The Journal of Logic and Algebraic Programming 83 (2014) 20–52

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

www.elsevier.com/locate/jlap

An extended constraint deductive database: Theory and implementation

Gabriel Aranda-López*, Susana Nieva, Fernando Sáenz-Pérez, Jaime Sánchez-Hernández

Facultad de Informática, Complutense University of Madrid, Spain

ARTICLE INFO

Article history:
Received 28 July 2011
Received in revised form 4 April 2013
Accepted 3 July 2013
Available online 9 July 2013

Keywords:
Deductive databases
Constraints
Hereditary Harrop formulas
Fixpoint semantics

ABSTRACT

The scheme of Hereditary Harrop formulas with constraints, $HH(\mathcal{C})$, has been proposed as a basis for constraint logic programming languages. In the same way that Datalog emerges from logic programming as a deductive database language, such formulas can support a very expressive framework for constraint deductive databases, allowing hypothetical queries and universal quantifications. As negation is needed in the database field, $HH(\mathcal{C})$ is extended with negation to get $HH_\neg(\mathcal{C})$. This work presents the theoretical foundations of $HH_\neg(\mathcal{C})$ and an implementation that shows the viability and expressive power of the proposal. Moreover, the language is designed in a flexible way in order to support different constraint domains. The implementation includes several domain instances, and it also supports aggregates as usual in database languages. The formal semantics of the language is defined by a proof-theoretic calculus, and for the operational mechanism we use a stratified fixpoint semantics, which is proved to be sound and complete w.r.t. the former. Hypothetical queries and aggregates require a more involved stratification than the common one used in Datalog. The resulting fixpoint semantics constitutes a suitable foundation for the system implementation.

1. Introduction

The extension of *LP* (Logic Programming) with constraints gave rise to the *CLP* (Constraint Logic Programming) scheme [26,25]. In a similar way, the $HH(\mathcal{C})$ scheme (Hereditary Harrop formulas with Constraints) [31,19] extends *HH* by adding constraints. 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 occur in goals, bodies of clauses, and answers.

As a programming language, $HH(\mathcal{C})$ can still be viewed as an extension of *CLP* in two main aspects. On the one hand, the logic *HH* introduces new connectives which are not available in Horn Clause logic, such as disjunction, implication and universal quantifiers [36]. On the other hand, and following Saraswat [45], in the scheme $HH(\mathcal{C})$, the notion of constraint system is established in such a way that any $\mathcal{C}$ satisfying certain minimal conditions can be considered as a possible instance for the scheme. In [45], as minimal conditions the language of constraints incorporates $\wedge$ and $\exists$. However, particular constraint systems may include more logical symbols as $\forall$ and $\Rightarrow$, together with the corresponding assumptions related to their behavior. Therefore, the language of constraints itself extends the common ones used in *CLP*, consequently facilitating the representation of more complex constraints.

* Corresponding author.
E-mail address: garanda@fdi.ucm.es (G. Aranda-López).

1567-8326/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jlap.2013.07.002

This paper extends other works [38,2] in which we investigated the use of $HH(\mathcal{C})$ not as a (general purpose) programming language, but as the basis for constraint deductive database (*CDDB*) systems [30,42]. The motivation is that, in the same way that Datalog [51,56] and Datalog with constraints [27] arise for modeling database systems inspired by *LP* and *CLP* respectively, the language $HH(\mathcal{C})$ can offer a suitable starting point for the same purpose. We show that the expressive power of $HH(\mathcal{C})$ improves existing languages by enriching the mechanisms for database definition and querying, with new elements that are useful and natural in practice. In particular, implications can be used to write hypothetical queries, and universal quantification allows encapsulation. The existence of constraints is exploited to represent answers and to finitely model infinite databases and answers. This is also the case of constraint databases, but the syntax of our constraints is also more expressive than the one commonly used in them, as it is the case of Datalog with constraints.

However, $HH(\mathcal{C})$, as it was originally introduced, lacks negation which, as we will see, is needed for our proposal to be complete with respect to relational algebra (*RA*). We have extended $HH(\mathcal{C})$ with negation, to obtain $HH_\neg(\mathcal{C})$ to be used as a database language. We have defined a proof-theoretic semantics to provide the meaning of goals (queries) and programs (databases). This meaning is represented by answer constraints, which can be obtained using the goal-oriented rules of a sequent calculus which combines intuitionistic inference rules with deductibility in a generic constraint system. Also, a stratified fixpoint semantics has been defined and proved to be sound and complete with respect to the previous one. The motivation for introducing this new semantics take into account several aspects:

- The fixpoint semantics provides a model for the whole database, while the proof-theoretic one (as well as top-down semantics in general) focuses only on the meaning of a query in the context of a database. In fact, the fixpoint of a database will correspond to the instance of the database. Thereby, the fixpoint semantics supplies a framework in which properties such as equivalence of databases can be easily analyzed, which gives formal support to the study of query optimization.
- In order to deal with recursion and negation, we have followed the stratified negation approach used in [51] which gives semantics to Datalog. The use of a fixpoint semantics as an operational mechanism has been adopted as a good choice in several deductive database systems as it is able to avoid the non-monotonicity inherent to negation. Then it guarantees termination, as long as the constraint answer sets are finite. Further, it facilitates to work with different constraint systems, relegating the problem of termination to the problem of finding intensional representations of data by the constraint system. This issue is dealt in works as [40], where safety conditions are imposed to the constraint system, and some particular systems are identified as satisfying such conditions. Hence, termination for any query is ensured for these systems. But, identifying such particular systems is out of the scope of this work.
- Introducing negation in goals makes a given database to may have several meanings [51]. Stratified negation is one of the approaches that, by imposing syntactical restrictions, guarantees a unique model for the database: the minimal fixpoint interpretation. Moreover, stratification has been previously used as a useful resource when dealing with hypothetical queries in Datalog [9], in order to get a unique minimal model for the database, as it is the case of our proposal.
- Stratification is a common technique to deal with aggregates [42], because it ensures monotonicity. Our stratified design of the fixpoint semantics has become a good framework to implement aggregates.

In order to define a stratified fixpoint semantics for $HH_\neg(\mathcal{C})$, we have adapted the usual notion of dependency graphs to include the dependencies derived from implications inside goals, as well as those derived from aggregate functions. The fixpoint of a database is computed as a set of pairs (A, C), where A is an atom and C a constraint. The atom A can be understood as an n-ary relation instance, where its arguments are constrained by C. According to the dependency graph, predicates are classified by strata and these pairs are computed by strata. 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. From the theoretical point of view, this issue does not make any obstacle, but it can be a drawback for a concrete implementation. In our approach, the fixpoint of the augmented database must be locally computed to solve the implication. But our proposal takes advantage of the stratification to avoid cycles during the computation. In addition, the dependency graph can be useful to reduce this computation to the part of the database that is involved in the implication.

The fixpoint semantics provides support for a concrete database system. We have implemented a Prolog prototype very close to the underlying theory as a proof-of-concept. Essentially, it incorporates all the features introduced in the paper, and moreover it supports aggregate functions. Two main components can be distinguished in the implementation of this database language. One corresponds to the implementation of the fixpoint semantics which is independent of the concrete constraint system. The other component corresponds to the implementation of the constraint system. We have considered and implemented solvers for the following constraint systems: Boolean, Reals, and Finite Domains, as instances of $\mathcal{C}$ in the scheme $HH_\neg(\mathcal{C})$. The implementation is designed in such a way that more than one constraint system can be used within the same database. We have designed a type system for identifying the constraint system each constraint in a database belongs to.

1.1. Related work

It is well known that negation in logic programming is a difficult issue and there has been a great amount of work about it [1], and also in constraint logic programming [46] and deductive databases [8] since long time ago. A first bag of issues comes from deriving incorrect answers or failing to derive others in presence of classical negation. Several problems arise in this setting, as unsoundness of *SLDNF*, which is workarounded by restricting a negative goal to be selected until it becomes ground [34]. However, this introduces another problem: floundering [5], which is avoided in the field of deductive databases with safety conditions [51]. Safety conditions have been also applied to constraint deductive databases [7,40,41] for particular constraint systems, enabling to develop closed-form constraints as answers. The closed-form evaluation requirement guarantees that it is possible for the query solver to calculate intensional forms for the answer. In constraint databases, where (non-ground) intensional data are managed, constructive negation [29,14,16] can be used instead of such safety conditions. Its rationale lies on allowing non-ground negative goals, which can construct answers by involving constraints on goal variables, therefore avoiding floundering. This is the very fundamental of constraint databases [42], where an answer to a negated goal is a set of constraints. This idea was used in *CLP* [46] as an approach to constructive negation to avoid floundering. Our work can also be seen from this perspective because the answer to a negated goal is also constructive. Nevertheless, the proposal of [46] is based on classical logic, while our approach is based on intuitionistic logic including implication and universal quantification.

A second bag of issues comes from assigning a model to a program valid to the user under an intended semantics. As mentioned before, stratified negation guarantees that only one minimal model can be assigned to a program [51]. Other approaches are based on non-monotonic logic, as inflationary semantics [13], which is also based on a two-valued logic and assigning one model to a program. Its drawback is that, in general, that model is not a minimal one and inflationary model semantics does not always meet the intended semantics. Next approaches are based on three-valued logic and produce in general several outcome models:

Gelfond and Lifschitz proposed Stable Models [21], a declarative semantics for logic programs with negation, based on autoepistemic logic. Another related approach is the Well-Founded semantics defined by Van Gelder et al. [52], where the main idea is the notion of *unfounded set*, which provides the basis for obtaining negative information in the semantics. Answer Set Programming (*ASP*) is a form of declarative programming oriented towards difficult combinatorial search problems [32,20]. It is based on the work about Stable Models and fast propositional solvers are used as computational mechanism for inference. Its key idea is to use ground instances of programs. Our scheme $HH_{\neg}(\mathcal{C})$ is designed to work with any generic constraint language $\mathcal{L}_{\mathcal{C}}$ and the use of ground instances would impose a serious limitation on the constraint systems allowed as instances of the scheme. This technique would be adequate for a Herbrand constraint system, but they are unfeasible for more sophisticated constraints. Notice that $HH_{\neg}(\mathcal{C})$ constraint systems work with intensional representations that are able to be more general than a simple equality. For example, constraints over real numbers should be excluded as a possible instance, as no grounding is possible (at least in a straightforward way) and it would be a serious drawback for our proposal. However, although *ASP* includes constraints, they are viewed as integrity constraints which discards models in the answer rather than syntax objects which can be dealt as first citizen constructions *á la CLP*. In addition, neither implications nor quantifiers are supported. It might be expected that introducing the implication, which dynamically produces an increasing of the database, in a system based on the approaches just mentioned, additional computations would be necessary, as it happens in our system.

Although stratification imposes certain syntactic restrictions to the language, these conditions are usually adopted in deductive database systems, as Datalog, for practical reasons. Other works add different syntactic conditions as [4], where the notion of *guarded negation* is adapted to database queries both for SQL and Datalog languages in order to improve performance. In the case of Datalog, guarded negation introduces an additional syntactical condition to stratification. Although it is computationally well-behaved and subsumes several well-known query languages as unions of conjunctive queries, monadic Datalog and frontier-guarded tuple-generating dependencies, it is actually subsumed by stratified Datalog. As $HH_{\neg}(\mathcal{C})$, in turn, subsumes stratified Datalog, it also subsumes Datalog with guarded negation.

In addition, the syntactic limitations associated to the stratification approach can be overcome in practical situations of potentially non-stratifiable programs, which can be modeled by equivalent stratifiable databases. We illustrate this point by an example showed in [21] and also related in [52] as a classical example of non-stratifiable program, that can be tackled both with the Well-Founded semantics and Stable Models.

Example 1. In this example it is shown a general scheme for a two-people game with a finite space of states. This scheme allows to determine the winning states of the game (those that guarantee the victory for the player in turn) by means of one single clause reflecting a simple idea: one wins if the opponent cannot win because it cannot move. The clause is:

$$\forall x \forall y\ winning(x) \Leftarrow move(x, y) \wedge \neg winning(y),$$

where *move* is defined for the concrete game. This program is not stratifiable due the negative cycle in *winning*. Nevertheless, it is easy to see that the winning strategy is based on the parity of the number of movements. Using that fact, it is straightforward to encode this strategy as follows:

$$\forall x \forall y\ canMove(x) \Leftarrow move(x, y),$$

$$\forall x \forall y\ possibleWinning(x) \Leftarrow oddMove(x, y) \wedge \neg canMove(y),$$

$$\forall x \forall y\ winning(x) \Leftarrow move(x, y) \wedge \neg possibleWinning(y).$$

Here $oddMove(x, y)$ represents a change from state x to state y in an odd number of movements and it is defined as:

$$\forall x \forall y \; oddMove(x, y) \Leftarrow move(x, y),$$

$$\forall x \forall y \forall z_1 \forall z_2 \; oddMove(x, y) \Leftarrow move(x, z_1) \wedge move(z_1, z_2) \wedge oddMove(z_2, y).$$

This program represents a stratifiable database which is suitable for $HH_\neg(\mathcal{C})$, and even for Datalog. The definition of the predicate *oddMove* can be simplified in $HH_\neg(\mathcal{C})$, making use of constraints. In addition, our language combines implication, negation and recursion, so more complex queries can be formulated as, for instance:

$$move(a, b) \Rightarrow winning(x),$$

that asks for the winner positions, assuming the existence of the new movement $move(a, b)$. □

One of the main advantages of our language is the use of hypothetical queries, an unusual feature in a database language. However, there exist some works as Hypothetical Datalog [9,10], that is an extension of Horn-clause logic allowing hypothetical queries in a similar way to our proposal. Queries are allowed to include the local (hypothetical) addition and deletion of tuples from the database. In both, additions (`A` ← `B` [`add:C`]) and deletions (`A` ← `B` [`del:C`]), the atomic term `C` is temporarily added or deleted from the extensional database in order to solve the query `B`, which can be understood as a dynamic modification of the database, similarly to the $HH_\neg(\mathcal{C})$ behavior. Nevertheless, this is a very restrictive form of hypothetical queries, which implies a great simplification of the approach, as $HH_\neg(\mathcal{C})$ allows complete clauses as the antecedent in hypothetical queries, i.e., it allows to dynamically change the intensional database (which corresponds to the formalization within the intuitionistic logic). For instance, in the previous example, the query $move(a, b) \Rightarrow winning(x)$ can be formulated also in Hypothetical Datalog. But, it is not possible to assume a more general rule for the predicate *move*, that defines the possible movements of the game. In contrast, for instance, the formula $\forall x \forall y (2 * x \approx y \Rightarrow move(x, y)) \Rightarrow winning(z)$ is a valid query in $HH_\neg(\mathcal{C})$. Moreover, when comparing Hypothetical Datalog and its theoretical foundations to $HH_\neg(\mathcal{C})$, we must emphasize that our logic combines connectives and constraints. In [15] another approach for hypothetical queries, also including positive and negative hypotheses, is proposed, but neither negation nor implications are allowed in clauses.

Previously, we pointed out the difficulty of dealing with negation in logic programming. The semantics of negation is even more complex in the presence of implication in goals, as it is the case of Hypothetical Datalog. Additional obstacles arise when considering the logic *HH*, due to the inclusion of universal quantifiers. There exist several proposals aimed to introduce negation in *HH* (see for instance [24,37,17]). [17] is the first work introducing negation as failure into N-Prolog. In [37] a natural calculus is proposed for *HH* with negation. [24] is closer to our approach in the sense that a sequent calculus as well as a fixpoint semantics is defined for this logic. Instead of stratification, in order to preserve monotonicity, two forcing relations (positive and negative) are introduced, and in addition completely and incompletely defined predicates must be distinguished. In practice, these issues yield to a hard computation of the fixpoint in a concrete implementation.

With respect to constraint database systems, during the last couple of decades, constraint databases have received much attention because they are specially suited for applications subject to geometric interpretation, notably in geographic information systems (GIS), spatial databases and spatio-temporal databases [22,44,42,30]. Current trends are oriented to develop efficient operators and indexing of geometric data. Specific academic systems include MLPQ/GIS [28], DISCO [11] and PReSTO [43,12]. Output from this research was transferred to commercial systems and nowadays, several database vendors offer constraint-based databases as IBM DB2, MS SQL Server, Oracle, PostgreSQL, mainly for GIS (Geographic Information System) applications. However, as in relational algebra, negation only occurs in difference operators, so that negated predicates cannot be queried, as we do allow. Negation in the specific field of *CDDB* systems has been also studied [23,42]. In [23], different uses of negation are identified and the more general one requires delaying for constraints to become ground, which is a severe restriction. The treatment of negation of [42] corresponds to stratified Datalog for safe constraint queries. In our language, negation is even more complex due to the presence of implication and universal quantification in goals.

1.2. Contributions and relationship to prior work

In this paper we give a complete picture of the $HH_\neg(\mathcal{C})$ language as an expressive constraint deductive database language, including its theoretical foundation, as well as the description of a prototype system based on it. The paper provides an integrating view of previous works related to this language, [38,2], and extends them in several aspects:

- In [38] the programming scheme $HH_\neg(\mathcal{C})$ was formalized defining a proof-semantics and a stratified fixpoint semantics, that is sound and complete w.r.t. the former. Here, we show in detail these two formalizations which support the scheme, emphasizing the purpose of the fixpoint semantics as an operational semantics that guides the implementation. In addition, we include full proofs for the equivalence results.
- In [2] we presented a first Prolog prototype implementing that scheme, based on the fixpoint semantics, that is independent on the particular constraint system. This prototype has been enhanced with different improvements. In this paper we show a detailed description of the improved $HH_\neg(\mathcal{C})$ system, the way in which technical problems have been

solved, and some selected examples evincing the usefulness of the scheme. The most relevant features added to the system in the present work are:

- We have investigated how to incorporate aggregate functions to the language. The most usual aggregate operations are integrated in the language as part of the constraint system, in such a way that the computation of aggregate functions is delegated to the constraint solver. This is a non-trivial issue in our context that has been solved by taking advantage of our stratification and dependency graph notions. With this aim, the dependency graph specification has been enriched by adding dependencies due to aggregate operations.
- We have also improved the constraint systems, which are now able to deal with a limited form of constraint combination. Specifically, the system can deal with more complex constraints because the generic interface of the constraint solvers can identify constraint belonging to different domains, split them, and send each part of the initial constraint to its corresponding domain solver.
- We have improved the computation of queries, avoiding the recomputation of the whole database from scratch, in cases where stratification changes.
- Finally, we have design a better and more complete user interface, also enhancing its performance.

