Máster en Investigación en Informática, Facultad de Informática,
Universidad Complutense de Madrid

Curso 2009 2010
Proyecto Fin de Máster en Programación y Tecnología
Software

Un sistema de bases de datos deductivas con restricciones

Autor:
Gabriel Aranda López

Director:
Jaime Sánchez Hernández

Colaboradores externos:
Susana Nieva Soto
Fernando Sáenz Pérez

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Un sistema de bases de datos deductivas con restricciones", realizado durante el curso académico 2009-2010 bajo la dirección de Jaime Sánchez Hernández y con la colaboración externa de dirección de Susana Nieva Soto y Fernando Sáenz Pérez, en el Departamento de Sistemas Informáticos y Computación y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Gabriel Aranda López

Summary

Hereditary Harrop formulas with constraints have been proposed as a basis for logic programming languages. In the same way that Datalog (with constraints) emerges from logic programming as a deductive database language, such formulas can support a very expressive framework for deductive databases.

This work first presents a comparison between deductive database systems and constraint databases. Then, it presents the theoretical foundations of the $HH_{\neg}(\mathcal{C})$ language and an implementation that shows the viability and expressive power of the proposal. The main contributions with respect to Datalog are the addition of hypothetical queries and universal quantifications. The language is designed in a flexible way in order to support different constraint domains. The implementation includes several domain instances, and it also supports aggregates as usual in database languages. The formal semantics of the language is defined by a proof-theoretic calculus, and for the operational mechanism we use a stratified fixpoint semantics, which is proved to be sound and complete w.r.t. the former. The resulting fixpoint semantics constitutes a suitable foundation for the system implementation. Hypothetical queries and aggregates require a more elaborated notion of dependency graph than the common one used in Datalog, which yields to an original stratification for databases. Moreover, the first one demands a sophisticated mechanism that implies a temporary, dynamic grow of the database. Finally, some concrete examples for the system for showing the expresivity of the proponsal are presented.

Keywords

Deductive Database Systems, Constraints, Hereditary Harrop Formulas, Fixpoint Semantics.

Resumen

Las fórmulas de Harrop hereditarias con restricciones se han usado como base para lenguajes de programación lógica. Al igual que la programación lógica da soporte a lenguajes de bases de datos deductivas como Datalog (con restricciones), este marco se usa como base para un sistema de bases de datos deductivas que mejora la expresividad de los sistemas aparecidos hasta el momento.

En este trabajo, en primer lugar, se hace una comparativa entre distintos sistemas de bases de datos deductivas, así como de sistemas de bases de datos con restricciones. Más adelante se muestran los resultados teóricos que fundamentan el lenguaje $HH_{\neg}(\mathcal{C})$ y una implementación concreta de este esquema que demuestra la viabilidad y expresividad del esquema. Las principales aportaciones con respecto a Datalog son el uso de consultas hipotéticas y cuantificadores universales. El sistema está diseñado de forma que soporta diferentes dominios de restricciones. La implementación incluye diferentes dominios concretos y también funciones de agregación que son habituales en otros lenguajes de bases de datos. El significado del lenguaje se define mediante una semántica de pruebas y el mecanismo operacional se define mediante una semánica de punto fijo que es correcta y completa con respecto a la primera. La semántica de punto fijo fundamenta la implementación del sistema. Para el cómputo de las consultas hipotéticas y de las funciones de agregación se hace uso de una noción de grafo de dependencias más compleja que la que usa Datalog. Este grafo de dependencias se utiliza para definir una estratificación para la base de datos. Además se debe tener en cuenta que la implementación debe estar diseñada contando con el posible crecimiento temporal de la base de datos debido a las consultas hipotéticas. Finalmente se presentan ejemplos del sistema que muestran la expresividad del lenguaje.

Palabras clave

Sistemas de bases de datos deductivas, restricciones, fórmulas hereditarias de Harrop, semántica de punto fijo.

Agradecimientos

Qusiera agradecer a Francisco Javier López Fraguas por darme la oportunidad de trabajar en el Grupo de Programación Declarativa Avanzada y dedicarme a la investigación durante estos años.

Gracias también a Susana, Jaime y Fernando por introducirme en el mundo de la investigación y por la dedicación mostrada en sus explicaciones y revisiones que han permitido realizar este trabajo.

Índice general

Capítulo 1

Introducción

Se puede decir de manera informal que una base de datos (BD) es un conjunto de datos relacionados entre sí que pertenecen a un mismo contexto y se almacenan para su uso posterior. Desde este punto de vista, un conjunto de libros impresos y textos, o una biblioteca, se pueden considerar también bases de datos, dado que se almacenan ordenados para que se puedan consultar posteriormente. En la actualidad, dado el desarrollo de la informática y la electrónica, existen un gran número de bases de datos en formato digital que permiten diversas soluciones eficientes para el almacenamiento de datos.

Los sistemas gestores de bases de datos (SGBD) permiten almacenar y posteriormente acceder a los datos de forma rápida, estructurada y eficiente. Las aplicaciones más usuales se usan en la gestión de datos de empresas así como en entornos científicos con el objeto de almacenar la información experimental. Una base de datos relacional es una base de datos que cumple con el modelo relacional, que es el modelo más utilizado en la actualidad para implementar bases de datos. Sin embargo, este modelo se ha mostrado insuficiente en la expresión de consultas a la base de datos. Un defecto importante es la ausencia de recursión, que impide expresar consultas sencillas como el cierre transitivo de un grafo. Este tipo de consultas pueden expresarse en lógica de predicados de primer orden. Dado que el objeto de este trabajo es describir un sistema de bases de datos deductivas se hace necesario introducir previamente conceptos de pensamiento deductivo así como su relación con la lógica matemática y con la programación lógica.

El pensamiento deductivo parte de categorías generales para hacer afirmaciones sobre casos particulares. Se habla de razonamiento deductivo cuando observando una entidad muchas veces se infiere lo visto en todas las entidades de una misma especie. La conclusión debe poder derivarse necesariamente de las premisas aplicando a éstas algunas de las reglas de inferencia según las reglas de transformación de un sistema deductivo o cálculo lógico.

Históricamente, los ordenadores se han programado utilizando lenguajes muy cercanos a las peculiaridades de la propia máquina: operaciones aritméticas simples e instrucciones de acceso a memoria. Un programa escrito de esta manera suele no ser explícito en cuanto a su propósito, incluso aunque el lector sea un programador for-

mado. Actualmente, estos lenguajes pertenecientes al paradigma de la programación imperativa han evolucionado de manera que ya no son tan crípticos. En cambio, la lógica matemática es una manera sencilla de expresar formalmente algunos problemas complejos y de resolverlos mediante la aplicación de reglas e hipótesis. De ahí que el concepto de programación lógica resulte atractivo en diversos campos donde la programación tradicional no funciona adecuadamente. La programación lógica se usa en aplicaciones de inteligencia artificial y sistemas relacionados:

- Sistemas expertos, donde un sistema de información imita las recomendaciones de un experto sobre algún dominio de conocimiento.
- Demostración automática de teoremas, donde un programa puede generar nuevos teoremas sobre una teoría existente.
- Reconocimiento de lenguaje natural, donde un programa es capaz de comprender, con limitaciones, la información contenida en una expresión lingüística humana.

La intersección de las bases de datos, la lógica y la inteligencia artificial ha dado lugar a las bases de datos deductivas (BDD). Una base de datos deductiva es un sistema que incluye mecanismos para definir reglas que pueden inferir o deducir información adicional af los hechos almacenados en una base de datos. Las reglas se especifican en un lenguaje declarativo y un motor de inferencia, o mecanismo de deducción, es capaz de inferir nuevos hechos de la base de datos. El modelo usado para las bases de datos deductivas está estrechamente relacionado con el modelo de datos relacional. También está muy relacionado con el lenguaje Prolog y, principalmente, con el lenguaje Datalog, un lenguaje muy cercano a Prolog orientado al campo de las BD. Aunque se ha investigado activamente y producido muchas propuestas, la semántica operacional de Datalog es aún un campo de investigación para la consecución de consultas eficaces.

Las BDD y sus lenguajes de consulta han ido haciéndose cada vez más necesarias en diferentes áreas de la ciencia. Lenguajes como Datalog han adquirido mucho interés en estos campos a la hora de buscar sistemas seguros con una teoría bien fundamentada. Podemos encontrar un gran número de sistemas de BDD como XSB [63], bddbddb [36], LDL++ [3], DES [62], ConceptBase [27], QL [57] y DLV [39].

El objeto de este trabajo es hacer una descripción precisa y detallada de un sistema de base de datos deductivas con restricciones basado en el esquema $HH_{\neg}(\mathcal{C})$ (*Hereditary Harrop formulas with Negation and Constraints*). Para ello se explican detenidamente los fundamentos teóricos, que sirven de base para entender el sistema y su mecanismo operacional y se hace una descripción técnica de los módulos que componen el sistema.

Para situar el marco del trabajo se hace, en el segundo capítulo, un estudio sobre bases de datos deductivas y bases de datos con restricciones; además se explican los problemas a los que es necesario enfrentarse a la hora de incorporar negación a los sistemas de bases de datos deductivas. Es importante incorporar negación dado que sin ella no se consigue completitud con respecto al álgebra relacional.

El esquema $HH(\mathcal{C})$, que no incorpora negación, fue propuesto originalmente en [37] como un lenguaje de programación lógica con restricciones extendido. Este esquema combina la lógica intuicionista que procede de las fórmulas de Harrop hereditarias (HH) junto con el uso de restricciones. La ventaja de la lógica subyacente es que mejora la expresividad, dado que permite objetivos que incluyen disyunciones, implicaciones anidadas y cuantificadores universales. Las restricciones aportan mayor eficiencia, encapsulación de partes del programa y resolución de problemas que la programación lógica clásica no puede resolver.

Para adaptar $HH(\mathcal{C})$ al campo de las BD es necesario introducir negación al esquema, resultando $HH_{\neg}(\mathcal{C})$ [50]. Lo novedoso de esta aproximación es la aplicación de los elementos expresivos que provienen de la lógica HH al campo de las bases de datos. Esto hace que, por ejemplo, se puedan formular consultas hipotéticas a una base de datos, dado que dispone de implicación, así como consultas que incluyen el cuantificador universal. Por tanto, en este trabajo presenta $HH_{\neg}(\mathcal{C})$ como un esquema para BDD con elementos expresivos que no manejan otros sistemas de BDD.

Al igual que ocurre con Datalog, para garantizar la terminación de los cómputos se definen la noción de grafo de dependencias y técnicas de estratificación, apropiadas para el nuevo esquema. Para dar significado a los programas se presenta una semántica de punto fijo estratificada para $HH_{\neg}(\mathcal{C})$ que ayuda a entender el mecanismo operacional y también se define una semántica de pruebas que proporciona referencia para la corrección y completitud del esquema.

En el trabajo se describe con todo detalle el primer prototipo del sistema implementado y basado en el esquema.

Obsérvese que el esquema $HH_{\neg}(\mathcal{C})$ es paramétrico con respecto al sistema de restricciones. Aquí $\mathcal{C}$ representa un sistema genérico que, al ser sustituido por uno concreto, dará lugar a una instancia del esquema. Esto ha permitido diseñar la implementación de forma que, siguiendo los fundamentos teóricos, el núcleo de la misma es independiente del sistema de restricciones concreto, cuyo funcionamiento es una caja negra para la teoría. Así mismo se han diseñado varios sistemas de restricciones para el esquema: booleanos, dominios finitos y reales. Tanto el núcleo del sistema como los sistemas de restricciones están implementados en SWI-Prolog [82].

Otra característica del sistema es la incorporación al sistema de funciones de agregación, que son habituales en los lenguajes de bases de datos. Por otro lado, cuando se formulan consultas hipotéticas, en ocasiones, es necesario el recálculo del punto fijo de la base de datos debido a que el programa aumenta. Mediante el almacenamiento del punto fijo en la base de datos ha disminuido el número de casos en que este recálculo es necesario haciendo el sistema más eficiente. Se muestran también mejoras del sistema de restricciones añadiendo a sus resolutores una fase de preresolución de restricciones cerradas.

A continuación, con el objetivo de mostrar las ventajas y expresividad de $HH_{\neg}(\mathcal{C})$ como lenguaje de bases de datos, se muestra un ejemplo de uso del esquema. Se considera la instancia $HH_{\neg}(\mathcal{FR})$, donde $\mathcal{FR}$ es un sistema de restricciones que combina dominios finitos y reales, que es un subconjunto del sistema de restricciones explicado en [15].

En el ejemplo se define una base de datos de las asignaturas cursadas por alumnos,

con información sobre *nombre de alumno, asignatura* y *calificación*. Se usa notación Prolog y además se usa `not(A)` para representar la negación de un átomo `A` y el símbolo `=>` representa una implicación anidada.

```
% curso(Alumno, Asignatura, Puntuación)
curso(angela, introduccion_programacion, 5.0).
curso(nicolas, introduccion_programacion, 7.0).
curso(david, introduccion_programacion, 2.0).

curso(angela, programacion_declarativa, 3.0).
```

Para definir la condición de haber aprobado (y cursado) *introducción a la programación* y haber cursado *programación declarativa* para poder matricularse de la asignatura *programación declarativa avanzada* puede escribirse la vista:

```
% matricula(Alumno, Asignatura).
matricula(X,programacion_declarativa_avanzada):-
  curso(X,introduccion_programacion,Y), Y>=5.0
  curso(X,programacion_declarativa,Z).
```

Dado que el esquema incorpora negación, una consulta a esta base de datos podría ser quién *no* puede matricularse en *programación declarativa avanzada*:

```
not(matricula(Alum,programacion_declarativa_avanzada)).
```

La respuesta es `Alum \= angela`, es decir, cualquier alumno distinto de Angela (que es la única que cumple ambos requisitos). Como se ha dicho, una de las novedades que incluye el lenguaje es la posibilidad de hacer consultas hipotéticas y una de las ampliaciones que presenta este trabajo es el uso de funciones de agregación. Un ejemplo de ambas es la siguiente consulta: suponiendo que el alumno José Luis sacara un 9.0 en *introducción a la programación* ¿cómo cambiaría la media de las calificaciones de los alumnos de esta asignatura?

```
curso(joseluis,introduccion_programacion,9.0)=>
  Avg=avg(curso(X,introducción_programacion,Z),Z).
```

La respuesta es `Avg = 5.75`.

El trabajo se divide en 8 capítulos con el siguiente contenido. En el capítulo 2, se da una visión panorámica de sistemas de bases de datos deductivas y sistemas de bases de datos con restricciones. Se trata de hacer una comparación prestando especial atención a características del sistema descrito, como son la negación y el uso de funciones de agregación. En el capítulo 3, se presenta esquema $HH_{\neg}(\mathcal{C})$, su sintaxis, la definición de sistema de restricciones y las semánticas que fundamentan este esquema. Una de

punto fijo y otra de pruebas que son equivalentes. En el capítulo 4, se da una breve descripción del sistema orientada al usuario, explicando cómo se hace la gestión de la base de datos, así como la formulación de consultas. En el capítulo 5, se muestra una implementación concreta de la semántica de punto fijo basada en $HH_{\neg}(\mathcal{C})$ en SWI-Prolog, detallando la forma en que se han resuelto los problemas surgidos al llevar a cabo este sistema. En el capítulo 6, se describe la implementación de los sistemas de restricciones y resolutores que se usan para resolver las restricciones que se envían desde el núcleo del sistema. Partiendo de la implementación presentada en los capítulos anteriores, en el capítulo 7, se muestra la ejecución de algunos ejemplos de aplicación del sistema, mostrando las posibilidades que ofrece la sintaxis del lenguaje. Finalmente en el capítulo 8, aparece un resumen de los objetivos conseguidos con este trabajo y una breve aproximación a las líneas de trabajo futuro.

Capítulo 2

Una visión global de bases de las datos deductivas con restricciones

En este capítulo se muestran, en primer lugar, unas nociones previas comunes a los sistemas de bases de datos deductivas. Se muestra el problema que supone incorporar negación a los esquemas estilo Datalog. A continuación se muestra una panorámica histórica de las bases de datos deductivas. Después se repasan algunas implementaciones concretas. Finalmente se hace un recorrido similar con las bases de datos con restricciones.

2.1. Bases de datos deductivas

Los sistemas de bases de datos deductivas son aquellos en los que se pueden hacer deducciones a través de inferencias. Se basan principalmente en reglas y hechos que se almacenan en la base de datos. Las bases de datos deductivas se llaman también bases de datos lógicas, dado que se basan en lógica matemática. Se pueden ver los sistemas de BDD como una extensión de los sistemas relacionales.

Una característica de las BDD compartida con las bases de datos relacionales es la propiedad de ser *declarativos*. Esto significa que permite al usuario hacer una consulta o actualización diciendo específicamente lo que quieren, en vez de cómo realizar la operación.

La lógica ha aportado un gran número de contribuciones a las bases de datos, entre las que se pueden destacar [43]:

- Formalización de base de datos, consulta y respuesta a una consulta.
- El reconocimiento de que la programación lógica extiende a las bases de datos relacionales.

- Comprensión clara de la semántica de múltiples clases de bases de datos que incluyen formas alternativas de negación y disyunción.
- Comprensión de las relaciones entre la teoría de modelos, teoría de punto fijo y procedimientos de demostración.
- Comprensión de las restricciones de integridad y la forma en que se pueden usar para realizar actualizaciones, optimizando la semántica de consultas, respuestas cooperativas y mezcla de bases de datos.
- Formalización y soluciones a los problemas de actualización de datos y de vistas.
- Comprensión de la recursión y de la recursión acotada, y la forma en que pueden ser implementadas prácticamente.
- Comprensión de las relaciones entre los sistemas basados en la lógica y los sistemas basados en el conocimiento.
- Formalización de la gestión de la información incompleta en sistemas de bases de conocimiento.
- Correspondencia entre formalismos alternativos de razonamiento no monótono y bases de datos y de conocimiento.

La mayoría de los sistemas deductivos están inspirados en Prolog. A la hora de implementar se debe tener en cuenta:

- La estrategia de primero en profundidad de Prolog conduce a cómputos posiblemente infinitos debido a los predicados recursivos, incluso con programas sin negación o también en ausencia de símbolos de función o aritméticos.
- La *corrección* y *completitud* del método de evaluación.
- En una aplicación típica de bases de datos, la cantidad de información es lo suficientemente grande como para ser parte del almacenamiento secundario. El acceso eficiente a estos datos es crucial para un buen rendimiento del sistema.

En resumen, un objetivo primordial de las bases de datos deductivas es tratar con un superconjunto del álgebra relacional (AR) que permita recursión sin que ello conlleve un gran número de accesos a disco, que sea terminante y con un método de evaluación correcto y completo.

2.1.1. Modelo lógico como base de datos

En bases de datos basadas en la lógica se suele dividir la información en dos categorías:

1. Parte *extensional* de la base de datos. Aquellos datos que se corresponden con los hechos del sistema lógico. Los representaremos como predicados con argumentos constantes.

   Por ejemplo, `madre(ana,maría)` dará la información sobre que la madre de Ana es María.

2. Parte *intensional* de la base de datos. Reglas de programa que aparecerán habitualmente en notación Prolog:

   ```
   p(X0) :-  q1(X1),...,qn(Xn).
   ```

   Los predicados intensionales se asimilan generalmente a las vistas en las bases de datos convencionales.

Los objetivos Prolog se verán como consultas a la base de datos.

2.1.2. Negación estratificada

Un sistema de bases de datos basado en un modelo lógico debe incorporar negación en los cuerpos de las consultas para ser completo con respecto al AR [75]. Al introducir la negación se da el problema de que un programa puede tener distintos significados en función del orden de evaluación de los predicados. Por ejemplo, dado el dominio `[a,b,c]` y el programa:

```
p(a).
p(b).
q(X):- not(p(X)).
```

El valor semántico del predicado `q` tendrá distinto significado:

- Si se calcula el significado de `q` antes que el de `p`, se tendrá: {`q(a),q(b),q(c)`}.
- Si se calcula entre `p(a)` y `p(b)`, se tendrá: {`q(b),q(c)`}.
- Si se calcula después del significado de `p` se tendrá: {`q(c)`}.

Para evitar este problema se impone la restricción de que cuando un predicado se define como negación de otro, se debe calcular el significado del segundo predicado después del significado del primero. En el ejemplo, se debe calcular el significado íntegro de `p` antes que el de `q`.

El significado de un programa con negación viene dado generalmente por un modelo pretendido. Se puede encontrar un amplio abanico de ejemplos de modelos para negación [11, 1, 67, 16, 79, 61].

Para resolver este problema se usan técnicas de estratificación. La idea que hay tras la estratificación es un cálculo del significado de los predicados por capas o estratos.

Antes de evaluar un programa se clasifican los predicados de éste asignando a cada uno un estrato. En el primer estrato están los hechos y los predicados que no tienen negación en los cuerpos de sus cláusulas. En general se debe cumplir que si un predicado aparece negado en el cuerpo de las cláusula de otro, el primero debe estar en un estrato superior al segundo.

Cuando se va a calcular el significado del programa se calcula primero el significado de los predicados en el primer estrato, luego los del segundo y así hasta el último estrato.

Ejemplo 1 El programa

```
r(b).
q(X):- not(r(a)).
p(X):- not(q(a)).
```

Se puede estratificar asignando a `r` el estrato 1, a `q` el estrato 2 y a `p` el estrato 3. Intuitivamente, significa que se calcula primero el significado de `r`, luego el significado de `q` y finalmente el significado de `p`. □

Ejemplo 2 El programa

```
p(X):- not(p(X)).
```

No es estratificable dado que el predicado `p` debería aparecer en dos estratos distintos simultáneamente. □

2.1.3. Revisión histórica de las bases de datos deductivas

Se puede encontrar el origen de las bases de datos deductivas en trabajos relacionados con demostradores automáticos de problemas y en la programación lógica. En un estudio realizado por Minker [42] se sugiere que Green y Raphael [17] fueron los primeros en relacionar la demostración de teoremas y la deducción en bases de datos. Estos desarrollaron una serie de sistemas consulta/respuesta que usaban una versión del principio de resolución de Robinson [60], demostrando así que que la deducción se puede usar de manera sistemática en el contexto de las bases de datos.

