CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL

UNIDAD ZACATENCO
DEPARTAMENTO DE COMPUTACIÓN

Recuperación de Información en Bases de Datos Relacionales mediante Razonamiento Automático

Tesis que presenta

Marie Ely Piceno Cabrera

Para Obtener el Grado de

Maestra en Ciencias

En

Computación

Director de la Tesis

Dr. Guillermo Benito Morales Luna

México, Distrito Federal **Febrero, 2012**

Agradecimientos

Agradezco infinitamente a mi familia por el apoyo y amor que siempre me ha brindado.

Agradezco al *CINVESTAV-IPN*, a los profesores y compañeros del Departamento de Computación, por la enseñanza y convivencia durante este periodo. En particular, quiero agradecer a mi director de tesis, el Dr. Guillermo B. Morales Luna, quien con su experiencia, conocimientos y paciencia dirigió mi proyecto de tesis.

Agradezco al Consejo Nacional de Ciencias y Tecnología (*CONACyT*) por el respaldo financiero otorgado durante mis estudios.

Resumen

El objetivo de este trabajo es mostrar cómo expresar la información de una base de datos relacional dada mediante la formalización de sus componentes en una lógica de primer orden y cómo éstos pueden llevar a información inferida a partir de información inicial obtenida mediante consultas. Por lo tanto, la base de investigación de este trabajo es provista por la lógica formal, la interpretación de información, la deducción automática y las bases de datos relacionales.

Para lograr la formalización a lógica de primer orden, fue necesario exponer la semántica y sintaxis de la misma, esquematizar todas las partes de una base de datos relacional respecto a la lógica de primer orden. Esto es, establecer el cómo denotar los atributos, los individuos dentro de la base de datos, el conjunto de valuación, la dependencia de atributos, etc. en el ámbito de la lógica.

En lo que respecta a deducción automática analizamos en qué consiste, cómo surgen tanto la deducción automática como los demostradores automáticos, y las características esenciales de los principales demostradores automáticos desarrollados hasta el momento.

Abstract

Using first order logic we deal with the problem to extract knowledge from a relational database. The main steps of this process are: translation of the relational database to the logical language for a special signature built from the database by recognizing constants as object names in the database, predicates as relations in the database and function names as relational dependencies, either given explicitly in the database design or by discovering them through the analysis of the database; automatic reasoning tools on the translated knowledge base; and a tracking of known or newly discovered facts.

The implementation of the logical system, whose main focus is in automatic deduction, is addressed thus to relational database applications.

Índice general

Capítulo 1

Introducción

1.1. Antecedentes

De la mano del desarrollo de las bases de datos relacionales está el origen del lenguaje de consulta SQL (Structured Query Language), el cual es un lenguaje de consulta estructurado utilizado para modificar y acceder a datos o información almacenada en una base de datos. Dentro de sus características más importantes tenemos el manejo del álgebra relacional y el cálculo relacional, gracias a esta fuerte base teórica y a su orientación al manejo de conjuntos de registros permite una alta productividad en lo que a orientación a objetos y codificación respecta. Algunos de los comandos `SQL` son `SELECT, UPDATE, INSERT INTO, DELETE, WHERE,` etc. cada uno de ellos con una sintaxis, para su uso, bien definida.

Ahora bien, el trabajo de tesis plantea traducir el álgebra relacional a una lógica de primer orden y, mediante demostradores automáticos de dominio público, comprobar que se ha realizado de manera correcta dicha traducción.

La lógica de primer orden es un sistema formal que estudia inferencias en los lenguajes de primer orden. Para cada lenguaje deberá definirse su sintaxis (el alfabeto del lenguaje), para así determinar cuándo nos encontramos ante una fórmula bien formada, sus axiomas y sus reglas de inferencia. A partir de esta construcción, en el desarrollo de cualquier teoría, es posible modelar la información mediante proposiciones (cada una con un valor de verdad) para así establecer premisas y conclusiones que podrán ser demostradas o refutadas. La automatización de este procedimiento provee la base del razonamiento automático.

La motivación de este trabajo de tesis recae en el artículo[1], donde se presenta el modelado de una base de datos relacional en términos de una lógica epistémica simple con el propósito de recuperar nueva información. Aunque en el presente trabajo no tratamos con ella, a continuación describimos brevemente en qué consiste y damos una corta introducción a la misma puesto que, contemplamos a futuro el uso de la lógica epistémica para interpretar la información presente en una base de datos con estas caracaterísticas.

La lógica epistémica simple es una lógica formal derivada de la lógica modal [2]. La lógica modal está basada en el cálculo proposicional, sin embargo, la lógica modal agrega operadores, conocidos como *operadores modales*, los cuales califican proposiciones como "*posibles*" o "*necesarias*". Los sistemas modales epistémicos están basados en los mismos principios y reglas que los sistemas de lógica clásica, dado que son variantes de los sistemas modales, con la diferencia de que permiten el uso de nuevos conectores. La diferencia entre la lógica modal epistémica y otras variantes modales, estriba principalmente en la interpretación de estos conectores y en las aplicaciones de los mismos. Para entender mejor de que se trata más adelante exponemos la sintaxis y semántica de la lógica epistémica.

Con respecto al área de deducción de información mediante lógica, los primeros trabajos fueron dados por Von Wright [3], en el trabajo titulado *An Essay in Modal Logic*.

Las bases de datos relacionales son bases de datos que cumplen con el modelo relacional, modelo más utilizado para la representación de problemas reales y para administrar dinámicamente los datos. Los principios de las bases de datos relacionales fueron postulados por Edgar Frank Codd [4] en 1970. Para representar el esquema de una base de datos relacional se debe dar el nombre de sus relaciones, los atributos de éstas, los dominios sobre los que se definen estos atributos, las claves primarias y las claves secundarias. Entre las características esenciales de una base de datos relacional mencionamos a las siguientes: la información está ordenada mediante tablas, cada tabla es única, todas las operaciones sobre la base de datos se hacen sobre las tablas, una tabla contiene un número fijo de campos, el nombre de los campos de una tabla es distinto, el orden de los registros y de los campos no está determinado, para cada campo existe un conjunto de valores posible, las claves primarias son la clave principal de un registro dentro de una tabla y éstas deben cumplir con la integridad de datos, la relación entre una tabla padre y una tabla hija se lleva a cabo por medio de las claves primarias y secundarias, las claves secundarias que se colocan en la tabla hija, contienen el mismo valor que la clave primaria del registro padre (por medio de éstas se hacen las relaciones).

1.2. Planteamiento del Problema

El objetivo de esta tesis es exponer la conversión que hicimos de `SQL` a lógica de primer orden y, el estudio y la experimentación que se hizo en diversos demostradores.

Para el desarrollo del presente trabajo, será necesario exponer la semántica y sintaxis de la lógica de primer orden, determinar cómo se relaciona la lógica con las bases de datos relacionales. Esquematizar todas las partes de una base de datos relacional. Así como establecer el cómo denotar los atributos, los individuos dentro de la base de datos, el conjunto de valuación, la dependencia de atributos, etc. en el ámbito de la lógica de primer orden. Utilizamos este desarrollo teórico para llevarlo a la programación y así tener un sistema de consultas lo más completo, eficiente, eficaz

y funcional. Para extender este trabajo al ámbito de la lógica epistemica será necesario además plantear los diferentes tipos de conocimiento y el tipo de información (cognoscible, a la larga cognoscible y potencialmente cognoscible).

Dentro de la clasificación `ACM` [5] la tesis se ubica en cuatro áreas principales: software, sistemas de información (por ser un sistema que se encarga del procesamiento de consultas y por tratarse de un lenguaje de consulta), teoría de la computación (por ser una lógica de primero orden, que es una rama de la lógica matemática y lenguajes formales) y metodología de la computaciíon (por manejar deducción y demostración de teoremas).

1.2.1. Metodología

Dada una base de datos relacional, lo suficientemente ilustrativa, reconocemos los elementos de su universo, los atributos, y las correspondientes funciones de evaluación para cada atributo. Posteriormente implementamos un intérprete entre la lógica y la base de datos, y viceversa. Para alcanzar nuestras metas ha sido necesaria la realización de las siguientes tareas:

1. Investigación del estado del arte.
2. Identificar las principales características de las bases de datos relacionales.
3. Investigación sobre el manejador de base de datos más conveniente.
4. Programación de reconocedores de fórmulas bien formadas.
5. Programación de operaciones booleanas sobre fórmulas bien formadas.
6. Selección de una base de datos relacional.
7. Identificación de la estructura de la base de datos elegida.
8. Identificación de la signatura para la lógica propuesta partiendo de la estructura de la base de datos.
9. Esquema de traducción (definición de la signatura).
10. Pruebas sobre la base de datos.

Una de las características que se busca en la base de datos a elegir es que sea suficiente hacer consultas con lenguaje `SQL` normal, dejando de lado consultas `SQL` con recursión. Ésto debido a que no existe interés en aplicar este trabajo a bases de datos deductivas.

Tras la correcta traducción entre la base de datos y la lógica de primer orden, ha sido preciso, con el fin de obtener información partiendo de la ya presente en la base de datos, el uso de un demostrador automático de teoremas, lo cual ha implicado:

- Investigar los demostradores existentes.
- Evaluar los demostradores mediante pruebas directas.
- Seleccionar los demostradores adecuados a nuestro sistema.
- Realizar la implementación de la traducción del sistema anterior al demostrador.
- Implementar la traducción del archivo de salida del demostrador seleccionado como óptimo.

1.3. Estructura de la Tesis

Como queda dicho en los objetivos, el propósito de la tesis es recuperar y extraer información y conocimiento de una base de datos formalizándola en una lógica de primer orden para realizar en esta última procedimientos de demostración automática. De hecho el conocimiento extraído ha de versar también sobre el "metaconocimiento" implícito en la base de datos, tal como *qué puede ser deducido de la base*, *qué se conoce sobre la base*, *cuál es la estructura de la base* (en cuanto a sus dependencias funcionales o en cuanto a sus capacidades deductivas). Así pues las tareas principales en el trabajo son:

Conversión entre BD y LPO Planteamos una equivalencia entre BD y LPO (lógica de primer orden), y ésta es efectivamente realizable. Aquí se ha de tener mecanismos de conversión entre las BD y la LPO.

Deducción automática en la lógica Una aseveración de tipo lógico sobre la base de datos se ha de plantear como una *meta* ("*goal*") a demostrarse en la LPO.

Correctitud del sistema lógico Lo deducido formalmente, efectivamente puede ser obtenido del álgebra relacional de la BD o bien es consistente con ella.

Completitud del sistema lógico Todas las posibles consecuencias del álgebra relacional de la BD son demostrables en la lógica propuesta.

En el capítulo 2 hacemos una revisión a la literatura técnica en conexión con el tema de la presente tesis. En el capítulo 3 presentamos inicialmente el desarrollo teórico, la representación de cada una de las partes que componen a una base de datos relacional y a sus equivalentes en términos de una lógica de primer orden.

En el capítulo 4 mostramos el diseño y la implementación de programas basados en la equivalencia mencionada entre las bases de datos y la lógica de primer orden. Más adelante se describen las tablas de la base de datos sobre las cuales se han trabajado. También presentamos algunas técnicas de deducción automática para extraer y recuperar conocimiento a partir de la lógica de primer orden.

En el siguiente capítulo, capítulo 5, se exponen los resultados de las pruebas que se hicieron para comprobar la funcionalidad del sistema. Finalmente, en el capítulo 6, se presentan las conclusiones a este trabajo y el trabajo que contemplamos a futuro para extender el presente.

Capítulo 2

Lógica y Bases de Datos

Este trabajo plantea mostrar cómo expresar toda la información de una base de datos relacional dada mediante los modelos lógicos y cómo éstos pueden llevar a información inferida a partir de la información inicial obtenida mediante consultas, por lo tanto, la base de investigación de este trabajo es provista por la lógica formal, la interpretación de información, la deducción automática y las bases de datos relacionales. En lo que respecta a deducción automática analizaremos en qué consiste, cómo surgen tanto la deducción automática como los demostradores automáticos, y las características esenciales de los principales demostradores automáticos desarrollados hasta el momento.

2.1. Lógicas Formales

La lógica formal se dedica al estudio de la inferencia mediante la construcción de lenguajes formales, sistemas deductivos y semánticas formales con el fin de capturar las características esenciales de los lenguajes naturales que al ser estructuras formales y susceptibles al análisis matemático permiten realizar demostraciones rigurosas sobre ellas. La lógica de primer orden es una lógica formal y está definida sobre un conjunto de proposiciones, conectivos y reglas que permiten realizar funciones de deducción sobre la información representada.

En principio se deberá definir formalmente el lenguaje de la lógica de primer orden que estaremos manejando. Para ello debemos definir su alfabeto y dar las reglas sintácticas de buena formación de entes compuestos. El alfabeto del lenguaje de la lógica está conformado por:

- Paréntesis: (,);
- Conectivos lógicos: $\neg$, $\wedge$, $\vee$, $\rightarrow$;
- Proposiciones:

  - Variables proposicionales: Cualquier letra con o sin subíndices.
  - Cadenas α tal que α represente una proposición extraída de la base de datos.
- Cuantificadores: $\forall$, $\exists$;

Los elementos del lenguaje de la lógica, que llamaremos *proposiciones*, se definen como sigue:

- Toda variable proposicional es una proposición,
- Un átomo es considerado una proposición,
- Si A y B son proposiciones entonces $(\neg A)$, $(A \wedge B)$, $(A \vee B)$ y $(A \rightarrow B)$ también son proposiciones.

Los axiomas, o más bien los esquemas de axiomas, de la lógica de primer orden son:

1. $A \rightarrow (B \rightarrow A)$
2. $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
3. $(\neg A \rightarrow \neg B) \rightarrow (B \rightarrow A)$
4. $\forall x A \rightarrow A(x/t)$
5. $\forall x(A \rightarrow B) \rightarrow (\forall x A \rightarrow \forall x B)$

donde A, B, C son cualesquiera fórmulas bien formadas de primer orden.

2.2. Deducción Automática

Una demostración es el razonamiento lógico que indica que un enunciado, proposición o fórmula matemática es verdadero. Una demostración consiste en un conjunto de supuestos fundamentales, llamados axiomas o premisas, que se combinan de acuerdo con las reglas lógicas para deducir como conclusión la fórmula que se está demostrando. Una demostración de una proposición o fórmula P es un razonamiento válido a partir de premisas verdaderas hasta llegar a P como conclusión.

De aquí, se tiene que la deducción es definida como la serie de etapas lógicas por la cual se llega directamente a una conclusión a partir de enunciados iniciales (premisas) mediante la aplicación de reglas de inferencia. En un esquema de *reducción al absurdo*, una deducción es válida si una proposición o enunciado que afirme las premisas y niegue la conclusión es contradictoria.

La conclusión que se obtiene se dice que es una consecuencia lógica de las premisas si cada paso que se da para llegar a la conclusión está determinado por una regla.

Una parte importante de la lógica formal es precisamente la referente a inferencia y deducción.

Las reglas de inferencia son reglas de transformación sintáctica que son usadas para inferir conclusiones a partir de premisas. Las reglas de inferencia están formadas como un conjunto de "secuentes premisas" (precediendo a $\vdash$), también denominados *antecedentes*, y un único "secuente conclusión" (posterior a $\vdash$) denominado *consecuente*.

A cada conectivo lógico se le asocia dos clases de reglas de inferencia, una denominada *introduction rule* la cual permite derivar una fórmula teniendo al conectivo en cuestión como conectivo principal; la otra se denomina *elimination rule*, permite la eliminación de proposiciones de una fórmula que tiene como conectivo principal, el conectivo sobre el cual trata la regla de inferencia.

Reglas de inferencia:

- Negación: Reducción al absurdo

$$(\alpha \to \beta)(\alpha \to \neg\beta) \vdash \neg\alpha$$

  Reducción al absurdo relativa a la exclusión

$$(\neg\alpha \to \beta)(\neg\alpha \to \neg\beta) \vdash \alpha$$
$$\neg\neg\alpha \vdash \alpha$$

- Conjunción:

$$(\alpha)(\beta) \vdash \alpha \wedge \beta$$
$$(\alpha \wedge \beta) \vdash \alpha$$
$$(\alpha \wedge \beta) \vdash \beta$$

- Disyunción:

$$(\alpha \vee \beta)(\alpha \to \gamma)(\beta \to \gamma) \vdash \gamma$$
$$(\alpha \vee \beta)(\neg\alpha) \vdash \beta$$
$$(\alpha \vee \beta)(\neg\beta) \vdash \alpha$$

- Implicación:

  Modus Ponens

$$(\alpha \to \beta)(\alpha) \vdash \beta$$

  Modus Tollens

$$(\alpha \to \beta)(\neg\beta) \vdash \neg\alpha$$

- Bicondicional:

$$(\alpha \to \beta)(\beta \to \alpha) \vdash (\alpha \leftrightarrow \beta)$$

$$\begin{gathered}(\alpha \leftrightarrow \beta)(\alpha) \vdash \beta \\ (\alpha \leftrightarrow \beta)(\beta) \vdash \alpha \\ (\alpha \leftrightarrow \beta)(\neg\alpha) \vdash \neg\beta \\ (\alpha \leftrightarrow \beta)(\neg\beta) \vdash \neg\alpha \\ (\alpha \leftrightarrow \beta)(\alpha \vee \beta) \vdash \alpha \wedge \beta \\ (\alpha \leftrightarrow \beta)(\neg\alpha \vee \neg\beta) \vdash \neg\alpha \wedge \neg\beta\end{gathered}$$

La programación lógica parte de la idea de considerar a la lógica de primer orden como un lenguaje de programación.

El lenguaje de programación lógica por excelencia es `Prolog`, ideado a principios de los años 70. En un principio se trataba de un lenguaje totalmente interpretado hasta que en 1983 se desarrolló un compilador capaz de traducir `Prolog` a un conjunto de instrucciones de máquina abstracta. Las versiones iniciales del lenguaje diferían en muchos aspectos sintácticos hasta que en 1995 se estableció el estándar `ISO-Prolog`. Desde las formulaciones iniciales de `Prolog`, su descripción y análisis han incluido una interpretación de las expresiones de este lenguaje como fórmulas del lenguaje de la lógica de primer orden. Asimismo, cada paso del mecanismo computacional de `Prolog` se ha interpretado como la aplicación de una regla de inferencia deductiva, siendo la resolución SLD el procedimiento básico de deducción de `Prolog`.

Por su estrecha relación con la deducción lógica se ha considerado preciso adentrarnos al lenguaje de programación `Prolog`, partiendo de su sintaxis y analizando su funcionamiento.

Sintaxis de `Prolog` La unidad sintáctica de `Prolog` es toda cláusula de la forma:

$$A :- B_0, ..., B_n$$

donde cada A y B_i, $i = 0, ..., n$ son *estructuras*; A es denominada *cabeza* y B_i, $i = 0, ..., n$ es denominado *cuerpo*. Las cláusulas cuyo cuerpo es vacío son llamadas *hechos* o *cláusulas unitarias* y, aunque depende de los gustos de diversos autores, generalmente se omite el símbolo :-, por lo tanto tienen la forma A.

Un programa en `Prolog` es un conjunto de cláusulas.

Una estructura tiene la forma $p(t_1, ..., t_n)$, $n \geq 1$, denominando a p como predicado y como término a cada t_i. Existen dos tipos de términos: variables y constantes. Tanto términos como predicados son una secuencia de letras regidas, para su identificación, bajo las siguientes reglas:

- Los predicados comienzan con una letra minúscula.
- Las variables comienzan con una letra mayúscula.
- Las constantes comienzan con una letra minúscula.

Dentro de los términos encontramos también términos compuestos, son ellos mismos estructuras.

El propósito de `Prolog` es responder consultas a partir de un determinado programa. Una consulta tiene la misma sintaxis que el cuerpo de una cláusula

$$B_0, ..., B_n.$$

Haciendo la analogía con la lógica formal, cada uno de los predicados representa una proposición y una consulta representa al conjunto de premisas sobre las cuales se realiza la demostración.

`Prolog` responde a una consulta C a partir de un programa, construyendo una secuencia de metas $M_0, ..., M_m$, las cuales deben de satisfacer:

- $M_0 = C$.
- $M_{i+1} = M_i oper C_j$ siendo C_j una cláusula y *oper* un operador de `Prolog`.

M_m es denominada meta vacía, no contiene ninguna estructura. M_i representa, análogamente a la lógica de primer orden, a cada paso del camino seguido para resolver la consulta dada.

