Implementing a Fixpoint Semantics for a Constraint Deductive Database based on Hereditary Harrop Formulas

Gabriel Aranda-López
Dpto. de Sistemas Informáticos y Computación,
Univ. Complutense de Madrid
garanda@fdi.ucm.es

Susana Nieva
Dpto. de Sistemas Informáticos y Computación,
Univ. Complutense de Madrid
nieva@sip.ucm.es

Fernando Sáenz-Pérez
Dpto. de Ingeniería del Software e Inteligencia Artificial,
Univ. Complutense de Madrid
fernan@sip.ucm.es

Jaime Sánchez-Hernández
Dpto. de Sistemas Informáticos y Computación,
Univ. Complutense de Madrid
jaime@sip.ucm.es

Abstract

This work is aimed to show a concrete implementation of a deductive database system based on the scheme $HH_{\neg}(\mathcal{C})$ (Hereditary Harrop Formulas with Negation and Constraints) following a fixpoint semantics proposed in a previous work. We have developed a Prolog implementation for this scheme that is constraint system independent, therefore allowing to use it as a base for any instance of the formal scheme. We have developed several specific constraint systems: Real numbers, integers, Boolean and user-defined enumerated types. We have added types to the database so that relations become typed (as tables in relational databases) and each constraint is mapped to its corresponding constraint system. The predicates that compute the fixpoint giving the meaning to a database are described. In particular, we show the implementation of a forcing relation (for derivation steps) and highlight how the inherent difficulties have been overcome in a system allowing hypothetical queries, which make the database dynamically grow.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Logic and Meaning of Programs*]: Semantics of Programing Languages; H.2.3 [*Database Management*]: Languages; H.2.4 [*Database Management*]: Systems

General Terms Algorithms, Languages, Theory, Experimentation.

Keywords Hereditary Harrop Formulas, Deductive Databases, Stratification, Constraints, Fixpoint Semantics, Prolog.

1. Introduction

Deductive databases (DDBs) and their query languages have received a great deal of attention recently in many areas, including ontologies [Fikes et al. 2004], the semantic web [Calì et al. 2009], social networks [Ronen and Shmueli 2009], and policy languages [Becker et al. 2007]. The high level expressivity of logic-based query languages has been therefore acknowledged as a useful feature for handling knowledge-based information systems. In particular, Datalog (along its extensions), from which many current references can be found, is playing the role of a renowned language in those settings.

Current deductive database systems (such as, e.g., XSB [Sagonas et al. 1994] –with inputs from the company XSB, Inc.– bddbddb [Lam et al. 2005], LDL++ [Arni et al. 2003], DES [Sáenz-Pérez 2009], ConceptBase [Jarke et al. 2008], .QL [Ramalingam and Visser 2007] –developed by Semmle, Ltd.– and DLV [Leone et al. 2006]) lack several features we provide in the scheme $HH_{\neg}(\mathcal{C})$ (Hereditary Harrop formulas with Negation and Constraints) [Nieva et al. 2008]. These features are helpful for knowledge systems in which more expressive ways of posing queries are needed. The scheme $HH(\mathcal{C})$ was presented in [Leach et al. 2001] as an extension of traditional *LP* (Logic Programming). The one hand, hereditary Harrop formulas extend Horn logic allowing disjunctions, intuitionistic implications and universal quantifiers, improving the expressivity; the other hand, it incorporates the advantages of constraints. Then, $HH_{\neg}(\mathcal{C})$ was obtained by adding negation to the previous scheme in order to conform to the foundations for a DDB, that extends Datalog in the two orthogonal directions, just mentioned.

In our system, a database is a logic program: a set of facts (ground atoms) defining the extensional database, and a set of clauses, defining the intensional database. Clauses can be seen as the definition of views in relational databases. The evaluation of a query with respect to a deductive database can be seen as the computation of a goal from a program (database), and the answer is a constraint. Since the constraint domain is parametric, it is possible to consider different instances (such as arithmetical constraints over real numbers and finite domain constraints).

Let us show the expressivity of our language with the following example written in an instance that allows both real and finite domain constraints.

EXAMPLE 1. Consider the following extensional database for a bank. We follow a syntax similar to Prolog. In addition we write `not` for negation, `=>` for implication, `ex(X,G)` representing $\exists$`X G`,

PPDP'09, September 7–9, 2009, Coimbra, Portugal.

and `fa(X,G)` representing $\forall$`X G`. Some other details of the syntax are deferred to next sections.

```
% 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).
```

where we assume that each client has, at most, a mortgage quote.

Moreover, we can define the following views.

```
% hasMortgage(Name)
hasMortgage(N):- ex(Q,mortgageQuote(N,Q)).
```

A debtor is a client who has a past due with an amount greater than his balance.

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

The interest rate that is applicable to a client is specified by the next relation:

```
% interestRate(Name, Rate)
interestRate(N,2):-
  client(N,B,S),
  B<1200.

interestRate(N,5):-
  client(N,B,S),
  B>=1200.
```

The next relation specifies that a non-debtor client can be given a new mortgage in two situations. First, if he has no mortgage, a mortgage quote smaller than the 40% of his salary can be given. And, second, if he has a mortgage quote already, then the sum of this quote and the new one has to be smaller than that percentage.

```
% newMortgage(Name, Quote)
newMortgage(N,Q) :-
  client(N,B,S),
  not(debtor(N)),
  not(hasMortgage(N,Q1)),
  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)).
```

If the client satisfies the requirements to be given a new mortgage, then it is possible to apply for a personal credit, whose amount is smaller than 6000. Otherwise, if the client does not satisfy that requirements, the amount must be between 6000 and 20000.

```
% personalCredit(Name, Amount)
personalCredit(N,A) :-
  (getMortgage(N),
   A<6000)
  ;
  (not(getMortgage(N)),
   A>=6000,A<20000).
```

For this database, we can query whether every client is a debtor:

```
fa(N,debtor(N)).
```

The answer is `false`.

Moreover it is possible to ask, for example, the quote and the salary of clients whose mortgage quote is greater than 100 with the next query:

```
ex(B,client(N,B,S),mortgageQuote(N,Q),(Q>=100)).
```

The answer constraint, that provides such information is the following disjunction:

```
(Q=400, S=1500, N=brown);
(Q=100, S=3000, N=mcandrew).
```

For knowing whether there are debtors with a past due amount greater than 1000, the following query can be formulated:

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

and the answer is `true`. Note that we are using quantifiers for `N` and `A`, meaning that there are no explicit conditions over them. Otherwise, the answer will be a constraint over them.

The next query corresponds to the question: if for a client we assume that has a balance greater than 2000, what would the interest rate be?

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

the answer is the constraint `R=5`. We are using nested implication to formulate hypothetical queries, in which we can assume both facts and constraints.

The next query involves negation and represents which clients can get a mortgage quote of 400 but *not* a personal credit.

```
newMortgage(N,400), not(personalCredit(N,A)).
```

And the answer is the constraint:

```
(N=mcandrew, A>=6000, A<20000)
```

This constraint means that it is possible to give a new mortgage to client McAndrew but it is *not* possible to give him a personal credit of an amount between 6000 and 20000. □

In this paper, we present an implementation of the fixpoint semantics presented in [Nieva et al. 2008], which is independent of the concrete constraint system. Also, we use a type system for identifying the constraint system to which each constraint in a database belongs. We propose several constraint systems as instances of $HH_{\neg}(\mathcal{C})$ and their solvers. And we explain how they are implemented.

