Towards a Constraint Deductive Database Language based on Hereditary Harrop Formulas

S. Nieva F. Sáenz-Pérez J. Sánchez-Hernández [1]

Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid
{nieva,fernan,jaime}@sip.ucm.es

Abstract

In the same way that Datalog and Datalog with constraints arise when modelling databases inspired on Prolog and *CLP* (Constraint Logic Programming), respectively, we introduce the constraint logic programming scheme $HH(\mathcal{C})$ (Hereditary Harrop formulas with Constraints) as the basis for a database language. We show that $HH(\mathcal{C})$ can fulfill all relational algebra operations but set difference, so that it has to be extended with a limited form of negation. Since the underlying logic of our system is an extension of Horn clauses, we will show that the resulting database language is more powerful than both relational algebra and calculus. For instance, it is possible to define recursive views. In addition, the use of constraints permits of modelling infinite databases, as in Datalog with constraints. Moreover, our approach improves the expressivity of recursive Datalog with negation because, for instance, hypothetical queries are allowed.

Key words: Constraint Databases, Deductive Databases, Hereditary Harrop Formulas, Logic Programming

1 Introduction

The scheme $HH(\mathcal{C})$ (Hereditary Harrop formulas with Constraints) [10,9,5] extends *HH* by adding constraints, in a similar way to the extension of *LP* (Logic Programming) with constraints gave rise to the *CLP* (Constraint Logic Programming) scheme [6]. In both cases, a parametric domain of constraints is assumed for which it is possible to consider different instances (such as arithmetical constraints over real numbers or finite domain constraints). The extension is completely integrated into the language: constraints are allowed to appear in goals, bodies of clauses and answers.

As a programming language, $HH(\mathcal{C})$ can still be viewed as an extension to *CLP* in two main aspects. Firstly, the logic framework that supports *HH*

[1] This work have been funded by projects TIN2005-09207-C03-03 and S-0505/TIC/0407.

allows to introduce new constructs in goals (and bodies of clauses), such as disjunction, implication and universal quantifiers. On the other hand, these logic connectives are propagated to constraints enriching the constraint language itself that will allow to express more complex conditions.

In this paper, we investigate the use of $HH(\mathcal{C})$ not as a (general purpose) programming language, but as the basis for database systems with constraints [8]. The motivation is that in the same way that Datalog [13,14] and Datalog with constraints [7] arise for modelling database systems inspired in Prolog and *CLP* respectively, the language $HH(\mathcal{C})$ can offer a suitable starting point for the same purpose. We show by means of examples that the expressive power of $HH(\mathcal{C})$ can be translated into the database field improving the existing languages not only theoretically. In particular, implications can be used to write hypothetical queries, and universal quantification allows the encapsulation of information.

However, $HH(\mathcal{C})$ lacks of negation which, as we will see, is needed for our proposal to be complete with respect to relational algebra. As it is well-known, the incorporation of negation into logic programming languages is a difficult task (see [2] for a survey). Negation in the specific field of deductive database systems has been also widely studied [1,3]. In our language, negation is even more complex due to the presence of implication and universal quantification in goals. A proof-theoretic meaning of goals (queries) from programs (databases) is given, when a limited form of negation is used, in such a way that the existence of constraints is exploited to represent answers and to finitely model infinite databases. This is also the case of constraint databases, but the syntax of our constraints is also more expressive than the used, for instance, in Datalog with constraints.

We also study one of the principal problems that arise when dealing with recursion and negation: termination. Following the ideas used in Datalog to ensure termination [13], we adapt the notions of dependency graphs and stratified negation to our context in order to establish syntactic conditions that characterize the limited form of negation introduced.

2 Introducing $HH(\mathcal{C})$ as a Query Language

This section is devoted to informally present $HH(\mathcal{C})$ as a suitable language for databases. Very close to a CLP program, a $HH(\mathcal{C})$ program consists of predicate definitions (clauses and facts) possibly including constraints. A goal is a formula whose answer with respect to a program is a constraint.

2.1 Infinite Data as Finite Representations

One of the advantages of using constraints in the (general) context of *LP* is that they provide a natural way for dealing with infinite data collections using finite (intensional) representations. Constraint databases [8] have inherited this feature. We illustrate this point with an example.

