, su evaluación es más eficiente y simple.

Veamos algunos ejemplos en los que se muestran programas de tipo simple subset-equational y programas de tipo simple estratificación subset-equational.

Programas del tipo simple subset-equational.

Para este tipo de programas puede que exista o no recursión, si existe tiene que ser a través de la misma función, pero no en términos de otras. Veamos los siguientes ejemplos:

Ejemplo 4.3 Sea el programa que encuentra los ancestros del tipo simple subset-equational.

padre (bob) = mark

padre (ann) = mark

padre (mark) = joe

padre (bob) = mary

padre (ann) = mary

padre (mark) = jane

relación (X) ⊇ { padre (X) }

relación (X) ⊇ { madre (X) }

anc (X) $\supseteq$ relación (X)

allanc (X) $\supseteq$ X

allanc ({ X\T }) $\supseteq$ anc (X)

Ejemplo 4.4 Programa *append* puede ser expresada de la siguiente forma:

append ([ ] , X) = X.

append ([H | T] , Y) = [H | append (T, Y)]

Como se puede observar en los ejemplos anteriores éstos cumplen caen en el tipo de programas subset-equational.

Programas del tipo simple estratificación subset-equational

La semántica de este tipo de programas indica que existe recursión sobre la misma o en términos de otra función, cada cláusula que compone el programa, debe de tratar a las funciones recursivas por niveles, es decir, deben estar estratificadas. Este tipo de semántica refina el programa por niveles, donde cada nivel contiene cláusulas que pueden ser recursivas o no. Estas tienen dos formas.

a) Recursión sin negación

b) Negación sin recursión

Esto indica que cada nivel debe aparecer sin recursión a través de la negación, para que se encuentre estratificado [Das92].

Ejemplo 4.5 Sea P el siguiente programa estratificado que determina si X està casado con Y.

p1:

casado (joe, mary).

casado (juan, paty).

hermano (joe, juan).

hermano (mary, paty).

esposo (X, Y) ← esposo (Y, X)

relación (X, Y) ← hermano (X, Y)

relación (X, Y) ← relación (Y, X)

relación (X, Y) ← relación (X, Z) ^ relación (Z, Y)

p2:

casado (X, Y) ← esposo (X, Z) ^ relación (Z, Y) ^ ¬ relación (X, Y)

casado (X, Y) ← casado (Y, X)

Si P se divide en los niveles p1 y p2 como se describen arriba, entonces cada cláusula "relación" que aparece en p1 y p2 son únicas y además la cláusula "casado" no aparece en la recursión "relación" de p1. Por lo tanto se encuentra estratificado.

Ejemplo 4.6 Sea P un programa que halla el predecesor de un número y se encuentra no estratificado.

par (0)

par (X) ← predecesor (X, Y) ¬ par (Y)

predecesor (1, 0).

predecesor (2, 1)

predecesor (3, 2)

el programa no se encuentra estratificado ya que existe la recursión de "par" a través de la negación.

Este tipo de semántica sería una alternativa para manejar cláusulas negativas, dichas cláusulas no dependerían de la consecuencia lógica, sino de los modelos mínimos, ya que si un programa se encuentra estratificado se asegura que existe un modelo mínimo.

Una vez ejemplificado lo que es un programa estratificado pasemos a ver un ejemplo de un programa del tipo simple estratificación subset-equational, para lo cual retomamos el ejemplo de los ancestros.

Ejemplo 4.7 Programa que halla los ancestros de X.

padre (bob) = mark
padre (ann) = mark
padre (mark) = joe
madre (bob) = mary
madre (ann) = mary
madre (mark) = jane
relación (X) $\supseteq$ { padre (X) }
relación (X) $\supseteq$ { madre (X) }
anc (X) $\supseteq$ relación (X)
anc (X) $\supseteq$ anc (madre (X))
anc (X) $\supseteq$ anc (padre (X))

Aquí las cláusulas "padre", "madre" y "relación" tienen nivel 1 y anc tiene nivel 2, invocándose a si misma.

Ejemplo 4.8 Programa que encuentra la conexión directa de los nodos de un grafo.

reach (X) $\supseteq$ X
reach ({ X \ _ }) $\supseteq$ reach (edge (X))
edge (2) $\supseteq$ {1}
edge (1) $\supseteq$ {2}

la relación "reach" tiene nivel 2, mientras la relación "edge" tiene nivel 1.