The semantics of a database is computed as a set of pairs (A, C), where A is an atom and C a constraint, that can be deduced from both the extensional and intensional parts of the database. A can be understood as a n-ary relation instance, where their arguments are constrained by C. These pairs are computed by strata, classifying predicates by strata with a new form of stratification driven by both negations and implications occurring in rules. Each stratum should become saturated before trying to saturate any other higher stratum. However, as an implication may occur in a goal, the computation must take into account that the database is augmented with the hypothesis posed in the implication antecedent. Therefore, a fixpoint computation has to be started from scratch since new pairs may be added at lower strata. So-nested subcomputations add a new complexity level with respect to usual bottom-up computations in deductive databases without implications.

Another complexity source comes again from implications, since the variables in $D \Rightarrow G$ can occur both in D and G. When a database Δ is augmented with the local clause D, those variables must be distinguished from other instances of the same variables in Δ. To this end, we recourse to Prolog attributed variables to identify them.

Finally, in order to find a stratification for ensuring finiteness of computations, a new dependency graph is described using a mutually recursive definition between the dependencies introduced by goals and clauses.

The rest of the paper is organized as follows. Section 2 recalls syntactical notions, the stratification needed for classifying predicates into strata due to both negation and implication, as well as stratified interpretations and the forcing relation. Section 3 introduces a user-oriented description of the system and the computation stages of the implementation. Section 4 describes the type system, constraint systems, their solvers and how they are implemented. Section 5 explains how the fixpoint semantics has been implemented by successive applications of an operator, which in turn implements the forcing relation of *HH*$_\neg(\mathcal{C})$. Section 6 describes a new form of the dependency graph needed to implement the forcing of the implication. Section 7 shows an actual, running example of the system in its current form. Section 8 summarizes some conclusions and sketches some future work.

2. Preliminaries

Here, we recall the foundations, presented in [Nieva et al. 2008], in which the implementation is based on.

2.1 Syntax

We consider a set of *defined predicate symbols*, representing the names of database relations, to build atoms, denoted by A, and *non-defined (built-in) predicate symbols*, including at least the equality predicate symbol $\approx$, to build constraints, denoted by C. We will also assume the existence of a set of constant and operator symbols, and a set of variables to build terms, denoted by t.

The constraints we consider belong to a generic system $\mathcal{C} = \langle \mathcal{L}_\mathcal{C}, \vdash_\mathcal{C} \rangle$, where $\mathcal{L}_\mathcal{C}$ is the constraint language and $\vdash_\mathcal{C}$ is a binary *entailment relation*. $\Gamma \vdash_\mathcal{C} C$ denotes that the constraint C is inferred in the constraint system $\mathcal{C}$ from the set of constraints Γ. Some minimal conditions are imposed on $\mathcal{C}$ to be a constraint system (see [Leach et al. 2001] for details). In particular, $\mathcal{C}$ is required to contain $\top$ (true) and $\bot$ (false), and to deal with $\wedge$, $\neg$, and the existential quantifier $\exists$; the constraint system has the responsibility of checking the satisfiability of answers in the constraint domain.

We say that a constraint C is $\mathcal{C}$-satisfiable if $\emptyset \vdash_\mathcal{C} \overline{\exists} C$, where $\overline{\exists} C$ stands for the existential closure of C. C and C' are $\mathcal{C}$-equivalent if $C \vdash_\mathcal{C} C'$ and $C' \vdash_\mathcal{C} C$.

The well-formed formulas in *HH*$_\neg(\mathcal{C})$ can be classified into clauses D (defining database relations) and goals (or queries) G. They are recursively defined by the following rules:

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

The programs, denoted by Δ, are sets of clauses and represent databases. Any Δ can always be given as an equivalent set, $elab(\Delta)$, of implicative clauses with atomic heads in the way we precise now. The *elaboration* of a program Δ is the set $elab(\Delta) = \bigcup_{D \in \Delta} elab(D)$, where $elab(D)$ is defined by:

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

2.2 Stratification

The notion of stratification is used as a syntactical criterion to determine if a query to a database can be potentially be computed in a finite number of steps. The idea is that when $\neg A$ is going to be proved, the stratum of A has been previously saturated (all the answers for A are available) and $\neg A$ can be correctly computed. A stratification for a database is based on the construction of a dependency graph for a set of formulas [Zaniolo et al. 1997].

The nodes of the graph are the defined predicate symbols of the set. An implication of the form $F_1 \Rightarrow F_2$ produces edges and/or paths in the graph from the defined predicate symbols inside F_1 to each defined predicate symbol inside F_2. An edge will be negatively labeled when the corresponding atom occurs negated on the left side of the implication. Notice that in *HH*$_\neg(\mathcal{C})$ implications may occur not only between the head and the body of a clause, but also inside the goals, and therefore in the body of any clause. Since constraints do not include defined predicate symbols, they do not produce dependencies.

Those two kinds of edges are sufficient to guarantee the consistency of the following theory. However, in the implementation, an additional case of producing a negatively labeled edge will be considered. This new case will be explained in Section 6, after motivating it in Section 5.4.

DEFINITION 1. *Given a set of formulas Φ, its corresponding dependency graph DG_Φ, and two predicates p and q, we say that*

- q depends *on p if there is a path from p to q in DG_Φ.*
- q negatively depends *on p if there is a path from p to q in DG_Φ with at least one negatively labeled edge.*

Let $P = \{p_1, \ldots, p_n\}$ be the set of defined predicate symbols of Φ. A stratification *of Φ is a mapping $s : P \to \{1, \ldots, n\}$, such that $s(p) \leq s(q)$ if q depends on p, and $s(p) < s(q)$ if q negatively depends on p. Φ is stratifiable if there is a stratification for it.*

The stratum of a formula F, denoted by $str(F)$, is the maximum $s(p)$, where p is in the set of predicate symbols occurring in F.

Figure 1 shows the dependency graph for the bank database of the introduction. Negative edges are labelled with $\neg$.

2.3 Stratified Interpretations and Forcing Relation

Let $\mathcal{W}$ be the set of stratifiable databases Δ, with respect to the same fixed stratification s, At be the set of open atoms, and $\mathcal{SL_C}$ be the set of $\mathcal{C}$-satisfiable constraints modulo $\mathcal{C}$-equivalence. Interpretations are classified on strata and each interpretation gives information up to its corresponding stratum.

DEFINITION 2. *Let $i \geq 1$, an* interpretation I over the stratum i *is a function $I : \mathcal{W} \to \mathcal{P}(At \times \mathcal{SL_C})$, such that for any $\Delta \in \mathcal{W}$, and any $j > i$, $[I(\Delta)]_j = \emptyset$, where*

$$[I(\Delta)]_i = \{(A, C) \in I(\Delta) \mid str(A) = i\}.$$

We denote by $\mathcal{I}_i$ the set of interpretations over i.

For every $i \geq 1$, an order on $\mathcal{I}_i$ can be defined.

DEFINITION 3. *Let $i \geq 1$ and $I_1, I_2 \in \mathcal{I}_i$,* I_1 is less or equal than I_2 at stratum i*, denoted by $I_1 \sqsubseteq_i I_2$, if for each $\Delta \in \mathcal{W}$ the following conditions are satisfied:*

- *$[I_1(\Delta)]_j = [I_2(\Delta)]_j$, for every $1 \leq j < i$.*

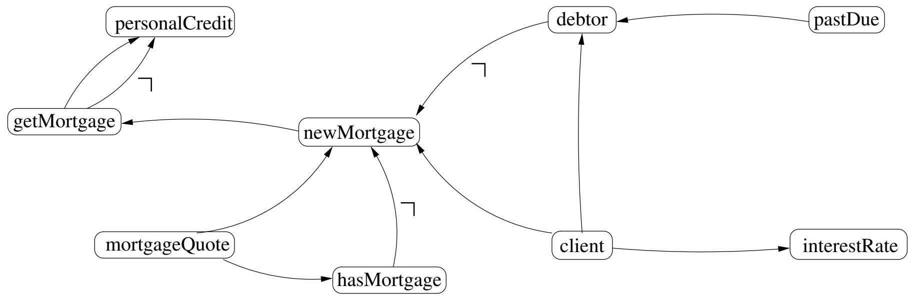

Figure 1. Dependency Graph for Example 1

- $[I_1(\Delta)]_i \subseteq [I_2(\Delta)]_i$.