Si `Prolog` puede construir una secuencia de estas características a partir del programa introducido responderá *sí* a la consulta C. En caso contrario, la respuesta será *no*.

`Prolog` utiliza un algoritmo específico para construir la secuencia de metas intermedias, en este algoritmo se establece un orden en el cual se analiza al programa ante cada meta no vacía, buscando alguna cláusula para la que el operador de `Prolog` esté definido. El algoritmo no es completo, es decir, existen situaciones en las que no llega a construir una secuencia que permita responder *sí* a una consulta, a pesar de que existe tal secuencia.

Análogamente a la lógica formal el algoritmo para construir la secuencia de metas son las reglas de inferencia y las metas obtenidas son los pasos de deducción al aplicar dichas reglas.

2.3. Demostradores Automáticos

Existen diversos congresos dedicados únicamente a la deducción de información mediante la lógica, tal es el caso de *Machine Learning and Knowledge Discovery in Databases* y *Logics in Artificial Intelligence* ambas realizadas en Europa anualmente, en cuyas memorias se pueden encontrar diversos trabajos relacionados a nuestro tema. A continuación exponemos ejemplos de trabajos concernientes al tema aquí tratado:

TPS(*Theorem Proving System*) [6] Sistema para la demostración de teoremas, basado en lógica de primer orden y teoría de tipos, programado en `Common Lisp`, cuyos autores son Peter Andrews, Matthew Bishop, Chad E. Brown y Sunil Issar. Las posibles aplicaciones del demostrador automático de teoremas incluyen el hardware y la verificación de software, la automatización parcial de diversas actividades matemáticas, promover el desarrollo de las teorías formales en una amplia variedad de disciplinas, así como los sistemas deductivos de información para estas disciplinas, los sistemas expertos sobre los cuales razonar, y determinados aspectos de la inteligencia artificial. TPS se puede utilizar para demostrar teoremas de la lógica de primer orden, de forma interactiva, de forma automática, o en una mezcla de estos modos, aunque se reporta que en modo automático es bastante primitiva en ciertos aspectos, como en el tratamiento de la igualdad. TPS ganó la primer competencia de sistemas de demostración automática de teoremas de orden superior que se han desarrollado. La competicia fue parte de *The CADE-22 ATP System Competition (CASC-22)* en la 22ª Conferencia Internacional sobre la Deducción Automática (CADE-22) en Montreal, Canadá, en Agosto de 2009.

ETPS (*Educational Theorem Proving System*) [7] Construido con las mismas bases que TPS pero con fines educativos. El programa permite a los estudiantes concentrarse en los problemas esenciales de la lógica subyacente a las pruebas, y les da una respuesta inmediata. ETPS permite a los estudiantes a trabajar deductivamente, inductivamente, o en una combinación de ambas, y proporciona facilidades para el reordenamiento de las pruebas, borrar partes de las pruebas, mostrar sólo las partes de las pruebas que están siendo estudiadas activamente, el ahorro de pruebas incompletas, e impresión en papel de las pruebas. El editor de fórmulas permite al estudiante extraer las fórmulas que se producen en cualquier lugar de la prueba, y construir nuevas fórmulas de ellas.

***Modal logics in the theory of relational databases* [8]** Establece un formalismo lógico para el razonamiento sobre las propiedades representadas mediante un conjunto de tuplas proviendo de significado a dos operadores con respecto del conjunto de atributos. Cuando es aplicado el primer operador sobre un conjunto de tuplas dado, extrae las tuplas que efectivamente satisfacen la propiedad representada por el conjunto en cuestión, y el segundo operador extrae las tuplas que posiblemente satisfagan dichas propiedades. Se asocia este tipo de propiedades con el lenguaje de algún sistema bien conocido de lógica modal para que el razonamiento sobre estas propiedades pueda ser trasladado al dominio de la lógica.

***Modal Logics as Labelled Deductive Systems*[9]** Amplia la lógica modal estándar para permitir el razonamiento sobre distintos mundos.

Por otro lado, los modelos basados en métodos formales de lógica han jugado un papel importante dentro de la especificación y la verificiación automática de protocolos.

En uno más de los trabajos [10] de esta área se hace uso de la lógica epistémica. Esta lógica puede ser diseñada o especificada para lograr un mayor alcance al formalizar aspectos deductivos de las bases de datos relacionales.

Nuestro objetivo principal es implementar un demostrador automático que funcione basándose en los axiomas y reglas de inferencia de la lógica de primer orden. Dada una proposición se deberá determinar si es posible obtenerla aplicando las reglas de inferencias pertinentes sobre la información obtenida de una base de datos.

La teoría de la demostración es el estudio del razonamiento matemático en un sentido general y abstracto. Se enfoca principalmente en qué camino seguir para poder derivar la proposición a demostrar basándose en axiomas preestablecidos.

Una demostración también se puede definir como un riguroso procesamiento matemático que inequívocamente demuestra la veracidad de una proposición dada. Una sentencia matemática que ha sido demostrada es denominada *teorema*.

Una manera muy simple, pero extremadamente costosa, para demostrar proposiciones lógicas es mediante el uso de tablas de verdad y circuitos lógicos donde cada una de las salidas depende de los valores de entrada y de la combinación de los elementos del circuito (compuertas AND, OR, NAND, NOR, NOT, XOR, etc.). Los valores que puede tomar cada entrada pueden ser verdadero o falso. Las leyes de De Morgan son utilizadas principalmente para el análisis y la simplificación de cada circuito, o en su caso de cada proposición.

Gracias al desarrollo de computadoras surge un nuevo tipo de deducción, la deducción automática. La deducción automática, es un área de investigación matemática de extraordinario potencial práctico. Se tienen sospechas de que el primer demostrador automático fue el denominado *Logic Theory Machine*, diseñado por H. Simon y A. Newell, e implementado por J.C. Shaw en 1955[11].

Desde una perspectiva histórica, podemos identificar tres tendencias principales dentro de la deducción automática:

- Simular el proceso de razonamiento humano.
- Centrarse en la obtención de resultados de manera automática.
- Utilizar enfoques interactivos, en los que el proceso de deducción mecanizada está guiado, en mayor o menor medida, por el usuario.

Actualmente existen varios sistemas enfocados en la demostración automática de teoremas que funcionan de diferentes formas y para diversos objetivos. La mayoría de ellos están desarrollados como una subarea de un razonador automático.

Algunos demostradores de teoremas inductivos requieren de un usuario que funja como guía para el sistema. Dependiendo del grado de automatización, el demostrador puede esencialmente reducirse a un verificador de pruebas que el usuario provee de manera formal, o también podría ser capaz de realizar tareas de demostración significativas de manera automática.

Otras veces la distinción es hecha entre la demostración de teoremas y otras técnicas, donde un proceso es considerado para ser demostrado si consiste en una demostración tradicional a partir de los axiomas y produciendo nuevos pasos de inferencia usando las reglas de inferencia. Otras técnicas pueden incluir un verificador de modelo, que es equivalente a la enumeración de los posibles estados.

Existen demostradores de teoremas híbridos que usan el modelo de verificación como una regla de inferencia. Hay inclusive programas que son escritos para demostrar un teorema en particular, si el programa culmina devolviendo un resultado en particular, se dice que el teorema es verdadero.

Otros demostradores de teoremas, basados en lógica de primer orden, son lo suficientemente potentes como para permitir la especificación de problemas arbitrarios, a menudo de una manera razonable e intuitiva. Por otro lado, todavía es semi-decidible, aunque se han desarrollado una serie de cálculos en la búsqueda de encontrar un sistema totalmente automatizado.

Lógicas como las lógicas de orden superior, permiten la expresión práctica de una amplia gama de problemas de lógica de primer orden, pero la teoría de pruebas para estas lógicas no está tan bien desarrollada.

Con el fin de construir el demostrador adecuado a nuestros requerimientos, enlistamos algunos de los demostradores automáticos, de dominio público, más importantes:

PVS Specification and Verification System Propuesto para capturar el estado de arte en métodos formales mecanizados. Posee una colección poderosa de procedimientos de inferencia primitivos, los cuales son aplicados interactivamente dentro de un marco de cálculo de secuentes. La secuencia primitiva viene optimizada para demostraciones largas e incluye reglas cuantificadoras, de inducción y de reescritura. También permite la abstracción de datos y predicados, y la verificación simbólica de modelos. Los procedimientos definidos por el usuario pueden cambiarse mediante la inferencia primitiva para lograr un alto nivel en las estrategias de demostración. Las demostraciones pueden ser complementadas con fórmulas adicionales, editadas o recompiladas; ésto permite que teoremas similares sean demostrados eficientemente, que las demostraciones sean ajustadas con un bajo costo en base a los requerimientos o el diseño, así como el desarrollo de demostradores legibles. Al instalarlo requiere `Allegro Common Lisp`, el cual tiene un costo elevado pero aumenta la velocidad de procesamiento, o `CMU Common Lisp`, software libre no soportado por el sistema operativo `Mac`. Se utiliza como generador de código y como un demostrador aleatorio.

SNARK Utiliza reglas de inferencia de resolución y paramodulación y fue desarrollado en `Common Lisp`.

PHOX Es un editor de demostraciones para lógicas de alto nivel. Implementa el sistema de tipos de Krivine el cual permite derivar programas con el fin de demostrar sus especificaciones, es decir, se encarga de la verificación de demostraciones.

HOL4 Implementa herramientas de demostración, tiene inmersos procedimientos de decisión, demuestra teoremas simples mientras que teoremas complejos deben ser demostrados por los usuarios y el programa se encargará de verificar la demostración.

SPASS Se define como un demostrador automático de teoremas para lógica de primer orden con igualdad. Por lo tanto, la entrada del demostrador es una fórmula de primer orden bajo la sintaxis propia del programa. Las salidas posibles del demostrador son `SPASS beiseite: Proof found.` en caso de que la fórmula sea valida y `SPASS beiseite: Completion found.` en caso de que la fórmula no sea valida o sea indecidible, si la fórmula no ha sido bien definida `SPASS` puede ejecutarse infinitamente sin ningún resultado final puesto que no encuentra un camino para validar o refutar. El archivo de entrada del programa se puede dividir en tres partes, en la primera parte se presenta la lista de símbolos a usar, en la segunda parte se exponen los axiomas que se quieren utilizar y la tercera parte contiene la o las conjeturas a las que se quieren llegar. Dado que el programa está basado en refutación, SPASS transforma las fórmulas a su forma normal conjuntiva y niega las conjeturas.

E Es un demostrador de teoremas para lógicas de primer orden con igualdad. Recibe como entrada un conjunto de fórmulas tomadas como axiomas y la conjetura, y da como resultado una prueba formal a la conjetura partiendo de los axiomas. Si la demostración es encontrada, el sistema es capaz de proveer una lista detallada de la demostración. Está implementado en `C`. `E` se encuentra, junto con `Vampire` y `SPASS`, en el núcleo de la estrategia *Isabelle's Sledgehammer*. `E` también es el motor de razonamiento en el `SINE` y `LEO-II`.

Vampire Es un demostrador automático de teoremas para la lógica de primer orden. Desarrollado por la Universidad de Manchester. Implementa el cálculo por resolución binaria y superposición. Para acelerar la respuesta del sistema se usa un algoritmo especializado en el tiempo de ejecución. Aunque el núcleo del sistema sólo funciona con proposiciones expresadas en forma normal conjuntiva, el preprocesador acepta un problema en la sintaxis de la lógica de primer orden, lo transforma a clausulas y realiza una serie de transformaciones útiles antes de pasar el resultado al núcleo. Cuando un teorema está demostrado, el sistema produce una prueba verificable.

Prover9 Demostrador automático para la lógica de primer orden y la lógica ecuacional, es sucesor de Otter. La entrada estándar del demostrador la podemos dividir en dos secciones principales, en la primera parte introducimos información que queremos sea tomada como axiomas y en la segunda parte la conjetura a la que se desea llegar. Es posible especificar el tiempo de ejecución que le queremos dar a nuestro programa así como el número de variables libres y demás información asociada a la demostración. Como salida el programa devuelve un archivo con los pasos seguidos

para la demostración. También es posible que devuelva mensaje de error si alguna parte de la información de entrada no cumple con la sintaxis.

Otter Se basa en la lógica de primer orden y en la lógica ecuacional. Actualmente no está siendo desarrollado, el soporte y mantenimiento brindado es mínimo.

Isabelle Demostrador interactivo de teoremas, sucesor de HOL. La aplicación principal es la formalización de las pruebas matemáticas y en la verificación formal particular, que incluye la prueba de la exactitud de hardware o de software y la verificación de las propiedades de los lenguajes de programación y protocolos. Implementado en ML. Es un sistema genérico, va desde la metalógica hasta la lógica de orden superior. `Isabelle` también cuenta con eficientes herramientas de razonamiento automático, como un motor de reescritura y una demostración por tablas de verdad, así como varios procedimientos de decisión. Es software libre y está publicado bajo la licencia BSD. Hoy en día la distribución más común de Isabelle es Isabelle/HOL, la cual proporciona un ambiente de un demostrador de teoremas de lógica de orden superior para ser utilizado por aplicaciones de tamaño considerable.

KeY Es una herramienta que permite integrar el diseño, la implementación, la especificación formal y la verificación de software orientado a objetos. El núcleo del sistema es un demostrador automático de teoremas para lógica de primer orden, programado en Java con el uso de una interfaz amigable para el usuario. El proyecto fue empezado en Noviembre de 1998 por la Universidad de Karlsruhe. En la actualidad es desarrollado conjuntamente por el Instituto de Tecnología de Karlsruhe, la Universidad Tecnológica de Chalmers, Gothenburg y TU Darmstadt. Existen diversas distribuciones y variantes del sistema `KeY`, entre los más relevantes encontramos `KeY-Hoare`, `KeYmaera`, `KeY para C`, `ASMKeY`.

ACL2 ***(A Computational Logic for Applicative Common Lisp)*** Fue diseñado para soportar el razonamiento automático sobre teoria de lógica inductiva, encaminado a la verificación de software y hardware. es parte de la familia de demostradores Boyer-Moore, cuyos autores recibieron el *ACM Software System Award (2005)*. El núcleo del demostrador está basado en la reescritura de términos. Es extendible, se agregan al núcleo los teoremas desmostrados por los usuarios con el proposito de incrementar las técnicas de demostración para las subsecuentes conjeturas.

Capítulo 3

Traducción entre Álgebra Relacional y Lógica de Primer Orden

La relación existente entre las bases de datos relacionales y la lógica ha sido ampliamente estudiada, analizada y explotada mediante implementaciones practicas.

Enfocamos esta sección en modelar problemas concernientes a las bases de datos relacionales a través del diseño de una lógica de primer orden particular, presentando una interpretación entre sentencias lógicas, el álgebra relacional y sentencias en `SQL`.

3.1. Formalización en LPO de una Base de Datos Relacional

3.1.1. Diseño de la Base de Datos

Denotemos por $\mathcal{A}$ al conjunto de atributos presente en la base de datos. Para cada $A \in \mathcal{A}$ se define el dominio de A mediante $D(A) \neq \emptyset$. Un esquema relacional es una palabra $\alpha \in \mathcal{A}^*$. Si $\alpha = A_{i_0} \cdots A_{i_{k-1}}$ entonces una relación$-\alpha$, o tabla$-\alpha$, se define como el conjunto $R \subset \prod_{j=0}^{k-1} D(A_{i_j})$. Las tuplas $\mathbf{r}$ que aparecen en R son llamados registros. Si A es un atributo dentro del esquema relacional α, $\mathbf{r}[A]$ denota la entrada del registro $\mathbf{r}$ correspondiente al atributo A.

Una base de datos relacional$-\mathcal{A}$ es una familia $\Gamma = \bigcup\{\Gamma_\alpha |\ \alpha \in \mathcal{A}_\Gamma \subset \mathcal{A}\}$ donde Γ_α es el conjunto de tablas$-\alpha$ y $\mathcal{A}_\Gamma$ es el conjunto de palabras en $\mathcal{A}$. Cada uno de estos elementos en la base de datos relacional Γ es una tabla.

Para cada tabla$-\alpha$ R, sea $V(R) = \bigcup\{D_A |\ A$ es un atributo dentro de $\alpha\}$ el conjunto de todos los objetos (potencialmente) involucrados en la relación R, sea $V(\Gamma_\alpha) = \bigcup\{V(R) |\ R \in \Gamma_\alpha\}$ y sea $V(\Gamma) = \bigcup\{V(\Gamma_\alpha) |\ \alpha \in \mathcal{A}_\Gamma\}$. Se tiene que $V(\Gamma)$ es el *universo* de todos los objetos contenidos es la base de datos relacional$-\mathcal{A}$ Γ.

Sea $v \in V(\Gamma)$ un objeto. Para cada esquema relacional $\alpha \in \mathcal{A}_\Gamma$, definimos como v_α al conjunto de tablas$-\alpha$ involucradas en v como $v_\alpha = \{R \in \Gamma_\alpha |\ v \in V(R)\}$. De manera análoga, se define $v_\Gamma = \bigcup\{v_\alpha |\ \alpha \in \mathcal{A}_\Gamma\}$ como el conjunto de relaciones en la base de datos relacionas$-\mathcal{A}\ \Gamma$.

Ahora bien, el problema principal a tratar consiste en decidir si un objeto dado $v \in V(\Gamma)$ está totalmente caracterizado por el conjunto v_Γ o por un fragmento de la forma $v_\Gamma \cap \mathcal{R}$, donde $\mathcal{R}$ es un conjunto de relaciones dentro de Γ.

La representación en la lógica de primer orden de la base de datos relacional Γ comienza por el reconocimiento de su *signatura*. Mediante la cual se reconocerán y representarán los objetos pertenecientes al universo $V(\Gamma)$, los símbolos de relación dentro de las tablas en Γ y los símbolos de función para denotar dependencias relacionales dentro de Γ o algunos de estos fragmentos. Así, una lógica de primer orden es construida sobre esta *signatura* si se desea extender esto a la lógica epistémica se deberá enriquecer el sistema con los operadores modales correspondientes a la noción de *recuperación de información*.

Una tabla R será identificada mediante la tupla $(\mathcal{V}_R, \mathcal{A}_R, \Pi_R)$ donde:

- $\mathcal{V}_R = V(R) \neq \emptyset$ es el universo de R como se definió anteriormente, $\mathcal{V}_R \subset V(\Gamma)$,
- $\mathcal{A}_R \subseteq \mathcal{A}$ es el esquema relacional correspondiente a R, y
- $\Pi_R = (\Pi_{AR})_{A \in \mathcal{A}_R}$ donde, $\forall A \in \mathcal{A}_R$: $\Pi_{AR} : \mathcal{V}_R \to \mathrm{Dom}(A)$ el conjunto de las funciones de proyección de cada atributo presente en la tabla R.

3.2. Sistema Lógico para una Base de Datos Relacional

Toda lógica de primer orden es construida sobre una *signatura* $\sigma = (\mathcal{S}_{cte}, \mathcal{S}_{fun}, \mathcal{S}_{rel})$, donde:

- $\mathcal{S}_{cte}$ corresponde al conjunto de constantes,
- $\mathcal{S}_{fun}$ corresponde al conjunto de funciones, y
- $\mathcal{S}_{rel}$ corresponde al conjunto de relaciones.

Dada una base de datos relacional es posible obtener su correspondiente sistema lógico mediante la idenificación de los componentes de la signatura.

Constantes Designan objetos de un tipo dado. Dentro de la base de datos dada cada atributo describe un tipo de constante. Los valores que toma cada tipo de constante deben ser identificados para así poder describir mediante una gramática, conjuntos previamente definidos o de manera explícita los dominios de cada una de las constantes.

Relaciones Corresponden a los nombres de las relaciones y a los tipos de constantes.

Funciones Corresponden a las dependencias relacionales que se tienen, ya sea dentro de toda la base de datos Γ o sólo en una parte de ésta.

Teniendo la base de datos Γ, es posible construir procedimentalmente la signatura para la lógica.

El reconocimiento de las constantes y los símbolos de relación es realizado a través de una prospección lineal de Γ. El reconocimiento de los símbolos de función es reducido al reconocimiento de relaciones de dependencia dentro de Γ, que en realidad es un problema muy difícil, intratable en la mayoría de los casos. Sin embargo, el diseño de la base de datos pueden ser de gran ayuda puesto que nos provee las dependencias relacionales más relevantes, es decir, los símbolos de función más relevante.