Otros de estos primeros sistemas son MRPPS [44], DEDUCE [12] y DADM [33]. El primero, MRPPS, era un intérprete que fue desarrollado por el grupo de Minker entre 1970 y 1978. De él se pueden destacar una de las primeras referencias a consultas recursivas. DEDUCE fue implementado por IBM en la década de los 70, usaba reglas basadas en cláusulas de horn recursivas lineales por la izquierda. Finalmente DADM hizo hincapié en la diferencia entre la parte extensional e intensional de la base de datos y estudió la representación de la intensional en forma de de *grafos de conexión*.

En 1976, van Emdem y Kowalski [78] mostraron que el mínimo punto fijo de un programa lógico con cláusulas de Horn era el modelo mínimo de Herbrand. Esto fundamentó la semántica de los programas lógicos, así como de las bases de datos deductivas, dado que el cómputo de punto fijo es la semántica operacional asociada a las bases de datos deductivas (al menos, a aquellas basadas en una evaluación ascendente).

Los primeros trabajos, antes mencionados, se centraron en buscar los objetivos del campo de BDD y el desarrollo de fundamentos semánticos. La siguiente fase se centró en el desarrollo de evaluación eficiente de consultas. Henschen y Naqvi propusieron una de las primeras técnicas eficientes para evaluar consultas en el contexto de las bases de datos [24]. Tras esto, en un artículo de Ullman se fijó un marco para la implementación [76]. Para ello el autor se centró no sólo en técnicas para evaluar consultas sino también llamó la atención sobre el problema de la no terminación.

El área de las bases de datos deductivas alcanzó gran importancia en 1984 con el comienzo de tres importantes proyectos. El proyecto Nail! en Stanford, LDL en Austin

y el proyecto de bases de datos deductivas ECRC representan la mayor contribución a las bases de datos deductivas fuera de las universidades. Se ven a continuación estos y otros proyectos dentro del campo de las BDD:

El proyecto ECRC fue coordinado por J. M Nicolas. La primera fase llevó al estudio de los algoritmos de desarrollo de los primeros prototipos, comprobación de integridad y un prototipo inicial que exploraba la comprobación de consistencia [9]. La segunda fase trajo prototipos más funcionales: Megalog, DedGin, EKS-V1. El sistema EKS daba soporte a restricciones de integridad y algunas funciones de agregación que usaban recursión [38]. De este trabajo se derivan investigaciones como las que lleva a cabo el Groupe Bull que desarrolla bases de datos deductivas comerciales y orientadas a objetos. La tecnología deductiva deriva del prototipo EKS.

El proyecto LDL comenzó también en 1984. En 1986 se vio que la combinación de Prolog con las bases de datos relacionales no era una solución satisfactoria, así que comenzaron con el desarrollo de técnicas ascendentes [74]. El prototipo LDL se desarrolló en 1988 y tuvo nuevas versiones entre 1989 y 1991. Este fue el primer sistema de bases deductivas de propósito general que estuvo disponible. Este prototipo soportaba negación estratificada y era compilado por un sistema que producía código C. Una presentación del lenguaje LDL se encuentra en [46]. El sistema LDL++ en MCC es el sucesor directo que comenzó en 1991. Este sistema incluye negación no estratificada y funciones de agregación [84].

El proyecto Nail! (*Not Another Implementation of Logic!*) comenzó en Stanford en 1985 siguiendo las ideas que aparecen en [76]. En colaboración con el grupo MCC apareció el primer artículo sobre conjuntos mágicos (*Magic Sets*) [6]. Se desarrolló un prototipo inicial [45] y finalmente fue abandonado dado que el paradigma puramente declarativo no resultaba cómodo para la realización de muchas aplicaciones.

El proyecto Aditi comenzó en 1988 en la Universidad de Melbourne. Las principales contribuciones de este proyecto son la formulación de una evaluación ingenua que ha sido muy usada en trabajos posteriores [5], la adaptación de conjuntos mágicos para programas estratificados [4], indexación y optimización de programas con restricciones [34]. El trabajo del grupo se encaminó hacia el desarrollo de su prototipo, haciendo un énfasis notable en las relaciones residentes en disco. Todas las operaciones relacionales son tratadas como residentes en disco. Se puede ver una visión general de este sistema en [77].

El sistema ConceptBase desarrollado por la Universidad de Passau y Aachen desde 1987 trata de combinar reglas deductivas con un modelo de datos semántico. Los principales aspectos del lenguaje se presentan en [28]. El sistema también tiene soporte de restricciones de integridad y se puede encontrar una descripción en [29]. ConceptBase se ha usado en numerosas aplicaciones en universidades europeas y tiene una versión comercial.

El proyecto CORAL pertenece a la Universidad de Wisconsin, comenzó en 1988. La idea original era el desarrollo del algoritmo de plantillas mágicas (*Magic Templates*) [52], que ofrece la posibilidad de usar tuplas no cerradas. Este proyecto aporta grandes contribuciones en el desarrollo de semánticas de multiconjuntos para programación lógica y optimización cuando se trata de comprobaciones de duplicados [40], los primeros resultados de técnicas ascendentes eficientes con respecto a espacio [48, 71] y evaluación de programas con funciones de agregado [56], entre otras. Los resultados de que la evaluación ascendente domina asintóticamente a la descendente en el contexto de programas con cláusulas de Horn se obtuvieron a través de este proyecto [70]. El primer prototipo del sistema CORAL fue operativo en 1990. Esta versión soportaba agregación no estratificada y negación usando un algoritmo propuesto en [53]. Una visión general del sistema se puede encontrar en [14] y la implementación aparece descrita en [54]. La extensión que soporta características orientadas a objetos es Coral++ y aparece descrita en [69].

El proyecto XSB es otro trabajo relacionado, coordinado por D.S. Warren. Desarrollaron un sistema que soporta negación estratificada y agregación (además de un meta-intérprete para programas bien fundamentados), tuplas no cerradas y relaciones residentes en disco. La implementación se basa en la resolución OLDT [73]. WAM [80], una máquina abstracta para implementar sistemas Prolog, se adaptó para usar la evaluación descendente que se usa en XSB.

El sistema educativo DES es un sistema de bases de datos deductivas desarrollado por la Universidad Complutense [62], es gratuito y de código abierto. La última versión es DES 2.0 que utiliza los lenguajes de programación Datalog y SQL. Soporta negación estratificada, depuración declarativa, generación de casos de prueba para vistas SQL, funciones y predicados de agregación, y predicados *join*, restricciones de integridad y tablas memo [68]. Se trata de un sistema también implementado en Prolog.

2.1.4. Implementaciones de bases de datos deductivas

En [51] se encuentra una tabla comparativa de varios sistema que se muestra adaptada en la figura 2.1.4. Los parámetros sobre los que se hace comparativa son:

1. *Recursión* (Rec.): Muchos de los sistemas permiten usar recursión general. Sin embargo, algunos limitan la recursión a una serie de casos restringidos relacionados con búsqueda de grafos.

2. *Negación*: La mayoría de los sistemas permiten negación en las reglas. Cuando esto ocurre suele haber más de un punto fijo minimal y el sistema debe seleccionar cuál de ellos en función del modelo pretendido.

3. *Agregación*: Un problema parecido al de la negación aparece con la agregación (suma, promedio, etc). Esto hace que aparezca más de un modelo minimal que se debe discriminar.

Nombre	Desarrollado	Ref.	Rec.	Negación	Agregación
Aditi	U. Melbourne	[77]	Sí	Estratificada	Estratificada
Concept Base	U. Aachen	[29]	Sí	Localmente Estratificada	No
CORAL	U. Wisconsin	[14]	Sí	Modularmete Estratificada	Modularmente Estratificada
EKS	ECRC	[38]	Sí	Estratificada	Estratificada
LDL LDL++	MCC	[13]	Sí	Estratificada Restringida	Estratificada Restringida
XSB	SUNY Stony Brook	[63]	Sí	Bien Fundada	Modularmente Estratificada
DES	U. Complutense	[62]	Sí	Estratificada	Estratificada

Figura 2.1: Comparativa de distintas implementaciones de sistemas de bases de datos deductivas.

2.2. Bases de datos con restricciones

Una de las ventajas del uso de restricciones en el contexto de programación lógica es la capacidad de estas de tratar con colecciones de datos infinitos mediante el uso de representaciones finitas. Las bases de datos con restricciones [35] heredan esta característica.

Ejemplo 3 Se puede determinar un rectángulo por su esquina inferior izquierda (X_1, Y_1) y su esquina superior derecha (X_2, Y_2) y este hecho se puede representar mediante la siguiente cláusula

```
rectangle(X1,Y1,X2,Y2,X,Y)  :- X>=X1, X=<X2, Y>=Y1, Y=<Y2.
```

Nótese que un rectángulo contiene un conjunto infinito de puntos que se representan de forma finita mediante el uso de restricciones reales. Desde la perspectiva de las bases de datos, se trata de una característica interesante, dado que fueron concebidas para tratar con piezas finitas de información. Mediante el uso de restricciones se hace posible tratar con conjuntos de datos potencialmente infinitos. □

2.2.1. Revisión histórica de bases de datos con restricciones

La investigación en bases de datos con restricciones (BDR) comenzó con el objetivo de definir una versión de la programación lógica con restricciones orientada al campo de las bases de datos, de la misma forma que se había definido Datalog como la versión de bases de datos de Prolog. El primer objetivo fue usar técnicas ascendentes para procesar reglas Datalog usando además restricciones. Así se hacía posible el uso de la programación lógica con restricciones para aplicaciones de manera que los datos podían representarse como conjuntos de restricciones (por ejemplo, datos espaciales). Según avanzaba la investigación resultó que el problema de soportar recursión en

presencia de restricciones no llegaba a ser resuelto de manera satisfactoria. Por tanto, en este campo se avanzó sobre todo centrándose en el caso no recursivo. Los lenguajes de consultas no recursivos con restricciones llevaron a la investigación de problemas interesantes y nada triviales, y estas aplicaciones todavía se usan en un gran número de dominios.

La idea de usar restricciones para objetivos de representación se había discutido en el campo de las matemáticas (por Whitney [81] por ejemplo) y en el de la investigación operativa (por Dantzig [47]), y se había usado en algunas aplicaciones de gestión de datos espaciales (la más notable CAD/CAM [65]).

En el campo de las bases de datos, Kanellatis et al [31] son los primeros en definir un marco de trabajo sistemático para el uso de restricciones como modelo de datos complejos y lenguajes de consulta sobre dichos datos. Se encuentran, sin embargo, trabajos previos sobre dominios específicos de aplicaciones que se pueden considerar como precursores de las BDR.

Se han definido álgebras y cálculos para aplicaciones concretas siguiendo algunas líneas del modelo relacional. Uno de los usos de estos sistemas es el modelado de tipos concretos de datos temporales, denominados eventos, y que aparecen en intervalos regulares. Este modelo aparece descrito en [30, 7] y también en el capítulo 13 de [35].

Otro modelo parecido es la propuesta de [21] para la representación de información de espacio en un GIS (*Geographic Information System*), como la intersección de semiplanos. El lenguaje de [21] es un caso particular del lenguaje de consulta con restricciones lineal general y sin proyección. Otra propuesta en la misma dirección es la que aparece en [22], en el que se trata de incluir información de dependencia del dominio en la base de datos.

2.2.2. Implementación de sistemas de bases de datos con restricciones y aplicaciones

El sistema MLPQ (*Management of Linear Programming Queries*), es un sistema de bases de datos con restricciones para bases de datos con restricciones lineales. Se desarrolló por la Unversidad de Nebraska-Lincoln. La primera versión se presentó en [59]. Esta versión incluye consultas SQL y programación lineal como funciones básicas para implementar funciones de agregación de máximo y mínimo. La segunda versión se presentó en [32]. Esta versión incluye consultas Datalog recursivas y no recursivas y una interfaz gráfica de usuario con operadores espaciales de dos dimensiones.

Se pueden encontrar dos grandes aplicaciones del MLPQ: la investigación operativa y el tratamiento con datos espaciales y espacio temporales.

El sistema DISCO (*Datalog with Integer Set of COnstraints*) es un sistema de bases de datos que implementa Datalog con restricciones booleanas de conjuntos de variables sobre conjuntos finitos o infinitos de enteros. Fue desarrollado por la Universidad de Nebraska. La primera versión del sistema fue presentada por Revesz en [10].

La segunda versión de DISCO está descrita en [64], esta versión incluye restricciones de desigualdad e igualdad booleana sobre conjuntos de enteros. En general la

igualdad y desigualdad booleana no puede ser usada de manera conjunta.

El prototipo DEDALE es una de las primeras implementaciones de un sistema de bases de datos basado en un sistema de restricciones lineales. Es un proyecto de INRIA con el grupo VERSO y el grupo VERTIGO. El prototipo DEDALE se utiliza para aplicaciones geométricas en diversas áreas como GIS o bases de datos espacio temporales. El sistema se describe en [20] y su modelo de datos en [19].

Otras aplicaciones de las bases de datos con restricciones son:

- En **visión por computador** para la indexación de bases de datos de imagenes en función de su forma y contorno [26, 18].
- En **bioinformática** para el desarrollo de un autómata para la descodificación del genoma humano, que es uno de los problemas más importantes en bioinfomática que se ha tratado usando el sistema LDL [58].
- En **modelado de entorno** por ejemplo para el desarrollo de mapas térmicos que se usan para evitar la propagación de fuego. [23].

Capítulo 3

Fundamentos teóricos del esquema $HH_{\neg}(\mathcal{C})$

Los primeros formalismos sobre las fórmulas de Harrop hereditarias con restricciones [37, 15] estaban orientados a ampliar la expresividad de la programación lógica añadiendo la implicación, la disyunción y el cuantificador universal en los objetivos, así como el uso de restricciones.

A pesar de la gran expresividad de $HH(\mathcal{C})$ como lenguaje de programación, la carencia de un mecanismo para expresar negación hace que este esquema no sea completo como lenguaje de BD. Por tanto se extendió la lógica de base con negación y se formalizó el esquema ampliado. En consecuencia, fue necesario definir nociones de estratificación y de grafo de dependencias apropiadas para el nuevo esquema. Añadiendo la negación se consigue un lenguaje de base de datos deductivas con restricciones completo con respecto al AR y además con mayor expresividad que otros lenguajes. Las nociones de estratificación han sido útiles también para el cálculo de funciones de agregación. Cuando se calcula el significado de un átomo negado $\neg A$ es necesario conocer completamente el significado del átomo A, como se ha visto en la sección 2.1.2. De la misma forma, cuando se calcula una función de agregación sobre un átomo A, el significado de este átomo debe estar también totalmente calculado.

En este capítulo se precisa, en primer lugar, la sintaxis del lenguaje $HH_{\neg}(\mathcal{C})$, las condiciones que se deben imponer a $\mathcal{C}$ para que sea sistema de restricciones. A continuación, se definen dos semánticas para dicho lenguaje: una semántica de pruebas con la cual se pueden hacer demostraciones de resolución de objetivos y una semántica de punto fijo que sirve de base para la implementación del sistema. Para el desarrollo de la semántica de punto fijo se hace necesario definir los conceptos de estratificación y de grafo de dependecias que se usan para el que el sistema soporte negación. Finalmente se dan resultados de corrección y completitud entre la semántica de pruebas y la semántica de punto fijo.

3.1. Sintaxis

Se distinguen un conjunto de *símbolos de predicados definidos* para construir átomos que representan relaciones en la base de datos, y un conjunto de *símbolos de predicado no definidos* para construir restricciones; estos últimos incluyen el símbolo de predicado de igualdad $\approx$ y el operador de comparación $\leq$. Para construir términos se dispone de un conjunto de símbolos de constantes y de operaciones, y de un conjunto de variables. Se usará la notación A para representar átomos, C para restricciones, Γ para conjuntos de restricciones y t para términos. El esquema $HH_{\neg}(\mathcal{C})$ es paramétrico respecto a un sistema de restricciones $\mathcal{C}$.

Los sistemas de restricciones que se consideran pertenecen a un sistema génerico $\mathcal{C} = \langle \mathcal{L}_{\mathcal{C}}, \vdash_{\mathcal{C}} \rangle$, donde:

- $\mathcal{L}_{\mathcal{C}}$ es el lenguaje de restricciones.
- $\vdash_{\mathcal{C}}$ es la *relación de deducibilidad*.

$\Gamma \vdash_{\mathcal{C}} C$ denota que la restricción C se deduce partiendo del conjunto de restricciones Γ en el sistema de restricciones $\mathcal{C}$.

A $\mathcal{C}$ se le deben imponer unas condiciones mínimas para constituir un sistema de restricciones:

- $\mathcal{L}_{\mathcal{C}}$ debe contener, al menos, a todas las fórmulas de primer orden que se construyen usando:
  - $\top$ (*true*), $\bot$ (*false*),
  - símbolos de predicado construidos,
  - los conectivos $\wedge, \neg$ y el cuantificador universal $\exists$.
- Con respecto a $\vdash_{\mathcal{C}}$:
  - Debe incluir las reglas de inferencia asociadas a las conectivas y cuantificadores mencionados, que son válidos para la lógica intuicionista con igualdad.
  - Debe ser compacto y cerrado bajo sustitución.

Nótese que $\mathcal{C}$ debe soportar negación; esto se debe a la incorporación de la negación en *HH*, lo cual lleva a que la negación se propague al sistema de restricciones. Las condiciones antes expuestas son las condiciones mínimas que se le imponen a un sistemas de restricciones, pero se pueden ampliar con más conectivas como $\vee$ y sus reglas de inferencia correspondientes, que están incorporados en todos los sistemas que se han implementado, como se puede ver en el capítulo 6. En concreto un ejemplo es el sistema de restricciones $\mathcal{R}$ donde $\mathcal{L}_{\mathcal{R}}$ es un lenguaje de primer orden con negación y además se cumple que $\Gamma \vdash_{\mathcal{R}} C$ si y sólo si $Ax_{\mathcal{R}} \cup \Gamma \vdash_{\mathcal{R}} C$ donde $Ax_{\mathcal{R}}$ es la axiomatización de Tarski de los número reales (Tarski, 1995).

El sistema de restricciones debe comprobar la satisfactibilidad de las respuestas en el dominio de restricciones. Se dice que una restricción C es $\mathcal{C}$-satisfactible si $\emptyset \vdash_{\mathcal{C}} \overline{\exists} C$, donde $\overline{\exists} C$ denota el cierre existencial de C. Se dice que C y C' son $\mathcal{C}$-equivalentes si se verifica $C \vdash_{\mathcal{C}} C'$ y $C' \vdash_{\mathcal{C}} C$.

Las fórmulas bien construidas en $HH_{\neg}(\mathcal{C})$ se pueden clasificar en cláusulas D, que definen relaciones en la base de datos y objetivos G, que definen las consultas a la base de datos. Se definen por recursión mutua como sigue:

$$D ::= A \mid G \Rightarrow A \mid D_1 \wedge D_2 \mid \forall x D$$
$$G ::= A \mid \neg A \mid C \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \Rightarrow G \mid C \Rightarrow G \mid \exists x G \mid \forall x G$$

Los programas, denotados como Δ, son conjuntos de cláusulas y representan bases de datos. No se permite la negación en la cabeza de una cláusula, pero sí en su cuerpo. Este hecho no resta expresividad puesto que si se quiere expresar la negación de un átomo `p(X)` en una cláusula, se puede definir una vista de la forma `q(X) :- not(p(X))`.

Con el fin de simplificar la notación, las cláusulas se escribirán como implicaciones con cabeza atómica al estilo habitual de la programación lógica, aunque obsérvese que el cuerpo de estas cláusulas puede contener restricciones, disyunciones, átomos negados, implicaciones y cuantificadores universales y existenciales. Cualquier Δ se puede reescribir siempre como un conjunto equivalente, $elab(\Delta)$, de cláusulas con implicaciones de cabeza atómica tal y como se define a continuación.

Definición 1 La *elaboración* de un programa Δ es el conjunto:

$$elab(\Delta) = \bigcup_{D \in \Delta} elab(D)$$

donde $elab(D)$ se define como:

- $elab(A) = \{\top \Rightarrow A\}$
- $elab(D_1 \wedge D_2) = elab(D_1) \cup elab(D_2)$
- $elab(G \Rightarrow A) = \{G \Rightarrow A\}$
- $elab(\forall x D) = \{\forall x D' \mid D' \in elab(D)\}$.

3.2. Semántica de pruebas

La manera más sencilla de explicar el significado de las bases de datos y las consultas en $HH_{\neg}(\mathcal{C})$ es utilizar una semántica de pruebas, es decir, determinar qué respuesta se puede deducir de un objetivo a partir de una base de datos mediante un sistema deductivo. El cálculo que fundamenta $HH_{\neg}(\mathcal{C})$ se denomina $\mathcal{UC}_{\neg}$ (*Uniform sequent calculus handling Constraints with Negation*). Se trata de un cálculo de secuentes que combina reglas de inferencia con la relación de de deducibilidad $\vdash_{\mathcal{C}}$ del sistema de restricciones. Los programas y los conjuntos de restricciones están en el lado izquierdo

de los secuentes y los objetivos aparecen en el lado derecho. Es decir, los secuentes son de la forma $\Delta; \Gamma \vdash G$, donde la notación $\Delta; \Gamma \vdash_{\mathcal{UC}_\neg} G$ significa que existe una deducción del secuente $\Delta; \Gamma \vdash G$ usando las reglas del cálculo $\mathcal{UC}_\neg$. Las reglas que definen el cálculo aparecen en la Figura 3.1.

Si se demuestra $\Delta; C \vdash_{\mathcal{UC}_\neg} G$ entonces C es la *restricción respuesta* a la consulta G en la base de datos Δ. La idea subyacente es que G es *deducible* de la base de datos Δ si se satisface la restricción C.