Example 2.1 Assume the instance $HH(\mathcal{R})$, i.e., the domain of arithmetic constraints over real numbers. We are interested in describing regions in the plane. A region is a set of points identified by its characteristic function (a Boolean function that evaluates to true over the points of such a region and to false over the rest of points of the plane). For example, a rectangle can be determined by its left-bottom corner (X_1, Y_1) and its right-top corner (X_2, Y_2) and its characteristic function can be defined as the intensional relation `rectangle(X1,Y1,X2,Y2,X,Y) :- X≥X1, X≤X2, Y≥Y1, Y≤Y2.`

Notice that a rectangle contains (in general) an infinite set of points and they are finitely represented in an easy way by means of real constraints. From a database perspective, this is a very interesting feature: databases were conceived to work with finite pieces of information, but introducing constraints makes possible to manage (potentially) infinite sets of data.

The goal `rectangle(0,0,4,4,X,Y), rectangle(1,1,5,5,X,Y)` computes the intersection of two rectangles and an answer can be represented by the constraint `(X ≥ 1) ∧ (X ≤ 4) ∧ (Y ≥ 1) ∧ (Y ≤ 4)`.

Example 2.2 Using also $HH(\mathcal{R})$, a circle can be defined by its center and radius, now using non-linear constraints:

`circle(XC,YC,R,X,Y) :- ((X-XC)**2 + (Y-YC)**2) ≤ R**2.`

We can ask, for instance, whether any pair (X, Y) such that $X^2 + Y^2 = 1$ (the circumference centered in the origin and radius 1) is inside the circle with center $(0, 0)$ and radius 2 by means of the goal:

`∀ X ∀ Y ((X**2+Y**2 ≈ 1) ⇒ circle(0,0,2,X,Y))`

which cannot be written in standard deductive database languages. However, $HH(\mathcal{C})$ is not expressive enough as we show next.

2.2 Incompleteness

What a database user might want is to have the basic relational operations available in this language. There are five basic relational algebra operators (projection, selection, Cartesian product, union, and set difference). They can be expressed within $HH(\mathcal{C})$, except set difference, that needs some kind of negation. There are some other situations, besides relational database applications, in which negation is quite interesting to be included in the language.

Example 2.3 Returning to Example 2.1, we find the region defined by the inner region of a large rectangle and the outer region of a small rectangle with the goal `rectangle(0,0,4,4,X,Y), ¬ rectangle(1,1,3,3,X,Y)`, and an answer can be represented by the constraint:

`((Y > 3) ∧ (Y ≤ 4) ∧ (X ≥ 0) ∧ (X ≤ 4)) ∨ ((Y ≥ 0) ∧ (Y < 1) ∧ (X ≥ 0) ∧ (X ≤ 4)) ∨ ((Y ≥ 0) ∧ (Y ≤ 4) ∧ (X > 3) ∧ (X ≤ 4)) ∨ ((Y ≥ 0) ∧ (Y ≤ 4) ∧ (X ≥ 0) ∧ (X < 1))`

In this last example, we assume that negation can be effectively handled by the constraint solver, an issue addressed in next sections.

3 Formalizing $HH(\mathcal{C})$ with Negation

The original formalisms in which $HH(\mathcal{C})$ is founded must be extended to introduce negation to obtain a formal *Constraint Deductive Database* (*CDDB*) language. The syntax of this language as well as the meaning of programs and goals will be introduced next.

3.1 Syntax

We will consider that there are two distinguished types of predicate symbols: *defined predicate symbols* that represent the names of database relations, and *non-defined (built-in) predicate symbols*, that depend on the particular constraint system including at least the equality predicate symbol $\approx$. We will also assume a set of constant and operator symbols of the constraint system, and a set of variables to build terms, denoted by t.

The Constraint System $\mathcal{C}$

The constraints we will consider belong to a generic system with a binary *entailment relation* $\vdash_{\mathcal{C}}$, where $\Gamma \vdash_{\mathcal{C}} C$ denotes that the constraint C is inferred in the constraint system $\mathcal{C}$ by the set of constraints Γ. $\mathcal{C}$ is required to satisfy some minimal conditions:

- Every first-order formula built up using $\top$ (true), $\bot$ (false), built-in predicate symbols, the connectives $\wedge, \neg$, the existential quantifier $\exists$, and possibly other connectives or quantifiers $(\vee, \Rightarrow, \forall)$ is in the constraint language $\mathcal{L}_{\mathcal{C}}$.
- All the inference rules related to $\exists, \wedge, \neg, \approx$ and the considered additional connectives, valid in intuitionistic logic with equality, are valid to infer entailments in the sense of $\vdash_{\mathcal{C}}$.
- Compactness: $\Gamma \vdash_{\mathcal{C}} C$ holds when $\Gamma_0 \vdash_{\mathcal{C}} C$ for some finite $\Gamma_0 \subseteq \Gamma$.
- If $\Gamma' \vdash_{\mathcal{C}} \Gamma$ and $\Gamma \vdash_{\mathcal{C}} C$, then $\Gamma' \vdash_{\mathcal{C}} C$. By convention, the notation $\Gamma \vdash_{\mathcal{C}} \Gamma'$ will mean that $\Gamma \vdash_{\mathcal{C}} C$ holds for all $C \in \Gamma'$.

We say that Γ and Γ' are $\mathcal{C}$-equivalent if $\Gamma \vdash_{\mathcal{C}} \Gamma'$ and $\Gamma' \vdash_{\mathcal{C}} \Gamma$.

Notice that $\mathcal{C}$ is required to deal with negation, because the incorporation of $\neg$ to *HH* is propagated to the constraint system, which has the responsibility of checking the satisfiability of answers in the constraint domain.

In previous examples, the considered system ($\mathcal{R}$) is assumed to verify the minimal conditions required to be a constraint system. Moreover, it also includes the connective $\vee$, constants to represent numbers, arithmetical operators, and built-in predicates ($\geq, \ldots$).

The Query Language

Now we make precise the syntax of the formulas of $HH(\mathcal{C})$ extended with negation, showing how the usual notions of programs and goals of Logic Programming can be translated into databases and queries, respectively.

The evaluation of a query to a deductive database can be seen as the computation of a goal from a set of facts (ground atoms) defining the extensional database, and a set of clauses, defining the intensional database. As it is common in deductive databases, the definition of a derived (intensional) predicate, by means of clauses, in our language can be seen as the definition of a view in relational databases.

Clauses D and goals G are recursively defined by the rules below.

$D ::= A \mid G \Rightarrow A \mid D_1 \wedge D_2 \mid \forall x D$

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

A represents an atom, i.e., a formula of the form $p(t_1, \ldots, t_n)$, where p is a defined predicate symbol of arity n. The incorporation of negated atoms in goals is the surplus to $HH(\mathcal{C})$.

Any program 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:

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

We will assume that a view defining a predicate is a set of elaborated clauses of the form $\forall x_1 \ldots \forall x_n (G \Rightarrow A)$ and, in the examples (as before), we will use the common notation A :- G, assuming that capital letters represent variables that are implicitly universally quantified, and incorporating the new connectives in goals. Negation is not allowed in the head of a clause, but inside its body. The condition of defining negation only over atoms is not a limitation, because A :- $\neg G$ can be defined by A' :- G, A :- $\neg A'$.

3.2 Semantics

Several kinds of semantics have been defined for $HH(\mathcal{C})$ without negation, including proof-theoretic, operational (both introduced in [10]) and fixed-point semantics [4]. The simplest way for explaining the meaning of programs and goals in the present framework is by using a proof-theoretic semantics. Queries formulated to a database are interpreted by means of the inference system that governs the underlying logic. This proof system, called $\mathcal{UC}$ (Uniform sequent calculus handling Constraints), combines traditional inference rules with the entailment relation of the generic constraint system $\mathcal{C}$. It provides only uniform proofs in the sense defined by Miller et. al. [11], which means goal-oriented proofs. The rules are applied backwards and, at any step, the applied rule is that including on the right the connective of the goal to be proved.

Provability in $\mathcal{UC}$ is defined as follows. Sequents have sets of programs and constraints on the left, and goals on the right (see Figure 1).

The motivation for the rule $(\exists_R)$ appears in [10]. It includes the fact that substitutions can be simulated by equality constraints, but it is more

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

$\forall x_1 \ldots \forall x_n(G \Rightarrow A')$ is a variant of a formula of $elab(\Delta)$

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

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

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

(*) $x_1, \ldots, x_n$, y do not occur free in the sequent of the conclusion

Fig. 1. Rules of the Sequent Calculus $\mathcal{UC}$

powerful because the use of constraints makes possible to find a proof for an existentially quantified formula, representing the witness of the existentially quantified variable by not necessarily a term, but with a constraint (e.g., $(x * x \approx 2)$ represents $\sqrt{2}$, that cannot be written as a term).

