Range Restriction for General Formulas

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
brass@informatik.uni-halle.de

Abstract. Deductive databases need general formulas in rule bodies, not only conjuctions of literals. This is well known since the work of Lloyd and Topor about extended logic programming. Of course, formulas must be restricted in such a way that they can be effectively evaluated in finite time, and produce only a finite number of new tuples (in each iteration of the T_P-operator: the fixpoint can still be infinite). It is also necessary to respect binding restrictions of built-in predicates: many of these predicates can be executed only when certain arguments are ground. Whereas for standard logic programming rules, questions of safety, allowedness, and range-restriction are relatively easy and well understood, the situation for general formulas is a bit more complicated. We give a syntactic analysis of formulas that guarantees the necessary properties.

1 Introduction

Deductive databases have not yet been very successful in practice (at least in terms of market share), although their basic idea is practically very important: Deductive databases aim at an integrated system of database and programming language that is based on the declarative paradigm which was so successful in database languages. Currently, database programming is typically done in languages like PHP or Java. The programs construct SQL statements, send them to the database server, fetch the results, and process them. The interface is not very smooth, and although the situation can be improved with specific database languages like PL/SQL and server-side procedures / user-defined functions within the DBMS, the language paradigms remain different. Object-oriented databases were one approach to develop an integrated system based on a single paradigm, but there the declarativity of the database query part was sacrificed, and they did not get a significant market share, too. Nevertheless, there is an obvious demand for integrated database/programming systems, and this demand has even grown because of object-relational features that need programming inside the database server, and because of web and XML applications.

As far as we know, the deductive database prototypes developed so far support only a Datalog variant, and do not support SQL. But SQL is a database standard, and many practical programmers are trained in the SQL language. It would certainly be helpful for the migration of people and projects to deductive technology, if a deductive system can first be used like a standard SQL DBMS,

and only when one wants to use advanced features like recursive views, powerful constraints, or stored procedures, one has to learn some form of Datalog.

However, supporting SQL in a deductive DBMS is not simply a matter of hiring a good programmer — it still needs research. A requirement is of course that everything that can be done in SQL can also be done in the deductive language (so that e.g., SQL queries can be translated internally to the deductive language, and then executed). But standard Datalog lacks some SQL features that are important in practice.

One obvious difference between SQL and Datalog is that SQL permits general formulas. Already Lloyd and Topor recommended that general formulas should be allowed in rule bodies [LT84,LT85,Llo87], and developed a transformation from these extended logic programming rules to standard rules (see also [LC05]). Although this transformation is an important yardstick to which newer solutions must be compared, it does not lead to a very efficient query evaluation. In case of disjunctive conditions, rules are split, which might lead in the worst case to exponentially many rules, and even in normal cases computation is duplicated. Therefore, it is worth to consider direct support for general formulas in rules. If a deductive database system should be successful, it must have performance for SQL queries that is more or less comparable to standard DBMS. Splitting complex conditions into many rules is not advantageous for that purpose.

In another paper we investigated how duplicates as in SQL can be supported in extended Datalog rules [Bra09]. There the duplicates that Prolog would generate after the Lloyd/Topor transformation from a rule did not agree with the duplicates from a very similar SQL query. This, too, shows that the Lloyd/Topor transfomation does not solve all problems with regard to general formulas in rule bodies.

In this paper we investigate a basic problem that every concrete deductive database system must solve: At least each iteration of the T_P-operator used in bottom-up evaluation must be effectively computable. It might be that the iteration does not terminate, but this is a quite different problem.

Furthermore, a concrete system has a lot of built-in predicates, for instance $<$ is essential for many database queries. In standard database systems (not applying a constraint solver), we must ensure that in a call to $<$ both arguments are ground. Since derived predicates can be called with different binding patterns, the system must be able to automatically find a possible evaluation sequence. This is not difficult for standard rules, where only the body literals must be reordered, but it is technically a bit more complicated for general rules.

2 Standard Rules

Let us first quickly repeat the usual solution to range-restriction in the case of standard rules with conjunctions of literals (positive and negative) in the body. The predicates are classified into

– EDB-predicates ("extensional database"), the given database relations: Of course, these predicates have finite extensions, i.e. they are a finite set of tuples.
– IDB-predicates ("intensional database"), the predicates defined by rules: Only these predicates can appear in the rule heads. They have finite extensions in each step of the fixpoint iteration with the T_P-operator (computing immediate consequences of rules). If the iteration does not terminate, the extension might be infinite in the minimal model, but that is a different problem (not subject of this paper).
– Built-in predicates like "<", which are defined by program code inside the DBMS. These predicates can have infinite extensions, i.e. they might be true for an infinite number of argument values.