For every $i \geq 1$, every chain of interpretations of $(\mathcal{I}_i, \sqsubseteq_i)$, $\{I_n\}_{n\geq 0}$, such that $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$, has a least upper bound $\bigsqcup_{n\geq 0} I_n$, which can be defined as:

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

for any $\Delta \in \mathcal{W}$.

The following definition formalizes what means that a query G is *true* for an interpretation I in the context of a database Δ, if the constraint C is satisfied.

DEFINITION 4. *Let $i \geq 1$. The* forcing relation $\Vdash$ *between pairs I, Δ and pairs (G, C) (where $I \in \mathcal{I}_i$, $str(G) \leq i$, and C is $\mathcal{C}$-satisfiable) is recursively defined by the rules below.*

- $I, \Delta \Vdash (C', C) \iff C \vdash_{\mathcal{C}} C'$.
- $I, \Delta \Vdash (A, C) \iff (A, C) \in I(\Delta)$.
- $I, \Delta \Vdash (\neg A, C) \iff$ *for every $(A, C') \in I(\Delta)$, $C \vdash_{\mathcal{C}} \neg C'$ holds. If there is no pair of the form (A, C') in $I(\Delta)$, then $C \equiv \top$.*
- $I, \Delta \Vdash (G_1 \wedge G_2, C) \iff$ *for each $i \in \{1, 2\}$, $I, \Delta \Vdash (G_i, C)$.*
- $I, \Delta \Vdash (G_1 \vee G_2, C) \iff$ *for some $i \in \{1, 2\}$ $I, \Delta \Vdash (G_i, C)$.*
- $I, \Delta \Vdash (D \Rightarrow G, C) \iff I, \Delta \cup \{D\} \Vdash (G, C)$.
- $I, \Delta \Vdash (C' \Rightarrow G, C) \iff I, \Delta \Vdash (G, C \wedge C')$.
- $I, \Delta \Vdash (\exists x G, C) \iff$ *there is C' such that $I, \Delta \Vdash (G[y/x], C')$, where y does not occur free in Δ, $\exists x G$, C, and $C \vdash_{\mathcal{C}} \exists y C'$.*
- $I, \Delta \Vdash (\forall x G, C) \iff I, \Delta \Vdash (G[y/x], C)$, *$y$ does not occur free in Δ, $\forall x G$, C.*

When $I, \Delta \Vdash (G, C)$, it is said that (G, C) is forced *by I, Δ.*

2.4 Fixpoint Semantics

The notion of truth at each stratum is given by means of the fixpoint of a continuous operator (for every stratum) transforming interpretations.

DEFINITION 5. *Let $i \geq 1$ represent a stratum. The* operator T_i : *$\mathcal{I}_i \longrightarrow \mathcal{I}_i$ transforms interpretations over i as follows. Let $I \in \mathcal{I}_i$, $\Delta \in \mathcal{W}$, and $(A, C) \in At \times \mathcal{SL}_{\mathcal{C}}$, then $(A, C) \in T_i(I)(\Delta)$ when:*

- *$(A, C) \in [I(\Delta)]_j$ for some $j < i$ or*
- *$s(A) = i$ and there is a variant $\forall \overline{x}(G \Rightarrow A')$ of a clause in $elab(\Delta)$, such that the variables $\overline{x}$ do not occur free in A, and $I, \Delta \Vdash (\exists \overline{x}(A \approx A' \wedge G), C)$.*

The operator T_1 has a least fixpoint $fix_1 = \bigsqcup_{n\geq 0} T_1^n(I_\perp)$, where the interpretation $I_\perp$ represents the constant function $\emptyset$. Proceeding successively on the same way, a chain:

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

can be defined for any stratum $i > 1$, and a fixpoint of it,

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

can be found. In particular, if k is the maximum stratum in Δ, we simplify fix_k writing fix. Then, $fix(\Delta)$ represents the pairs (A, C) such that A can be deduced from Δ if C is satisfied.

3. System Description

In this section, we briefly introduce a user-oriented description of the system and the computation stages of the implementation.

The system incorporates the predefined data types `bool` (with `true` and `false` as elements) and `real`, an infinite data type, whose real numeric range is system-dependent. As well, the user is able to define new enumerated data types. A data type declaration is written as:

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

Intervals for integers are allowed in data type declarations, as in:

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

An n-arity predicate type declaration is written as:

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

For instance, `type(client(client_dt, real))` is a type declaration, where `client_dt` can be defined as:

```
domain(client_dt, [smith, brown, mcandrew]).
```

The syntax for clauses is essentially as introduced in examples of Section 1, except for constraints, for which we use the syntax `constr(Dom,C)`, denoting a constraint `C` ranging over the domain `Dom`.

When, in the context of a database Δ, a user query `Q` is posed at the system prompt, it is translated into a clause `D` $\equiv$ `A :- Q`, where `A` is an atom whose predicate symbol is `query` and whose arguments are the free variables in `Q` (they are implicitly existentially quantified in `Q` and universally quantified in `D`). In addition, the types for `query` are inferred and provided as the type declaration `type(query(`*`Types`*`))`.

Solving this query entails to add `D` to the current database Δ, i.e., to consider $\Delta' = \Delta \cup \{D\}$ for the following computation stages: 1) Check and infer predicate types; 2) Build the dependency graph of Δ'; 3) Compute a stratification for Δ' if there is any. If it is not stratifiable the system throws an error message an stops; 4) If the previous step success, compute $fix(\Delta')$. The answer constraint to the query Q is the constraint C such that (A,C)$\in fix(\Delta')$.

Next, we describe the different components of the implementation in detail.

4. Implementing Constraint Solving

This section focuses on the implementation of constraint solving for the following particular constraint systems: Real numbers, integers, Boolean and user-defined enumerated types. Firstly, we comment on the type system needed to identify the types of variables which are used to send a constraint to its corresponding solver. Then, the constraint systems are described, including their predefined data values, functions and operators. Finally, we show the implementation of the constraint solvers, which makes use of SWI-Prolog [Wielemaker 2009] underlying constraint solvers.

4.1 Types

We have implemented a type checking and inferrer system for $HH_\neg(\mathcal{C})$ programs which is able to detect type inconsistencies and lack of type declarations, and to infer types for user queries. Types are locally annotated for each predicate symbol. A type annotation consists of storing the type of a variable in an attribute of this variable (cf. attributed variables [Holzbaur 1990]). A type is known in the context of a set of clauses: either a) an atom provides its type (i.e., because of its corresponding predicate type), or b) a constraint `constr(Dom,C)` provides its type. A type-conflict exception is raised when different types are tried to be assigned to the same variable. A lack-of-type-declaration exception is raised when no type is assigned to a variable.