The rule $(Clause)$ represents backchaining. In order to obtain a proof of an atomic goal, the body of a clause will be proved. However, it is not required to unify the head of the clause with the atom to be proved, but to solve an existential quantification that, as it was explained before, will search for a constraint to be satisfied that makes equal the atom and the head of the clause. The notation $A' \approx A$ stands for $t'_1 \approx t_1 \wedge \ldots \wedge t'_n \approx t_n$, where $A \equiv p(t_1, \ldots, t_n)$ and $A' \equiv p(t'_1, \ldots, t'_n)$.

The Meaning of Negated Atoms

Derivability in $\mathcal{UC}$ provides proof-theoretic semantics for $HH(\mathcal{C})$. The incorporation of negation makes necessary to extend the notion of derivability, because there is no rule for this connective in $\mathcal{UC}$. An inference system that extends $\mathcal{UC}$ with a new rule to incorporate derivability of negated atoms will be considered. The idea of interpreting the query $\neg A$ from a database Δ, by means of an answer constraint C, is that whenever C' is a possible answer to the query A from Δ, then $C \vdash_{\mathcal{C}} \neg C'$. This is formalized with the "metarule":

$$\frac{\Gamma \vdash_{\mathcal{C}} \neg\Gamma' \quad for\ any\ \ \Delta;\Gamma' \vdash A}{\Delta;\Gamma \vdash \neg A}\ (\neg_R)$$

where $\Gamma \vdash_{\mathcal{C}} \neg\Gamma'$ means $\Gamma \vdash_{\mathcal{C}} \neg C$, for every $C \in \Gamma'$. We say that $(\neg_R)$ is a metarule since its premise considers any derivation $\Delta;\Gamma' \vdash A$ of the atom A.

The inference system that combines the rules of $\mathcal{UC}$ with $(\neg_R)$ is called $\mathcal{UC}^{\neg}$. The notation $\Delta;\Gamma \vdash_{\mathcal{UC}^{\neg}} G$ means that the sequent $\Delta;\Gamma \vdash G$ has a proof using the rules of $\mathcal{UC}$ and $(\neg_R)$.

Definition 3.1 If $\Delta; C \vdash_{\mathcal{UC}^\neg} G$ then C is called an *answer constraint* of G from Δ. The *meaning of a query* G from a database Δ can be defined as the set of the answer constraints of G with respect to Δ up to $\mathcal{C}$-equivalence.

Due to the conditions imposed to a generic system $\mathcal{C}$ to produce a valid instance of our scheme (for instance, it must be closed under $\neg$) and the definition of proof-theoretic semantics based on $\mathcal{UC}^\neg$-derivations, we can say that closed-form evaluation [12] is assured in $HH(\mathcal{C})$, because output databases are represented by answer constraints that belong to the same constraint system used to define the input database.

Defining an operational semantics for $HH(\mathcal{C})$ with negation can be tackled by adapting the query solver procedure for $HH(\mathcal{C})$ introduced in [10] (which is complete w.r.t. $\mathcal{UC}$) for computing goals including negated subgoals. In particular, some finiteness conditions must be imposed to define the interpretation of negated atoms. Assuming that, for an atom A and a program Δ, only a finite number of different non-$\mathcal{C}$-equivalent answer constraints $C_1, \ldots, C_n$ are obtained with such a mentioned procedure, the subgoal $\neg A$ will success if the current partially calculated answer C is such that $C \vdash_{\mathcal{C}} \neg C_1 \wedge \ldots \wedge \neg C_n$.

What remains to do is to imposse conditions that guarantee to have only a finite number of non-equivalent computed answer constraints for any atom that occurs negated in some goal. In this way, it is possible to impose the following syntactic restrictions as a strong condition: negation is not allowed over predicates that depend on others that include recursion (the notion of dependency is formalized in Section 5).

Note that, even in the case of adopting this strong syntactic restriction, completeness with respect to relational algebra remains, since relational algebra does not include recursion.

Theorem 3.2 (Completeness) *$HH(\mathcal{C})$ with negation is complete with respect to relational algebra.*

Proof (Sketch) The proof is similar to the completeness of Datalog [13]. □

Note also that our use of constraints and disjunction, in particular, provides a richer expressivity than common database languages, as we show next.