4.3 ALGORITMO FLATTENING

Normalmente la integración de lenguajes funcionales y lenguajes lógicos se apoya fundamentalmente en la utilización de técnicas basadas en reducción, ya que éstas han predominado como mecanismos operacionales para intérpretes lógicos funcionales. Es por esto que en nuestro trabajo se optó por la utilización de un algoritmo "flattening" Este se caracteriza por ser utilizado en contextos donde se desea emular funciones. A diferencia de algunos métodos similares encontrados en algunos artículos [JP89], en nuestro caso se diseño completamente para lograr la re - escrituración de Programas Estratificados de Orden Parcial a Programas Normales. Este paso es relevante, considerando que se diseño completamente el procedimiento que más adelante se describe. Por tal motivo, en nuestro trabajo de investigación este algoritmo es relevante y nos permite corroborar su eficiencia en la transformación a programas normales.

La notación usada para introducir cláusulas de subconjuntos serán referidos de la forma de conjuntos. Pero, formalmente, veremos a cada cláusula escrita como una cláusula corta de una definida. Esto es hecho mediante la transformación "flattening" de todas las expresiones tales que los argumentos de todas las llamadas a funciones sean términos. Los pasos a seguir para nuestra semántica son:

1) Una ves dado el programa, a cada cláusula se le aplica el "flattening" La forma "flattening" significa que cada función que aparece en el lado derecho de la cláusula debe estar en términos de f (términos) $\supseteq$ expresión, pero expresión debe tiene que ser un termino y no otra función.

f (términos) $\supseteq$ expresión.

se transforma en:

f (términos) $\supseteq$ v:- expresión = v.

Ejemplo 4.9 Sea el siguiente programa.

$$f(\{X\}, \{Y \backslash S\}) \supseteq \{1\}$$
$$f(X, \{Y \backslash S\}) \supseteq g(h(X), k(Y,S))$$

con el "flattening" se obtiene lo siguiente:

$$f(X, \{Y \backslash S\}) \supseteq X3 \text{ :- } h(X) = X1, k(Y, S) = X2, g(X1, X2) = X3$$

Esto se expresa así para poder completar el programa y obtener el resultado de la igualdad en subset-assertions, que se encuentra en la siguiente forma.

$$f(X, \{Y \backslash S\}) = \{1\} \cup \{X3 \text{ tal que } h(X) = X1, k(Y, S) = X2, g(X1, X2) = X3\}$$

2) Se traslado cada cláusula de forma "flattening" a su forma "relacional", ya que en Prolog la forma de manejar estas cláusulas es de este estilo, veamos el siguiente ejemplo.

Ejemplo 4.10 Sea el siguiente programa, donde la función es recursiva consigo misma y encuentra los puntos a donde se dirige cada arco.

$$\text{reach}(X) \supseteq \{X\}$$
$$\text{reach}(\{ X \backslash \_ \}) \supseteq \text{reach}(\text{edge}(X))$$

$$\text{edge}(1) \supseteq \{2\}$$
$$\text{edge}(2) \supseteq \{1\}$$

sea el query: ? reach ({1}) = Resp

su forma **"flattening"** es la siguiente:

```
reach (X) contains {X}
reach ( { X \ _ } ) contains T1 :- s1 = edge (X), T1 = reach (S1)

edge (1) contains {2}
edge (2) contains {1}
```

esto se realiza únicamente en las funciones que se encuentran anidadas en el lado derecho de la subset-assertions.

su forma **"relacional"** es:

```
reach_contains (X, scons (X, phi) ).
reach_contains (scons (X, _), T1) :- edge_eq (X, S1), reach_contains (S1, T1)

edge_contains (1, scons (2, phi) ).
edge_contains (2, scons (1, phi) ).
```

fue necesario hacer esto para que Prolog reconociera este tipo de cláusulas en su forma habitual.