4.2 Constraint Systems

As introduced, a constraint system provides a constraint language for expressing constraints and an entailment relation for ensuring satisfiability of constraints (this relation will be covered in the next subsection). Our constraint systems include the concrete syntax for the required values, symbols, connectives, and quantifiers as follows: "`true`", "`false`", "`=`", "`,`", "`not`" and "`ex(X,C)`" which represent, respectively, $\top$, $\bot$, $\approx$, $\wedge$, $\neg$ and $\exists$X C. In addition, we also include "`;`" for $\vee$ and "`/=`" for the negation of $\approx$.

We have proposed three constraint systems for the scheme $HH_\neg(\mathcal{C})$: Boolean, Reals, and Finite Domains. The first one consists of just the required components plus the universal quantifier. The constraint system Reals includes the type `real` (infinite set of real numeric values) and real constraint operators (`+`, `-`, `*`, ...) and functions (`abs`, `sin`, `exp`, `min`, ...).

Finite Domains represent a family of specific constraint systems ranging over denumerable sets. Enumerated types are included as well as (finite) integer numeric types. Whereas the constraint systems Boolean and Reals have attached predefined types, Finite Domains do not. This system also includes comparison operators (`>`, `>=`, ...), universally quantified constraints (`fa(X,C)`, as above), and the domain constraint `X in Range`, where `Range` is a subset of data values built with `V1..V2`, which denotes the set of values in the closed interval between `V1` and `V2`, and `R1\/R2`, which denotes the union of ranges. A finite domain may also include constraint operators (as `+`, `-`, ...) and constraint functions (as `abs`, `min`, ...). Note that relevant primitive functions for each system should be clear from their intended semantics (`+` might not be relevant for Booleans, although it can be used). We allow to use the same symbols to build constraints in different systems; for instance, both `constr(real, X>Y)` and `constr(month, X>Y)` make sense in their respective constraint systems.

4.3 Constraint Solvers

We have considered the entailment relation of the classical logic for every constraint system. This entailment satisfies the minimal condition imposed to constraint systems. For implementing this relation, we provide a constraint solver with a generic interface `solve(C,SC)` for C $\vdash_{\mathcal{C}}$ SC, intended to solve a constraint C, check its satisfiability and produce a *solved* form SC. A solved form SC corresponding to a constraint C is a simplified, more readable form of the constraint wrt. C. A solved form can be a disjunction of simple constraints, where a simple constraint does neither include disjunctions nor quantifications, nor negations. This generic interface is implemented as follows:

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

Its first call partitions the input constraint into a list whose components belong to different constraint domains. The next call posts each component to its corresponding solve as a call to the predicate `solveFD` (described later). After, the solved constraint represented as a list is transformed back into a constraint data structure. Finally, this constraint is simplified by logical axioms as De Morgan's laws.

In addition to the generic interface, the particular interface `solve(Dom,C,SC)` is also provided, which is useful when the domain `Dom` is already known and can be directly posted to its corresponding solver.

Next, we describe our implementation of the constraint solvers for the constraint systems we propose as practical instances of $HH_\neg(\mathcal{C})$.

We rely on the underlying constraint solvers already available in SWI-Prolog [Wielemaker 2009] for implementing the constraint systems Finite Domains, Boolean and Reals. For certain constraints, we are able to map them to constraints in the underlying SWI-Prolog finite domain solver because we map data values to integers. Before posting to this solver, a constraint is rewritten with the mapped integer values and, after solving, the solved constraint is rewritten back with the corresponding enumerated values. On the other hand, there are constraints that the underlying solvers cannot directly handle (quantifiers and disjunctions) which we explicitly handle as will be shown later. Since SWI- Prolog does not provide a Boolean solver, we resort to the finite domain constraint solver for solving Boolean constraints, and provide the predefined constraint system `bool` which is handled as any other enumerated constraint system.

For the solvers of the constraint systems Finite Domains and Boolean, the following predicates are available:

- `solveFD(+Domain,+Constraint,-SolvedConstraint)`
  It solves the input `Constraint` over `Domain` and returns its solved form `SolvedConstraint` associated to `Domain` , if it is satisfiable.
- `constr_conjFD(+Domain,-C1,+C2,+C)`
  It is read as "`C1,C2 = C`", and computes the component `C1` of the conjunction `C` under the given domain.

Since we consider classical logic for these particular constraint systems, the following implementation for the second predicate is sound:

```
constr_conjFD(Domain,C1,C2,C) :-
  solveFD(Domain,(not(C2);C),C1),
  solveFD(Domain,(C1,C2),SC).
```

Whilst the second line is intended to compute C1 under the assumption of success, the following lines check that the computed constraint is satisfiable.

The code excerpt of Figure 2 implements the required behaviour:

Note that line (05) is intended to replace quantified variables by fresh ones in order to avoid a name clash. Line (07) maps domain data values with integers, whereas line (16) replaces back the (integer) computed data values by the corresponding, mapped data values. The core of constraint solving lays between lines (09)-(11), where, first, the constraint is tried to be solved (see next paragraph describing the predicate solveFD_ctr). Second, it is checked for satisfiability, that is, trying to find a single, concrete solution via labeling. And, third, the underlying constraint store is projected with respect to the relevant variables (i.e., those occurring in the input constraint plus the possible new ones computed by the underlying solver). Lines (13)-(15) are simply intended for data structure formatting.

Next, we describe the predicate:

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

which receives a constraint and returns whether it is satisfiable or not. The first case of this predicate corresponds to a constraint supported by the constraint solver of SWI-Prolog (where #> is the finite domain constraint comparison operator provided by this solver):

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

Negation is, as shown below, explicitly handled because it can apply to unsupported constraints. The predicate

```
complement(+Constraint,-ComplementedConstraint)
```

computes the complemented constraint before solving it.

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

An example of handling unsupported constraints is disjunction, which is computed by collecting all answers (cf. line (08)). Solving this constraint is as follows:

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

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

Finally, we describe quantifiers. Firstly, the existential quantifier is implemented as follows, where in the last but one line satisfiable(FC,true) tries to find a concrete value satisfying FC:

```
solveFD_ctr(ex(X,C),B) :-
  !,
  % Replace X by a fresh variable _FX in C:
  swap(X,_FX,C,FC),
  get_current_domain(DN),
  constrain_domains(FC,DN),
  % Solving:
  (solveFD_ctr(FC,true),
   % Checking satisfiability:
   satisfiable(FC,true),
   B=true ; B=false).
```

The universal quantifier is solved by imposing a conjunctive constraint C for all the values of X in the solving domain (cf. the call to solve_forall):

```
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).
```

The constraint solver for Reals follows a similar but simpler route for its implementation since there are neither universal quantifiers, nor domain data values to map.

5. Implementing the Fixpoint Semantics

In this section, we present the implementation of the core system, which is independent from the concrete constrain systems explained in the previous section.

5.1 Fixpoint by Strata

For the fixpoint computation we assume a stratified database Δ, i.e., a partition $st_1, \ldots, st_k$ over the predicate symbols defined in it (the stratification algorithm will be explained in Section 6). A clause of the form `A :- G` is interpreted as $\forall X_1, \ldots, X_n(\mathtt{G} \Rightarrow \mathtt{A})$, being $X_1, \ldots, X_n$ the free variables of $(\mathtt{A}, \mathtt{G})$, and is encoded as the Prolog term

```
rule(St,Vars,A,G)
```

where St $= str(\mathtt{A})$ and Vars$= [X_1, \ldots, X_n]$.

The fixpoint is computed stratum by stratum (although a stratum may require the computation of the fixpoint for a previous stratum when the program is enlarged due to implications as we will see in Section 5.4). The predicate

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

computes Fix $= fix_{\mathtt{St}}(\mathtt{Delta})$. Then, if Delta represents a database such that St $= str(\mathtt{Delta}) = k$, this predicate gives $fix_k(\mathtt{Delta})$, computing previous fixpoints from St $= 0$ to St $= k$.