In bottom-up evaluation, which is the basis of deductive databases, all rules must be range-restricted. In the most basic case, this means that every variable that is used in a rule must appear at least once in a positive body literal with an EDB or IDB predicate (not a built-in predicate with an infinite extension). In this way, every variable is first bound to a finite set of possible values, and then negative body literals and literals with built-in predicates like "<" can be evaluated. This also ensures that every single rule application produces only finitely many result tuples, containing only values that appeared in the finite database relations.

However, these restrictions are very severe. For instance, they do not permit to give names to arbitrary subformulas, e.g. the following would not be range-restricted:

```
lt(X, Y) :- X < Y.
```

Note that such a view also cannot be defined in SQL databases. However, deductive databases must offer features that go beyond SQL, and since derived predicates are such an important construct in deductive databases, a much stronger support can be expected.

The next step in the development was to adapt the definition of range-restriction to the magic-set transformation. There, predicates are assigned binding patterns, which define which arguments are bound (given inputs), and which are free (searched outputs) when a predicate is called. For instance, the above rule would be legal when `lt` has only the binding pattern `bb` (both arguments are bound).

Definition 1 (Binding Pattern). *A binding pattern β for a predicate of arity n is a string over the alphabet* $\{\mathtt{b},\mathtt{f}\}$ *of length n.*

A predicate with more interesting binding patterns is `sum`, where `sum(X,Y,Z)` means `X+Y=Z`. This predicate supports binding patterns `bbf`, `bfb`, `fbb` and `bbb`. I.e. given two arguments, the third can be computed.

In the following, we assume that terms are constants or variables. To some degree, function symbols (term constructors) can be replaced by built-in predicates. Consider for example the standard list append predicate:

```
append([], L, L).
append([F|R], L, [F|RL]) :- append(R, L, RL).
```

By introducing a new variable for each composed term, and using a predicate `cons(X,Y,Z)` $\equiv$ `Z=[X|Y]`, one gets the following definition of `append` without structured terms:

```
append([], L, L).
append(X, L, Y) :- cons(F,R,X), append(R,L,RL), cons(F,RL,Y).
```

The predicate `cons` supports the binding patterns `bbf` (list construction), and `ffb` (splitting a list). Of course, structured terms are a useful and compact notational convenience on the user level. However, internally, there terms can be flattened as shown in the example. The advantage is that terms with evaluable functions like `+` and terms with record constructors like `[_|_]` can be handled in the same framework. For deductive databases this is important since SQL programmers are used to terms with the standard arithmetic operators. The disadvantage of this solution is that data structures with variables in them cannot be handled conveniently. While there are nice applications of such terms in logic programming, they are very uncommon in database applications (and anyway cannot be stored in classical databases).

Definition 2 (Binding Pattern Specification, Valid Binding Patterns). *A binding pattern specification is a mapping* bp *which defines for each predicate* p*, a set of binding patterns* $\mathsf{bp}(\mathsf{p}) \neq \emptyset$*, called the valid binding patterns for this predicate. If* A *is an atomic formula with predicate* p*, we permit to write* $\mathsf{bp}(\mathsf{A})$ *for* $\mathsf{bp}(\mathsf{p})$*.*

Definition 3 (Allowed Interpretation). *Given a binding pattern specification* bp*, an interpretation* $\mathcal{I}$ *is called allowed if it satisfies the binding pattern restrictions of* bp *in the following sense:*

- *Let* n *be the arity of* p *and* $1 \leq i_1 < \cdots < i_k \leq n$ *be the index positions with* $\beta(i_j) = \mathsf{b}$*.*
- *Then for all values* $c_1, \ldots, c_k$ *from the domain of* $\mathcal{I}$*, the following set is finite:*

$$\{(d_1, \ldots, d_n) \in \mathcal{I}[\![\mathsf{p}]\!] \mid d_{i_1} = c_1,\ \ldots,\ d_{i_k} = c_k\}$$

- *Furthermore, it is possible to effectively compute this set for given* $c_1, \ldots, c_k$*.*

For built-in predicates, the valid binding patterns correspond to the implemented variants of a predicate, e.g. $\mathsf{bp}(\texttt{sum}) = \{\mathsf{bbf}, \mathsf{bfb}, \mathsf{fbb}\}$. It is possible, but not necessary to add `bbb`. E.g. if one has an implementation for `bbf`, one can execute `sum(1,2,3)` like `sum(1,2,X)` $\wedge$ `X=3`. Thus, a binding pattern β is more general than a binding pattern β' iff they have the same length n, and $\beta_i = \mathsf{b}$ implies $\beta'_i = \mathsf{b}$ for $i = 1, \ldots, n$.

For standard EDB predicates, it suffices to have one binding pattern $\mathsf{ff}\ldots\mathsf{f}$. This corresponds to the "full table scan". If there are indexes, other binding patterns might become interesting.

For user-defined predicates, mode declarations or a program analysis defines the valid binding patterns.

