However, there are still some limitations due to the use of current FOL theorem provers. For instance, we only obtain boolean answers to existential queries, instead of providing the values that satisfy the query. Another restriction is the lack of goal-oriented proof-search techniques, which prevents the use of FOL theorem provers for reasoning on large ontologies. We try to overcome this last problem by pre-processing queries in order to remove non-relevant axioms for each particular query.

Our most immediate future aim is to continue expanding the translation of OWL DL into FOL in order to cover OWL DL syntax in its entirety. For this purpose, first we have to translate OWL DL properties into FOL. In our opinion, the translation of properties is very similar to the one of instances. Hence, we expect to fulfill this task soon.

Once the treatment of OWL DL ontologies is finished, we also plan to reason with more expressive ontologies, as discussed in [6]. Anyway, the problem of reasoning on ontologies like SUMO [8] using FOL theorem provers is still open.

References

[1] Baader, F., D. Calvanese, D. L. McGuinness, D. Nardi and P. F. Patel-Schneider, editors, "The Description Logic Handbook. Theory, Implementation, and Applications," Cambridge University Press, 2003.

[2] Bechhofer, S. and I. Horrocks, *Hoolet: An OWL reasoner with support for rules* (2004).
URL http://owl.man.ac.uk/hoolet/

[3] Bechhofer, S., F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider and L. A. Stein, *OWL web ontology language reference*, *W3C recommendation* (2004).
URL http://www.w3.org/TR/owl-ref

[4] Borgida, A., *On the relative expressiveness of description logics and predicate logics*, Artificial Intelligence **82** (1996), pp. 353–367.

[5] Horrocks, I. and P. F. Patel-Schneider, *Reducing OWL entailment to description logic satisfiability*, Journal of Web Semantics **1** (2004), pp. 345–357.

[6] Horrocks, I. and A. Voronkov, *Reasoning support for expressive ontology languages using a theorem prover*, in: J. Dix and S. J. Hegner, editors, *Foundations of Information and Knowledge Systems (FoIKS 2006)*, Lecture Notes in Computer Science **3861** (2006), pp. 201–218.

[7] Knublauch, H., R. W. Fergerson, N. F. Noy and M. A. Musen, *The Protégé OWL plugin: An open development environment for semantic web applications*, in: S. A. McIlraith, D. Plexousakis and F. van Harmelen, editors, *Proceedings of the International Semantic Web Conference (ISWC 2004)*, Lecture Notes in Computer Science **3298** (2004), pp. 229–243.

[8] Niles, I. and A. Pease, *Towards a standard upper ontology*, in: *Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS '01)* (2001), pp. 2–9.

[9] Pelletier, F. J., G. Sutcliffe and C. B. Suttner, *The development of CASC*, AI Communications **15** (2002), pp. 79–90.

[10] Riazanov, A. and A. Voronkov, *The design and implementation of VAMPIRE*, AI Communications **15** (2002), pp. 91–110.

[11] Schulz, S., *E – a brainiac theorem prover*, Journal of AI Communications **15** (2002), pp. 111–126.

[12] Sirin, E., B. Parsia, B. C. Grau, A. Kalyanpur and Y. Katz, *Pellet: A practical OWL-DL reasoner*, Journal of Web Semantics **5** (2007), pp. 51–53.

[13] Sutcliffe, G. and C. B. Suttner, *The state of CASC*, AI Communications **19** (2006), pp. 35–48.

[14] Tsarkov, D. and I. Horrocks, *FaCT++ description logic reasoner: System description*, in: U. Furbach and N. Shankar, editors, *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2006)*, Lecture Notes in Computer Science **4130** (2006), pp. 292–297.

[15] Tsarkov, D., A. Riazanov, S. Bechhofer and I. Horrocks, *Using Vampire to reason with OWL*, in: S. A. McIlraith, D. Plexousakis and F. van Harmelen, editors, *Proceedings of the International Semantic Web Conference (ISWC 2004)*, Lecture Notes in Computer Science **3298** (2004), pp. 471–485.

[16] Zhang, Z. and J. A. Miller, *Ontology query languages for the semantic web. A performance evaluation*, Technical Report UGA-CS-LSDIS-TR-05-011, Department of Computer Science, University of Georgia, Athens (2005).

Implementación de una semántica de punto fijo para un sistema de bases de datos deductivas con restricciones

Gabriel Aranda-López[†], Susana Nieva[†], Fernando Sáenz-Pérez[‡] y Jaime Sánchez-Hernández[†] [1]

[†] *Dept. Sistemas Informáticos y Computación and* [‡] *Dept. Ingeniería del Software e Inteligencia Artificial. Universidad Complutense de Madrid, Spain*
{nieva,fernan,jaime}@sip.ucm.es, garanda@fdi.ucm.es

Abstract

El objeto del trabajo es mostrar una implementación concreta de un sistema de bases de datos deductivas basado en el esquema $HH_{\neg}(C)$ (Hereditary Harrop Formulas with Negation and Constraints) siguiendo una semántica de punto fijo definida en un trabajo anterior. Para este esquema hemos desarrollado una implementación en Prolog independiente del sistema de restricciones concreto, lo que hace que sirva de base para cualquier instancia. Proponemos y describimos varios sistemas de restricciones específicos: números reales, enteros, booleanos y tipos enumerados definidos por el usuario. Añadimos tipos a la bases de datos y, por tanto, las relaciones ahora están tipadas (como las tablas en las bases de datos relacionales) y cada restricción está asociada a un sistema de restricciones concreto. Se explican los predicados que sirven para calcular el punto fijo que da significado a una base de datos. En particular, mostramos la implementación de una relación semántica de forzado y destacamos cómo se han resuelto las dificultades inherentes a este sistema, que permite consultas hipotéticas, lo que provoca que la base de datos crezca dinámicamente.

Keywords: Bases de datos deductivas, Restricciones, Hereditary Harrop Formulas, Semántica de punto fijo

1 Introducción

Las bases de datos deductivas (BDD) y sus lenguajes de consulta han sido objeto recientemente de una gran atención en muchas áreas como, por ejemplo, las ontologías [5], la web semántica [4], las redes sociales [13] y los lenguajes de directivas [3]. El alto nivel de expresividad de estos lenguajes ha sido, por tanto, reconocido como una herramienta útil para la gestión de información basada en conocimiento. En particular, Datalog (y sus extensiones), del que se pueden encontrar múltiples referencias, ha adquirido un renovado interés en estos campos.

Los actuales sistemas BDD (como XSB [15], bddbddb [7], LDL++ [2], DES [11], ConceptBase [1], .QL [12] y DLV [9]) carecen de mecanismos que aportamos en el

[1] Este trabajo ha sido parcialmente financiado por los proyectos TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407 y UCM-BSCH-GR58/08-910502.

IX Jornadas sobre Programación y Lenguajes
I Taller de Programación Funcional
P. Lucio, G. Moreno y R. Peña, Eds.
San Sebastián, 8-11 de septiembre de 2009

esquema $HH_\neg(C)$ (Hereditary Harrop formulas with Negation and Constraints) [11], útiles para sistemas basados en conocimiento en los que son necesarias consultas más expresivas, y sirve como fundamento a un sistema BDD que combina cuantificadores, restricciones y consultas hipotéticas, extendiendo $HH(C)$ [8] con la negación.