1.3. Organization of the paper

The rest of the paper is organized as follows. In Section 2, $HH_{\neg}(\mathcal{C})$ is informally presented as a query language by means of examples. In Section 3, the syntax of the language is formally introduced and concrete examples, which illustrate its potential and expressiveness, are shown. In Section 4, the proof-theoretic semantics for $HH_{\neg}(\mathcal{C})$ is defined as a foundation for the scheme. In Section 5, a fixpoint semantics, based on the notion of stratification, is presented and proved to be sound and complete with respect to the proof-theoretic semantics. Section 6 introduces a user-oriented description of the current system and the aggregate functions, and schematizes the computation stages of the system. Section 7 is devoted to the description of constraint domains and the implementation of the corresponding solvers. Aggregate functions are introduced as a part of the constraint language. Section 8 is an overview of the fixpoint semantics implementation, making emphasis on the implementation of the forcing relation which supports the semantics of $HH_{\neg}(\mathcal{C})$. In particular, the difficulties that have been overcome to implement the forcing of the implication are explained. Section 9 summarizes some conclusions and sketches future works. In Appendix A we include the full proofs for the results belonging to Section 4. Appendix B describes a form of the dependency graph needed to implement the forcing of the implication and constraints including aggregates. Finally, in Appendix C, the implementation of the constraint systems is provided, and Appendix D includes implementation details for the forcing relation.

2. $HH(\mathcal{C})$ as a database language: A first glance

This section is devoted to informally present $HH(\mathcal{C})$ as a suitable language for constraint databases. In our system, a database is a logic program: a set of clauses. Facts (ground atoms) define the extensional database, and clauses with body define the intensional database. The last ones 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 (query) from a program (database), and the answer is a constraint.

2.1. Relations and predicates

The relational model deals with (finite) relations which can be defined both extensionally, as sets of tuples, and intensionally, by means of views. A relation has a name, an arity, and its meaning may be understood as a set of tuples. As well, a predicate has a name, an arity, and its meaning can be understood as a set of constraints over its arguments. Predicates can also be defined both extensionally, by means of facts, and intensionally, by means of clauses.

Example 2. Fig. 1 defines some relations extensionally (*client* and *mortgageQuote*), and intensionally (*accounting*) both in *RA* and in $HH(\mathcal{C})$. Extensional relations are defined as tables in the relational model in (a) and as extensional predicates in $HH(\mathcal{C})$ (facts of (c)). The relation *accounting* is defined as a view in the relational model in (b), and as an intensional predicate in $HH(\mathcal{C})$ (clauses with a non-empty right-hand side of (c)).

In *RA* the result of computing the view *accounting* is the relation:

name	*salary*	*quote*
brown	1500	400
mcandrew	3000	100

accounting

In $HH(\mathcal{C})$ this query corresponds to the computation of the goal $accounting(n, s, q)$ whose answer is $(n \approx brown \wedge s \approx 1500 \wedge q \approx 400) \vee (n \approx mcandrew \wedge s \approx 3000 \wedge q \approx 100)$. This introductory example shows some relations that could also be

(a) Relations extensionally defined as relational tables:

name	*balance*	*salary*
smith	2000	1200
brown	1000	1500
mcandrew	5300	3000

client

name	*quote*
brown	400
mcandrew	100

mortgageQuote

(b) A relation defined as a relational view:

$$accounting \leftarrow \pi_{name,salary,quote}(\sigma_{quote \geqslant 100}(client \bowtie mortgageQuote))$$

(c) The above relations defined as an $HH(\mathcal{C})$ program:

$$\begin{array}{ll} client(smith, 2000, 1200). & mortgageQuote(brown, 400). \\ client(brown, 1000, 1500). & mortgageQuote(mcandrew, 100). \\ client(mcandrew, 5300, 3000). & \end{array}$$

$$\begin{array}{l} \forall name \forall salary \forall quote \forall balance (accounting(name, salary, quote) \Leftarrow \\ \qquad client(name, balance, salary) \wedge \\ \qquad mortgageQuote(name, quote) \wedge \\ \qquad quote \geqslant 100). \end{array}$$

Fig. 1. Relations vs. $HH(\mathcal{C})$ predicates.

computed in Datalog with constraints [27], but it will be extended later in Example 6 with some new relations that exceed the capabilities of such a system. □

2.2. 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 [30] have inherited this feature. We illustrate this point with the following example.

Example 3. 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 which 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 expressed by the next clause:

$$\bar{\forall} rectangle(x_1, y_1, x_2, y_2, x, y) \Leftarrow x \geqslant x_1 \wedge x \leqslant x_2 \wedge y \geqslant y_1 \wedge y \leqslant y_2,$$

where $\bar{\forall}$ represents the universal closure of a formula. Analogously, $\bar{\exists}$ will represent the existential closure.

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 it possible to manage (potentially) infinite sets of data.

The goal $rectangle(0, 0, 4, 4, x, y) \wedge rectangle(1, 1, 5, 5, x, y)$ computes the intersection of two rectangles and an answer can be represented by the constraint:

$$(x \geqslant 1) \wedge (x \leqslant 4) \wedge (y \geqslant 1) \wedge (y \leqslant 4).$$

A circle can be defined by its center and radius, using non-linear constraints now:

$$\bar{\forall} circle(xc, yc, r, x, y) \Leftarrow (x - xc)^2 + (y - yc)^2 \leqslant r^2.$$

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:

$$\bar{\forall}\big(x^2 + y^2 \approx 1 \Rightarrow circle(0, 0, 2, x, y)\big).$$

This goal is not expressible in standard deductive database languages because, in addition to constraints, it involves universal quantifiers and implication. Even Hypothetical Datalog cannot deal with this goal due to the universal quantifiers. These components constitute a big step in expressivity. □

Since $HH(\mathcal{C})$ does not support negation, it is not still complete w.r.t. *RA* as we show next.

- Projection. $E = \pi_{i_1}, \ldots, \pi_{i_k}(E_1)$

$\bar{\forall} e(x_{i_1}, \ldots, x_{i_k}) \Leftarrow e_1(x_1, \ldots, x_n)$.

- Selection. $E = \sigma_{t_1 \theta t_2}(E_1)$

$\bar{\forall} e(x_1, \ldots, x_n) \Leftarrow e_1(x_1, \ldots, x_n) \wedge C_\theta$.

- Cartesian product. $E = E_1 \times E_2$

$\bar{\forall} e(x_1, \ldots, x_n, x_{n+1}, \ldots, x_m) \Leftarrow e_1(x_1, \ldots, x_n) \wedge e_2(x_{n+1}, \ldots, x_m)$.

- Set union. $E = E_1 \cup E_2$

$\bar{\forall} e(x_1, \ldots, x_n) \Leftarrow e_1(x_1, \ldots, x_n) \vee e_2(x_1, \ldots, x_n)$.

- Set difference. $E = E_1 - E_2$

$\bar{\forall} e(x_1, \ldots, x_n) \Leftarrow e_1(x_1, \ldots, x_n) \wedge \neg e_2(x_1, \ldots, x_n)$.

E and E_i (resp.) are relational expressions represented as e and e_i (resp.) predicates. C_θ is the constraint corresponding to the condition $t_1 \theta t_2$.

Fig. 2. Relational operators as $HH\neg(\mathcal{C})$ programs.

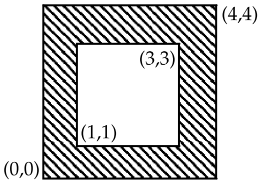

Fig. 3. Regions in the plane.

2.3. Need for negation

What a database user might want is to have the basic relational operations available in this language. In fact, a database language is complete w.r.t. *RA* if these operations can be expressed within the language. As it is shown in Fig. 2, $HH(\mathcal{C})$ can express projection, Cartesian product, union, and selection. For the last one, it is required that the constraint system $\mathcal{C}$ incorporates (or can express) the operators θ, in order to build the corresponding constraint C_θ. For instance, $\sigma_{\$i \leqslant \$j}$ corresponds to $x_i \leqslant x_j$. As we will see in Section 3.1, any constraint system in our scheme satisfies this requirement. But expressing set difference needs some kind of negation. So we have added the connective $\neg$ to $HH(\mathcal{C})$, obtaining a complete database language which will be formalized in the next section. There are some other situations, besides relational database requirements, in which negation is needed.

Example 4. Returning to Example 3, we define the dashed frame depicted in Fig. 3 by the inner region of a large rectangle and the outer region of a small rectangle with the goal

$$rectangle(0, 0, 4, 4, x, y) \wedge \neg rectangle(1, 1, 3, 3, x, y),$$

and an answer can be represented by the constraint:

$$(y > 3 \wedge y \leqslant 4 \wedge x \geqslant 0 \wedge x \leqslant 4) \vee (y \geqslant 0 \wedge y < 1 \wedge x \geqslant 0 \wedge x \leqslant 4) \vee$$
$$(y \geqslant 0 \wedge y \leqslant 4 \wedge x > 3 \wedge x \leqslant 4) \vee (y \geqslant 0 \wedge y \leqslant 4 \wedge x \geqslant 0 \wedge x < 1).$$

In this example, we assume that negation can be effectively handled by the constraint solver, an issue addressed later in this paper. □

3. The language $HH_\neg(\mathcal{C})$

The formalisms which $HH(\mathcal{C})$ is founded on [31,19] do not support any kind of negation, so the language is not expressive enough in the field of database systems. We have extended $HH(\mathcal{C})$ including negation to obtain a *CDDB* language which is complete w.r.t. *RA*. In this section, we make precise the syntax of the formulas of $HH(\mathcal{C})$ extended with negation, denoted by $HH_\neg(\mathcal{C})$; next we introduce more database examples.

3.1. Syntax

As usual, to build the syntactic objects of the logic, we consider a set of variables and a signature containing:

- defined predicate symbols, representing the names of database relations, to build atoms,
- non-defined (built-in) predicate symbols, including at least the comparison $\leqslant$ and the equality predicate symbol $\approx$, to build atomic constraints, and
- constant and operator symbols, which depend on the particular constraint system, to build terms.

Since our interest is to represent databases, we only use finite signatures.

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:

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

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

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, and t_i are terms; C represents a constraint. The incorporation of negated atoms in goals is the addition to $HH(\mathcal{C})$. Negation is not allowed in the head of a clause, but inside its body.

3.1.1. The constraint system $\mathcal{C}$

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 to $\mathcal{C}$ to be a constraint system:

- $\mathcal{L}_{\mathcal{C}}$ contains at least every first-order formula built up using:
  - $\top$ (*true*), $\bot$ (*false*),
  - built-in predicate symbols,
  - the connectives $\wedge, \neg$, and the existential quantifier $\exists$.
- Regarding to $\vdash_{\mathcal{C}}$:
  - It includes inference logic rules for the considered connectives and quantifiers.
  - It is *compact*, i.e., $\Gamma \vdash_{\mathcal{C}} C$ implies that there exists a finite set $\Gamma' \subseteq \Gamma$, such that $\Gamma' \vdash_{\mathcal{C}} C$.
  - It is *closed under substitution*, i.e., $\Gamma \vdash_{\mathcal{C}} C$ implies $\Gamma\sigma \vdash_{\mathcal{C}} C\sigma$ for every substitution σ.

Let us remark that $\mathcal{C}$ is required to deal with negation, because the incorporation of the connective $\neg$ to the language *HH* yields to the need for incorporating the negation in the constraint system, which has the responsibility of checking the satisfiability of answers in the constraint domain.

We say that a constraint C is $\mathcal{C}$-satisfiable if $\emptyset \vdash_{\mathcal{C}} \bar{\exists} C$, where $\bar{\exists} C$ 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 constraint systems of the previous examples verify the required minimal conditions aforementioned. Moreover, they also include the connective $\vee$ (as usual), constants to represent numbers and names, arithmetical operators, and more built-in predicates $(>, \geqslant, \ldots)$.

For instance, for the constraint system $\mathcal{R}$ of Real-closed Fields, $\mathcal{L}_{\mathcal{R}}$ is a first-order language with all classical logical connectives including negation, and $\Gamma \vdash_{\mathcal{R}} C$ holds when $Ax_{\mathcal{R}} \cup \Gamma \vdash_{\approx} C$, where $Ax_{\mathcal{R}}$ is Tarski's axiomatization of the real numbers, and $\vdash_{\approx}$ is the entailment relation of classical logic with equality. An example of a concrete constraint is $\neg(x \approx 0.2)$, also written as $x \not\approx 0.2$ for the sake of simplicity.

It is easy to see that every formula allowed in the selection operation of *RA* can be expressed with an equivalent constraint, since for any $\mathcal{C}$, the language $\mathcal{L}_{\mathcal{C}}$ contains the built-in predicates $\leqslant$ and $\approx$, and the connectives $\wedge$ and $\neg$.

3.1.2. $HH\neg(\mathcal{C})$ programs

Programs, denoted by Δ, are sets of clauses and represent databases. As usual in Logic Programming, they can still be viewed as sets of implicative formulas with atomic head, 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:

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

So, elaborated clauses are formulas of the form $\forall x_1 \ldots \forall x_n (G \Rightarrow A)$ (or simply $\forall \bar{x}(G \Rightarrow A)$), but notice that clauses inside G are not required to be elaborated. The use of the elaborated form, instead of general $HH\neg(\mathcal{C})$ clauses, has some practical benefits:

- It permits to specify a database view that defines a predicate (database relation) p by means of a set of elaborated clauses whose heads are atoms beginning with the predicate symbol p, as it is done in logic programs with Horn clauses.

- It permits to define a calculus governing $HH\neg(\mathcal{C})$ without rules introducing connectives in the left, providing the uniformity property of the calculus, which guarantees completeness of goal-oriented search for proofs. Notice that in the calculus $\mathcal{UC}_\neg$ (introduced in Section 4) there is only the rule (*Clause*) to deal with atomic goals, that corresponds to the SLD-Resolution rule of logic programming.
- It facilitates the formalization of the fixpoint operator used to define the fixpoint semantics introduced in Section 5.2.

For convenience, we will also use the common notation $\forall x_1 \ldots \forall x_n(A \Leftarrow G)$ as in previous examples.

3.2. Examples of $HH_\neg(\mathcal{C})$

Once we have formalized the syntax of our language, we introduce more examples showing the advantages of our proposal w.r.t. other common database languages. As an important benefit of our approach, we stress the ability to formulate hypothetical and universally quantified queries. In addition, variables can be explicitly and existentially quantified in queries avoiding the computation of an explicit answer for these variables.

In these examples, the instance $HH_\neg(\mathcal{FR})$ is used, where $\mathcal{FR}$ is a hybrid constraint system which 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 [18], a hybrid constraint system subsuming $\mathcal{FR}$ is presented.

Example 5. Consider the following travel database. The predicate *flight*(*Origin*, *Destination*, *Time*) represents an extensional database relation of direct flights from *Origin* to *Destination* and duration *Time*:

$$flight(mad, par, 1.5),$$
$$flight(par, ny, 10),$$
$$flight(london, ny, 9).$$

In turn, *travel*(*Origin*, *Destination*, *Time*) represents an intensional database relation, expressing that it is possible to travel from *Origin* to *Destination* in a time greater or equal than *Time*, possibly concatenating some flights:

$$\bar{\forall} travel(x, y, t) \Leftarrow flight(x, y, w) \wedge t \geqslant w,$$
$$\bar{\forall} travel(x, y, t) \Leftarrow flight(x, z, t_1) \wedge travel(z, y, t_2) \wedge t \geqslant t_1 + t_2.$$

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 $11 \geqslant t + 9$ which is $\mathcal{FR}$-equivalent to the final answer $t \leqslant 2$.

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

We can also compare $HH_\neg(\mathcal{C})$ to relational calculus (whose underlying logic is richer than the used on implemented *CDDB* languages). For instance, the query $\neg(\exists t\ flight(x, y, t)) \wedge x \not\approx y$, or its equivalent $(\forall t \neg flight(x, y, t)) \wedge x \not\approx y$, which represents the cities in the database that have no direct flights between them, is not safe in the domain relational calculus, because it contains a negated formula whose free variables are not limited. This problem is avoided in our system because formulas are interpreted in the context of the constraint domain of the particular instance and no test for this kind of safety is needed. In fact, $(\forall t \neg flight(x, y, t)) \wedge x \not\approx y$ represents a valid $HH_\neg(\mathcal{F}R)$ query, which has as answer constraint:

$$(x \not\approx mad \vee y \not\approx par) \wedge (x \not\approx par \vee y \not\approx ny) \wedge (x \not\approx lon \vee y \not\approx ny)$$

in the domain of the cities registered in the current database. However, the query is not allowed in Datalog with constraints due, in this case, to the quantifier occurrence.

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

$$\bar{\forall} deltravel(x, y, t) \Leftarrow flight(x, y, t_1) \wedge delay(x, y, t_2) \wedge t \geqslant t_1 + t_2,$$
$$\bar{\forall} deltravel(x, y, t) \Leftarrow flight(x, z, t_1) \wedge delay(x, z, t_2) \wedge deltravel(z, t, t_3) \wedge t \geqslant t_1 + t_2 + t_3.$$

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

$$\forall x\big(delay(par, x, 1) \wedge delay(mad, par, 0.5)\big) \Rightarrow deltravel(mad, ny, t)$$

represents the query: What is the time needed to travel from Madrid to New York assuming that for any destination there is a delay of one hour from Paris, and the flight from Madrid to Paris is half an hour delayed? According to its proof-theoretic interpretation, the clause $\forall x(delay(par, x, 1) \wedge delay(mad, par, 0.5))$ will be added locally to the database to solve the goal $deltravel(mad, ny, t)$, and it will be discarded after the computation as they are hypothetical assumptions. 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:

$$\bar{\forall} trip(x, y, t) \Leftarrow nondeltravel(x, y, t) \vee deltravel(x, y.t),$$

$$\bar{\forall} nondeltravel(x, y, t) \Leftarrow travel(x, y, t) \wedge \neg delayed(x, y),$$

$$\forall x \forall y\ delayed(x, y) \Leftarrow \exists t\ deltravel(x, y, t).$$

Notice that the last formula is equivalent to $\bar{\forall} delayed(x, y) \Leftarrow deltravel(x, y, t)$. Since explicit existential quantifiers are allowed in $HH_{\neg}(\mathcal{C})$, they can also be used to improve readability and facilitate the specification of some predicates. □

Example 6. In this example we extend the database for a bank introduced in Example 2. The extensional database is given by facts for the relations $client(Name, Balance, Salary)$, $mortgageQuote(Name, Quote)$, $pastDue(Name, Amount)$ and $branch(Office, Name)$ as follows:

$client(smith, 2000, 1200).$ $\quad mortgageQuote(brown, 400).$

$client(brown, 1000, 1500).$ $\quad mortgageQuote(mcandrew, 100).$

$client(mcandrew, 5300, 3000).$

$pastDue(smith, 3000).$ $\quad branch(lon, smith).$

$pastDue(mcandrew, 100).$ $\quad branch(mad, brown).$

$branch(par, mcandrew).$

As an additional restriction we assume that each client has, at most, one mortgage quote. Next, we introduce some views defining the intensional part of the database. The first one captures the clients that have a mortgage: a client has a mortgage if there exists a mortgage quote associated to him:

$$\bar{\forall} hasMortgage(x) \Leftarrow mortgageQuote(x, y).$$

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

$$\bar{\forall} debtor(x) \Leftarrow client(x, y, z) \wedge pastDue(x, w) \wedge w > y.$$

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

$$\bar{\forall} interestRate(x, 2) \Leftarrow client(x, y, z) \wedge y < 1200,$$

$$\bar{\forall} interestRate(x, 5) \Leftarrow client(x, y, z) \wedge y \geqslant 1200.$$

The next relation $newMortgage(Name, Quote)$ specifies that a non-debtor client $Name$ can be given a new mortgage with $Quote$ in two situations. First, if he has no mortgage, a mortgage quote smaller than 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:

$$\bar{\forall} newMortgage(x, w) \Leftarrow client(x, y, z) \wedge \neg debtor(x) \wedge \neg hasMortgage(x) \wedge w \leqslant 0.4 * z,$$

$$\bar{\forall} newMortgage(x, w) \Leftarrow client(x, y, z) \wedge \neg debtor(x) \wedge mortgageQuote\big(x, w'\big) \wedge w + w' \leqslant 0.4 * z,$$

$$\bar{\forall} gotMortgage(x) \Leftarrow newMortgage(x, w).$$

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 such a client does not satisfy that requirements, the amount must be between 6000 and 20,000. The relation $personalCredit(Name, Amount)$ formalizes these conditions:

$$\bar{\forall} personalCredit(x, y) \Leftarrow \big(gotMortgage(x) \wedge y < 6000\big) \vee \big(\neg gotMortgage(x) \wedge y \geqslant 6000 \wedge y < 20{,}000\big).$$

Moreover, it is possible to define a view with the quote and the salary of clients whose mortgage quote is greater than 100 with the following relation *accounting*(*Name*, *Salary*, *Quote*) which corresponds to the predicate *accounting* of Example 2:

$$\bar{\forall} accounting(x, z, w) \Leftarrow client(x, y, z) \wedge mortgageQuote(x, w) \wedge w \geqslant 100.$$

The previous predicates define the database that we are going to use for illustrating some queries. As a first example, we can query whether every client is a debtor:

$$\forall x debtor(x),$$

for which the answer is $\bot$.