Esta signatura nos describe un alfabeto el cual, mediante reglas libres de gramática, nos permite construir y reconocer términos, átomos y fórmulas booleanas, mediante procedimientos sintácticos.

Por lo tanto, existen procedimientos eficaces para la traducción de una base de datos relacional a la propuesta lógica de primer orden.

En la siguiente sección vamos a ilustrar algunos de los principales procedimientos.

3.3. Construcción de un Lenguaje Lógico para un Esquema Relacional

3.3.1. Signatura

Suponga que tenemos una tabla R en una base de datos en `Excel`. En principio ésta es transformada a formato `ASCII` para poder trabajar sobre ella.

La primera fila de cada archivo contendrá el nombre de los atributos presentes en la tabla, denotemoslos mediante A_R. Para cada J, el dominio de los atributos A_{jR} consiste en todos los valores de la j-ésima columna del archivo de texto que está siendo analizado. Se obtiene un nuevo archivo por cada atributo de manera tal que el nombre de este archivo corresponde al nombre del atributo A_{jR} y su contenido es el dominio de dicho atributo. Un operador unión, que no permite la existencia de archivos bajo el mismo nombre mas toma en cuenta el conjunto completo del dominio de atributos con el mismo nombre, es implementada.

Estos archivos se analizan nuevamente con el objetivo de encontrar una fórmula que los represente de la manera más fiel posible. Ya sea mediante una gramática, dando los valores del dominio de manera explícita o asociandolos con un rango o conjunto relevante. Muchas características de los dominios de los atributos son analizadas: la cardinalidad (si, $\text{card}(\text{Dom}(A_{jR})) \leq k$), tipo de dato, etc.

Para determinar la manera en la que se representará el dominio lo primero que se

hace es preguntar la cardinalidad del dominio de cada atributo. Si para determinado atributo su cardinalidad es menor que un número dado, se expone el dominio de manera explícita. De tener un dominio mayor, se averigua qué tipo de datos se están manejando, si se trata de únicamente valores enteros, reales, cadenas alfanuméricas, etc. En caso de tener únicamente números enteros o números reales, se devuelve el intervalo en que estos valores corren especificando si se trata de números reales o enteros.

De otra manera, se procede a calcular la gramática de los elementos del dominio. Si se obtiene más de un número determinado de gramáticas que representen al dominio y en éstas la densidad de letras es considerablemente mayor que la cantidad de dígitos, se dice que el dominio del atributo que está siendo analizado cae sobre el conjunto de expresiones alfanuméricas. Si el número de gramáticas obtenidas no sobrepasa los límites impuestos entonces se dice que el dominio del atributo que está siendo analizado está dado por las gramáticas obtenidas.

Así, se obtiene un archivo que contiene el nombre de los atributos con su correspondiente dominio.

3.3.2. Proceso de Análisis Sintáctico

Enseguida se exponen los principales pasos para generar la síntesis gramatical para tipos de datos alfanuméricos.

Símbolos del Lenguaje

Se consideran símbolos del lenguaje a constantes, funciones y relaciones.

Constantes Sea $\mathcal{A}$ el conjunto de atributos de la base de datos relacional Γ la cual está conformada por un conjunto de tablas $\mathcal{T}$, se tiene que c es una constante si para cualquier $\alpha \in \mathcal{A}$, $c \in Dom\langle\alpha\rangle$.

El programa implementado para reconocer constantes recibe como entrada la cadena a analizar y, a partir del archivo donde se encuentran descritos todos los atributos de la base de datos con sus correspondientes dominios, determina si esta cadena pertenece o no a alguno de los dominios. En caso de pertenecer, la cadena es reconocida como constante y se despliega el tipo de la misma, de otra forma se anuncia que la cadena no es una constante.

Funciones El programa reconocedor de funciones recibe como entrada un archivo `ASCII` donde se encuentran las cadenas a reconocer, que serán extraídas linea a linea del archivo. Las funciones son símbolos que denotan aplicaciones $f : D_1 \times ... \times D_{narg} \to C$, por lo tanto, al ser analizada una cadena que efectivamente represente una función el programa devuelve una estructura consistente de:

- Nombre de la función: f
- Aridad: $narg$
- Dominio:

$$D_1 :: Tipo_1 \ldots D_{narg} :: Tipo_{narg}$$

- Contradominio: $TipoC$

También permite la definición de nuevas funciones. La especificación de las funciones es guardada en un archivo `ASCII` nombrado *funciones.txt*.

Relaciones Para obtener la información de las relaciones presentes, el programa recibe como entrada los archivos donde se definen los atributos y sus dominios. Si A_i es el nombre de un atributo, A_i será reconocida como una relación devolviendo la información siguiente:

- Nombre de la relación: A_i
- Aridad: $narg$
- Universo
- Contradominio

Esta información es almacenada en un archivo `ASCII` denominado `relaciones.txt`.

Una vez reconocidos los símbolos del lenguaje se procede a reconocer los términos, átomos y fórmulas.

Términos

Sean Var un conjunto de variables previamente definidas y $Const$ el conjunto de constantes, se tiene:

- Si $\delta \in \{Var \cup Const\}$, entonces $\delta \in Términos$.
- Si f es una función y $\alpha_0, ..., \alpha_n \in Términos$, entonces $f(\alpha_0, ..., \alpha_n) \in Términos$.

El procedimiento utilizado para el análisis sintáctico se realiza recursivamente. Dada una cadena α, se procede a analizar si es una constante, de no serlo el algoritmo verifica si se trata de una variable comparándolo con el conjunto de variables previamente definido. Si la cadena es una variable se expone como tal y se reconoce como término.

Si por ninguno de los medios anteriores se ha reconocido a la cadena α como término, se accede al archivo *funciones.txt* para revisar si la cadena coincide con alguna de las definiciones de las funciones. Si se encuentra coincidencia con alguna función, el programa devuelve que efectivamente la cadena a analizar es un término y que no lo es en caso contrario.

Átomos

Los átomos se definen de la siguiente manera.

Si f es una relación y $\alpha_0, ..., \alpha_n \in \textit{Términos}$, entonces $f(\alpha_0, ..., \alpha_n) \in \textit{Átomos}$.

El programa reconocedor de átomos accede al archivo *relaciones.txt* para revisar si la cadena coincide con alguna de las definiciones de las relaciones. Si se encuentra coincidencia con alguna relación, se procede a examinar los argumentos el programa devuelve que efectivamente la cadena a analizar es un átomo y que no lo es en caso contrario.

Fórmulas bien formadas

A partir de que se han identificado los átomos es posible definir cuando se tiene o no una fórmula bien formada dentro de la lógica. Las reglas de construcción se enuncian formalmente a continuación:

- $\varphi \in \textit{Átomos} \Rightarrow \varphi \in \textit{Fórmulas}$
- $\varphi, \psi \in \textit{Fórmulas} \Rightarrow \varphi \boxplus \psi \in \textit{Fórmulas}$.
- $\varphi \in \textit{Fórmulas} \Rightarrow \neg\varphi \in \textit{Fórmulas}$
- $\exists x, \varphi$ es una fórmula donde x es una variable y $\varphi \in \textit{Formulas}$
- $\forall x, \varphi$ es una fórmula donde x es una variable y $\varphi \in \textit{Formulas}$ (de manera alternativa, $\forall x, \phi$ puede ser definida como una abreviación de $\neg\exists x, \neg\varphi$).

Donde $\boxplus = \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$. En caso de trabajar con fórmulas dentro de la lógica epistemica simple, habrá que añadir a las reglas anteriores:

- $\varphi \in \textit{Fórmulas} \Rightarrow \boxdot\varphi \in \textit{Fórmulas}$

donde $\boxdot$ representa a los operadores modales de necesidad ($\Box$) y posibilidad ($\Diamond$).

Al recibir una cadena se verifica que ésta empate con la estructura que se tiene en las fórmulas bien formadas.

3.3.3. Conversión de SQL al Lenguaje Lógico

Se desea establecer comunicación del gestor de la base de datos con el lenguaje lógico antes definido para poder dotar de información al sistema lógico. Por lo tanto, es preciso definir la conversión de cualquier consulta `SQL` al lenguaje lógico.

Para poder realizar dicha conversión se deben de identificar los enunciados propios de `SQL`, su sintaxis y su significado para así poder darles una interpretación adecuada dentro de la lógica.

A continuación se presenta la sintaxis de consultas `SQL` y sus respectivas equivalencias tanto sintáctica como semánticamente, dentro del lenguaje lógico.

Sea $\mathcal{A}$ el conjunto de atributos de la base de datos, R una tabla dentro de la base de datos, $\mathcal{A}_R$ el conjunto de atributos de la tabla R.

SELECT.

Se considerarán varios tipos de consultas `SELECT` con su correspondiente traducción a la lógica de primer orden.

Consulta en forma general

`SELECT` *nombre_atrib* `FROM` *tabla complemento*

donde *tabla* $= R$ es el nombre de una relación R en la base de datos. Denotemos por

$$\mathcal{A}_R = \{A_1, A_2, \ldots, A_i, \ldots, A_n\}$$

al esquema relacional de R. Se ha de tener que *nombre_atrib* $= (A_i)_{i\in I}$, con $A_i \in \mathcal{A}_R$, es una lista de atributos en $\mathcal{A}_R$. El argumento *complemento* será descrito posteriormente. La consulta anterior es equivalente dentro del álgebra relacional a:

$$R' = \{\mathcal{V}'_R, \mathcal{A}'_R, \Pi'_R\} \text{ donde}$$

- $\mathcal{V}'_R$=$\{v \in \prod_{i\in I} \mathrm{Dom}(A_i) : \exists u \in \prod_{j\notin I} \mathrm{Dom}(A_j), v * u \in R\}$
- $\mathcal{A}'_R$=$(A_i)_{i\in I}$
- Π'_R=$(\Pi_{AR'})_{A\in A'_R}$

y es equivalente en la lógica a:

$$\bigwedge\{\rho(\texttt{--A}_1, \texttt{--A}_2, \ldots, a_i, \ldots, \texttt{--A}_n)|\ a_i \in \mathrm{Dom}(A_i) \forall i \in I\},$$

donde $\texttt{--A}_j$ denota a variables del tipo A_j para $j \in \{1, \ldots, n\} - I$, y ρ es un símbolo de predicado correspondiente a R.

O una sentencia más general

`SELECT * FROM` *tabla complemento*

donde *table* $= R$, y $\mathcal{A}_R = \{A_1, A_2, \ldots, A_n\}$. Esta es equivalente dentro del álgebra relacional a:

$$R' = \{\mathcal{V}'_R, \mathcal{A}'_R, \Pi'_R\}$$

tal que:

- $\mathcal{V}'_R$=$\prod_{A\in\mathcal{A}_R}\mathrm{Dom}(A)$
- $\mathcal{A}'_R$=$\mathcal{A}_R$
- Π'_R=$(\Pi_{AR})_{A\in A_R}$

Para encontrar las formas equivalentes en la lógica observamos que para cualquier tabla dentro de la base de datos, $tabla = R$, con atributos $\mathcal{A}_R = \{A_1, A_2, \ldots, A_n\}$, la conjunción

$$\bigwedge\{\rho(a_1, a_2, \ldots, a_n)|\ a_i \in \mathrm{Dom}(A_i)\forall i = 1, \ldots, n\}$$

equivale a la tabla.

Otro ejemplo

`SELECT ALL` *nombre_atrib* `FROM` *tabla complemento*

En este caso, la connotación de `SQL` es tomar en cuenta inclusive repeticiones. Esto no es posible en Lógica, pues $\alpha \wedge \alpha$ es equivalente a α. La traducción que se hace, en este caso, es la de la proposición:

`SELECT * FROM` *tabla complemento*

Como otro ejemplo

`SELECT DISTINCT` *nombre_atrib* `FROM` *tabla complemento*

donde $table = R$, $nombre\_atrib = A_j$ $A_j \in \mathcal{A}_R$. Dentro del álgebra relacional es equivalente a:

$$R' = \{\mathcal{V}'_R, \mathcal{A}'_R, \Pi'_R\}$$

tal que:

- $\mathcal{V}'_R$=$\prod_{A\in\mathcal{A}_R}\mathrm{Dom}(A)$
- $\mathcal{A}'_R$=$\{A_1, ..., A_n\}$
- Π'_R=$(\Pi_{AR'})_{A\in\mathcal{A}'_R}$
- $\Pi_{AR'} : \mathcal{V}'_R \to \mathrm{Dom}(A)$, es una función uno a uno.

Su fórmula equivalente dentro de la lógica es:

$$\bigwedge(\texttt{--}\mathtt{A}_1, \texttt{--}\mathtt{A}_2, \ldots, a_j, \ldots, \texttt{--}\mathtt{A}_n),\ \text{con } a_j \in \mathrm{Dom}(A_j)$$

y a_j único, $\texttt{--}\mathtt{A}_j$ denota variables del tipo A_j para $j = 1, \ldots, n$.

INTERSECT

Consideraremos varias sentencias con la función `INTERSECT`:

`INTERSECT` [`ALL`] `SELECT_statement2`

Sean R_1 y R_2 conjuntos resultantes de la sentencia `SELECT_statement2`. La consulta es equivalente dentro del álgebra relacional a:

$$R' = \{\mathcal{V}'_R, \mathcal{A}'_R, \Pi'_R\}$$

tal que:

- $\mathcal{V}'_R$=$\{u \in \Pi_{A_iR_1} Dom(A_i) \wedge v \in \Pi_{A_iR_2} Dom(A_i))$: $\Pi_{A_iR_1}(u) = \Pi_{A_iR_2}(v)$, $A_i \subset \mathcal{A}_{R_1} \cap \mathcal{A}_{R_2}\}$
- $\mathcal{A}'_R$=$\{A : A \in \mathcal{A}_{R_1} \cap \mathcal{A}_{R_2}\}$
- Π'_R=$(\Pi_{AR_1} \cup \Pi_{A_iR_2})_{A \in \mathcal{A}'_R}$

Ahora bien, suponga que el resultado de la consulta precedente devuelve:

$$\bigwedge(\texttt{--A}_1, a_2, \ldots, a_j, \ldots, a_i, \ldots, \texttt{--A}_n)$$

y `SELECT_statement2` genera

$$\bigwedge(\texttt{--A}_1, a_2, \ldots, a_k, \ldots, a_i, \ldots, \texttt{--A}_n)$$

`INTERSECT` [`ALL`] `SELECT_statement2`.

Su fórmula equivalente dentro de la lógica es

$$\bigwedge(\texttt{--A}_1, a_2, \ldots, a_j, \ldots, a_i, \ldots, \texttt{--A}_n) \cap$$

$$\bigwedge(\texttt{--A}_1, a_2, \ldots, a_k, \ldots, a_i, \ldots, \texttt{--A}_n)$$

UNION

Consideremos la única de varias tablas:

`UNION` [`ALL`] `SELECT_statement2`

Sean R_1 y R_2 conjuntos resultantes de la consulta `SELECT_statement2`. La sentencia inicial es equivalente dentro del álgebra relacional a:

$$R' = \{\mathcal{V}'_R, \mathcal{A}'_R, \Pi'_R\}$$

tal que:

- $\mathcal{V}'_R$=$\{\Pi_{A_iR_1} Dom(A_i) \cup \Pi_{A_iR_2} Dom(A_i)\}$

- $\mathcal{A}'_R$=$\{A : A \in \mathcal{A}_{R_1} \cup \mathcal{A}_{R_2}\}$
- Π'_R=$(\Pi_{A_i R_1})_{A_i \in \mathcal{A}_{R_1}} \cup (\Pi_{A_i R_2})_{A_i \in \mathcal{A}_{R_2}}$

Suponga que la consulta precedente genera:

$$\bigwedge(\texttt{--A}_1, a_2, \ldots, a_j, \ldots, a_i, \ldots, \texttt{--A}_n)$$

y `SELECT_statement2` genera

$$\bigwedge(\texttt{--A}_1, a_2, \ldots, a_k, \ldots, a_i, \ldots, \texttt{--A}_n)$$

`UNION` [`ALL`] `SELECT_statement2`

Entonces, esta consulta es equivalente dentro de la lógica a la fórmula:

$$\bigwedge(\texttt{--A}_1, a_2, \ldots, a_j, \ldots, a_i, \ldots, \texttt{--A}_n) \cup$$

$$\bigwedge(\texttt{--A}_1, a_2, \ldots, a_k, \ldots, a_i, \ldots, \texttt{--A}_n)$$

INSERT INTO

Consideremos algunas formas de esta consulta y sus correspondientes traducciones dentro de la lógica de primer orden.

Consulta en su forma general `INSERT INTO` *tabla* [*atributo*] `VALUES` *attr_valor*

donde $table = R$, $atributo = A_j$ $A_j \in \mathcal{A}_R$, $attr\_valor = a_j$ $a_j \in Dom(A_j)$.

Suponga $atributos = A_2, A_3, A_4$ y $attr\_valor = a_2, a_3, a_4$.

La consulta anterior es equivalente dentro del álgebra relacional a:

$$R' = \{\mathcal{V}'_R, \mathcal{A}'_R, \Pi'_R\}$$

tal que:

- $\mathcal{V}'_R$=$\mathcal{V}_R \bigcup \Pi_{A_i R}$, $i = 1, \ldots, n$
- $\mathcal{A}'_R$=$\mathcal{A}_R$
- Π'_R=$(\Pi_{A_i R})_{A_i \in A'_R} \bigcup \Pi_{A_i R}$

La equivalencia lógica es la representación de la tabla concatenada con el nuevo registro $(\texttt{--A}_1, a_2, a_3, a_4, \ldots, \texttt{--A}_n)$.

Como otro ejemplo tenemos la siguiente consulta, que selecciona varios registros y los inserta en una nueva tabla.

`SELECT` *atributos* `INTO` R_n (tabla_nueva) `FROM` R_o (tabla_orig) [`WHERE` *criterio*]

donde $atributos = A_i$ tal que $A_i \subset \mathcal{A}_{R_o}$ es equivalente a:

$$R_n = \{\mathcal{V}_{R_n}, \mathcal{A}_{R_n}, \Pi_{R_n}\}$$

tal que:

- $\mathcal{V}_{R_n}$=$\{v \in \Pi_{A_i R_o} Dom(A_i) : criterio(v) = true\}$
- $\mathcal{A}_{R_n}$=A_i
- Π_{R_n}=$(\Pi_{AR_o})_{A \in A_i}$

Cuya equivalencia lógica es $\bigwedge(a_1, a_2, a_3, \ldots, a_j)$ si $A_i = \{A_1, A_2, A_3, \ldots, A_j\}$

DELETE

Consideremos la sintaxis de la consulta `DELETE`

La consulta en su forma general

`DELETE FROM` *tabla*[`WHERE` *criterio*]
donde $tabla = R$

Equivalente dentro del álgebra relacional a:

$$R' = \{\mathcal{V}'_R, \mathcal{A}'_R, \Pi'_R\}$$

tal que:

- $\mathcal{V}'_R$=$\emptyset$
- $\mathcal{A}'_R$=$\emptyset$
- Π'_R=$\emptyset$

La equivalencia lógica es la negación de la tabla.

Sea $\mathcal{A}_R = \{A_1, \ldots, A_n\}$ el conjunto de atributos dentro de R si $(a_1, \ldots, a_n) \in R$, con $a_j \in Dom(A_j)$ $j = 1, \ldots, n$, entonces $\neg(a_1, \ldots, a_n) \in R'$. R' será la tabla equivalente.

Esta conversión está restringida a los comandos (`SELECT, INSERT INTO` y `DELETE`).

Una especificación común para varios comandos dentro de `SQL` es `WHERE` la cual trataremos a continuación:

WHERE

Enunciado de la consulta `WHERE`
`WHERE` *criterio*

La sintaxis de *criterio* es la siguiente:

[(]*nombre_atrib*[`NOT`] < | <= | = | => | > |`IS` [`NOT`]`NULL`|`IN`|`LIKE`|`BETWEEN` *valor*1 [`AND`*valor*2][`ESCAPE` *esc_mark*]][(*valores*)][)] [`AND`|`OR`][*criterio*]

Ésto es equivalente dentro del álgebra relacional a:

$$R' = \{\mathcal{V}'_R, \mathcal{A}'_R, \Pi'_R\}$$

tal que:

- $\mathcal{V}'_R$=$\{v \in \Pi_{A_{iR}} : criterio(v) = true\}$
- $\mathcal{A}'_R$=$\mathcal{A}_R$
- Π'_R=$(\Pi_{AR})_{A \in \mathcal{A}'_R} \bigcup \Pi_{A_iR}(x) = v_i,\ i = \{1, \ldots, n\},\ \forall x \in \mathcal{V}_R$

Sintácticamente equivalente a:

[`NOT`]*nombre_atrib* < | <= | = | => | > |`ISNULL`|`IN`|`LIKE`|`BETWEEN` *valor*1[)]
[[`NOT`(](*valor*1 < *nombre_atrib*)`AND`(*nombre_atrib* < *valor*2]
[`ESCAPE` *esc_mark*]][(*valores*)][)][`AND`|`OR`][*criterio*]

3.3.4. Conversión de LPO a SQL

En esta sección consideramos fórmulas bien formadas en la lógica de primer orden (LPO) para ser traducidas a sentencias propias de `SQL`.