En nuestro sistema, una base de datos es un programa lógico: un conjunto de hechos que definen la parte extensional de la base de datos y un conjunto de cláusulas, que definen la parte intensional. La evaluación de una consulta con respecto a la base de datos deductiva se puede ver como el cálculo de un objetivo de un programa lógico. Dado que el sistema de restricciones es paramétrico es posible usar instancias diferentes (como los números reales y los dominios finitos).

Para mostrar la expresividad de nuestro lenguaje explicamos a continuación un ejemplo en una instancia que permite dominios finitos y reales a la vez.

Ejemplo 1.1 Considérese la siguiente base de datos extensional para un banco[2].

```
% client(Name, Balance, Salary)
client(smith,2000,1200).
client(brown,1000,1500).
client(mcandrew,5300,3000).
```

```
% pastDue(Name, Amount)
pastDue(smith,3000).
pastDue(mcandrew,100).
% mortgageQuote(Name, Quote)
mortgageQuote(brown,400).
mortgageQuote(mcandrew,100).
```

Además podemos definir las siguientes vistas:

```
% debtor(Name)
debtor(N) :-
  client(N,B,S),
  pastDue(N,A), A>B.
```

```
% interestRate(Name, Rate)
interestRate(N,2) :- client(N,B,S), B<1200.
interestRate(N,5) :- client(N,B,S), B>=1200.
% hasMortgage(Name)
hasMortgage(N):- ex(Q,mortgageQuote(N,Q)).
```

La siguiente relación indica que puede concederse una hipoteca a un cliente, en caso de que no tenga otra ya concedida, si la suma de la cuota actual y la nueva cuota no supera el límite del 40% del sueldo del cliente y en ambos casos siempre que no haya deudas pendientes de pago.

```
% newMortgage(Name, Quote)
newMortgage(N,Q) :- client(N,B,S), not(hasMortgage(N)),
                    not(debtor(N)), Q<=0.4*S.
newMortgage(N,Q) :- client(N,B,S), not(debtor(N)),
                    mortgageQuote(N,Q2), Q+Q2<=0.4*S.
% getMortgage(Name)
getMortgage(N):- ex(Q,newMortgage(N,Q)).
```

Para casos en que no sea posible conceder una hipoteca, podemos conceder un pequeño crédito personal puntual, según la siguiente relación:

```
% personalCredit(Name, Amount)
personalCredit(N,A) :- not(getMortgage(N)), A>=1500;
                       getMortgage(N), A<1500.
```

Podemos formular las siguientes consultas a la base de datos. Si queremos saber si hay morosos en nuestra base de datos con facturas impagadas de una magnitud

[2] Usaremos notación Prolog y además `not`, `=>`, `ex(X,G)` para representar $\exists X\ G$, y `fa(X,G)` para $\forall X\ G$.

superior a 1000, se puede formular la consulta:

`ex(N, ex(A, (debtor(N), pastDue(N,A),(A>1000)))).`

cuya respuesta es `true`[3]. Otra posible consulta es: suponiendo que un cliente cualquiera tuviera un saldo superior a 2000, ¿qué tipo de interés le corresponde?

`fa(N, ex(S, ex(B, (client(N,B,S)=> B>2000 => interestRate(N,R))))).`

La respuesta a esta pregunta es la restricción `R=5`. Estamos usando la implicación para formular consultas hipotéticas en las que podemos suponer tanto hechos como restricciones. La siguiente consulta representa la selección de clientes a los que sería posible dar una hipoteca con una cuota superior a 400 pero *no* un crédito personal puntual: `newMortgage(N,400), not(personalCredit(N,A))`, y la respuesta es la restricción `(N=mcandrew, A>=1500)`. □

En este artículo se presenta una implementación en Prolog de la semántica de punto fijo presentada en [11] independiente del sistema de restricciones concreto. Este núcleo se describe en la sección 4. Además hemos implementado distintos sistemas de restricciones que darán lugar a diferentes instancias de $HH_\neg(C)$. En la sección 3 describimos el sistema de tipos, los sistemas de restricciones, sus resolutores y su implementación aprovechando los resolutores de restricciones de SWI-Prolog [16].

La semántica de una base de datos se calcula como un conjunto de pares (A, C), donde A es un átomo y C es una restricción, que pueden ser deducidos de las definiciones extensionales e intensionales de la base de datos. A se puede entender como una instancia de una relación n-aria, donde los argumentos están restringidos por C. Estos pares se calculan por estratos, clasificando los predicados en estratos según una nueva forma de estratificación guiada por las negaciones y las implicaciones que aparecen en el cuerpo de las cláusulas y que se describe en la sección 5. Cada estrato se debe saturar antes que cualquier otro superior. Sin embargo, cuando aparecen las implicaciones, las derivaciones tienen lugar en el contexto de una base de datos aumentada con la hipótesis planteada en la implicación. Por lo tanto, se debe reiniciar el cálculo del punto fijo dado que se pueden añadir nuevos pares en estratos inferiores. Estos cálculos anidados añaden un nuevo nivel de complejidad a los habituales cálculos *bottom-up* de las BDD sin implicaciones.

Otra fuente de complejidad procede también de las implicaciones, dado que las variables de $D \rightarrow G$ pueden aparecer tanto en D como en G cuando una base de datos Δ se aumente con la cláusula local D. Estas variables se deben distinguir de las otras instancias de las mismas variables de Δ. Con este fin recurrimos a las variables con atributos (Prolog) para su identificación.

Finalmente, con objeto de encontrar una estratificación que asegure finitud en los cálculos, describimos un nuevo grafo de dependencias usando una definición mutuamente recursiva entre las dependencias introducidas por los objetivos y las cláusulas.

[3] Nótese que se aplican cuantificadores existenciales para `N` y `A`, indicando que no hay condiciones explícitas sobre estas variables. En caso contrario, la respuesta sería una restricción sobre ellas.

2 Preliminares

En esta sección se resumen los fundamentos, presentados en [11], en los que se basa la implementación.

2.1 *Sintaxis*

Se considera un conjunto de *símbolos de predicados definidos* para construir átomos que representan relaciones en la base de datos, además un conjunto de ***símbolos de predicado no definidos*** para construir restricciones; estos últimos incluyen el símbolo de predicado de igualdad $\approx$. También se dispone de un conjunto de constantes y símbolos de operación, y de un conjunto de variables con los que se construyen términos. Usaremos la notación A para representar átomos, C para restricciones y t para términos.

Las restricciones que vamos a considerar 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 y $\vdash_\mathcal{C}$ es la *relación de deducibilidad*. $\Gamma \vdash_\mathcal{C} C$ denota que la restricción C se deduce del sistema de restricciones $\mathcal{C}$ partiendo de las restricciones del conjunto Γ. A $\mathcal{C}$ se le imponen unas condiciones mínimales para que sea un sistema de restricciones (véase [8] para más detalles). En particular $\mathcal{C}$ debe contener $\top$ (cierto) y $\bot$ (falso), las conectivas $\wedge, \neg$, y el cuantificador existencial $\exists$; el sistema de restricciones se encarga de 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 la clausura existencial de C. C y C'' son $\mathcal{C}$-equivalentes si $C \vdash_\mathcal{C} C''$ 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 (o consultas) G. 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. 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. La *elaboración* de un programa Δ es el conjunto $elab(\Delta) = \bigcup_{D \in \Delta} elab(D)$, donde $elab(D)$ se define como:

$elab(A) = \{\top \Rightarrow A\}$, $elab(D_1 \wedge D_2) = elab(D_1) \cup elab(D_2)$,

$elab(G \Rightarrow A) = \{G \Rightarrow A\}$, $elab(\forall x D) = \{\forall x D' \mid D' \in elab(D)\}$.