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

$$\exists x \exists y\ debtor(x) \wedge pastDue(x, y) \wedge y > 1000,$$

and the answer is $\top$. Note that we are using quantifiers for variables x and y, meaning that there are no explicit conditions over them. Otherwise, the answer will be a constraint over such variables.

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

$$\forall x \exists y \exists z \bigl(client(x, y, z) \Rightarrow \bigl(y > 2000 \Rightarrow interestRate(x, w)\bigr)\bigr).$$

Here we are using nested implication to formulate a hypothetical query. The answer is the constraint $w \approx 5$.

The next query involves negation and can be read as: which clients can get a mortgage quote of 400 but *not* a personal credit?

$$newMortgage(x, 400) \wedge \neg personalCredit(x, y),$$

and the answer is the constraint $x \approx mcandrew \wedge y \geqslant 6000 \wedge y < 20{,}000$, which means that it is possible to give a new mortgage to the client McAndrew, but it is *not* allowed to give him a personal credit of an amount between 6000 and 20,000. □

4. Proof-theoretic semantics

Several kinds of semantics have been defined for $HH(\mathcal{C})$ without negation, including proof-theoretic, operational [31] and fixpoint semantics [19], as well as for its higher-order version [33]. The proof-theoretic and fixpoint approaches have been adapted in order to formalize the extension $HH_{\neg}(\mathcal{C})$. In addition we have proven that they are equivalent.

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 which governs the underlying logic. This proof-system, called $\mathcal{UC}_{\neg}$ (Uniform calculus handling Constraints and negation) is a sequent calculus that combines traditional inference rules with the entailment relation $\vdash_{\mathcal{C}}$ of the generic constraint system $\mathcal{C}$.

Sequents have the form $\Delta; \Gamma \vdash G$, where programs Δ and sets of constraints Γ are on the left, and goals on the right. The notation $\Delta; \Gamma \vdash_{\mathcal{UC}_{\neg}} G$ means that the sequent $\Delta; \Gamma \vdash G$ has a proof using the rules of $\mathcal{UC}_{\neg}$. A proof of a sequent is a finite tree whose root is the sequent to be proved and the nodes are sequents. The rules regulate relationship between child nodes and parent nodes and the leaves are nodes of the form $\Gamma \vdash_{\mathcal{C}} C$. If $\Delta; C \vdash_{\mathcal{UC}_{\neg}} G$, then C is called an *answer constraint* to the query G in the database Δ, that can be understood as the meaning of the query G formulated to the database Δ. The idea is that G is true for Δ if the constraint C is satisfied. $\mathcal{UC}_{\neg}$ carries out only uniform proofs in the sense defined by Miller et. al. [35], i.e., goal-oriented proofs. The rules are applied backwards and, at any step, only one rule of the calculus can be applied, that is the corresponding to the structure of the goal. Notice that we are assuming that any constraint will be treated as a whole, and the only applicable rule in this case is (C). (*Clause*) is used for atoms beginning with defined predicate symbols, the rest of the rules correspond to the outermost connective/quantifier of the goal (non-constraint) to be proved.

This proof system is an extension of the calculus $\mathcal{UC}$, introduced in [31] which provided proof-theoretic semantics for $HH(\mathcal{C})$. The incorporation of negation to the language makes it necessary to extend the notion of derivability, because there is no rule for this connective in $\mathcal{UC}$. Therefore, $\mathcal{UC}_{\neg}$ incorporates a new rule $(\neg)$ to formalize derivability of negated atoms. The rules defining the extended calculus appear in Fig. 4.

Next, we explain the rules $(\exists)$, (*Clause*) and $(\neg)$, the others correspond to widespread intuitionistic rules introducing connectives on the right of the sequent (see, e.g., [35]), except (C) which deals with goals that are pure constraints.

$(\exists)$ captures the fact that the witness in the proof of an existentially quantified formula can be represented by a constraint which can be more general than an equality $x \approx t$ simulating a substitution (e.g., $(x * x \approx 2)$ represents the witness $\sqrt{2}$, which cannot be written as a term).

(*Clause*) represents backchaining and allows one to prove an atomic goal $A \equiv p(t_1, \ldots, t_n)$, where p is a defined predicate symbol, using a program clause whose head $A' \equiv p(t'_1, \ldots, t'_n)$ is not required to unify with A, but rather solving a new

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

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

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

$$\dfrac{\Delta, D; \Gamma \ \vdash \ G}{\Delta; \Gamma \ \vdash \ D \Rightarrow G} \ (\Rightarrow) \qquad \dfrac{\Delta; \Gamma, C \ \vdash \ G}{\Delta; \Gamma \ \vdash \ C \Rightarrow G} \ (\Rightarrow_{\mathcal{C}})$$

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

$$\dfrac{\Gamma \ \vdash_{\mathcal{C}} \ \neg C \ \text{ for every } \Delta; C \ \vdash \ A}{\Delta; \Gamma \ \vdash \ \neg A} \ (\neg)$$

$(*)$ $x_1, \ldots, x_n$ fresh for A
$(**)$ y fresh for the formulas in the conclusion of the rule

Fig. 4. Rules of the sequent calculus $\mathcal{UC}_\neg$.

existentially quantified goal that, by applying the $(\exists)$ rule, will result in a search for a constraint that implies the equality $A' \approx A$ (this stands for $t'_1 \approx t_1 \wedge \cdots \wedge t'_n \approx t_n$).

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 $(\neg)$. We say that $(\neg)$ is a *metarule* since its premise considers any derivation $\Delta; C \ \vdash \ A$ of the atom A. In practice, there is a derivation of $\neg A$ when the set of answer constraints of A from Δ is finite.

Next we show two examples of proof derivation trees.

Example 7. Consider a fragment of the travel database of Example 5.

$$\begin{aligned}\text{Let } \Delta = \big\{ & \mathit{flight}(\mathit{mad}, \mathit{par}, 1.5),\ \mathit{flight}(\mathit{par}, \mathit{ny}, 10),\ \mathit{flight}(\mathit{lon}, \mathit{ny}, 9), \\ & \bar{\forall}\big((\mathit{flight}(x, y, w) \wedge t \geqslant w\big) \Rightarrow \mathit{travel}(x, y, t)\big), \\ & \bar{\forall}\big((\mathit{flight}(x, z, t_1) \wedge \mathit{travel}(z, y, t_2) \wedge t \geqslant t_1 + t_2\big) \Rightarrow \mathit{travel}(x, y, t)\big)\big\}\end{aligned}$$

and $G \equiv \forall t(t > 1.5 \Rightarrow \exists y \ \mathit{travel}(\mathit{mad}, y, t))$.

The following is a derivation of the sequent $\Delta; \{\top\} \ \vdash \ G$. We use the abbreviations $\Gamma = \{\top, t > 1.5, y \approx \mathit{par}\}$ and $\Gamma' = \Gamma \cup \{x' \approx \mathit{mad}, y' \approx \mathit{par}, t' \approx t, w' \approx 1.5\}$. $(\exists^4)$ denotes 4 successive applications of the rule $(\exists)$, the four corresponding side conditions referring to $\vdash_{\mathcal{FR}}$ are put together, and abbreviated as $\Gamma \vdash_{\mathcal{FR}} \exists x'(x' \approx \mathit{mad}) \ldots \Gamma' \vdash_{\mathcal{FR}} \exists w'(w' \approx 1.5)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{\Gamma' \vdash_{\mathcal{FR}} x' \approx \mathit{mad} \wedge \cdots \wedge t' \geqslant w'}{\Delta; \Gamma' \vdash x' \approx \mathit{mad} \wedge \cdots \wedge t' \geqslant w'} \ (C) \quad \dfrac{\dfrac{\Gamma' \vdash_{\mathcal{FR}} x' \approx \mathit{mad} \wedge y' \approx \mathit{par} \wedge w' \approx 1.5}{\Delta; \Gamma' \vdash x' \approx \mathit{mad} \wedge y' \approx \mathit{par} \wedge w' \approx 1.5} \ (C)}{\Delta; \Gamma' \vdash \mathit{flight}(x', y', w')} \ (\mathit{Clause})}{\Delta; \Gamma' \ \vdash \ (x' \approx \mathit{mad} \wedge \cdots \wedge t' \geqslant w') \wedge \mathit{flight}(x', y', w')} \ (\wedge) \\ \Gamma \vdash_{\mathcal{FR}} \ \exists x'(x' \approx \mathit{mad}) \ldots \Gamma' \vdash_{\mathcal{FR}} \ \exists w'(w' \approx 1.5)\end{array}}{\begin{array}{c}\Delta; \Gamma \ \vdash \ \exists x' \exists y' \exists t' \exists w'(x' \approx \mathit{mad} \wedge y' \approx y \wedge \\ t \approx t' \wedge t' \geqslant w' \wedge \mathit{flight}(x', y', w'))\end{array}} \ (\exists^4)}{\Delta; \{t > 1.5, y \approx \mathit{par}\} \vdash \mathit{travel}(\mathit{mad}, y, t)} \ (\mathit{Clause}) \qquad \{t > 1.5\} \vdash_{\mathcal{FR}} \ \exists y(y \approx \mathit{par})}{\dfrac{\dfrac{\Delta; \{t > 1.5\} \vdash \exists y \ \mathit{travel}(\mathit{mad}, y, t)}{\Delta; \{\top\} \vdash t > 1.5 \Rightarrow \exists y \ \mathit{travel}(\mathit{mad}, y, t)} \ (\Rightarrow_{\mathcal{C}})}{\Delta; \{\top\} \vdash \forall t \ (t > 1.5 \Rightarrow \exists y \ \mathit{travel}(\mathit{mad}, y, t))} \ (\forall)} \ (\exists)$$

□

Example 8. Recall Examples 3 and 4. Let Δ be the set:

$$\big\{\bar{\forall}\big(x \geqslant x_1 \wedge x \leqslant x_2 \wedge y \geqslant y_1 \wedge y \leqslant y_2 \Rightarrow \mathit{rectangle}(x_1, y_1, x_2, y_2, x, y)\big)\big\},$$

and $G \equiv \mathit{rectangle}(0, 0, 4, 4, x, y), \neg\mathit{rectangle}(1, 1, 3, 3, x, y)$. The answer constraint:

$$\begin{aligned}C \equiv\ & \big((y > 3) \wedge (y \leqslant 4) \wedge (x \geqslant 0) \wedge (x \leqslant 4)\big) \vee \\ & \big((y \geqslant 0) \wedge (y < 1) \wedge (x \geqslant 0) \wedge (x \leqslant 4)\big) \vee \\ & \big((y \geqslant 0) \wedge (y \leqslant 4) \wedge (x > 3) \wedge (x \leqslant 4)\big) \vee \\ & \big((y \geqslant 0) \wedge (y \leqslant 4) \wedge (x \geqslant 0) \wedge (x < 1)\big)\end{aligned}$$

can be obtained by the following deduction:

$$\dfrac{\dfrac{\dfrac{C \vdash_{\mathcal{R}} \exists a_1 \exists a_2 \exists b_1 \exists b_2 \exists x_1 \exists y_1 (a_1 \approx 0 \wedge x_1 \approx x \wedge \cdots)}{\begin{array}{l}\Delta; C \vdash \exists a_1 \exists a_2 \exists b_1 \exists b_2 \exists x_1 \exists y_1 (a_1 \approx 0 \wedge x_1 \approx x \wedge x_1 \geqslant a_1 \wedge \\ \quad a_2 \approx 0 \wedge y_1 \approx y \wedge x_1 \leqslant b_1 \wedge b_1 \approx 4 \wedge y_1 \geqslant a_2 \wedge b_2 \approx 4 \wedge y_1 \leqslant b_2)\end{array}}\,(C)}{\Delta; C \vdash \mathit{rectangle}(0, 0, 4, 4, x, y)}\,(\mathit{Clause}) \quad \mathbf{D}}{\Delta; C \vdash \mathit{rectangle}(0, 0, 4, 4, x, y) \wedge \neg \mathit{rectangle}(1, 1, 3, 3, x, y)}\,(\wedge)$$

where **D** is a deduction for $\Delta; C \vdash \neg \mathit{rectangle}(1, 1, 3, 3, x, y)$ whose last steps have the form:

$$\dfrac{C \vdash_{\mathcal{R}} \neg(\begin{array}{l} x \geqslant 1 \wedge y \geqslant 1 \wedge \\ x \leqslant 3 \wedge y \leqslant 3)\end{array} \quad \dfrac{\langle \mathit{rest\ of\ derivation} \rangle}{\Delta;\ \begin{array}{l} x \geqslant 1 \wedge y \geqslant 1 \wedge \\ x \leqslant 3 \wedge y \leqslant 3 \end{array} \vdash \mathit{rectangle}(1, 1, 3, 3, x, y)}}{\Delta; C \vdash \neg \mathit{rectangle}(1, 1, 3, 3, x, y)}\,(\neg) \qquad \square$$

In order to define a sound and complete goal solving procedure, some finiteness conditions must be imposed to make viable the metarule ($\neg$). That is, it has to be guaranteed that the set of answer constraints for an atom (that occurs negated in a goal) is finite, and that this set can be computed in a finite number of steps. As usual in the constraint database field, finiteness of the set of computed answers can be ensured by imposing different safety conditions to the constraint systems [42]. A technique that guarantees termination, provided finiteness of the constraint answers sets, is stratification.

We have adopted it because it is easy to combine with our notion of constraint system, giving a clear operational semantics to the scheme $HH_{\neg}(\mathcal{C})$ by providing meaning to the whole database (in the presence of safety conditions).

The stratified negation that we propose is widely explained in the next section, where a stratified fixpoint semantics is presented as the basis of an implementation of $HH_{\neg}(\mathcal{C})$.

5. Fixpoint semantics

We have extended and adapted the semantics presented in [19] in order to interpret full $HH_{\neg}(\mathcal{C})$ using a stratification technique. The semantics defined was based on a *forcing relation* among programs, sets of constraints and goals that states whether an interpretation makes true a goal G in the context $\langle \Delta, \varGamma \rangle$ of a program Δ and a set of constraints $\varGamma$. Interpretations were defined as functions able to give meaning to every pair $\langle \Delta, \varGamma \rangle$ as sets of atoms. The interpretation had to depend on this context because, when computing implicative goals, Δ or $\varGamma$ may be augmented. Here, interpretations are defined as functions able to give meaning to a database as a set of pairs (*Atom*, *Constraint*), and are classified on strata. Following [51], the stratification of a database is based on the definition of a dependency graph. Next we introduce these notions for our language.

5.1. Stratification and dependency graph

Given a set of clauses and goals $\varPhi$, the corresponding dependency graph $DG_{\varPhi}$ is a directed graph whose nodes are the defined predicate symbols in $\varPhi$, and the edges are determined by the implication symbols of the formulas.

Here, we adapt those notions as a useful starting point of a fixpoint semantics for our language. But now, 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 $F_1 \Rightarrow F_2$ produces edges in the graph from the defined predicate symbols inside F_1 to every defined predicate symbol inside F_2. An edge will be negatively labeled when the corresponding atom occurs negated on the left of the implication. Since constraints do not include defined predicate symbols, they cannot produce dependencies.

Example 9. Let Δ be the bank database of Example 6. Fig. 5 shows the dependency graph for Δ (the predicate *branch* corresponds to an isolated node which is not represented in the figure). Negative edges are labeled with $\neg$. □

Definition 1. Given a set of formulas $\varPhi$, its corresponding dependency graph $DG_{\varPhi}$, and two predicates p and q, we say:

- q *depends* on p if there is a path from p to q in $DG_{\varPhi}$.
- q *negatively depends* on p if there is a path from p to q in $DG_{\varPhi}$ with at least one negatively labeled edge.

Definition 2. Let $\varPhi$ be a set of formulas and $P = \{p_1, \ldots, p_n\}$ the set of defined predicate symbols of $\varPhi$. A *stratification* of $\varPhi$ is any mapping $s : P \to \{1, \ldots, n\}$ such that $s(p) \leqslant s(q)$ if q depends on p, and $s(p) < s(q)$ if q negatively depends on p. $\varPhi$ is stratifiable if there is a stratification for it.

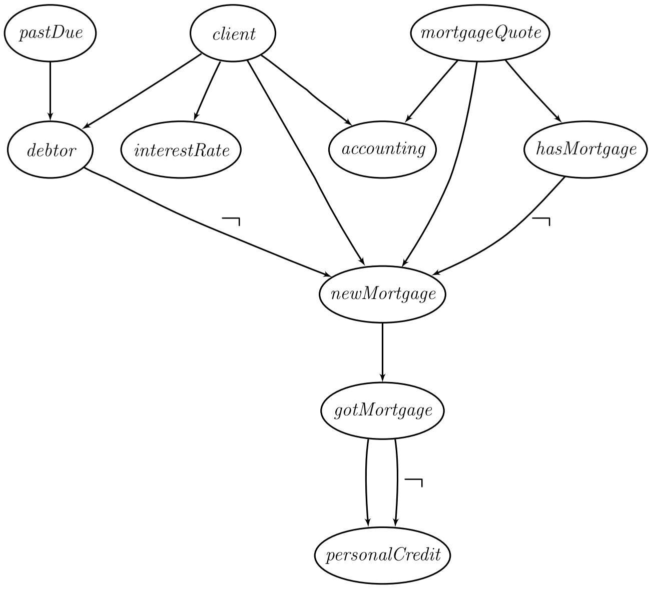

Fig. 5. Dependency graph for Example 6.