4 Expressiveness of $HH(\mathcal{C})$ with Negation

We introduce several examples showing the advantages of our proposal w.r.t. other common database languages. In these examples, a travel database is defined using a hybrid constraint system $\mathcal{FR}$ that combines constraints over finite and real numbers domains, ensuring domain independence. Instantiating the scheme with mixed constraint systems will be very useful in the context of databases. In [4], a hybrid instance subsuming $\mathcal{FR}$ is presented.

Example 4.1 An important benefit of our approach is the ability to formulate hypothetical and universally quantified queries. Consider the database:

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

The predicate `flight(O,D,T)` represents an extensional database relation among origin (`O`), destination (`D`) and duration (`T`). In turn, `travel(X,Y,T)` represents an intensional database relation, expressing that a travel from `X` to `Y` can be done in a time `T`, possibly concatenating some flights.

The next goal asks for the duration of a flight from Madrid to London in order to be able to travel from Madrid to New York in 11 hours at most.

`flight(mad, london, T)` $\Rightarrow$ `travel(mad, ny, 11).`

The answer constraint of this query will be $11 \geq T + 9$ that is $\mathcal{FR}$-equivalent to the final answer $T \leq 2$.

Another hypothetical query to the previous database can be the question that if it is possible to travel from Madrid to some place in any time greater than 1.5. The goal formulation $\forall$ `T (T` $>$ `1.5` $\Rightarrow$ $\exists$ `Y travel(mad,Y,T))` includes also universal quantification, and the corresponding answer is *true*.

Example 4.2 Assume now a more realistic situation in which flight delays may happen, which is represented by the following definition.

```
deltravel(X,Y,T) :- flight(X,Y,T1), delay(X,Y,T2), T ≥ T1+T2.
deltravel(X,Y,T) :- flight(X,Z,T1), delay(X,Z,T2),
                    deltravel(Z,Y,T3), T ≥ T1+T2+T3.
```

Tuples of `delay` may be in the extensional database or may be assumed when the query is formulated. For instance, the goal:

(∀ `X delay(par,X,1),delay(mad,par,0.5))` $\Rightarrow$ `deltravel(mad,ny,T)`

represents the query: What is the time needed to travel from Madrid to New York assumed that every flight from Paris has a delay of one hour and the flight from Madrid to Paris is half an hour delayed? According to its proof-theoretic interpretation, the clauses `delay(par,X,1)` and `delay(mad,par,0.5)` will be added locally to the database in order to solve the goal `deltravel(mad,ny,T)`, and they will not be considered any more once it is solved.

Since flights may or may not be delayed, a more general view can be defined in order to know the expected time of a trip:

```
trip(X,Y,T) :- nondeltravel(X,Y,T) ; deltravel(X,Y,T).
nondeltravel(X,Y,T) :- ¬ delayed(X,Y), travel(X,Y,T).
delayed(X,Y) :- ∃T delay(X,Y,T).
```

5 Stratified Negation and Dependency Graphs

In order to obtain an operational semantics equivalent to the proof-theoretic one, it is needed to guarantee that for any $\neg A$ in a goal, A has a finite number

of answer constraints. Even so, another well known problem arises in deductive database languages when negation and recursive programs are considered. This is the case of non-terminating proofs under the operational semantics. In Datalog, this problem has been undertaken introducing the notions of *stratified negation*, which is based on the definition of a *dependency graph* for a program (see [14] for details). Here, we adapt these notions as an useful starting point of an operational semantics for our language.

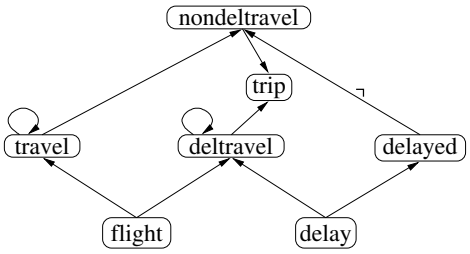
The construction of dependency graphs must consider the fact that implications may occur not only between the head and the body of a clause, but also inside the goals, and therefore in any clause body. This feature will be taken into account in the following way: an implication of the form $F \Rightarrow p(s_1, \ldots, s_m)$ is interpreted as p *depends on* every defined predicate symbol inside the formula F. Quantifiers and constraints in goals must also be treated. However, as constraints do not include defined predicate symbols, they cannot produce dependencies, which means that they are not relevant for determining the graph.