3) Como Prolog se encarga de hacer la "unificación" del "query", entonces se tuvo que cambiar la descripción de conjuntos a macros para poder utilizar conjuntos, ya que Prolog no hace uso de este tipo de argumentos, quedando de la siguiente forma.

```
reach_contains (X, scons (X, phi) ).
reach_contains (scons (X, _), T1) :-  break (X100, X, _),
                                      edge_eq (X, S1),
                                      reach_contains (S1, T1).
edge_contains (1, scons (2, phi) ).
edge_contains (2, scons (1, phi) ).
```

el "macro" break, maneja al conjunto con la palabra "scons" y sus dos argumentos X y _, esto se encuentra en forma relacional, pues es el estilo de programación para Prolog.

Ahora veamos como se realizó la expansión de cada cláusula.

4) La expansión de cada cláusula se agregó al programa para poder utilizar el concepto subset-assertions y transformar la cláusula con recursión de cerradura transitiva, a la forma estratificada, ya que para estos casos se ciclaría, por lo que se introdujo la estratificación en código Prolog. Veamos el resultado de la expansión.

```
reach_contains1 (A1, phi, _).
reach_contains1 (A1, scons (X, Y), Z) :-
    reach_contains (A1, Z1),
    member (X, Z1),
    not (member (X, Z) ),
    reach_contains1 (A1, Y, scons (X, Z) ),
    not (member (X, Y) ).
reach_not_subset (A1, X) :-
    reach_contains (A1, Z1),
    member (Z, Z1),
    not (member (Z, X) ).
reach_eq1 (A1, X) :-
    reach_contains1 (A1, X, phi),
    not (reach_not_subset (A1, X) ).
reach_contains1 (A1, X, phi) :-
    reach_eq1 (A1, X), !.
edge_contains1 (A1, phi, _).
edge_contains1 (A1, scons (X, Y), Z) :-
    edge_contains (A1, Z1),
    member (X, Z1),
    not (member (X, Z) ),
```

```
        edge_contains1 (A1, Y, scons (X, Z) ),
        not (member (X, Y) ),
    edge_not_subset (A1, X) :-
        edge_contains (A1, Z1),
        member (Z, Z1),
        not (member (Z, X) ),
    edge_eq1 (A1, X) :-
        edge_eq1 (A1, X), !.
    member (X, scons (X, _) ).
    member (X, scons (Z, Y) ) :- member (X, Y).
    break (scons (X, Y), X, Y).
    break (scons (X, Y), X1, scons (X, Y1) ) :- break (Y, X1, Y1).
```

como se puede observar, la idea fundamental de este proceso es el de transformar un programa que utiliza el enfoque funcional a un programa que se base en el enfoque relacional. El proceso crea nuevas variables, las cuales nos permitirán una interpretación relacional y una evaluación simplificada. Cabe señalar que scons es el constructor empleado por nuestra implementación para la manipulación de conjuntos.

Ejemplo 4.11

1.- Forma Original para la obtención de las posibles permutaciones de elementos de un conjunto.

```
permu (phi) contains {phi}
permu ( {H \ T} ) contains distr(H, permu (T)).
distr(H, {T \ _} ) contains { [H | T] }.
```

Al aplicarle el proceso de transformación antes mencionado obtenemos:

```
permu_contains1 (A1, phi, _).
```

```
permu_contains1 (A1, scons (X, Y), Z) :-
    permu_contains (A1, Z1),
    member (X, Z1),
    not (member (X, Z) ),
    permu_contains1 (A1, Y, scons (X, Z) ),
    not (member (X, Y) ).
permu_not_subset (A1, X) :-
    permu_contains (A1, Z1),
    member (Z, Z1),
    not (member (Z, X) ).
permu_eq1 (A1, X) :-
    permu_contains1 (A1, X, phi),
    not (permu_not_subset (A1, X) ).
permu_eq (A1, X) :-
    permu_eq1 (A1, X), !.
distr_contains1 (A1, A2, phi, _).
distr_contains1 (A1, A", scons (X, Y), Z) :-
    distr_contains (A1, A2, Z1),
    member (X, Z1),
    not (member (X, Z) ),
    distr_contains1 (A1, A2, Y, scons (X, Z) ),
    not (member (X, Y) ).
distr_not_subset (A1, A2, X) :-
    distr_contains (A1, A2, Z1),
    member (Z, Z1),
    not (member (Z, X) ).
distr_eq1 (A1, A2, X) :-
    distr_contains1 (A1, A2, X, phi),
    not (distr_not_subset (A1, A", X) ).
distr_eq (A1, A2, X) :-
    distr_eq1 (A1, A2, X), !.
```

```
member (X, scons (X, _) ).
member (X, scons (Z, Y) ) :- member (X, Y).
break (scons (X, Y), X, Y).
break (scons (X, Y), X1, scons (X, Y1) ) :- break (Y, X1, Y1).
```