2.2 *Estratificación*

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 es que cuando $\neg A$ se va a probar, el estrato de A se ha saturado previamente (todas las respuestas para A están disponibles) y $\neg A$ se puede calcular correctamente. Una estratificación se basa en la construcción de un grafo de dependencias para un conjunto de fórmulas [17].

Para construir este grafo se analizan las fórmulas en cuestión. Los nodos son los símbolos de predicado del conjunto. Una implicación de la forma $F_1 \Rightarrow F_2$ 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 a la izquierda 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 definidos de predicado, no generan dependencias.

Definición 2.1 *Dado un conjunto de fórmulas Φ, su grafo de dependencias correspondiente DG_Φ, y dos predicados p y q, diremos que q* depende *de p si hay un camino de p a q en DG_Φ y que q* depende negativamente *de p si hay un camino de p a q en DG_Φ con al menos un arco etiquetado negativamente.*

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. Φ es* estratificable *si existe una estratificación para él.*

Se dice que el estrato de una fórmula F, denotado $str(F)$, es el máximo $s(p)$, donde p recorre los símbolos de predicado definidos que aparecen en F.

2.3 *Interpretaciones estratificadas 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, y $S\mathcal{L}_\mathcal{C}$ el conjunto de restricciones $\mathcal{C}$-satisfactibles módulo $\mathcal{C}$-equivalencia. Las interpretaciones se clasifican en estratos y cada una da información hasta su estrato.

Definición 2.2 *Sea $i \geq 1$, una* interpretación *I sobre el estrato i es una función $I : \mathcal{W} \to \mathcal{P}(At \times S\mathcal{L}_\mathcal{C})$, tal que para cualquier $\Delta \in \mathcal{W}$ y cualquier $j > i$, $[I(\Delta)]_j = \emptyset$, donde $[I(\Delta)]_i = \{(A, C) \in I(\Delta) \mid str(A) = i\}$. Se denota como $\mathcal{I}_i$ el* conjunto de interpretaciones sobre *i.*

Definición 2.3 *Sea $i \geq 1$ y $I_1, I_2 \in \mathcal{I}_i$. 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* $1 \leq j < i$.
- $[I_1(\Delta)]_i \subseteq [I_2(\Delta)]_i$.

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 cota superior mínima $\bigsqcup_{n \geq 0} I_n$, 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.

Definición 2.4 *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 con las reglas que se muestran a continuación. Cuando $I, \Delta \Vdash (G, C)$, se dice que (G, C) es* forzado *por I, Δ.*

- $I, \Delta \Vdash (C', C) \iff C \vdash_\mathcal{C} C'$.
- $I, \Delta \Vdash (A, C) \iff (A, C) \in I(\Delta)$.

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

2.4 Semántica de punto fijo

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

Definición 2.5 *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* $(A, C) \in At \times S\mathcal{L}_C$, *se tiene* $(A, C) \in T_i(I)(\Delta)$ *cuando:*

- $(A, C) \in I(\Delta)|_j$ *para algunos* $j < i$, *o*
- $s(A) = i$ *y hay una variante* $\forall \overline{x}(G \Rightarrow A')$ *de una cláusula en* $elab(\Delta)$, *tal que las variables* $\overline{x}$ *no aparecen libres en* A, *y* $I, \Delta \models (\exists \overline{x}(A \approx A' \wedge G), C)$.

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

3 Implementación de la resolución de restricciones

Esta sección se centra en la implementación de la resolución de restricciones en los siguientes sistemas de restricciones específicos: números reales, enteros, booleanos y tipos enumerados definidos por el usuario. En primer lugar se introduce el sistema de tipos necesario para determinar el resolutor al que se debe enviar cada restricción. Después se describen los sistemas de restricciones, incluyendo sus valores, funciones predefinidas y operadores. Finalmente, mostramos la implementación de los resolutores, que se apoya en los resolutores de restricciones del sistema Prolog subyacente.

3.1 Tipos

El sistema incorpora como predefinidos los tipos de datos `bool` (con elementos `true` y `false`) y `real` (un tipo infinito cuyo rango numérico depende del sistema Prolog subyacente). Además admite la definición de nuevos tipos de datos de dominio finito.

Una declaración de tipo de datos se escribe como `domain(tipo_de_datos, [constante_1, ..., constante_n])`. Se permiten intervalos de enteros en las declaraciones de tipos de datos, como en `domain(meses, 1..12)`. Una declaración de tipo para un predicado n-ario se escribe `type(predicado(tipo_1, ..., tipo_n))`. Por ejemplo, `type(cliente(td_cliente, real))` es una declaración de tipo donde `td_cliente` se puede definir como `domain(td_cliente, [smith, brown, mcandrew])`.

Hemos implementado un comprobador de tipos para programas $HH_\neg(\mathcal{C})$ capaz de detectar tanto inconsistencias como ausencias en sus declaraciones. Se anotan los tipos localmente en cada regla, que consiste en el almacenamiento del tipo de cada variable en un atributo (cfr. variables atribuidas [6]). Los tipos de las variables son conocidos en el contexto de una regla porque: a) un átomo proporciona su tipo (i.e., proporcionado por la declaración de tipos de su predicado correspondiente), o b) una restricción `constr(Dom, C)` proporciona su tipo, donde `constr` es la sintaxis que hemos escogido para denotar una restricción `C` sobre el dominio `Dom`. Se genera una excepción de conflicto de tipos cuando se intenta asignar diferentes tipos a la misma variable, y de ausencia de tipo cuando no se le puede asignar ninguno.

3.2 Sistemas de restricciones

Como se introdujo, un sistema de restricciones proporciona un lenguaje para expresar restricciones y una relación de decidibilidad para asegurar la satisfactibilidad de restricciones (esta relación se trata en la siguiente subsección). Nuestro sistema de restricciones incluye una sintaxis específica para los valores, símbolos, conectivas y cuantificadores requeridos "`true`", "`false`", "`=`", "`,`", "`not`" y "`ex(X,C)`" que representan respectivamente a $\top$, $\perp$, $\approx$, $\wedge$, $\neg$ y $\exists$X C. Además, también se incluye "`;`" para $\vee$ y "`/=`" para la negación de $\approx$.

Proponemos tres sistemas de restricciones para $HH_\neg(\mathcal{C})$: booleanos, reales, y dominios finitos. El primero contiene los componentes mínimos requeridos y el cuantificador universal `fa(X,C)`. El sistema de restricciones sobre reales incluye el tipo `real` (conjunto infinito de valores numéricos reales) y operadores de restricciones reales (`+`, `-`, `*`, ...) además de las funciones (`abs`, `sin`, `exp`, `min`, ...).