El cálculo $\mathcal{UC}_\neg$ es un cálculo uniforme en el sentido de Milner et al. [41], para el lenguaje $HH_\neg(\mathcal{C})$, ya que las demostraciones a las que da lugar están orientadas al objetivo. La novedad con respecto al los lenguajes presentados en [41] es la incorporación de restricciones y el manejo de la negación. Se explican las más relevantes y diferentes:

$$\frac{\Gamma \vdash_{\mathcal{C}} C}{\Delta; \Gamma \vdash C}\ (C) \qquad \frac{\Delta; \Gamma \vdash \exists x_1 \ldots \exists x_n((A' \approx A) \wedge G)}{\Delta; \Gamma \vdash A}\ (Clause)\,(*), \text{donde}$$

$$\forall x_1 \ldots \forall x_n(G \Rightarrow A') \text{ es una variante de una fórmula de } elab(\Delta)$$

$$\frac{\Delta; \Gamma \vdash G_i}{\Delta; \Gamma \vdash G_1 \vee G_2}\ (\vee)\ (i = 1, 2) \qquad \frac{\Delta; \Gamma \vdash G_1 \quad \Delta; \Gamma \vdash G_2}{\Delta; \Gamma \vdash G_1 \wedge G_2}\ (\wedge)$$

$$\frac{\Delta, D; \Gamma \vdash G}{\Delta; \Gamma \vdash D \Rightarrow G}\ (\Rightarrow) \qquad \frac{\Delta; \Gamma, C \vdash G}{\Delta; \Gamma \vdash C \Rightarrow G}\ (\Rightarrow_C)$$

$$\frac{\Delta; \Gamma, C \vdash G[y/x] \quad \Gamma \vdash_{\mathcal{C}} \exists y C}{\Delta; \Gamma \vdash \exists x G}\ (\exists)(**) \qquad \frac{\Delta; \Gamma \vdash G[y/x]}{\Delta; \Gamma \vdash \forall x G}\ (\forall)(**)$$

$$\frac{\Gamma \vdash_{\mathcal{C}} \neg C \quad \text{para todo } \Delta; C \vdash A}{\Delta; \Gamma \vdash \neg A}\ (\neg)$$

$(*)$ $x_1, \ldots, x_n$ nuevas para A

$(**)$ y debe ser nueva para la fórmula del secuente de la conclusión

Figura 3.1: Reglas del Cálculo $\mathcal{UC}_\neg$

- En la regla $(\exists)$ se trata de buscar una restricción que sirva de testigo para una demostración de un existencial, por ejemplo $(x * x \approx 2)$ representa el testigo de $\sqrt{2}$ el cual no puede ser representado mediante un término.
- $(Clause)$ representa el encadenamiento hacia atrás. Para probar los objetivos atómicos de la forma $A \equiv p(t_1, \ldots, t_n)$, es necesario utilizar una cláusula de programa con cabeza $A' \equiv p(t'_1, \ldots, t'_n)$, lo que da lugar a un nuevo objetivo $\exists x_1 \ldots \exists x_n((A' \approx A) \wedge G)$ que se resuelve mediante sucesivas aplicaciones de la regla $(\exists)$. Cuando se resuelve un objetivo existencialmente cuantificado, mediante el uso de la regla existencial, dará lugar a una búsqueda de una restricción que implique la igualdad $A' \approx A$ (lo que significa $t'_1 \approx t_1 \wedge \ldots \wedge t'_n \approx t_n$).
- En el caso de la regla $(\neg)$, la idea es que para que C sea una respuesta a la consulta $\neg A$, a partir de la base de datos Δ, debe ocurrir que si la restricción C' es una posible respuesta a la consulta del átomo sin negar A entonces $C \vdash_{\mathcal{C}} \neg C'$. Se dice que $(\neg)$ es en realidad una *metarregla* dado que su premisa considera todas las derivaciones posibles $\Delta; C \vdash A$ del átomo A. La inclusión de esta metarregla hace que se impongan unas condiciones de finitud. Es decir, es necesario

garantizar que haya un número finito de respuestas no equivalentes para cada átomo negado que aparece en un objetivo. Para ello se han usado técnicas de estratificación similares a las vistas en la sección 2.1.2. La estratificación se usa como base para una semántica de punto fijo que se verá en la siguiente sección.

Ejemplo 4 Considérese la siguiente base de datos para gestión de vuelos. Para este ejemplo se usará una instancia $HH_{\neg}(\mathcal{FR})$. Nótese que se usa notación Prolog habitual y que por tanto las cláusulas se sobreentiende que representan la cuantificación universal de las fórmulas sobre las variables libres.

```
flight(mad, par, 1.5).
flight(par, ny, 10).
flight(london, ny, 9).
travel(X,Y,T) :- flight(X,Y,D), T >= D.
travel(X,Y,T) :- flight(X,Z,T1), travel(Z,Y,T2), T >= T1+T2.
```

Los elementos del dominio `mad`, `par`, `lon` representan Madrid, París y Londres respectivamente. El predicado `flight(O,D,T)` constituye la parte extensional de la base de datos. Esta relación representa un vuelo entre el origen (`O`) y el destino (`D`) que dura un tiempo (`T`). En cambio, `travel(X,Y,T)` constituye la parte intensional de la base de datos, representa que un viaje desde `X` hasta `Y` que transcurre en un tiempo que debe ser mayor o igual que la suma de los tiempos de los viajes intermedios `T`.

Se puede formular la siguiente consulta hipotética a la base de datos: ¿Es posible volar desde Madrid hasta cualquier ciudad en un tiempo superior a `1.5`?

$$\texttt{fa}(\texttt{T}, (\texttt{T} > \texttt{1,5} \Rightarrow \texttt{ex}(\texttt{Y}, \texttt{travel}(\texttt{mad}, \texttt{Y}, \texttt{T})))).$$

La respuesta es `true`. Para mostrar la derivación de secuentes se considera el conjunto de la elaboración de Δ asociado a la base de datos:

$$\begin{array}{l}\Delta = \{flight(mad, par, 1{,}5),\ flight(par, ny, 10),\ flight(lon, ny, 9),\\ \forall x \forall y \forall t \forall d((flight(x, y, d) \wedge t \geq d) \Rightarrow travel(x, y, t)),\\ \forall x \forall y \ldots ((flight(x, z, t_1) \wedge travel(z, y, t_2) \wedge t \geq t_1 + t_2) \Rightarrow travel(x, y, t))\}\end{array}$$

y el objetivo

$$G \equiv \forall t\ (t > 1{,}5 \Rightarrow \exists y\ travel(mad, y, t)).$$

A continuación, se muestra la derivación del secuente $\Delta; \{\top\} \vdash G$. Se usan las siguientes abreviaturas:

- $\Gamma = \{\top, t > 1{,}5,\ y \approx par\}$
- $\Gamma' = \Gamma \cup \{x' \approx mad,\ y' \approx par,\ t' \approx t,\ d' \approx 1{,}5\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \vdash_{\mathcal{FR}} x' \approx mad \wedge \ldots \wedge t' \geq d'}{\Delta; \Gamma' \vdash x' \approx mad \wedge \ldots \wedge t' \geq d'}\ (C) \quad \dfrac{\Gamma' \vdash_{\mathcal{FR}} x' \approx mad \wedge y' \approx par \wedge d' \approx 1{,}5}{\Delta; \Gamma' \vdash x' \approx mad \wedge y' \approx par \wedge d' \approx 1{,}5}\ (C)}{\Delta; \Gamma' \vdash flight(x', y', d')}\ (Clause)}{\Delta; \Gamma' \vdash x' \approx mad \wedge \ldots \wedge t' \geq d' \wedge flight(x', y', d')}\ (\wedge)}{\Gamma \vdash_{\mathcal{FR}} \exists x'\ (x' \approx mad) \ldots \Gamma \vdash_{\mathcal{FR}} \exists d'\ (d' \approx 1{,}5)}}{\Delta; \Gamma \vdash \exists x' \exists y' \exists t' \exists d'\ (x' \approx mad \wedge y' \approx y \wedge t \approx t' \wedge t' \geq d' \wedge flight(x', y', d'))}\ (\exists)}{\Delta; \{t > 1{,}5, y \approx par\} \vdash travel(mad, y, t)\{t > 1{,}5\} \vdash_{\mathcal{FR}} \exists y\ (y \approx par)}\ (Clause)}{\Delta; \{t > 1{,}5\} \vdash \exists y\ travel(mad, y, t)}\ (\exists)}{\Delta; \{\top\} \vdash t > 1{,}5 \Rightarrow \exists y\ travel(mad, y, t)}\ (\Rightarrow_C)}{\Delta; \{\top\} \vdash \forall t\ (t > 1{,}5 \Rightarrow \exists y\ travel(mad, y, t))}\ (\forall)$$

□

3.3. Semántica de punto fijo

Tanto en programación lógica como en bases de datos deductivas es habitual utilizar semánticas de punto fijo para dar significado a los programas [1, 75]. Estás semánticas se basan en la iteración de un operador que extrae la información de los programas de manera secuencial y va almacenando dicha información en interpretaciones, que habitualmente están compuestas por átomos de los símbolos de predicado definidos del programa. Dado que la lógica ampliada incluye negación, será necesario definir también un modelo para dicho conectivo [11, 1, 67, 16, 79, 61].

En esta sección se definirá la semántica de punto fijo para el lenguaje $HH_{\neg}(\mathcal{C})$ incluyendo el significado de la negación de átomos. La diferencia con otras semánticas es que la información se almacena como conjuntos de pares átomo y restricción, cumpliendo que cada átomo se hace cierto si su restricción correspondiente se satisface.

Las interpretaciones son funciones que se aplican a bases de datos Δ, dando como resultado un conjunto de pares (átomo, restricción). La semántica de Δ está basada en una relación de forzado, que indica la restricción que debe satisfacerse para que una interpretación haga cierta una consulta en un contexto Δ. En la definición de dicha relación de forzado es necesario el contexto Δ, ya que la base de datos puede aumentar dinámicamente al calcular respuestas para objetivos que contengan implicación.

Estratificación y grafo de dependencias El concepto de estratificación sirve como un criterio sintáctico para determinar si una consulta a una base de datos puede ser potencialmente computada en un número finito de pasos. La idea de hay tras la estratificación es un cómputo por capas o estratos. De manera que se calcula el significado del programa comenzando por los símbolos de predicado definidos del primer estrato de manera ordenada hasta el último. Para la negación se debe cumplir que cuando $\neg A$ se va a probar, el estrato de A se ha saturado previamente, es decir, todas las respuestas para A están disponibles y $\neg A$ se puede calcular correctamente.

Sea Φ un conjunto de fórmulas de $HH_{\neg}(\mathcal{C})$. El grafo de dependecias DG_{Φ} para el conjunto Φ consta de:

- Un conjunto de nodos correspondientes a los símbolos de predicado definidos en las fórmulas del conjunto Φ.
- Un conjunto de arcos que provienen de las implicaciones de la forma $F_1 \Rightarrow F_2$. Cada implicación produce arcos (o caminos) en el grafo entre los símbolos de predicado definidos de F_1 y cada uno de los símbolos de predicado definidos de F_2.

Un arco se etiqueta negativamente cuando el átomo correspondiente aparezca negado en el antecedente de la implicación. Obsérvese que en $HH_{\neg}(\mathcal{C})$ una implicación puede aparecer no sólo entre la cabeza y el cuerpo de una cláusula, sino también en cualquier objetivo y, por tanto, en el cuerpo de una cláusula. Dado que las restricciones no incluyen símbolos de predicado definidos, en principio, no generan dependencias. En las siguientes secciones de este trabajo se verá como esta condición cambia cuando las restricciones incorporan funciones de agregación que contienen símbolos de predicado. Esto llevará a ampliar la definición de dependencia.

Definición 2 Dado un conjunto de fórmulas Φ, su grafo de dependencias correspondiente DG_Φ, y dos predicados p y q, se dice que:

- q *depende* de p si hay un camino de p a q en DG_Φ
- q *depende negativamente* de p si hay un camino de p a q en DG_Φ con al menos un arco etiquetado negativamente.

Ejemplo 5 Dado el siguiente programa en $HH_{\neg}(\mathcal{C})$:

```
q(X):- b(X).
c(X):- b(X) => q(X).
p(X):- not(b(X)).
r(X):- p(X).
```

Su grafo de dependencias aparece en la Figura 3.2, se usa el símbolo $\neg$ para denotar la dependencia negativa que aparece. Nótese que `p` depende negativamente de `b`, por el arco etiquetado negativamente que los une y `r` depende negativamente de `b` porque hay un camino desde `b` hasta `r` con un arco etiquetado negativamente. □

Definición 3 Sea Φ un conjunto de fórmulas de $HH_{\neg}(\mathcal{C})$ y sea $P = \{p_1, \ldots, p_n\}$ el conjunto de símbolos de predicados definidos de Φ.

Una *estratificación* de Φ es cualquier función $s : P \to \{1, \ldots, n\}$ tal que $s(p) \leq s(q)$ si q depende de p, y $s(p) < s(q)$ si q depende negativamente de p. El conjunto Φ es estratificable si existe una estratificación para él.

Definición 4 Sea F un objetivo o una cláusula. El *estrato de* F, denotado como $str(F)$, se define recursivamente como:
$str(p(t_1, \ldots, t_n)) = s(p)$ donde $s(p)$ es el estrato del símbolo de predicado p,
$str(F_1 \square F_2) = max(str(F_1), str(F_2))$, donde $\square \in \{\wedge, \vee, \Rightarrow\}$,
$str(QxF) = str(F)$, donde $Q \in \{\exists, \forall\}$,
$str(\neg A) = 1 + str(A)$,
$str(C) = 1$.
El *estrato de un conjunto de fórmulas* Φ es $str(\Phi) = max\{str(F) \mid F \in \Phi\}$.

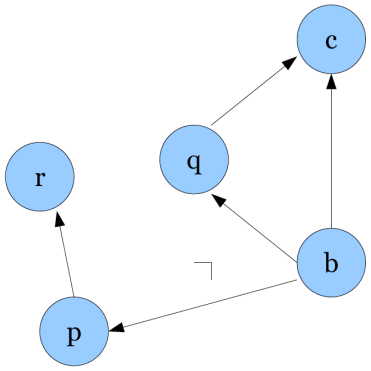

Figura 3.2: Grafo de dependencias para el programa del ejemplo 5.

Interpretaciones estraficadas y relación de forzado Sea $\mathcal{W}$ el conjunto de bases de datos estratificables Δ, con respecto a la misma estratificación fijada s, At el conjunto de átomos abiertos, $\mathcal{SL}_\mathcal{C}$ el conjunto de restricciones $\mathcal{C}$-satisfactibles módulo $\mathcal{C}$-equivalencia, $[C]$ el conjunto de todas las fórmulas $\mathcal{C}$-equivalentes a C (la clase de equivalencia de C) y $\mathcal{P}(At \times \mathcal{SL}_\mathcal{C})$ el conjunto partes del producto cartesiano de At y $\mathcal{SL}_\mathcal{C}$, es decir serán conjuntos de pares de la forma $(A, [C])$.

Las interpretaciones se clasifican en estratos de manera que cada interpretación I_i nos dará toda la información que se deduce de la base de datos desde el estrato 1 hasta su estrato i.

Definición 5 Sea $i \geq 1$, una *interpretación* I sobre el estrato i es una función:

$$I : \mathcal{W} \to \mathcal{P}(At \times \mathcal{SL}_\mathcal{C}),$$

tal que para cualquier $\Delta \in \mathcal{W}$ y cualquier $j > i$, no hay pares $(A, [C]) \in I(\Delta)$ tales que $str(A) = j$. Se denota como $\mathcal{I}_i$ el conjunto de interpretaciones sobre i.

Para simplificar notación se escribe:

- $(A, C) \in At \times \mathcal{SL}_\mathcal{C}$, en vez de $(A, [C])$, asumiendo que C es un representante de su clase de equivalencia $[C]$.
- $[I(\Delta)]_i$ para representar el conjunto $\{(A, C) \in I(\Delta) \mid str(A) = i\} \subseteq I(\Delta)$.

Nótese que si $str(\Delta) = k$, entonces $\{[I(\Delta)]_i \mid 1 \leq i \leq k\}$ es una partición de $I(\Delta)$.

Para cada $i \geq 1$, se puede definir un orden para las $\mathcal{I}_i$ como se ve a continuación:

Definición 6 Sea $i \geq 1$ y $I_1, I_2 \in \mathcal{I}_i$. Se dice que I_1 *es menor o igual que* I_2 *en el estrato* i y se denota $I_1 \sqsubseteq_i I_2$, si para cada $\Delta \in \mathcal{W}$ se satisfacen las siguientes condiciones:

- $[I_1(\Delta)]_j = [I_2(\Delta)]_j$, para cada $1 \leq j < i$.
- $[I_1(\Delta)]_i \subseteq [I_2(\Delta)]_i$.

Es sencillo comprobar que para cada $i \geq 1$, $(\mathcal{I}_i, \sqsubseteq_i)$ es un retículo.

La idea tras esta definición es que cuando se incrementa el índice i del estrato de una interpretación, la información del estrato inferior no varía. De esa forma, si $str(\neg A) = i$, dado que que el átomo A debe estar en el estrato inferior, $str(A) = i-1$, cuando se compute $\neg A$ todo lo que se sabe para el estrato $i-1$ está ya calculado y el estrato i se puede calcular correctamente. Por eso se puede garantizar la monotonía también en este caso.

Además se cumple que:

Lema 1 Para cada $i \geq 1$, toda cadena de interpretaciones de $(\mathcal{I}_i, \sqsubseteq_i)$, $\{I_n\}_{n\geq 0}$, tal que $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$, tiene una mínima cota superior $\bigsqcup_{n\geq 0} I_n$, que se define como:

$$(\bigsqcup_{n\geq 0} I_n)(\Delta) = \bigcup_{n\geq 0} \{I_n(\Delta)\},$$

para cada $\Delta \in \mathcal{W}$.

A continuación se formaliza la idea de que una interpretación I haga *cierta* una consulta G en el contexto de una base de datos Δ, si se satisface la restricción C. Se supone que la estratificación s es válida no sólo para Δ, sino también para $\Delta \cup \{G\}$.

Definición 7 Sea $i \geq 1$, la *relación de forzado* $\Vdash$ entre pares I, Δ y pares (G, C) (donde $I \in \mathcal{I}_i$, $str(G) \leq i$, y C es $\mathcal{C}$-satisfactible) se define recursivamente mediante las reglas que se muestran a continuación. Cuando se verifica $I, \Delta \Vdash (G, C)$, se dice que (G, C) es *forzado* por I, Δ.

- $I, \Delta \Vdash (C', C) \iff C \vdash_{\mathcal{C}} C'$.
- $I, \Delta \Vdash (A, C) \iff (A, C) \in I(\Delta)$.
- $I, \Delta \Vdash (\neg A, C) \iff$ para cada $(A, C') \in I(\Delta)$, es cierto que $C \vdash_{\mathcal{C}} \neg C'$. Si no existen pares de la forma (A, C') en $I(\Delta)$, entonces $C \equiv \top$.
- $I, \Delta \Vdash (G_1 \wedge G_2, C) \iff$ para cada $i \in \{1, 2\}$, $I, \Delta \Vdash (G_i, C)$.
- $I, \Delta \Vdash (G_1 \vee G_2, C) \iff$ para algún $i \in \{1, 2\}$ $I, \Delta \Vdash (G_i, C)$.
- $I, \Delta \Vdash (D \Rightarrow G, C) \iff I, \Delta \cup \{D\} \Vdash (G, C)$.
- $I, \Delta \Vdash (C' \Rightarrow G, C) \iff I, \Delta \Vdash (G, C \wedge C')$.
- $I, \Delta \Vdash (\exists x G, C) \iff$ existe C' tal que $I, \Delta \Vdash (G[y/x], C')$, donde y no aparece libre en Δ, $\exists x G$, C, y $C \vdash_{\mathcal{C}} \exists y C'$.
- $I, \Delta \Vdash (\forall x G, C) \iff I, \Delta \Vdash (G[y/x], C)$, donde y no aparece libre en Δ, $\forall x G$, C.

La noción de verdad para cada estrato viene dada por el punto fijo de un operador continuo (para cada estrato) que transforma interpretaciones y que se define a continuación.

Definición 8 Sea $i \geq 1$ un estrato. El *operador* $T_i : \mathcal{I}_i \longrightarrow \mathcal{I}_i$ transforma interpretaciones sobre i como sigue. Para $I \in \mathcal{I}_i$, $\Delta \in \mathcal{W}$ y $(A, C) \in At \times \mathcal{SL}_{\mathcal{C}}$, se tiene $(A, C) \in T_i(I)(\Delta)$ cuando:

- $(A, C) \in [I(\Delta)]_j$ para algún $j < i$, o
- $str(A) = i$ y hay una variante $\forall x_1 \ldots \forall x_n (G \Rightarrow A')$ de una cláusula en $elab(\Delta)$, tal que las variables $x_1 \ldots x_n$ no aparecen libres en A, y $I, \Delta \models (\exists x_1 \ldots \exists x_n (A \approx A' \wedge G), C)$.

Un aspecto importante del operador T_i es que para una base de datos Δ, T_i añade la información que se obtiene exclusivamente de las cláusulas Δ, cuyas cabezas son átomos del estrato i, y la información del estrato inferior permanece invariable. Nótese que si $str(A) = i$, entonces $str(\exists x_1 \ldots \exists x_n (A \approx A' \wedge G)) \leq i$. El operador T_i es monótono y continuo, es decir:

Lema 2 (Monotonía de T_i) Sea $i \geq 1$ y $I_1, I_2 \in \mathcal{I}_i$ tal que $I_1 \sqsubseteq_i I_2$. Entonces, $T_i(I_1) \sqsubseteq_i T_i(I_2)$.

Lema 3 (Continuidad de T_i) Sea $i \geq 1$ y $\{I_n\}_{n \geq 0}$ una familia numerable de interpretaciones sobre i, tal que $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$. entonces:

$$T_i(\bigsqcup_{n \geq 0} I_n) = \bigsqcup_{n \geq 0} T_i(I_n).$$

El operador T_1 tiene un mínimo punto fijo, denotado fix_1, tal que

$$fix_1 = \bigsqcup_{n \geq 0} T_1^n(I_\perp),$$

donde la interpretacion $I_\perp$ representa la función constante $\emptyset$. Procediendo de manera similar, se puede definir una cadena:

$$fix_{i-1} \sqsubseteq_i T_i(fix_{i-1}) \sqsubseteq_i T_i(T_i(fix_{i-1})) \sqsubseteq_i \ldots, \sqsubseteq_i T_i^n(fix_{i-1}), \ldots,$$

para cada estrato $i > 1$, y se puede encontrar el siguiente punto fijo de dicha cadena:

$$fix_i = \bigsqcup_{n \geq 0} T_i^n(fix_{i-1}).$$

En particular, si k es el estrato máximo en Δ, se simplifica fix_k escribiendo fix. Por tanto, $fix(\Delta)$ representa los pares (A, C) tales que A se puede deducir de Δ si C se satisface.