Now let $\mathsf{vars}(\mathsf{t})$ be the set of variables that appears in term t. Since terms are only constants or variables, the set is a singleton or empty. For a formula φ, we write $\mathsf{vars}(\varphi)$ for the free variables in that formula.

The following notion of "input variables" for a literal is helpful to define range-restriction:

Definition 4 (Input Variables). *Given an atomic formula* $\mathsf{A} = \mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ *and a binding pattern* $\beta = \beta_1 \ldots \beta_n$ *for* p*, the set of input variables of* A *with respect to* β *is*

$$\mathsf{input}(\mathsf{A}, \beta) := \bigcup \{\mathsf{vars}(\mathsf{t}_i) \mid 1 \leq i \leq n,\ \beta_i = \mathsf{b}\}$$

(i.e. all variables that appear in bound arguments).

Input variables in body literals must be bound before the literal can be called. Input variables in head literals are bound when the rule is executed. Now range-restriction for standard rules can be defined as follows:

Definition 5 (Range-Restricted Standard Rule). *A rule*

$$\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n \wedge \neg \mathsf{B}_{n+1} \wedge \cdots \wedge \neg \mathsf{B}_m$$

is range-restricted given a binding pattern β *for the head literal, iff there is a permutation* π *of* $\{1, \ldots, m\}$ *such that*

- *for every* $i \in \{1, \ldots, m\}$ *with* $\pi(i) \leq n$ *there is* $\beta_i \in \mathsf{bp}(\mathsf{B}_{\pi(i)})$ *such that*

$$\mathsf{input}(\mathsf{B}_{\pi(i)}, \beta_i) \subseteq \mathsf{input}(\mathsf{A}, \beta) \cup \mathsf{vars}(\mathsf{B}_{\pi(1)} \wedge \cdots \wedge \mathsf{B}_{\pi(i-1)}),$$

- *for every* $i \in \{1, \ldots, m\}$ *with* $\pi(i) > n$ *it holds that*

$$\mathsf{vars}(\mathsf{B}_{\pi(i)}) \subseteq \mathsf{input}(\mathsf{A}, \beta) \cup \mathsf{vars}(\mathsf{B}_{\pi(1)} \wedge \cdots \wedge \mathsf{B}_{\pi(i-1)}),$$

- *and furthermore it holds that*

$$\mathsf{vars}(\mathsf{A}) \subseteq \mathsf{vars}(\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n \wedge \neg \mathsf{B}_{n+1} \wedge \cdots \wedge \neg \mathsf{B}_m) \cup \mathsf{input}(\mathsf{A}, \beta).$$

The permutation π determines a possible evaluation sequence for the body literals. Note that different binding patterns for the head literal might need different evaluation sequences of the body literals. E.g., when `append` is called with binding pattern `bbf`, the given order of body literals works fine:

```
append(X, L, Y) :- cons(F,R,X), append(R,L,RL), cons(F,RL,Y).
```

If, however, `append` is called with binding pattern `ffb`, the following evaluation sequence is needed:

```
append(X, L, Y) :- cons(F,RL,Y), append(R,L,RL), cons(F,R,X).
```

In deductive databases, possible queries cannot be anticipated, therefore there is a strong need to support different binding patterns for derived predicates.

Interestingly, when the magic set transformation is applied to the rules, the result is a program in which each rule is range-restricted for the binding pattern $\mathsf{ff} \ldots \mathsf{f}$, thus bottom-up evaluation can be easily applied afterwards.

3 Extended Rules

In extended logic programming, the rule bodies can be arbitrary first order formulas. Since a formula is a complex tree structure, we can no longer use a simple permutation in order to define an evaluation sequence. Consider

$$\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$$

and suppose that the following binding restrictions are given: $\mathsf{p}\colon\mathsf{bf}$, $\mathsf{q}\colon\mathsf{bf}$, $\mathsf{r}\colon\mathsf{f}$. Then the only possible evaluation sequence is $\mathsf{r},\mathsf{p},\mathsf{q}$. Of course, one could require that the user writes the formula in a way that left-to-right evaluation is possible. That would simplify the definition a bit, but it would contradict the declarative paradigm. Furthermore, it would not be practical if the derived predicate supports several binding patterns.

The first task is now to generalize the notion of binding patterns from predicates to formulas:

Definition 6 (Generalized Binding Pattern). *A generalized binding pattern for a formula φ is a pair of sets of variables, written $\mathsf{X}_1,\ldots,\mathsf{X}_n \longrightarrow \mathsf{Y}_1,\ldots,\mathsf{Y}_m$, such that $\{\mathsf{X}_1,\ldots,\mathsf{X}_n,\mathsf{Y}_1,\ldots,\mathsf{Y}_m\} \subseteq \mathsf{vars}(\varphi)$.*

This should mean that given values for $\mathsf{X}_1,\ldots,\mathsf{X}_n$, we can compute a finite set of candidate values for $\mathsf{Y}_1,\ldots,\mathsf{Y}_m$. The final decision, whether the formula is true or false in a given interpretation can usually be done only when we have values for all free variables in the formula.