Example 10. A stratification for the database Δ of Example 5 will collect all the predicates in the stratum 1 except *nondeltravel* and *trip*, which will be in stratum 2. Intuitively, this means that for evaluating *nondeltravel*, the rest of predicates (except *trip*) should be evaluated before (in particular, *delayed*). Formulating the query: $G \equiv \exists t\ deltravel(x, y, t) \Rightarrow delayed(x, y)$, the augmented set $\Delta \cup \{G\}$ remains stratifiable, but if $G' \equiv trip(mad, lon, T) \Rightarrow delay(mad, ny, t)$ is formulated, the extended set $\Delta \cup \{G'\}$ results non-stratifiable. It is because G' adds the dependency $trip \rightarrow delay$, and then, any stratification s must satisfy $s(trip) \leqslant s(delay) \leqslant s(delayed) < s(nondeltravel) \leqslant s(trip)$, which is impossible. □

From now on, we assume the existence of a fixed stratification s for the considered sets $\Delta \cup \{G\}$. It is useful to have a notion of the stratum of an atom (i.e., the stratum of its predicate symbol), but also to extend this notion to any formula or set of formulas.

Definition 3. Let F be a goal or a clause. The *stratum of a formula* F, denoted by $str(F)$, is recursively defined as:

$$\begin{array}{ll} str(C) = 1 & \\ str(\neg A) = 1 + str(A) & str\big(p(t_1, \ldots, t_n)\big) = s(p) \\ str(F_1 \Box F_2) = max\big(str(F_1), str(F_2)\big), & \text{where } \Box \in \{\wedge, \vee, \Rightarrow\} \\ str(Q\,xF) = str(F), & \text{where } Q \in \{\exists, \forall\} \end{array}$$

The *stratum of a set of formulas* Φ is $str(\Phi) = max\{str(F) \mid F \in \Phi\}$.

5.2. Stratified interpretations and forcing relation

Let $\mathcal{W}$ be the set of stratifiable databases with respect to the same fixed stratification s, which can be built from a particular signature. Interpretations and the fixpoint operator will be applied to the databases of $\mathcal{W}$ and they operate over strata.

Let At be the set of open atoms, i.e., defined predicate symbols of the signature applied to variables (up to variable renaming); and let $\mathcal{SL}_{\mathcal{C}}$ be the set of $\mathcal{C}$-satisfiable constraints modulo $\mathcal{C}$-equivalence. The set $At \times \mathcal{SL}_{\mathcal{C}}$ is finite because we consider finite signatures and compact constraint systems. As we define below, an interpretation over a stratum i of a database will be a set of pairs $(A, [C]) \in At \times \mathcal{SL}_{\mathcal{C}}$, where $str(A) \leqslant i$ and $[C]$ represents the set of constraints $\mathcal{C}$-equivalent to C.

Definition 4. Let $i \geqslant 1$. An *interpretation I over the stratum i* is a function $I : \mathcal{W} \rightarrow \mathcal{P}(At \times \mathcal{SL}_{\mathcal{C}})$, such that for every $\Delta \in \mathcal{W}$, if $(A, [C]) \in I(\Delta)$ then $str(A) \leqslant j$. We denote by $\mathcal{I}_i$ the *set of interpretations over* i.

In order to simplify the notation, we write:

- $(A, C) \in At \times \mathcal{SL}_{\mathcal{C}}$, instead of $(A, [C])$, assuming that C is any representative of its equivalence class $[C]$.
- $[I(\Delta)]_i$ to represent the set $\{(A, C) \in I(\Delta) \mid str(A) = i\}$.

Notice that if $str(\Delta) = k$, then $\{[I(\Delta)]_i \mid 1 \leqslant i \leqslant k\}$ is a partition of $I(\Delta)$. For every $i \geqslant 1$, an order on $\mathcal{I}_i$ can be defined as follows.

Definition 5. Let $i \geqslant 1$ and $I_1, I_2 \in \mathcal{I}_i$. I_1 *is less than or equal to* 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 $1 \leqslant j < i$.
- $[I_1(\Delta)]_i \subseteq [I_2(\Delta)]_i$.

It is straightforward to check that for any $i \geqslant 1$, $(\mathcal{I}_i, \sqsubseteq_i)$ is a poset. The idea behind this definition is that when an interpretation over a stratum i increases, the information of the smaller strata remains invariable. In such a way, if $str(\neg A) = i$, since $str(A) = i - 1$, the truth value of $\neg A$ at the stratum i will remain invariable and monotonicity of the truth relation can be guaranteed even for negative atoms, as we will show. In addition, the following result holds.

Lemma 1. *For any $i \geqslant 1$, any chain of interpretations of $(\mathcal{I}_i, \sqsubseteq_i)$, $\{I_n\}_{n\geqslant 0}$, such that $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$, has a least upper bound $\bigsqcup_{n\geqslant 0} I_n$, which can be defined as: $(\bigsqcup_{n\geqslant 0} I_n)(\Delta) = \bigcup\{I_n(\Delta) \mid n \geqslant 0\}$, for any $\Delta \in \mathcal{W}$.*

Proof. For any $\Delta \in \mathcal{W}$ we define the function $\hat{I}$ as $\hat{I}(\Delta) = \bigcup\{I_n(\Delta) \mid n \geqslant 0\}$, or simply $\bigcup_{n\geqslant 0} I_n(\Delta)$. It must be checked that $\hat{I}$ is the least upper bound of the chain $\{I_n\}_{n\geqslant 0}$.

– $\hat{I}$ is an upper bound of the chain. Let $k \geqslant 0$. For any Δ, we have that:
$[I_k(\Delta)]_j = \bigcup_{n\geqslant 0}[I_n(\Delta)]_j = [\hat{I}(\Delta)]_j$, for $1 \leqslant j < i$, and
$[I_k(\Delta)]_i \subseteq \bigcup_{n\geqslant 0}[I_n(\Delta)]_i = [\hat{I}(\Delta)]_i$.
– Now, we prove that it is the least upper bound of the chain.
Let us assume that I' is an upper bound of $\{I_n\}_{n\geqslant 0}$. For each $k \geqslant 0$, $I_k \sqsubseteq_i I'$ implies that for any Δ, $[I_k(\Delta)]_j = [I'(\Delta)]_j$, for $1 \leqslant j < i$, and $[I_k(\Delta)]_i \subseteq [I'(\Delta)]_i$. Therefore, $\bigcup_{n\geqslant 0}[I_n(\Delta)]_j = [I'(\Delta)]_j$, for $1 \leqslant j < i$, and $\bigcup_{n\geqslant 0}[I_n(\Delta)]_i \subseteq [I'(\Delta)]_i$, for any $\Delta \in \mathcal{W}$. Thus, $\hat{I} \sqsubseteq_i I'$. □

The following definition formalizes the notion of a query G being “true” for an interpretation I in the context of a database Δ, if the constraint C is satisfied.

As already said, we assume that s is not only a stratification for Δ, but also for $\Delta \cup \{G\}$.

Definition 6. Let $i \geqslant 1$. The *forcing relation* $\Vdash$ between pairs I, Δ and pairs (G, C) (where $I \in \mathcal{I}_i$, $str(G) \leqslant i$, and C is $\mathcal{C}$-satisfiable) is recursively defined by the rules below. When $I, \Delta \Vdash (G, C)$, it is said that (G, C) is *forced* by I, Δ.

- $I, \Delta \Vdash (C', C) \iff C \vdash_{\mathcal{C}} C'$.
- $I, \Delta \Vdash (A, C) \iff (A, C) \in I(\Delta)$.
- $I, \Delta \Vdash (\neg A, C) \iff$ for every $(A, C') \in I(\Delta)$, $C \vdash_{\mathcal{C}} \neg C'$ holds, and 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 $C \vdash_{\mathcal{C}} \exists y C'$.
- $I, \Delta \Vdash (\forall x G, C) \iff I, \Delta \Vdash (G[y/x], C)$ where y does not occur free in Δ, $\forall x G$, C.

Those rules are well-defined because if s is a stratification for $\Delta \cup \{G\}$, with $str(G) \leqslant i$, and G' is a subformula of G, then s is also a stratification for $\Delta \cup \{G'\}$, and $str(G') \leqslant i$. Notice that, for the particular case $G \equiv D \Rightarrow G'$, s will be also a stratification for $\Delta \cup \{D, G'\}$.

From now on, when we write $I, \Delta \Vdash (G, C)$ we will assume that if $I \in \mathcal{I}_i$, then $str(G) \leqslant i$ and C is $\mathcal{C}$-satisfiable. The relation $\Vdash$ is not defined otherwise. Formally, $\Vdash$ should be denoted $\Vdash_i$, because there is a forcing relation for each $\mathcal{I}_i$. We avoid the subindex in order to simplify the notation.

The following lemma establishes the monotonicity of the forcing relation.

Lemma 2. *Let $i \geqslant 1$ and $I_1, I_2 \in \mathcal{I}_i$ such that $I_1 \sqsubseteq_i I_2$. Then, for any $\Delta \in \mathcal{W}$, and $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, if $I_1, \Delta \Vdash (G, C)$, then $I_2, \Delta \Vdash (G, C)$.*

Proof. The proof is inductive on the structure of G. The full proof is in Appendix A. Here only a few significative cases are presented.

($\neg A$) If $I_1, \Delta \Vdash (\neg A, C)$, then $C \vdash_{\mathcal{C}} \neg C'$ for every C' such that $(A, C') \in I_1(\Delta)$, or there is no such C' and $C \equiv \top$. Since $str(\neg A) \leqslant i$, obviously $str(A) = j$, for some $j < i$. Then $[I_2(\Delta)]_j = [I_1(\Delta)]_j$, because $I_1 \sqsubseteq_i I_2$, and $I_1, \Delta \Vdash (\neg A, C)$ is equivalent to $I_2, \Delta \Vdash (\neg A, C)$.

($\forall x G'$) $I_1, \Delta \Vdash (\forall x G', C) \iff \Delta \Vdash (G'[y/x], C)$, where y does not occur free in Δ, $\forall x G'$, C. By induction hypothesis $I_2, \Delta \Vdash (G'[y/x], C)$, therefore $I_2, \Delta \Vdash (\forall x G', C)$. □

Lemma 3. *Let $i \geqslant 1$ and let $\{I_n\}_{n \geqslant 0}$ be a denumerable family of interpretations over the stratum i, such that $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$. Then, for any Δ, G and C,*

$$\bigsqcup_{n \geqslant 0} I_n, \Delta \Vdash (G, C) \quad \iff \quad \textit{there exists } k \geqslant 0 \textit{ such that } I_k, \Delta \Vdash (G, C).$$

Proof. In order to simplify the notation we write $\hat{I}$ for $\bigsqcup_{n \geqslant 0} I_n$. The implication from right to left is a consequence of Lemma 2, since $I_k \sqsubseteq_i \hat{I}$ holds for any k. The converse is proved using the result of Lemma 1 ($\hat{I}(\Delta) = \bigcup_{n \geqslant 0} I_n(\Delta)$), by induction on the structure of G. As before, we present only some cases, and the others appear in Appendix A.

($\neg A$) $\hat{I}, \Delta \Vdash (\neg A, C) \iff$ for every C' such that $\hat{I}, \Delta \Vdash (A, C')$, $C \vdash_{\mathcal{C}} \neg C'$, or there is not such C'. We are assuming that $str(\neg A) \leqslant i$ so $str(A) < i$. $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$ implies that $[I_0(\Delta)]_j = [I_1(\Delta)]_j = \cdots = [\bigcup_{n \geqslant 0} I_n(\Delta)]_j = [\bigcup_{n \geqslant 0} I_n(\Delta)]_j$. So for any $k \geqslant 1$, $I_k, \Delta \Vdash (\neg A, C)$.

($D \Rightarrow G'$) $\hat{I}, \Delta \Vdash (D \Rightarrow G', C) \iff \hat{I}, \Delta \cup \{D\} \Vdash (G', C) \Rightarrow$ there is $k \geqslant 0$ such that $I_k, \Delta \cup \{D\} \Vdash (G', C)$, by induction hypothesis $\Rightarrow$ there is $k \geqslant 0$ such that $I_k, \Delta \Vdash (D \Rightarrow G', C)$. □

Next, a continuous operator for every stratum transforming interpretations is defined. Its least fixpoint supplies the expected version of truth at each stratum.

Definition 7. Let $i \geqslant 1$ represent a stratum. The *operator* $T_i : \mathcal{I}_i \to \mathcal{I}_i$ transforms interpretations over i as follows. For any $I \in \mathcal{I}_i$, $\Delta \in \mathcal{W}$, and $(A, C) \in At \times \mathcal{SL}_{\mathcal{C}}$, $(A, C) \in (T_i(I))(\Delta)$ when:

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

The crucial aspect of T_i is: For a database Δ, T_i incorporates information obtained exclusively from the clauses of Δ, whose heads are atoms of the stratum i, and the information of smaller strata remains invariable. Notice that if $str(A) = i$, then $str(\exists \bar{x}(A \approx A' \wedge G)) \leqslant i$ and T_i is well-defined.

In order to establish the existence of a fixpoint of T_i, it will be proved to be monotonous and continuous.

Lemma 4 *(Monotonicity of T_i). Let $i \geqslant 1$ and $I_1, I_2 \in \mathcal{I}_i$ such that $I_1 \sqsubseteq_i I_2$. Then, $T_i(I_1) \sqsubseteq_i T_i(I_2)$.*

Proof. Let us consider any Δ and $(A, C) \in (T_i(I_1))(\Delta)$. This implies that $str(A) \leqslant i$. If $str(A) = j < i$, then $(A, C) \in [I_1(\Delta)]_j = [I_2(\Delta)]_j$, because $I_1 \sqsubseteq_i I_2$ and $j < i$. Hence $(A, C) \in (T_i(I_2))(\Delta)$, by definition of T_i. If $str(A) = i$, then there is a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ, such that the variables $\bar{x}$ do not occur free in A, and $I_1, \Delta \Vdash (\exists \bar{x}(A \approx A' \wedge G), C)$. Using Lemma 2 and the fact that $I_1 \sqsubseteq_i I_2$, we obtain $I_2, \Delta \Vdash (\exists \bar{x}(A \approx A' \wedge G), C)$, which implies $(A, C) \in T_i(I_2)(\Delta)$, by definition of T_i. □

Lemma 5 *(Continuity of T_i). Let $i \geqslant 1$ and $\{I_n\}_{n \geqslant 0}$ be a denumerable family of interpretations over i, such that $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$. Then $T_i(\bigsqcup_{n \geqslant 0} I_n) = \bigsqcup_{n \geqslant 0} T_i(I_n)$.*

Proof. The inclusion $\supseteq$ is a consequence of the monotonicity of T_i. Let us prove the inclusion $\subseteq$. Consider any Δ and $(A, C) \in (T_i(\bigsqcup_{n \geqslant 0} I_n))(\Delta)$. Then $str(A) \leqslant i$. If $str(A) = j < i$, $(A, C) \in [(T_i(\bigsqcup_{n \geqslant 0} I_n))(\Delta)]_j = [I_0(\Delta)]_j$, then $(A, C) \in (T_i(I_0))(\Delta) \subseteq \bigcup_{n \geqslant 0} (T_i(I_n))(\Delta) = (\bigsqcup_{n \geqslant 0} T_i(I_n))(\Delta)$. If $str(A) = i$, there is a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ, such that the variables $\bar{x}$ do not occur free in A, and $\bigsqcup_{n \geqslant 0} I_n, \Delta \Vdash (\exists \bar{x}\ (A \approx A' \wedge G), C)$. Thanks to Lemma 3, there exists

Stratum	Iteration	Considered clause	Deduced pair			
1	$T_1(\emptyset)$	$p(a)$	$(p(x), x \approx a)$	fix_1	fix_2	fix_3
		$p(b)$	$(p(x), x \approx b)$			
		$\forall x(p(x) \Rightarrow t(x))$	None $(*)$			
	$T_1(T_1(\emptyset))$	$p(a)$	None $(**)$			
		$p(b)$	None $(**)$			
		$\forall x(p(x) \Rightarrow t(x))$	$(t(x), x \approx a \vee x \approx b)$			
2	$T_2(\mathit{fix}_1)$	$\forall x(\neg p(x) \Rightarrow q(x))$	$(q(x), x \not\approx a \wedge x \not\approx b)$			
		$\forall x((p(x) \Rightarrow q(x)) \Rightarrow r(x))$	None $(*)$			
3	$T_3(\mathit{fix}_2)$	$\forall x(\neg q(x) \Rightarrow u(x))$	$(u(x), x \approx a \vee x \approx b)$			

Fig. 6. Stratified fixpoint of the elaborated database of Example 11.

$k \geqslant 0$, such that $I_k, \Delta \Vdash (\exists \bar{x}(A \approx A' \wedge G), C)$, and therefore $(A, C) \in (T_i(I_k))(\Delta)$. As a consequence, $(T_i(\bigsqcup_{n\geqslant 0} I_n))(\Delta) \subseteq \bigcup_{n\geqslant 0}(T_i(I_n))(\Delta) = (\bigsqcup_{n\geqslant 0} T_i(I_n))(\Delta)$. □

Proposition 1. *The operator T_1 has a least fixpoint, which is $\bigsqcup_{n\geqslant 0} T_1^n(I_\perp)$, where the interpretation $I_\perp$ represents the constant function $\emptyset$.*

Proof. By the Knaster–Tarski fixpoint theorem [49], using Lemma 5. □

Let fix_1 denote $\bigsqcup_{n\geqslant 0} T_1^n(I_\perp)$, i.e., the least fixpoint at stratum 1.

Consider now the following sequence $\{T_2^n(\mathit{fix}_1)\}_{n\geqslant 0}$ of interpretations in $(\mathcal{I}_2, \sqsubseteq_2)$. Using the properties of T_i, it is easy to prove by induction on $n \geqslant 0$ that this sequence is a chain:

$$\mathit{fix}_1 \sqsubseteq_2 T_2(\mathit{fix}_1) \sqsubseteq_2 T_2\big(T_2(\mathit{fix}_1)\big) \sqsubseteq_2 \cdots \sqsubseteq_2 T_2^n(\mathit{fix}_1) \sqsubseteq_2 \cdots.$$

As before, in accordance to Lemmas 1 and 5, $\{T_2^n(\mathit{fix}_1)\}_{n\geqslant 0}$ has a least upper bound, $\bigsqcup_{n\geqslant 0} T_2^n(\mathit{fix}_1)$, in $(\mathcal{I}_2, \sqsubseteq_2)$ that is a fixpoint of T_2, denoted by fix_2. Proceeding successively in the same way, a chain:

$$\mathit{fix}_{i-1} \sqsubseteq_i T_i(\mathit{fix}_{i-1}) \sqsubseteq_i T_i\big(T_i(\mathit{fix}_{i-1})\big) \sqsubseteq_i \cdots \sqsubseteq_i T_i^n(\mathit{fix}_{i-1}) \sqsubseteq_i \cdots$$

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

$$\mathit{fix}_i = \bigsqcup_{n\geqslant 0} T_i^n(\mathit{fix}_{i-1})$$

can be found. In particular, if $\mathit{str}(\Delta) = k$, we simplify fix_k writing fix. Then, $\mathit{fix}(\Delta)$ represents the pairs (A, C) such that A can be deduced from Δ if C is satisfied. Notice that $\mathit{fix}(\Delta)$ is computed by saturating strata sequentially from $\mathit{fix}_1(\Delta)$ up to $\mathit{fix}_k(\Delta)$, using for every i only the clauses of the stratum i.