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

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

Each fixpoint is evaluated by iterating the fixpoint operator as follows:

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

I represents the current computed interpretation and FixSt will be the fixpoint for the stratum under consideration. The operator is iterated until no more information can be added to the interpretation (I==TI), i.e., we have reached the fixpoint for the stratum St. The predicate opT is detailed below.

```
solveFD(DN,C,SC) :-
  set_current_domain(DN),          % A flag storing the current domain
  copy_term(C,FC),                 % Input variables keep untouched
  get_vars(C,Vars),                % Input variables are held to be
  get_vars(FC,FVars),              %  mapped to the solved new vars
  swap_qvars_by_fvars(FC,QFC),     % Replace quantified vars by fresh ones
  constrain_domains(QFC,DN),       % Constrain variables to the current domain
  domain_to_int(QFC,DN,IC),        % Domain mapping from enumerated to integer
  bagof((FVars,Cs,Sat),            % (Fresh vars,Constraints,Satisfiable)
    (solveFD_ctr(IC,true),         % Solving
     satisfiable(IC,Sat),          % Check satisfiability
     project_ctrs(FVars,Vars,Cs)   % Project constraints wrt. input vars
       ), LFVarsCsS), !            % List of (Fresh vars,Constraints,Satisfiable)
  filter_ctr_list(LFVarsCsS,LICs), % Pick solved constraints
  simplify_disj_list(LICs,SLICs),  % Simplify the disjunctive list
  disj_list_to_ctr(SLICs,ISC),     % Convert list to constraint
  int_to_domain(ISC,DN,SC).        % Map domain from integer to enumerated
```

Figure 2. The Predicate `solveFD` for solving Finite Domain Constraints

5.2 Fixpoint Operator

The predicate `opT` corresponds to the application of the operator T_i (for some stratum i) to a given interpretation. Following Definition 5, the predicate

`opT(+Rules,+Delta,+St,+I,-TI)`

receives in `I` the set of pairs of $T_i^n(fix_{i-1})(\texttt{Delta})$ for some $n \geq 0$, the stratum $i = \texttt{St}$ and computes $\texttt{TI} = T_i^{n+1}(fix_{i-1})(\texttt{Delta})$. The call to `opT` from `iterT` has the form

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

taking `Delta` twice because it uses each clause of `Delta` separately, but the forcing relation will need the full database `Delta`. This operator only uses the clauses of the current stratum `St` (second clause) and skips the rest (last clause).

```
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),
  (
    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).
```

The second clause performs some initial transformations on the rule `rule(St,Vars,A,G)`: the predicates `rename`, `flatHead` and `buildExists` build the goal to be forced

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

being $\forall$ `Vars1` (`G1` $\Rightarrow$ `A1`) a variant of `rule(St,Vars,A,G)`. Then it tries to force the obtained goal `G2` using `Delta` and the current interpretation `I`. If it succeeds, we obtain the associated constraint `C` and we add the pair `(A2,C)` to such an interpretation. Finally, `opT` performs the same operation on the rest of rules `Rs`.

5.3 Forcing Relation

We implement the forcing relation $\Vdash$ of Definition 4 by means of the predicate

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

Given $\texttt{I} = T_i^n(fix_{i-1})(\texttt{Delta})$ for some $n \geq 0$ and a fixed stratum $i > 0$, `force` is successful if $T_i^n(fix_{i-1}), \texttt{Delta} \Vdash (\texttt{G},\texttt{C})$. An important point to understand the implementation is to keep in mind the deterministic nature of this predicate. The definition of $\Vdash$ establishes conditions on a constraint C in order to satisfy $\texttt{I}, \texttt{Delta} \Vdash (\texttt{G}, C)$, but the predicate `force` must build a concrete constraint `C`. In addition, each possible answer constraint for a goal must be captured in a single answer constraint (possibly) using disjunctions. There is a clause of `force` for each goal structure. We explain them shortly, except for the case of implication, that will be studied in the next subsection:

```
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).
```

The first clause stands for the forcing of a constraint `C` within a domain `Dom`, that is processed by calling the constraint solver. The second stands for a conjunction `G1,G2`; it forces both goals, and then solves the conjunction of the resulting answer constraints. For a disjunction `G1;G2` (third clause) there are four possible (and exclusive) situations: both goals can be forced, only `G1`, only `G2`, or neither of two; the answer constraint is obtained by solving the corresponding constraints or failing in the last case. The fourth clause of `force` corresponds to an implication with a constraint as antecedent; in this case the predicate `constr_conj` obtains a constraint `C2` such that if `I` forces `(G,C1)` then the conjunction `C2,C` is equivalent to `C1`.

For the universal quantifier, according to the Definition 4, to find `C` such that $\mathtt{I},\mathtt{Delta} \Vdash (\forall \mathtt{X}\ \mathtt{G},\mathtt{C})$, we obtain `G1` as the result of replacing `X` by a new variable `X1` in `G`; then we prove $\mathtt{I},\mathtt{Delta} \Vdash (\mathtt{G1},\mathtt{C1})$ and finally `C` is obtained by solving $\forall \mathtt{X1}\ \mathtt{C1}$. For the existential quantifier, according to the Definition 4, we find `C` such that there is `C'` satisfying $\mathtt{I},\mathtt{Delta} \Vdash (\mathtt{G}[\mathtt{X1}/\mathtt{X}],\mathtt{C}')$ and $\mathtt{C} \vdash_{\mathcal{C}} \exists \mathtt{X1}\ \mathtt{C}'$. Then we can use `C` as the solved form of $\exists \mathtt{X1}\ \mathtt{C}'$ in the implementation.

For negated atoms `not(At)`, thanks to the stratification we can ensure that every possible atom `At` deducible from the database is already present in the current interpretation `I`. Then, by means of `lookUpAll(At,I,Ls)` we find the list `Ls=[C1,...,Cn]` such that `(At,Ci)`$\in$`I`. The variable `NLs` is used to build the constraint $\neg\mathtt{C1}\wedge\ldots\wedge\neg\mathtt{Cn}$ (or `true` if `Ls=[]`), that we must solve to obtain the constraint `C` we are looking for.

The last (default) case is the forcing of an atom `At`. As before, we search for all the pairs `(At,C1),...,(At,Cn)`$\in$`I` and then we build the disjunction $\mathtt{C1}\vee\ldots\vee\mathtt{Cn}$ and solve it with `solve`.

5.4 The Case of `D => G` in the Forcing Relation

Implementing `force(Delta,I,(D=>G),C)` requires some special treatment. In this case, according with the definition of the relation $\Vdash$ (see Definition 4), `Delta` is augmented with the clause `D`. Remains that the current set `I` has been computed in accordance with the database `Delta`, in such a way that if i and n are, respectively, the stratum and iteration under construction, $(A,C) \in \mathtt{I} \Leftrightarrow (A,C) \in T_i^n(I')(\mathtt{Delta})$, where I' is the fixpoint for the stratum $i-1$, built from `Delta`. According to the theory, the next step will be to prove $T_i^n(I'),\mathtt{Delta}\cup\{\mathtt{D}\} \Vdash (\mathtt{G},\mathtt{C})$. But the question is how to compute $T_i^n(I')(\mathtt{Delta}\cup\{\mathtt{D}\})$. Notice that `I` is not useful here. First, because $I(\Delta) \subseteq I(\Delta \cup \{D\})$ does not hold for every I,Δ,D. Second, because `I` has been built considering always `Delta`, in particular the fixpoint I' has been computed for `Delta`, then it represents $fix_{i-1}(\mathtt{Delta})$. So nothing is known about the needed set $T_i^n(I')(\mathtt{Delta}\cup\{\mathtt{D}\})$.