Ejemplo 6 Dado el dominio finito [a,b,c], considérese el programa ya elaborado:
$\Delta = \{p(a),\ p(b), \forall x(p(x) \Rightarrow t(x)), \forall x(\neg p(x) \Rightarrow q(x)),\ \forall x((p(x) \Rightarrow q(x)) \Rightarrow r(x)),$
$\forall x(\neg q(x) \Rightarrow s(x))\}$

En este programa p y t pertenecen al primer estrato, q y r al estrato 2 y s al estrato 3. A continuación se muestra el el contenido punto fijo para cada uno de los estratos de manera incremental. La relación de forzado es no determinista, por tanto, las restricciones que se muestran corresponden a representantes de sus respectivas clases de equivalencia.

- Para el primer estrato, se considera el operador T_1:
  - La primera iteración corresponde a $(T_1(\emptyset))(\Delta)$. Se considera la cláusula $p(a)$ y se tratará de probar si hay una C tal que:

    $$\emptyset, \Delta \Vdash ((x \approx a) \wedge \top), C).$$

    Una restricción válida es $C \equiv (x \approx a)$ y por tanto, el punto fijo de este primer estrato debe contener el par $(p(x), x \approx a)$. De una manera similar debe contener el par $(p(x), x \approx b)$ considerando la cláusula $p(b)$, por tanto:

    $$T_1(\emptyset)(\Delta) = \{(p(x), x \approx a), (p(x), x \approx b\}.$$

    Este operador también considera la cláusula que queda del estrato 1, pero al considerarse sobre el conjunto vacío no existen pares para esta cláusula que deba contener $T_1(\emptyset)(\Delta)$.
  - La segunda iteración corresponde a $(T_1(T_1(\emptyset)))(\Delta)$, considerando ahora la cláusula restante del primer estrato, $\forall x(p(x) \Rightarrow t(x))$, se trata de probar que:

    $$T_1(\emptyset), \Delta \Vdash (\exists x'(x \approx x' \wedge p(x')), C).$$

    Para eliminar el cuantificador existencial, de acuerdo con la definición, se sustituye x' por una nueva variable y y se obtiene:

    $$T_1(\emptyset), \Delta \Vdash (x \approx y \wedge p(y)), C').$$

    De forma que $C \vdash_{\mathcal{C}} \exists y C'$. Para la conjunción, siguiendo de nuevo la definición de la relación de forzado, se debe verificar $T_1(\emptyset), \Delta \Vdash (x \approx y, C')$ y $T_1(\emptyset), \Delta \Vdash (p(y), C')$. Por ejemplo, la restricción $C \equiv (x \approx a) \vee (x \approx b)$ satisface las condiciones necesarias. Por lo tanto $T_1(\emptyset)(\Delta)$ contiene el par $(t(x), x \approx a \vee x \approx b)$ que completa el punto fijo del primer estrato[1]. En adelante, los pasos referentes a la eliminación de $\exists$ y $\approx$ no se mostrarán para simplificar.
- Para el segundo estrato se considera el operador T_2, que opera sobre fix_1 y sobre Δ, por tanto, su primera y única iteración corresponde al cálculo de $(T_2(fix_1))(\Delta)$:

[1] En esta iteración se vuelven a considerar $p(a)$ y $p(b)$ pero no se pueden añadir nuevos pares a los ya existentes.

- Haciendo uso de la cláusula $\forall x(\neg p(x) \Rightarrow q(x))$, se tendrá que verificar:

$$fix_1, \Delta \Vdash (\exists x'(x \approx x' \wedge \neg p(x')), C).$$

Lo que lleva a que C debe verificar $C \vdash_{\mathcal{C}} \neg C'$ para todas las C' tales que $(p(x), C') \in fix_1(\Delta)$. Para ello se usan los pares pertenecientes a $fix_1(\Delta)$ y se obtienen $(x \approx a)$ y $(x \approx b)$. Por lo que $T_2(fix_1(\Delta))$ debe contener el par $(q(x), x \neq a \wedge x \neq b)$.

- Considerando la cláusula $\forall x((p(x) \Rightarrow q(x)) \Rightarrow r(x))$. Se tendrá que comprobar si:

$$fix_1, \Delta \Vdash (\exists x'(x \approx x' \wedge p(x') \Rightarrow q(x')), C).$$

Lo cual lleva a aumentar la Δ con $p(x')$ y tratar de probar si:

$$fix_1, \Delta \cup \{p(x')\} \Vdash (q(x'), C').$$

No es posible encontrar, en este caso, una C' que satisfaga esta relación y por tanto $T_2(fix_1) = T_2^j(fix_1)$ para $j \geq 1$, por lo que será un punto fijo para el estrato 2.

- Para el tercer estrato se procede como en estratos anteriores aplicando T_3 sobre fix_2 y sobre Δ. Se considera la cláusula con el predicado en el tercer estrato $\forall x(\neg q(x) \Rightarrow s(x))$. Procediendo de manera similar el par $(s(x), x \approx a \vee x \approx b)$ debe aparecer en $(T_3(fix_2))(\Delta)$ junto con los pares correspondientes a $fix_2(\Delta)$.

Por tanto, el punto fijo queda: $\{(p(x), x \approx a)$, $(p(x), x \approx b)$,$(t(x), x \approx a \vee x \approx b)$,$(q(x), x \neq a \wedge x \neq b)$,$(s(x), x \approx a \vee x \approx b)\}$. □

Corrección y completitud En [15] se demuestra que la semántica de punto fijo para $HH(\mathcal{C})$ es correcta y completa para el cálculo $\mathcal{UC}$. Al introducir la negación es necesario probar la corrección y completitud de $HH_{\neg}(\mathcal{C})$ con respecto al cálculo extendido también con la negación. Decir que el esquema $HH_{\neg}(\mathcal{C})$ es correcto y completo con respecto al cálculo $\mathcal{UC}_{\neg}$ significa decir que la relación de forzado, considerando el punto fijo del último estrato de la base de datos, junto con una consulta, coincide con la derivación en el cálculo $\mathcal{UC}_{\neg}$ para dicha consulta. Más concretamente, si $str(G) = i$ y el punto fijo del estrato i, fix_i fuerza (G, C) en el contexto de Δ, se debe cumplir que C es una restricción respuesta de G para Δ. Esta demostración aparece con todo detalle en [50], en este trabajo se tratará solamente de dar algunas intuiciones.

Para el caso en que las consultas no tengan negación, la demostración de la corrección y la completitud es similar a la que aparece en [15] para $HH(\mathcal{C})$. Las siguientes proposiciones corresponden a este caso:

Proposición 1 Para todo $\Delta \in \mathcal{W}$, para todo par $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, tal que $str(G) = 1$ entonces:

$$fix_1, \Delta \Vdash (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_{\neg}} G.$$

Proposición 2 Para todo $i \geq 1$, $\Delta \in \mathcal{W}$, y para todo par $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, tal que G que no contiene negación si $str(G) \leq i$, entonces:

$$fix_i, \Delta \models (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G.$$

Teorema 1 (Corrección y Completitud) Para todo $i \geq 1$, $\Delta \in \mathcal{W}$, y para todo par $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, si $str(G) \leq i$ entonces:

$$fix_i, \Delta \models (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G.$$

La demostración de este teorema se hace por inducción sobre i. La proposición 1 es el caso $i = 1$ y demostrando la proposición 2 se demuestra muchos de los casos de $i > 1$. Se centrará la explicación en el caso que incluye negación:

- La proposición 1 captura el caso en que $i = 1$.
- Para el caso $i > 1$, se asume la siguiente hipótesis de inducción: para todo Δ, G, C, con $str(G) \leq i - 1$ se cumple $fix_{i-1}, \Delta \models (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G$.
  - La proposición 2 se corresponde con todos los casos de las posibles formas de G menos $\neg A$.
  - Para el caso $\neg A$, se parte de que si $fix_i, \Delta \models (\neg A, C) \iff$ para todo C' tal que $(A, C') \in fix_i(\Delta)$, se debe demostrar $C \vdash_{\mathcal{C}} \neg C'$, o bien que no existe dicha C' y $C \equiv \top$. Obviamente, $str(A) \leq i - 1$, es equivalente a decir que para toda restricción C' tal que se cumpla $fix_{i-i}, \Delta \models (A, C')$, se debe demostrar $C \vdash_{\mathcal{C}} \neg C'$, o bien no existe dicha C' y $C \equiv \top$. Aplicar la hipótesis de inducción es equivalente a decir que: o bien por cada C' tal que $\Delta; C' \vdash_{\mathcal{UC}_\neg} A$, se demuestra $C \vdash_{\mathcal{C}} \neg C'$, o bien no hay tal C' y $C \equiv \top$. Esto equivale a la definición de la regla del cálculo $\Delta; C \vdash_{\mathcal{UC}_\neg} \neg A$, como se puede ver en la definición 3.1:

$$\frac{\Gamma \vdash_{\mathcal{C}} \neg C \quad \text{para todo } \Delta; C \vdash A}{\Delta; \Gamma \vdash \neg A} \ (\neg)$$

Todos los análisis de casos mencionados aparecen en [50]. Como consecuencia del teorema se extrae que:

$$(A, C) \in fix(\Delta) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} A.$$

Lo que significa que los átomos del punto fijo de la base de datos son aquellos que se derivan del cálculo.

La ventaja de la semántica de punto fijo es que se puede usar como base para una implementación del esquema $HH_\neg(\mathcal{C})$. Para estos formalismos se usa un sistema de restricciones genérico $\mathcal{C}$. Este sistema se puede ver como una caja negra que comprueba la $\mathcal{C}$-satisfactibilidad de una restricción de entrada C. A la hora de implementar, se ha buscado que el prototipo sea muy cercano a los fundamentos teóricos, concretamente a la semántica de punto fijo. Como se verá en los siguientes capítulos, la implementación que se ha desarrollado del esquema $HH_\neg(\mathcal{C})$ es independiente del sistema de restricciones concreto.

Capítulo 4

Un sistema basado en $HH_{\neg}(\mathcal{C})$

Una vez que se han establecido los fundamentos teóricos de $HH_{\neg}(\mathcal{C})$, en este capítulo se trata de dar una visión general del sistema implementado. Se describe como se relacionan los componentes que forman parte del sistema, así como las acciones que tienen lugar desde que se carga una base de datos hasta que se procesa una consulta. Se puede ver este capítulo por un lado como una versión concisa de los contenidos de los capítulos restantes del trabajo. Cada una de sus secciones será explicada con detenimiento en capítulos posteriores. Por otro lado también se puede ver este capítulo como una guía que orienta al usuario del sistema. Se dan detalles de cómo se especifican las declaraciones de dominios y las declaraciones de tipos de los predicados. También se describe cómo se carga una base de datos en el sistema y cómo formular consultas a la base de datos. Finalmente se describe el funcionamiento general de las funciones de agregación, la influencia de estas en el grafo de dependencias, así como ejemplos concretos.

El sistema está disponible en la dirección `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems`.

4.1. Funcionamiento general del sistema

Cuando se inicia el sistema, se muestra el inductor de comandos `HHn(C)>`. Para cargar un archivo con una base de datos `Delta`, el usuario debe usar el comando `run` de la siguiente manera:

`HHn(C)> run(`*`filename`*`).`

donde *`filename`* es el archivo que contiene las cláusulas que definen `Delta`. La ejecución de este comando produce el cómputo de la semántica de punto fijo de `Delta` siguiendo las siguientes etapas:

1. Se comprueban e infieren los tipos de los predicados (véase la sección 4.2).
2. Se calcula el grafo de dependencias para la base de datos `Delta` (véase la sección 5.5).

3. Se calcula, si existe, una estratificación s para `Delta`. Si no existe, el sistema lanza un mensaje de error y finaliza su ejecución (véase de nuevo la sección 5.5).

4. Si el paso anterior ha finalizado correctamente, se calcula $fix(\texttt{Delta})$ (véase la sección 5.3).

Tras estos pasos el sistema almacena en memoria información sobre:

- El punto fijo $fix(\texttt{Delta})$.
- La estratificación s.
- El grafo de dependencias de `Delta`.

Esta información se mantiene mientras no se procese una nueva base de datos o alguna consulta cambie el grafo de dependencias y la estratificación. Cuando el usuario introduce una consulta `G` en el inductor del sistema, en primer lugar, se calcula, si existe, una nueva estratificación s' para el conjunto $\texttt{Delta} \cup \{\texttt{G}\}$. Igual que sucedía en los pasos previos, si no existe s' el sistema finaliza la ejecución de `G`, en otro caso se distingue:

- Si $s = s'$, el punto fijo almacenado es válido para evaluar la consulta `G`. La restricción respuesta `C` se obtiene ejecutando el predicado que implementa:

  $$fix(\texttt{Delta}); \texttt{Delta} \models (\texttt{Q}, \texttt{C})$$

  como se puede ver en la sección 5.3.

- Si $s \neq s'$, los símbolos de predicados implicados en el cálculo de `G` pueden estar en distinto estrato que cuando se calculaba $fix(\texttt{Delta})$. Por tanto, el punto fijo almacenado no es válido para el cálculo de la consulta `G` y se debe recalcular. En este caso, dicha consulta `G` se traduce a una cláusula temporal `A :- G`, donde `A` es un átomo cuyo símbolo de predicado es `query` y cuyos argumentos son las variables libres en `G` (estas están cuantificadas existencialmente de manera implícita en `G` y universalmente en `A :- G`). Además el tipo para `query` se infiere y añade como declaración de tipo `type(query(`*argument_types*`))`. Después, el sistema utiliza

  $$\texttt{Delta}' = \texttt{Delta} \cup \{\texttt{A} : -\texttt{G}\}$$

  para calcular $fix(\texttt{Delta}')$. La respuesta de `G` es la restricción `C` tal que `(A,C)`$\in fix(\texttt{Delta}')$.

Además del comando `run` que se ha explicado, el sistema dispone de los siguientes comandos:

- El comando `listing` muestra por pantalla las cláusulas correspondientes a la última base de datos que se ha procesado, las declaraciones de dominio y anotaciones de tipo.
- El comando `fix` muestra el punto fijo de la última base de datos que se ha procesado.

- El comando `spy(`*`predicate`*`)` establece un punto espía en el predicado *predicate* para depuración.
- El comando `nospyall` elimina todos los puntos espía que se hayan fijado.
- El comando `ppdebug` muestra anotaciones para predicados en la depuración.
- El comando `noppdebug` elimina las anotaciones en los predicados.
- El comando `prolog(`*`goal`*`)` llama a un objetivo del sistema Prolog subyacente.
- El comando `cmd(`*`comand`*`)` llama a un comando del sistema operativo.
- El comando `exit` termina la ejecución del sistema.

El menú con todos los comandos disponibles aparece mediante el comando `help`. Cualquier otra entrada que se introduzca en el inductor, distinta a los comandos mencionados, se interpreta como una consulta a la base de datos.

4.2. Tipos

Al igual que ocurre con las tablas en las bases de datos relacionales, los predicados de las bases de datos del sistema están tipados. Cuando se va a definir un predicado el usuario debe declarar su tipo. Una declaración de tipo para un predicado se escribe:

`type(`*`predicate`*`(`*`type_1`*`, ..., `*`type_n`*`)).`

Se han implementado sistemas de comprobación y de inferencia de tipos para los programas $HH_{\neg}(\mathcal{C})$ que detectan inconsistencias de tipo y ausencia de declaración de tipos respectivamente. Además, este sistema es capaz de inferir el tipo de las consultas que formula el usuario. Los tipos se anotan para cada símbolo de predicado.

Las variables que forman parte de los predicados también tienen un tipo anotado. Para esta anotación de tipo se hace uso de las variables atribuidas de Prolog [25], almacenando en un atributo el tipo de la variable. Con respecto al sistema de tipos se pueden dar las siguientes excepciones con sus correspondientes mensajes de error:

- Excepción de conflicto de tipo (`type-conflict exception`) cuando se trata de asignar tipos distintos a una misma variable.
- Excepción de ausencia de declaración de tipo (`lack-of-type-declaration exception`) cuando no se le asigna tipo a una variable.

4.3. Dominios

Los tipos que incorpora el sistema tienen asociados unos dominios predefinidos:

- El dominio asociado al tipo `bool` está compuesto por los elementos `true` y `false`.

- El dominio asociado al tipo `integer` está compuesto por números enteros y necesita que se fije, previamente a su uso, un límite superior y un límite inferior.
- El dominio asociado al tipo `real` está compuesto por números reales y es un tipo de datos infinito.

Además, el usuario puede definir nuevos dominios de datos enumerados. Una declaración de dominio enumerado se escribe:

`domain(`*`data_type`*`, [`*`constant_1`*`, ..., `*`constant_n`*`]).`

Dónde *`data_type`* es el nombre del dominio y *`constant_1`*`, ..., `*`constant_n`* son sus elementos.

Ejemplo 7 Algunas declaraciones de dominio válidas son:

```
domain(name,[angela,david,nicolas]).
domain(city,[ber,lon,par]).
```

Donde los valores `ber`, `lon` y `par` representan a las ciudades Berlín, Londres y París respectivamente. □

Además, se permiten intervalos en la declaraciones de tipos de datos: tanto para especificar los límites del dominio predefinido `integer`:

```
domain(integer, -100..200).
```

Como para definir un dominio nuevo:

```
domain(months, 1..12).
```

Una restricción `C` se escribe `constr(`*`dom`*`,C)`, donde *`dom`* es el nombre del dominio asociado a la restricción, como por ejemplo: `constr(real, X>2.5)` o `constr(city, Y=mad)`. Mediante la inferencia se puede conocer el dominio de una expresión que aparece dentro de una restricción. El dominio en el contexto de una restricción se usa para asignar dicha restricción a su resolutor correspondiente. Por ejemplo, las restricciones anotadas con los tipos `city` o `integer` se enviarán al resolutor de dominio finito y aquellas anotadas con el dominio `real` se envían al resolutor de restricciones reales. La implementación de los sistemas de restricciones se describe en el capítulo 6. A pesar de que las restricciones estén explícitamente anotadas con el tipo, dentro de los distintos resolutores se generan nuevas restricciones y estas se manejan sin anotación de tipo. El motivo de esta anotación explícita era permitir la total combinación de dominios dentro de los distintos resolutores del sistema. De momento solo se ha conseguido una combinación de dominios limitada, pero la estructura del interfaz `solve` está orientada al manejo de restricciones con total combinación de dominios que se abordará como trabajo futuro.

Combinación de dominios En el sistema implementado se pueden usar distintos dominios para una misma base de datos pero, de momento, estos dominios no se pueden combinar totalmente. Como se ha introducido, de manera general los resolutores no pueden resolver restricciones asociadas a más de un dominio diferente. Desde el punto de vista del usuario, se pueden definir predicados con argumentos de distintos tipos de datos *sólo* para la parte extensional de la base de datos, como se puede ver en el ejemplo a continuación.

Ejemplo 8 Para el ejemplo de gestión de vuelos que aparece la sección 3.2:

```
flight(mad, par, 1.5).
flight(par, ny, 10).
flight(lon, ny, 9).
travel(X,Y,T) :- flight(X,Y,D), constr(real,T >= D).
travel(X,Y,T) :- flight(X,Z,T1), travel(Z,Y,T2),
                 constr(real,T >= T1+T2).
```

Es necesario haber declararado el dominio:

```
domain(city,[lon,par,mad]).
```

Se puede definir de manera extensional un predicado que asocie cada valor del dominio `city` con un valor del dominio predefinido `real`, con el fin de que el predicado `travel` tenga variables de tipo solo `real`.

```
type(city_id(city,real)).
city_id(lon,1).
city_id(par,2).
city_id(mad,3).
city_id(ny,4).
```

De esta forma se evita la combinación de dominios para el predicado `travel` que de otra manera tendría tipos `city` y `real`. La declaración de tipo de los predicados es:

```
type(flight(real,real,real)).
type(travel(real,real,real)).
```

Finalmente la parte extensional de la base de datos quedará definida como:

```
flight(3, 2, 1.5).
flight(2, 4, 10).
flight(1, 4, 9).
```

Lo cual no modifica el significado original del programa. Se debe tener en cuenta también que es bastante habitual identificar con valores numéricos datos de este tipo en el contexto de las bases de datos. Para evitar la combinación de dominios no es necesario, en este caso, modificar la parte intensional de la base de datos (el predicado `travel`). □

4.4. Funciones de agregación

El problema de la resolución de agregados se ha abordado tanto en el campo de las bases de datos con restricciones (véase el capítulo 6 de [35]) como en el de las bases de datos deductivas [83].

Las funciones de agregación habitualmente se usan para calcular valores numéricos a partir de un conjunto de valores no necesariamente numérico. En la implementación realizada, la resolución de agregados se delega al resolutor de restricciones.

A la hora de incluir funciones de agregación es necesario tener en cuenta las siguientes consideraciones:

- Las funciones de agregación reciben un conjunto de valores como argumento que satisfacen una relación y devuelven un valor. Al igual que sucede con la negación, para calcular el valor de una función de agregación sobre una relación es necesario conocer todos los valores que satisfacen dicha relación.
- La respuesta a estas consultas en este tipo de lenguajes suelen ser restricciones que representan respuestas intensionales, en lugar de valores explícitos.
- Además, dado que las restricciones pueden representar valores infinitos, la función de recuento (`count`) sobre estos valores puede no tener sentido.

Por otra parte, el hecho de que el sistema esté preparado para soportar la negación hace que se trate también de un buen marco para incorporar funciones de agregación. Las funciones de agregación se pueden representar como restricciones y la resolución de estas se delega en su correspondiente resolutor (en función de los valores sobre los que se calcule el agregado). De esta forma se utiliza la ventaja de la eficiencia de los resolutores.

A la hora del cómputo de funciones de agregación, al igual que ocurre con el cálculo de la negación de un átomo se impone la condición de que el significado de un átomo A, sobre el que se calcula la función, debe estar ya calculado.

El significado de los predicados se va almacenando por estratos en interpretaciones que se componen de pares átomo y restricción.