Generalized binding patterns are related to finiteness dependencies [RBS87]. Finiteness dependencies have first been studied for infinite relations (with attributes instead of variables). The definition of when a finiteness dependency is satisfied for a given relation in [RBS87] is a bit unclear: "If $r(\mathsf{X}_1,\ldots,\mathsf{X}_n)$ is finite, then $r(\mathsf{Y}_1,\ldots,\mathsf{Y}_m)$ is finite." Finiteness dependencies have been used for general formulas in [EHJ93], but there the definition of satisfaction is based on the number of function applications that lead from X-values to Y-values.

Our own definition of the meaning of $\mathsf{X}_1,\ldots,\mathsf{X}_n \longrightarrow \mathsf{Y}_1,\ldots,\mathsf{Y}_m$ adds to the basic finiteness requirement an important computability property (and links it to the given binding patterns for the predicates). In order not to overload the semantics of "finiteness dependency" further, we used a different name.

Definition 7 (Valid Generalized Binding Pattern). *A generalized binding pattern $\mathsf{X}_1,\ldots,\mathsf{X}_n \longrightarrow \mathsf{Y}_1,\ldots,\mathsf{Y}_m$ for a formula φ is valid iff for every allowed interpretation $\mathcal{I}$ and all values $d_1,\ldots,d_n$ from the domain of $\mathcal{I}$*

- *the set*

$$\{(\mathcal{A}(\mathsf{Y}_1),\ldots,\mathcal{A}(\mathsf{Y}_m)) \mid (\mathcal{I},\mathcal{A}) \models \varphi,\ \mathcal{A}(\mathsf{X}_1) = d_1,\ldots,\mathcal{A}(\mathsf{X}_n) = d_n\}$$

  is finite (i.e. there are only finitely many possible assignments to the Y_i that make the formula true, given values for the X_i), and
- *a finite superset of this set is effectively computable.*

Consider again the case $\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$. The binding restrictions of the subformula $\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X})$ can be described with the generalized binding patterns:

- $\emptyset \longrightarrow \mathsf{X}$ (because r supports binding pattern f)
- $\mathsf{Y} \longrightarrow \mathsf{Z}$ (because q supports binding pattern bf)

When we have computed a set $\mathcal{D} = \{d_1, \ldots, d_n\}$ of values for X according to the first binding pattern, we cannot say yet whether the entire formula will be true or false. But what has to be guaranteed is that the formula will be certainly false if X has a value outside the set $\mathcal{D}$, no matter what values the other variables will have. The second binding pattern $\mathsf{Y} \longrightarrow \mathsf{Z}$ means that when we already have a single value (or finite set of values) for Y, then there are only finitely many variable assignments for Z such that the formula is true, and Y has the given value (or one from the finite set).

Since negation can be used everywhere inside a formula, not only before atomic formulas, we also need to clarify the meaning of a generalized binding pattern in negated context: In this case we are interested to get finitely many values such that the formula is false. Again, computing a superset is possible. It must only be guaranteed that the formula is true for every value outside the computed set.

Given a set of generalized binding patterns, the following closure operation computes immediate consequences. This closure is for instance used after the union of sets of binding patterns done for a conjunction. Consider again the formula $\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$. The left subformula $\mathsf{p}(\mathsf{X},\mathsf{Y})$ gives the generalized binding pattern $\mathsf{X} \longrightarrow \mathsf{Y}$ corresponding to the binding pattern bf for p (actually, it also gives many more implied generalized binding patterns, see below). The right subformula discussed above yields (among others) $\emptyset \longrightarrow \mathsf{X}$ and $\mathsf{Y} \longrightarrow \mathsf{Z}$. Given these three generalized binding patterns, we can compose them to get $\emptyset \longrightarrow \mathsf{X},\mathsf{Y},\mathsf{Z}$ which means that the complete formula is evaluable with a finite result. This composition is done by the closure operator defined in a minute. An additional purpose of the closure operator is to add trivially implied generalized binding patterns. E.g., when we have $\mathsf{X} \longrightarrow \mathsf{Y}$, this implies $\mathsf{X} \longrightarrow \mathsf{X},\mathsf{Y}$ and $\mathsf{X},\mathsf{Y} \longrightarrow \mathsf{X}$ and $\mathsf{X},\mathsf{Z} \longrightarrow \mathsf{Y},\mathsf{Z}$. Such implied generalized binding patterns are important e.g. for intersections done for disjunctive conditions.