Los dominios finitos representan a una familia de sistemas de restricciones específicos sobre conjuntos enumerables, que incluyen tanto tipos enumerados como números enteros (finitos). Esta familia incluye también los operadores de comparación (`>`, `>=`, ...), restricciones de cuantificación universal (`fa(X,C)`, como antes) y la restricción de dominio `X in Range`, donde `Range` es un subconjunto de valores de datos construidos con `V1..V2`, que denota un conjunto de valores en el intervalo cerrado entre `V1` y `V2`, y con `R1\/R2`, que denota una unión de rangos. Un dominio finito puede también incluir operadores (`+`, `-`, ...) y funciones de restricciones (`abs`, `min`, ...). Nótese que las funciones primitivas relevantes para cada sistema deben ser coherenes con su semántica pretendida (`+` puede no ser relevante para booleanos, aunque sea posible usarlo).

Aunque en ocasiones usamos los mismos símbolos para construir restricciones en diferentes sistemas, como por ejemplo `constr(real, X>Y)` y `constr(meses, X>Y)`, su significado se distingue por el tipo asociado.

Para la implementación de la relación de decidibilidad proporcionamos un resolutor de restricciones con una interfaz genérica `solve(C,SC)` para $C \vdash_C SC$, que resuelve una restricción C, comprueba su satisfactibiliad y produce una forma *resuelta* SC. Una forma resuelta es una forma simplificada y más legible que la de la restricción original. Las formas resueltas son disyunciones de restricciones simples, donde una restricción simple nunca incluye disyunciones, cuantificadores ni negaciones. Además, proporcionamos también la interfaz `solve(Dom,C,SC)`, útil cuando se conoce el dominio `Dom` y, por tanto, C se puede enviar directamente a su resolutor correspondiente.

Los resolutores subyacentes de SWI-Prolog nos sirven de base para la implementación de los sistemas de restricciones de dominios finitos, booleanos y reales. Para ciertas restricciones podemos asociar la resolución de restricciones entre nuestro sistema de restricciones y SWI-Prolog porque realizamos una correspondencia entre los valores de los tipos enumerados y los números enteros antes de la resolución, que permite recuperarlos después. Por otro lado, hay restricciones que el resolutor subyacente no puede manejar (cuantificadores y disyunciones) que se manejan explícitamente como se muestra más adelante. Dado que SWI-Prolog no proporciona un resolutor de booleanos, usamos el resolutor de dominios finitos y definimos el sistema de restricciones predefinido `bool`, gestionado como cualquier otro sistema de restricciones enumerado.

Para los dos primeros resolutores disponemos del predicado `solveFD(+Dominio, +Restricción, -RestricciónResuelta)`, que resuelve la restricción de entrada si es satisfactible y devuelve una forma resuelta bajo el dominio de tipos dado. La restricción de entrada debe estar construida como un término Prolog con las funciones, operadores y valores de datos enunciados en la sección 3.2. El siguiente fragmento de código implementa el comportamiento descrito:

```
(00) solveFD(DN,C,SC) :-
(01)   set_current_domain(DN),             % Una etiqueta que almacena el dominio actual
(02)   copy_term(C,FC),                    % Las variables de entrada no se modifican
(03)   get_vars(C,Vars),                   % Las variables de entrada se guardan para
(04)   get_vars(FC,FVars),                 %  asociarlas a las nuevas variables resueltas
(05)   swap_qvars_by_fvars(FC,QFC),        % Reemplaza las variables cuantificadas por nuevas
(06)   constrain_domains(QFC,DN),          % Se restringen las variables a su dominio
(07)   domain_to_int(QFC,DN,IC),           % Correspondencia entre dominio enumerado y enteros
(08)   bagof((FVars,Cs,Sat),               % (VariablesNuevas,Restricciones,Satisfactible)
(09)     (solveFD_ctr(IC,true),            % Resolucion de restricciones
(10)      satisfiable(IC,Sat),             % Comprueba la satisfactibilidad
(11)      project_ctrs(FVars,Vars,Cs)      % Restricciones proyectadas sobre variables de entrada
(12)       ), LFVarsCsS), !                % Lista (VariablesNuevas,Restricciones,Satisfactible)
(13)   filter_ctr_list(LFVarsCsS,LICs),    % Se filtran las restricciones resueltas
(14)   simplify_disj_list(LICs,SLICs),     % Simplificacion de la lista disyuntiva
(15)   disj_list_to_ctr(SLICs,ISC),        % Convierte la lista en una restriccion
(16)   int_to_domain(ISC,DN,SC).           % Traduce el dominio de enteros al enumerado
```

Nótese que en la línea `(05)` se reemplazan las variables cuantificadas por variables nuevas con objeto de evitar colisiones de nombres. La línea `(07)` realiza la correspondencia entre los valores del dominio de datos con los números enteros, mientras que la línea `(16)` reemplaza los valores enteros resueltos por los valores correspondientes del dominio de datos. El núcleo de la resolución de restricciones se encuentra entre las líneas `(09)-(11)` donde, en primer lugar, se intenta resolver la restricción (véase el siguiente párrafo en el que se describe este predicado). En segundo lugar, se comprueba su satisfactibilidad tratando de encontrar una solución

concreta mediante etiquetado. Finalmente, se proyectan las restricciones almacenadas en el almacén de restricciones de SWI-Prolog a las variables relevantes (i.e., las que aparecen en la restricción de entrada, además de las nuevas que hayan sido producidas por el resolutor). Las líneas `(13)-(15)` son necesarias simplemente para adecuar las estructuras de datos. La línea `(17)` corresponde al fallo en la resolución, y se devuelve un valor `false` que indica insatisfactibilidad.

A continuación se describe el predicado `solveFD_ctr(+C,-B)` que recibe una restricción y devuelve si es satisfactible o no. El primer caso corresponde a una restricción soportada por el resolutor de SWI-Prolog (donde `#>` es el operador de comparación de restricciones de este resolutor):

```
solveFD_ctr(X>Y,true) :- !, X#>Y.
```

Es necesario realizar una gestión específica de la negación como se muestra a continuación, porque el resolutor subyacente no puede manejarla directamente en presencia de restricciones no soportadas.

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

El predicado `complement(+Restricción,-RestricciónComplementada)` calcula la restricción complementada usando los axiomas de la lgica de primer orden.

La disyunción es un ejemplo de restricción no soportada, que se calcula recopilando todas las soluciones (cfr. línea `(08)`):

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

Finalmente se describen los cuantificadores. En primer lugar, el cuantificador existencial se implementa como sigue, donde la penúltima línea trata de encontrar valores concretos para las variables tales que satisfagan FC (i.e., resultado `true`):

```
solveFD_ctr(ex(X,C),B) :- !,
  swap(X,_FX,C,FC), % Remplaza X en C por una variable nueva _FX
  get_current_domain(DN), constrain_domains(FC,DN),
  (solveFD_ctr(FC,true), % Resuelve
   satisfiable(FC,true), % Comprueba satisfactibilidad
   B=true ; B=false).
```

El cuantificador universal se resuelve mediante la imposición de una restricción conjuntiva C sobre todos los valores de X en el dominio de resolución (cfr. última línea):

```
solveFD_ctr(fa(X,C),B) :- !,
  get_current_domain(Domain), domain_bounds(Domain,L,U),
  (solve_forall(X,C,L,U) -> B=true ; B=false).
```

donde `solve_forall(+Var,+Restricción,+Inferior,+Superior)` impone la conjunción de `Restricción` para todos los valores de `Var` comprendidos entre los límites `Inferior` y `Superior`, como se muestra a continuación (el predicado `swap` reemplaza las apariciones de la variable X por el valor L en la restricción C dando como resultado LC).

```
solve_forall(_,_,L,U) :- L>U, !.
solve_forall(X,C,L,U) :- swap(X,L,C,LC), solveFD_ctr(LC,true), L1 is L+1,
  solve_forall(X,C,L1,U).
```

La implementación del resolutor de reales es parecida pero más sencilla porque no hay cuantificadores universales, ni valores del dominio de datos que asociar.