Suponga que se tiene una base de conocimiento como una colección de fórmulas de primer orden.

Para cada símbolo de función ξ se define una relación de dependencia, y para cada término $\tau_k \equiv \xi(\tau_0, \ldots, \tau_{k-1})$ definiremos una constante c_{τ_k} denominada *constante de Skolem*. Cada aparición del término $\xi(\tau_0, \ldots, \tau_{k-1})$, dentro de la base de conocimiento, será reemplazado por τ_k y se agregará a la base de datos la dependencia relacional $\xi(\tau_0, \ldots, \tau_{k-1}) \rightarrow \tau_k$.

A cada símbolo de relación ρ le será asociado la relación $R(\rho)$. Un Procedimiento de Unificación [12] ordinario sobre todos los átomos correspondientes a ρ nos permitirá determinar:

- la aridad de la relación $R(\rho)$, u
- el tipo de cada argumento dentro de ρ.

La lista de los tipos de argumento de la relación $R(\rho)$ nos proveerá su esquema relacional.

Los conectivos booleanos son traducidos directamente al conjunto de operadores de equivalencia teórica, es decir, la conjunción será equivalente a la intersección, la disyunción equivaldrá a la unión y la negación a la función complemento.

El cuantificador existencial será equivalente a la función de proyección y el cuantificador universal equivaldrá a verificar si la proyección de un argumento dado bajo una relación coincide con el dominio del tipo de variable cuantificada.

De esta manera, para cualquier término τ dentro de la lógica le corresponde una constante c_τ, y para cualquier fórmula en primer orden ϕ, en la lógica de primer orden, le corresponde una expresión relacional R_ϕ tal que:

$$\text{LPO} \vdash \phi(\tau_0, \ldots, \tau_{k-1}) \iff (c_{\tau_0}, \ldots, c_{\tau_{k-1}}) \in R_\phi.$$

Por otra parte, a cualquier metateorema de la lógica de la forma:

$$\{\phi_0, \ldots, \phi_{k-1}\} \vdash \phi_k$$

se le asocia la expresión relacional de la forma $\bigcap_{j=0}^{k-1} R_{\phi_j} \subseteq R_\phi$, la cual corresponde a un enunciado de `SQL`.

Todas los conversiones descritas son realizadas procedimentalmente.

Capítulo 4

Ingeniería de Software para el Sistema Desarrollado

4.1. Obtención de Componentes para el Sistema Lógico

Como trabajo a futuro, en busca de deducir información de manera eficiente dentro de la lógica de primer orden, es necesario extraer apropiadamente la información presente en la base de datos, identificar adecuadamente el rol que jugará dicha información dentro del sistema lógico y especificar el cómo se deberá manejar dicha información. Por ejemplo, identificar qué información compondrá el conjunto de átomos y definir el cómo se operará sobre ellos.

Los átomos son el componente fundamental del sistema, sobre ellos estará definida nuestra lógica.

El paso inicial para encontrar los átomo es determinar los elementos que componen la *signatura*.

Toda lógica de primer orden está construida sobre una signatura, que es definida como la tripleta

$$\sigma = (S_{func}, S_{rel}, S_{const})$$

donde,S_{func} representa el conjunto de funciones, S_{rel} el conjunto de relaciones y S_{const} el conjunto de constantes. Por lo tanto, es preciso implementar un programa para identificar los elementos que conforman la signatura. Sea A el conjunto de atributos presentes en la base de datos:

- El conjunto de constantes S_{const} está formado por el conjunto de valores que puede tomar cada uno de los atributos, esto es:

$$S_{func} = \{k : k \in Dom(A), \forall A \in\}$$

- El conjunto de relaciones S_{rel} está compuesto por cada uno de los atributos presentes en la base de datos:

$$S_{rel} = \{A : A \in \mathcal{A}\}$$

- El conjunto de funciones S_{func} está integrado por todas las opciones definidas por el usuario.

El programa permite identificar las relaciones, funciones y constantes. Con respecto a las relaciones y funciones el programa devuelve el nombre de la relación o función, la aridad y el tipo de cada parámetro de la relación; en caso de las funciones añade el dominio de la función. El programa permite la creación tanto de nuevas relaciones como de nuevas funciones.

En caso de las constantes el programa las reconoce explícitamente siempre y cuando la cardinalidad del dominio del atributo sea relativamente pequeña. En caso de que la cardinalidad sea más grande y se trate con tipos alfanuméricos, el programa describe la gramática que describe a dichas constantes. Si las constantes son número enteros o reales, se da el conjunto o rango donde caen los valores de las constantes.

En resumen se requiere que dada una base de datos, el programa sea capaz de reconocer y extraer las funciones y las constantes ahí presentes, y permita la definición de funciones referentes a la base de datos.

Una vez obtenida la signatura se procede al análisis sintáctico, en busca de obtener el conjunto de átomos P_0 sobre el que se definirá nuestra lógica. Para hacer el análisis sintáctico se procede de la siguiente manera:

Sea A el conjunto de atributos presente en la base de datos, definimos:

- Constantes: $\forall \delta \in$ A, si $c \in Dom\langle\delta\rangle$, δ es una constante ($\delta \in Const$).
- Términos:
  - Si $\delta \in Var \cup Const$, entonces δ es un término ($\delta \in Terms$).
  - Si f es una función, ($f \in S_{func}$) y $\alpha_0, ..., \alpha_n \in Terms$, entonces $f(\alpha_0, ..., \alpha_n) \in Terms$.
- Átomos: Si f es una relación, ($f \in S_{rel}$) y $\alpha_0, ..., \alpha_n \in Terms$, entonces $f(\alpha_0, ..., \alpha_n) \in Atomos$.

El conjunto Var es el conjunto de variables previamente definidas.

Así hemos encontrado el conjunto finito de átomos sobre el cual se construye nuestra lógica. Ahora podemos definir, bajo las siguientes reglas, una fórmula bien formada:

- $\mu \in Atoms \Rightarrow \mu \in Formulas$
- $\mu, \varphi \in Formulas \Rightarrow \mu \boxplus \varphi \in Formulas$, donde $\boxplus = \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$

- $\mu \in Formulas \Rightarrow \neg\mu \in Formulas$

De esta manera hemos obtenido, partiendo de la información presente en la basade de datos, el conjunto de átomos adecuados para nuestra lógica y hemos definido también la sintaxis de una fórmula bien formada dentro de la misma.

Por otro lado, para establecer la comunicación entre el gestor de la base de datos y el sistema lógico, es necesario hacer la traducción de consultas `SQL` a fórmulas bien formadas, dentro de la lógica de primer orden, y de fórmulas a consultas `SQL`.

Se diseñaron dos programas, uno de ellos recibirá como entrada una fórmula bien formada y devolverá la consulta equivalente en `SQL`, el segundo recibirá como entrada una sentencia `SQL` y devuelve su fórmula lógica equivalente.

4.2. Incorporación de la LPO con el Demostrador Automático

Una vez que la información presente en la base de datos ha sido traducida en términos lógicos, es posible operar sobre ella con algunos de los demostradores automáticos existentes en el mercado. A continuación presentamos el estado en el que se encuentra la información de la base de datos en términos de nuestra lógica, los sistemas de deducción automática que encontramos más apropiados para nuestro trabajo y el camino que se siguió para su acoplamiento.

La información obtenida de la base de datos será tomada como axiomas para nuestro demostrador y la información a verificar será la conjetura a demostrar. Los átomos dentro de la lógica de primer orden extraídos de la base de datos, tienen la forma:

$$tablai(a_1, ..., a_n)$$

donde $a_i \in Dom(A_i)$. Sea A_{tablai} el conjunto de todos los atributos presentes en la tabla $tablai$, $A_i \subseteq A_{tablai}$.

En principio se deberá alimentar al demostrador automático con información de la base de datos proveniente de alguna consulta. De la consulta se obtiene información del tipo:

$$:: tablai(a_{1_1}, ..., a_{1_n}) AND :: tablai(a_{2_1}, ..., a_{2_n}) AND ... AND :: tablai(a_{k_1}, ..., a_{k_n})$$

como se mencionó anteriormente, se desea aplicar los axiomas y reglas de inferencia sobre este conocimiento extraído de la base de datos, conjunto que ha de ser finito.

Podrá ser necesario transformar una fórmula en otra para aplicar las reglas de inferencia, esto se logra transformando la fórmula en otra equivalente.

$$(\alpha \rightarrow \beta) \leftrightarrow (\neg\alpha \vee \beta)$$
$$(\alpha \leftrightarrow \beta) \leftrightarrow [(\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha)]$$

Así se obtiene un conjunto más grande de proposiciones que podrán ser utilizadas para llegar a la proposición a analizar.

Si es posible llegar a una proposición o descubrir una proposición dentro de la lógica de primer orden, será posible también llegar a dicha proposición vista como una fórmula en el ámbito del álgebra relacional o como el resultado de una consulta si se tratase desde el punto de vista de `SQL`.

Dentro de los demostradores existentes que se acoplan a los requerimientos se tienen:

`Prover9` [13] es un demostrador automático para la lógica de primer orden, se considera el sucesor de `Otter prover`. `Prover9` tiene un modulo de automatización completo en el cual el usuario introduce las fórmulas que representan el problema.

Define internamente término, fórmula atómica, literal, cláusula y fórmula desde el punto de vista de la lógica. La diferencia primordial entre `Otter` y `Prover9` es que en `Otter`, las cláusulas y las fórmulas son tipos distintos mientras que `Prover9` trata a las cláusulas como subconjuntos de fórmulas.

Las clases son un subconjunto de fórmulas. Todas las fórmulas de entrada, incluyendo las cláusulas, aparecen en una lista encabezada por `formulas(list_name)`.

La regla por defecto para distinguir variables de constantes, es que las variables comiencen con minúsculas de la `u` a la `z`. Por ejemplo, en la fórmula `P(a,x)`, el término `a` es una constante y `x` es una variable. Las reglas de inferencia de `Prover9` operan sobre las cláusulas. Si la entrada no es una cláusula, `Prover9` la transforma mediante su conversión a fórmula normal conjuntiva (FNC).

La forma prefija estándar de un objeto con aridad n, `f` sigue:

$$f(argumento_1, ..., argumento_n)$$

Los espacios en blanco son admitidos en cualquier lugar excepto entre símbolos.

A continuación se describe cada uno de los elementos del demostrador.

- Símbolos: incluyen variables, constantes, símbolos de funciones, símbolos de predicados, conectivos lógicos. No incluyen los paréntesis ni comas. `Prover9` reconoce varios tipos de símbolos, símbolos ordinarios, especiales y una lista de símbolos vacía. Los objetos (términos y fórmulas) son construidos a partir de símbolos, paréntesis y comas.

- Términos: Cualquier término puede escribirse en forma prefija, por ejemplo, `f(g(x),y)`. Una noción de lista similar a la de `Prolog` puede ser usada para escribir términos que representen listas.

- Fórmulas atómicas: El símbolo binario de igualdad (=) es usualmente escrito como una relación infija. El símbolo de desigualda binario (!=) es una abreviación de la "no igualdad". Es decir la fórmula `a!=b`, es equivalente a ¬`(a=b)`, o de manera más precisa, ¬`(=(a,b))`. Desde el punto de vista sintáctico, el

símbolo binario = es el único símbolo de igualdad para las reglas de inferencia que usan igualdad.

- Cláusulas: El símbolo de disyunción usado es |, el de negación es −. El símbolo de disyunción tiene precedencia mayor al símbolo de igualdad por lo tanto se vuelven innecesarios los paréntesis para las ecuaciones que se encuentran dentro de las cláusulas. Cada cláusula termina con un punto.

- Fórmulas

Significado	*Conectivo*	*Ejemplo*
Negación	$-$	$(-q)$
Disyunción	$\mid$	$(q \mid p)$
Conjunción	&	$(q\&p)$
Implicación	$->$	$(q->p)$
Implicación inversa	$<-$	$(q<-p)$
Equivalencia	$<->$	$(q<->p)$
Cuantificador Universal	all	$(all\ x\ p(x))$
Cuantificador Existencial	$exists$	$(exist\ x\ p(x))$

Cuando `Prover9` hace la demostración de un teorema, despliega la información de la prueba en un archivo bajo un formato estándar, el cual contiene la justificación de cada paso deductivo con suficiente detalle para reestructurar o comprobar la demostración. Estructura del archivo de entrada, *Nombre_Archivo.in*:

- Asignación de tiempo, de constantes y de peso.
  assign(max_seconds, 30).
  assign(new_constants, 1).
  assign(max_weight,25).

- Declaración de fórmulas:
  formulas(sos).
  ...
  end_of_list.

- Declaración de metas:
  formulas(goals).
  ...
  end_of_list.

Estructura básica del archivo de salida:

- `Prover9/end of head`: Versión, fecha, host, comandos.

- `INPUT/end of input`: Aquí se encuentra una copia de la entrada. Todo lo que no sea parte de la entrada se encuentra comentado (%) por lo que la información aquí presente puede usarse para crear un nuevo archivo de entrada.

- `PROCESS GOALS/end of process goals`: La búsqueda de la respuesta se da siempre por refutación, por lo que en esta parte se muestra cómo son negadas las metas en preparación para la búsqueda.

- `PROCESS INITIAL CLAUSES`: Esta sección muestra las cláusulas iniciales (después de aplicar la Skomemization) para luego mostrar algo de lo que `Prover9` hace para preparar la búsqueda. Ésto incluye *predicate_elim*, la toma de decisiones para llevar acabo el ordenamiento, los ajustes $auto\_inference$. En esta etapa, las cláusulas pueden ser borradas mediante la subsumición y las ecuaciones pueden ser copiadas a la lista *demulators*. Se deberá checar la bandera *process_initial_sos* para saber si ha habido un cambio.

- `CLAUSES FOR SEARCH/end of clauses for search`: Esta sección muestra las cláusulas justo antes de empezar la búsqueda, esto es, justo antes de la selección de la primera cláusula dada.

- `SEARCH/end of search`: Esta sección típicamente muestra la secuencia de las cláusulas dadas y permite incluir las secciones `PROOF` y `STATISTICS`.

  - `PROOF/end of proof`: Una demostración en un formato estándar.
  - `STATISTICS/end of statistics`: Muestra estadísticas de la demostración.

SPASS

- [14] Demostrador de teoremas basado en lógica de primer orden, también puede ser usado como un analizador formal de software, sistemas y protocolos. También sirve como una aproximación formal a la planeación de procedimientos de decisión, de demostradores de teoremas basados en lógica modal y de diseños en inteligencia artificial.

  La entrada del demostrador es una fórmula de primer orden escrita en la sintaxis propia del sistema. La ejecución de SPASS puede arrojar dos posibles resultados: `Proof found` si la fórmula es válida y `Completion found` si la fórmula no es valida o si es indecidible mediante la lógica de primer orden.

- Sintaxis del archivo de entrada:

```
begin problem(Nombre).
list of descriptions.
name({*Nombre*}).
author({*Nombre Autor*}).
status().
description([*Descripcion*]).
end of list.
list of symbols.
```

```
      functions[(nombre funcion,cardinalidad)].
      predicates[(nombre pred1,aridad)...(nombre predn,aridad)].
end of list.
list of formulae(axioms).
      formula(nombre predi(nombre funcion)).
      formula(forall([x],implies(...))).
end of list.
list of formulae(conjetures).
      formula(form demostrar).
end of list.
end problem.
```

Como se puede observar, éste consiste de tres partes. La primera parte describe el programa como el nombre del autor, el nombre del programa, estatus, y es posible hacer una breve reseña sobre lo que se quiere demostrar. La segunda parte es la especificación de símbolos, se requiere especificar todas las funciones y predicados que se estarán utilizando. La tercera y última parte es la especificación de fórmulas la cual se divide en dos, una parte de axiomas y otra parte de conjeturas que se quieren demostrar, las fórmulas son siempre escritas en notación prefija.

- `SPASS` [14] trata de demostrar que la conjunción de todas las fórmulas especificadas como axiomas implican la disyunción de todas las conjeturas. Una vez que `SPASS` ha analizado el problema puede regresar alguna de las siguientes especificaciones:
  - `This is a monadic Horn problem without equality.`
  - `This is a problem that has, if any, a finite domain model.`
  - `There are no function symbols.`
  - `This is a problem that contains sort information.`
  - `The conjecture is ground.`
  - `The following monadic predicates have finite extensions: algún_predicado.`
  - `Axiom clauses: 2 Conjecture clauses: 1`

E

- `E` [15] es un demostrador automático de teoremas para lógicas de primer orden con igualdad. Acepta la especificación de problemas, los cuales consisten típicamente por un conjunto de cláusulas o fórmulas en primer orden, y una conjetura, de igual manera escrita como una cláusula o fórmula en primer orden. El sistema trata de encontrar una demostración formal para la conjetura a partir de los axiomas especificados.

El desarrollo de `E` comenzó como parte del proyecto `E-SETHEO`. Su primer aparición pública fue en 1998, siendo actualizado y mejorado continuamente desde entonces. Se cree que `E` es, dentro de los sistemas de razonamiento sobre lógicas de primer orden, el más poderoso y amigable para el usuario. Ha participado en numerables competencias obteniendo resultados satisfactorios.

El demostrador `E` viene acompañado de una multitud de opciones, muchas de ellas afectan el resultado en la demostración e incluso, interactúan sobre la demostración de manera inesperada. La manera más fácil de obtener un buen desempeño es el ejecutarlo en el modo *automático*.

Por defecto, `E` detendrá la búsqueda después de la primer respuesta, la opción `--answers` puede ser usada para especificar después de cuantas respuestas `E` deberá detener la búsqueda.

- Si se encuentra una demostración para la fórmula inicial, el sistema es capaz de proveer una lista detallada de los pasos que se siguieron en la demostración, con el fin de que éstos sean verificados manualmente. Si la conjetura es existencial (es decir, con la forma "existe X que satisfaga la propiedad P"), la ultima versión de este demostrador devuelve las posibles respuestas (los valores de X).

4.3. Detalles de la Implementación

Para poder establecer una correcta comunicación entre la base de datos y el demostrador automático es necesario traducir la información presentada en base al sistema lógico que hemos desarrollado a información legible para el demostrador seleccionado.

Enseguida se expone el procedimiento seguido y las implementaciones requeridas.

4.3.1. Tipos de Cláusulas

A cada cláusula dentro del archivo se le asocia un identificador y una *justificación* que puede referirse a los identificadores de otras cláusulas.

Una justificación es una lista consistente de un *paso primario* y algunos *pasos secundarios*. En su mayoría, los pasos primarios son reglas de inferencia aplicadas a cláusulas dadas, y la mayoría de los pasos secundarios consisten en simplificaciones, reescritura o igualdades.

La mayoría de los tipos de pasos se refieren a posiciones de las literales o términos dentro de las cláusulas.

- Las literales son identificadas por los caracteres 'a' (primera literal), 'b' (segunda literal), etc.

- Los términos son identificados mediante un identificador de literal ('a','b', ...,) seguido de una secuencia de enteros que determinan la posición del término dentro del literal. Por ejemplo, la posición `c,4,2` se interpreta como el cuarto y el segundo argumento de la tercer literal.

Interpretación de las palabras reservadas asociadas a los pasos primarios A continuación se enlistan las palabras reservadas usadas en los pasos de demostración y la interpretación de cada una de ellas.

`assumption` denota a las fórmulas de entrada.

`clausify` equivalencia en forma normal conjuntiva de cláusulas no axiomáticas.

`goal` fórmula de entrada a demostrar.

`deny` negación de alguna fórmula a demostrar (goal) expresada en forma normal conjuntiva(FNC).