Definition 8 (Closure of Sets of Generalized Binding Patterns). *Let $\mathcal{B}$ be a set of generalized binding patterns for a formula φ. Then*

$$\begin{aligned}\mathsf{cl}_\varphi(\mathcal{B}) := \{\mathcal{X} \longrightarrow \mathcal{Y} \mid\ & \mathcal{X} \subseteq \mathsf{vars}(\varphi), \mathcal{Y} \subseteq \mathsf{vars}(\varphi),\\ & \textit{there are } n \in \mathbb{N}_0,\ \mathcal{X}_1 \longrightarrow \mathcal{Y}_1\ , \ldots,\ \mathcal{X}_n \longrightarrow \mathcal{Y}_n \in \mathcal{B}\\ & \textit{such that for } i = 1, \ldots, n\textit{:}\\ & \mathcal{X}_i \subseteq \mathcal{X} \cup \textstyle\bigcup_{j=1}^{i-1} \mathcal{Y}_j,\\ & \mathcal{Y} \subseteq \mathcal{X} \cup \textstyle\bigcup_{j=1}^{n} \mathcal{Y}_j\}.\end{aligned}$$

Theorem 1. *If every generalized binding pattern in $\mathcal{B}$ is valid, then also every binding pattern in $\mathsf{cl}_\varphi(\mathcal{B})$ is valid.*

It is known that the Armstrong axioms for functional dependencies are sound and complete also for finiteness dependencies [RBS87]. Since generalized binding patterns have a somewhat different semantics, this result does not automatically carry over, but at least the soundness is obvious:

- If $\mathcal{X} \subset \mathcal{Y}$, then $\mathcal{X} \longrightarrow \mathcal{Y}$ is valid (Reflexivity).
- If $\mathcal{X} \longrightarrow \mathcal{Y}$ is valid, then $\mathcal{X} \cup \mathcal{Z} \longrightarrow \mathcal{Y} \cup \mathcal{Z}$ is valid (Augmentation).
- If $\mathcal{X} \longrightarrow \mathcal{Y}$ and $\mathcal{Y} \longrightarrow \mathcal{Z}$ are valid, then $\mathcal{X} \longrightarrow \mathcal{Z}$ is valid.

Definition 9 (Computation of Generalized Binding Patterns).
We need the following auxillary operation:

$$\begin{aligned}\mathsf{intersect}_\varphi(\mathcal{B}_1, \mathcal{B}_2) := \{\mathcal{X} \longrightarrow \mathcal{Y} \mid\ & \mathcal{X} \in \mathsf{vars}(\varphi),\ \mathcal{Y} \in \mathsf{vars}(\varphi),\\ & \textit{there are } \mathcal{X}_1 \longrightarrow \mathcal{Y}_1 \in \mathcal{B}_1,\ \mathcal{X}_2 \longrightarrow \mathcal{Y}_2 \in \mathcal{B}_2\\ & \textit{with } \mathcal{X}_1 \cup \mathcal{X}_2 \subseteq \mathcal{X},\ \textit{and } \mathcal{Y} \subseteq \mathcal{Y}_1 \cap \mathcal{Y}_2\}.\end{aligned}$$

The functions gbp^+ *and* gbp^- *define sets of generalized binding patterns for arbitrary formulas (in positive/negated context):*

- *If* φ *is an atomic formula* $\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ *(where* p *is not* $=$*):*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{cl}_\varphi\big(\{\mathcal{X} \longrightarrow \mathsf{vars}(\varphi) \mid \textit{there is } \beta \in \mathsf{bp}(\mathsf{p}) \textit{ with } \mathcal{X} = \textstyle\bigcup_{\beta(i)=\mathsf{b}} \mathsf{t}_i\}\big)\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{cl}_\varphi(\{\mathsf{vars}(\varphi) \longrightarrow \mathsf{vars}(\varphi)\})\end{aligned}$$

- *If* φ *is an atomic formula* $\mathsf{t}_1 = \mathsf{t}_2$*:*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{cl}_\varphi\big(\{\mathcal{X} \longrightarrow \mathsf{vars}(\varphi) \mid \mathcal{X} = \mathsf{vars}(\mathsf{t}_1) \textit{ or } \mathcal{X} = \mathsf{vars}(\mathsf{t}_2)\}\big)\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{cl}_\varphi(\{\mathsf{vars}(\varphi) \longrightarrow \mathsf{vars}(\varphi)\})\end{aligned}$$

- *If* φ *is a negated formula* $\neg\varphi_0$*:*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{gbp}^-(\varphi_0)\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{gbp}^+(\varphi_0)\end{aligned}$$

- *If* φ *is a conjunction* $\varphi_1 \wedge \varphi_2$*:*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{cl}_\varphi\big(\mathsf{gbp}^+(\varphi_1) \cup \mathsf{gbp}^+(\varphi_2)\big)\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{cl}_\varphi\big(\mathsf{intersect}(\mathsf{gbp}^-(\varphi_1),\ \mathsf{gbp}^-(\varphi_2))\big)\end{aligned}$$