4 Implementación de la semántica de punto fijo

4.1 Punto fijo por estratos

Asumimos una base de datos estratificable Delta y la correspondiente partición por estratos $st_1, \dots, st_k$ de sus símbolos de predicado (el algoritmo de estratificación se verá en la Sección 5). Una cláusula de la forma `A :- G` se interpreta como $\forall X_1, \dots, X_n$(G ⇒ A), siendo $X_1, \dots, X_n$ las variables libres de (A,G), y se codifica en un término Prolog `rule(St,Vars,A,G)`, donde St = str(A),[4] y Vars = $[X_1, \dots, X_n]$.

El punto fijo se calculará estrato a estrato (aunque un estrato puede requerir el cálculo del punto fijo para un estrato previo cuando el programa aumenta debido a la implicación, como se verá en la sección 1.4). El predicado `fixPointStrat(+Delta,+St,-Fix)` calcula Fix = fix_{St}(Delta). De este modo, si Delta representa una base de datos tal que St = str(Delta) = k, este predicado da fix_k(Delta), calculando los puntos fijos previos desde St = 0 hasta St = k.

```
fixPointStrat(_Delta,0,[]) :- !.
fixPointStrat(Delta,St,FixSt) :- St1 is St-1,
  fixPointStrat(Delta,St1,FixSt1), iterT(Delta,St,FixSt1,FixSt).
```

Cada punto fijo se evalúa iterando el operador de punto fijo como sigue:

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

I representa la interpretación calculada hasta el momento y FixSt será el punto fijo para el estrato considerado St. El operador se itera hasta que no se pueda añadir más información a la interpretación (I==TpI), i.e., hemos alcanzado el punto fijo del estrato St. A continuación detallamos el predicado opT.

4.2 El operador de punto fijo

El predicado opT corresponde a la aplicación del operador T_i (para algún estrato i) a una interpretación dada. Siguiendo la definición 2.5, el predicado `opT(+Rules,+Delta,+St,+I,-TI)` recibe en I el conjunto de pares de $T_i^n(fix_{i-1})$(Delta) para algún $n \geq 0$, el estrato i = St y calcula TI= $T_i^{n+1}(fix_{i-1})$(Delta). La llamada a opT desde iterT tiene la forma de `opT(Delta,Delta,St,I,TI)`, duplicando el parámetro Delta porque utiliza cada una de las cláusulas de Delta por separado, pero la relación de forzado necesitará la base de datos completa Delta. Este operador utiliza solo reglas del estrato actual St (segunda cláusula) y obvia el resto (última cláusula).

```
opT([],_Delta,_St,I,I).
opT([rule(St,Vars,A,G)|Rs],Delta,St,I,TI) :-
  !, rename(Vars,(A,G),Vars1,(A1,G1)),
  flatHead(A1,A2,Cs), buildExists(Vars1,(Cs,G1),G2),
```

[4] La información de estrato se podría anotar una vez por predicado en vez de anotarla en cada regla, pero tal como está simplificará el cómputo posterior


```
  ( force(Delta,I,G2,C), !, addItemLst([(A2,C)],[],I,I1) ; I1=I ),
  opT(Rs,Delta,St,I1,TI).
opT([_|Rs],Delta,St,I,I1) :- opT(Rs,Delta,St,I,I1).
```

La segunda cláusula primero hace algunas transformaciones sobre la regla `rule(St,Vars,A,G)`: los predicados rename, flatHead y buildExists construyen el objetivo a forzar G2 = ∃Vars1(G1 ∧ A1 ≈ A2), siendo ∀Vars1(G1 ⇒ A1) una variante de `rule(St,Vars,A,G)`. Después intenta forzar el objetivo obtenido G2 utilizando Delta y la interpretación computada hasta el momento I. Si tiene éxito obtenemos la restricción asociada C y añadimos el par (A2,C) a dicha interpretación. Finalmente, opT realiza la misma operación con el resto de reglas Rs.

4.3 Relación de forzado

Implementamos la relación de forzado ⊩ de la definición 2.4 mediante el predicado `force(+Delta,+I,+G,-C)`. Dada I = $T_i^n(fix_{i-1})$(Delta) para algún $n \geq 0$ y un estrato fijado $i > 0$, este predicado tiene éxito si $T_i^n(fix_{i-1})$, Delta ⊩ (G,C). Para comprender la implementación es importante tener presente la naturaleza determinista de este predicado. La definición de ⊩ establece condiciones sobre una restricción C con el fin de satisfacer I,Delta ⊩ (G,C), pero nuestro predicado force debe construir una restricción determinada C. Además, cada posible restricción respuesta para un objetivo debe ser capturada en una única restricción, usando disyunciones. Hay una cláusula de force para cada posible forma de objetivo. Las explicamos brevemente, excepto el caso de la implicación que se estudiará con más detenimiento en la próxima subsección:

```
force(_Delta,_I,constr(Dom,C),C1) :- !, solve(Dom,C,C1).
force(Delta,I,(G1,G2),C) :-
  !, force(Delta,I,G1,C1), force(Delta,I,G2,C2), solve((C1,C2),C).
force(Delta,I,(G1;G2),C) :- !,
  ( force(Delta,I,G1,C1), !,
    ( force(Delta,I,G2,C2), !, solve((C1;C2),C) ; solve(C1,C) )
  ; force(Delta,I,G2,C2), solve(C2,C) ).
force(Delta,I,(constr(Dom,C)=>G),C2) :-
  !, force(Delta,I,G,C1), constr_conj(Dom,C2,C,C1).
force(Delta,I,ex(X,G),C) :- !, replace(X,X1,G,G1),
  force(Delta,I,G1,C1), solve(ex(X1,C1),C).
force(Delta,I,fa(X,G),C) :- !, replace(X,X1,G,G1),
  force(Delta,I,G1,C1), solve(fa(X1,C1),C).
force(_Delta,I,not(At),C) :- !, lookUpAll(At,I,Ls),
  ( Ls==[], !, C=true ; buildNegConj(Ls,NLs), solve(NLs,C) ).
force(_Delta,I,At,C) :- !, lookUpAll(At,I,Cs),
  buildDisj(Cs,C1), solve(C1,C).
```