Example 11. Given the finite domain $[a, b, c]$, let us consider the elaborated database:

$$\Delta = \big\{p(a), p(b), \forall x\big(p(x) \Rightarrow t(x)\big), \forall x\big(\neg p(x) \Rightarrow q(x)\big), \forall x\big(\big(p(x) \Rightarrow q(x)\big) \Rightarrow r(x)\big), \forall x\big(\neg q(x) \Rightarrow u(x)\big)\big\}.$$

In this database, p and t belong to stratum 1, q and r belong to stratum 2 and u belongs to stratum 3. In Fig. 6 we summarize the pairs that compose the successive iterations of the corresponding fixpoint for each stratum. Note that there are two situations in which the considered clause does not lead to new pairs in the current interpretation, if they are already in it $(**)$, and if it is not possible to find any constraint allowing to force the body of the clause $(*)$. The forcing relation is non-deterministic, therefore the constraint we show for each pair corresponds to a representative of its equivalence class. □

5.3. Soundness and completeness

The fixpoint semantics defined in [19] for $HH(\mathcal{C})$ was proven to be sound and complete with respect to the calculus $\mathcal{UC}$. Now, the coexistence of constraints, negation and implication, as well as the use of $HH_\neg(\mathcal{C})$ as a database system have made necessary to extend $\mathcal{UC}$ to $\mathcal{UC}_\neg$ and to define the concept of stratified interpretation composed of pairs (*Atom, Constraint*), instead of simply atoms as it was done in [19]. The new fixpoint semantics combines stratification techniques with the forcing relation. In this section we prove soundness and completeness of the new fixpoint semantics for $HH_\neg(\mathcal{C})$ with respect to the extended calculus $\mathcal{UC}_\neg$. This means that the forcing relation, considering the least fixpoint at the last stratum of a database and a query, coincides with derivability in $\mathcal{UC}_\neg$. More precisely, if $\mathit{str}(G) = i$, then (G, C) is forced by fix_i in the context of Δ if and only if C is an answer constraint of G from Δ. Although without negation, any database Δ

and query G have a stratification with only one stratum, and then soundness and completeness are similar to those results for $HH(\mathcal{C})$, the general case is not trivial. For this reason, we present the proof of the soundness and completeness in rather detail.

The definitions below introduce a measure of complexity which will be used to perform induction in the proof of soundness. Let $i \geqslant 1$ and $\mathcal{S}_i = \{\langle \Delta, G, C\rangle \in \mathcal{W} \times \mathcal{G} \times \mathcal{SL}_\mathcal{C} \mid \mathit{fix}_i, \Delta \Vdash (G, C)\}$. Given any $\langle \Delta, G, C\rangle \in \mathcal{S}_i$, Lemma 3 guarantees that the set $\mathcal{S} = \{k \geqslant 0 \mid T_i^k(\mathit{fix}_{i-1}), \Delta \Vdash (G, C)\}$[1] is non-empty. Therefore, it is possible to define $\mathit{ord}(\langle \Delta, G, C\rangle) = \mathit{min}\,\mathcal{S}$. Let us consider the partially ordered set $(\mathcal{S}_i, <_i)$, where $<_i$ is defined as follows. Given any $\langle \Delta, G, C\rangle$, $\langle \Delta', G', C'\rangle \in \mathcal{S}_i$, $\langle \Delta, G, C\rangle <_i \langle \Delta', G', C'\rangle$ if:

- $\mathit{ord}(\langle \Delta, G, C\rangle) < \mathit{ord}(\langle \Delta', G', C'\rangle)$, or
- $\mathit{ord}(\langle \Delta, G, C\rangle) = \mathit{ord}(\langle \Delta', G', C'\rangle)$ and G is a renaming of a strict subformula of G'.

Such partial order is well-founded.

Proposition 2. *For every $\Delta \in \mathcal{W}$, and every pair $(G, C) \in \mathcal{G} \times \mathcal{SL}_\mathcal{C}$, such that $str(G) = 1$ then:*

$$\mathit{fix}_1, \Delta \Vdash (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G.$$

Proof. The proof is an adaptation of those presented in [19] to the definition of the forcing relation defined now for $HH_\neg(\mathcal{C})$. Notice that, since we are assuming that $str(G) = 1$, then the case $G \equiv \neg A$ has not to be considered. □

Now, we deal with the general case.

Proposition 3. *For every $i \geqslant 1$, $\Delta \in \mathcal{W}$, and every pair $(G, C) \in \mathcal{G} \times \mathcal{SL}_\mathcal{C}$, such that G does not contain negation, if $str(G) \leqslant i$, then:*

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

Proof. The more difficult case to prove is for G atomic. We show here the proof of this case. The full proof is detailed in Appendix A.

$\Rightarrow$) This implication can be proved by induction on the structural order $(\mathcal{S}_i, <_i)$. Let us take $\langle \Delta, G, C\rangle \in \mathcal{S}_i$ and assume that, for any other $\langle \Delta', G', C'\rangle \in \mathcal{S}_i$, $\langle \Delta', G', C'\rangle <_i \langle \Delta, G, C\rangle$ implies that $\Delta'; C' \vdash_{\mathcal{UC}_\neg} G'$. Then, let us conclude $\Delta; C \vdash_{\mathcal{UC}_\neg} G$ by case analysis on the structure of G. For the case $G \equiv A$:

$\langle \Delta, A, C\rangle \in \mathcal{S}_i$ implies that $\mathit{fix}_i, \Delta \Vdash (A, C)$. Let $k = \mathit{ord}(\langle \Delta, A, C\rangle)$, then $T_i^k(\mathit{fix}_{i-1}), \Delta \Vdash (A, C)$, which is equivalent to $(A, C) \in (T_i^k(\mathit{fix}_{i-1}))(\Delta)$. Hence, there is a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ such that the variables $\bar{x}$ do not occur free in A, and $T_i^{k-1}(\mathit{fix}_{i-1}), \Delta \Vdash (\exists \bar{x}(A \approx A' \wedge G), C)$. In this case, $\langle \Delta, \exists \bar{x}(A \approx A' \wedge G), C\rangle <_i \langle \Delta, A, C\rangle$, so the induction hypothesis can be applied, obtaining that $\Delta; C \vdash_{\mathcal{UC}_\neg} \exists \bar{x}(A \approx A' \wedge G)$. Using the rule (*Clause*) with the elaborated clause $\forall \bar{x}(G \Rightarrow A')$, it follows that $\Delta; C \vdash_{\mathcal{UC}_\neg} A$.

$\Leftarrow$) It is proved by induction on the height h of the tree proof for $\Delta; C \vdash_{\mathcal{UC}_\neg} G$. The case $G \equiv A$ is an inductive case. We suppose that $\Delta; C \vdash A$ has a proof of height h. Let us prove that $\mathit{fix}_i, \Delta \Vdash (A, C)$. Obviously, the rule employed in the bottom of such proof is (*Clause*). So there must exist a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ such that $\bar{x}$ do not occur free in A, and that $\Delta; C \vdash \exists \bar{x}(A \approx A' \wedge G)$ has a proof of height $h - 1$. By induction hypothesis, $\mathit{fix}_i, \Delta \Vdash (\exists \bar{x}(A \approx A' \wedge G), C)$. Using the definition of the operator T_i, the latter implies $(A, C) \in (T_i(\mathit{fix}_i))(\Delta) = \mathit{fix}_i(\Delta)$, then $\mathit{fix}_i, \Delta \Vdash (A, C)$. □

Termination is guaranteed by the previous lemmas if Δ is stratifiable.

Theorem 1 *(Soundness and completeness). For every $i \geqslant 1$, $\Delta \in \mathcal{W}$, and every pair $(G, C) \in \mathcal{G} \times \mathcal{SL}_\mathcal{C}$, if $str(G) \leqslant i$ then:*

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

Proof. By induction on i. Proposition 2 is the proof of the case $i = 1$.

For $i > 1$, assume the induction hypothesis: for every Δ, G, C, with $str(G) \leqslant i - 1$: $\mathit{fix}_{i-1}, \Delta \Vdash (G, C) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} G$. Proposition 3 corresponds to the proof of every case of G except for $\neg A$. Let us analyze this case: $\mathit{fix}_i, \Delta \Vdash (\neg A, C) \iff$ for every C' such that $(A, C') \in \mathit{fix}_i(\Delta)$, it holds $C \vdash_\mathcal{C} \neg C'$, or there is no such C' and $C \equiv \top$. Obviously, $str(A) \leqslant i - 1$, then the previous sentence is equivalent to say that for every C' such that $\mathit{fix}_{i-i}, \Delta \Vdash (A, C')$, it holds $C \vdash_\mathcal{C} \neg C'$, or there is no such C' and $C \equiv \top$. Applying the induction hypothesis, it is equivalent to say that either for every C' such that $\Delta; C' \vdash_{\mathcal{UC}_\neg} A$, $C \vdash_\mathcal{C} \neg C'$ holds, or there is not such C' and $C \equiv \top$. This is equivalent to $\Delta; C \vdash_{\mathcal{UC}_\neg} \neg A$. □

[1] $I_\perp$ instead of fix_{i-1} for $i = 1$.

As a consequence of this theorem: $(A, C) \in fix(\Delta) \iff \Delta; C \vdash_{\mathcal{UC}_\neg} A$. This means that the atoms in the fixpoint of a database are those that can be derived by the calculus.

One of the advantages of the fixpoint semantics over the proof-theoretic one is that the former provides a model for the whole database, while the latter focuses only on the meaning of a query in the context of a database. In [31] a goal solving procedure for $HH(\mathcal{C})$ based on the uniform calculus (without negation) was presented, but the presence of the negation in the language makes unavailing that procedure. The fixpoint semantics can be considered as the formal basis of particular implementations of database systems based on $HH_\neg(\mathcal{C})$. In fact, the fixpoint of a database definition will correspond to the instance of the database. Once that instance is computed, the forcing relation, in which this semantics is based on, provides a formal basis for a goal-oriented computation of answers.

Notice that the previous formalisms are defined for a generic constraint system $\mathcal{C}$ as a black box for which the existence of a solver which checks $\mathcal{C}$-satisfiability has been assumed. So, we have easily developed an implementation for the scheme $HH_\neg(\mathcal{C})$, based on this semantics, which is independent from any constraint system.

The rest of the paper is devoted to present the implementation of our language as a database system.

6. Introducing a database system based on $HH_\neg(\mathcal{C})$

Having stated $HH_\neg(\mathcal{C})$ as a formal language, this section deals with a general description of our proposal to the implementation of a Constraint Deductive Database including $HH_\neg(\mathcal{C})$ as a query language. Also, here we show some examples of use of the system.

Two main components can be distinguished in the implementation of this query language. One corresponds to the implementation of the fixpoint semantics which is very close to the theory and is independent of the concrete constraint system (cf. Section 8). The other component corresponds to the implementation of the constraint system (cf. Section 7).

The system is available at https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems including a user-oriented manual and a bundle of examples. In the following, the concrete syntax for clauses and queries is quite similar to the usual Prolog syntax, where predicate symbols and constants start with lowercase, whereas variables start with uppercase. In addition, we write `not` for negation, `=>` for implication, `ex(X,G)` representing $\exists xG$, `fa(X,G)` representing $\forall xG$, and `constr(Dom,C)` denoting a constraint C together with its domain. The $HH_\neg(\mathcal{C})$ system also requires explicit type declaration, `type(`*`predicate`*`(`*`dom_1`*`, ..., `*`dom_n`*`))`, for predicates.

Although several solvers can be used together within the same database, they cannot be combined for the moment, i.e., constraints of different domains cannot be freely mixed to get a heterogeneous compound constraint. Predicates with arguments of different domains are restricted to those extensionally defined and are only intended for informative purposes. For instance, in the bank database of Example 6, in order to avoid the combination of domains in more complex relations (that would imply the combination of constraint systems during computation), we associate to each client a real value which represents the client identifier:

```
client_id(smith,1.0).    client_id(brown,2.0).    client_id(mcandrew,3.0).
```

The predicates introduced in Example 6 are defined in the same way, but changing the client names by real values.

6.1. Computation stages

Next, we briefly summarize the stages of computation performed by the system to calculate the fixpoint semantics of a database `Delta`:

1. Check and infer predicate types (cf. Section 7).
2. Build the dependency graph of `Delta` (cf. Appendix B).
3. Compute a stratification s for `Delta`, if there is any. Otherwise, the system throws an error message and stops (cf. Appendix B).
4. If the previous step succeeds, compute *fix*(`Delta`) (cf. Section 8).

The system keeps in memory the computed fixpoint *fix*(`Delta`), the stratification s consisting of a list of pairs $\langle$`defined_predicate_symbol,stratum`$\rangle$, and the dependency graph of `Delta`. Regarding the implementation of the dependency graph, defined in Section 5, new negatively labeled edges have been considered to deal with aggregate functions (cf. Section 7.1) and nested implication (cf. Section 8.1).

For instance, for the database of Example 6, $s = [\langle \texttt{client}, 1\rangle, \langle \texttt{pastDue}, 1\rangle, \langle \texttt{mortgageQuote}, 1\rangle, \langle \texttt{debtor}, 1\rangle, \langle \texttt{interestRate}, 1\rangle, \langle \texttt{hasMortgage}, 1\rangle, \langle \texttt{accounting}, 1\rangle, \langle \texttt{client\_id}, 1\rangle, \langle \texttt{branch}, 1\rangle, \langle \texttt{newMortgage}, 2\rangle, \langle \texttt{gotMortgage}, 2\rangle, \langle \texttt{personalCredit}, 3\rangle]$. And *fix*(`Delta`) contains the pairs corresponding to the extensional database as `(client(3.0,5300.0,3000.0), true)`, as well as the pairs obtained from the intensional database:

- In stratum 1:

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

- In stratum 2:

```
(newMortgage(X,Y), ((Y=<200.0, X=2.0); (Y=<1100.0, X=3.0))),
(gotMortgage(X), (X=2.0; X=3.0))
```

- In stratum 3:

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

6.2. Querying

When a query G is submitted at the prompt HHn(C)>, the system computes, if it exists, a new stratification s' for the set Delta $\cup$ {G}. If there is not such s', the query cannot be computed and the system stops; otherwise, the kept fixpoint, computed for Delta is valid to evaluate the query G. According to the theory, the answer constraint C must satisfy *fix*(Delta); Delta $\Vdash$ (G, C). This forcing relation is implemented by means of the predicate force(Delta,Stratification,I,G,C), as it will be explained in Section 8. This predicate makes use of the current stratification. Let us distinguish two cases, depending on whether the previous stratification s is equal to the new s' or not:

- If $s = s'$, then as *fix*(Delta) was calculated with s, the answer constraint C can be obtained by executing force(Delta, s, *fix*(Delta), G, C).
- If $s \neq s'$, it is because G contains some subgoal of the form D => G′. In this case, the dependency graph of Delta$\cup${G}, from which s' has been obtained, contains the edges of Delta plus the edges corresponding to the new implications introduced by G. Hence the new stratification s' is also a valid stratification for Delta, and the same fixpoint *fix*(Delta) would have been obtained by working with s'. Therefore, as before, the answer constraint C for the submitted query G can be computed by executing force(Delta, s', *fix*(Delta), G, C). The information of the stratification is needed, because solving G requires to solve the implication subgoals inside it. When some D => G′ is going to be solved, in accordance with the definition of the forcing relation, Delta is locally augmented with D in order to find the answer of G′. Since s' has been defined taking into account those implications, it is also a stratification of Delta $\cup$ {D}, that will be used in this local computation.

In summary, in both cases the kept fixpoint *fix*(Delta) is the needed interpretation to find the answer, that is obtained just executing:

force(Delta, s', *fix* (Delta) ,G,C).

Following with the example of the bank, the user can ask whether every client belongs to Madrid office:

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

This query does not imply any change in the dependency graph, it can be solved using the kept fixpoint. A universal quantification over a finite domain is naturally translated into a conjunctive constraint obtained by instantiating the quantified variable with each element in the domain. Then the result is:

```
Answer: false
```

Dealing with negation, the user can ask for the clients that have no mortgage:

```
HHn(C)> not(hasMortgage(N)).
```

This query does not change the stratification and the system uses again the kept fixpoint. Every constraint C associated to a pair (hasMortgage(N),C) in the fixpoint is collected and a negative conjunction is built, which is sent to the solver. Finally, the system returns:

```
Answer: N/=3.0, N/=2.0
```

Let us show a contrived query only to illustrate a situation in which the stratification changes:

```
HHn(C)> newMortgage(N,R) => interestRate(N,R).
```

This query introduces a new dependency between newMortgage and interestRate, then the second predicate must be now in stratum 2. However, the already computed fixpoint is still valid for calculating the answer, which is:

```
Answer: (R=2.0, N=2.0); (R=5.0, N=1.0); (R=5.0, N=3.0)
```

7. Implementing constraint solving

This section focuses on the implementation of constraint solving for several particular constraint systems. We also introduce aggregates in our system, for the first time, as constraint functions belonging to concrete constraint systems.

7.1. Aggregates

Aggregate functions are useful for computing single values from a set of numerical or other-type values. For instance, common predefined functions of relational query languages are the average and the maximum of a numeric attribute. As our deductive database with constraints scheme gives a natural analogy of the relational calculus, a crucial question concerns with how to extend standard aggregation constructs from the relational model to our scheme in the context of a constraint domain. Certain proposals to solve this question have been formulated both in geometric constraint databases (see, e.g., Chapter 6 of [30]) and deductive database settings [47,55,54,39].

In attempting to add aggregates to constraint query languages, several obstacles come up. Aggregate functions take a set of values as input and return a single value, but the output of queries in those languages are constraints which represent intensional answers, not an explicit set of values. In addition, since a constraint answer can represent an infinite set, some aggregate functions, as count, make no sense.

We have taken advantage of certain aspects of the operational semantics of our database system in order to deal with these problems. On the one hand, the stratified fixpoint computation, designed to support negation, is a good framework to incorporate aggregates. The rationale behind this is ensuring monotonicity as, along building the fixpoint for a given stratum, the result of an aggregate function taking values in this stratum may change. For instance, the count of tuples corresponding to a relation is only known after such a relation has been completely computed. This is guaranteed if the sum is computed in a stratum above the relation stratum. This can be achieved by introducing a negative dependency as we will see later in this section. On the other hand, aggregates can be represented as functions of a constraint system, and then its computation can be relegated to the corresponding constraint solver.

As a general requirement for computing an aggregate function over a predicate p, each pair $(p(x_1,\dots,x_n),C)$ in the current interpretation must hold that C restricts each variable $x_1,\dots,x_n$ to a single value. Otherwise, the system is not able to compute it. Our supported aggregates include count, ranging over an atom, and sum (summatory), avg (average), min (minimum), and max (maximum), ranging over an atom and a program variable. Implementation details are given in Appendix C.2.1.