- *If* φ *is a disjunction* $\varphi_1 \vee \varphi_2$*:*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{cl}_\varphi\big(\mathsf{intersect}(\mathsf{gbp}^+(\varphi_1),\ \mathsf{gbp}^+(\varphi_2))\big)\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{cl}_\varphi\big(\mathsf{gbp}^-(\varphi_1) \cup \mathsf{gbp}^-(\varphi_2)\big)\end{aligned}$$

- *If* φ *has the form* $\exists X\colon \varphi_0$ *or* $\forall X\colon \varphi_0$*:*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{cl}_\varphi(\{\mathcal{X} \longrightarrow (\mathcal{Y} - \{X\}) \mid \mathcal{X} \longrightarrow \mathcal{Y} \in \mathsf{gbp}^+(\varphi_0),\ X \notin \mathcal{X}\})\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{cl}_\varphi(\{\mathcal{X} \longrightarrow (\mathcal{Y} - \{X\}) \mid \mathcal{X} \longrightarrow \mathcal{Y} \in \mathsf{gbp}^-(\varphi_0),\ X \notin \mathcal{X}\})\end{aligned}$$

Theorem 2.

- *Every* $\mathsf{X}_1, \ldots, \mathsf{X}_n \longrightarrow \mathsf{Y}_1, \ldots, \mathsf{Y}_m \in \mathsf{gbp}^+(\varphi)$ *is valid.*
- *In the same way, for* $\mathsf{X}_1, \ldots, \mathsf{X}_n \longrightarrow \mathsf{Y}_1, \ldots, \mathsf{Y}_m \in \mathsf{gbp}^-(\varphi)$ *there are only finitely many assignments for the* Y_i *that make the formula false (given values for the* X_i*), and finite supersets of these sets are effectively computable.*

Up to now, we have computed only an upper bound for the values that make a formula true. As explained above for the example $\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$, the reason was that when we consider subformulas, we might not know values for all variables yet that appear in the subformula. Of course, in the end, we want to know the exact set of variable assignments that make the formula true.

This is possible when we have computed candidate assignments for all variables that occur in the formula. Then we can recursively step down the formula and check for every given variable assignment whether the corresponding subformula is true or false. For the atomic formulas this is obvious (our definition of allowed interpretation implies that we can test whether a given tuple of values is contained in the extension of a predicate). For $\wedge$, $\vee$, $\neg$ it is also clear how the truth values computed for the subformulas can be combined. The interesting case are the quantifiers. Let us consider the example

$$\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge \exists\, \mathsf{Z}\colon (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X})).$$

Suppose that the valid binding patterns for the predicates are $\mathsf{p}\colon \mathsf{bf}$, $\mathsf{q}\colon \mathsf{bf}$, $\mathsf{r}\colon \mathsf{f}$. Consider the subformula $\exists\, \mathsf{Z}\colon (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$. In the first phase, we can compute a set of possible values for X, i.e. we get the generalized binding pattern $\emptyset \longrightarrow \mathsf{X}$. But we have no chance to check whether the existential condition is indeed true without having a value for Y (which can only be computed after we have a value for X). But from the finite set of candidate values for X we can compute a finite number of assignments for (X,Y) which must be checked.

In the second phase, we can assume that we have values for all free variables in a subformula. In the example, we must check whether $\exists\, \mathsf{Z}\colon (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$ is indeed true, given values for X and Y. This is possible if there are only a finite number of candidate values for the quantified variable Z which must be tried. So we need that the generalized binding pattern $\mathsf{X},\mathsf{Y} \longrightarrow \mathsf{Z}$ holds for the quantified subformula $\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X})$. This is indeed the case because q supports the binding pattern bf.

Note how different the situation would be if q permitted only the binding pattern bb. We could still compute a finite set of candidate assignments for X and Y, i.e. be sure that the formula $\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge \exists\, \mathsf{Z}\colon (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$ is false outside this set. But we had no possibility to check whether the existential condition is indeed satisfied without "guessing" values for the quantified variable Z.

For universally quantified variables, we need that the quantified formula can be false only for a finite set of values, so that it suffices to explicitly check this set. E.g. consider

$$\mathsf{r}(\mathsf{X}) \wedge \forall\, \mathsf{Y}\colon \mathsf{p}(\mathsf{X},\mathsf{Y}) \rightarrow \mathsf{r}(\mathsf{Y}).$$

(again with the binding patterns $\mathsf{p}\colon \mathsf{bf}$ and $\mathsf{r}\colon \mathsf{f}$). For the universal quantifier to be evaluable, the condition is $\mathsf{X} \longrightarrow \mathsf{Y} \in \mathsf{gbp}^-(\mathsf{p}(\mathsf{X},\mathsf{Y}) \rightarrow \mathsf{r}(\mathsf{Y}))$.