What it is happening is that the definition of the fixpoint operator T_i is not constructive for the case of implication due to the increase of the set of clauses. To solve this obstacle, we have adopted a conservative position: to compute locally the fixpoint of the stratum j for $\mathtt{Delta}\cup\{\mathtt{D}\}$, where j is the stratum of `G`, that means $fix_j(\mathtt{Delta}\cup\{\mathtt{D}\})$, and then prove if $fix_j,\mathtt{Delta}\cup\{\mathtt{D}\} \Vdash (\mathtt{G},\mathtt{C})$.

Of course, the complexity of the algorithm is considerably augmented on this case. But the code keeps simple. The corresponding clause for the predicate `force` is as follows:

```
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).
```

Calling to `elab(D,De)`, `localRules(De,Ls)`, `getStrat(G,StG)` and `addLocalRules(Ls,Delta,Delta1)`, the elaboration of the set of clauses $\mathtt{Delta}\cup\{\mathtt{D}\}$, is produced giving the corresponding set `Delta1` in the used format. The execution of

$$\mathtt{fixPointStrat}(\mathtt{Delta1},\mathtt{StG},\mathtt{Fix})$$

finds $\mathtt{Fix} = fix_j(\mathtt{Delta1})$, where $j = \mathtt{StG}$ is the stratum of `G`, the consequent of the initial goal `D => G`. Once `Fix` is computed, it is needed to force `G` with it and the augmented set `Delta1`. This corresponds to prove

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

that implies $T_i^n(I'),\mathtt{Delta}\cup\{\mathtt{D}\} \Vdash (\mathtt{G},\mathtt{C})$, as we wanted to prove.

This solution causes the following problem. Consider a clause in `Delta` of the form `A :- D => G`, such that $i = str(\mathtt{A})$ and $j = str(\mathtt{G})$; from Definition 1, $j \le i$ can be deduced. During the computation of $fix_i(\mathtt{Delta})$, the predicate `opT` takes this clause into account, in order to look for a pair `(A,C)` to be added to the current `I`. Then

$$\mathtt{force}(\mathtt{Delta},\mathtt{I},(\mathtt{D} \Rightarrow \mathtt{G}),\mathtt{C})$$

is executed which calls to

$$\mathtt{fixPointStrat}(\mathtt{Delta1},\mathtt{j},\mathtt{Fix}),$$

where $\mathtt{Delta1} = \mathtt{Delta}\cup\{\mathtt{D}\}$ (except elaboration and variable renaming). If $j = i$, that means to build $fix_i(\mathtt{Delta1})$, so the clause `A :- D => G` will be tried again, because the stratum of `A` is i. This gives rise to a non-terminating loop, since `Delta1` is augmented with the elaboration of `D` once more, and so on. However, if $j < i$, $\mathtt{Fix} = fix_j(\mathtt{Delta1})$ can be correctly built. This is the reason why, in the construction of dependency graphs, a new kind of negatively labeled edges has been incorporated, that ensures $str(\mathtt{G}) < str(\mathtt{A})$ in these cases. The details are explained in the following section.

6. Implementing the Dependency Graph

In [Nieva et al. 2006], we defined an algorithm to compute the dependency graph of any set of $HH_{\neg}(\mathcal{C})$ formulas. The main ideas and definitions are introduced in Section 2.2. Due to the problem introduced by nested implications, that we have exposed previously, a stronger definition of stratifiable database has been adopted in the current implementation. Now, these implications will introduce additional negative dependencies in the dependency graph. More precisely, if $G \Rightarrow A$ is a clause, such that G contains a subgoal of the form $D \Rightarrow G'$, this nested implication produces negatively labeled edges from the definite predicate symbols of G' to the predicate symbol of A.

- $dpClause(A) = \langle \emptyset, \{p_A\}, \emptyset \rangle$
- $dpClause(D_1 \wedge D_2) = \langle E_1 \cup E_2, N_1 \cup N_2, I_1 \cup I_2 \rangle$
  if $dpClause(D_1) = \langle E_1, N_1, I_1 \rangle$ and $dpClause(D_2) = \langle E_2, N_2, I_2 \rangle$
- $dpClause(\forall x\, D) = dpClause(D)$
- $dpClause(G \Rightarrow A) =$
  $\langle E_G \cup \bigcup_{n \in (N_G \setminus I_G)} \{n \rightarrow p_A\} \cup \bigcup_{\neg n \in N_G} \{n \xrightarrow{\neg} p_A\} \cup \bigcup_{n \in I_G} \{n \xrightarrow{\neg} p_A\}, \{p_A\}, I_G \rangle$
  if $dpGoal(G) = \langle E_G, N_G, I_G \rangle$

- $dpGoal(A) = \langle \emptyset, \{p_A\}, \emptyset \rangle$
- $dpGoal(\neg A) = \langle \emptyset, \{\neg p_A\}, \emptyset \rangle$
- $dpGoal(C) = \langle \emptyset, \emptyset, \emptyset \rangle$
- $dpGoal(C \Rightarrow G) = dpGoal(G)$
- $dpGoal(G_1 \wedge G_2) = dpGoal(G_1 \vee G_2) = \langle E_1 \cup E_2, N_1 \cup N_2, I_1 \cup I_2 \rangle$
  if $dpGoal(G_1) = \langle E_1, N_1, I_1 \rangle$ and $dpGoal(G_2) = \langle E_1, N_1, I_1 \rangle$
- $dpGoal(\forall x\, G) = dpGoal(\exists x\, G) = dpGoal(G)$
- $dpGoal(D \Rightarrow G) = \langle E_D \cup E_G \cup \bigcup_{m \in N_G} (\bigcup_{n \in N_D} \{n \rightarrow m\} \cup \bigcup_{\neg n \in N_D} \{n \xrightarrow{\neg} m\}), N_D \cup N_G, N_G \rangle$
  if $dpClause(D) = \langle E_D, N_D, I_D \rangle$ and $dpGoal(G) = \langle E_G, N_G, I_G \rangle$

Notation: p_A stands for the predicate symbol of the atom A

Figure 3. Dependency Graph for Clauses and Goals

The algorithm for calculating the dependency graph is expressed by means of the mutually recursive functions $dpClause$ and $dpGoal$ defined in Figure 3, depending on the structure of the formula. Both they return a triple $\langle E, N, I \rangle$, where E is a set of edges of the form $p \rightarrow q$ or $p \xrightarrow{\neg} q$, N and I are auxiliary sets of link-nodes. N is used to store information about the positive-negative predicates, and I stores the predicates involved in nested implications. Using the function $dpClause$ it is straightforward to calculate the dependency graph of a set of clauses as the union of the edges obtained for each element of the set. The dependency graph is used to define the stratification in $HH_\neg(\mathcal{C})$, that is a syntactic condition for ensuring finiteness in the computations with negated atoms.