`resolve` Expresa la manera en la que se obtuvo la fórmula precedente. Ejemplo `resolve(59,b,47,c)` es traducida como que se ha resuelto la segunda literal de la cláusula 59 con la tercer literal de la cláusula 47.

`hyper` interpreta la lista como un identificador de clase mediante la tripleta ⟨ `literal,clause-ID,literal`⟩, la inferencia es presentada como una secuencia de pasos de resolución binaria. Ejemplo `hyper(59, b,47,a, c,38,a)` se parte de la cláusula 59, después resuelve la literal 'b' con el uso de la cláusula 47 en la literal 'a'; con el resultado del primer paso, se resuelve la literal 'c' mediante la cláusula 38 en la literal 'a'. Se tiene un caso especial, "xx", significa establecer la resolución con x=x.

`ur` La lista de comandos es interpretada al igual que con la palabra clave `hyper`, denota la unidad resultante.

`para` La *paramodulación* es una regla de inferencia que hace el trabajo de todos los axiomas de igualdad, salvo la reflectividad. Reemplaza subtérminos iguales en las cláusulas. El ejemplo `para(47(a,1),28(a,1,2,2,1))` es interpretado como que se realizo la paramodulación de la cláusula 47 a la cláusula 28 en la posición mostrada.

`copy` Denota que es el resultado de copiar una cláusula.

Ejemplo `copy(59)`, copia la cláusula 59.

`back_rewrite` Denota que es el resultado de copiar una cláusula.

Ejemplo `back_rewite(59)`, copia la cláusula 59.

`back_unit_del` Denota que es el resultado de copiar una cláusula.

Ejemplo `back_unit_del(59)`, copia la cláusula 59.

`new_symbol` Introduce una nueva constante.

Ejemplo `new_symbol(59)`, nueva constante con identificador 59.

`factor` Unificación de literales.

Ejemplo `factor(59,b,c)`, se tiene la cláusula 59 mediante la unificación de la segunda y tercer literal.

`xx_res` Resuelve mediante la igualdad x=x.

Ejemplo `xx_res(59,b)` resuelve la segunda literal de la cláusula 59 con x=x.

A continuación se presentan palabras reservadas no utilizadas en demostraciones estándar.

- `propositional`
- `instantiate`
- `ivy`

Enseguida describiremos las palabras reservadas de los *pasos secundarios* y su interpretación. Cada uno supone una cláusula dentro de la demostración, que es el resultado de un paso primario o de pasos secundarios previos. Con el fin de clarificar su interpretación hacemos uso de ejemplos.

flip Ejemplo `flip(c)` la tercer literal es una igualdad que ha sido invertida.

merge Ejemplo `merge(d)` la cuarta literal ha sido eliminada por ser idéntica una literal precedente.

unit_del Ejemplo `unit_del(b,38)` la segunda literal ha sido eliminada por ser una instancia de la negación de la cláusula 38 (la cual es una cláusula unitaria).

xx Ejemplo `xx(b)` la segunda literal ha sido eliminada por ser una instancia x!=x.

rewrite Ejemplo `rewrite([38(5,R),47(5),59(6,R)])` reescritura (demodulación) con las ecuaciones 38, 47, luego 59, los argumentos (5), (5) y (6) determinan las posiciones de los subtérminos reescritos (de un modo oscuro), y el argumento `R` indica que el demodulador es utilizado hacia atrás (de derecha a izquierda).

4.4. Interfaz Propia

Para realizar las pruebas pertinentes a este trabajo decidimos usar el demostrador `Prover9`, en busca de lograr una comunicación ideal de la información presente en la base de datos con el demostrador nos fue preciso realizar las siguientes tareas:

- Programa de conversión de archivos `.out` a `SQL`
- Programa *piping* de `SQL` hasta la generación de archivos `.in`
- Ejecución del demostrador en el brackground: conversión del `.in` al `.out`
- Piping de la conversión de archivos `.out` hasta `SQL`.
- Ensamblaje automático de los tres puntos anteriores.

Todo lo anterior fue englobado en un programa que maneja tres procesos principales:

- Conversión de archivos `SQL` a `.in`
- Ejecución del demostrador en el brackground: conversión del `.in` al `.out`
- Conversión de archivos `.out` hasta `SQL`.

La primera parte del programa solicita al usuario introducir la consulta a realizar. Dicha consulta deberá estar hecha bajo la sintaxis de `SQL`, por lo cual, exponemos la sintaxis de `SQL` al usuario de manera breve.

```
Introduzca la consulta bajo las siguientes reglas

DELETE FROM tabla [WHERE criterios]
CREATE TABLE tabla atributo tipo
INSERT INTO tabla [atributos] VALUE valores
INSERT INTO tabla1 SELECT FROM tabla2 atributos VALUE valores
SELECT [DISTINCT|ALL|*|atributos]
        FROM tabla [ WHERE where criteria]
[INTERSECT|UNION[ ALL] SELECT enunciado]
        WHERE [(]atributo {[NOT] <|<=|=|=>|>|IS[ NOT ]
                    NULL|IN|LIKE|BETWEEN} value1 [AND value2]
                            [ESCAPE esc mark]
                    [({values})] [)AND(|)OR(] [ condition][)]}
```

Una vez obtenida la *consulta*, el programa despliega la opción de elegir entre dos demostradores, `Prover9` o `Spass`, bajo que demostrador quiere trabajar, aunque únicamente hemos programado la conversión completa de archivos para el demostrador `Prover9`.

Posteriormente solicita al usuario establecer la conjetura o las conjeturas a demostrar, las cuales deberán satisfacer la sintaxis expuesta.

```
Por demostrar:
((NomAtrib ValAtrib)+ Tabla)
```

El programa se encargará de traducir la información resultante de la consulta introducida a axiomas del demostrador elegido y la sentencia a demostrar será a su vez traducida a una sentencia propia del demostrador. Todo esto siguiendo las reglas de sintaxis de los archivos de entrada para cada demostrador.

Si se ha elegido el demostrador `Prover9`, se crea un objeto de la clase denominada `Prover9` y manda a llamar a la función `escribir_archivo`, la cual recibe en sus parámetros de entrada la consulta deseada y la conjetura. La función `escibe_archivo` llama a su vez a varias funciones.

El archivo obtenido por `escribir_archivo` será el archivo de entrada al demostrador `Prover9`. La nomenclatura del archivo de salida sigue el siguiente formato `/home/ely/Dropbox/Programasjava/SQLLES05/consultaProver9cons.in`, tal que `cons` contiene la consulta introducida por el usuario sin espacios.

Lo que contendrá el archivo `.in` será la sintaxis propia de `Prover9`:

- Una primer sección donde se especifica el máximo de tiempo para resolver la demostración, la cantidad de nuevas constantes que se manejan en la demostración y el peso máximo de la demostración. En nuestro caso utilizamos las medidas estándar.

- En la segunda sección se describen las fórmulas obtenidas de la consulta en forma de lista, las cuales son obtenidas mediante la función `examina_consulta`.

- A continuación tenemos la sección donde se establecen las fórmulas a demostrar, estas fórmulas serán obtenidas por la función conjetura que tendrá como parámetro de entrada las conjeturas a demostrar.

Una vez obtenido el `archivo.in`, el mismo objeto manda a llamar al demostrador. Éste recibe como parámetro de entrada un archivo, que contiene tanto la información resultante de la consulta, es decir la información expresada en términos de fórmulas legibles para el demostrador, como la conjetura que se desea demostrar, mediante `prover9 -f archivo.in`, tal comando ejecuta el demostrador `Prover9` para el archivo `archivo.in` y el resultado es escrito en un archivo denominado `archivo.out`.

El programa pregunta al usuario si desea la interpretación del archivo de salida. Esta opción es disponible sólo sí se ha elegido hacer la demostración mediante el demostrador `Prover9`.

En caso de requerir la interpretación, la clase `outles` entra en función. Dicha clase se encarga de todo lo que conlleva la interpretación del archivo de salida del demostrador. El objeto de la clase `outles` activa la función `leerarchivo`.

El resultado de analizar e interpretar el archivo de salida del demostrador será escrito en un archivo con la extensión `.itp`, bajo la siguiente nomenclatura `/home/ely/Dropbox/Programasjava/SQLLES05/consultaProver9cons.itp`, e igual que en el caso anterior, `cons` será la constante introducida por el usuario sin espacios.

En la clase `outles` se manda a llamar a la función `leerarchivo`. Esta función se encarga de examinar el archivo .out en busca de la demostración y de los pasos que se siguieron para la demostración. Una vez obtenidos los pasos de la demostración cada paso es analizado y reinterpretado en términos entendibles para el usuario .

En caso de elegir el demostrador `Spass`, se crea un objeto de la clase `Spass` mediante el cual se llama a la función `escribir_archivo`, ésta hace uso de la consulta y la conjetura introducidas por el usuario para poder obtener un archivo que sirva como entrada al demostrador `Spass`.

Capítulo 5

Resultados Experimentales

Dentro de los resultados relevantes se tienen la programación de reconocedores de los componentes de la lógica de primer orden, es decir, reconocedores de fórmulas bien formadas, la programación de operaciones booleanas sobre fórmulas bien formadas, la evaluación de dichas fórmulas, y la representación de un fórmula bien formada en su correspondiente árbol sintáctico.

El modelado de la base de datos en una lógica de primer orden, aunado a ello tenemos la interpretación de la información presente en la base de datos y la traducción de las consultas a fórmulas bien formadas de la lógica desarrollada. También, tenemos el desarrollo del sistema de comunicación a un demostrador automático y el descifrado del resultado para hacer éste entendible al usuario.

La traducción de una consulta a la base de datos se puede hacer de dos formas; la primera es hacer la traducción de la tabla y luego especificar mediante una fórmula bien formada qué clausulas estarán presentes en el archivo respuesta, y la segunda, es realizar la consulta sobre la base de datos y aplicar la traducción directamente a la tabla que el manejador de la base de datos devuelve como respuesta de la consulta. Es de interés saber cuál de estos métodos resulta más eficiente, posteriormente se muestra una tabla comparativa de los resultados obtenidos en este aspecto.

Una vez obtenida la información de la base de datos en términos de la lógica de primer orden se podrá obtener información mediante las reglas de inferencia propias de la lógica, por tal caso es preciso el uso de un demostrador. Buscando en la bibliografía encontramos tres demostradores funcionales para nuestro propósito, el sistema `E`, `Prover9` y `SPASS`. El demostrador más ad hoc es el `Prover9`, ésto con respecto a la presentación de la información aunque tuvimos que adecuar la salida de las consultas para que sirvieran de entrada al demostrador, también se implementó la traducción del archivo de salida del demostrador a un lenguaje entendible por el usuario.

5.1. Caso de Estudio

5.1.1. Presentación de la Base de Datos

El caso de estudio que trataremos a continuación se toma como uno en particular, lo exponemos con el fin de mostrar el trabajo de traducción e interpretación que hemos realizado. Por esta razón, la presentación de la base de datos es sucinta y no se ahonda en ella.

La base de datos sobre la cual se están haciendo las pruebas es una base de datos de tipo geográfico . Contiene información referente a localidades dentro de la República Mexicana; contiene tablas que describen a las localidades, su ubicación, la cantidad de habitantes de cada lugar, la cantidad de viviendas, las vialidades, nombres de las vialidades más importantes, entre otras. Esta base de datos tiene un tamaño de 26624 Kbytes y fue obtenida mediante el `INEGI`. Con el fin de poner en contexto la información posterior describiremos algunas de las clausulas presentes en la base de datos, sin dejar de hacer notar que la base de datos con la que se trabajó puede ser sustituida por cualquier otra. Por el momento damos una traducción arbitraria.

Una *Región* dentro de la base de datos está definida por la tupla $(CVE_ENT, CVE_MUN, CVE_LOC)$, ésto es la clave de la entidad, la clave del municipio y la clave de la localidad. Más adelante se hace uso de algunos registros presentes en la base de datos, con el fin de poner en contexto dichos registros, a continuación se enlistan y describen.

- `TIPO`:Tipo de localidad, rural o urbana, consta de un carácter R o U.
- `LONGITUD`: Describe la ubicación de la región con respecto al meridiano de Greenwich. Son descritos por una cadena.
- `LATITUD`: Describe la ubicación de la región con respecto al ecuador. Igualmente, descrita mediante una cadena.
- `POB_TOTAL`: Total de población que habita la región descrita en el registro. El dominio de este registro cae en los número naturales.
- `TOT_VIV`: Total de viviendas dentro de la región descrita por el registro. El dominio, al igual que el en el caso anterior, son los números naturales.

5.1.2. Signatura

Como se mencionó anteriormente, con el fin de adentrarnos a los sistemas lógicos, programamos reconocedores de los componentes de la lógica y las diversas formas de representación de una proposición (enfijo, prefijo, árbol sintáctico) y la construcción de las tablas de verdad.

Las funciones `fb_enfijo` y `fb_prefijo` reciben una proposición y determinan si son fórmulas bien construidas en forma enfija o prefija, respectivamente. Para ello se debieron definir previamente los átomos y los conectivos lógicos. Suponga que se tiene la

fórmula `f1=['then',['and',['or','a','b'],'c'],'d']`, la función `fb_prefijo(f1)` respondería a esta consulta como verdadera mientras que la función `fb_enfijo(f1)` devolvería falso. Programamos también una función que nos permite pasar una fórmula de una notación a otra, por ejemplo, la consulta `prefijo_enfijo(f1)`, regresa `['then', ['and', ['a', 'or', 'b'], 'c'], 'd']`.

La función `arbol_sintactico` da la representación en árbol sintáctico de la proposición lógica que recibirá como entrada. Denotando al nodo raíz, que no será más que el operador principal, al hijo izquiero y al hijo derecho de cada nodo. Un hijo podrá ser o bien una proposición o un átomo; en caso de ser una proposición el nuevo nodo contendrá un conectivo lógico que será raíz de un nuevo subárbol, y en caso de tratarse de un átomo, este nodo será una hoja. Procesa tanto fórmulas en forma infija como fórmulas bien formadas en notación prefija. Siguiendo con la fórmula del ejemplo anterior, `arbol_sintactico(f1)` devolverá:

`['raiz(then)', 'nodo_izq', ['raiz(and)', 'nodo_izq', ['raiz(or)', 'nodo_izq', ['hoja', 'a'], 'nodo_der', ['hoja', 'b']],'nodo_der', ['hoja', 'c']],'nodo_der', ['hoja', 'd']]`

También programamos la tabla de verdad para una proposición dada. Por ejemplo, se tiene la proposición $((P \vee H) \wedge \neg P)$ cuya tabla de verdad es:

P	H	$(P \vee H)$	$((P \vee H) \wedge \neg P)$
F	F	F	F
F	V	V	F
V	F	V	V
V	V	V	F

Se introduce en el programa como `tabla_de_verdad([(P ∨ H) ∧ ¬P],L)` y la respuesta obtenida es `L=[(0,0,0),(0,1,0),(1,0,1),(1,1,0)]`.

Por otro lado, se logró modelar de manera eficiente la información presente en la base de datos para así lograr construir nuestro sistema lógico. Identificamos adecuadamente el conjunto de átomos sobre el cual se construye nuestra lógica, para ello fue necesario reconocer los componentes que conforman la signatura de la lógica, las constantes, los términos y las relaciones. En especifico trabajamos con una base de datos cuyas tablas contienen información referente a asentamientos de México, a continuación mostramos ejemplos de los resultados obtenidos:

Los elementos que conforman a la signatura se exponen enseguida.

Constantes Designan objetos de tipos. Siguiendo la definición de constante, sea $\mathcal{A}$ el conjunto de atributos de la base de datos relacional, se tiene que c es una constante si para cualquier $\alpha \in \mathcal{A}$, $c \in Dom\langle\alpha\rangle$. Algunos de los tipos de constantes encontrados son:

Latitud. *Latitudes* a considerar

Longitud. *Longitudes* a considerar

Poblado. *Poblados* a considerar

Región. *Regiones* a considerar

Numeral_Entero. *Números enteros*

Numeral_Real. *Números reales*

Fecha_Vist. *Fecha* de la última visita.

Fecha_Act. *Fecha* de la última actualización.

TipImg. *Tipo de imagen* (GPS, Ortoimagen, etc.)

TipoLoc. *Tipo de localidad* (Rural, Urbana).

Año. *Año* a considerar.

Viviendas. Número de *viviendas* a considerar.

Población. Cantidad de *población* a considerar.

Funciones Las funciones son símbolos que denotan aplicaciones $f : D_1 \times ... \times D_{narg} \to C$.

PobUrb(*Región*). (Unaria) *Población urbana* de una región dada, es decir, el número de habitantes que viven en las zonas urbanas de una región dada.

$$PobUrb : Región \to N$$

PobRur(*Región*). (Unaria) *Población rural* de una región dada, es decir, la cantidad de habitantes que viven en las zonas rurales de una región dada.

$$PobRur : Región \to N$$

PobTot(*Región*). (Unaria) *Población total* de una región dada. Esto es, la cantidad de habitantes que viven en una región dada sin importar el tipo de comunidad.

$$PobTot : Región \to N$$

TotViv(*Región*). (Unaria) *Total de viviendas* en una región dada.

$$TotViv : Región \to N$$

TotLoc(*Región*). (Unaria) *Total de localidades* en una región dada.

$$TotLoc : Región \to N$$

$TasUrb(Región)$**.** (Unaria) *Tasa de urbanización*, se define como el porcentaje de la población urbana que habita una región determinada.

$$\forall R : [EsUnaRegión(R) \Rightarrow TasUrb(R) = PobUrb(R)/PobTot(R)] .$$

$$PobUrb : Región \rightarrow \Re$$

$TasRur(Región)$**.** (Unaria)*Tasa de población rural*, se define como el porcentaje de la población urbana que habita una región determinada.

$$\forall R : [EsUnaRegión(R) \Rightarrow TasRur(R) = PobRur(R)/PobTot(R)] .$$

$$PobUrb : Región \rightarrow \Re$$

$Superficie(Región)$**.** (Unaria) *Superficie* de una región dada. Se sigue la definición tradicional euclidiana.

$$Superficie : Región \rightarrow \Re$$

$DensPob(Región)$**.** (Unaria) *Densidad poblacional* de una región:

$$\forall R : [EsUnaRegión(R) \Rightarrow DensPob(R) = PobTot(R)/Superficie(R)] , Superficie(R) \neq 0.$$

$$DensPob : Región \rightarrow \Re$$

$DensPobProm(Región)$**.** (Unaria) *Densidad poblacional promedio* en localidades de una región.

$$\forall R : [EsUnaRegión(R) \Rightarrow DensPobProm(R) = DensPob(R)/TotLoc(R)] , TotLoc(R) \neq 0.$$

$$DensPobProm : Región \rightarrow \Re$$

$DensHab(Región)$**.** (Unaria) *Densidad habitacional* de una región, es decir, cantidad de viviendas en una región por km^2.

$$\forall R : [EsUnaRegión(R) \Rightarrow DensHab(R) = TotViv(R)/Superficie(R)] .$$

$$DensHab : Región \rightarrow \Re$$

$AltNivelUrb(Región_1, Región_2)$**.** (Binaria) Cual de dos regiones tiene el más *alto nivel de urbanización*

$$AltNivelUrb(Región_1, Región_2) = max(TasUrb(Región_1,), TasUrb(Región_2))$$

$$AltNivelUrb : Región \rightarrow Región$$

$HuelEcol(Región)$**.** (Unaria) *Huella ecológica* de una región:
$\forall R : [EsUnaRegión(R) \Rightarrow HuelEcol(R) = Superficie(R)/PobTot(R)]$, por tanto, $\forall R : [EsUnaRegión(R) \Rightarrow HuelEcol(R) = 1/DensPob(R)]$.

$$HuelEcol : Región \rightarrow \Re$$

$PobVivProm(Región)$**.** (Unaria) *Número de personas promedio por vivienda* en una región.

$$\forall R : [EsUnaRegión(R) \Rightarrow PobVivProm(R) = (TotPob(R)/TotViv(R))$$
$$PobVivProm : Región \rightarrow \Re$$

Relaciones Si A_i es el nombre de un atributo, A_i será reconocida como una relación. En seguida se presentan las relaciones de tipo, metarelaciones, presentes en la base de datos que estamos analizando.

`EsUnaLongitud.` (Unaria) Relación que determina si se está tratando o no de con una longitud.

`EsUnPoblado.` (Unaria) Determina si tenemos o no un poblado o asentamiento.

`EsUnaRegion.` (Binaria) Nos dice si es una región válida dentro de la zona a considerar.