En nuestro trabajo de tesis, el proceso de transformación a la forma "flattening" se realizó adaptando las diversas formas encontradas en la literatura incluyendo [JMO95], el cual está basado en la sustitución sistemática de todas las funciones halladas por un *, que posteriormente es sustituido por una variable generada, sacando de esta forma, la evaluación de las funciones a un solo nivel. El algoritmo empleado, se basa principalmente en la utilización de un árbol binario el cual proporciona mayores beneficios tales como el análisis sintáctico del programa, un mejor performance tanto en su transformación como en su ejecución, ya que éste solo requiere de hacer un recorrido sistemático sobre dicho árbol.

CONCLUSIONES

Este trabajo de investigación sobre la necesidad de extender los lenguajes para consultas a bases de datos deductivas o de programación lógica nos muestra que es posible contar con semánticas claras y sencillas, así como con una metodología no costosa para la implementación y prueba de las nuevas ideas y conceptos que surgen día a día en el área de la computación y, particularmente en la extensión de los lenguajes basados en cláusulas de Horn como se describe en el capítulo 3 y 4.

Otro punto relevante y principal en nuestro trabajo fue la incorporación del manejo de conjuntos a través del paradigma subset-equational. Y lo que es mas sobresaliente, el diseño de un algorítmo de transformación a programas normales (flattening), así como la re - escritura a forma relacional para poder corroborar que nuestros resultados son correctos.

Se obtuvo que la semántica operacional nos permite la realización de un sin fin de operaciones sobre conjuntos, que no utilizan recursión (ver ejemplos 3.1 y 3.2) gracias a el matching que se propone. Este además nos brinda una mayor eficiencia que si usaremos unificación, el cual es mas costoso y complejo.

La semántica propuesta en nuestro trabajo de investigación permite que esta sea adaptada fácilmente a cualquier lenguaje similar sin mucha dificultad por ser esta independiente.

El manejo de los constructores de conjuntos es directo, a diferencia de como lo manejan otros lenguajes, tales como LDL1 en el que su manejo es de forma oculta. Por lo anterior en nuestra propuesta se pueden resolver problemas que en otros lenguajes simplemente son resueltos de manera compleja y no clara para el programador (ver ejemplos del capítulo 3 y 4).

También se muestra la implementación de un enfoque que nos permite la manipulación de conjuntos de una manera clara y sencilla (sección 3.6). La cual puede ser fácilmente asimilada por estudiantes del ramo. Esto a nivel educativo es muy importante, ya que los estudiantes de cualquier curso en el que se trate este tema, podrán fácilmente estudiarlo para posteriormente enriquecerlo haciendo agregaciones a esta implementación. Esto debido a el lenguaje C que es estándar y muy accesible, así como el hardware (PC - compatible) el cual es accesible para cualquier centro educativo.

Por otro lado, es importante notar que la presente implementación tiene como finalidad el brindar un avance en la construcción de lenguajes con características significativas respecto a la manipulación de grandes volúmenes de información, lo cual es uno de los grandes problemas de los lenguajes enfocados a la manipulación de bases de datos deductivas.

Otro aspecto importante, es que la presente investigación sienta las bases para trabajos futuros en los cuales se piense en una implementación mas completa como lo es SuRE. El cual es un proyecto mas ambicioso y que podría trabajarse dentro de un proyecto doctoral.

Se plantea la implementación de un compilador del lenguaje SuRE, al que se le incorporen algunas características relevantes que existen tanto en la literatura como en algunos otros lenguajes (como CORAL, LDL1, etc.).