Definition 10 (Range-Restriction). *A rule $\mathsf{A} \leftarrow \varphi$ is range-restricted given a binding pattern β for A iff*

1. $\mathcal{X} \longrightarrow \mathcal{Y} \in \mathsf{gbp}^+(\varphi)$ *where*
   - $\mathcal{X} := \mathsf{input}(\mathsf{A}, \beta)$ *(variables occurring in bound arguments in the head)*
   - $\mathcal{Y} := \mathsf{vars}(\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \leftarrow \varphi)$ *(all variables in the rule except quantified ones).*
2. *for every subformula $\exists\, \mathsf{Z}\colon \varphi_0$ in positive (unnegated) context, or subformula $\forall\, \mathsf{Z}\colon \varphi_0$ in negated context:*

$$(\mathsf{vars}(\varphi_0) - \{\mathsf{Z}\}) \longrightarrow \mathsf{Z} \in \mathsf{gbp}^+(\varphi_0),$$

3. *for every subformula $\forall\, \mathsf{Z}\colon \varphi_0$ in positive context, or subformula $\exists\, \mathsf{Z}\colon \varphi_0$ in negated context:*

$$(\mathsf{vars}(\varphi_0) - \{\mathsf{Z}\}) \longrightarrow \mathsf{Z} \in \mathsf{gbp}^-(\varphi_0).$$

Theorem 3. *The immediate consequences of a range-restricted rule (according to the T_P-operator) can be effectively computed, given values for the input arguments of the head literal.*

As mentioned above, the magic set transformation turns a rule that is range-restricted for a binding pattern β into a rule that is range-restricted for the binding pattern $\mathsf{ff}\ldots\mathsf{f}$ (by adding a condition to the body that binds the input arguments). Then the immediate consequences of the rule can be computed without further restrictions on input arguments.

4 A Possible Extension

It is also possible to define a slightly more liberal version of range-restriction that requires only that variables are bound in the subformula in which they are used. E.g., $\mathsf{p}(\mathsf{X}) \leftarrow \mathsf{q}(\mathsf{X}) \vee \mathsf{r}(\mathsf{X},\mathsf{Y})$ would not be range-restricted according to Definition 10. That is no real problem, since one can write $\mathsf{p}(\mathsf{X}) \leftarrow \mathsf{q}(\mathsf{X}) \vee \exists\, \mathsf{Y}\colon \mathsf{r}(\mathsf{X},\mathsf{Y})$, or alternatively, $\mathsf{p}(\mathsf{X}) \leftarrow (\mathsf{q}(\mathsf{X}) \wedge \mathsf{Y} = \mathsf{nil}) \vee \mathsf{r}(\mathsf{X},\mathsf{Y})$. Nevertheless, it would also be possible (and an improvement for the user) to accept the original version of the rule.

The important insight here is that for an existentially quantified variable (including variables that are free in the rule, but appear only in the body) it is not necessary that the quantified formula is true only for a finite set of values. It is only required that we have to check only a finite set of values. In the example, $\mathsf{q}(\mathsf{X}) \vee \mathsf{r}(\mathsf{X},\mathsf{Y})$ might be true for an infinite set of Y-values (when $\mathsf{q}(\mathsf{X})$ is already true). However, values outside the extension of r will all behave in the same way, therefore it suffices to check a single such value.

In [Bra92] (page 21), we have solved the problem by computing bottom-up not only sets of bound variables (in positive/negated context), but also unbound

variables (in positive/negated context). In the critical condition $\mathsf{q}(\mathsf{X}) \vee \mathsf{r}(\mathsf{X},\mathsf{Y})$ the variable Y is neither bound nor unbound (while X is bound). A generalization of this idea to the case with built-in predicates is subject of our further work.

Another idea is to have a weaker version of generalized binding patterns, where $\mathcal{X} \longrightarrow \mathcal{Y}$ means that given values for the $\mathcal{X}$, it suffices to check a finite set of values for the $\mathcal{Y}$: If the formula is not true for any of these values, there cannot be any assignment that makes it true (with the given values for the $\mathcal{X}$).

5 Related Work

Of course, questions of domain independence and safety (finite answers) have been studied for quite a long time, [Dem92] gives a good overview over earlier work. [Dem92] generalizes this to arbitrary formulas, but does not consider built-in predicates.

[RBS87] have finiteness dependencies, which are similar to our generalized binding patterns, but consider only standard rules. As explained above, the main difference between finiteness dependencies and generalized binding patterns is our additional computability requirement.

[EHJ93] have finiteness dependencies and arbitrary formulae, but their semantics of finiteness dependencies is again different: Their goal is to prove domain independence of a formula and a finiteness dependency $\mathcal{X} \longrightarrow \mathcal{Y}$ means that values for the $\mathcal{Y}$ can be only a bounded number of function applications farther away from the active domain than values for the $\mathcal{X}$. So they consider computable functions, but the built-in predicates discussed here are more general because they can support several binding patterns. The paper also aims at computing the result of a formula, but the method is very different than ours. They investigate the translation of formulas into relational algebra. The last step is explained only by an example, and it seems that sometimes it might be necessary to enumerate the entire active domain.

