# Relational Specifications of Infinite Query Answers *

Jan Chomicki

Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742
chomicki@brillig.umd.edu

Tomasz Imieliński

Dept.of Computer Science
Rutgers University
New Brunswick, NJ 08903
imielins@aramis.rutgers.edu

## Abstract

We investigate here functional deductive databases: an extension of DATALOG capable of representing infinite phenomena. Rules in functional deductive databases are Horn and predicates can have arbitrary unary and limited $k$-ary function symbols in one fixed position. This class is known to be decidable. However, least fixpoints of functional rules may be infinite. We present here a method to finitely represent infinite least fixpoints and infinite query answers as *relational specifications*. Relational specifications consist of a finite set of tuples and of a finitely specified congruence relation. Our method is applicable to every domain-independent set of functional rules.

## 1 Introduction

A lot of attention has been recently devoted to deductive databases with "function-free" Horn rules (DATALOG) [CH85, Ull88]. While DATALOG rules constitute an important and practical class of rules, there are phenomena which cannot be expressed in DATALOG. Such processes as flow of time, state transitions, construction of plans and recursive data structures (see later in this section for the examples of these) require limited use of function symbols. Storing data about infinite processes in a database and reasoning about it requires extending DATALOG to allow limited occurrences of function symbols.

There is a number of well known problems resulting from the *unrestricted* introduction of function symbols. The two most important ones are *undecidability* of query processing in the presence of function symbols and *unsafety*: least fixpoints of rules and answers to some queries may become infinite because of the infiniteness of the Herbrand universe.

*Research supported by: NSF grants IRI-87-04614 and IRI-86-09170, ARO grant DAAL-03-88-K0087 and UMIACS.



*Example:* The following rule schedules the meetings of graduate students with their common advisor (the fact Meets(t,x) means that the student x meets his advisor on the day t):

Meets(t,x), Next(x,y) →Meets(t+1,y).

Assume the database $\mathcal{D}$ contains the following tuples (0 stands for the first day of school, 1 for the second etc.):

Meets(0,Tony).
Next(Tony,Jan).
Next(Jan,Tony).

The answer to the query $\mathcal{Q}$={(t,x):Meets(t,x)} contains the following tuples:

Meets(0,Tony), Meets(1,Jan), Meets(2,Tony)...

and is infinite. *End of example.*

A standard solution to the problem of infinite answers is to detect such *unsafe* queries and simply disallow them [RBS87]. In [CI88], we argued that such solution is unnatural. Intuitively, we'd like to be able to finitely represent the infinite object, here: *"every second day"*, and return the representation as the answer to $\mathcal{Q}$. In [CI88], we presented a technique to represent infinite answers in a finite way for a restricted class of rules and queries. But that approach suffered from a lack of generality: for example, it was unable to handle the example above.

Here we provide an alternative approach which is much more powerful and general than the one presented in [CI88]. We notice that least fixpoints of functional rules have a repetitive structure and can be collapsed to finite models using the notion of *congruence*. More precisely, infinite objects are represented as classes of a congruence ($\cong$). Each of those classes can be infinite but there are only finitely many of them. In the above example, there are two such classes:

$a_1 = \{0, 2, 4, \ldots\}$.
$a_2 = \{1, 3, 5, \ldots\}$.

The elements of each class are intuitively equivalent, because they induce similar truth values of facts. For example:

$\forall$x, Meets(**0**,x)$\equiv$Meets(**2**,x)$\equiv$Meets(**4**,x) ...

Therefore it is enough to store one such truth assignment per class. We choose a representative term for each class, here **0** and **1**, and store its truth assignment as the relation:

Meets(**0**,Tony).
Meets(**1**,Jan).

This set of tuples will be called the *primary database*. In addition to this database, the finite specification of the least fixpoint encodes the connections between congruence classes. We propose two ways of doing this.

First, the function symbol (**+1**) of the original rules is represented by a finite function **f** in the following way:

**f(0)=1**.
**f(1)=0**.

The specification contains the graph of this finite function. To check whether Tony meets his advisor on a given day **n**, we apply **n** times the finite function **f** to **0** and check whether the result is **0** (the representative day for Tony's meetings). For Jan, we would check whether the result is **1**.

Alternatively, the congruence is represented equationally as a finite set of ground equations $\mathcal{R}$. Here $\mathcal{R}$ contains **0**$\cong$**2** (from which it can be deduced that **1**$\cong$**3**, **2**$\cong$**4** etc. because $\cong$ is a congruence). Thus from this equation taken together with the primary database, it can be inferred that Tony meets the advisor on even and Jan on odd days. To check whether Jan has a meeting on a given day, it is enough to verify whether this day is odd. In general, this simple arithmetic is not sufficient and a more general procedure, *congruence closure* [DST80], is necessary.

Due to the special role of the congruence relation, we use the term *relational specification* to cover both graph and equational specifications.