EXAMPLE 2. Consider the clause:
$D \equiv \forall x(G \Rightarrow p(x))$, where
$G \equiv \exists y(q(x,y) \Rightarrow (r(x) \wedge s(y))) \wedge \neg t(x)$. Then
$dpGoal(G) =$
$\langle \{q \rightarrow r, q \rightarrow s\}, \{q, r, s, \neg t\}, \{r, s\} \rangle$,
$dpClause(D) =$
$\langle \{q \rightarrow r, q \rightarrow s, q \rightarrow p, r \xrightarrow{\neg} p, s \xrightarrow{\neg} p, t \xrightarrow{\neg} p\}, \{p\}, \{r, s\} \rangle$.
The first component of the tuple $dpClause(D)$ is the dependency graph associated to D. A database with just this clause is stratifiable, but if the clause:

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

is also present, the database becomes non stratifiable. □

The concrete algorithm for finding a stratification for Δ (or for checking that it is not stratifiable) associates to each predicate symbol p an integer variable $X_p \in [1..N]$, where N is the number of predicate symbols of Δ, and generates an inequation system: each dependency $p \rightarrow q$ produces $X_p \leq X_q$ and $p \xrightarrow{\neg} q$ produces $X_p < X_q$. Then, solving this system (if possible) provides the stratum of each p in X_p. The stratification algorithm ends with a concrete stratification if there exists one or stops with an error message (in a polynomial time with respect to the number of predicate symbols in the database).

A stratification for the clause D of Example 2 will collect all the predicates in the stratum 1 except p, which will be in the stratum 2. In particular $X_q < X_p$. Intuitively, this means that for evaluating p, the rest of predicates should be evaluated before, in particular q, that takes part of a nested implication. If the previous clause D' is considered, we would also have $X_q \geq X_p$ and the inequation system does not have any solution.

The new negative dependencies introduced in the graph due to nested implications restrict the class of stratifiable programs, i.e., the syntax of our programs. Nevertheless, in practice this restriction does not means a loss of expressivity in the language, that is much more powerful than relational algebra or Datalog.

In the next section, we show (in Figure 4) the whole dependency graph associated to the bank database plus the queries of Example 1. This set is stratifiable. Notice that the edge `interestRate` $\xrightarrow{\neg}$ `query4` is due to the first nested implication inside the clause defining `query4`:

```
query4(R) :- fa(N,ex(S,ex(B,(client(N,B,S) =>
   constr(real,B>2000) => interestRate(N,R))))).
```

This implication produces also `client` $\rightarrow$ `interestRate` and `client` $\rightarrow$ `query4`. So, by transitivity, `query4` negatively depends on `interestRate`, but it also negatively depends on `client`, because `interestRate` depends on `client`.

7. A System Session

Next, we show the result of executing our system for the database and queries Δ that we have shown in Example 1. In this example, the following enumerated domain and types are declared:

```
domain(client_dt,[smith,brown,mcandrew]).
```

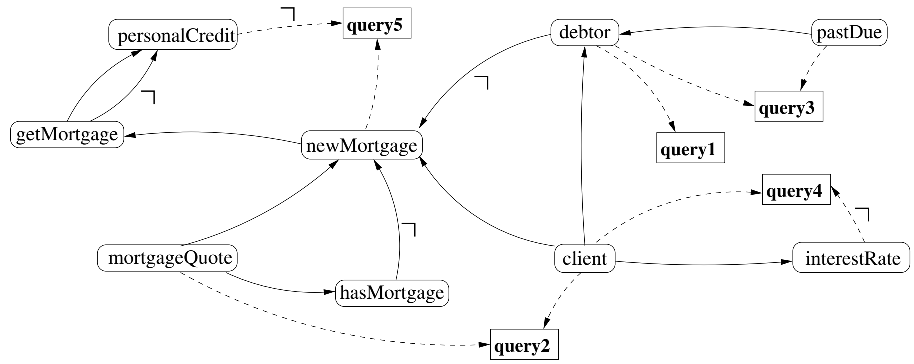

Figure 4. Dependency Graph for Example 1 with some queries.

```
type(client(client_dt,real,real)).
type(pastDue(client_dt,real)).
type(mortgageQuote(client_dt,real)).
type(hasMortgageQuote(client_dt)).
type(debtor(client_dt)).
type(interestRate(client_dt,real)).
type(newMortgage(client_dt,real)).
type(getMortgage(client_dt)).
type(personalCredit(client_dt,real)).
```

The following clauses corresponding to a number of queries are added to the bank database. They are shown along with their types, which are inferred in the context of the above declarations.

```
type(query1).
query1 :- fa(N,debtor(N)).

type(query2(client_dt,real,real)).
query2(N,S,Q) :-
  ex(B,client(N,B,S),mortgageQuote(N,Q),
  constr(real,Q>=100)).

type(query3).
query3 :-
  ex(N,ex(A,(debtor(N),pastDue(N,A),
  constr(real,A>1000)))).

type(query4(real)).
query4(R) :-
  fa(N,ex(S,ex(B,(client(N,B,S) =>
              constr(real,B>2000) =>
              interestRate(N,R))))).

type(query5(client_dt,real)).
query5(N,A) :-
  newMortgage(N,400), not(personalCredit(N,A)).
```

The dependency graph calculated for the current set of clauses is shown in Figure 4 (we use dashed lines for dependencies introduced by the queries).

From this graph, the stratification algorithm associates:

- Stratum 1 to client, pastDue, mortgageQuote, debtor, interestRate, hasMortgage, query1, query2 and query3.
- Stratum 2 to newMortgage,getMortgage, and query4.
- Stratum 3 to personalCredit.
- Stratum 4 to query5.

Since Δ is stratifiable, the computation of

$$\texttt{fixPointStrat}(\Delta,4,\texttt{Fix})$$

begins calculating $fix_i(\Delta)$, stratum by stratum from $i = 1$ to 4, in order to obtain $\texttt{Fix} = fix_4(\Delta)$.

1. Computation of $fix_1(\Delta)$.

   The first iteration of T_1 over the empty set, that corresponds to the execution of $\texttt{opT}(\Delta,\Delta,1,[],\texttt{TI})$, obtains in TI the pairs associated to the extensional database:

   ```
   (client(smith,2000,1200), true),
   (client(brown,1000,1500), true),
   (client(mcandrew,5300,3000), true)
   (pastDue(smith,3000), true),
   (pastDue(mcandrew,100, true),
   (mortgageQuote(brown,400), true),
   (mortgageQuote(mcandrew,100), true)
   ```

   The fixpoint computation of this first stratum requires one more iteration of T_1. After this, the following pairs are added:

   ```
   (debtor(X), X=smith),
   (interestRate(smith, 2), true),
   (interestRate(X,Y),
      ((X=brown, Y=5);
       (X=mcandrew, Y=5))),
   (query2(X,Y,Z),
      ((Y=400, Z=1500, X=brown);
       (Y=100, Z=3000, X=mcandrew))),
   (query3, true),
   (hasMortgage(X), (X=brown;X=mcandrew))
   ```

   Note that no pair due to query1 is added at this stage since the universally quantified constraint in this clause amounts to a conjunctive constraint over the domain of debtor, i.e., imposing that *all* the clients in client_dt are debtors, which is not the case.

2. Computation of $fix_2(\Delta)$.

   Determining whether a pair (query4(X),C) can be added to the current set of pairs gives to locally recalculate fix_1, but this time for $\Delta \cup \{\texttt{client(N,B,S)}\}$.