Given a set of clauses and goals Φ, the corresponding dependency graph $DG_\Phi = <N, E>$ is a directed graph whose nodes N are the defined predicate symbols in Φ. The edges E of the graph are determined by the implication symbols of the clauses.

The function *dp* defined in Figure 2 computes dependencies (in the previous sense) of any formula of $HH(\mathcal{C})$ with negation. It returns a pair $<E, L>$, where E is a set of edges and L is a set of link-nodes. Edges in the graph can be of the form $p \rightarrow q$ (which means that q has a dependency on p, or, alternatively, that q depends on p) and $p \stackrel{\neg}{\rightarrow} q$ (q has a negative dependency on p, or q negatively depends on p). The set L of link-nodes is just an auxiliary structure used to link subgraphs obtained by recursion to the total graph; it can contain "negated" nodes of the form $\neg p$ that will be transformed into a node p and negative labelled outcoming edges of the form $\stackrel{\neg}{\rightarrow}$. Using this function, it is straightforward to calculate the dependency graph of a set of clauses and goals as the union of the edges obtained for each element in the set.

It is easy to check that our algorithm for calculating the dependency graph, when applied to Datalog programs, produces the same dependency graph defined for that language.

Example 5.1 Consider a program Δ consisting of the predicates defined in the examples of Section 4. The dependency graph for Δ is:



- $dp(C) = <\emptyset, \emptyset>$
- $dp(A) = <\emptyset, \{pred(A)\}>$
- $dp(\neg A) = <\emptyset, \{\neg pred(A)\}>$
- $dp(Qx\ \varphi) = dp(\varphi)$, where $Q \in \{\forall, \exists\}$
- $dp(\varphi \wedge \psi) = <E_\varphi \cup E_\psi, N_\varphi \cup N_\psi>$, if $dp(\varphi) = <E_\varphi, N_\varphi>$ and $dp(\psi) = <E_\psi, N_\psi>$
- $dp(\varphi \vee \psi) = <E_\varphi \cup E_\psi, N_\varphi \cup N_\psi>$, if $dp(\varphi) = <E_\varphi, N_\varphi>$ and $dp(\psi) = <E_\psi, N_\psi>$
- $dp(\varphi \Rightarrow \psi) = <E_\varphi \cup E_\psi \cup \bigcup_{m \in N_\psi} (\bigcup_{n \in N_\varphi} \{n \rightarrow m\} \cup \bigcup_{\neg n \in N_\varphi} \{n \stackrel{\neg}{\rightarrow} m\}, N_\psi>$, if $dp(\varphi) = <E_\varphi, N_\varphi>$ and $dp(\psi) = <E_\psi, N_\psi>$

Notation:
$pred(A)$: Predicate symbol of atom A; φ, ψ: Formulas of $HH(\mathcal{C})$ with negation

Fig. 2. Dependency Graph for Clauses and Goals

The goal $G \equiv \exists$ `T (deltravel(X,Y,T)` $\Rightarrow$ `delayed(X,Y))` would introduce the new edge $deltravel \rightarrow delayed$ into the previous graph. Another interesting example is $G' \equiv \forall$ `T` $\exists$ `T' ((trip(mad,lon,T)` $\Rightarrow$ `delay(mad,lon,T'))` $\Rightarrow$ `(trip(mad,lon,T)` $\Rightarrow \neg$`delayed(mad,lon))).` Both parts of the main implication are recursively evaluated according to the last case in Figure 2, obtaining: $<\{trip \rightarrow delay\}, \{delay\}>$ and $<\{trip \stackrel{\neg}{\rightarrow} delayed\}, \{\neg delayed\}>$. Finally, using the link-nodes the algorithm produces:

$<\{trip \rightarrow delay,\ trip \stackrel{\neg}{\rightarrow} delayed,\ delay \stackrel{\neg}{\rightarrow} delayed\}, \{\neg delayed\}>$

The dependency graph is used to define stratification in $HH(\mathcal{C})$. Our concept of stratifiable program gives a syntactic condition ensuring finite computations for negated atoms.