Como un último comentario, podemos señalar que en nuestra implementación se hizo uso de la explotación de las estructuras de datos empleadas (solo se construye una tabla principal sobre la que descansan todos los datos necesarios, esta es una lista ligada que cuenta con apuntadores a la raíz del árbol sintáctico y a las cláusulas restantes referidas a la misma definición), con la finalidad de reducir al máximo la creación y manipulación de demasiadas estructuras que hicieran mas compleja su manipulación y entendimiento para trabajos posteriores.

BIBLIOGRAFIA Y REFERENCIAS

[ABW88] K. Apt, H. Blair y A. Walker, "Toward a Theory of Declarative Knowledge," Poc. of the Workshop on Foundations of Deductive Databases and Logic Programming pp. 89-148, 1988

[AG91] S. Abiteboul y S. Grumbach "A Rule Based Language with Functions and Sets", ACM Transactions on Databases Systems, pp. 16(1):1-30,1991

[BF91] N. Bidoit C. Froidevaux "Negation by default and unstratified logic programs", Theoret. Comput. Sei. 78, pp. 85-112, 1991

[BN87] C.Beeri, S.Naqvi, et al "Sets and Negation in a Logic Databases Language (LDL1)", Proc. 6th ACM Principles of Databases Systems, pp. 21-37, 1987.

[Bra90] I.Bratko, "Prolog Programming for Artificial Intelligence", Addison-Wesley, 1990.

[BRSS92] C. Beeri, R. Ramakrishnan, D. Srivastava, y S. Sudarshan "The valid model semantics for logic programs" in Proceedings of the ACM Symposium on Principles of Database Systems, pages 91-104, June 1992.

[CM81] W.F. Clocksin y C.S.Mellish "Programming in Prolog" Springer-Verlag, New York, 1981.

[Das92] Subrata Kumar Das, "Deductive Databases and Logic Programming", Addison-Wesley, 1992.

[Dix91] J. Dix "Classifing Semantics of Logic Programs" Proc. 1th International Workshop of Logic Programming and Non-monotonic Reasoning, pp.166-180, Washington, D.C., July 1991.

[DOPR91] Dovier, A., Omodeo E. G., Pontelli E., y Rossi, G. "{log}: A Logic Programming Language with Finite Sets", Proc. 8th International Conference of Logic Programming, pp. 111-124, Paris, Junio 1991.

[GGZ91] G.Ganguly, S.Greco, y C.Zaniolo, "Minimum and maximum predicates in logic programs" Proceedings of the ACM Symposium on Principles of Databases Systems, pp.154-163, 1991.

[GL88] Gelfond y V. Lifschitz, "The Stable Model Semantics for Logic Programming," Proc. 5th International Conference of Logic Programming, pp. 1070-1080, Seattle, August 1988.

[J90] B. Jayaraman, "Implementation of Subset-Equational Programs," Por aparecer en: Journal of Logic Programming.

[Jay92] B. Jayaraman, "Implementation of Subset-Equational Programs", Journal of Logic Programming, 11(4):299-324, 1992.

[Jan94] D. Jana, "Semantics of Subset-Logic Languages", dissertation submitted to SUNY-Buffalo, 1994.

[JL87] J.Jaffar y J.L.Lassez, "Constraint Logic Programming", it Proc. Principles of Programming Languages Munich, 1987.

[JM95] B. Jayaraman y K. Moon, "Implementation of Subset Logic Programs", Submitted for publication.

[JN88] Jayaraman, B. y Nair A., "Subset-Logic Programming: Aplication and Implementation", Proc. JICSLP, pp. 841-858, Seattle, 1988.

[JOM95] B. Jayaraman, M. Osorio, and K.Moon, "Partial Order Programming, (revisited)", Proc, AMAST, Springer-Verlag, July 1995

[JP87] H. Jayaraman y D.A. Plaisted, "Functional Programming with Sets", Proc. Third FPCA, pp. 194-210, Springer-Verlag, 1987.

[JP89] Jayaraman, B. y Plaisted, D.A., "Programming with Equations, Subsets, and Relations", Proc. North American Conference of Logic Programming 89, pp.1051-1068, Cleveland, October 1989.

[JP90] B. Jayaraman y D.A. Plaisted, "Semantics of Stratified Subset-Equational Programming", submitted for journal publication, 1990

[KdMS90] G.Kiernan, C. de Maindre Ville y E. Simon. "Making deductive database a partial tecnology" en Procedings of the ACM SIGMOD Conf. on Managment of data, 1990.