Con respecto al cómputo de las funciones de agregación se impone también la condición de que para cada par de la forma (A, C), C instancia al átomo A, haciéndolo cerrado. En otro caso, el sistema no puede calcular dicha función.

Un ejemplo de par en que C instancia a A haciéndolo cerrado es (`(p(X),X=5)`) dado que equivale a `p(5)`. Sin embargo, si en la interpretación aparece un par no cerrado no se puede calcular una función de agregación sobre el predicado de ese par y el sistema lanza una excepción (`Unsupported-constraint exception`). Como por ejemplo sobre el predicado `p` del par`(p(X),X>5)` no se puede calcular una función de agregación. Esta limitación se impone dado que, a diferencia de otros sistemas de bases de datos, en este lenguaje se trata con datos infinitos debido al uso de restricciones.

El sistema calcula las siguientes funciones de agregación:

- `count(`*Atom*`)`. *Número de elementos*: Devuelve el número de instancias cerradas de *Atom* en la interpretación. Esta función se puede aplicar a cualquier dominio.

- `sum`(*Atom*, *Var*). *Sumatorio*: Devuelve el sumatorio de las instancias cerradas de *Var* para todas las apariciones de *Atom* en la interpretación, dónde *Atom* incluye *Var* como argumento. Esta función se aplica a dominios numéricos.
- `avg`(*Atom*, *Var*). *Media*: Devuelve la media aritmética de las instancias cerradas de *Var* para todas las apariciones de *Atom* en la interpretación, donde *Atom* incluye *Var* como argumento. Esta función se aplica a dominios numéricos.
- `min`(*Atom*, *Var*). *Mínimo*: Devuelve el mínimo de las instancias cerradas de *Var* para todas las apariciones de *Atom* en la interpretación, donde *Atom* incluye *Var* como argumento. Esta función se puede aplicar a todos los dominios que tienen definido un orden entre sus elementos.
- `max`(*Atom*, *Var*). *Máximo*: Análogo a `min`.

Ejemplo 9 Se considera una base de datos extensional para un banco.

La siguiente relación da información sobre el identificador, el saldo y el salario de un cliente del banco:

```
% client(Ident, Balance, Salary)
client(1,2000,1200).
client(2,1000,1500).
client(3,5300,3000).
```

La relación `client_id` asigna un nombre de cliente con un identificador.

```
% client_id(Name,Ident)
client_id(smith,1).
client_id(brown,2).
client_id(mcandrew,3).
```

A continuación se pueden definir vistas con funciones de agregación, por ejemplo:

```
liquid(Amount) :- constr(real, Amount = sum(client(N,B,S),B)).
```

permite calcular el activo líquido como el sumatorio de los saldos de todos los clientes, mediante el uso de la función de agregado `sum`. Se puede especificar el salario medio mediante la vista:

```
avg_salary(Average) :- constr(real, Average =
          sum(client(N1,B1,S1),B1) / count(client(N2,B2,S2))).
```

o directametne con la función `avg`:

```
avg_salary(Average) :-
        constr(real, Average = avg(client(N,B,S),S))).
```

Se puede usar el agregado `min` para encontrar el nombre del primer cliente en el orden lexicográfico:

```
first_client_name(Name) :-
    constr(real, Name = min(client_id(N,Id),N)).
```

Este ejemplo se utiliza en el capítulo 7 para explicar cómo se calcula el punto fijo de una BD. □

Dado que las restricciones contienen funciones de agregación y estas incluyen predicados, se debe tener en cuenta algunas consideraciones adicionales. Por un lado estas restricciones incluirán nuevas dependencias en el grafo de dependencias. De manera similar a lo que ocurre con los átomos negados, si una cláusula que define un predicado p contiene un agregado sobre otro predicado q, el cálculo del significado de q debe haber finalizado antes de que comience el cálculo del significado de p. Por ejemplo, cuando se procese la cláusula `p(X):- constr(real, X=count(q(Y))`, se debe saber toda la información sobre `q` antes de calcular la información sobre `p`. Esto se garantiza si el estrato de uno es menor que el de otro, es decir $s(q) < s(p)$. Además, el resolutor debe conocer el significado de q para calcular la función de agregación. Para el ejemplo 9, que se acaba de mostrar, se debe cumplir que:

- $s($`client`$) < s($`liquid`$)$,
- $s($`client`$) < s($`avg_salary`$)$,
- $s($`client_id`$) < s($`first_client_name`$)$.

Para resolver una restricción que contiene una función de agregación es necesario, como se ha visto, conocer el valor de la interpretación actual. Este hecho motiva que la interpretación sea un parámetro de la interfaz `solve` como se ve en la sección 6.2. La resolución de funciones de agregación se explica en la sección 6.3. En los siguientes capítulos se describen los diferentes componentes de la implementación, empezando por el módulo de la semántica de punto fijo.

Capítulo 5

Implementación de la semántica de punto fijo

En este capítulo se presenta el núcleo de la implementación: el cálculo del punto fijo. Se muestra una implementación concreta de un sistema basado en el esquema $HH_{\neg}(\mathcal{C})$ siguiendo la semántica de punto fijo definida en el la sección 3.3. Se ha desarrollado una implementación en Prolog independiente del sistema de restricciones concreto, lo que hace que sirva de base para cualquier instancia. Se explican los predicados que sirven para calcular el punto fijo que da significado a una base de datos. En particular, se muestra con detenimiento la implementación de la relación semántica de forzado y se destaca cómo se han resuelto las dificultades inherentes a este sistema, al permitir consultas hipotéticas provoca que la base de datos crezca dinámicamente.

5.1. Punto fijo por estratos

Para el cálculo del punto fijo de una base de datos estratificable `Delta`, se supone que se ha calculado previamente una estratificación y se ha almacenado en una lista `Stratification` que contiene pares de la forma

```
(defined_predicate_symbol,stratum).
```

La implementación del algoritmo de estratificación se explica en la sección 5.5.

Una cláusula de la forma `A :- G` representa a la fórmula $\forall X_1, \ldots, \forall X_n(\texttt{G} => \texttt{A})$, donde $X_1, \ldots, X_n$ son las variables libres de $(\texttt{A}, \texttt{G})$, y se representa como un término Prolog:

```
hhcnClause(Vars,A,G),
```

donde `Vars`$= [X_1, \ldots, X_n]$.

El punto fijo se calcula estrato por estrato. Es posible que un estrato requiera calcular el punto fijo de otro estrato previo si la base de datos se ha ampliado localmente debido a implicaciones anidadas, como se explicará en la sección 5.4.

El predicado

```
fixPointStrat(+Delta, +Stratification, +St, -Fix)
```

calcula `Fix`$= fix_{\texttt{St}}$(`Delta`), usando `Stratification`. Si `Delta` representa una base de datos tal que str(`Delta`) $= k$, este predicado devuelve fix_k(`Delta`) mediante el cómputo de los puntos fijos previos desde `St` $= 0$ hasta `St` $= k$.

```
fixPointStrat(_Delta,_Stratification,0,[]):- !.

fixPointStrat(Delta,Stratification,St,FixSt):-
  St1 is St-1,
  fixPointStrat(Delta,Stratification,St1,FixSt1),
  iterT(Delta,Stratification,St,FixSt1,FixSt).
```

Cada punto fijo se evalúa mediante la iteración del operador de punto fijo como se expone a continuación, la información se va almacenando de manera parcial en la interpretación:

```
iterT(Delta,Stratification,St,I,FixSt) :-
  opT(Delta,Delta,Stratification,St,I,TI),
  (
    I==TI,!, FixSt=I
  ;
    iterT(Delta,Stratification,St,TI,FixSt)
  ).
```

El parámetro `I` representa la interpretación calculada hasta el momento y `FixSt` será el punto fijo resultante para el estrato `St`. Este operador iterará hasta que no haya más información que se pueda añadir a la interpretación (esto se comprueba con la condición `I==TI`). Esto es, se alcanza el punto fijo para el estrato `St`. Se explicará el predicado `opT` en la siguiente sección.

5.2. Operador de punto fijo

El predicado `opT` corresponde a la aplicación del operador T_i a una interpretación del estrato i. Siguiendo la definición del operador de punto fijo (definición 8 de la página 30), el predicado

```
opT(+Clauses,+Delta,+Stratification,+St,+I,-TI)
```

recibe en la interpretación `I` el conjunto de pares $T_i^n(fix_{i-1})$(`Delta`) para algún $n \geq 0$, y en `St` el estrato i, y calcula `TI` $= T_i^{n+1}(fix_{i-1})$(`Delta`). Como hemos visto en la sección anterior la llamada a `opT` de `iterT` tiene la forma:

```
opT(Delta,Delta,Stratification,St,I,TI)
```

```
opT([],_Delta,_Stratification,_St,I,I).

opT([clause(Vars,A,G)|Rs],Delta,Stratification,St,I,TI):-
  functor(A,P,_),
  member((P,St),Stratification),
  !,
  rename(Vars,(A,G),Vars1,(A1,G1)),
  flatHead(A1,A2,Cs),
  buildExists(Vars1,(Cs,G1),G2),
  (
   force(Delta,Stratification,I,G2,C),
   !,
   addItemLst([(A2,C,[])],[],I,I1)
   ;
   I1=I
  ),
  opT(Rs,Delta,Stratification,St,I1,TI).

opT([_|Rs],Delta,Stratification,St,I,I1):-
  opT(Rs,Delta,Stratification,St,I,I1).
```

El predicado utiliza `Delta` dos veces porque por un lado cada cláusula que se usa para añadir información a la interpretación, se va eliminando de la primera lista que contiene a `Delta`, pero por otro lado, la relación de forzado necesita mantener la base de datos `Delta` al completo. Este operador utiliza las cláusulas del estrato actual `St` (segunda cláusula) e ignora el resto (última cláusula).

La segunda cláusula de `opT`, en primer lugar comprueba que la cláusula de la base de datos actual corresponde al predicado del estrato que se está procesando. Tras esto, tienen lugar algunas transformaciones de la cláusula `hhcnClause(Vars,A,G)`. Los predicados `rename`, `flatHead` y `buildExists` construyen, siguiendo la definición 8 de la página 30, el objetivo:

$$\texttt{G2} = \exists\ \texttt{Vars1}\ (\texttt{G1} \wedge \texttt{A1} \approx \texttt{A2}),$$

siendo $\forall$ `Vars1` (`G1` $\Rightarrow$ `A1`) una variante de `hhcnClause(Vars,A,G)`, donde `Vars1` son las variables `Vars` renombradas. Después intenta forzar el objetivo obtenido `G2` usando `Delta` y la interpretación `I`. El predicado `force` se explica en la siguiente sección. Si este paso tiene éxito, se obtiene la restricción asociada `C` y se añade el par `(A2,C)` a la interpretación, si no tiene éxito no se añaden nuevos pares para esta cláusula. Finalmente, `opT` realiza la misma operación sobre el resto de las cláusulas `Rs`.

5.3. Relación de forzado

Se ha implementado la relación $\Vdash$ relativa a la definición 7 de la página 29 por medio del predicado:

```
force(+Delta,+Stratification,+I,+G,-C)
```

Dado $\texttt{I} = T_i^n(fix_{i-1})(\texttt{Delta})$ para algún $n \geq 0$ y un estrato fijado $i > 0$, el predicado `force` tiene éxito si se cumple:

$$T_i^n(fix_{i-1}), \texttt{Delta} \Vdash (\texttt{G}, \texttt{C}).$$

Un punto importante que se debe tener en cuenta a la hora de entender la implementación es que el predicado `force` es de naturaleza determinista. La definición de $\Vdash$ establece condiciones sobre la restricción `C` que hagan que se satisfaga $\texttt{I}, \texttt{Delta} \Vdash (\texttt{G},\texttt{C})$. Sin embargo, el predicado `force` debe construir una restricción concreta `C`.

Además, cada posible respuesta para una consulta debe ser *capturada* en una restricción respuesta lo más simple posible, para esto se utiliza una disyunción. De la simplificación y construcción de la disyunción de las restricciones se encarga el resolutor cuya implementación se verá en el siguiente capítulo. La conexión con el resolutor se hace a través del predicado `solve`. La llamada es de la forma `solve(I,C,SC)`, donde `C` es la restricción de entrada, `SC` es la restricción resuelta de salida e `I` es la interpretación que contiene información que se usará para el cómputo de funciones de agregación.

En la figura 5.1 se muestra la implementación concreta de `force`. Hay una cláusula `force` para cada objetivo (consulta) de la sintaxis. Se explicará brevemente salvo el caso de la implicación `D =>G` (cláusula `(5)`), a la que se le dedica la próxima sección.

- La cláusula `(1)` representa el forzado de una restricción `C` sobre un dominio `Dom`, que se procesa mediante una llamada al resolutor.
- La cláusula `(2)` representa la conjunción `G1,G2`. En este caso se fuerzan ambos objetivos, después el resolutor se encarga de resolver la conjunción de ambas restricciones respuesta.
- Para la disyunción `G1;G2` en la cláusula `(3)`, se pueden dar cuatro situaciones: que se puedan forzar ambos objetivos, solo `G1`, solo `G2` o que no se pueda forzar ninguno de los dos. La restricción respuesta se obtiene resolviendo la restricción correspondiente a cada caso. Si no se puede resolver falla.
- La cláusula `(4)` corresponde a una implicación con una restricción en el antecedente. En este caso, en primer lugar se intenta resolver el antecedente `C2` (para comprobar satisfactibilidad). Si no es satisfactible (segunda parte de la disyunción) la respuesta es trivialmente `true`. En caso de que sea satisfactible, se debe forzar el predicado `G`, con lo que se obtiene la restricción respuesta `C`. En este caso se utilizará el predicado `constr_conj` para encontrar una `C1` tal que $\texttt{C1},\texttt{C2} \vdash_{\mathcal{C}} \texttt{C}$ como se puede ver en la definición 7.
- Para el cuantificador existencial (cláusula `(6)`) y de acuerdo con la definición de $\Vdash$, para encontrar `C` tal que

$$\texttt{I}, \texttt{Delta} \Vdash (\texttt{ex}(\texttt{X}, \texttt{G}), \texttt{C})$$

```
(1) force(_Delta,_Stratification,I,constr(Dom,C),C1):- !,
      solve(Dom,I,C,C1).

(2) force(Delta,Stratification,I,(G1,G2),C):- !,
      force(Delta,Stratification,I,G1,C1),
      force(Delta,Stratification,I,G2,C2),
      solve(I,(C1,C2),C).

(3) force(Delta,Stratification,I,(G1;G2),C):- !,
      ( force(Delta,Stratification,I,G1,C1), !,
        ( force(Delta,Stratification,I,G2,C2), !, solve(I,(C1;C2),C)
        ; solve(I,C1,C) )
      ; force(Delta,Stratification,I,G2,C2), solve(I,C2,C) ).

(4) force(Delta,Stratification,I,(constr(D,C2)=>G),C1):- !,
      ( solve(D,I,C2,_), !, force(Delta,Stratification,I,G,C),
        constr_conj(D,I,C1,C2,C)
      ; C1=true ).

(5) force(Delta,Stratification,_I,(D=>G),C):- !, ...

(6) force(Delta,Stratification,I,ex(X,G),C):- !, replace(X,X1,G,G1),
      force(Delta,Stratification,I,G1,C1), solve(I,ex(X1,C1),C).

(7) force(Delta,Stratification,I,fa(X,G),C):- !, replace(X,X1,G,G1),
      force(Delta,Stratification,I,G1,C1), solve(I,fa(X1,C1),C).

(8) force(_Delta,_Stratification,I,not(At),C):- !, lookUpAll(At,I,Ls),
      ( Ls==[], !, C=true ; buildNegConj(Ls,NLs), solve(I,NLs,C) ).

(9) force(_Delta,_Stratification,I,At,C):-
      lookUpAll(At,I,Cs), buildDisj(Cs,C1), solve(I,C1,C).
```

Figura 5.1: Relación de forzado

se obtiene `G1` como resultado del reemplazo de `X` por una nueva variable `X1` en `G`. Tras esto se prueba

$$\texttt{I}, \texttt{Delta} \models (\texttt{G1}, \texttt{C1})$$

Finalmente se obtiene `C` resolviendo `ex(X1,C1)`.

- El tratamiento del cuantificador universal (cláusula `(7)`) es análogo al existencial, salvo que finalmente se debe resolver `fa(X1,C1)`.
- Para el átomo negado `not(At)` (la cláusula `(8)`), debido a la estratificación, se puede asegurar que cualquier átomo `At` que se pueda deducir de la base de datos está ya presente en la interpretación actual `I`. Por medio de `lookUpAll (At,I,Ls)`, se encuentra la lista `Ls=[C1,...,Cn]` tal que`(At,Ci)`$\in$`I`, donde `i` toma valores entre `1` y `n`.

  A partir de la lista `Ls` se construye la restricción $\neg$`C1`$\wedge \ldots \wedge \neg$`Cn` (o `true` en caso de `Ls=[]`) y se almacena en la variable `NLs`, tras esto se resuelve para obtener la restricción `C` que se buscaba. Esto se delega, de nuevo, en el resolutor y se verá en la sección 6.2.
- Finalmente la cláusula `(9)` (caso por defecto) es el forzado de un átomo `At`. Como antes, se hace una búsqueda de todos los pares $(\texttt{At}, \texttt{C1}), \ldots, (\texttt{At}, \texttt{Cn}) \in \texttt{I}$, se construye la disyunción `C1=C1`$\vee \ldots \vee$`Cn` y se resuelve mediante `solve`.

5.4. El forzado de la implicación

La implementación de:

```
force(Delta,I,(D=>G),C)
```

requiere una atención especial. De acuerdo con la definición de la relación $\models$ (ver definición 7), `Delta` se aumenta con la cláusula `D`.

Cuando se llega a este punto del cómputo, suponiendo que el conjunto `I` se ha calculado con respecto a la base de datos `Delta`, teniendo en cuenta el estrato i y la iteración n, se debe verificar:

$$(A, C) \in I \Leftrightarrow (A, C) \in T_i^n(I')(\texttt{Delta}),$$

donde I' es el punto fijo del estrato $i-1$, que se ha construido para `Delta`. Tal y como se ha visto en la teoría, el siguiente paso es probar:

$$T_i^n(I'), \texttt{Delta} \cup \{\texttt{D}\} \models (\texttt{G}, \texttt{C}).$$

Pero el problema surge en cómo debe calcularse:

$$T_i^n(I')(\texttt{Delta} \cup \{\texttt{D}\}).$$

La interpretación `I` no es válida para este cálculo.

- En primer lugar porque $I(\Delta) \subseteq I(\Delta \cup \{D\})$ no tiene porque ser cierto para cualesquiera I, Δ y D.
- En segundo lugar, porque `I` se ha construido teniendo en cuenta la base de datos incial `Delta`. En concreto, el punto fijo I' se ha calculado para `Delta`, y por ello representa fix_{i-1}(`Delta`).

En conclusión, ¿qué se sabe sobre el conjunto $T_i^n(I')$(`Delta` $\cup$ {`D`})? Nada.

El problema es que la definición del operador T_i no es *constructiva* para el caso de la implicación, debido al incremento de cláusulas en `Delta`. Para solventar este problema se ha tomado una posición conservadora:

- Se busca el máximo estrato de los predicados que aparecen en `G`, esto es:

$$\texttt{StG} = max\{St \mid (p, St) \in \texttt{Stratification}, \text{ siendo } p \text{ un símbolo de predicado en } \texttt{G}\}.$$

- Se calcula localmente el punto fijo del estrato `StG` para `Delta` $\cup$ {`D`}, que se denominará $fix_{\texttt{StG}}$.
- Finalmente se usa este punto fijo local para forzar el objetivo

$$fix_{\texttt{StG}}, \texttt{Delta} \cup \{\texttt{D}\} \models (\texttt{G}, \texttt{C}).$$

El código es el que aparece a continuación:

```
force(Delta,Stratification,I,(D=>G),C) :-
  !,
  elab(D,De),
  localClauses(De,Ls),
  addLocalClauses(Ls,Delta,Delta1),
  getMaxStrat(G,Stratification,StG),
  fixPointStrat(Delta1,Stratification,StG,Fix),
  force(Delta1,Stratification,Fix,G,C).
```

El predicado `elab` devuelve las reglas elaboradas de las cláusulas de `D`, tras esto `localClauses` las transforma en la representación que ya se había visto `hhcnClause (Vars,Head,Body)`. Llamando a `addLocalClauses` se obtiene la base de datos extendida `Delta1` = `Delta` $\cup$ {`D`}. La ejecución

$$\texttt{fixPointStrat}(\texttt{Delta1}, \texttt{Stratification}, \texttt{StG}, \texttt{Fix})$$

devuelve `Fix` = $fix_{\texttt{StG}}$(`Delta1`).

Una vez que se ha calculado `Fix`, se usará para el cálculo de `G` con el conjunto aumentado `Delta1`. Esto corresponde a calcular:

$$\texttt{force}(\texttt{Delta1}, \texttt{Stratification}, \texttt{Fix}, \texttt{G}, \texttt{C}),$$

lo cual implica $T_i^n(I'), \texttt{Delta} \cup \{\texttt{D}\} \models (\texttt{G}, \texttt{C})$, que era lo que se buscaba. Esta solución acarrea un problema adicional: Considérese `Delta` de la forma `A :- D =>G`, con $i = str(\texttt{A})$ siguiendo la defínción 4, se deduce que $\texttt{StG} \leq i$.

Durante el cómputo de $fix_i(\texttt{Delta})$, el predicado `opT` utiliza esta cláusula para buscar pares `(A,C)` que añadir a la interpretación `I`. Después se llama a:

```
force(Delta,Stratification,I,(D=>G),C)
```

y este a su vez llama a:

$$\texttt{fixPointStrat}(\texttt{Delta1}, \texttt{Stratification}, \texttt{StG}, \texttt{Fix}),$$

donde

$$\texttt{Delta1} = \texttt{Delta} \cup \{\texttt{D}\}$$

(módulo elaboración y renombramiento de variables). Entonces, si $\texttt{StG} = i$, es necesario construir $fix_i(\texttt{Delta1})$. Por tanto se volverá a añadir la cláusula `A :- D =>G` dado que el estrato de `A` es i. Lo cual lleva a un bucle infinito dado que `Delta1` se aumenta con `D` una vez más. Sin embargo, imponiendo la condición $\texttt{StG} < i$, el punto fijo $\texttt{Fix} = fix_{\texttt{StG}}(\texttt{Delta1})$ se puede construir correctamente.