Example 12. For instance, the view:

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

defined for Example 6, allows to compute the liquid assets as the cumulative sum of balances of all the clients, by including the aggregate sum in a constraint expression. The average salary can be specified by:

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

or directly with the aggregate avg:

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

The richness of the database language $HH_{\neg}(\mathcal{C})$ allows to calculate an aggregate function under an assumption which changes the values used in the aggregation. For instance, consider the following view:

```
view(X) :- pastDue(brown,200.0) => constr(real,X=sum(pastDue(N,A),A)).
```

It assumes that client Brown has a past due and then compute the sum of all the past dues in the database. Adding this clause to the current database, the query

```
HHn(C)> view(X)
```

has an answer X=3300. □

Since constraints can now contain aggregate functions, and aggregates include defined predicates, additional considerations must be taken into account in order to compute them. Computing aggregate functions requires that the involved predicates are completely known, i.e., the part of the fixpoint corresponding to those predicates has been fully computed. Similarly to what happens with a clause containing a negated atom in its body, if a clause, defining a predicate p, contains an aggregate over a predicate q, the computation of q must be finished when the computation of p begins. This condition is easily achieved by introducing a negative dependency from q to p, which guarantees $s(q) < s(p)$ in the stratification. For the clauses of Example 12, we get $s(\texttt{client}) < s(\texttt{liquid})$, $s(\texttt{client}) < s(\texttt{avg\_salary})$, and $s(\texttt{pastDue}) < s(\texttt{view})$. Full details on the dependency graph construction and stratification can be found in Appendix B.

7.2. Constraint systems and solvers

We have proposed three constraint systems as possible instances $\mathcal{C}$ of the scheme $HH_{\neg}(\mathcal{C})$: Boolean, Reals, and Finite Domains, representing a family of specific constraint systems ranging over denumerable sets. Enumerated types are included as well as (finite) integer numeric types. Our constraint systems include the concrete syntax for the required values, symbols, connectives, and quantifiers as follows: "true", "false", "=", ",", "not" and "ex(X,C)"; in addition, we also include ";", "/=" , ">", ">=", all of them with the usual meaning. Numeric constraints include arithmetic operators (as "+", "-", ...) and constraint functions (as "abs", ...). Moreover, Boolean and Finite Domain constraints admit the universal quantifier "fa(X,C)". The Finite Domain incorporates the particular domain constraint "X in Range", where Range is a subset of data values built with both 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.

We have incorporated a simple type checking and inferrer mechanism in the system. It expects a type declaration for each predicate symbol in the database and warns about missing declarations or misleading use of predicates. Types for queries are inferred from the type information of the database. In addition to the well-known benefits of using a typed language, we use type information to know the constraint system a constraint belongs to.

We have considered the entailment relation of the classical logic with equality 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(I,C,SC) for $\texttt{C} \vdash_{\mathcal{C}} \texttt{SC}$, intended to solve a constraint C, i.e., to produce a *solved form* SC, if it is satisfiable, or $\perp$ otherwise. A solved form SC corresponding to a constraint C is a simplified, more readable form of the constraint w.r.t. C. A solved form is either a *simple constraint* or a disjunction of simple constraints, where a simple constraint is a constraint that does neither include disjunctions nor quantifications, nor negations. The generic interface solve for solving constraints is as follows:

```
solve(+Interpretation,+Constraint,-SolvedConstraint)
```

which solves Constraint as SolvedConstraint under Interpretation, where this interpretation is included to allow to compute aggregates.

For implementing the constraint systems, instead of starting completely from scratch, we rely on the underlying constraint solvers already available in SWI-Prolog [53,50]. In this way, we develop a solver layer which is built over these underlying (simpler) solvers which is capable of solving all the constraints in such (more complex) $\mathcal{C}$-instances. As an example to illustrate such an implementation, let's consider the $\mathcal{C}$-instance Finite Domains. Let's denote supported constraints in the underlying solver as primitive constraints. On the one hand, certain constraints of the constraint system Finite Domains can be mapped to primitive constraints. This mapping involves relating enumerated data values (non-integer in general) in such constraint system with integers in the underlying solver. Before posting to this solver, a constraint is rewritten with the mapped integer values and, after solving, solved constraints in the constraint store are rewritten back with the corresponding enumerated values. On the other hand, there are constraints in the $\mathcal{C}$-instance that the underlying solver cannot directly solve (quantifiers and disjunctions). For them we develop specific constraint solving as shown in Appendix C.3.

Constraint solving is expected to terminate since solver operations are always monotonic (constraints are added, possibly pruning the search space, but not removed) and enumeration (universal quantification) is only provided on finite domains. Completeness for finite domains is related to completeness of the underlying finite domain solver.

8. Implementing the fixpoint semantics

In this section we summarize the main ideas that guide the implementation of the core system: the fixpoint computation. This implementation is very close to the theoretical framework developed in previous sections, but there are some critical points, regarding nested implication, where we must take pragmatic decisions. We will provide a general view of such an implementation and provide relevant details for nested implications. The constraint solvers are used as black boxes with the appropriate interface predicate solve.

In this section we assume a stratifiable database Δ, with a stratification that has been previously computed and stored as an association list `Stratification`. The fixpoint is then computed stratum by stratum (although a stratum may require to compute the fixpoint for a previous stratum for the database locally enlarged due to nested implications, as we will explain in Section 8.1). The predicate

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

computes $\texttt{Fix} = \mathit{fix}_{\texttt{St}}(\texttt{Delta})$, using `Stratification`.Then, if `Delta` represents a database such that $\mathit{str}(\texttt{Delta}) = k$, this predicate gives $\mathit{fix}_k(\texttt{Delta})$ by computing previous fixpoints from stratum $\texttt{St} = 0$ to $\texttt{St} = k$.

Each fixpoint is evaluated by iterating the fixpoint operator following Definition 7, that relies on the forcing relation, implemented by means of the predicate

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

Given $\texttt{I} = T_i^n(\mathit{fix}_{i-1})(\texttt{Delta})$, for some $n \geqslant 0$ and a fixed stratum $i > 0$, `force` is successful if

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

This predicate is implemented in a deterministic way, by making a case distinction on the syntax of the goal `G` to be forced (see Fig. 9 in Appendix D). But the theory still contains another source of non-determinism: the definition of $\Vdash$ establishes conditions on a constraint C in order to satisfy $\texttt{I},\texttt{Delta} \Vdash (\texttt{G},\texttt{C})$, and the predicate `force` must build a concrete constraint `C` in a deterministic way. In addition, each possible answer constraint for a goal must be captured in a single answer constraint (possibly) using disjunctions. The concrete Prolog code for the predicate `force` is presented and explained in Appendix D. Next we will explain the case of implication which is the farthest from theory.

8.1. The case of `D => G` *in the forcing relation*

Implementing `force(Delta,I,(D => G),C)` requires some special treatment. In this case, according to the definition of the relation $\Vdash$ (see Definition 6), `Delta` is augmented with the clause `D`. At this point, the current set `I` has been computed w.r.t. to the database `Delta`. Then, if i and n are, respectively, the stratum and iteration under construction, $(A, C) \in \texttt{I}$ implies $(A, C) \in T_i^n(I')(\texttt{Delta})$, where I' is the fixpoint for the stratum $i - 1$, built from `Delta`. As stated in the theory, the next step will be to prove $T_i^n(I'), \texttt{Delta}\cup\{\texttt{D}\} \Vdash (\texttt{G},\texttt{C})$. But the question is how to compute $T_i^n(I')(\texttt{Delta}\cup\{\texttt{D}\})$. 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`, and then it represents $\mathit{fix}_{i-1}(\texttt{Delta})$. So nothing is known about the needed set $T_i^n(I')(\texttt{Delta} \cup \{\texttt{D}\})$.

The actual fact 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. Let $\texttt{StG} = \mathit{max}\{St \mid \langle p, St\rangle \in \texttt{Stratification}$, p a predicate symbol in $\texttt{G}\}$. Then, the fixpoint of the stratum `StG` for $\texttt{Delta}\cup\{\texttt{D}\}$ is locally computed, and finally it is proved if $\mathit{fix}_{\texttt{StG}}$, $\texttt{Delta} \cup \{\texttt{D}\} \Vdash (\texttt{G},\texttt{C})$.

This solution causes the following problem. Consider a clause in `Delta` of the form `A :- D => G`, such that $i = \mathit{str}(\texttt{A})$. From Definition 3, $\texttt{StG} \leqslant i$ can be deduced. During the computation of $\mathit{fix}_i(\texttt{Delta})$, the fixpoint operator takes this clause into account in order to look for a pair `(A,C)` to be added to the current `I`. Following Definition 7, $\texttt{I},\texttt{Delta} \Vdash (\exists\bar{x}(\texttt{A} \approx \texttt{A}' \wedge \texttt{D => G}), \texttt{C})$ must be proved, then after the existential quantifiers are eliminated, the predicates

$\texttt{force(Delta,Stratification,I,A} \approx \texttt{A}'\texttt{,C)}$, and

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

will be executed (except variable renaming). The second `force` will call to

```
fixPointStrat(Delta1,Stratification,StG,Fix),
```

where $\texttt{Delta1} = \texttt{Delta} \cup \{\texttt{D}\}$ (modulo elaboration and variable renaming). If $\texttt{StG} = i$, this means to build $\mathit{fix}_i(\texttt{Delta1})$, 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 `force(Delta1,Stratification,I,(D => G),C)` will be executed and `Delta1` will be augmented with the elaboration of `D` once more, obtaining `Delta2`, which is again augmented with the same clause, and so on. However, if $\texttt{StG} < i$, then $\texttt{Fix} = \mathit{fix}_{\texttt{StG}}(\texttt{Delta1})$ can be correctly built, because `A :- D => G` is not considered in strata less that i.

The target $\texttt{StG} < \mathit{str}(\texttt{A})$ would be ensured if the predicate with maximum stratum in `G` negatively depends on the predicate symbol of `A`. This is achieved by negatively labeling the edges from the defined predicate symbols of `G` to the predicate symbol of `A`. Although these new dependencies impose additional syntactical conditions on databases to be stratifiable, the implementation remains sound w.r.t. the stratified fixpoint semantics defined in Section 5.

The following example sketches the computation of the fixpoint for a predicate defined using an embedded implication.

Example 13. Let us consider the following database `Delta`:

```
q(a).           r(c).
q(b).           p(X) :- q(X) => r(X).
```

According to the previous explanations, if we consider a `Stratification` where all the predicates `p`, `q` and `r` are in stratum 1, then we would have the following infinite sequence of calls:

```
fixPointStrat(Delta, Stratification, 1, Fix)
fixPointStrat(Delta∪{q(X)}, Stratification, 1, Fix)
fixPointStrat(Delta∪{q(X)}∪{q(X)}, Stratification, 1, Fix)
...
```

But, with the current definition of dependency graph, the `Stratification` raises `p` to stratum 2, while `q` and `r` are in stratum 1. For the first stratum `fixPointStrat(Delta,Stratification,1,Fix1)` gets `Fix1={(q(X),X=a), (q(X),X=b),(r(X),X=c)}`, because `p(X) :- q(X) => r(X)` is not considered here. For the second (and last) stratum we have the call

```
fixPointStrat(Delta,Stratification,2,Fix2)
```

`Fix2` will be built from `Fix1` introducing pairs related to `p`. In the first iteration of the fixpoint operator, and as we have remarked above, the clause defining `p` will require to execute

```
force(Delta,Stratification,Fix1,q(X) => r(X),C),
```

in order to calculate `C` and then introduce a pair `(p(X), C)` into `Fix1`. Two main steps can be distinguished in such execution (see Appendix D for details):

1. Extending the database `Delta` with `q(X)`, obtaining `Delta1` and locally evaluating the fixpoint of the stratum 1 (the stratum of `r`) for the extended database `Delta1`. This is achieved by calling

   ```
   fixPointStrat(Delta1,Stratification,1,Fix1').
   ```

   Since `p` is not considered now in the stratum 1, `Fix1'` can be correctly calculated as

   ```
   Fix1'= {(q(X),true), (q(X),X=a), (q(X),X=b)(r(X),X=c)}.
   ```

2. Forcing the goal `r(X)` with this new fixpoint, by calling

   ```
   force(Delta1,Stratification,Fix1',r(X),C)
   ```

   which computes the constraint `C` as `X=c`.

Then `(p(X), X=c)` is added to the previous interpretation. Since no more pairs can be added,

```
Fix2 = {(p(X),X=c), (q(X),X=a), (q(X),X=b) (r(X),X=c)}
```

is obtained as the final fixpoint for `Delta` we were looking for. □

We finish this section with an example, illustrating how to obtain a dependency graph, in which some negative labeled edges are generated, as explained before, due to an embedded implication.

Example 14. Consider the clause:

$$D \equiv \forall x\big(G \Rightarrow p(x)\big), \quad \text{where } G \equiv \exists y\big(q(x, y) \Rightarrow \big(r(x) \wedge u(y)\big)\big) \wedge \neg t(x).$$

From G the edges $q \to r$ and $q \to u$ are added to the dependency graph. Now G must be connected with the outer context of the clause D, i.e., the predicate symbols q, r, u, t have to be connected with p. This introduces a first edge $q \to p$ and the rest of edges will be negatively labeled: t occurs explicitly negated and r, u occur in a nested implication. So the edges $r \xrightarrow{\neg} p$, $u \xrightarrow{\neg} p$, $t \xrightarrow{\neg} p$ are introduced. Then, collecting all the edges, the dependency graph for D is:

$$\{q \to r, q \to u, q \to p, r \xrightarrow{\neg} p, u \xrightarrow{\neg} p, t \xrightarrow{\neg} p\}.$$

According to Definition 2, a stratification for D is any mapping s from $\{p, q, r, u, t\}$ to natural numbers satisfying $s(q) \leqslant s(r)$, $s(q) \leqslant s(u)$, $s(q) \leqslant s(p)$, $s(r) < s(p)$, $s(u) < s(p)$, $s(t) < s(p)$. For instance, $s(p) = 2$ and $s(q) = s(r) = s(u) = s(t) = 1$. This way, a database with just this clause is stratifiable. Intuitively, this means that for evaluating p, the rest of predicates should be evaluated before; in particular q, which takes part of a nested implication.

But if we add the clause $D' \equiv \forall x \forall y(p(x) \Rightarrow q(x, y))$, the edge $p \to q$ is generated, and we would also have the inequality $s(p) \leqslant s(q)$. Then, the system of inequalities does not have any solution, i.e., the augmented database is not stratifiable. □

The complete specification of the dependency graph used in the implementation (including the additional negatively labeled edges introduced in this section and those introduced by aggregates explained in Section 7.1) and the stratification algorithm can be found in Appendix B.

9. Conclusions and future work

In this work we have presented the formalization and implementation of the constraint logic programming scheme $HH_{\neg}(\mathcal{C})$ as an expressive deductive database language. The main additions of this language w.r.t. the existing ones come from the fact that the intuitionistic logic of hereditary Harrop formulas which it is based on supports implication and universal quantification in goals. This allows to manage hypothetical queries and encapsulation of variables. In addition, $HH_{\neg}(\mathcal{C})$ incorporates the benefits of using constraints. The proposal includes within the same language these features together with stratified negation and a flexible architecture for different constraint systems. In fact, the framework is independent of the concrete constraint system. In particular, we have provided Boolean, Real, and Finite Domains constraint systems. Also, a limited combination of constraints from different constraint systems is allowed. As a result, we obtain a very powerful database language with a well-formalized theoretical framework and a prototype system that implements it.

$HH_{\neg}(\mathcal{C})$ is a mature proposal from the theoretical point of view, and the prototype shows its viability. But the prototype presented in this work must be enhanced to set it as a practical system. The current implementation is very close to the theory and is a valuable tool for understanding and develop such a theory, but as a consequence it has an expected penalty in efficiency.

A first source of inefficiency comes from the forcing of implication, which dynamically changes the database. This involves the local computation of the fixpoint for augmented databases, which yields to start computations from lower strata. Two important improvements oriented to reduce the number of extra computations can be done. First, when solving $D \Rightarrow G$, where $str(D) = i$, D is included into the current database Δ in order to solve G. However, the fixpoint of the augmented database has not to be computed from scratch, but from the known $fix_{i-1}(\Delta)$, that in fact is equal to $fix_{i-1}(\Delta \cup \{D\})$. Second, for computing the answer constraint associated to G, only the semantics of the predicates which G depends on is necessary. These predicates can be identified from the dependency graph, so only the fixpoint for a fragment of the database, corresponding to the clauses defining such predicates, must be computed.

The bottom-up computation by strata we have chosen as computation mechanism offers a number of benefits as we have explained, but it is also a second source of inefficiency. In this line, well-known methods as magic set transformations [6] and tabling [48] could be worth adapting to the current implementation. Therefore, a more efficient top-down-driven bottom-up computation can be achieved, as it is guided by goals. This is also related to widen the set of computable queries and databases that could relax our stratification restrictions. In particular, by using tabling we can avoid the incorporation of negative dependencies to deal with implications in the body of a clause. The idea for adapting tabling to our scheme is not only memorizing answers but also assumed clauses.

In addition, efficiency can be upgraded if some fixpoint components are mapped to relational tables, in this way some operations can be solved with SQL queries by taking advantage of the existing relational technology performance and memory scalability.

There are several interesting extensions that can be incorporated to the system, as strong integrity constraints. This is useful in a number of circumstances, as in Example 6, where it is assumed that each client has, at most, one mortgage quote. This condition could be naturally expressed as an integrity constraint, better than an assumption. A first approach to this issue has already been addressed in [3], where we sketched how to support primary keys, foreign keys and functional dependencies in $HH_{\neg}(\mathcal{C})$. Another possible extension refers to aggregate functions. We have imposed the requirement that every atom over which the aggregate works on must be ground in the interpretation. While this restriction is quite natural and responds to the most common use of aggregates, the possibility of dropping such a restriction and its practical applications can be studied.

Acknowledgements

This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502). Also thanks to Jan Wielemaker for providing SWI-Prolog [53] and Markus Triska [50] for both providing its FD library and adding new features we needed. Finally, we would like to express our thanks to the anonymous referees who really helped us in improving the paper.

Appendix A. Proofs of Section 5

Lemma 2. *Let $i \geqslant 1$ and $I_1, I_2 \in \mathcal{I}_i$ such that $I_1 \sqsubseteq_i I_2$. Then, for any $\Delta \in \mathcal{W}$, and $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, if $I_1, \Delta \Vdash (G, C)$, then $I_2, \Delta \Vdash (G, C)$.*

Proof. The proof is inductive on the structure of G.

(C') This case is trivial by the definition of $\Vdash$.

(A) $I_1, \Delta \Vdash (A, C) \iff (A, C) \in I_1(\Delta)$. In fact $(A, C) \in [I_1(\Delta)]_j$, $1 \leqslant j \leqslant i$. Then, since $I_1 \sqsubseteq_i I_2$ implies that $[I_1(\Delta)]_j \subseteq [I_2(\Delta)]_j \subseteq I_2(\Delta)$, $(A, C) \in I_2(\Delta)$ and therefore $I_2, \Delta \Vdash (A, C)$.