La primera cláusula fuerza una restricción C asociada a un dominio Dom invocando al resolutor. La segunda fuerza una conjunción G1,G2 mediante el forzado de ambos objetivos y después resolviendo la conjunción de las restricción respuesta. Para una disyunción G1;G2 (tercera cláusula) hay cuatro posibles situaciones excluyentes: ambos objetivos se pueden forzar, solo G1, solo G2 o ninguno de ellos: la restricción respuesta se obtiene resolviendo las restricción correspondiente o fallando (último caso). La cuarta clásula de force corresponde a una implicación con una restricción como antecedente: en este caso el predicado `constr_conj` obtiene una restricción C2 tal que si I fuerza (G,C1) entonces la conjunción C2,C equivale a C1. Para el universal, de acuerdo con la definición 2.4, para encontrar C tal que I.Delta ⊩ (∀XG,C) se obtiene G1 como resultado de remplazar X por una nueva variable X1 en G; después se prueba I.Delta ⊩ (G1,C1) y por último C se obtiene resolviendo ∀X1 C1. Para el existencial, según la definición 2.4 buscamos C tal que exista C' con I.Delta ⊩ (G[X1/X],C') y C $\vdash_{\mathcal{C}}$ ∃X1 C', lo que permite en la implementación tomar C como la forma resuelta de ∃X1 C'.

Para átomos negados `not(At)`, gracias a la estraficación sabemos que cada posible átomo At deducible de la base de datos, está ya presente en la interpretación actual I. Entonces, mediante `lookUpAll(At,I,Ls)` se calcula la lista Ls=[C1,...,Cn] tal que (At,Ci)∈I. En la variable NLs se construye la restricción ¬C1∧...∧¬Cn (o true si Ls=[]), que debemos resolver para obtener la restricción C que buscamos.

El último caso (por defecto) es el forzado de un átomo At. Como antes, buscamos todos los pares (At,C1),...,(At,Cn)∈I y después construimos la disyunción C1=C1∨...∨Cn y la resolvemos con `solve`.

4.4 El caso D => G *de la relación de forzado*

La implementación de `force(Delta,I,(D=>G),C)` requiere un trato especial. En este caso, de acuerdo con la definición de la relación ⊩ (ver Definición 2.4), Delta se aumenta con la cláusula D. Por lo tanto el conjunto I actual no es válido ahora, dado que fue calculado para la base de datos Delta. Esto significa $(A,C) \in$ I $\Leftrightarrow$ $(A,C) \in T_i^n(I')$(Delta), para el estrato i y la iteración n que estamos construyendo, donde I' es el punto fijo del estrato $i-1$, construido para Delta. Siguiendo la teoría, el siguiente paso sería demostrar $T_i^n(I')$,Delta ∪ {D} ⊩ (G,C). Pero obsérvese que no se verifica que, para cualesquiera I, Δ, D sea cierto $I(\Delta) \subseteq I(\Delta \cup \{D\})$, luego aquí el conjunto I no va a ser útil ya que solo se han calculado los valores de la función $T_i^n(I')$ para Delta, y además el punto fijo I' también se ha calculado para Delta, pero no sabemos nada de $T_i^n(I')$(Delta ∪ {D}).

Lo que ocurre es que la definición del operador de punto fijo no es constructiva para el caso de la implicación, debido al incremento del conjunto de cláusulas. Para solventar este obstáculo, se ha adoptado una posición conservadora: construir localmente el punto fijo del estrato j para Delta ∪ {D}, donde j es el estrato de G, es decir fix_j(Delta ∪ {D}) y a continuación probar si fix_j, Delta ∪ {D} ⊩ (G,C).

La correspondiente cláusula para el predicado `force` es como sigue:

```
force(Delta,I,(D=>G),C) :- !,
  elab(D,De), localRules(De,Ls), getStrat(G,StG),
  addLocalRules(Ls,Delta,Delta1), fixPointStrat(Delta1,StG,Fix),
  force(Delta1,Fix,G,C).
```

Las llamadas `elab(D, De)`, `localRules(De, Ls)` y `addLocalRules(Ls, Delta, Delta1)` elaboran el conjunto de cláusulas Delta ∪ {D} para obtener el correspondiente conjunto Delta1 en el formato usado. La llamada `fixPointStrat(Delta1, StG, Fix)` encuentra Fix $=$ fix_j(Delta1), donde j = StG es el estrato de G, el consecuente del objetivo inicial D => G.

Pero de esta forma surge el siguiente problema. Sea A :- D => G una cláusula

en Delta, tal que $str(A) = i$. Cuando se intenta añadir un par (A, C) a la actual I, se ejecuta `force(Delta,I,(D=>G),C)` (excepto renombramiento de las variables de D => G). Si el estrato de G es también i, `fixPointStrat(Delta1,i,Fix)` será invocado y de nuevo se ensayará si la cláusula A :- D => G incorpora información al punto fijo, ya que pertenece al estrato i. Esto da lugar a un bucle al ser Delta1 aumentado con D una vez más, y así sucesivamente. Sin embargo, si el estrato de G es j, $j < i$, entonces Fix $= fix_j$(Delta1) se puede construir correctamente. Esta es la razón por la cual es necesaria una estratificación más fuerte a la que aparece en los preliminares y que se explica en la siguiente sección. Nótese que a pesar de restringir la sintaxis del lenguaje, su capacidad expresiva sigue siendo mucho mayor que la del álgebra relacional (que no tiene recursión) y la de Datalog (que carece de cuantificaciones e implicación).

Una vez calculado el conjunto Fix, es necesario forzar G con esta nueva interpretación y el conjunto aumentado Delta1. Esto se corresponde con `force(Delta1, Fix, G, C)`, que implica $T_i^n(I')$, Delta ∪ {D} ⊩ (G,C), como queríamos demostrar.

5 Implementación del grafo de dependencias

En [10] se definió un algoritmo que calcula el grafo de dependencias para cualquier conjunto de fórmulas de $HH_{\neg}(\mathcal{C})$; las principales ideas y definiciones aparecen en la sección 2.2. Debido a la problemática introducida por las implicaciones anidadas, que hemos expuesto previamente, en esta implementación se ha adoptado una definición de base de datos estratificable más restrictiva. Ahora las implicaciones anidadas introducirán nuevas dependendencias negativas en el grafo. Más precisamente, si $G \Rightarrow A$ es una cláusula, tal que G contiene un subobjetivo de la forma $D \Rightarrow G'$, entonces el predicado de A también depende negativamente de los predicados en G'.

El algoritmo de cálculo del grafo de dependencias se expresa mediante las funciones mutamente recursivas $dpClause$ y $dpGoal$ que se definen en la figura 1, y dependen de la estructura de la fórmula. Se devuelve una terna $<E, N, I>$, donde E es un conjunto de arcos, N e I son conjuntos auxiliares de nodos enlace que almacenan información sobre los predicados positivos/negativos, y de aquellos involucrados en implicaciones anidadas, respectivamente. Los arcos en el grafo son de la forma $p \rightarrow q$ (q depende de p) o $p \xrightarrow{\neg} q$ (q depende negativamente de p).

Mediante el uso de la función $dpClause$ es sencillo calcular el grafo de dependencias de un conjunto de clásulas mediante la unión de los arcos que se obtienen para cada elemento del conjunto.

El grafo de dependencia se usa para definir la estratificación en $HH_{\neg}(\mathcal{C})$, que es una condición sintáctica que asegura un cálculo finito para átomos negados.