Definition 5.2 Given a set of formulas Φ, its corresponding dependency graph DG_Φ, and two predicates p and q, we say:

- 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 labelled edge.
- q *strongly negatively depends* on p if q negatively depends on p and p belongs to a cycle in DG_Φ.

Definition 5.3 A set of formulas Φ is stratifiable if DG_Φ does not contain any predicate that strongly negativelly depends on any other.

It is easy to see that the program of Example 5.1 is stratifiable because its dependency graph does not contain any strong negative dependency. Moreover, it is also stratifiable when adding the edges generated by the goals G and G' of such an example. But the goal:

$$G'' \equiv \texttt{deltravel(mad,lon,T)} \Rightarrow \texttt{delayed(mad,lon,T)}$$

adds the dependency *deltravel* $\rightarrow$ *delayed*. Then, `nondeltravel` strongly negativelly depends on `deltravel`.

Definition 5.4 Let Φ be a set of stratifiable formulas and $P = \{p_1, \ldots, p_n\}$ the set of defined predicate symbols of Φ. A stratification of Φ is any mapping $s : P \rightarrow \{1, \ldots, n\}$ such that $s(p) \leq s(q)$ if q depends on p, and $s(p) < s(q)$ if q negativelly depends on p.

A stratification for the program of Example 5.1 will collect all the predicates in the stratum 1 except `nondeltravel`, which will be in stratum 2. Intuitively, this means that for evaluating `nondeltravel`, the rest of predicates should be evaluated before (in particular, `delayed`). If the previous goal G'' is considered, the finiteness condition to evaluate $\neg$ `delayed` might not be satisfied, because of the strong negative dependency it introduces.

6 Conclusions and Future Work

We have presented a suitable constraint deductive database query language based on $HH(\mathcal{C})$ that, with such end, has been adequately extended to incorporate negation. We claim that the resulting extension subsumes known database query languages as relational algebra and calculus, Datalog, and Datalog with constraints w.r.t. basic operations (for instance, we do not deal with aggregates up to now). Our proposal includes constructs to define databases and queries not found in such known languages. The expressivity of $HH(\mathcal{C})$ as a *CDDB* language has been shown by means of examples.

We have defined a proof-theoretic semantic framework such that the meaning of a database query is represented by the set of constraints that can be derived.

Regarding operational semantics, it is well-known that stratified negation is a good setting for deductive databases with recursion and negation. Following this approach, we have defined our particular concept of dependency graph and stratified program.

As future work, we plan to devise a fixed-point semantics whose fix-point operator is applied stratum by stratum over any stratified program. An implementation is expected to be developed in the near future including the proposal presented here in connection with an adequate constraint solver.

References

[1] Abiteboul, S., R. Hull and V. Vianu, "Foundations of Databases," Addison-Wesley, 1995.

[2] Apt, K. and R. Bol, *Logic Programming and Negation: A Survey*, Journal of Logic Programming **19&20** (1994), pp. 9–71.

[3] Benedikt, M. and L. Libkin, *Safe constraint queries*, in: *PODS '98: Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1998), pp. 99–108.

[4] García-Díaz, M. 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** (2003).

[5] García-Díaz, M. and S. Nieva, *Providing Declarative Semantics for HH Extended Constraint Logic Programs*, in: *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, Verona, Italy, 2004, pp. 55 – 66.

[6] Jaffar, J. and J.-L. Lassez, *Constraint Logic Programming*, in: *14th ACM Symp. on Principles of Programming Languages (POPL'87)* (1987), pp. 111–119.

[7] Kanellakis, P. C., G. M. Kuper and P. Z. Revesz, *Constraint Query Languages*, in: *Symposium on Principles of Database Systems*, 1990, pp. 299–313.

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

[9] Leach, J. and S. Nieva, *A Higher-Order Logic Programming Language with Constraints*, in: H. Kuchen and K. Ueda, editors, *Proc. Fifth International Symposium on Functional and Logic Programming (FLOPS'01)*, LNCS **2024** (2001), pp. 108–122.

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

[11] Miller, D., G. Nadathur, F. Pfenning and A. Scedrov, *Uniform Proofs as a Foundation for Logic Programming*, Annals of Pure and Applied Logic **51** (1991), pp. 125–157.

[12] Revesz, P. Z., *Safe query languages for constraint databases*, ACM Trans. Database Syst. **23** (1998), pp. 58–99.

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

[14] Zaniolo, C., 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.