Ejemplo 10 Dado el programa {`d(mad), g(mad), a(X):- d(X)=>g(X)`} se muestra a continuación el comportamiento del sistema en función de la estratificación asignada:

- Si los predicados `a`, `d` y `g` están todos en el estrato 1, calcular el significado de `a` lleva a aumentar el programa con una instancia fresca del antecedente `d(X')` y volver a iterar. Después se trata de extraer información para `a` y de nuevo se aumenta el programa con `d(X'')`. Y así sucesivamente, lo que lleva a un bucle infinito.
- Sin embargo, si se impone la condición $str(\texttt{g}) < str(\texttt{a})$, el predicado `a` pasa a estar en el estrato 2, y por tanto se calcularía localmente sólo hasta el estrato primero, no entraría a un bucle infinito y se llegaría al punto fijo formado por los pares: `(d(mad),true)`, `(g(mad),true)` y `(a(mad),true)`. □

De acuerdo con lo anteriormente visto se han incorporado nuevas dependencias al grafo de dependencias, de manera que se asegure que $\texttt{StG} < str(\texttt{A})$ para estos casos, como se verá en la siguiente sección.

5.5. Implementación del grafo de dependencias

En [49] se define el algoritmo para calcular el grafo de dependencias de las fórmulas de $HH_\neg(\mathcal{C})$. Debido a las nuevas restricciones que se deben imponer por las implicaciones anidadas y las funciones de agregación, se hace necesaria una nueva definición de estratificación para una base de datos. Las implicaciones anidadas introducen dependencias negativas en el grafo. Más concretamente, si `A :- G` es una cláusula y `G` contiene un subobjetivo de la forma `D =>G'`, el estrato de los símbolos de predicados

- $dpClause(A) = <\emptyset, \{p_A\}>$
- $dpClause(D_1 \wedge D_2) = <E_1 \cup E_2, N_1 \cup N_2>$
  si $dpClause(D_1) = <E_1, N_1>$ y $dpClause(D_2) = <E_2, N_2>$
- $dpClause(\forall x\ D) = dpClause(D)$
- $dpClause(G \Rightarrow A) = <E_G \cup \bigcup_{n \in N_G} \{n \rightarrow p_A\} \cup \bigcup_{\neg n \in N_G} \{n \stackrel{\neg}{\rightarrow} p_A\}, \{p_A\}>$
  si $dpGoal(G) = <E_G, N_G>$

- $dpGoal(A) = <\emptyset, \{p_A\}>$
- $dpGoal(\neg A) = <\emptyset, \{\neg p_A\}>$
- $dpGoal(C) = <\emptyset, \neg preds(C)>$
- $dpGoal(C \Rightarrow G) = <E \cup \bigcup_{n \in preds(C), m \in preds(G)} \{n \stackrel{\neg}{\rightarrow} m\}, N \cup \neg preds(C)>$
  si $dpGoal(G) = <E, N>$
- $dpGoal(G_1 \wedge G_2) = dpGoal(G_1 \vee G_2) = <E_1 \cup E_2, N_1 \cup N_2>$
  si $dpGoal(G_1) = <E_1, N_1>$ y $dpGoal(G_2) = <E_2, N_2>$
- $dpGoal(\forall x\ G) = dpGoal(\exists x\ G) = dpGoal(G)$
- $dpGoal(D \Rightarrow G) =$
  $<E_D \cup E_G \cup \bigcup_{m \in preds(G)} (\bigcup_{n \in N_D} \{n \rightarrow m\} \cup \bigcup_{\neg n \in N_D} \{n \stackrel{\neg}{\rightarrow} m\}),$
  $N_D \cup \neg preds(G)>$
  si $dpClause(D) = <E_D, N_D>$ y $dpGoal(G) = <E_G, N_G>$

Notación:

- p_A: símbolo de predicado definido del átomo A
- $preds(F) = \{p \mid p$ es un símbolo de predicado definido que aparece $F\}$
- $\neg S = \{\neg p \mid p \in S\}$

Figura 5.2: Grafo de dependencias

en `G'` debe ser menor que el estrato del símbolo de predicado `p` perteneciente a `A`. Por tanto, las implicaciones anidadas producen arcos negativos desde los símbolos de predicado de `G` hacia p.

El algoritmo que calcula el grafo de dependencias aparece en la figura 5.2. Su definición está basada en las funciones *dpClause* y *dpGoal* que se definen por recursión mutua en la estructura de la fórmula a la que se aplican.

Estas funciones devuelven pares de la forma $<E, N>$, donde E es un conjunto de arcos de la forma $p \rightarrow q$ o $p \stackrel{\neg}{\rightarrow} q$, y N es un conjunto auxiliar de *nodos-enlace* que almacenan los símbolos de predicado que aparecen en la fórmula. Estos símbolos de predicado tienen anotaciones en tres casos:

- Si aparecen en un átomo negado, o
- Si aparecen en una implicación anidada, o
- Si aparecen en una función de agregación.

Esta anotación se propaga a los arcos que salen de dichos nodos. Mediante el uso de la función *dpClause* y *dpGoal* es sencillo calcular el grafo de dependencias de un conjunto de fórmulas Φ como la unión de los arcos que se obtienen para cada elemento del conjunto:

$$DG_\Phi = \bigcup_{D \in \Phi} \{E_D | dpClause(D) = <E_D, N>\} \cup \bigcup_{G \in \Phi} \{E_G | dpGoal(G) = <E_G, N>\}.$$

El grafo de dependencias se usa para definir la estratificación de programas del esquema $HH_\neg(\mathcal{C})$, que es, como se ha dicho, una condición sintáctica que asegura la terminación de los cómputos con átomos negados, implicaciones anidadas y funciones de agregación.

El algoritmo que encuentra una estratificación para un conjunto de fórmulas Φ (o bien detecta que no es estratificable) asocia a cada símbolo de predicado p un valor entero $X_p \in [1..N]$, donde N es el número de símbolos de predicado Φ. Tras esto se genera un sistema de inecuaciones, de forma que:

- cada dependencia $p \rightarrow q$ produce $X_p \leq X_q$
- cada dependencia $p \stackrel{\neg}{\rightarrow} q$ produce $X_p < X_q$.

Después, si es posible, se asocia un estrato a cada p en X_p, resolviendo el sistema de ecuaciones. En caso de que no sea posible resolver el sistema de ecuaciones, se envía un mensaje de fallo (`Non stratifiable program!`) y el sistema finaliza su ejecución.

Ejemplo 11 Considérese la cláusula:

$$D \equiv \forall x(G \Rightarrow p(x)), \text{ donde } G \equiv \exists y(q(x,y) \Rightarrow (r(x) \wedge s(y))) \wedge \neg t(x)$$

El cálculo de $dpClause(D)$ requiere calcular

$$dpGoal(G) = <\{q \rightarrow r, q \rightarrow s\}, \{q, \neg r, \neg s, \neg t\}>$$

Ambos arcos $q \rightarrow r, q \rightarrow s$ informan de que el punto fijo de q se debe computar antes o la vez que el de r y s. Los nodos $q, \neg r, \neg s, \neg t$ generan dependencias en el contexto externo de (D); r y s se etiquetan negativamente porque aparecen en un lado derecho o en una implicación anidada y t porque aparece explícitamente negada. Para la cláusula completa tenemos:

$$dpClause(D) = <\{q \rightarrow r, q \rightarrow s, q \rightarrow p, r \stackrel{\neg}{\rightarrow} p, s \stackrel{\neg}{\rightarrow} p, t \stackrel{\neg}{\rightarrow} p\}, \{p\}>$$

Los primeros dos arcos se han calculado previamente y los cuatro restantes provienen de la implicación de D (la más externa). Se conectan los nodos enlace obtenidos en *dpGoal* con el símbolo de predicado de p que aparece en la cabeza de D. Por otra parte, las marcas negativas $\neg$ producen arcos etiquetados negativamente.

Una estratificación es cualquier asociación de los símbolos de predicado $\{p, q, r, s, t\}$ con números naturales, de forma que se satisfagan el siguiente conjunto de inecuaciones:

$$\{X_q \leq X_r, X_q \leq X_s, X_q \leq X_p, X_r < X_p, X_s < X_p, X_t < X_p\}.$$

Por ejemplo, $X_p = 2$ y el resto asociadas con 1, es decir, todos los predicados en el estrato 1, salvo p en el estrato 2.

Intuitivamente, esto significa que antes de la evaluación de p, el resto de predicados se deben evaluar previamente, en particular q, que aparece en la implicación anidada.

Pero si se añade la cláusula:

$$D' \equiv \forall x \forall y(p(x) \Rightarrow q(x, y))$$

se genera el arco $p \rightarrow q$, y consecuentemente se añade la inecuación

$$X_p \leq X_q.$$

Lo cual hace que el sistema no tenga solución y que la base de datos que se está tratando no sea estratificable. □

Las dependencias negativas introducidas en el grafo, debido a implicaciones anidadas y funciones de agregación, hacen que haya un menor número de bases de datos estratificables y suponen una restricción al esquema original que permitía expresar cualquier base de datos de acuerdo con la sintaxis del lenguaje.

Sin embargo, como se ha establecido en el capítulo 2, el uso de mecanismo de estratificación para el cálculo de funciones de agregación es algo habitual en BDD, como se puede ver en la figura 2.1.4 de la página 18. Y la otra limitación se establece para las implicaciones dentro de las vistas y las consultas, que no están permitidas en otros lenguajes, lo cual hace que el sistema siga siendo más expresivo que otros lenguajes de bases de datos deductivas como Datalog.

Capítulo 6

Implementación de los sistemas de restricciones

En esta sección se describe la implementación del sistema de restricciones. En primer lugar se describe el sistema de tipos necesario para identificar los tipos de las variables, que se usan para identificar las restricciones con su resolutor correspondiente. Después se describe el sistema de restricciones incluyendo los valores, funciones y operadores predefinidos. Tras esto, se explica la implementación de un resolutor concreto de dominios finitos que hace uso de los resolutores subyacente SWI-Prolog [82]. Finalmente se detalla cómo se resuelven las funciones de agregación.

6.1. Sistema de restricciones

De acuerdo a lo introducido en la sección 3.1, un sistema de restricciones debe incluir el lenguaje de restricciones y la relación de deducibilidad entre restricciones. Los sistemas de restricciones implementados incluyen la sintaxis específica para construir restricciones y resolutores para comprobar la satisfactibilidad de estas. En cuanto a la construcción de restricciones se dispone de los siguientes valores, símbolos, conectivas y cuantificadores: "`true`", "`false`", "`=`","` =<`", "`,`", "`not`" and "`ex(X,C)`" que representan $\top$, $\bot$, $\approx$, $\leq$, $\wedge$, $\neg$ y $\exists$`X C`. Además también se incluye "`;`" para $\vee$ y "`/=`" para la negación de $\approx$.

Se han implementado tres sistemas de restricciones para el esquema $HH_{\neg}(\mathcal{C})$: booleanos, reales y dominios finitos. El primero incluye el tipo `bool` y consta de los componentes habituales e incorpora el cuantificador universal (`fa(X,C)`, donde `X` es una variable y `C` una restricción).

El sistema de restricciones real incluye el tipo `real` (conjunto infinito de valores numéricos), y operadores de restricciones reales (`+`, `-`, `*`, ...) además de las funciones (`abs`, `sin`, `exp`, `min`, ...).

El sistema de dominios finitos son la familia de sistemas de restricciones específicas sobre conjuntos enumerables. Se incluyen tipos enumerados así como tipos numéricos enteros (finitos). A diferencia de los sistemas anteriores, los de dominios finitos no

tienen un tipo predefinido dado que el usuario puede declarar los tipos que necesite. El sistema incluye operadores de comparación como (`>`, `>=`, ...), restricciones universalmente cuantificadas (`fa(X,C)`), y la restricción de dominio `X in Range`, donde `Range` es un subconjunto de valores especificados con `V1..V2`, que incluye el rango de valores en el intervalo cerrado entre `V1` y `V2`, así como `R1\/R2`, que representa la unión de rangos. Un dominio finito numérico incluye también operadores (como `+`, `-`, ...) y funciones sobre restricciones (como `abs`, `min`, ...). Nótese que las funciones primitivas relevantes deben ser coherentes con su semántica pretendida (`+` puede no ser relevante para booleanos, aunque se puede usar en el sistema implentado). Se permite el uso de los mismos símbolos en diferentes sistemas, tanto `constr(real, X>Y)` como `constr(month, X>Y)` tienen sentido en los sistemas de restricciones `real` y `month` respectivamente.

Las funciones de agregación `count`, `min`, y `max` se pueden aplicar en cualquier sistema, mientras que `sum` y `avg` sólo se pueden aplicar a dominios numéricos.

6.2. Resolutores

Para los sistemas de restricciones se hace uso de la relación de deducibilidad de la lógica clásica con igualdad. Esta relación de deducibilidad satisface las condiciones mínimas vistas en la sección 3.1. Para implementar esta relación se ha desarrollado un interfaz genérico `solve(I,C,SC)` para $\texttt{C} \vdash_{\mathcal{C}} \texttt{SC}$ que resuelve la restrición `C`. Es decir, produce una forma *resuelta* que se denomina `SC`, si `C` es satisfactible, o falla en otro caso.

Una forma resuelta `SC` asociada a la restricción `C` es una restricción simplificada y más legible que consiste en una restricción simple o una disyunción de restricciones simples, teniendo en cuenta que una restricción simple no incluye disyunciones, cuantificadores o negación.

El interfaz genérico `solve` que resuelve restricciones se implementa con el siguiente código:

```
solve(I,C,SC) :-
  simplify_ground_ctr(C,SGC),
  partition_ctr(I,SGC,DCs),
  solve_ctr_list(I,DCs,SDCs),
  ctr_list_to_ctr(SDCs,CC),
  simplify_ctr(CC,SC).
```

La interpretación `I` se incluye para el cómputo de funciones de agregación tal y como se verá más adelante. El código, en primer lugar, llama a `simplify_ground _ctr`, que se encarga de simplificar restricciones primitivas cerradas. A continuación, la llamada al predicado `partition_ctr` divide la restricción de entrada y devuelve una lista compuesta de restricciones. Esta división se puede siempre llevar a cabo si la restricción pertenece a un único dominio. Cuando existe la combinación de dominios no siempre se puede dividir correctamente. En estos casos el sistema informa mediante una excepción (`Unsupported constraint combination`).

La llamada al predicado `solve_ctr_list` envia cada componente a su resolutor correspondiente, como por ejemplo el de dominio finito `solveFD` (que se describe más adelante). Tras esto, la restricción resuelta, representada por una lista, se vuelve a transformar en una restricción conjuntiva mediante el predicado `ctr_list_to_ctr`. Finalmente se simplifica usando las reglas de De Morgan mediante el predicado `simplify_ctr`.

La interfaz `solve(Dom,I,C,SC)` añade el parámetro `Dom` y se usa cuando el dominio es conocido. Así se envía directamente a su resolutor correspondiente.

Se describe a continuación la implementación de uno de los resolutores concretos propuestos para el esquema $HH_{\neg}(\mathcal{C})$, el de dominios finitos. En general se hace uso de los resolutores subyacentes de SWI-Prolog [82] en la implementación de los sistemas de restricciones booleanos, reales y de dominios finitos. Para restricciones asociadas a dominios finitos definidos por el usuario es necesario asociar los valores no numéricos de dominios finitos con enteros. Después de la fase de resolución estos valores se traducen a sus valores iniciales del dominio origen. Para aquellas restricciones que el resolutor subyacente de dominios finitos no puede manejar (cuantificadores o disyunciones) se han implementado mecanismos específicos que se verán más adelante. Dado que SWI-Prolog no incluye resolutor de booleanos, se ha hecho uso del resolutor de dominios finitos para manejar este tipo de restricciones. Además se ha incluido el tipo predefinido `bool`, que se trata como cualquier otro tipo enumerado.

Se tienen disponibles los siguientes predicados para los sistemas de restricciones de dominio finito y booleano.

- `solveFD(+Domain,+Interpretation,+Constraint,-SolvedConstraint)`
  Resuelve la restricción de entrada `Constraint` sobre `Domain` haciendo uso de la interpretación `Interpretation` y devuelve `SolvedConstraint` como la forma resuelta, si es satisfactible; en otro caso devuelve `false`.

- `constr_conjFD(+Domain,+Interpretation,-C1,+C2,+C)`
  Calcula (usando `Interpretation`) la componente `C1` de la conjunción `C1,C2` tal que `C1,C2` $\vdash_{\mathcal{C}}$ `C`, donde $\mathcal{C}$ es el sistema de restricciones asociado a `Domain`. Este predicado se usa para el forzado de la implicación como se ha visto en la sección 5.3.

Para todos los sistemas de restricciones se ha considerado la deducibilidad de la lógica clásica. Por tanto este último predicado se puede implementar de la siguiente forma:

```
constr_conjFD(Dom,I,C1,C2,C) :-
  solveFD(Dom,I,(not(C2);C),C1), !,
  C1\==false.
```

El codigo mostrado en la figura 6.1 implementa el codigo del resolutor de dominio finito `solveFD`. En la línea `(05)` se sustituyen variables cuantificadas por nuevas para evitar un conflicto por captura de variables (*name clash*). En la línea `(07)` se asocian los valores del dominio finito con valores enteros. Después en la línea `(21)` se vuelven a cambiar por los valores del dominio orginal. El núcleo de la resolución

de restricciones aperece en las líneas `(09)-(11)`, donde en primer lugar, la restricción trata de resolverse (como se verá en el siguiente párrafo mediante el predicado `solveFD_ctr`). En segundo lugar, se comprueba la satisfactibilidad, esto es, se intenta encontrar un solución concreta mediante etiquetado (labeling). En tercer lugar, el almacén de variables subyacente se proyecta con respecto a las variables relevantes. Se denominan variables relevantes a aquellas que aparecen en la restricción inicial, las nuevas que calcula el resolutor subyacente y también aquellas asociadas a la consulta que formula el usuario, es decir, aquellas asociadas al predicado `query` que se genera internamente cuando se hace una consulta como se ha visto en la sección 4.1. En las lineas `(13)-(15)` se da formato a la estructura de datos de salida.

```
(01) solveFD(Dom,I,C,SC) :-
(02)   copy_term(C,FC),                % Las variables de entrada se guardan
(03)   get_vars(C,Vars),               % se cambian las variables
(04)   get_vars(FC,FVars),             %  por las nuevas
(05)   swap_qvars_by_fvars(FC,QFC),    % Se reemplazan las cuantificadas
(06)   constrain_domains(QFC,Dom),     % Variables restringidas al dominio actual
(07)   domain_to_int(QFC,Dom,IC),      % Asociación enumerado a entero
(08)   bagof((FVars,Cs,Sat),           % Lista (Vars,Restric,Satisfac)
(09)     (solveFD_ctr(IC,Dom,I,true),  % Resolución
(10)      satisfiable(IC,Sat),         % Satifactibilidad
(11)      project_ctrs(FVars,Vars,Cs)  % Proyección de restricciones
(12)     ), LFVarsCsS), !              % Lista  (VarsNuevas,Restric,Satisfac)
(13)   filter_ctr_list(LFVarsCsS,LICs), % Filtrado de restricciones
(14)   simplify_disj_list(LICs,SLICs), % Simplificación en disyunción
(15)   disj_list_to_ctr(SLICs,ISC),    % Conversión Lista->Restricción
(16)   get_vars(ISC,FVSC),
(17)   (fresh_vars_option(off),        % No se permiten variables nuevas en la
(18)    fresh_vars(Vars,FVSC) ->       %  restricción salida
(19)    SC = C                         % En caso de fallo se descarta la salida
(20)    ;                              %  y se devuelve la entrada
(21)    int_to_domain(ISC,Dom,SC)).    % Si no, intercambio al Dom original
(22) solveFD(_Dom,_I,_C,false).        % Si no, falso.
```

Figura 6.1: El predicado `solveFD` para resolución de restricciones de dominio finito

A continuación se describe el predicado:

`solveFD_ctr(+Constraint,+DomainName,+Interpretation,-Satisfiable)`,

el cual recibe una restricción, un nombre de dominio y una interpretación y devuelve el valor `true` si la restricción es satisfactible o bien el valor `false` si no lo es. Si es satisfactible, la forma resuelta permanece almacenada en el resolutor subyacente de SWI-Prolog, está restricción se recuperará cuando se muestre el punto fijo final.

El siguiente código corresponde a uno de los casos para este predicado, en este caso la restricción puede ser manejada por el resolutor de SWI-Prolog (aquí, `#>` es el operador de comparación de dominio finito que incluye el resolutor subyacente). El predicado `replace_aggrs_by_result` sustituye la aparición de agregados por su resultado, como se describirá en la siguiente sección.

```
solveFD_ctr(X>Y,Dom,I,true) :-
  !,
  replace_aggrs_by_result(X,Dom,I,EX),
  replace_aggrs_by_result(Y,Dom,I,EY),
  EX#>EY.
```

La negación, como se muestra a continuación, se maneja explícitamente dado que se puede aplicar a restricciones que no soporte directamente el resolutor Prolog subyacente:

```
solveFD_ctr(not(C),Dom,I,B) :-
  !,
  complement(C,NotC),
  solveFD_ctr(NotC,Dom,I,B).
```

En estas líneas el predicado `complement` calcula el complemento de una restricción (por ejemplo, `X#=<Y` es la restricción complementaria de `X#>Y`).