Our ultimate goal in this paper is to provide an extension to DATALOG which has the following properties:

1. the query processing problem is still decidable (although it may be not in PTIME).
2. least fixpoints of every domain-independent [1] set of rules are finitely representable by our mechanism. Moreover, this finite representation is effectively computable and explicit. The representation is *explicit* in the sense that once it is computed, the original deductive rules may be forgotten. Single elements of the answer (individual tuples) can then be obtained via some simple computation, e.g. graph traversal or congruence closure.

[1] Intuitively, a least fixpoint of a *domain-independent* set of rules does not depend on the choice of the domain from which the database elements are drawn. The formal definition will be given later.

In other words, we would like to extend DATALOG to be more expressive, but at the same time preserve two important features of DATALOG: termination of yes-no queries (decidability) and the existence of a finite representation of least fixpoints. Later in the paper, we also mention one additional property that our extension of DATALOG shares with it: the existence of a *canonical form* for any set of rules.

We achieve those goals by first choosing a *decidable* class of Horn rules with function symbols as the extension of DATALOG. The occurrences of function symbols are restricted to one fixed position in every predicate. The function symbols are arbitrary unary and limited $k$-ary. We call rules in this class *functional*.

*Finite representability* is much more difficult and the methods to achieve it constitute the main contribution of this paper. In particular, we prove that for every domain-independent set of functional rules $\mathcal{Z}$ and every finite database $\mathcal{D}$, the least fixpoint $LFP(\mathcal{Z},\mathcal{D})$ can be finitely represented. Consequently, query answers can also be finitely represented. This has to be contrasted with our previous result [CI88] where we could only finitely represent least fixpoints and query answers for a limited class of functional rules.

The paper is organized as follows. In the rest of this section, we show further applications of functional rules. In section 2, we define the class of functional rules (our postulated extension of DATALOG) and basic concepts of query processing. In section 3, we present sound and complete methods to compute finite graph and equational specifications of least fixpoints. In section 4, we discuss the computational complexity issues. In section 5, we show that from a relational specification of the least fixpoint, relational specification of query answers may be easily obtained. In section 6, we compare our work with other related research, including our own [CI88] and draw the conclusions. The proofs are omitted for the lack of space.

At the beginning, we showed how functional rules can express periodic events. States were identified with time instants. For every time instant **t**, there was exactly one successor, namely **t+1**. Here we present examples where the number of successor states for a given state may be greater than one and even depend on the database.

*Situation-calculus planning.* Here the functional variable s plays the role of a *state* (situation). Function symbols correspond to *operators* available to a robot [Gre69]. For example, **move**(s,p1,p2) stands for " robot moving from position p1 to p2"

At(**0**,p0).
At(s,p1), Connected(p1,p2) $\rightarrow$At(**move**(s,p1,p2),p2).

The least fixpoint of this set of rules is infinite. So is the answer to the query {y:At(y,p)} which asks about all sequences of moves that can lead the robot to position p. Both can be finitely represented, because there are only finitely many positions that the robot can assume. The notion of periodicity is no longer applicable and is replaced by a more general no-

tion of *congruence*. Intuitively, once the robot is again in the same position it faces the same set of possible moves. On every possible infinite path, there must be a cycle. Representing one cycle traversal is enough: any subsequent traversals can be derived from this representation.

*Simple list processing.* Here the functional variable represents a list. The term **ext(s,x)** corresponds to the list s extended by the element x [2] . The term **0** stands for an empty list. The following rules describe lists containing elements from a set specified by the unary relation P:

P(x) →Member(**ext(0,x)**,x).
P(y), Member(s,x) →Member(**ext(s,y)**,y).
P(y), Member(s,x) →Member(**ext(s,y)**,x).

When two lists **s1** and **s2** have identical elements ($\forall$x, Member(**s1**,x) $\equiv$ Member(**s2**,x)), their extended versions **ext(s1,c)** and **ext(s2,c)** have also identical elements. Two lists with identical sets of elements, e.g. **ext(ext(0,a),b)** and **ext(ext(0,b),a)**, belong to the same congruence class. There are only finitely many congruence classes and the congruence can be finitely specified, yielding a finite representation for the infinite relation Member. We elaborate on this example in section 3.

# 2 Basic notions

We assume that the reader is familiar with the basic notions of syntax and semantics of logic programs [vEK76, Llo87]: Horn rule (clause), query, database, Herbrand universe, Herbrand interpretation and model, least fixpoint of a set of Horn rules. We will use these notions in their standard sense, except for a slightly different definition of the Herbrand universe, interpretation and model.