   To obtain $fix_2(\Delta)$, in the first iteration and after the appropriate computations to calculate $fix_1(\Delta \cup \{\texttt{client(N,B,S)}\})$, the following pairs are added to $fix_1(\Delta)$:

```
(query4(X), X=5),
(newMortgage(X,Y),
   ((Y=<200, X=brown);
    (Y=<1100, X=mcandrew)))
```

And, in the second iteration, the next pair is added:

```
(getMortgage(X), (X=brown;X=mcandrew))
```

3. Computation of $fix_3(\Delta)$.

   Here, a pair for the predicate `personalCredit` is added to the previous fixpoint:

```
(personalCredit(X,Y),
   ((Y>=6000,Y<20000, X=smith);
    (Y<6000, X=brown);
    (Y<6000, X=mcandrew)))
```

4. Computation of $fix_4(\Delta)$.

   The final fixpoint requires one iteration of T_4 over the fixpoint of the third stratum

   $$\texttt{iterT}(\Delta, 4, fix_3(\Delta), \texttt{FixSt}),$$

   obtaining the following new pair:

```
(query5(X,Y),
   (X=mcandrew, Y>=6000, Y=<20000))
```

   This completes the result, and $fix_4(\Delta) = \texttt{FixSt}$ captures the semantics of our database and queries.

In the example, the stratification and fixpoint have been calculated for the database together with all the queries we had formulated. Hence they can be seen as predefined views. It is not the case that the fixpoint should be recomputed each time a query is posed. A more reusable behaviour is also possible in many cases. For a database `Delta`, a stratification s and a fixpoint $\texttt{Fix} = fix(\texttt{Delta})$ can be computed and stored. If the stratification s is valid for the posed query `Q`, then the expected answer constraint `C` can be obtained by executing: `force(Delta,Fix,Q,C)`.

8. Conclusions and Future Work

In [Nieva et al. 2008] we presented a formalization of the constraint logic programing scheme $HH_\neg(\mathcal{C})$ as an expressive deductive database system that returns constraints as answer of the queries. A semantics was developed, following stratification and fixpoint techniques, usual in the framework of deductive database semantics. But the underlying logic of our system embraces both constraints and new connectives on the goals or queries (implications, negation and quantifiers). This fact enlarges expressivity and efficiency, but introduces some penalties in the implementation.

We have developed a prototype of a deductive database system that shows the feasibility of the fixpoint semantics as a base for an actual implementation. The core of this implementation is independent of the concrete constraint system. Several constraint systems are implemented as instances of this scheme. In particular, we have considered real numbers, integers, Booleans and user defined enumerated types (all of these, but reals, belong to the finite domain constraint family). They have been implemented by taking advantage of the underlying constraint solvers in SWI-Prolog. We have added types to programs so that relations become typed (as tables in relational databases) and each constraint is mapped to its solver.

The big difficulties in the implementation of our stratified fixpoint semantics consist of the adaptation of the usual techniques for not only working with constraints but also taking into account that a database can dynamically be augmented with local clauses, when an hypothetical query is formulated. The definition of the fixpoint operator is not constructive for the case of nested implications, then a stronger definition of dependency graph has been formulated to ensure a constructive and terminating fixpoint computation.

Future work The prototype presented in this work can be enhanced to set it as a practical system. The current implementation is very close to the theory developed in our previous works and is a valuable tool for understanding such a theory, but as a consequence it has an expected penalty in efficiency. On the one hand, we have implemented a naïve stratification algorithm for this first prototype that can be easily improved. On the other hand, a more serious source of inefficiency comes from the forcing of implication. In this line, well-known methods as magic set transformations [Beeri and Ramakrishnan 1991] and tabling [Tamaki and Sato 1986] could be worth to be adapted to the current implementation. This is also related to widen the set of computable queries and programs, by adapting the ideas found in the well-founded model [Van Gelder et al. 1991], that could relax our stratification restrictions. This can also be coupled with efficient solving methods [Shen et al. 2002]. In addition, to use existing efficient relational technology to solve concrete queries which do not need the more powerful (less-efficient) database engine we currently provide.

Moreover, in the field of databases, the useful constraint systems are often combinations of different domains. The constraint systems we have implemented work together, but do not cooperate. Due to the nature of the logic involved in our system, finding methods for proving satisfiability of constraints in a mixed domain is a complex task, because the syntax of such constraints will allow, among other aspects, combining existential and universal quantifications for variables of the considered domains. In order to develop a mixed solver, we will consider the existing works that combine concrete domains in the context of $HH_\neg(\mathcal{C})$ [García-Díaz and Nieva 2003] and the combination of decision methods with techniques applied to constraint solvers. This line comes from a fruitful research line in combining constraint systems to cope with problems that, either cannot be handled by a domain constraint solver alone, or its solving can be significatively improved by cooperation of constraint solvers [Hofstedt and Pepper 2007, Castro and Monfroy 2004, Granvilliers et al. 2001].

Acknowledgments

This work has been partially supported by the projects STAMP (TIN2008-06622-C03-0)1, PROMESAS-CAM (S-0505/TIC/0407) and UCM-BSCH-GR58/08-910502. We are also grateful to Jan Wielemaker, author of SWI-Prolog, and Markus Triska, author of the finite domain constraint library for this system, who was very kind to support us in developing new features used in our finite domain constraint solver.

References

F. Arni, K. Ong, S. Tsur, Haixun Wang, and C. Zaniolo. The Deductive Database System LDL++. *Theory and Practice of Logic Programming*, 3(1):61–94, 2003.

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, WashingtonFrancesco, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2819-8. doi: `http://dx.doi.org/10.1109/CSF.2007.18`.

C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3-4):255–299, 1991. ISSN 0743-1066. doi: `http://dx.doi.org/10.1016/0743-1066(91)90038-Q`.

A. Calì, G Gottlob, and T. Lukasiewicz. Datalog±: a unified approach to ontologies and integrity constraints. In *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, pages 14–30, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-423-2. doi: `http://doi.acm.org/10.1145/1514894.1514897`.

C. Castro and E. Monfroy. Designing hybrid cooperations with a component language for solving optimisation problems. In *International Conference on Artificial Intelligence: Methodology, Systems and Applications 2004*, volume 3192 of *LNCS*, pages 447–458. Springer, 2004.

R. Fikes, P. J. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the Semantic Web. *Journal of Web Semantics*, 2(1):19–29, 2004. URL `http://www.informatik.uni-trier.de/~ley/db/journals/ws/ws2.html#FikesHH04`.

M. García-Díaz and S. Nieva. Solving Constraints for an Instance of an Extended CLP Language over a Domain based on Real Numbers and Herbrand Terms. *Journal of Functional and Logic Programming*, 2003 (2), September 2003.

L. Granvilliers, E. Monfroy, and F. Benhamou. Cooperative solvers in constraint programming: a short introduction. *ALP Newsletter*, 14(2), 2001.

P. Hofstedt and P. Pepper. Integration of declarative and constraint programming. *Theory and Practice of Logic Programming*, 7(1-2):93–121, 2007. ISSN 1471-0684. doi: `http://dx.doi.org/10.1017/S1471068406002833`.

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

M. Jarke, M. A. Jeusfeld, and C. Quix. ConceptBase V7.1 User Manual. Technical report, RWTH Aachen, April 2008.

M. S. Lam, S. Whaley, B. V. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In Chen Li, editor, *Symposium on Principles of Database Systems (PODS)*, pages 1–12. ACM, 2005. ISBN 1-59593-062-0.

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

N. Leone, G. Pfeifer, W. Faber, T. Eiter, G Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

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.

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.

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*, 2007. ACM. ISBN 978-1-59593-620-2.

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. ISBN 978-1-60558-422-5. doi: `http://doi.acm.org/10.1145/1516360.1516427`.

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/`.

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. ISBN 0-89791-639-5. doi: `http://doi.acm.org/10.1145/191839.191927`.

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

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. ISBN 0-387-16492-8.

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. ISSN 0004-5411. doi: `http://doi.acm.org/10.1145/116825.116838`.

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

C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Pages 180–183, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-443-X.