`EsUnNumeral_Entero.` (Unario) Relación que indica si el número con el que se está trabajando es un entero válido.

`EsUnNumeral_Real.` (Unaria) Indica si el número es un real válido.

`EsUnaFecha_Vist.` (Unaria) Determina si se trata de una fecha que satisface las reglas de escritura determinadas.

`EsUnaFecha_Act.` (Unaria)Determina si se trata de una fecha que satisface las reglas de escritura determinadas.

`EsUnTipImg.` (Unaria) Indica si es un tipo de imagen válido.

`EsUnTipoLoc.` (Unaria)Indica si es un tipo de localidad válida (rural o urbana).

`Viviendas.` Número de *viviendas* a considerar en un asentamiento.

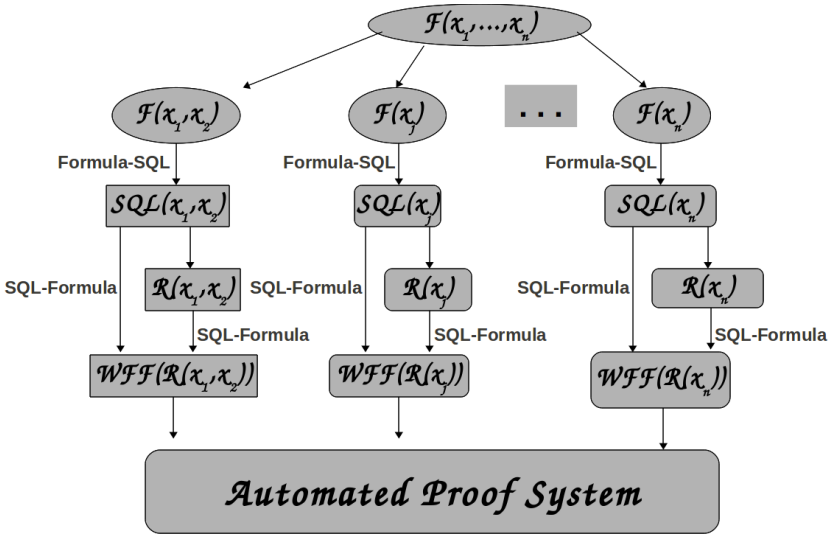
`Población.` Cantidad de *población* a considerar en un asentamiento.

La conjunción de estas relaciones definen a su vez nuevas relaciones. Supóngase que una tabla contiene los atributos A_T, sea $A_j \subseteq A_T$ arbitrario, tal que $A_j = A_1, A_2, ..., A_n$ es posible definir una nueva relación en base a A_j, R_j talque $R_j \subseteq (A_1 \times A_2 \times ... \times A_n)$

Una vez obtenida la signatura es posible definir los términos y los átomos para nuestra lógica, mediante las reglas de construcción antes expuestas, ahora podremos reproducir e identificar fórmulas bien formadas.

5.1.3. Lógica de Predicados Particular

El siguiente paso en nuestro trabajo consistió en el desarrollo de un sistema capaz de establecer la comunicación entre la base de datos y el sistema lógico que hemos construido, por lo tanto, se diseño un programa capaz de traducir una consulta en `SQL` a una fórmula bien formada dentro de la lógica que hemos desarrollado y de una fórmula lógica a su correspondiente consulta `SQL`, pasando por su interpretación en el álgebra relacional. Como se mencionó anteriormente hay dos maneras de obtener la traducción, así pues analizamos los tiempos que toma cada transformación con el fin de determinar que método de traducción resulta más eficiente.



5.1.4. Deducción Automática en la Lógica

En 5.1 se exponen las consultas realizadas sobre las tablas, el tamaño de los archivos resultantes de la consulta y el tiempo de ejecución que tomó cada consulta. Los tiempos fueron obtenidos a partir del entorno de desarrollo integrado, para las consultas en el lenguaje `SQL` se utilizó `pgAdmin` que es la interfaz gráfica para el manejador `PostgreSQL`.

	Convertidor SQL-LPO		Consulta enSQL	
	Tiempo(ms)	Tamaño(Mbytes)	Tiempo(ms)	Tamaño(Mbytes)
SELECT NUMEXT CODIGO NOMSERV TIPOAREA TIPODOM TIPOSERV FROM 211140001ne	7	3.1	6	2.9
DELETE FROM 211140001ne	41	15.5	96	0
DELETE FROM cigel08	26	9.9	41	0
SELECT ALL cigel08	21	9.5	20	8.3
SELECT ALL 211140001ne	39	16	37	15
SELECT IDUNICO CVEASEN TIPOASEN NOMASEN CODIGO TIPOAREA FROM 211140001ne WHERE NOMVIAL = AVENIDA OR NOMREF2 = COLONIA	90	4.3	45	4

Tabla 5.1: Tabla Comparativa

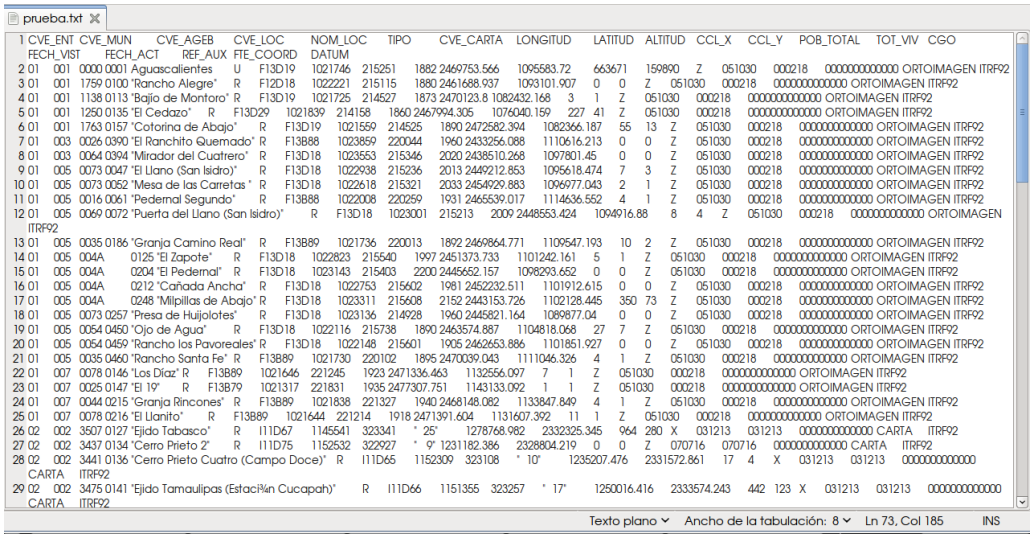
Para ejemplificar se generó una tabla más pequeña que las presentes en la base de datos, con veinte atributos y cien registros. Los atributos son, descrito uno a uno y

en el orden en el que aparecen en la tabla,clave de entidad, clave de área geográfica, clave de localidad, nombre de la localidad, tipo localidad (rural o urbano), clave carta, longitud, latitud, altitud, coordenada c en x, coordenada cónica conforme de Lambert en y, total de población, total de viviendas, código, fecha de visita, fecha de actualización de datos, referencia auxiliar, fuente de coordenadas y *datum*.

El datum es el conjunto de parámetros que especifican la superficie de referencia o el sistema de coordenadas de referencia empleado para el cálculo de coordenadas de puntos en la tierra. Es decir, es el conjunto de parámetros que establecen el origen teórico para las coordenadas terrestres latitud y longitud. Entre los posibles valores se tienen:

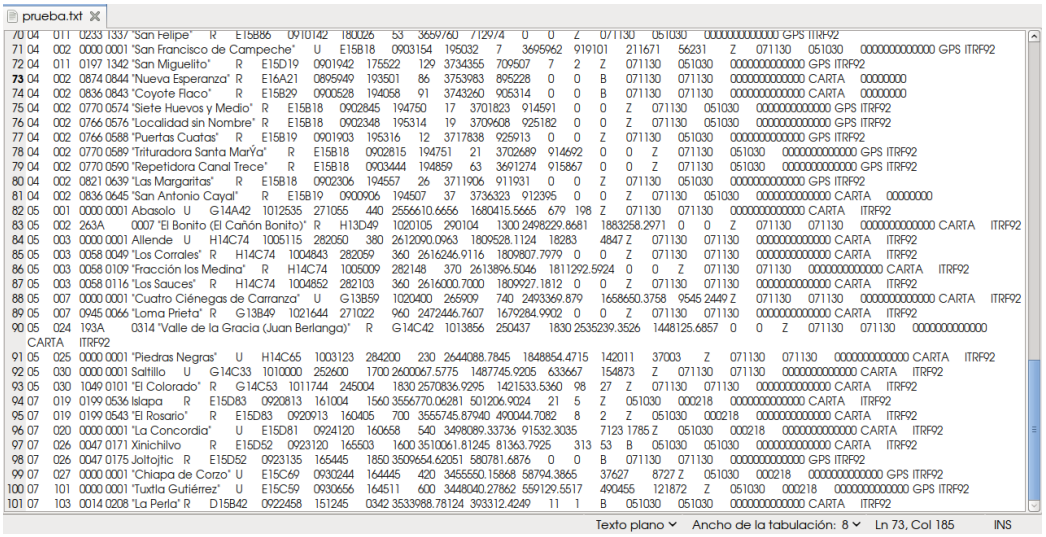
- NAD27: Datum Norteamérica de 1927
- WGS84: Sistema Geodésico mundia de 1984
- ITRF92: Marco de referencia terrestres internacional de 1992
- NAD38: Datum Norteamericano de 1983
- Clarcke 1866: Clarcke
- WGS84: Sistema geodésico mundial de 1984
- GRS80: Sistema geodésico de referencia de 1980

La tabla se ve de esta manera en el formato .txt:


prueba.txt

```
CVE_ENT CVE_MUN CVE_AGEB CVE_LOC NOM_LOC TIPO CVE_CARTA LONGITUD LATITUD ALTITUD CCL_X CCL_Y POB_TOTAL TOT_VIV CGO FECH_VIST FECH_ACT REF_AUX FTE_COORD DATUM
01 001 0000 0001 Aguascalientes U F13D19 1021746 215251 1882 2469753.566 1095583.72 663671 159890 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 001 1759 0100 "Rancho Alegre" R F12D18 1022221 215115 1880 2461688.937 1093101.907 0 0 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 001 1138 0113 "Bajío de Montoro" R F13D19 1021725 214527 1873 2470123.8 1082432.168 3 1 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 001 1250 0135 "El Cedazo" R F13D29 1021839 214158 1860 2467994.305 1076040.159 227 41 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 001 1763 0157 "Cotorina de Abajo" R F13D19 1021559 214525 1890 2472582.394 1082366.187 55 13 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 003 0026 0390 "El Ranchito Quemado" R F13B88 1023859 220044 1960 2433256.088 1110616.213 0 0 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 003 0064 0394 "Mirador del Cuatrero" R F13D18 1023553 215346 2020 2438510.268 1097801.45 0 0 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 0073 0047 "El Llano (San Isidro)" R F13D18 1022938 215236 2013 2449212.853 1095618.474 7 3 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 0073 0052 "Mesa de las Carretas" R F13D18 1022618 215321 2033 2454929.883 1096977.043 2 1 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 0016 0061 "Pedernal Segundo" R F13B88 1022008 220259 1931 2465539.017 1114636.552 4 1 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 0069 0072 "Puerta del Llano (San Isidro)" R F13D18 1023001 215213 2009 2448553.424 1094916.88 8 4 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 0035 0186 "Granja Camino Real" R F13B89 1021736 220013 1892 2469864.771 1109547.193 10 2 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 004A 0125 "El Zapote" R F13D18 1022823 215540 1997 2451373.733 1101242.161 5 1 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 004A 0204 "El Pedernal" R F13D18 1023143 215403 2200 2445652.157 1098293.652 0 0 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 004A 0212 "Cañada Ancha" R F13D18 1022753 215602 1981 2452232.511 1101912.615 0 0 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 004A 0248 "Milpillas de Abajo" R F13D18 1023311 215608 2152 2443153.726 1102128.445 350 73 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 0073 0257 "Presa de Huijolotes" R F13D18 1023136 214928 1960 2445821.164 1089877.04 0 0 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 0054 0450 "Ojo de Agua" R F13D18 1022116 215738 1890 2463574.887 1104818.068 27 7 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 0054 0459 "Rancho los Pavoreales" R F13D18 1022148 215601 1905 2462653.886 1101851.927 0 0 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 005 0035 0460 "Rancho Santa Fe" R F13B89 1021730 220102 1895 2470039.043 1111046.326 4 1 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 007 0078 0146 "Los Díaz" R F13B89 1021646 221245 1923 2471336.463 1132556.097 7 1 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 007 0025 0147 "El 19" R F13B79 1021317 221831 1935 2477307.751 1143133.092 1 1 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 007 0044 0215 "Granja Rincones" R F13B89 1021838 221327 1940 2468148.082 1133847.849 4 1 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
01 007 0078 0216 "El Llanito" R F13B89 1021644 221214 1918 2471391.604 1131607.392 11 1 Z 051030 000218 000000000000 ORTOIMAGEN ITRF92
02 002 3507 0127 "Ejido Tabasco" R I11D67 1145541 323341 " 25" 1278768.982 2332325.345 964 280 X 031213 031213 000000000000 CARTA ITRF92
02 002 3437 0134 "Cerro Prieto 2" R I11D75 1152532 322927 " 9" 1231182.386 2328804.219 0 0 Z 070716 070716 000000000000 CARTA ITRF92
02 002 3441 0136 "Cerro Prieto Cuatro (Campo Doce)" R I11D65 1152309 323108 " 10" 1235207.476 2331572.861 17 4 X 031213 031213 000000000000 CARTA ITRF92
02 002 3475 0141 "Ejido Tamaulipas (Estaci¾n Cucapah)" R I11D66 1151355 323257 " 17" 1250016.416 2333574.243 442 123 X 031213 031213 000000000000 CARTA ITRF92
```

Texto plano · Ancho de la tabulación: 8 · Ln 73, Col 185 · INS

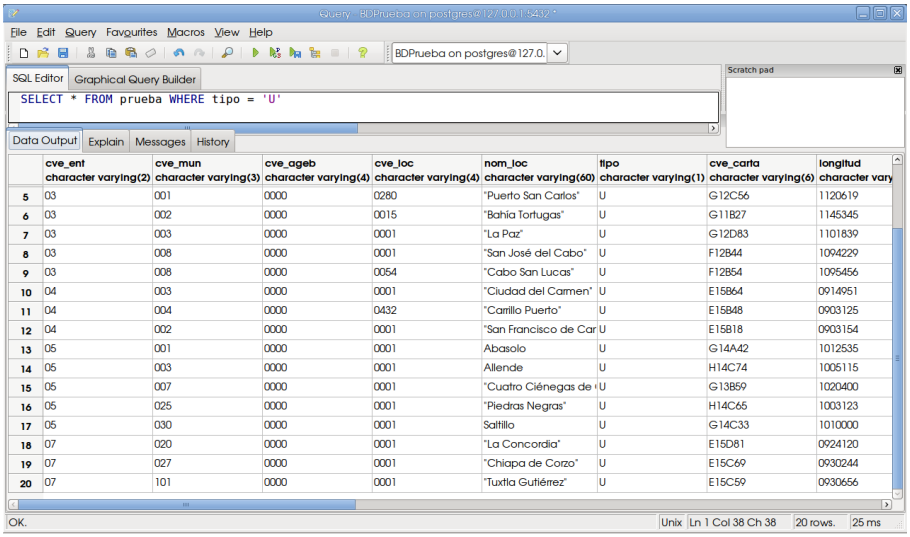


Supóngase que se requieren todas las poblaciones urbanas presentes en nuestra tabla *prueba*, para ello se tiene que resolver la consulta:

```
SELECT ALL FROM prueba WHERE  TIPO = U
```

El archivo respuesta de nuestro sistema después de procesar la consulta es:

La respuesta que nos da la misma consulta sobre la misma tabla al usar el gestor pogAdmin es:



Una vez que hemos obtenido la información en términos de nuestra lógica, habrá que transformarla para que sirva de entrada al demostrador de teoremas como un archivo `.in` y deberá satisfacer la sintaxis que rigen estos archivos. Una vez que hemos obtenido dicho archivo, se introducirá al demostrador para ser procesado y obtener el archivo `.out`. El cual será a su vez reinterpretado con el fin de que sea entendible por el usuario.

Al introducir esta consulta a alguno de los demostradores, el programa preguntará al usuario qué demostrador prefiere utilizar y generará un archivo con el siguiente nombre:

`NomDemSELECTALLFROMpruebaWHERETIPO=U`

Con extensión .in en caso de que el usuario haya decidido usar el demostrador `Prover9`. El archivo generado se muestra en las figuras 5.1.4– 5.3.

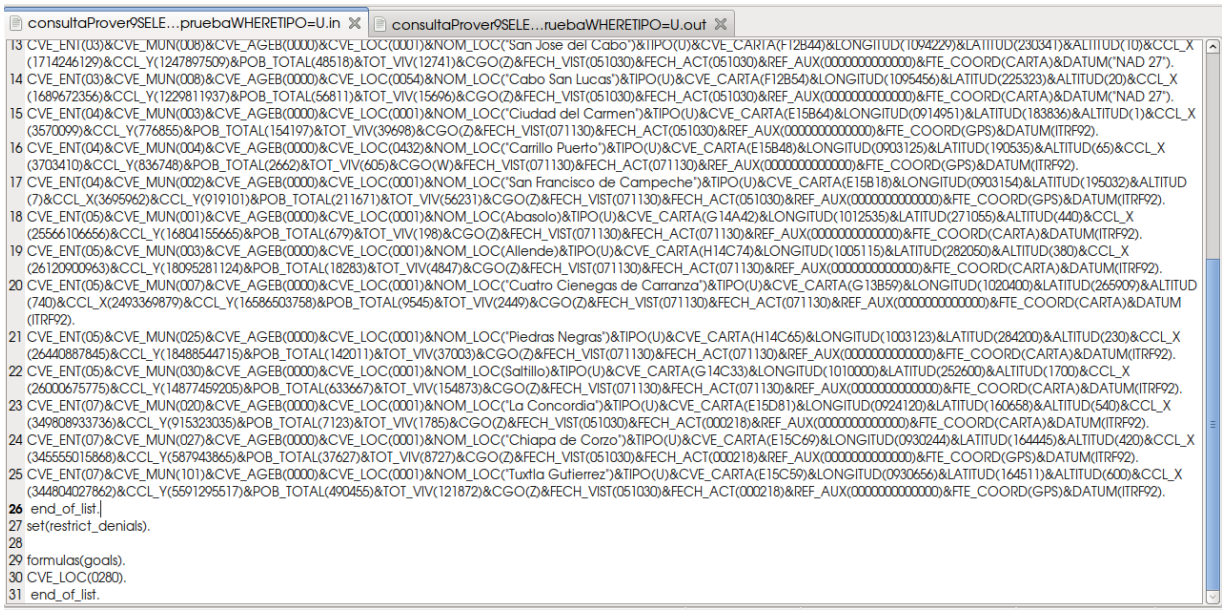
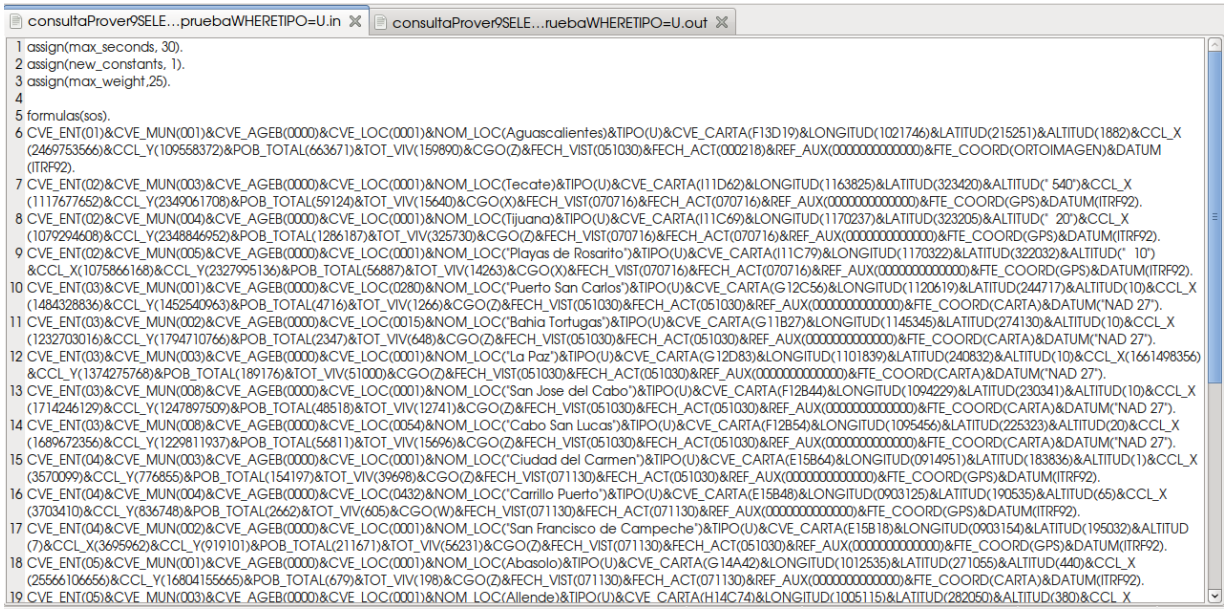