Ejemplo 5.1 Considérese la cláusula:
$D \equiv \forall x(G \Rightarrow p(x))$, donde $G \equiv \exists y(q(x,y) \Rightarrow (r(x) \wedge s(y))) \wedge \neg t(x)$, entonces,

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

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

La primera componente de la tupla $dpClause(D)$ es el grafo de dependencias asociado a D. Una base de datos con solo esta cláusula es estratificable, pero si se

(i) $dpClause(A) = <\emptyset, \{p_A\}, \emptyset>$

(ii) $dpClause(D_1 \wedge D_2) = <E_1 \cup E_2, N_1 \cup N_2, I_1 \cup I_2>$
si $dpClause(D_1) = <E_1, N_1, I_1>$ y $dpClause(D_2) = <E_2, N_2, I_2>$

(iii) $dpClause(\forall x\, D) = dpClause(D)$

(iv) $dpClause(G \Rightarrow A) =$
$<E_G \cup \bigcup_{n \in E_G} \{n \to p_A\} \cup \bigcup_{n \in N_G} \{n \neg\!\!\to p_A\} \cup \bigcup_{n \in I_G} \{n \dashrightarrow p_A\}, \{p_A\}, I_G>$
si $dpGoal(G) = <E_G, N_G, I_G>$

(i) $dpGoal(A) = <\emptyset, \{p_A\}, \emptyset>$

(ii) $dpGoal(\neg A) = <\emptyset, \{\neg p_A\}, \emptyset>$

(iii) $dpGoal(C) = <\emptyset, \emptyset, \emptyset>$

(iv) $dpGoal(C \Rightarrow G) = dpGoal(G)$

(v) $dpGoal(G_1 \wedge G_2) = dpGoal(G_1 \vee G_2) = <E_1 \cup E_2, N_1 \cup N_2, I_1 \cup I_2>$
si $dpGoal(G_1) = <E_1, N_1, I_1>$ y $dpGoal(G_2) = <E_2, N_2, I_2>$

(vi) $dpGoal(\exists x\, G) = dpGoal(\forall x\, G) = dpGoal(G)$

(vii) $dpGoal(D \Rightarrow G) = <E_D \cup E_G \cup \bigcup_{m \in N_G} (\bigcup_{n \in N_D} \{n \to m\} \cup \bigcup_{n \in N_D} \{n \dashrightarrow m\}), N_D \cup N_G, N_G>$
si $dpClause(D) = <E_D, N_D, I_D>$ y $dpGoal(G) = <E_G, N_G, I_G>$

Notación: p_A representa el símbolo de predicado del átomo A

Fig. 1 Grafo de dependencias para cláusulas y objetivos

añade la cláusula $D' \equiv \forall x \forall y (p(x) \Rightarrow q(x, y))$ se convierte en no estratificable. □

El algoritmo específico que encuentra una estratificación para Δ (o comprueba que no es estratificable) asocia a cada predicado p una variable entera $X_p \in [1..N]$, N número de predicados de Δ y genera el sistema de inecuaciones: cada dependencia $p \to q$ produce $X_p \leq X_q$ y $p \neg\!\!\to q$ produce $X_p < X_q$. Entonces, si el sistema se puede resolver se asocia el estrato X_p a cada p.

Una estratificación para la cláusula D del ejemplo 5.1 dará el estrato 1 a todos sus predicados menos a p, que estará en el estrato 2. En concreto $X_q < X_p$. Intuitivamente, para evaluar p el resto de predicados se deben evaluar antes, en particular q que interviene en una implicación anidada. Considerando la cláusula D' anterior, tenemos $X_q \geq X_p$, con lo que el sistema de inecuaciones no tiene solución.

6 Conclusión

Para terminar mostraremos el resultado del cálculo del punto fijo del ejemplo 1.1. La estratificación, asocia el estrato 1 a `client`, `pastDue`, `mortgageQuote`, `debtor`, `interestRate`; el estrato 2 a `newMortgage` y el estrato 3 a `personalCredit`. En la primera iteración del primer estrato se obtienen los pares correspondientes a la base de datos extensional. El punto fijo del primer estrato requiere una iteración más. A continuación se calcula el punto fijo del segundo estrato y por último el punto fijo final será el del tercer estrato, que estará formado por los pares antes mencionados y los siguientes:

```
[ (debtor(smith), true), (interestRate(brown, 2), true),
(interestRate(X, Y), X=smith, Y=5;X=mcandrew, Y=5),
(newMortgage(X, Y), Y<=200.0, X=brown;Y<=1100.0, X=mcandrew)
```



```
(personalCredit(X, Y), X/=brown, X/=mcandrew, Y>=1500.0;
Y<1500.0, X=brown; Y<1500.0, X=mcandrew)]
```

Las grandes dificultades de la implementación de nuestro sistema han consistido en adaptar las técnicas habituales de construcción de puntos fijos estratificados no solo para que se pueda trabajar con restricciones, sino también para que se tenga en cuenta el hecho de que la base de datos crece dinámicamente con cláusulas locales, cuando se formula una consulta hipotética. El próximo paso será extender el prototipo para poder trabajar con sistemas de restricciones cooperantes.

Agradecimientos: A Jan Wielemaker, autor de SWI-Prolog, y a Markus Triska, autor de la biblioteca de dominios finitos para este sistema, por su amabilidad al ayudarnos en el desarrollo de nuevas características que necesitamos para la implementación de nuestros sistemas de restricciones.

References

[1] ConceptBase V7.1 User Manual. Technical report, RWTH Aachen, April 2008.

[2] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo. The deductive database system ldl++. *TPLP*, 3(1):61–94, 2003.

[3] M. Becker, C. Fournet, and A. Gordon. Design and semantics of a decentralized authorization language. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 3–15, Washington, DC, USA, 2007. IEEE Computer Society.

[4] A. Calì, G. Gottlob, and T. Lukasiewicz. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In *ICDT*, pages 14–30, 2009.

[5] R. Fikes, P. J. Hayes, and I. Horrocks. Owl-ql - a language for deductive query answering on the semantic web. *J. Web Sem.*, 2(1):19–29, 2004.

[6] C. Holzbaur. Realization of forward checking in logic programming through extended unification. report tr-90-11, oesterreichisches forschungsinstitut fuer *Artificial Intelligence*, 1990.

[7] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In C. Li, editor, *PODS*, pages 1–12. ACM, 2005.

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

[9] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[10] 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.

[11] 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 *Lecture Notes in Computer Science*, pages 289–304, Ise, Japan, 2008. Springer-Verlag.

[12] G. Ramalingam and E. 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.

[13] R. Ronen and O. Shmueli. Evaluating very large datalog queries on social networks. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 577–587, New York, NY, USA, 2009. ACM.

[14] F. Sáenz-Pérez. Datalog educational system. user's manual version 1.6.2. Technical report, Faculty of Computer Science, UCM, march 2009. Available from http://des.sourceforge.net/.

[15] K. Sagonas, T. Swift, and D. S. Warren. Xsb as an efficient deductive database engine. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 442–453, New York, NY, USA, 1994. ACM.

[16] J. Wielemaker. Swi-prolog. user's manual version 5.6.64. Technical report, 2009. Available from http://www.swi-prolog.org/.