[Mah97] studies finiteness constraints, which have the form $\varphi \Rightarrow \mathcal{X} \rightarrow_{fin} \mathcal{Y}$ and mean that for each fixed assignment for the variables in $\mathcal{X}$, there are only finitely many values of the variables in $\mathcal{Y}$ in the tuples of p satisfying φ with the given values of the variables in $\mathcal{X}$. The paper mainly investigates the implication problem for these dependencies and for constrained functional dependencies. One might think that when p simply contains the free variables of φ as attributes, then $\varphi \Rightarrow \mathcal{X} \rightarrow_{fin} \mathcal{Y}$ basically means the same as $\mathcal{X} \longrightarrow \mathcal{Y} \in \mathsf{gbp}^+(\varphi)$. However, the purpose is very different. For instance, in Maher's approach, φ is restricted by a constraint domain, with a typical case being linear arithmetic constraints over integers. In our approach, φ is a more or less arbitrary first order formula. It is also not clear how knowledge about binding patterns for used predicates can be specified in Maher's approach: He considers only a single relation besides the very special predicates in the condition φ. This is no fault of the approach, the goals are simply different. Furthermore, we do not use a constraint solver as Maher, but do a simple syntactic bottom-up computation. Of course, this also gives different results. For instance, from $5 \leq \mathsf{X} \wedge \mathsf{X} \leq 5$, Maher would

conclude that X has only a single possible value. In our approach, this formula is evaluable only for a given value of X, since otherwise the binding restrictions for the built-in predicate $\leq$ are not satisfied.

6 Conclusions

I am convinced that deductive databases can still become a serious competitor to standard relational and object-relational databases for many practical projects. Declarative programming has many advantages, and for single queries this is already standard in the database field (SQL is a declarative language). Deductive databases would lift the declarativity to the level of programs, but this is not as easy as it was expected in the times when deductive databases were a hype topic. More research is still needed.

In this paper, we attacked a very basic problem: Which formulas should be allowed in rule bodies? Of course, we need that they have finite solutions, and that the solutions are effectively computable. In a realistic setting, a deductive database will have many built-in predicates with different binding restrictions. The necessary definition is technically not very easy, but still natural and understandable.

Questions of domain independence and safety for general formulas have been investigated before, and finiteness dependencies studied in the literature behave quite similar to our generalized binding patterns. However, the coupling of finiteness conditions with the computability of an upper bound, and the two-step approach to the evaluation of a formula are unique features of the current paper.

Our long-term goal is to develop a deductive database system that supports stepwise migration from classical SQL.

References

[Bra92] S. Brass: Defaults in Deductive Databases (in German). Doctoral Thesis, University of Hannover, 1992.

[Bra09] S. Brass: A Logic with Duplicates for an SQL-compatible Deductive Database. Submitted for publication.

[Dem92] R. Demolombe: Syntactical characterization of a subset of domain-independent formulas. *Journal of the ACM (JACM) 39:1*, 71–94, 1992.

[EHJ93] M. Escobar-Molano, R. Hull, D. Jacobs: Safety and translation of calculus queries with scalar functions. In *Proc. of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS'93)*, 253–264, ACM, 1993.

[LC05] M. Leuschel, S. Craig: A reconstruction of the Lloyd-Topor transformation using partial evaluation. In P. Hill (ed.), *Pre-Proceedings of LOPSTR'05*, Imperial College, London, UK, 2005.
`http://eprints.ecs.soton.ac.uk/11198/`.

[Llo87] J. W. Lloyd: *Foundations of Logic Programming*, 2nd edition. Springer-Verlag, Berlin, 1987.

[LT84] J. W. Lloyd, R. W. Topor: Making Prolog more expressive. *The Journal of Logic Programming 1 (1984)*, 225–240.

[LT85] J. W. Lloyd, R. W. Topor: A basis for deductive database systems. *The Journal of Logic Programming 2 (1985)*, 93–109.

[LTT99] V. Lifschitz, L. R. Tang, H. Turner: Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence 25:3–4 (1999)*, 369–389. `http://citeseer.ist.psu.edu/lifschitz99nested.html`.

[Mah97] M. J. Maher: Constrained Dependencies. *Theoretical Computer Science 173 (1997)*, 113–149. `http://www.cse.unsw.edu.au/~mmaher/pubs/cdb/condep.ps`.

[RBS87] R. Ramakrishnan, F. Bancilhon, A. Silberschatz: Safety of recursive Horn clauses with infinite relations. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS'87)*, 328–339, ACM, 1987.