Figura 5.1: Entrada a Prover9 2



Entrada a Prover9 1

En caso de que el demostrador elegido por el usuario haya sido `SPASS`, el archivo generado como entrada a este demostrador seguirá su estructura. A continuación se presentan capturas de pantalla de dicho archivo, en las figuras 5.6– 5.11

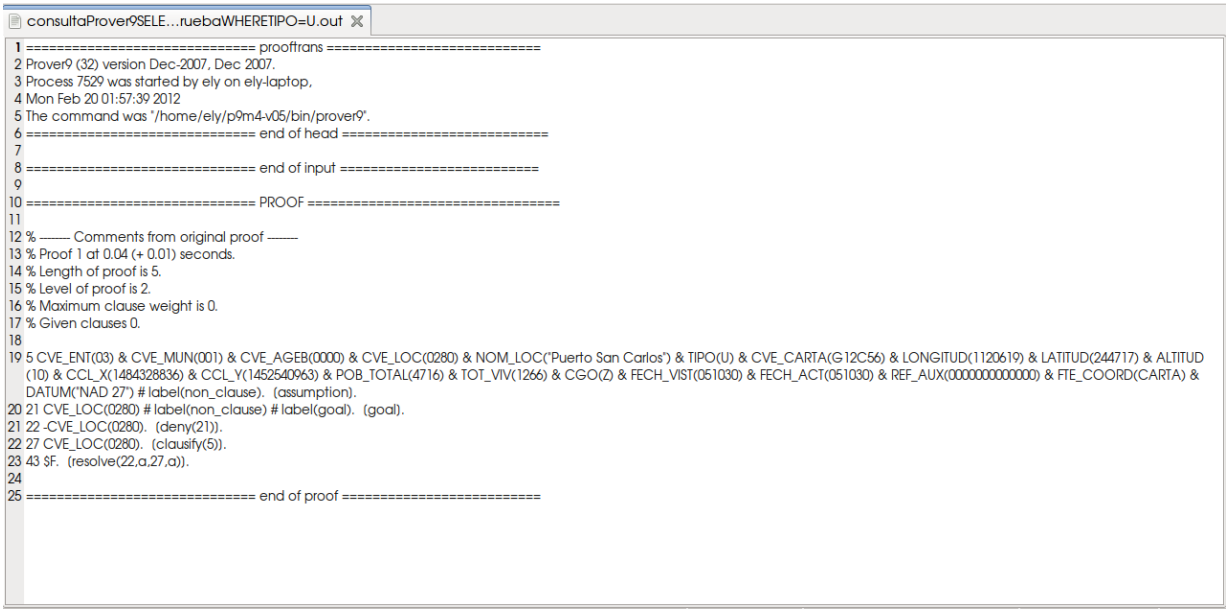


Figura 5.2: Salida a Prover9 1

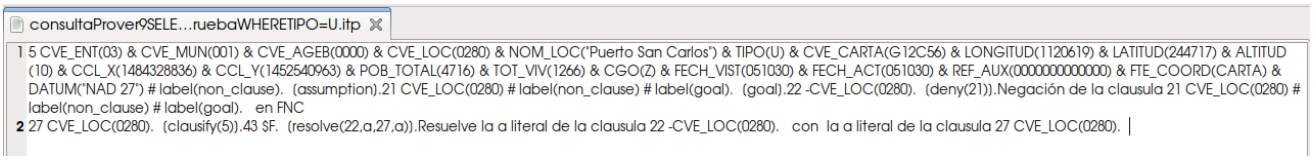


Figura 5.3: Salida a Prover9 2

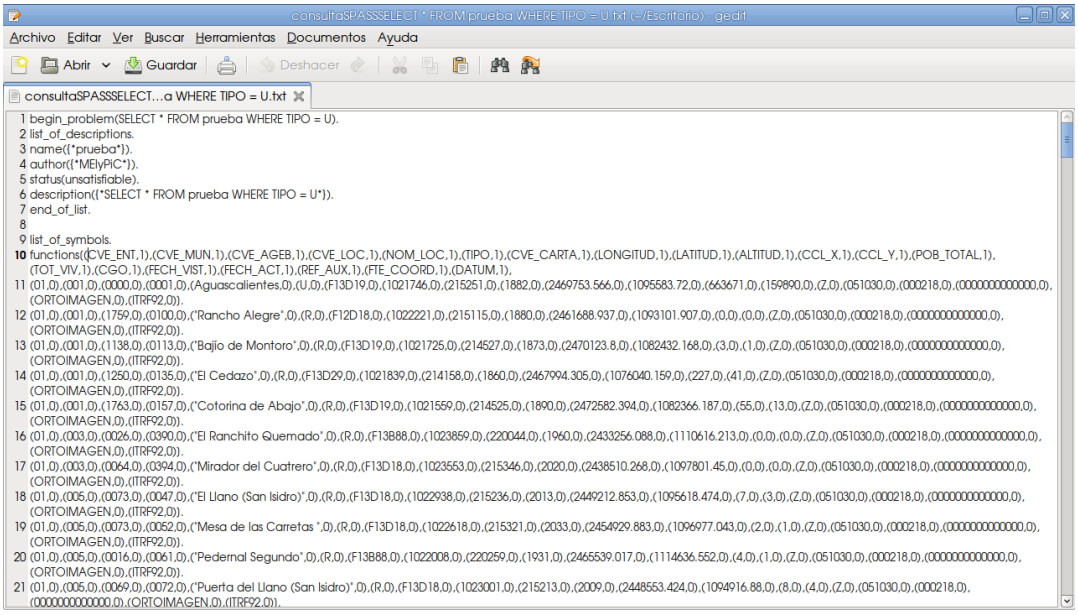


Figura 5.4: Entrada a SPASS 1

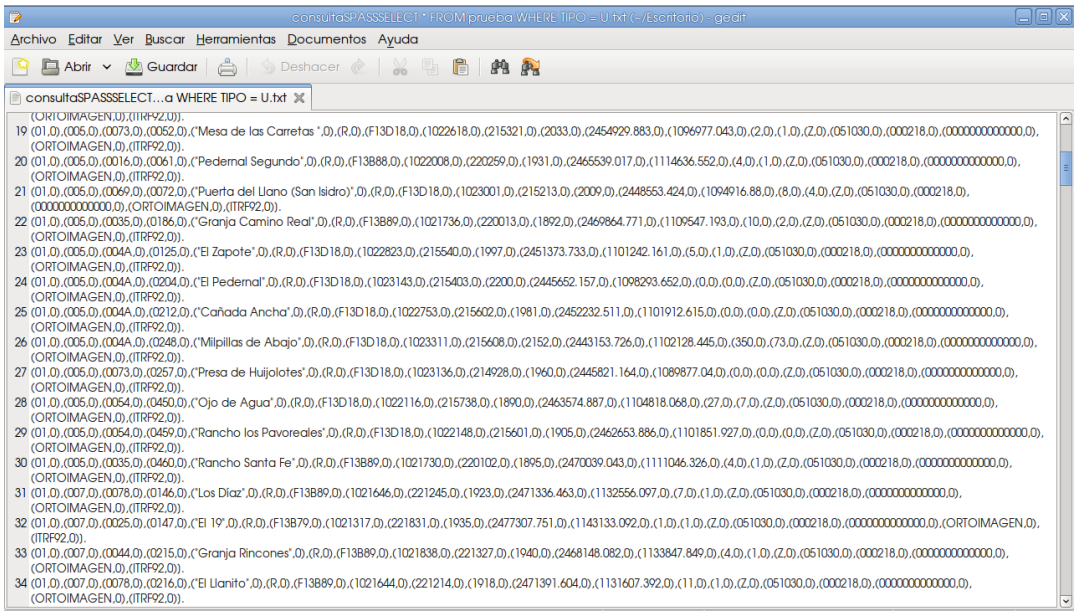


Figura 5.5: Entrada a SPASS 2

Figura 5.6: Entrada a SPASS 3

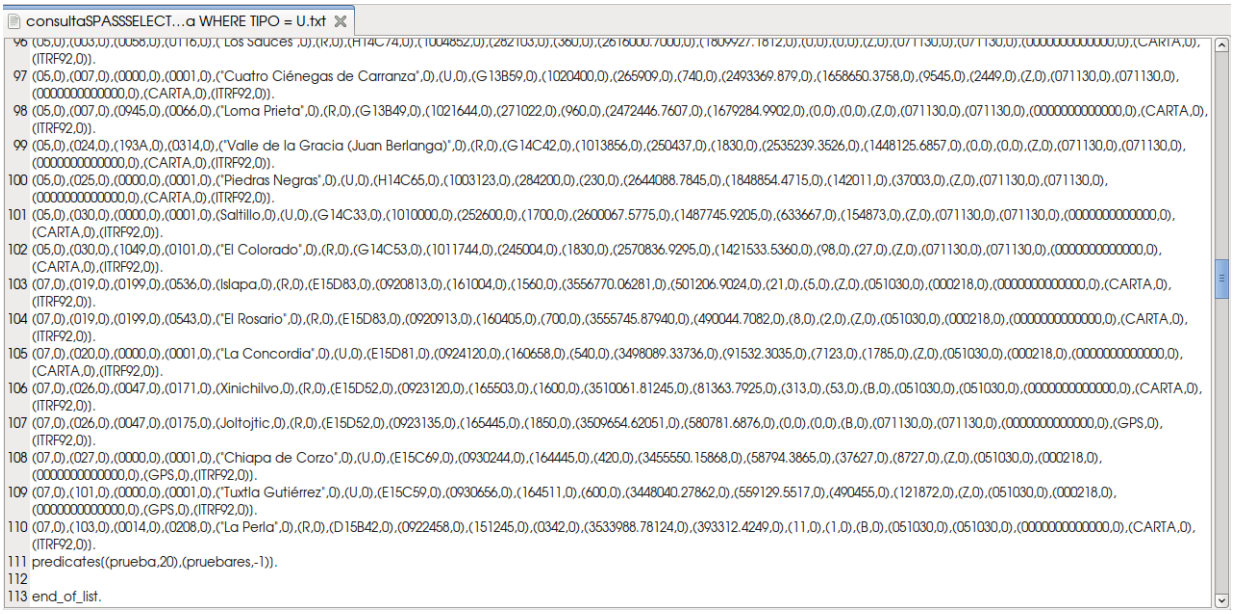


Figura 5.7: Entrada a SPASS 4

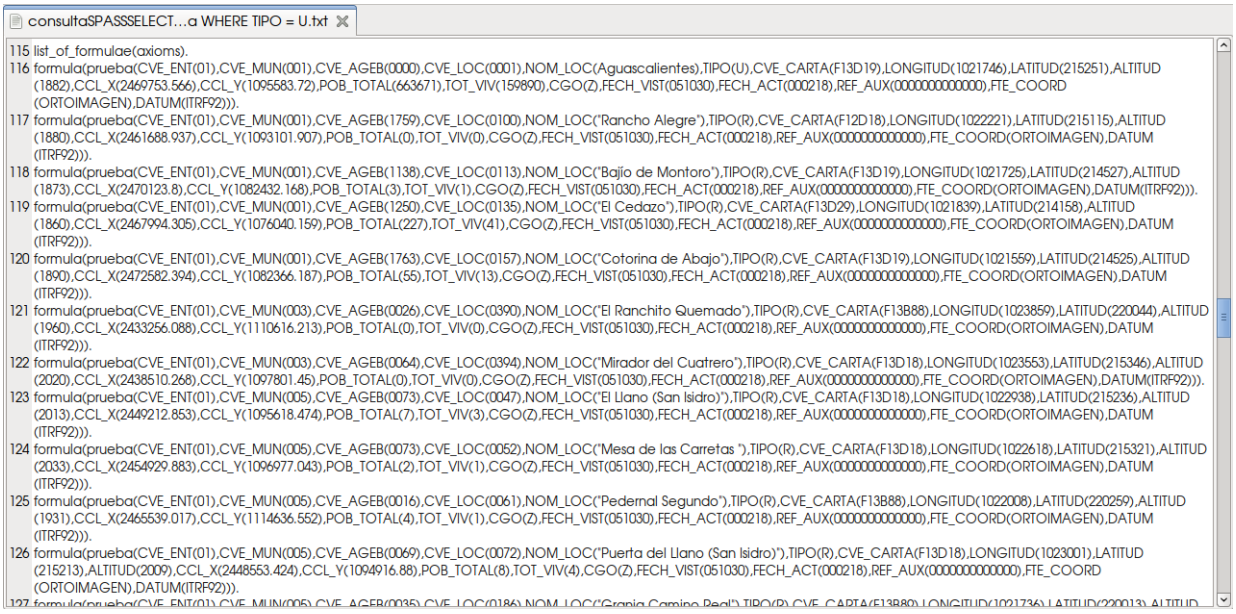
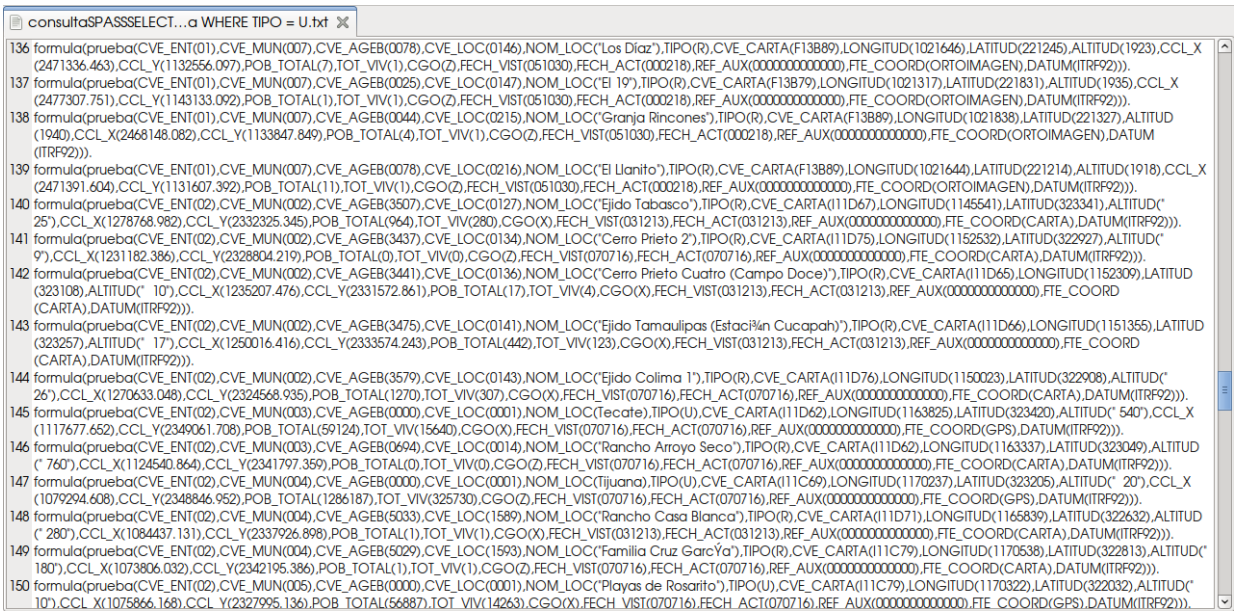


Figura 5.8: Entrada a SPASS 5


consultaSPASSSELECT...a WHERE TIPO = U.txt

```
136 formula(prueba(CVE_ENT(01),CVE_MUN(007),CVE_AGEB(0078),CVE_LOC(0146),NOM_LOC("Los Díaz"),TIPO(R),CVE_CARTA(F13B89),LONGITUD(1021646),LATITUD(221245),ALTITUD(1923),CCL_X
(2471336.463),CCL_Y(1132556.097),POB_TOTAL(7),TOT_VIV(1),CGO(Z),FECH_VIST(051030),FECH_ACT(000218),REF_AUX(000000000000000),FTE_COORD(ORTOIMAGEN),DATUM(ITRF92))).
137 formula(prueba(CVE_ENT(01),CVE_MUN(007),CVE_AGEB(0025),CVE_LOC(0147),NOM_LOC("El 19"),TIPO(R),CVE_CARTA(F13B79),LONGITUD(1021317),LATITUD(221831),ALTITUD(1935),CCL_X
(2477307.751),CCL_Y(1143133.092),POB_TOTAL(1),TOT_VIV(1),CGO(Z),FECH_VIST(051030),FECH_ACT(000218),REF_AUX(000000000000000),FTE_COORD(ORTOIMAGEN),DATUM(ITRF92))).
138 formula(prueba(CVE_ENT(01),CVE_MUN(007),CVE_AGEB(0044),CVE_LOC(0215),NOM_LOC("Granja Rincones"),TIPO(R),CVE_CARTA(F13B89),LONGITUD(1021838),LATITUD(221327),ALTITUD
(1940),CCL_X(2468148.082),CCL_Y(1133847.849),POB_TOTAL(4),TOT_VIV(1),CGO(Z),FECH_VIST(051030),FECH_ACT(000218),REF_AUX(000000000000000),FTE_COORD(ORTOIMAGEN),DATUM
(ITRF92))).
139 formula(prueba(CVE_ENT(01),CVE_MUN(007),CVE_AGEB(0078),CVE_LOC(0216),NOM_LOC("El Llanito"),TIPO(R),CVE_CARTA(F13B89),LONGITUD(1021644),LATITUD(221214),ALTITUD(1918),CCL_X
(2471391.604),CCL_Y(1131607.392),POB_TOTAL(11),TOT_VIV(1),CGO(Z),FECH_VIST(051030),FECH_ACT(000218),REF_AUX(000000000000000),FTE_COORD(ORTOIMAGEN),DATUM(ITRF92))).
140 formula(prueba(CVE_ENT(02),CVE_MUN(002),CVE_AGEB(3507),CVE_LOC(0127),NOM_LOC("Ejido Tabasco"),TIPO(R),CVE_CARTA(I11D67),LONGITUD(1145541),LATITUD(323341),ALTITUD("
25"),CCL_X(1278768.982),CCL_Y(2332325.345),POB_TOTAL(964),TOT_VIV(280),CGO(X),FECH_VIST(031213),FECH_ACT(031213),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
141 formula(prueba(CVE_ENT(02),CVE_MUN(002),CVE_AGEB(3437),CVE_LOC(0134),NOM_LOC("Cerro Prieto 2"),TIPO(R),CVE_CARTA(I11D75),LONGITUD(1152532),LATITUD(322927),ALTITUD("
9"),CCL_X(1231182.386),CCL_Y(2328804.219),POB_TOTAL(0),TOT_VIV(0),CGO(Z),FECH_VIST(070716),FECH_ACT(070716),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
142 formula(prueba(CVE_ENT(02),CVE_MUN(002),CVE_AGEB(3441),CVE_LOC(0136),NOM_LOC("Cerro Prieto Cuatro (Campo Doce)"),TIPO(R),CVE_CARTA(I11D65),LONGITUD(1152309),LATITUD
(323108),ALTITUD("  10"),CCL_X(1235207.476),CCL_Y(2331572.861),POB_TOTAL(17),TOT_VIV(4),CGO(X),FECH_VIST(031213),FECH_ACT(031213),REF_AUX(000000000000000),FTE_COORD
(CARTA),DATUM(ITRF92))).
143 formula(prueba(CVE_ENT(02),CVE_MUN(002),CVE_AGEB(3475),CVE_LOC(0141),NOM_LOC("Ejido Tamaulipas (Estaci¾n Cucapah)"),TIPO(R),CVE_CARTA(I11D66),LONGITUD(1151355),LATITUD
(323257),ALTITUD("  17"),CCL_X(1250016.416),CCL_Y(2333574.243),POB_TOTAL(442),TOT_VIV(123),CGO(X),FECH_VIST(031213),FECH_ACT(031213),REF_AUX(000000000000000),FTE_COORD
(CARTA),DATUM(ITRF92))).
144 formula(prueba(CVE_ENT(02),CVE_MUN(002),CVE_AGEB(3579),CVE_LOC(0143),NOM_LOC("Ejido Colima 1"),TIPO(R),CVE_CARTA(I11D76),LONGITUD(1150023),LATITUD(322908),ALTITUD("
26"),CCL_X(1270633.048),CCL_Y(2324568.935),POB_TOTAL(1270),TOT_VIV(307),CGO(X),FECH_VIST(070716),FECH_ACT(070716),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
145 formula(prueba(CVE_ENT(02),CVE_MUN(003),CVE_AGEB(0000),CVE_LOC(0001),NOM_LOC(Tecate),TIPO(U),CVE_CARTA(I11D62),LONGITUD(1163825),LATITUD(323420),ALTITUD(" 540"),CCL_X
(1117677.652),CCL_Y(2349061.708),POB_TOTAL(59124),TOT_VIV(15640),CGO(X),FECH_VIST(070716),FECH_ACT(070716),REF_AUX(000000000000000),FTE_COORD(GPS),DATUM(ITRF92))).
146 formula(prueba(CVE_ENT(02),CVE_MUN(003),CVE_AGEB(0694),CVE_LOC(0014),NOM_LOC("Rancho Arroyo Seco"),TIPO(R),CVE_CARTA(I11D62),LONGITUD(1163337),LATITUD(323049),ALTITUD
(" 760"),CCL_X(1124540.864),CCL_Y(2341797.359),POB_TOTAL(0),TOT_VIV(0),CGO(Z),FECH_VIST(070716),FECH_ACT(070716),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
147 formula(prueba(CVE_ENT(02),CVE_MUN(004),CVE_AGEB(0000),CVE_LOC(0001),NOM_LOC(Tijuana),TIPO(U),CVE_CARTA(I11C69),LONGITUD(1170237),LATITUD(323205),ALTITUD("  20"),CCL_X
(1079294.608),CCL_Y(2348846.952),POB_TOTAL(1286187),TOT_VIV(325730),CGO(Z),FECH_VIST(070716),FECH_ACT(070716),REF_AUX(000000000000000),FTE_COORD(GPS),DATUM(ITRF92))).
148 formula(prueba(CVE_ENT(02),CVE_MUN(004),CVE_AGEB(5033),CVE_LOC(1589),NOM_LOC("Rancho Casa Blanca"),TIPO(R),CVE_CARTA(I11D71),LONGITUD(1165839),LATITUD(322632),ALTITUD
(" 280"),CCL_X(1084437.131),CCL_Y(2337926.898),POB_TOTAL(1),TOT_VIV(1),CGO(X),FECH_VIST(031213),FECH_ACT(031213),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
149 formula(prueba(CVE_ENT(02),CVE_MUN(004),CVE_AGEB(5029),CVE_LOC(1593),NOM_LOC("Familia Cruz GarcÝa"),TIPO(R),CVE_CARTA(I11C79),LONGITUD(1170538),LATITUD(322813),ALTITUD("
180"),CCL_X(1073806.032),CCL_Y(2342195.386),POB_TOTAL(1),TOT_VIV(1),CGO(Z),FECH_VIST(070716),FECH_ACT(070716),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
150 formula(prueba(CVE_ENT(02),CVE_MUN(005),CVE_AGEB(0000),CVE_LOC(0001),NOM_LOC("Playas de Rosarito"),TIPO(U),CVE_CARTA(I11C79),LONGITUD(1170322),LATITUD(322032),ALTITUD("
10"),CCL_X(1075866.168),CCL_Y(2327995.136),POB_TOTAL(56887),TOT_VIV(14263),CGO(X),FECH_VIST(070716),FECH_ACT(070716),REF_AUX(000000000000000),FTE_COORD(GPS),DATUM(ITRF92))).
```