Un ejemplo de manejo de restricciones no soportadas por el resolutor es la disyunción, que se calcula recolectando todas las respuestas posibles (véase la linea `(08)` en figura 6.1). Se resuelve esta restricción como sigue:

```
solveFD_ctr((C1;_C2),Dom,I,true) :-
  solveFD_ctr(C1,Dom,I,true).
solveFD_ctr((_C1;C2),Dom,I,true) :-
  !,
  solveFD_ctr(C2,Dom,I,true).
```

Finalmente se describen los cuantificadores. El cuantificador existencial se implementa haciendo uso del predicado `satisfiable(FC,Domain,true)`. Este predicado intenta encontrar un valor concreto que satisfaga la restricción `FC`:

```
solveFD_ctr(ex(X,C),Dom,I,B) :-
  !,
  % Reemplazo de X por una nueva variable _FX en C:
  swap(X,_FX,C,FC),
  constrain_domains(FC,Dom),
  (solveFD_ctr(FC,Dom,I,true),
   % Comprobación de satisfactibilidad:
   satisfiable(FC,Dom,true),
   B=true
   ;
   B=false
  ).
```

En el caso del cuantificador universal, la restricción `fa(X,C)` se reemplaza por la conjunción `C[X/v1], ...,C[X/vn]`, donde vi (`1`$\leq i \leq$`n`) son todos posibles valores del dominio de `X`.

```
solveFD_ctr(fa(X,C),Dom,I,B) :-
  !,
  get_domain_bounds(Dom,L,U),
  (solve_forall(X,Dom,C,L,I,U) ->
   B=true
   ;
   B=false
  ).
```

Nótese que los cortes al comienzo del cuerpo de cada cláusula se deben a la existencia de un caso por defecto que corresponde a una restricción no válida y que implica el lanzamiento de una excepción, como se puede observar en el código:

```
solveFD_ctr(C,_DN_I,_B) :-
  nl, write('EXCEPTION: Unsupported constraint: '), write(C), nl,
  !,
  fail.
```

El resolutor de restricciones para reales está implementado de una manera similar pero más sencilla dado que no tiene cuantificador universal ni necesita la conversión de dominio de sus valores a enteros. Los predicados `solveR` y `constr_conjR` están disponibles de manera análoga a `solveFD` y `constr_conjFD` respectivamente.

6.3. Funciones de agregación

Se han implementado funciones de agregación sobre el dominio real y sobre los dominios finitos. Dentro de una restricción de la forma `constr(Domain, Exp1 opComp Exp2)`, se pueden encontrar expresiones de agregado en ambas expresiones `Exp1` y `Exp2`, donde `opComp` es un operador de comparación (`=`, `/=`, `>`, `>=`, `<`, `=<`).

Para esta implementación ha sido necesario introducir la interpretación en el resolutor. La interpretación se propaga dentro del resolutor concreto, `solveFD` o `solveR`.

Como se ha explicado, el predicado `replace_aggrs_by_result` cambia cada aparición de una función de agregación por un valor concreto antes de la llamada al resolutor subyacente de SWI-Prolog en todos los operadores de comparación antes mencionados

Este predicado llama a `compute_aggr` que calcula el resultado de cada operador de agregación, como se puede ver a continuación:

```
compute_aggr(sum(At,Var),I,Res) :-
   !,
   get_values(I,At,Var,S),
   compute_sum(S,0,Res).
```

El predicado `get_values` llama a `lookUpAll` para obtener valores concretos de las igualdades cerradas de la forma `Var = value` para la interpretación `I`, con respecto al átomo `At` y la variable `Var`. Después, estos valores se almacenan en la lista `S`.

Finalmente, `S` se usa para calcular el resultado final `Res` con respecto a la función concreta.

La implementación de la función `count` es ligeramente distinta dado que no tiene una variable como argumento. En este caso también se hace uso del predicado `lookUpAll`, pero después el resolutor calcula el número de apariciones del átomo dentro de la interpretación.

Capítulo 7

El sistema en acción

En este capítulo se mostrarán las distintas etapas de cálculo de punto fijo y evaluación de consultas cuando se trabaja con ejemplos concretos. En primer lugar se verá con todo de detalle un ejemplo para ilustrar el comportamiento del sistema y a continuación se verán otros ejemplos de manera más concisa. El sistema, junto con una batería de ejemplos, entre ellos `bank.hhc`, que ilustra la primera sección, está disponible en:

`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems`.

7.1. Base de datos para gestión bancaria

En esta sección se presenta una extensión del ejemplo que se utilizó para ilustrar los agregados en la sección 4.4. Este ejemplo utiliza tipos enumerados y valores reales. Veremos en primer lugar cómo se define la base de datos. A continuación cómo se calcula el punto fijo y finalmente cómo funciona el sistema cuando se plantean consultas.

7.1.1. Definición de la base de datos

En primer lugar se deben definir los dominios sobre los que se va a trabajar, en concreto el dominio de los clientes y las sucursales bancarias:

```
domain(client_dt,[smith,brown,mcandrew]).
domain(branch_dt,[lon,mad,par]).
```

Los valores `lon`, `mad` y `par` representan Londres, Madrid y Paris respectivamente. Las declaraciones de tipo para los predicados `branch` y `client_id`, que se definen posteriormente, son de la forma:

```
type(branch(branch_id,client_dt)).
type(client_id(client_dt,real)).
```

Así mismo, se proporciona el tipo para el resto de predicados, cuyos argumentos son de tipo `real`. Las declaraciones quedarían:

```
type(client(real,real,real)).
type(pastDue(real,real)).
type(mortgageQuote(real,real)).
type(debtor(real)).
type(interestRate(real,real)).
type(newMortgage(real,real)).
type(personalCredit(real,real)).
type(hasMortgage(real)).
type(gotMortgage(real)).
type(liquid(real)).
type(avg_salary(real)).
```

La base de datos *extensional* viene dada por los hechos que aparecían en el ejemplo de la sección 4.4:

```
% client_id(Name, Ident)    % client(Ident, Balance, Salary)
client_id(smith,1.0).       client(1.0,2000.0,1200.0).
client_id(brown,2.0).       client(2.0,1000.0,1500.0).
client_id(mcandrew,3.0).    client(3.0,5300.0,3000.0).
```

Además se añade información sobre deudas con el predicado `pastDue`, cuota hipotecaria con `mortgageQuote` y el predicado `branch` que asocia a los clientes con las oficinas:

```
% pastDue(Ident, Amount)    % mortgageQuote(Ident, Quote)
pastDue(1.0,3000.0).        mortgageQuote(2.0,400.0).
pastDue(3.0,100.0).         mortgageQuote(3.0,100.0).

% branch(Office, Name)
branch(lon,smith).
branch(mad,brown).
branch(par,mcandrew).
```

Como restricción adicional se asume que cada cliente tiene una única cuota hipotecaria. A continuación definimos la parte *intensional* de la base de datos por medio de vistas. La primera introduce la idea de que un cliente tiene una hipoteca si hay una cuota asociada a él.

```
% hasMortgage(Ident)
hasMortgage(I):- ex(Q,mortgageQuote(I,Q)).
```

Un moroso es un cliente que adeudan cantidades superiores a su saldo.

```
% debtor(Ident)
debtor(I):- client(I,B,S), pastDue(I,A),constr(real, A>B).
```

El interés aplicable a un cliente es variable, depende del sueldo de dicho cliente y se especifica mediante la siguiente relación:

```
% interestRate(Ident, Rate)
interestRate(I,2.0):- client(I,B,S),constr(real, B<1200.0).
interestRate(I,5.0):- client(I,B,S),constr(real, B>=1200.0).
```

Es decir, a los clientes con un saldo menor a 1200 se les aplica un interés de 2 % y al resto un interés de 5 %. La siguiente relación especifica que a un cliente que no sea moroso se le puede conceder una hipoteca en dos casos. Primero, en caso de que aún no tenga una hipoteca y la cuota sea menor que el 40 % de su salario. Y segundo, en caso de que aunque tenga una cuota, la suma de la antigua y la nueva cuota no alcance este porcentaje.

```
% newMortgage(Ident, Quote)
newMortgage(I,Q) :- client(I,B,S),
                    not(debtor(I)),
                    not(hasMortgage(I,Q1)),
                    constr(real, Q<=0.4*S).
newMortgage(I,Q) :- client(I,B,S),
                    not(debtor(I)),
                    mortgageQuote(I,Q2),
                    constr(real, Q+Q2<=0.4*S).
% gotMortgage(Ident)
gotMortgage(I):- ex(Q,newMortgage(I,Q)).
```

Si un cliente cumple los requisitos para que se le conceda una hipoteca sólo le puede dar un prestamo personal de una cantidad inferior a 6000. La idea es que al cumplir los requisitos para que se le conceda la hipoteca en caso de pedir más dinero se debería hacer pidiendo una hipoteca. En otro caso se le puede conceder una cantidad entre 6000 y 20000.

```
% personalCredit(Ident, Amount)
personalCredit(I,A) :- (gotMortgage(I),
                        constr(real, A<6000.0))
                       ;
                       (not(gotMortgage(I)),
                        constr(real, (A>=6000.0, A<20000.0))).
```

Se puede definir una vista de los clientes cuya cuota hipotecaria es mayor que 100:

```
% accounting(Ident, Salary, Quote)
accounting(I,S,Q):- client(I,B,S),
                    mortgageQuote(I,Q),
                    constr(real, Q>=100).
```

A continuación incluimos algunos ejemplos con funciones de agregación vistos en el ejemplo de la sección 4.4. El activo líquido se expresa como suma de los saldos de los clientes:

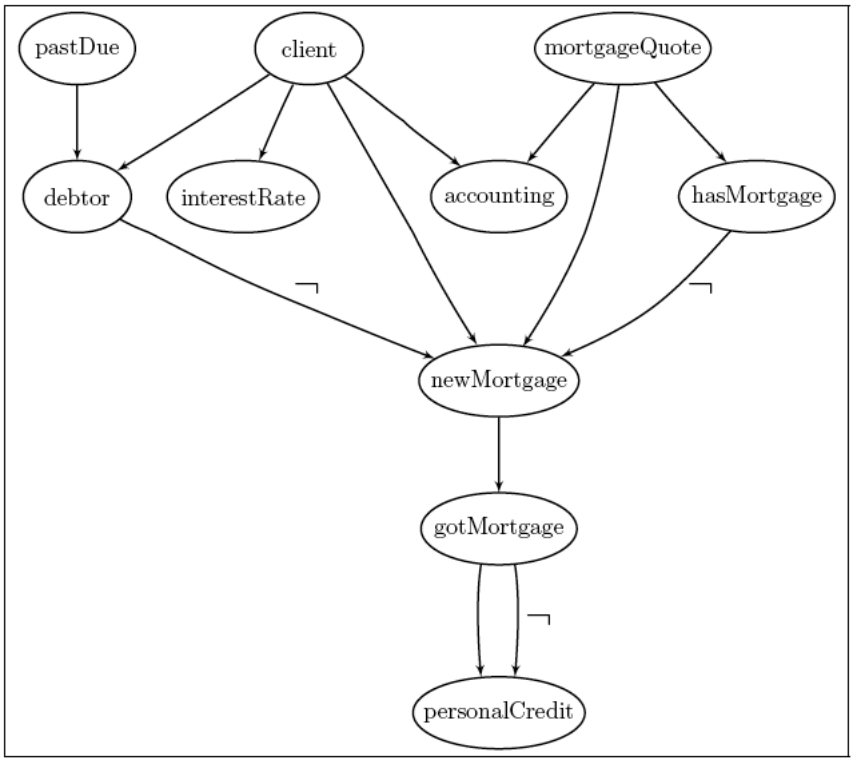

Figura 7.1: Grafo de dependencias del ejemplo 5.

```
% liquid(Amount)
liquid(A) :- constr(real, A = sum(client(I,B,S),B)).
```

El salario medio de los clientes se define como:

```
% avg_salary(Average)
avg_salary(Avg) :- constr(real, Avg = avg(client(I,B,S),S)).
```

7.1.2. Cálculo del punto fijo

En lo que sigue, `Delta` representa el conjunto de cláusulas de la sección anterior correspondiente a la base de datos intensional y extensional del banco. Para el cálculo del punto fijo, en primer lugar, se calcula el grafo de dependencias correspondiente a `Delta`. Las dependencias para este ejemplo aparecen en la figura 7.1.

Los nodos asociados a los predicados `branch` y `client_id` no aparecen por tratarse de nodos aislados. Además, en este ejemplo hay dos dependencias negativas debidas a las funciones de agregación que no aparecen en la figura para simplificar, una desde `client` a `liquid` y otra desde `client` a `avg_salary`. El grafo de dependencias

completo se representa internamente como sigue:

```
[ (client->debtor),
(pastDue->debtor),
(client->interestRate),
(client->interestRate),
(client->newMortgage),
(hasMortgage*->newMortgage),
(debtor*->newMortgage),
(client->newMortgage),
(debtor*->newMortgage),
(mortgageQuote->newMortgage),
(gotMortgage->personalCredit),
(gotMortgage*->personalCredit),
(mortgageQuote->hasMortgage),
(newMortgage->gotMortgage),
(client*->liquid),
(client*->avg_salary)]
```

Donde `->` representa una dependencia simple y `*->` representa una dependencia negativa. Después se calcula la estratificación para ese grafo que asocia:

- Estrato 1 a `client`, `pastDue`, `mortgageQuote`, `debtor`, `interestRate`, `hasMortgage`, `accounting`, `client_id` y `branch`.
- Estrato 2 a `newMortgage`, `gotMortgage`, `liquid` y `avg_salary`.
- Estrato 3 a `personalCredit`.

Intuitivamente, `personalCredit` debe estar en un estrato superior a `gotMortgage` dado que depende negativamente de `gotMortgage`, como puede verse en:

```
personalCredit(I,A) :- (gotMortgage(I),
                        constr(real, A<6000.0))
                       ;
                       (not(gotMortgage(I)),
                        constr(real, (A>=6000.0, A<20000.0))).
```

De la misma forma, `avg_salary` está en el estrato 2 dado que depende negativamente `client`, en este caso por la presencia de una función de agregación.

```
avg_salary(Avg) :- constr(real, Avg = avg(client(I,B,S),S)).
```

Dado que `Delta` es estratificable, se calcularán los puntos fijos de la forma fix_i (`Delta`) para i, desde $i = 1$ hasta 3. El punto fijo final `Fix=`fix_3(`Delta`) se obtiene cuando el sistema ejecuta el predicado:

`fixPointStrat`(`Delta`, `Stratification`,3, `Fix`).

A continuación se muestra el cálculo por estratos de dicho punto fijo.

1. Cálculo de fix_1(`Delta`). La primera iteración del operador T_1 sobre el conjunto vacío, corresponde a la ejecución de:

$$\texttt{opT}(\texttt{Delta}, \texttt{Delta}, \texttt{Stratification}, 1, [], \texttt{TI})$$

(veasé sección 5.1) que obtiene en `TI` los pares que definen la base de datos extensional:

```
(client(1.0, 2000.0, 1200.0), true),
(client(2.0, 1000.0, 1500.0), true),
(client(3.0, 5300.0, 3000.0), true),
(branch(lon, smith), true),
(branch(mad, brown), true),
(branch(par, mcandrew), true),
(client_id(smith, 1.0), true),
(client_id(brown, 2.0), true),
(client_id(mcandrew, 3.0), true)
(pastDue(1.0,3000.0), true),
(pastDue(3.0,100.0, true),
(mortgageQuote(2.0, 400.0), true),
(mortgageQuote(3.0, 100.0), true)
```

El cómputo del punto fijo de este primer estrato requiere una iteración más de T_1, lo que añade los siguientes pares:

```
(debtor(1.0), true),
(interestRate(2.0, 2.0), true),
(interestRate(X,Y), ((X=1.0, Y=5.0); (X=3.0, Y=5.0))),
(accounting(X,Y,Z), ((Y=400.0, Z=1500.0, X=2.0);
                     (Y=100.0, Z=3000.0, X=3.0))),
(hasMortgage(X), (X=2.0;X=3.0))
```

La siguiente iteración no añade nada, por tanto, se ha alcanzado el punto fijo.

2. Cálculo de fix_2(`Delta`). La primera iteración del operador T_2 comienza con fix_1(`Delta`) y añade los siguiente pares al punto fijo:

```
(newMortgage(X,Y), ((Y=<200.0, X=2.0); (Y=<1100.0, X=3.0)))
(avg_salary(1900.0), true)
(liquid(8300.0), true)
```

En la segunda iteración, T_2 completa el punto fijo añadiendo el par:

```
(gotMortgage(X), (X=2.0;X=3.0))
```

3. Cálculo de fix_3(Delta). El punto fijo final requiere una iteración más de T_3 sobre el punto fijo fix_2(Delta) previamente calculado. Esta iteración se lleva a cabo ejecutando

$$\texttt{iterT}(\texttt{Delta}, \texttt{Stratification}, 3, fix_2(\texttt{Delta}), \texttt{FixSt}),$$

que introduce el último par correspondiente al predicado personalCredit:

```
(personalCredit(X,Y), ((Y>=6000.0,Y<20000.0, X/=2.0, X/=3.0);
                       (Y<6000.0, X=2.0);
                       (Y<6000.0, X=3.0)))
```

Esto completa el cómputo del punto fijo y FixSt=fix_3(Delta) contiene los pares (A, C) que corresponden con dicho punto fijo, es decir, se captura la semántica de la base de datos. Esta información, así como el grafo de dependencias y la estratificación se almacenará para futuras manipulaciones de la base de datos.

7.1.3. Consultas

El usuario puede ahora formular consultas a la base de datos. Como ejemplo de una primera consulta sencilla se puede plantear: *¿pertenecen todos los clientes a la oficina de Madrid?* que se formula como:

```
HHn(C)> fa(A,branch(mad,A)).
Answer: false
```

Tal y como se explica en la sección 4.1, cuando una consulta no cambia la estratificación, el sistema usa el punto fijo almacenado como parámetro del predicado force. Como esta consulta no implica ningún cambio en el grafo de dependencias, dado que no introduce ninguna dependencia, se puede resolver usando el punto fijo almacenado (correspondiente al apartado anterior). Una cuantificación universal sobre un dominio finito se traduce de manera natural a una restricción conjuntiva que se obtiene instanciando la variable cuantificada con cada elemento del dominio. En este ejemplo, fa(A,branch(mad,A)) requiere saber la restricción asociada a branch(mad,A). Para ello se explora el punto fijo mediante la llamada lookUpAll(branch(mad,A),I,Cs). Como resultado se obtiene la restricción:

$$(\texttt{mad} = \texttt{lon}, \texttt{A} = \texttt{smith}); (\texttt{mad} = \texttt{mad}, \texttt{A} = \texttt{brown}); (\texttt{mad} = \texttt{par}, \texttt{A} = \texttt{mcandrew})$$

lo cual es equivalente a A=brown. Después se trata fa(A,(A=brown)), para resolver el cuantificador universal se instancia la variable con todos los posibles valores del dominio, en este caso se transforma en la conjunción:

$$(\texttt{brown} = \texttt{smith}), (\texttt{brown} = \texttt{brown}), (\texttt{brown} = \texttt{mcandrew}),$$

lo cual es trivialmente falso.

Un ejemplo de consulta con negación puede ser preguntar *qué cliente no tiene hipoteca*:

```
HHn(C)> not(hasMortgage(I)).
Answer: I/=3.0, I/=2.0
```

Para resolver esta consulta se llama al predicado `lookUpAll` para encontrar los pares que tienen la forma `(hasMortgage(N),C)` en el punto fijo. Como se puede ver en la sección anterior la `C` asociada es `(X=2.0; X=3.0)`. La restricción respuesta `C'` se construye enviando la negación de esta restricción al resolutor (`not(X=2.0; X=3.0)`) lo que devuelve una conjunción negativa como respuesta `(I/=2.0, I/=3.0)`.

Si interesa saber, para la misma consulta, el nombre de los clientes además del identificador, se haría uso del predicado `client_id`.

```
HHn(C)> not(hasMortgage(I)),client_id(N,I).
Answer: (N=smith, I=1.0;N=brown, I=2.0;N=mcandrew, I=3.0),
        I/=2.0, I/=3.0
```

Un ejemplo de uso de funciones de agregación con consultas hipotéticas es: *asumiendo que el cliente 2 (Brown) tiene una deuda, determinar cuál es la suma de las deudas en la base de datos*:

```
HHn(C)> pastDue(2.0,200.0) => constr(real, X=sum(pastDue(N,A),A)).
Answer: X=3300.0
```

A pesar de la implicación, este es otro caso en que la consulta no requiere que se cambie la estratificación puesto que la dependencia del predicado `pastDue` hacia si mismo no cambia el grafo. Para encontrar la respuesta se llama de nuevo al predicado `force` con el punto fijo almacenado. Como se puede ver en la sección 5, el forzado de la implicación requiere el cálculo del máximo estrato del antecedente. En este caso es el primer estrato. El sistema calcula el punto fijo para el programa extendido con el antecedente, mediante la llamada:

$$\texttt{fixPointStrat}(\texttt{Delta} \cup \{\texttt{pastDue(2.0,200.0) }\}, \texttt{Stratification}, 1, \texttt{Fix1}).$$

para obtener el punto fijo de la base de datos incrementada, en este caso, con el antecedente de la implicación. Finalmente se llamará al predicado `force` otra vez para forzar:

$$\texttt{constr}(\texttt{real}, \texttt{X} = \texttt{sum}(\texttt{pastDue}(\texttt{N}, \texttt{A}), \texttt{A})),$$

pero esta vez con el punto fijo temporal:

$$\texttt{Fix1} = fix_1(\texttt{Delta} \cup \{\texttt{pastDue}(2{,}0, 200{,}0)\})$$

como argumento. Esta llamada devolverá la restricción respuesta.

Nótese que este cálculo no requiere en este caso volver a computar todo el punto fijo $fix_3(\texttt{Delta})$ porque la implicación no hace cambiar la estratificación. El sistema se encarga de comprobarlo para evitar cómputos innecesarios.

A continuación se muestra un ejemplo de la situación contraria. Se trata de una consulta que se ha puesto ad hoc para mostrar esta situación y por tanto no tiene un significado que sea interesante.

```
HHn(C)> newMortgage(N,R) => interestRate(N,R).
Answer: (R=2.0,N=2.0) ; (R=5.0,N=1.0) ; (R=5.0,N=3.0)
```

La estratificación almacenada `Stratification` no es válida para resolver esta consulta dado que introduce una nueva dependencia entre los predicados `newMortgage` e `interestRate`. Por tanto el predicado `interestRate` debe estar en el estrato 2. Dado que se ha cambiado la estratificación, el punto fijo almacenado ahora no puede ser usado para resolver esta consulta y se añade una cláusula temporal a la base de datos de la forma:

$$\texttt{D} \equiv \texttt{query(N,R) :- newMortgage(N,R) =>interestRate(N,R)}$$

Para el cálculo del punto fijo local $\texttt{Fix}'$ se ha de usar:

$$\texttt{fixPointStrat}(\texttt{Delta}', \texttt{Stratification}', 3, \texttt{Fix}'),$$

donde $\texttt{Stratification}'$ es la nueva estratificación y $\texttt{Delta}' = \texttt{Delta} \cup \{\texttt{D}\}$.