[Kup90] Kuper, G.M., "Logic Programming with Sets", JCSS, 41(1);44-64, 1990.

[Llo87] J.W. Lloyd, "Fundations of Logic Programming", Springer-Verlag, 1987

[Lloy87] J. Lloyd, "Fundations of Logic Programming", (2 ed.), Springer-Verlag, 1987

[MPR90] I.S. Mumick, H. Pirahesh, and R. Ramakrishnan, "The Magic Sets and Aggregates", Proc. 16th VLDB, pp.264-277, 1990.

[N88] Nair, A., "Compilation of Subset.Logic Programs", M.S. Thesis, University of N. Carolina at Chapel Hill, December 1988.

[NM88] G. Nadathur y D. Miller, "An Overview of λProlog", In 5th Int'l Logic Programming Conference, Seattle, August 1988.

[NT89] S. Naqvi y S. Tsur, "A Logical Language for Data and Knowledge Bases", Computer Science Press New York

[O88] R.A. O'Keefe, Practical Prolog for Real Programmers", Tutorial Notes from 5th Intl. Logic Programming Conference, Seattle, August 1988.

[O95] M.Osorio, "Semantics of Logic Programs with Sets," Tesis doctoral, University of New York at Buffalo, septiembre, 1995

[OJ93] M.Osorio and B.Jayaraman, "Subset Assertions and Negation as Failure," TR93-026, SUNY-Buffalo, July 1993. Also a poster paper in Proc. International Logic Programming Symposium, Vancouver 1993.

[OJ96] M. Osorio, B. Jayaraman, "Aggregation and Well-Founded Semantics", Proceeding of the JICSLP96 Postconference Workshop W1, pp.30-47

[Prz88] T.C.Przymusinski, "On the Declarative Semantics of Stratified Deductive Databases," in J. Minker (de.), Foundations of Deductive Databases and Logic Programming, 1988, pp.193-216.

[PP90] T.C.Przymusinski and H.Przymusinska, "Semantic Issues in Deductive Databases and Logic Programs," in R.B.Banerji (de.), formal Techniques in Artificial Intelligence, 1990, pp.321-367.

[Ros94] K. Ross, "Modular Stratification and Magic Sets for DATALOG Programs with negation," Journal of the ACM, Vol. 41, No. 6, November 1994, pp. 1216-1266.

[RPDS88] Raghu, Praveen, Divesh y Sudarshan, "THE CORAL USER MANUAL A Tutorial introduction to CORAL", Computer Sciences Department, university of wisconsin-Madison, WI 53706, U.S.A. 1988.

[RU93] Raghu Ramakrishanan y Jeffrey D. Ullman, "A survey of Research on Deductive Database Systems", November 15, 1993.

[She92] J.C. Shepherdson, "Logics for negation as Failure", Logic from Computer Science, MSRI Publications 21, Springer-Verlag 1992.

[T85] D. A. Turner, "Miranda: A non-strict functional language with polymorphic types", in Conf. on Functional Prog. Langs. and Comp. Arch., Nancy, France, , pp. 1-16, Sep. 1985

[Van92] A. Van Gelder. "The well-founded semantics of aggregation". In proceedings of the ACM Symposium on Principles of Database Systems, pages 127-138, 1992.

[VRS91] A. Van Gelder, K.A. Ross y J.S. Schlipf, "The Well-Founded Semantics for General Logic Programs," JACM, 38(3):620-650.

[WMSG90] A.Walker, M.McCord, J.Sowa, W. Wilson, "Knowledge Systems and Prolog (second edition)," Addison - wesley

Los abajo firmantes, integrantes de jurado para el examen de grado que sustentará **el Lic. Fernando Zacarias Flores,** declaramos que hemos revisado la tesis titulada:

"BASES DE DATOS DEDUCTIVAS CON EXTENSION A CONJUNTOS FINITOS", consideramos que cumple con los requisitos para obtener el grado de Maestro en Ciencias, con especialidad en Ingeniería Eléctrica.

Atentamente

Dr. Sergio V. Chapa Vergara

Dr. Manuel Lazo Cortés

Dr. Mauricio Osorio Galindo