Figura 5.9: Entrada a SPASS 6


consultaSPASSSELECT...a WHERE TIPO = U.txt

```
204 formula(prueba(CVE_ENT(05),CVE_MUN(024),CVE_AGEB(193A),CVE_LOC(0314),NOM_LOC("Valle de la Gracia (Juan Beriangu)"),TIPO(R),CVE_CARTA(G14C42),LONGITUD(1015656),LATITUD
(250437),ALTITUD(1830),CCL_X(2535239.3526),CCL_Y(1448125.6857),POB_TOTAL(0),TOT_VIV(0),CGO(Z),FECH_VIST(071130),FECH_ACT(071130),REF_AUX(000000000000000),FTE_COORD
(CARTA),DATUM(ITRF92))).
205 formula(prueba(CVE_ENT(05),CVE_MUN(025),CVE_AGEB(0000),CVE_LOC(0001),NOM_LOC("Piedras Negras"),TIPO(U),CVE_CARTA(H14C65),LONGITUD(1003123),LATITUD(284200),ALTITUD
(230),CCL_X(2644088.7845),CCL_Y(1848854.4715),POB_TOTAL(142011),TOT_VIV(37003),CGO(Z),FECH_VIST(071130),FECH_ACT(071130),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM
(ITRF92))).
206 formula(prueba(CVE_ENT(05),CVE_MUN(030),CVE_AGEB(0000),CVE_LOC(0001),NOM_LOC(Saltillo),TIPO(U),CVE_CARTA(G14C33),LONGITUD(1010000),LATITUD(252600),ALTITUD(1700),CCL_X
(2600067.5775),CCL_Y(1487745.9205),POB_TOTAL(633667),TOT_VIV(154873),CGO(Z),FECH_VIST(071130),FECH_ACT(071130),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
207 formula(prueba(CVE_ENT(05),CVE_MUN(030),CVE_AGEB(1049),CVE_LOC(0101),NOM_LOC("El Colorado"),TIPO(R),CVE_CARTA(G14C53),LONGITUD(1011744),LATITUD(245004),ALTITUD
(1830),CCL_X(2570836.9295),CCL_Y(1421533.5360),POB_TOTAL(98),TOT_VIV(27),CGO(Z),FECH_VIST(071130),FECH_ACT(071130),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
208 formula(prueba(CVE_ENT(07),CVE_MUN(019),CVE_AGEB(0199),CVE_LOC(0536),NOM_LOC(Islapa),TIPO(R),CVE_CARTA(E15D83),LONGITUD(0920813),LATITUD(161004),ALTITUD(1560),CCL_X
(3556770.06281),CCL_Y(501206.9024),POB_TOTAL(21),TOT_VIV(5),CGO(Z),FECH_VIST(051030),FECH_ACT(000218),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
209 formula(prueba(CVE_ENT(07),CVE_MUN(019),CVE_AGEB(0199),CVE_LOC(0543),NOM_LOC("El Rosario"),TIPO(R),CVE_CARTA(E15D83),LONGITUD(0920913),LATITUD(160405),ALTITUD(700),CCL_X
(3555745.87940),CCL_Y(490044.7082),POB_TOTAL(8),TOT_VIV(2),CGO(Z),FECH_VIST(051030),FECH_ACT(000218),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
210 formula(prueba(CVE_ENT(07),CVE_MUN(020),CVE_AGEB(0000),CVE_LOC(0001),NOM_LOC("La Concordia"),TIPO(U),CVE_CARTA(E15D81),LONGITUD(0924120),LATITUD(160658),ALTITUD
(540),CCL_X(3498089.33736),CCL_Y(91532.3035),POB_TOTAL(7123),TOT_VIV(1785),CGO(Z),FECH_VIST(051030),FECH_ACT(000218),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
211 formula(prueba(CVE_ENT(07),CVE_MUN(026),CVE_AGEB(0047),CVE_LOC(0171),NOM_LOC(Xinichilvo),TIPO(R),CVE_CARTA(E15D52),LONGITUD(0923120),LATITUD(165503),ALTITUD(1600),CCL_X
(3510061.81245),CCL_Y(81363.7925),POB_TOTAL(313),TOT_VIV(53),CGO(B),FECH_VIST(051030),FECH_ACT(051030),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
212 formula(prueba(CVE_ENT(07),CVE_MUN(026),CVE_AGEB(0047),CVE_LOC(0175),NOM_LOC(Joltojtic),TIPO(R),CVE_CARTA(E15D52),LONGITUD(0923135),LATITUD(165445),ALTITUD(1850),CCL_X
(3509654.62051),CCL_Y(580781.6876),POB_TOTAL(0),TOT_VIV(0),CGO(B),FECH_VIST(071130),FECH_ACT(071130),REF_AUX(000000000000000),FTE_COORD(GPS),DATUM(ITRF92))).
213 formula(prueba(CVE_ENT(07),CVE_MUN(027),CVE_AGEB(0000),CVE_LOC(0001),NOM_LOC("Chiapa de Corzo"),TIPO(U),CVE_CARTA(E15C69),LONGITUD(0930244),LATITUD(164445),ALTITUD
(420),CCL_X(3455550.15868),CCL_Y(58794.3865),POB_TOTAL(37627),TOT_VIV(8727),CGO(Z),FECH_VIST(051030),FECH_ACT(000218),REF_AUX(000000000000000),FTE_COORD(GPS),DATUM(ITRF92))).
214 formula(prueba(CVE_ENT(07),CVE_MUN(101),CVE_AGEB(0000),CVE_LOC(0001),NOM_LOC("Tuxtla Gutiérrez"),TIPO(U),CVE_CARTA(E15C59),LONGITUD(0930656),LATITUD(164511),ALTITUD
(600),CCL_X(3448040.27862),CCL_Y(559129.5517),POB_TOTAL(490455),TOT_VIV(121872),CGO(Z),FECH_VIST(051030),FECH_ACT(000218),REF_AUX(000000000000000),FTE_COORD(GPS),DATUM
(ITRF92))).
215 formula(prueba(CVE_ENT(07),CVE_MUN(103),CVE_AGEB(0014),CVE_LOC(0208),NOM_LOC("La Perla"),TIPO(R),CVE_CARTA(D15B42),LONGITUD(0922458),LATITUD(151245),ALTITUD(0342),CCL_X
(3533988.78124),CCL_Y(393312.4249),POB_TOTAL(11),TOT_VIV(1),CGO(B),FECH_VIST(051030),FECH_ACT(051030),REF_AUX(000000000000000),FTE_COORD(CARTA),DATUM(ITRF92))).
216 formula(forall([ x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, x16, x17, x18, x19], implies(prueba(CVE_ENT(x0), CVE_MUN(x1), CVE_AGEB(x2), CVE_LOC(x3),
NOM_LOC(x4), TIPO(x5), CVE_CARTA(x6), LONGITUD(x7), LATITUD(x8), ALTITUD(x9), CCL_X(x10), CCL_Y(x11), POB_TOTAL(x12), TOT_VIV(x13), CGO(x14), FECH_VIST(x15), FECH_ACT(x16),
REF_AUX(x17), FTE_COORD(x18), DATUM(x19)),pruebaresp(CVE_ENT(x0), CVE_MUN(x1), CVE_AGEB(x2), CVE_LOC(x3), NOM_LOC(x4), TIPO(x5), CVE_CARTA(x6), LONGITUD(x7), LATITUD(x8),
ALTITUD(x9), CCL_X(x10), CCL_Y(x11), POB_TOTAL(x12), TOT_VIV(x13), CGO(x14), FECH_VIST(x15), FECH_ACT(x16), REF_AUX(x17), FTE_COORD(x18), DATUM(x19))))).
217 end_of_list.
218
```

Figura 5.10: Entrada a SPASS 7

Figura 5.11: Entrada a SPASS 8

5.2. URL del trabajo

Es posible disponer de la información aquí expuesta y de los programas desarrollados mediante la `http://computacion.cs.cinvestav.mx/ mpiceno/Page/Tesis.html`. En la primera parte muestra un resumén de nuestro trabajo, las exposiciones presentadas al largo del desarrollo, un árticulo referente a nuestro trabajo de tesis y la exposición hecha para su presentación.

También hacemos disponible los programas realizados, su documentación y algunos casos de prueba.

5.3. Programas

Entre los programas encontramos los de reconocimiento de fórmulas bien formadas, de construcción de la tabla de verdad de una fórmula y de construcción del árbol sintáctico de la fórmula dada, todos ellos capaces de manejar fórmulas tanto en notación infija como prefija.

Encontramos aquí el código de programa que enlaza a una consulta SQL con nuestra lógica y con las respectivas transformaciones que sirven de entrada a los demostradores `Prover9` y `SPASS`; y la interpretación del archivo salida del demostrador `Prover9`.

Capítulo 6

Conclusiones

6.1. Conclusiones inmediatas

Se logró la formalización, en lógica de primer orden, de la información presente en una base de datos relacional, mediante un prototipo para la obtención de la signatura de la lógica que a su vez deducirá los átomos sobre los cuales está definida nuestra lógica. Se logró la correcta traducción de una consulta hecha a una base de datos en términos de la lógica de primer orden diseñada. Esta traducción se adecuó para servir de entrada a uno de los demostradores automáticos que se estudiaron, para después interpretar el archivo de salida del demostrador, ésto con el fin de hacer la demostración más legible para cualquier usuario.

Para lograr lo anterior se debieron realizar pruebas de reconocimiento de fórmulas bien formadas en diversas notaciones, la construcción del árbol sintáctico y la tabla de verdad de las mismas.

Obtuvimos una base de datos lo suficientemente substanciosa sobre la cual se realizaron las pruebas y la publicación de un artículo relevante a la tesis en el que se muestra que fue posible obtener los elementos que conforman la signatura de manera exitosa.

6.2. Trabajo a Futuro

La lógica de primer orden es un modelo adecuado para el razonamiento matemático, sin embargo, es necesario ajustar ese modelo a nuestras intuiciones acerca del razonamiento natural con el propósito de ampliar la potencia expresiva del lenguaje lógico en áreas como el razonamiento hipotético, razonamiento temporal, razonamiento epistémico y razonamiento no monótono, tomando siempre como referencia la lógica clásica.

Todo esto da lugar a plantearnos el uso de la lógica modal como un trabajo a futuro. La lógica modal está basada en la lógica de primer orden, sin embargo, la lógica

modal agrega operadores, conocidos como operadores modales los cuales califican proposiciones como *"posibles"* o *"necesarias"*. Las distintas ramas de la lógica modal se diferencian entre sí mediante la aplicación e interpretación de sus operadores. Entre sus variantes encontramos a la Lógica Temporal Modal, a la Lógica Dinámica, a la Lógica Epistémica, y a la Lógica Deóntica.

Ahora bien, la lógica epistémica se dice ser la lógica del razonamiento del conocimiento. Propiamente la palabra *"epistémica"* se deriva de la palabra griega *episteme* [16] que significa conocimiento, por lo tanto la lógica epistémica es la lógica del conocimiento, sin embargo, es tomada también como la lógica de creencias, de ahí que se considere como la lógica del razonamiento.

Mientras que la epistemología posee una larga tradición filosófica que se origina en la Grecia Antigua, la lógica epistémica es un desarrollo mucho más reciente con aplicaciones en numerosos campos, tales como filosofía, teoría de la computación, inteligencia artificial, economía y lingüística. Mientras que los filósofos a partir de Aristóteles han discutido la lógica modal, y los filósofos medievales tales como Ockham y Duns Scotus desarrollaron numerosas observaciones, el primero en realizar un tratamiento simbólico y sistemático de este tema fue C.I. Lewis [17]. Tiempo después S. Kripke[18] estableció, mediante un ensayo, las bases de la lógica modal de hoy en día. Mientras que los primeros trabajos dentro del área de deducción de información fueron dados por Von Wright [3] en 1951, en el trabajo titulado *An Essay in Modal Logic*. Pero no fue hasta que J. Hintikka [19] sugiere el uso de la lógica modal para capturar la semántica del conocimiento.

Retomando, las *estructuras de Kripke* [18] (o *modelos de Kripke*) nos permiten representar el razonamiento de uno o varios agentes acerca del conocimiento de otro u otros agentes o incluso de su propio conocimiento. Podemos no sólo representar el conocimiento particular sino también el conocimiento común o distribuido de un determinado grupo de agentes que pertenecen al conjunto de agentes asociados al modelo sobre el que esté trabajando.

Para extender este trabajo al ámbito de la lógica epistemica será necesario además plantear los diferentes tipos de conocimiento y el tipo de información (cognoscible, a la larga cognoscible y potencialmente cognoscible).

A partir un conjunto finito, denominado conjunto de átomos P_0, se generan dos nuevos conjuntos, P_1 y P_2.

$P_1 = \beta(P_0)$ se define recursivamente mediante las siguientes dos reglas:

- $\phi \in P_0 \Rightarrow \phi \in P_1$, en otras palabras, $P_0 \subset P_1$.
- $[\phi_1, \phi_2 \in P_1 \Rightarrow \neg\phi_1, \phi_1 \vee \phi_2, \phi_1 \wedge \phi_2 \in P_1]$

Y el conjunto $P_2 = \mathbf{K}\beta(P_0)$, es el conjunto de fórmulas definido por:

- $\phi \in P_1 \Rightarrow \mathbf{K}\phi \in P_2$

Así la lógica epistémica simple, con átomos en P_0, se define como:

$$LES(P_0) = P_1 \cup P_2$$

Otros proyectos a considerar como posibles trabajos futuros tenemos los siguientes:

- Aplicar este trabajo enfocándonos en aspectos de seguridad con el fin de dar a conocer la mayor cantidad de información, es decir información que pueda hacerse pública, siempre y cuando la información privada se conserve como tal.
- Diseñar otro sistema lógico basándonos en la lógica espistémica simple.
- Diseñar otro sistema lógico basándonos en alguna otra rama de la lógica modal.
- Hacer comparaciones entre los sistemas desarrollados.

Bibliografía

[1] G. Morales-Luna, "Simple epistemic logic for relational database," in *MICAI* (C. A. C. Coello, A. de Albornoz, L. E. Sucar, and O. C. Battistutti, eds.), vol. 2313 of *Lecture Notes in Computer Science*, pp. 234–241, Springer, 2002.

[2] Y. V. Patrick Blackburn, Maarten de Rijke, *Modal Logic.* Cambridge University Press, 2001.

[3] G. H. V. Wright, "An essay in modal logic," *Studies in the logic and the Foundations of Mathematics*, no. 4, 1951.

[4] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[5] A. for Computing Machine, "Advancing computing as a science and profession." `http://www.acm.org/`.

[6] P. A. et. al, "Theorem proving system." `http://gtps.math.cmu.edu/tps-about.html`.

[7] P. A. et. al, "Educational theorem proving system." `http://gtps.math.cmu.edu/tps-about.html`.

[8] S. K. Das, "Modal logics in the theory of relational databases," *Inf. Process. Lett.*, vol. 57, no. 1, pp. 1–7, 1996.

[9] A. Russo, "Generalising propositional modal logic using labelled deductive systems," in *Frontiers of Combining Systems (FroCos)*, pp. 57–73, 1996.

[10] T. sheng Hsu, C.-J. Liau, and D.-W. Wang, "A logical model for privacy protection," in *ISC* (G. I. Davida and Y. Frankel, eds.), vol. 2200 of *Lecture Notes in Computer Science*, pp. 110–124, Springer, 2001.

[11] N. A. y Simon Herbert A., "The logic theory machine: A complex information processing system," *IRE Transactions on Information Theory IT-2*, 1956.

[12] U. Nilsson and J. Maluszynski, *Logic, programming and Prolog.* Wiley, 1990.

[13] W. McCune, "Prover9 and mace4." `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[14] U. B. et. al, "Spass." `http://www.spass-prover.org/`, 1991–2009.

[15] S. Schulz, "System description: E 0.81," in *Proceedings of the 2nd International Joint Conference on Automated Reasoning (Springer) LNAI*, pp. 223–228, 2004.

[16] J. W. Ilkka Niiniluoto, Matti Sintonen, *Handbook of Epistemology*. Springer, 2004.

[17] C. I. Lewis, "Emch's calculus and strict implication," *J. Symb. Log.*, vol. 1, no. 3, pp. 77–86, 1936.

[18] S. Kripke, "A completeness theorem in modal logic," *J. Symb. Log.*, vol. 24, no. 1, pp. 1–14, 1959.

[19] J. Hintikka, "Reasoning about knowledge in philosophy: The paradigm of epistemic logic," in *TARK*, pp. 63–80, 1986.