Finalmente la restricción respuesta será aquella `C` que cumpla que $(\texttt{query}(\texttt{N}, \texttt{R}), \texttt{C})$ esté en $\texttt{Fix}'$, tal y como se ve en la sección 4.1. Tras esto la cláusula temporal `D` y el punto fijo $\texttt{Fix}'$ se deshechan y el punto fijo anterior se reestablece (no es necesario su recómputo).

7.2. Gestión de estudiantes de programación

Este ejemplo se corresponde con el introducido en el capítulo 1. El usuario declara los dominios para alumnos y asignaturas:

```
domain(alum_dt,[angela,david,joseluis,nicolas]).
domain(asignatura_dt,[introdProg, programDeclativa, prograDecAvz]).
```

La base de datos extensional tiene información de la identificación de los estudiantes, asignaturas y calificaciones.

```
 type(alum_id(alum_dt,real)).
 alum_id(angela,1.0).
 alum_id(david,2.0).
 alum_id(joseluis,3.0).
 alum_id(nicolas,4.0).

type(asignatura_id(asignatura_dt,real)).

asignatura_id(introdProg,5).
asignatura_id(programDeclativa,25).
asignatura_id(prograDecAvz,50).

type(curso(real,real,real)).
curso(1.0, 5.0, 5.0).
curso(3.0, 5.0, 7.0).
```

```
curso(2.0, 5.0, 2.0).

curso(1, 25.0, 3.0).
```

La parte extensional de la base de datos que captura los prerrequisitos de matrícula. Solo puede se matricular de programación declarativa avanzada quien haya aprobado introducción a la programación y tenga aprobada o matriculada programación declarativa. Se define de manera sencilla con las siguientes cláusulas:

```
type(matricula(real,real)).
matricula(X,50.0):-
    curso(X,5.0,Y), constr(real,Y>=5.0),
    curso(X,25.0,Z).
```

Para saber *quién no se puede matricular de programación declarativa avanzada*, puede lanzarse la consulta:

```
HHn(C)> not(matricula(X,50.0)),alum_id(N,X).
Answer: (N=angela, X=1.0;N=david, X=2.0;
         N=joseluis, X=3.0;N=nicolas, X=4.0),
         X/=1.0
```

Es decir, solo Ángela puede matricularse. El resto de alumnos no pueden por no cumplir los requisitos. La consulta *cómo cambiaría la media de la clase si el alumno Jose Luis saca un* 9 *en introducción a la programación* se formula de la siguiente manera:

```
HHn(C)> curso(3.0,5.0,9.0) =>
        constr(real, Avg=avg(curso(Y,5.0,X),X)).
Answer: Avg=5.75
```

Como puede verse, los resultados son los mismos que en el ejemplo del capítulo 1.

Capítulo 8

Conclusiones y trabajo futuro

En este trabajo se ha presentando la formalización e implementación del esquema de programación lógico $HH_{\neg}(\mathcal{C})$ como lenguaje de base de datos deductivas con restricciones. Las principales aportaciones con respecto a otros sistemas de bases de datos deductivas provienen de la lógica intuicionista de las fórmulas de Harrop hereditarias. Esta lógica subyacente incorpora implicación y cuantificación universal en el cuerpo de las cláusulas, lo cual permite realizar consultas hipotéticas y generalizadas. Se han implementado resolutores específicos para el sistema porque estos conectivos no aparecen en otros resolutores existentes. Concretamente se ha implementado una versión para dominios finitos del cuantificador universal. Por otra parte, el esquema incorpora el uso de restricciones lo que permite aumentar la eficiencia y representar datos potencialmente infinitos.

Como principales resultados del trabajo se deben mencionar una precisa descripción del lenguaje de consultas basado en un marco teórico bien formalizado y un prototipo que lo implementa. Como se ha visto, el uso de la semántica de punto fijo como mecanismo operacional se ha adoptado en gran número de sistemas de bases de datos deductivas [55, 77]. En el sistema implementado, el marco es independiente del sistema de restricciones concreto. Se han implementado sistemas de restricciones para booleanos, reales y dominios finitos. Se ha trabajado para permitir una combinación limitada de dominios cuya ampliación será abordada como trabajo futuro.

En [50] se presentó el esquema de programación $HH_{\neg}(\mathcal{C})$, que se basa en una semántica de pruebas y en una semántica estratificada de punto fijo que es correcta y completa con respecto a la primera. Más tarde, se presentó [2] donde aparece el primer prototipo Prolog que implementaba este esquema, basado en la semántica de punto fijo e independiente del sistema de restricciones. En este trabajo se presenta una visión integrada de ambos trabajos, así como algunas extensiones.

Con respecto a [50], este trabajo revisa la especificación del grafo de dependencias. Se ha conseguido una nueva versión más simplificada, que además tiene en cuenta funciones de agregación.

Con respecto a [2], en este trabajo se da una descripción más detallada del sistema y de su implementación. Además, aquí se presenta también la implementación de las funciones de agregación más habituales de las bases de datos relacionales. Esto se ha

desarrollado usando las ventajas que proporcionan las nociones de estratificación y grafo de dependencias. De hecho, como se ha establecido en este trabajo, las funciones de agregación se han tratado de una manera muy similar a la negación. Se han mejorado los resolutores. Además se ha mejorado la eficiencia del sistema en las últimas versiones limitando el número de ocasiones en las que es necesario el recálculo del punto fijo.

Aportaciones

- Se ha realizado una comparación entre distintos sistemas de bases de datos deductivas, basados fundamentalmente en Datalog y muy orientada al manejo de la negación y las funciones de agregación.
- Se han revisado los sistemas de bases de datos con restricciones más importantes y sus aplicaciones.
- Se ha presentado una nueva noción de grafo de dependencias que tiene en cuenta no solo la negación sino también las funciones de agregación.
- Se han integrado al sistema la resolución de funciones de agregación delegando ese trabajo al resolutor de restricciones.

También se puede destacar que se han unificado los conceptos teóricos y la implementación del esquema basado en $HH_{\neg}(\mathcal{C})$ en un único documento científico. Finalmente otro de los puntos que se ha mejorado ha sido la interfaz de usuario. En versiones recientes del sistema, se han incorporado el inductor de comandos para las consultas, así como algunos comandos que son útiles a la hora de trabajar con el sistema, como se puede ver al principio de la sección 4.1.

Trabajo futuro

El prototipo que se presenta en este trabajo es muy cercano a la teoría y tiene valor para el desarrollo y conocimiento de dicha teoría, pero como consecuencia la eficiencia se ha visto mermada. En primer lugar, la ineficiencia viene por dada los cálculos ascendentes. En esta línea, se conocen métodos como el de transformaciones *magic sets* [8] o *tabling* [72] que podrían ser adaptados para esta implementación.

También como trabajo relacionado se está estudiando adaptar las ideas que fundamentan el *well-founded model* [79], lo cual ayudaría a relajar las condiciones impuestas de estratificación. O bien, se puede conseguir el mismo objetivo con métodos de *tabling*, como se hace en [66]. Se ha pensado combinar la tecnología de los lenguajes relacionales con el sistema para mejorar su eficiencia.

La segunda fuente de ineficiencia proviene del forzado de la implicación, que cambia dinámicamente la base de datos. Esto requiere en ocasiones hay que calcular un nuevo punto fijo local desde el principio. Se está trabajando en identificar condiciones que evitarían recómputos.

Por otro lado, hay un gran número de extensiones que se pueden incorporar al sistema como, por ejemplo, las restricciones de integridad. Con respecto a los agregados, en esta implementación se pueden aplicar estas funciones a átomos cerrados en la interpretación. A pesar de que esta limitación parece natural en el uso más habitual de funciones de agregación, se está investigado qué supondría la eliminación de dichas limitaciones.

Publicaciones en las que ha colaborado el autor

- G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Implementing a Fixpoint Semantics for a Constraint Deductive Database based on Hereditary Harrop Formulas. In *Procedings of the 11th International ACM SIGPLAN Symposium of Principles and Practice of Declarative Programing (PPDP'09)*, pages 117–128. ACM Press, 2009.

- G. Aranda-López, S. Nieva, F. Sáenz-Pérez y J. Sánchez-Hernández, Implementación de una semántica de punto fijo para un sistema de bases de datos deductivas con restricciones, IX Jornadas sobre Programación y Lenguajes, PROLE'09, ISBN: 978-84-692-4600-9, pp. 161-175 , San Sebastián, Spain, September, 2009.

- G. Aranda-López, S. Nieva, F. Sáenz-Pérez y J. Sánchez-Hernández, A Prototype Constraint Deductive Database System based on $HH_{\neg}(\mathcal{C})$, X Jornadas sobre Programación y Lenguajes, PROLE'10, Valencia, Spain, September, 2010.

Estos trabajos son el resultado que a lo largo de más de dos años ha realizado el autor dentro del Grupo de Programación Declarativa Avanzada que pertenece al Departamento de Sistemas Informáticos y Computación de la Universidad Complutense de Madrid. Aunque si bien, es cierto que las principales aportaciones han sido en la parte de la implementación del sistema, al principio de incorporarse al proyecto de investigación se centró más en al parte teórica que fundamenta el sistema. Para acercar estas nociones se realizaron demostraciones de la mayoría de los programas que aparecen junto con el sistema en la versión que se puede descargar de la web. El ejemplo de demostración de base de datos con respecto a la semántica de punto fijo que aparece en el capítulo 3 es fruto de este trabajo. Una vez comprendidas las nociones teóricas el autor se centró en la implementación del sistema. Para la primera publicación científica en la que participó, se desarrollaron ejemplos que trataban de mostrar todas las posibles formas de la sintaxis. La búsqueda de ejemplos llevó, en muchas ocasiones, a la detección y correción de fallos del sistema para lo cual se hizo necesario trabajar de manera extensa depurando el sistema en Prolog. Una vez que el conocimiento del sistema estuvo más maduro se trabajó en el código del sistema. Lo primero que se realizó fue un inductor de comandos, lo cual obligaba a entender bien como se relacionan los módulos que componen el sistema, para poder manejar correctamente la gestión de consultas. Más adelante, correspondiendo con las últimas publicaciones, trabajó en la resolución de agregados delegada a los sistemas de restricciones. En primer lugar en el resolutor de dominios finitos, lo cual tuvo más complicaciones debido a la comunicación con el resolutor subyacente de SWI-Prolog.

La extensión de las funciones de agregado al resolutor de reales resultó mucho más sencilla. Se añadió una fase de preresolución de restricciones a la interfaz genérica del resolutor que resolvió algunos problemas y mejoró la eficiencia. Finalmente se ha tratando de resolver el problema de la combinación de dominios que se deja como trabajo futuro y se ha conseguido para algunos casos sencillos.

Bibliografía

[1] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. *Foundations of deductive databases and logic programming*, pages 89–148, 1988.

[2] G. Aranda, S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Implementing a Fixpoint Semantics for a Constraint Deductive Database based on Hereditary Harrop Formulas. In *Procedings of the 11th International ACM SIGPLAN Symposium of Principles and Practice of Declarative Programing (PPDP'09)*, pages 117–128. ACM Press, 2009.

[3] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. The Deductive Database System LDL++. *TPLP*, 3(1):61–94, 2003.

[4] Isaac Balbin, Graeme S. Port, Kotagiri Ramamohanarao, and Krishnamurthy Meenakshi. Efficient bottom-up computation of queries on stratified databases. *J. Log. Program.*, 11(3&4):295–344, 1991.

[5] Isaac Balbin and Kotagiri Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *J. Log. Program.*, 4(3):259–262, 1987.

[6] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM.

[7] Marianne Baudinet, Marc Nizette, and Pierre Wolper. On the representation of infinite temporal data and queries, 1991.

[8] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3-4):255–299, 1991.

[9] François Bry, Hendrik Decker, and Rainer Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *EDBT '88: Proceedings of the International Conference on Extending Database Technology*, pages 488–505, London, UK, 1988. Springer-Verlag.

[10] Jo-Hag Byon and Peter Z. Revesz. Disco: A constraint database system with sets. In *In CONTESSA Workshop on Constraint Databases and Applications*, pages 68–83. Springer-Verlag, 1995.

[11] Ashok K. Chandra and David Harel. Horn clauses queries and generalizations. *J. Log. Program.*, 2(1):1–15, 1985.

[12] C.L. Chang. Deduce 2: Further investigation of deduction in relational data bases. In *IBM, Res.R. No.RJ2147, San Jose; ACM Computing Reviews 40,416.* ACM Computing Reviews, May 1978.

[13] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The ldl system prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2:76–90, 1990.

[14] Raghu Ramakrishnan Divesh, Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Coral—control, relations and logic. In *In Proceedings of the International Conference on Very Large Databases*, pages 238–250, 1992.

[15] M. García-Díaz and S. Nieva. Providing Declarative Semantics for HH Extended Constraint Logic Programs. In *Proceedings of the 6th ACM SIGPLAN Int. Conf. on PPDP*, pages 55 – 66, 2004.

[16] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.

[17] C. Cordell Green and Bertram Raphael. The use of theorem-proving techniques in question-answering systems. In *ACM '68: Proceedings of the 1968 23rd ACM national conference*, pages 169–181, New York, NY, USA, 1968. ACM.

[18] W. I. Grosky and R. Mehrotra. Guest editors' introduction: Image database management. *Computer*, 22(12):7–8, 1989.

[19] Stphane Grumbach, Philippe Rigaux, Le Chesnay, and Luc Segoufin. Spatio-temporal data handling with constraints, 1998.

[20] Stphane Grumbach, Philippe Rigaux, Universit'a Di Roma Tre, Le Chesnay, and Luc Segoufin. The dedale system for complex spatial queries, 1998.

[21] O. Günther and J. Bilmes. Tree-based access methods for spatial databases: Implementation and performance evaluation. *IEEE Trans. on Knowl. and Data Eng.*, 3(3):342–356, 1991.

[22] M. R. Hansen, B. S. Hansen, P. Lucas, and P. van Emde Boas. Integrating relational databases and constraint languages. *Comput. Lang.*, 14(2):63–82, 1989.

[23] W. W. Hargrove, R. H. Gardner, M. G. Turner, W. H. Romme, and D. G. Despain. Simulating fire patterns in heterogeneous landscapes, 2000.

[24] Lawrence J. Henschen and Shamim A. Naqvi. On compiling queries in recursive first-order databases. *J. ACM*, 31(1):47–85, 1984.

[25] Christian Holzbaur. Realization of forward checking in logic programming through extended unification. Report TR-90-11, Oesterreichisches Forschungsinstitut fuer. *Artificial Intelligence*, 1990.

[26] H. V. Jagadish. A retrieval technique for similar shapes. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 208–217, New York, NY, USA, 1991. ACM.

[27] Matthias Jarke, Manfred A. Jeusfeld, and Christoph Quix (Eds.). ConceptBase V7.1 User Manual. Technical report, RWTH Aachen, April 2008.

[28] Manfred Jeusfeld and Matthias Jarke. From relational to object-oriented integrity simplification, 1991.

[29] Manfred Jeusfeld and Martin Staudt. Query optimization in deductive object bases, 1993.

[30] F. Kabanza, J m. Stevenne, and P. Wolper. Handling infinite temporal data. In *Journal of Computer and System Sciences*, pages 392–403, 1990.

[31] Par C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint Query Languages. In *Symposium on Principles of Database Systems*, pages 299–313, 1990.

[32] Pradip Kanjamala, Peter Z. Revesz, and Yonghui Wang. Mlpq/gis: A gis using linear constraint databases. In *Proc. Ninth International Conference on Management of Data*, pages 389–393. McGraw Hill, 1998.

[33] Charles Kellogg and Larry Travis. Reasoning with data in a deductively augmented data management system. In *Advances in Data Base Theory*, pages 261–295, 1979.

[34] David B. Kemp, Kotagiri Ramamohanarao, Isaac Balbin, and Krishnamurthy Meenakshi. Propagating constraints in recusive deduction databases. In *NACLP*, pages 981–998, 1989.

[35] G. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 2000.

[36] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In Chen Li, editor, *PODS*, pages 1–12. ACM, 2005.

[37] J. Leach, S. Nieva, and M. Rodríguez-Artalejo. Constraint Logic Programming with Hereditary Harrop Formulas. *TPLP*, 1(4):409–445, 2001.

[38] Alexandre Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases. *New Generation Comput.*, 12(2):131–160, 1994.

[39] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[40] Michael J. Maher and Raghu Ramakrishnan. D'ej'a vu in fixpoints of logic programs. In *in Proceedings of the North American Conference on Logic Programming*, pages 963–980. The MIT Press, 1989.

[41] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[42] Jack Minker. Perspectives in deductive databases (abstract). In *PODS*, page 135, 1987.

[43] Jack Minker. Logic and databases: A 20 year retrospective. In *Logic in Databases*, pages 3–57, 1996.

[44] Jack Minker and Jean-Marie Nicolas. On recursive axioms in deductive databases. *Inf. Syst.*, 8(1):1–13, 1983.

[45] Katherine A. Morris, Jeffrey F. Naughton, Yatin P. Saraiya, Jeffrey D. Ullman, and Allen Van Gelder. Yawn! (yet another window on nail!). *IEEE Data Eng. Bull.*, 10(4):28–43, 1987.

[46] S. Naqvi and S. Tsur. *A logical language for data and knowledge bases*. Computer Science Press, Inc., New York, NY, USA, 1989.

[47] John C. Nash. The (dantzig) simplex method for linear programming. *Computing in Science and Engg.*, 2(1):29–31, 2000.

[48] Jeffrey F. Naughton and Raghu Ramakrishnan. How to forget the past without repeating it. *J. ACM*, 41(6):1151–1177, 1994.

[49] S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Towards a constraint deductive database language based on hereditary harrop formulas. In P. Lucio and F. Orejas, editors, *Sextas Jornadas de Programación y Lenguajes, PROLE*, pages 171–182, 2006.

[50] S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Formalizing a Constraint Deductive Database Language based on Hereditary Harrop Formulas with Negation. In *FLOPS'08, Proceedings*, volume 4989 of *LNCS*, pages 289–304, Ise, Japan, 2008. Springer-Verlag.

[51] R. Ramakrishnan and J.D. Ullman. A survey of research on Deductive Databases. *The Journal of Logic Programming*, 23(2):125–149, 1993.

[52] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Journal of Logic Programming*, pages 140–159, 1988.

[53] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *In Joint Intl. Conference and Symposium on Logic Programming*, pages 273–287, 1992.

[54] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the coral deductive database system, 1993.

[55] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The coral deductive system. *The VLDB Journal*, 3:161–210, 1994.

[56] Sudarshan Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *In Proceedings of the International Conference on Very Large Databases*, pages 501–511, 1991.

[57] G. Ramalingam and Eelco Visser, editors. *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*. ACM, 2007.

[58] Peter Z. Revesz. Refining restriction enzyme genome maps. *Constraints*, 2(3/4):361–375, 1997.

[59] P.Z. Revesz and Yiming Li. Mlpq: a linear constraint database system with aggregate operators. *Database Engineering and Applications Symposium, International*, 0:132, 1997.

[60] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[61] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. *J. ACM*, 41(6):1216–1266, 1994.

[62] F. Sáenz-Pérez. Datalog Educational System. User's Manual version 2.0. Technical report, Faculty of Computer Science, UCM, august 2010. Available from http://des.sourceforge.net/.

[63] Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. In *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.

[64] A. Salomon. *Implementation of a Database System with Boolean Algebra Constraints*. PhD thesis, University of Nebraska, 1998.

[65] Jami J. Shah and Martti Mantyla. *Parametric and Feature Based CAD/Cam: Concepts, Techniques, and Applications*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[66] Y. Shen, L. Yuan, and J. You. Slt-resolution for the well-founded semantics. *Journal of Automated Reasoning*, 28:53–97, 2002.

[67] J.C Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Kaufmann, Los Altos, CA, 1988.

[68] C. Shih and S.W. Dietrich. Extension Table Evaluation of Datalog Programs with Negation. In *Proceedings of the IEEE International Phoenix Conference on Computers and Communications*, volume AZ, pages 792–798. Scottsdale, March 1991.

[69] Divesh Srivastava, Raghu Ramakrishnan, Praveen Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *In Proceedings of the International Conference on Very Large Data Bases*, pages 158–170. Morgan Kaufmann Publishers, Inc, 1993.

[70] S. Sudarshan and Raghu Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms: extended abstract. In *ILPS '93: Proceedings of the 1993 international symposium on Logic programming*, pages 557–574, Cambridge, MA, USA, 1993. MIT Press.

[71] S. Sudarshan, Divesh Srivastava, Raghu Ramakrishnan, and Jeffrey F. Naughton. Space optimization in the bottom-up evaluation of logic programs. In *in: Proc. SIGMOD*, pages 5370–6, 1990.

[72] H. Tamaki and T. Sato. Old resolution with tabulation. In *Proceedings on Third international conference on logic programming*, pages 84–98, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

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

[74] Shalom Tsur and Carlo Zaniolo. Ldl: A logic-based data language. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 33–41. Morgan Kaufmann, 1986.

[75] J.D. Ullman. *Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1995.

[76] Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Syst.*, 10(3):289–321, 1985.

[77] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, and Peter J. Stuckey. Design overview of the aditi deductive database system. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 240–247, Washington, DC, USA, 1991. IEEE Computer Society.

[78] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[79] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991.

[80] David Scott Warren. The xwam: A machine that integrates prolog and deductive databases. In *Technical Report*, 1989.

[81] Hassler Whitney. *Geometric integration theory.* Princeton University Press, Princeton, N. J., 1957.

[82] J. Wielemaker. SWI-Prolog. User's Manual version 5.6.64, 2009. Available from http://www.swi-prolog.org/.

[83] Carlo Zaniolo. Key constraints and monotonic aggregates in deductive databases. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 109–134, London, UK, 2002. Springer-Verlag.

[84] Carlo Zaniolo, Natraj Arni, and Kayliang Ong. Negation and aggregates in recursive rules: the ldl++ approach, 1993.