[17] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

Programación y Lenguajes

IX Jornadas sobre Programación y Lenguajes, PROLE'09
I Taller de Programación Funcional, TPF'09

San Sebastián, España

del 8 al 11 de Septiembre de 2009

Editores:

PAQUI LUCIO, GINÉS MORENO Y RICARDO PEÑA

:dita:
'aqui Lucio Carrasco
;inés Moreno Valverde
ticardo Peña Marí

'ilmación e impresión:
;ráficas Michelena

)epósito Legal:
;S-990-2009

SBN:
·78-84-692-4600-9

Comité de Programa de PROLE-2009

Presidente Comité: Ginés Moreno (U. de Castilla-La Mancha)

Elvira Albert (U. Complutense de Madrid)
Jesús Almendros (U. de Almería)
María Alpuente (U. Politécnica de Valencia)
Gilles Barthe (IMDEA)
Miquel Bertran (U. Ramón Llull)
Santiago Escobar (U. Politécnica de Valencia)
Antonio Fernández (U. de Málaga)
Victor Gulías (U. de A Coruña)
Paqui Lucio (U. del País Vasco)
Narciso Martí-Oliet (U. Complutense de Madrid)
Susana Muñoz (U. Politécnica de Madrid)
Marisa Navarro (U. del País Vasco)
Manuel Núñez (U. Complutense de Madrid)
Fernando Orejas (U. Politécnica de Catalunya)
Yolanda Ortega (U. Complutense de Madrid)
Francisco Ortín (U. de Oviedo)
Ernesto Pimentel (U. de Málaga)
Germán Puebla (U. Politécnica de Madrid)
Enric Rodríguez (U. Politécnica de Catalunya)
Jaime Sánchez (U. Complutense de Madrid)
Germán Vidal (U. Politécnica de Valencia)

Comité de Programa de TPF-2009

Presidente Comité: Ricardo Peña (U. Complutense de Madrid)

Victor Gulías (U. de A Coruña)
Francisco Gutiérrez (U. de Málaga)
Salvador Lucas (U. Politécnica de Valencia)
Pablo Nogueira (U. Politécnica de Madrid)
Julio Rubio (U. de la Rioja)
Alberto Verdejo (U. Complutense de Madrid)
Mateu Villaret (U. de Girona)

Comité Organizador de JISBD/PROLE-2009

Comité Ejecutivo de PROLE-2009

Revisores adicionales de PROLE/TPF-2009

Índice

Prólogo

Las Jornadas sobre PROgramación y LEnguajes (PROLE) se vienen con- solidando como un marco propicio de reunión, debate y divulgación para los grupos españoles que investigan en temas relacionados con la programación y los lenguajes de programación. La investigación en este campo está en continuo desarrollo y comprende todo el estudio de conceptos, métodos, técnicas, funda- mentos y aplicaciones relativos a la tarea de programar y a los lenguajes que se utilizan en ella. El evento, de carácter anual, pretende fomentar tanto el inter- cambio de experiencias y resultados, como la comunicación y cooperación entre los grupos de investigadores españoles que trabajan en el área de programación y lenguajes, manteniendo un año más la enriquecedora trayectoria de las ocho ediciones previas celebradas en Almagro (2001), El Escorial (2002), Alicante (2003), Málaga (2004), Granada (2005), Sitges (2006), Zaragoza (2007) y Gijón (2008).

En esta ocasión, la IX edición de las Jornadas (PROLE'09) va precedida por primera vez en su historia del I Taller sobre Programación Funcional (TPF'09). Ambos eventos se celebran entre el 8 y el 11 de septiembre de 2009, dentro de la XXVIII edición de los Cursos de Verano de San Sebastián. Como en oca- siones previas, la organización de esta conferencia se realiza en paralelo con las Jornadas de Ingeniería del Software y Bases de Datos (JISBD'09), compar- tiendo conferencias invitadas, actos sociales, publicidad, etc. Para información más detallada puede consultarse `http://www.mondragon.edu/prole2009/`. La organización conjunta de ambos eventos ha sido auspiciada por la Sociedad de Ingeniería del Software y Tecnologías de Desarrollo de Software (SISTEDES). Agradecemos desde aquí el soporte, la infraestructura y el apoyo prestado por todos los agentes arriba mencionados.

En el ámbito de PROLE'09 se han seleccionado este año un total de 31 tra- bajos, que cubren tanto aspectos teóricos como prácticos relativos a la especifi- cación, diseño, implementación, análisis y verificación de programas y lenguajes de programación, además de herramientas tangibles y sistemas software que incrementan el carácter pragmático del área. Por su parte, el Taller de Progra- mación Funcional TPF'09 que precede a PROLE'09, inicia su recorrido como una actividad independiente y complementaria a PROLE, con un comité de programa propio que ha seleccionado 5 trabajos recogidos también en estas ac- tas, y que se centran en aspectos relacionados con lenguajes de programación con una fuerte componente funcional (en esta edición los trabajos se refieren a los lenguages Haskell, Lisp y Maude), incluyendo herramientas y experiencias docentes y de investigación en torno a este tipo de lenguajes. Como parte del Taller, se celebra también en esta ocasión una mesa redonda acerca de la in- clusión de la programación funcional y lógica en los nuevos planes de Grado que empezarán a impartirse en 2009.

Este volumen recopila por tanto un total de 36 trabajos que fueron rig- urosamente revisados cada uno de ellos por 3 miembros de ambos comités de programa y/o revisores adicionales, a los cuales es necesario agradecer su in- estimable ayuda y reconocer su gran profesionalidad. También en consonancia con este agradecimiento, es justo felicitar a los autores por la calidad de sus trabajos y su contribución a que esta edición sea la de mayor participación en la evolución histórica de PROLE, lo que garantiza la buena salud del evento.

Por otro lado, además de las tres conferencias invitadas que compartimos con la planificación de JISBD'09, el programa de PROLE'09 cuenta este año con una excelente conferencia específica (un resumen de la misma se incluye en este volumen) que, bajo el título de "Understanding program verification", será impartida por K. Rustan M. Leino, de Microsoft Research, USA, a quien agradecemos el haber aceptado tan amablemente nuestra invitación.

Finalmente, queremos agradecer la confianza que han depositado en nosotros, para conducir la presente edición de estas jornadas, a todos los miembros del comité ejecutivo de PROLE, y esperamos no haberles defraudado. En el desempeño de esta tarea, ha sido determinante la ayuda y experiencia prestada por Jesús Almendros, quien presidió la anterior edición de PROLE en Gijón, y a quien aprovechamos para dar mil gracias desde aquí.

Septiembre de 2009

Paqui Lucio
Ginés Moreno
Ricardo Peña

PATROCINADORES

MONDRAGON UNIBERTSITATEA
Sistedes
Universidad del País Vasco / Euskal Herriko Unibertsitatea
EUSKO JAURLARITZA / GOBIERNO VASCO
GOBIERNO DE ESPAÑA
MINISTERIO DE CIENCIA E INNOVACIÓN
MONDRAGON
INTERSYSTEMS
Asociación de Técnicos de Informática
novática UPGRADE
CAJA LABORAL EUSKADIKO KUTXA
ULMA