$(\neg A)$ If $I_1, \Delta \Vdash (\neg A, C)$, then $C \vdash_{\mathcal{C}} \neg C'$ for every C' such that $(A, C') \in I_1(\Delta)$, or there is no such C' and $C \equiv \top$. Since $str(\neg A) \leqslant i$, obviously $str(A) = j$, for some $j < i$. Then $[I_2(\Delta)]_j = [I_1(\Delta)]_j$, because $I_1 \sqsubseteq_i I_2$, and $I_1, \Delta \Vdash (\neg A, C)$ is equivalent to $I_2, \Delta \Vdash (\neg A, C)$.

$(G_1 \wedge G_2)$ If $I_1, \Delta \Vdash (G_1 \wedge G_2, C)$, then $I_1, \Delta \Vdash (G_1, C)$ and $I_1, \Delta \Vdash (G_2, C)$. In both cases the induction hypothesis can be used, notice that the strata of G_1 and G_2 are less than or equal to i, so $I_2, \Delta \Vdash (G_1, C)$ and $I_2, \Delta \Vdash (G_2, C)$, which implies that $I_2, \Delta \Vdash (G_1 \wedge G_2, C)$.

$(G_1 \vee G_2)$ $I_1, \Delta \Vdash (G_1 \vee G_2, C) \iff$ there is $k \in \{1, 2\}$ such that $I_1, \Delta \Vdash (G_k, C)$. By induction hypothesis, $I_2, \Delta \Vdash (G_k, C)$, hence $I_2, \Delta \Vdash (G_1 \vee G_2, C)$.

$(D \Rightarrow G')$ $I_1, \Delta \Vdash (D \Rightarrow G', C) \iff I_1, \Delta \cup \{D\} \Vdash (G', C)$. Then, by induction hypothesis, $I_2, \Delta \cup \{D\} \Vdash (G', C)$ holds, so $I_2, \Delta \Vdash (D \Rightarrow G', C)$.

$(C' \Rightarrow G')$ $I_1, \Delta \Vdash (C' \Rightarrow G', C) \iff I_1, \Delta \Vdash (G', C \wedge C')$. $str(G') = str(C' \Rightarrow G')$, so $str(G') \leqslant i$. Hence, by induction hypothesis, $I_2, \Delta \Vdash (G', C \wedge C')$ holds, which implies that $I_2, \Delta \Vdash (C' \Rightarrow G', C)$.

$(\exists x G')$ $I_1, \Delta \Vdash (\exists x G', C) \iff I_1, \Delta \Vdash (G'[y/x], C')$, where y does not occur free in Δ, $\exists x G'$, C, and $C \vdash_{\mathcal{C}} \exists y C'$. By induction hypothesis $I_2, \Delta \Vdash (G'[y/x], C')$, hence $I_2, \Delta \Vdash (\exists x G', C)$.

$(\forall x G')$ $I_1, \Delta \Vdash (\forall x G', C) \iff \Delta \Vdash (G'[y/x], C)$, where y does not occur free in Δ, $\forall x G'$, C. By induction hypothesis $I_2, \Delta \Vdash (G'[y/x], C)$, therefore $I_2, \Delta \Vdash (\forall x G', C)$. □

Lemma 3. *Let $i \geqslant 1$ and let $\{I_n\}_{n \geqslant 0}$ be a denumerable family of interpretations over the stratum i, such that $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$. Then, for any Δ, G and C,*

$$\bigsqcup_{n \geqslant 0} I_n, \Delta \Vdash (G, C) \quad \iff \quad \textit{there exists } k \geqslant 0 \textit{ such that } I_k, \Delta \Vdash (G, C).$$

Proof. In order to simplify the notation we write $\hat{I}$ as $\bigsqcup_{n \geqslant 0} I_n$. The implication from right to left is a consequence of Lemma 2, since $I_k \sqsubseteq_i \hat{I}$ holds for any k. The converse is proved using the result of Lemma 1 ($\hat{I}(\Delta) = \bigcup_{n \geqslant 0} I_n(\Delta)$), by induction on the structure of G.

(C) $\hat{I}, \Delta \Vdash (C, C') \iff I_k, \Delta \Vdash (C, C')$ is true independently of $k \geqslant 0$.

(A) $\hat{I}, \Delta \Vdash (A, C) \iff (A, C) \in \hat{I}(\Delta) = \bigcup_{n \geqslant 0} I_n(\Delta)$. Therefore, there exists $k \geqslant 0$ such that $(A, C) \in I_k(\Delta)$, hence, for that k, $I_k, \Delta \Vdash (A, C)$.

$(\neg A)$ $\hat{I}, \Delta \Vdash (\neg A, C) \iff$ for every C' such that $\hat{I}, \Delta \Vdash (A, C')$, $C \vdash_{\mathcal{C}} \neg C'$, or there is not such C'. We are assuming that $str(\neg A) \leqslant i$ so $str(A) < i$. $I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \cdots$ implies that $[I_0(\Delta)]_j = [I_1(\Delta)]_j = \cdots = [\bigcup_{n \geqslant 0} I_n(\Delta)]_j = [\bigcup_{n \geqslant 0} I_n(\Delta)]_j$. So for any $k \geqslant 1$, $I_k, \Delta \Vdash (\neg A, C)$.

$(G_1 \wedge G_2)$ $\hat{I}, \Delta \Vdash (G_1 \wedge G_2, C) \iff \hat{I}, \Delta \Vdash (G_j, C)$, for each $j \in \{1, 2\}$. In both cases the induction hypothesis can be used, so there exist $k_1, k_2 \geqslant 0$ such that $I_{k_j}, \Delta \Vdash (G_j, C)$ for each $j \in \{1, 2\}$. Let $k = max(k_1, k_2)$. Then $I_k, \Delta \Vdash (G_j, C)$ for each $j \in \{1, 2\}$ in virtue of Lemma 2, and hence $I_k, \Delta \Vdash (G_1 \wedge G_2, C)$.

$(G_1 \vee G_2)$ $\hat{I}, \Delta \Vdash (G_1 \vee G_2, C) \iff$ there is $j \in \{1, 2\}$ such that $\hat{I}, \Delta \Vdash (G_j, C)$. The induction hypothesis can be used, so there exists $k \geqslant 0$ such that $I_k, \Delta \Vdash (G_j, C)$, and therefore $I_k, \Delta \Vdash (G_1 \vee G_2, C)$.

$(D \Rightarrow G')$ $\hat{I}, \Delta \Vdash (D \Rightarrow G', C) \iff \hat{I}, \Delta \cup \{D\} \Vdash (G', C)$. Then there is $k \geqslant 0$ such that $I_k, \Delta \cup \{D\} \Vdash (G', C)$, by induction hypothesis. Therefore there is $k \geqslant 0$ such that $I_k, \Delta \Vdash (D \Rightarrow G', C)$, by definition of the relation $\Vdash$.

$(C' \Rightarrow G')$ $\hat{I}, \Delta \Vdash (C' \Rightarrow G', C) \iff \hat{I}, \Delta \Vdash (G', C \wedge C')$. Then there is $k \geqslant 0$ such that $I_k, \Delta \Vdash (G', C \wedge C')$, by induction hypothesis, which means that there is $k \geqslant 0$ such that $I_k, \Delta \Vdash (C' \Rightarrow G', C)$.

$(\forall x G')$ $\hat{I}, \Delta \Vdash (\forall x G', C) \iff$ there is a variable y such that y does not occur free in Δ, C, $\forall x G'$, such that $\hat{I}, \Delta \Vdash (G'[y/x], C)$. By induction hypothesis, it happens that there exists $k \geqslant 0$ such that $I_k, \Delta \Vdash (G'[y/x], C)$. Hence $I_k, \Delta \Vdash (\forall x G', C)$ for some $k \geqslant 0$.

$(\exists x G')$ $\hat{I}, \Delta \Vdash (\exists x G', C) \iff$ there is a variable y such that y does not occur free in Δ, C, $\exists x G'$, and a constraint C', such that $\hat{I}, \Delta \Vdash (G'[y/x], C')$, and $C \vdash_{\mathcal{C}} \exists y C'$. By induction hypothesis, it happens that there exists $k \geqslant 0$ such that $I_k, \Delta \Vdash (G'[y/x], C')$. Hence, by definition of $\Vdash$, $I_k, \Delta \Vdash (\exists x G', C)$ for some $k \geqslant 0$. □

Proposition 3. *For every $i \geqslant 1$, $\Delta \in \mathcal{W}$, and every pair $(G, C) \in \mathcal{G} \times \mathcal{SL}_{\mathcal{C}}$, such that G does not contain negation, if $str(G) \leqslant i$, then:*

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

Proof. $\Rightarrow$) This implication can be proved by induction on the structural order $(\mathcal{S}_i, <_i)$. Let us take $\langle \Delta, G, C \rangle \in \mathcal{S}_i$ and assume that, for any other $\langle \Delta', G', C' \rangle \in \mathcal{S}_i$, $\langle \Delta', G', C' \rangle <_i \langle \Delta, G, C \rangle$ implies that $\Delta'; C' \vdash_{\mathcal{UC}_\neg} G'$. Then, let us conclude $\Delta; C \vdash_{\mathcal{UC}_\neg} G$ by case analysis on the structure of G.

$C \in \mathcal{SL}_\mathcal{C}$ If $\langle \Delta, C', C\rangle \in \mathcal{S}_i$ then $C \vdash_\mathcal{C} C'$, therefore $\Delta; C \vdash_{\mathcal{UC}_\neg} C'$ by (C).

$A \in At$ $\langle \Delta, A, C\rangle \in \mathcal{S}_i$ implies that $\mathit{fix}_i, \Delta \Vdash (A, C)$. Let $k = \mathit{ord}(\langle \Delta, A, C\rangle)$, then $T_i^k(\mathit{fix}_{i-1}), \Delta \Vdash (A, C)$, which is equivalent to $(A, C) \in (T_i^k(\mathit{fix}_{i-1}))(\Delta)$. Hence, there is a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ such that the variables $\bar{x}$ do not occur free in A, and $T_i^{k-1}(\mathit{fix}_{i-1}), \Delta \Vdash (\exists \bar{x}(A \approx A' \wedge G), C)$. In this case, $\langle \Delta, \exists \bar{x}(A \approx A' \wedge G), C\rangle <_i \langle \Delta, A, C\rangle$, so the induction hypothesis can be applied, obtaining that $\Delta; C \vdash_{\mathcal{UC}_\neg} \exists \bar{x}(A \approx A' \wedge G)$. Using the rule (*Clause*) with the elaborated clause $\forall \bar{x}(G \Rightarrow A')$, it follows that $\Delta; C \vdash_{\mathcal{UC}_\neg} A$.

$G_1 \wedge G_2$ Then $\langle \Delta, G_1 \wedge G_2, C\rangle \in \mathcal{S}_i$ implies that $\mathit{fix}_i, \Delta \Vdash (G_k, C)$ for each $k \in \{1, 2\}$. G_1, G_2 are strict subformulas of $G_1 \wedge G_2$, hence $\langle \Delta, G_k, C\rangle <_i \langle \Delta, G_1 \wedge G_2, C\rangle$, for each $k \in \{1, 2\}$. Then, by the induction hypothesis, $\Delta; C \vdash_{\mathcal{UC}_\neg} G_k$, for each $k \in \{1, 2\}$. So $\Delta; C \vdash_{\mathcal{UC}_\neg} G_1 \wedge G_2$. applying $(\wedge)$ rule.

$G_1 \vee G_2$ Similar to the previous case.

$D \Rightarrow G$ Now $\langle \Delta, D \Rightarrow G, C\rangle \in \mathcal{S}_i$ implies that $\mathit{fix}_i, \Delta \cup \{D\} \Vdash (G, C)$. Clearly, $\mathit{ord}(\langle \Delta, D \Rightarrow G, C\rangle) = \mathit{ord}(\langle \Delta \cup \{D\}, G, C\rangle)$ and G is a strict subformula of $D \Rightarrow G$, so $\langle \Delta \cup \{D\}, G, C\rangle <_i \langle \Delta, D \Rightarrow G, C\rangle$. Therefore, by the induction hypothesis, $\Delta, D; C \vdash_{\mathcal{UC}_\neg} G$. Thanks to the rule $(\Rightarrow)$, it follows that $\Delta; C \vdash_{\mathcal{UC}_\neg} D \Rightarrow G$.

$C' \Rightarrow G$ Then $\langle \Delta, C' \Rightarrow G, C\rangle \in \mathcal{S}_i$ implies that $\mathit{fix}_i, \Delta \Vdash (G, C \wedge C')$. Clearly, $\langle \Delta, G, C \wedge C'\rangle <_i \langle \Delta, C' \Rightarrow G, C\rangle$. Then, by the induction hypothesis, $\Delta; C \wedge C' \vdash_{\mathcal{UC}_\neg} G$. Hence it is easy to show $\Delta; C \vdash_{\mathcal{UC}_\neg} C' \Rightarrow G$, using $(\Rightarrow_\mathcal{C})$ rule.

$\exists xG$ Then $\langle \Delta, \exists xG, C\rangle \in \mathcal{S}_i$ implies that there is C', such that $C \vdash_\mathcal{C} \exists yC'$ and $\mathit{fix}_i, \Delta \Vdash (G[y/x], C')$, where y does not occur free in Δ, $\exists xG$ and C. Then $G[y/x]$ is a renaming of a strict subformula of $\exists xG$, and $\langle \Delta, G[y/x], C'\rangle <_i \langle \Delta, \exists xG, C\rangle$. Therefore $\Delta; C' \vdash_{\mathcal{UC}_\neg} G[y/x]$ by the induction hypothesis, so $\Delta; C, C' \vdash_{\mathcal{UC}_\neg} G[y/x]$, trivially. Hence $\Delta; C \vdash_{\mathcal{UC}_\neg} \exists xG$, by using the rule $(\exists)$, because $C \vdash_\mathcal{C} \exists yC'$.

$\forall xG'$ Then $\langle \Delta, \forall xG, C\rangle \in \mathcal{S}_i$ implies that $\mathit{fix}_i, w \Vdash (G[y/x], C)$, where the variable y does not occur free in Δ, $\forall xG$, C. Clearly, $\mathit{ord}(\langle \Delta, \forall xG, C\rangle) = \mathit{ord}(\langle \Delta, G[y/x], C\rangle)$ and $G[y/x]$ is a renaming of a strict subformula of $\forall xG$, so $\langle \Delta, G[y/x], C\rangle <_i \langle \Delta, \forall xG, C\rangle$. Therefore, by the induction hypothesis, we obtain $\Delta; C \vdash_{\mathcal{UC}_\neg} G[y/x]$. Applying now $(\forall)$, it follows that $\Delta; C \vdash_{\mathcal{UC}_\neg} \forall xG$.

$\Leftarrow$) It is proved by induction on the height h of the tree proof for $\Delta; C \vdash_{\mathcal{UC}_\neg} G$.

Base case: $h = 1$. The only possibility is that $G \equiv C' \in \mathcal{SL}_\mathcal{C}$. Obviously $\mathit{fix}_i, \Delta \Vdash (C', C)$, because we are assuming $\Delta; C \vdash_{\mathcal{UC}_\neg} C'$, so $C \vdash_\mathcal{C} C'$.

Inductive case: We suppose that $\Delta; C \vdash G$ has a proof of height h. Let us prove that $\mathit{fix}_i, \Delta \Vdash (G, C)$, by case analysis on the rule employed in the bottom of such proof.

(*Clause*) There must exist a variant $\forall \bar{x}(G \Rightarrow A')$ of a clause of Δ such that $\bar{x}$ do not occur free in A, and that $\Delta; C \vdash \exists \bar{x}(A \approx A' \wedge G)$ has a proof of height $h - 1$. By induction hypothesis, $\mathit{fix}_i, \Delta \Vdash (\exists \bar{x}(A \approx A' \wedge G), C)$. Using the definition of the operator T_i, the latter implies $(A, C) \in (T_i(\mathit{fix}_i))(\Delta) = \mathit{fix}_i(\Delta)$, then $\mathit{fix}_i, \Delta \Vdash (A, C)$.

$(\wedge)$ There must exist goals G_1, G_2 such that $G \equiv G_1 \wedge G_2$ and the sequent $\Delta; C \vdash G_k$ has a proof of height less than h for each $k \in \{1, 2\}$. By induction hypothesis, $\mathit{fix}_i, \Delta \Vdash (G_1, C)$, and $\mathit{fix}_i, \Delta \Vdash (G_2, C)$. As a consequence, $\mathit{fix}_i, \Delta \Vdash (G_1 \wedge G_2, C)$.

$(\vee)$ Similar to the previous case.

$(\Rightarrow)$ Then $G \equiv D \Rightarrow G'$ and the sequent $\Delta, D; C \vdash G'$ has a proof of height $h - 1$. By induction hypothesis, $\mathit{fix}_i, \Delta \cup \{D\} \Vdash (G', C)$. Therefore $\mathit{fix}_i, \Delta \Vdash (D \Rightarrow G', C)$.

$(\Rightarrow_\mathcal{C})$ Now $G \equiv C' \Rightarrow G'$ and the sequent $\Delta; C, C' \vdash G'$ has a proof of height $h - 1$. By the properties of $\vdash_{\mathcal{UC}_\neg}$, also $\Delta; C \wedge C' \vdash G'$ has a proof of height $h - 1$. Applying now the induction hypothesis, $\mathit{fix}_i, \Delta \Vdash (G', C \wedge C')$. Therefore $\mathit{fix}_i, \Delta \Vdash (C' \Rightarrow G', C)$.

$(\exists)$ Then $G \equiv \exists xG'$, and there must exist a constraint C' and a variable y not occurring free in Δ, C, $\exists xG'$, such that $\Delta; C \wedge C' \vdash G'[y/x]$ has a proof of height $h - 1$ and $C \vdash_\mathcal{C} \exists yC'$. Then $\mathit{fix}_i, \Delta \Vdash (G'[y/x], C \wedge C')$, by induction hypothesis, and therefore $\mathit{fix}_i, \Delta \Vdash (\exists xG', C)$, because $C \vdash_\mathcal{C} \exists y(C \wedge C')$.

$(\forall)$ G must be of the form $\forall xG'$, and there must exist a variable y not occurring free in Δ, C, $\forall xG'$ such that $\Delta; C \vdash G'[y/x]$, has a proof of height $h - 1$. By induction hypothesis, $\mathit{fix}_i, \Delta \Vdash (G'[y/x], C)$, and, as a consequence, $\mathit{fix}_i, \Delta \Vdash (\forall xG', C)$. □

Appendix B. Dependency graph and stratification

The algorithm for calculating the dependency graph is expressed by means of the mutually recursive functions *dpClause* and *dpGoal* defined in Fig. 7, depending on the structure of the formula. These functions return a pair $\langle E, N\rangle$, where E is a set of edges of the form $p \rightarrow q$ or $p \xrightarrow{\neg} q$, and N is an auxiliary set of *link*-nodes which stores the predicate symbols occurring in the formula. Moreover, each of these predicate symbols has a negative annotation in three cases: if they occur in a negated atom, in a nested implication or in an aggregate function (notice that the link-nodes, $\neg \mathit{preds}(C)$, in the fourth case in the definition of *dpGoal* correspond to the aggregate functions occurring in C). This annotation will be then propagated to the edges coming out of those nodes.

By using the function *dpClause* and *dpGoal*, it is straightforward to calculate the dependency graph of a set of formulas Φ (and in particular, for a database) as the union of the edges obtained for each element of the set:

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

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

Notation:

- p_A: predicate symbol of the atom A
- $preds(F) = \{p \mid p$ is a definite predicate symbol occurring in $F\}$
- $\neg S = \{\neg p \mid p \in S\}$

Fig. 7. Dependency graph for clauses and goals.

$$DG_\Phi = \bigcup_{D \in \Phi} \{E_D \mid dpClause(D) = \langle E_D, N \rangle\} \cup \bigcup_{G \in \Phi} \{E_G \mid dpGoal(G) = \langle E_G, N \rangle\}.$$

Once we have the dependency graph, the particular algorithm for finding a stratification for Δ (or for checking that it is not stratifiable) associates to each predicate symbol p an integer variable $X_p \in [1 \ldots N]$, where N is the number of predicate symbols of Δ, and generates a system of inequalities: each dependency $p \rightarrow q$ produces $X_p \leqslant X_q$ 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).

Appendix C. Implementation of constraint solving

This appendix includes implementation details of the constraint solvers, focusing on the finite domain and aggregates.

C.1. Implementing `solve`

The generic interface to the constraint solvers is implemented as follows:

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

This code first calls `simplify_ground_ctr`, which simplifies trivial ground primitive constraints (as, e.g., =, >, ...). Next, the call to predicate `partition_ctr` partitions the input constraint into a list whose components belong to different constraint domains. This partition is always possible when the constraint ranges over a single domain. When it combines different domains, the current implementation is able to achieve the partition in some cases. If it is unable to do it, then an exception is raised.

The call to `solve_ctr_list` posts each component to its corresponding solver as a call to the predicate `solveFD` (described later). After, the solved constraint, which is represented as a list, is transformed back into a conjunctive constraint via `ctr_list_to_ctr`. Finally, this constraint is simplified by logical axioms as De Morgan's laws. In addition to this generic interface, the particular interface

```
solve(+Domain,+Interpretation,+Constraint,-SolvedConstraint).
```

is also provided, which is useful when the domain `Dom` is already known and can be directly posted to its corresponding solver.

```
(01) solveFD(Dom,I,C,SC) :-
(02)   copy_term(C,FC),                 % Input variables keep untouched
(03)   get_vars(C,Vars),                % Input variables are held to be
(04)   get_vars(FC,FVars),              %  mapped to the solved new vars
(05)   swap_qvars_by_fvars(FC,QFC),     % Replace quantified vars by fresh ones
(06)   constrain_domains(QFC,Dom),      % Constrain variables to the current domain
(07)   domain_to_int(QFC,Dom,IC),       % Domain mapping from enumerated to integer
(08)   bagof((FVars,Cs,Sat),            % List of (Fresh vars,Constraints,Satisfiable)
(09)     (solveFD_ctr(IC,Dom,I,true),   % Solving
(10)      satisfiable(IC,Sat),          % Check satisfiability
(11)      project_ctrs(FVars,Vars,Cs)   % Project constraints wrt. input vars
(12)     ), LFVarsCsS), !               % List of Fresh vars,Constraints,Satisfiable
(13)   filter_ctr_list(LFVarsCsS,LICs), % Pick solved constraints
(14)   simplify_disj_list(LICs,SLICs),  % Simplify the disjunctive list
(15)   disj_list_to_ctr(SLICs,ISC),     % Convert list to constraint
(16)   get_vars(ISC,FVSC),              % If the output constraint contains
(17)   (fresh_vars(Vars,FVSC) ->        %  fresh variables
(18)    SC = C                          % Then discard the solved constraint
(19)    ;                               %  and return the input constraint
(20)    int_to_domain(ISC,Dom,SC)).     % Else, map domain from integer to enumerated
(21) solveFD(_Dom,_I,_C,false).         % Return false upon unsatisfiability
```

Fig. 8. The Predicate `solveFD` for solving finite domain constraints.

C.2. Implementing `solveFD`

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

- `solveFD(+Domain,+Interpretation,+Constraint,-SolvedConstraint)`
  It solves the input `Constraint` over `Domain` using `Interpretation` and returns its solved form `Solved Constraint`, if it is satisfiable; otherwise, returns `false`.
- `constr_conjFD(+Domain,+Interpretation,-C1,+C2,+C)`
  It computes (using `Interpretation`) the component `C1` of the conjunction `C1,C2` such that `C1,C2` $\vdash_{\mathcal{C}}$ `C`, where $\mathcal{C}$ is the constraint system corresponding to `Domain`.

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

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

Predicate `solveFD` includes calls to the solving of constraints, which are computed with the predicate:

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

which receives a constraint, a domain name and an interpretation, returning whether it is satisfiable (`true`) or not (`false`).

The code excerpt of Fig. 8 implements the required behavior for `solveFD`. 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 `(21)` 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 input constraint is tried to be solved (see next paragraph describing the predicate `solveFD_ctr`). Second, it is checked for satisfiability, that is, try 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.

C.2.1. Aggregates

During evaluating expressions, the predicate `compute_aggr` is responsible of computing the outcome of each aggregate function, as illustrated by one of its clauses below:

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

The predicate `get_values` calls the predicate `lookUpAll` to get concrete values from ground equalities of the form `Var = value` from an interpretation `I`, with respect to an atom `At` and a variable `Var`. Then, these values are stored in a list `S`. Finally, `S` is used to compute the final result `Res` w.r.t. the particular function.

The implementation of the function `count` is slightly different from the others because the call to this function has not a variable as an argument. In this case, we also call the predicate `lookUpAll`, but then, the solver computes the number of occurrences of the atom to be counted from the interpretation.

C.3. Solving primitive constraints

Whereas some constraints can be posted to the underlying solver, others cannot, as negation which is, as shown below, explicitly handled because it can apply to constraints that are not supported by the underlying Prolog solver:

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

Here, the predicate `complement` computes the complemented constraint (e.g., `X#=<Y` is the complemented constraint of `X#>Y`).

An example of unsupported constraint is disjunction, which is computed by collecting all answers (cf. line `(08)` in Fig. 8). Solving this constraint is as follows:

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

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

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

For the universal quantifier, a constraint `fa(X,C)` is replaced by the conjunction `C[X/v1], ..., C[X/vn]`, where `vi` $(1 \leqslant i \leqslant n)$ are the values in the domain of `X`. Note that cuts at the beginning in the body of each clause occur because we add a default case corresponding to an illegal constraint, which involves the raising of an exception.

The constraint solver for Reals follows a similar but simpler route for its implementation since universal quantifiers are not supported, and there are no domain data values to map. Both `solveR` and `constr_conjR` are provided, analogously to `solveFD` and `constr_conjFD` respectively.

Appendix D. Implementation of the forcing relation

The forcing relation $\Vdash$ of Definition 6 is implemented by means of the predicate

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

whose meaning is: given $\mathtt{I} = T_i^n(\mathit{fix}_{i-1})(\mathtt{Delta})$, for some $n \geqslant 0$ and a fixed stratum $i > 0$, `force` is successful if $T_i^n(\mathit{fix}_{i-1}), \mathtt{Delta} \Vdash (\mathtt{G}, \mathtt{C})$. 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 $\mathtt{I}, \mathtt{Delta} \Vdash (\mathtt{G}, \mathtt{C})$, 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.

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

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

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

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

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

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

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

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

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

Fig. 9. Forcing relation.

Fig. 9 shows the implementation of force. There is a clause of force for each goal structure. We explain them shortly.

Clause (1) stands for the forcing of a constraint C over a domain Dom, which is processed by calling the constraint solver. Clause (2) 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 (clause (3)) there are four possible (and mutually 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.

Clause (4) corresponds to an implication with a constraint as antecedent. In this case, we first try to solve the antecedent C2 in order to check its satisfiability. If it is not satisfiable (second part of the disjunction) then the implication trivially holds and the answer constraint C1 is true. If it is satisfiable, then the consequent G must be forced, obtaining an answer constraint C. This answer constraint is obtained using the predicate constr_conj in order to find C1 such that C1,C2 $\vdash_{\mathcal{C}}$ C as stated in the theory (see Definition 6).

Clause (5) corresponds to the case of an implication D => G which introduces additional difficulties as explained in Section 8.1, as it involves a computation of a new fixpoint for the extended database. The predicate elab provides the rules corresponding to the elaboration of the clause D as explained in Section 3.1, then localClauses transforms them into the used representation hhcnClause(Vars,Head,Body). Calling to addLocalClauses the extended database $\texttt{Delta1} = \texttt{Delta} \cup \{\texttt{D}\}$ is obtained. The execution of

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

finds $\texttt{Fix} = \mathit{fix}_{\texttt{StG}}(\texttt{Delta1})$.

Once Fix is computed, it is used to force G with the augmented set Delta1. This corresponds to prove force(Delta1, Stratification, Fix, G, C), which implies $T_i^n(I'), \texttt{Delta} \cup \{\texttt{D}\} \Vdash (\texttt{G}, \texttt{C})$, which is what we wanted to prove.

For the existential (clause (6)), according to the definition of $\Vdash$, to find C such that

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

(analogously for fa(X, G), clause (7)), we obtain G1 as the result of replacing X by a new variable X1 in G; then we prove $\texttt{I}, \texttt{Delta} \Vdash (\texttt{G1}, \texttt{C1})$, and finally C is obtained by solving ex(X1, C1) (fa(X1, C1), respectively).

For a negated atom `not(At)` (clause `(8)`), 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)` ∈ `I`. The variable `NLs` is used to build the constraint `not(C1),...,not(Cn)` (or `true` if `Ls=[]`), that we must solve to obtain the constraint `C` we are looking for.

Clause `(9)` (default case) is the forcing of an atom `At`. As before, we search for all the pairs `(At,C1),...,(At,Cn)` ∈ `I` and then we build the disjunction `C1=C1;...;Cn` and solve it with `solve`.

References

[1] K. Apt, R. Bol, Logic programming and negation: A survey, J. Log. Program. 19 (1994) 9–71.

[2] G. Aranda, S. Nieva, F. Sáenz-Pérez, J. Sánchez, Implementing a fixpoint semantics for a constraint deductive database based on hereditary Harrop formulas, in: Proceedings of the 11th International ACM SIGPLAN Symposium of Principles and Practice of Declarative Programing, PPDP'09, ACM Press, 2009, pp. 117–128.

[3] G. Aranda, S. Nieva, F. Sáenz-Pérez, J. Sánchez, Incorporating integrity constraints to a deductive database system, in: XI Jornadas sobre Programación y Lenguajes, PROLE'11, 2011, pp. 141–152.

[4] V. Bárány, B. ten Cate, M. Otto, Queries with guarded negation, in: Proceedings of the VLDB Endowment, vol. 5, 2012, pp. 1328–1339.

[5] R. Barbuti, M. Martelli, A tool to check the non-floundering logic programs and goals, in: Proceedings of the Programming Language Implementation and Logic Programming 1st International Workshop, PLILP'88, in: LNCS, vol. 348, Springer, 1988, pp. 58–67.

[6] C. Beeri, R. Ramakrishnan, On the power of magic, J. Log. Program. 10 (1991) 255–299.

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

[8] N. Bidoit, Negation in rule-based database languages: A survey, Theoret. Comput. Sci. 78 (1) (1991) 3–83.

[9] A. Bonner, Hypothetical datalog: Complexity and expressibility, Theoret. Comput. Sci. 76 (1) (1990) 3–51.

[10] A. Bonner, A logical semantics for hypothetical rulebases with deletion, J. Log. Program. 32 (2) (1997) 119–170.

[11] J.H. Byon, P.Z. Revesz, DISCO: A constraint database system with sets, in: Proceedings of the Workshop on Constraint Databases and Applications, in: LNCS, vol. 1034, Springer-Verlag, 1995, pp. 68–83.

[12] M. Cai, D. Keshwani, P.Z. Revesz, Parametric rectangles: A model for querying and animation of spatiotemporal databases, in: Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology, EDBT'00, in: LNCS, vol. 1777, Springer-Verlag, 2000, pp. 430–444.

[13] S. Ceri, G. Gottlob, L. Tanca, Logic Programming and Databases, Springer-Verlag, 1990.

[14] D. Chan, An extension of constructive negation and its application in coroutining, in: Proceedings of the North American Conference on Logic Programming, NACLP'89, MIT Press, 1989, pp. 477–493.

[15] H. Christiansen, T. Andreasen, A practical approach to hypothetical database queries, in: Transactions and Change in Logic Databases, in: LNCS, vol. 1472, Springer, 1998, pp. 340–355.

[16] W. Drabent, What is failure? An approach to constructive negation, Acta Inform. 32 (1995) 27–29.

[17] D.M. Gabbay, N-prolog: An extension of prolog with hypothetical implication II – Logical foundations, and negation as failure, J. Log. Algebr. Program. 2 (1985) 251–283.

[18] M. García-Díaz, S. Nieva, Solving constraints for an instance of an extended CLP language over a domain based on real numbers and Herbrand terms, J. Funct. Log. Program. 2 (2003).

[19] M. García-Díaz, S. Nieva, Providing declarative semantics for HH extended constraint logic programs, in: Proceedings of the 6th ACM SIGPLAN International Conference of Principles and Practice of Declarative Programing, PPDP'04, ACM Press, 2004, pp. 55–66.

[20] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2012.

[21] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of the International Conference on Logic Programming/Symposium on Logic Programming, MIT Press, 1988, pp. 1070–1080.

[22] M.F. Goodchild, Twenty years of progress: Giscience in 2010, J. Spatial Inform. Sci. 1 (2010) 3–20.

[23] R. Gross, Implementation of constraint database systems using a compile-time rewrite approach, PhD thesis, ETH, Zurich, 1996.

[24] J. Harland, Success and failure for hereditary Harrop formulae, J. Log. Program. 17 (1993) 1–29.

[25] J. Jaffar, J.L. Lassez, Constraint logic programming, in: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, POPL'87, ACM Press, 1987, pp. 111–119.

[26] J. Jaffar, M.J. Maher, Constraint logic programming: A survey, J. Log. Program. 19/20 (1994) 503–581.

[27] P. Kanellakis, G. Kuper, P. Revesz, Constraint query languages, J. Comput. System Sci. 51 (1995) 26–52.

[28] P. Kanjamala, P.Z. Revesz, Y. Wang, MLPQ/GIS: A GIS using linear constraint databases, in: Proceedings of the Ninth International Conference on Management of Data, McGraw–Hill, 1998, pp. 389–393.

[29] K. Kunen, Negation in logic programming, J. Log. Program. 4 (1987) 289–308.

[30] G. Kuper, L. Libkin, J. Paredaens, Constraint Databases, Springer, 2000.

[31] J. Leach, S. Nieva, M. Rodríguez-Artalejo, Constraint logic programming with hereditary Harrop formulas, Theory Pract. Log. Program. 1 (2001) 409–445.

[32] V. Lifschitz, Introduction to answer set programming, in: Introductory course at the 16th European Summer School in Logic, Language and Information, Unpublished draft, available at: www.cs.utexas.edu/users/vl/mypapers/esslli.ps, 2004.

[33] J. Lipton, S. Nieva, Higher-order logic programming languages with constraints: A semantics, in: Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications, in: LNCS, vol. 4583, Springer-Verlag, 2007, pp. 272–289.

[34] J.W. Lloyd, Foundations of Logic Programming, Springer, 1987.

[35] D. Miller, G. Nadathur, F. Pfenning, A. Scedrov, Uniform proofs as a foundation for logic programming, Ann. Pure Appl. Logic 51 (1991) 125–157.

[36] D. Miller, G. Nadathur, A. Scedrov, Hereditary Harrop formulas and uniform proof systems, in: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, LICS'87, IEEE Computer Society, 1987, pp. 98–105.

[37] A. Momigliano, Minimal negation and hereditary Harrop formulae, in: Proceedings of the Logical Foundations of Computer Science (LFCS), in: LNCS, vol. 620, Springer, 1992, pp. 326–335.

[38] S. Nieva, F. Sáenz-Pérez, J. Sánchez, Formalizing a constraint deductive database language based on hereditary Harrop formulas with negation, in: Proceedings of the International Symposium on Functional and Logic Programming, FLOPS'08, in: LNCS, vol. 4989, Springer-Verlag, 2008, pp. 289–304.

[39] K. Ramamohanarao, J. Harland, An introduction to deductive database languages and systems, VLDB J. 3 (1994) 107–122.

[40] P.Z. Revesz, Safe datalog queries with linear constraints, in: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming, CP98, in: LNCS, vol. 1520, Springer, 1998, pp. 355–369.

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

[42] P.Z. Revesz, Introduction to Constraint Databases, Springer, 2002.
[43] P. Revesz, MLPQ/Presto Users' Manual, Department of Computer Science and Engineering, University of Nebraska–Lincoln, 2004.
[44] P. Rigaux, M. Scholl, A. Voisard, Spatial Database: With Application to GIS, Morgan Kaufmann Ser. Data Manage. Syst., Morgan Kaufmann, 2002.
[45] V.A. Saraswat, The category of constraint systems is cartesian-closed, in: Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science, LICS'92, IEEE Computer Society, 1992, pp. 341–345.
[46] P. Stuckey, Negation and constraint logic programming, Inform. and Comput. 118 (1995) 12–33.
[47] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, G. Pfeifer, Aggregate functions in DLV, in: Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 2nd International ASP'03 Workshop, 2003, CEUR-WS.org.
[48] H. Tamaki, T. Sato, Old resolution with tabulation, in: Proceedings of the 3rd International Conference on Logic Programming, ICLP'86, in: LNCS, vol. 255, 1986, pp. 84–98.
[49] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific J. Math. 5 (1955) 285–309.
[50] M. Triska, Generalising constraint solving over finite domains, in: Proceedings of the 24th International Conference on Logic Programming, ICLP'08, in: LNCS, vol. 5366, Springer-Verlag, 2008, pp. 820–821.
[51] J. Ullman, Database and Knowledge-Base Systems, vols. I (Classical Database Systems) and II (The New Technologies), Computer Science Press, 1995.
[52] A. Van Gelder, K.A. Ross, J.S. Schlipf, The well-founded semantics for general logic programs, J. ACM 38 (1991) 619–649.
[53] J. Wielemaker, An overview of the SWI-prolog programming environment, in: Proceedings of the 13th International Workshop on Logic Programming Environments, Katholieke Universiteit Leuven, Department of Computer Science, 2003, pp. 1–16.
[54] C. Zaniolo, Key constraints and monotonic aggregates in deductive databases, in: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II, Springer-Verlag, 2002, pp. 109–134.
[55] C. Zaniolo, N. Arni, K. Ong, Negation and aggregates in recursive rules: The LDL++ approach, in: Proceedings of the International Conference on Deductive and Object-Oriented Databases, DOOD, Springer, 1993, pp. 204–221.
[56] C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, R. Zicari, Advanced Database Systems, Morgan Kaufmann Publishers, 1997.