Given a database $\mathcal{D}$, a set of rules $\mathcal{Z}$ and a (possibly open) query $\mathcal{Q}$ ($\exists\mathcal{Q}$ is $\mathcal{Q}$ closed with existential quantifiers), we are interested in the following query processing problems:

- *yes-no:* " Does $\mathcal{D} \wedge \mathcal{Z}$ imply $\exists\mathcal{Q}$?" (this is equivalent to: "Is $\mathcal{D} \wedge \mathcal{Z} \wedge \neg\exists\mathcal{Q}$ unsatisfiable?").
- *all-answers:* "For what ground answer substitutions $\theta$, $\mathcal{D} \wedge \mathcal{Z}$ implies $\mathcal{Q}\theta$?"

If the function symbols are present in rules, there may be infinitely (but countably) many ground answer substitutions. When queries are positive atomic, we can talk about the set $\{\mathcal{Q}\theta : \mathcal{D} \wedge \mathcal{Z}$ implies $\mathcal{Q}\theta\}$ as the *query answer* (set of tuples).

## 2.1 Syntax

In this section, we define the class of functional rules: our postulated extension of DATALOG. The following definitions capture the intuition that in functional deductive databases the occurrences of function symbols are restricted to one component of predicates.

*Language.* The language contains an infinite (countable) number of predicate, function and variable symbols. As usual, we call 0-ary function symbols *constants*. Variables and constants are (disjointly) partitioned into *functional* and *non-functional* ones. To simplify the exposition, we assume that there is exactly one functional constant **0**. Our results carry to the general case. A function symbol is either *pure* (unary) or *mixed* (of arity at least 2). A predicate symbol are either *functional* or *non-functional*.

*Terms.* A *non-functional term* is either a non-functional constant (a standard database constant) or a non-functional variable. A *functional term* is defined inductively:

1. the functional constant **0** is a pure functional term.
2. a functional variable is a pure functional term.
3. if **v** is a pure functional term and **f** is a pure function symbol, then **f(v)** is a pure functional term.
4. if **v** is a mixed functional term and **f** is a pure function symbol, then **f(v)** is a mixed functional term.
5. if **v** is a functional term and **g** is a mixed $k$-ary function symbol and $\bar{x}$ is a vector of $k-1$ non-functional terms, then **g(v,**$\bar{x}$**)** is a mixed functional term.
6. there are no other functional terms.

Intuitively, a pure term contains only pure (unary) function symbols, functional constants and functional variables. Term which do not contain any variables are called *ground*. The only ground non-functional terms are constants, while ground functional terms may be arbitrarily deep. Note that full-fledged $k$-ary function symbols ($k > 1$), i.e. containing more than one functional term as an argument, are not allowed.

*Formulas.* If P is a functional predicate symbol and R is a non-functional predicate symbol, **v** is a functional term and $\bar{x}$ is a vector of non-functional terms of appropriate arity, then P(**v**,$\bar{x}$) is a functional atom and R($\bar{x}$) is a non-functional atom.

A *functional database* is a set of ground functional and non-functional atoms (tuples). Functional Horn rules (clauses) are defined in the standard way, except that they have to be built from the atoms defined above. Functional queries are existentially quantified conjunctions of functional atoms. A *functional deductive database* is a set of functional rules and a functional database.

Functional rules are a generalization of *temporal* rules [CI88] where only one unary function symbol (**+1**) was allowed. Functional terms will appear in **bold** font. Predicate names will start with a capital letter, constants with a letter from the beginning of the alphabet (a,b..., A,B...) and variables with a small letter from the end of the alphabet (z,y...).

[2] This is just a version of the more familiar "cons", with the arguments reversed to follow the syntactic convention used throughout this paper.

## 2.2 Semantics

We will be using a slightly different notion of Herbrand universe than [vEK76, Llo87]. We assume that all function symbols (including constants) are drawn from a fixed non-empty domain. Herbrand interpretations and models are defined with respect to this Herbrand universe. Both the Herbrand universe and the Herbrand base of a functional deductive database are infinite.

Assume that $\mathcal{Z}$ is a set of functional rules and $\mathcal{D}$ is a functional database. It is well known [vEK76] that the least (Herbrand) model of $\mathcal{Z} \wedge \mathcal{D}$ is equal to the least fixpoint $LFP(\mathcal{Z}, \mathcal{D})$ of some mapping $T_p$. This result is equally true in our setting.

By our definition of Herbrand universe, the least fixpoint $LFP(\mathcal{Z}, \mathcal{D})$ contains the answers to all possible queries to $\mathcal{Z}$ and $\mathcal{D}$. That's why we limit our attention initially to the problem of finite specification of infinite least fixpoints. Later, we show how to obtain in a simple way the specifications of query answers from the specification of the least fixpoint.

## 2.3 Domain-independence

A set $\mathcal{Z}$ of functional rules is *domain-independent* if for every functional database $\mathcal{D}$, the least fixpoint $LFP(\mathcal{Z}, \mathcal{D})$ does not depend on the choice of the domain from which the function symbols (including constants) are drawn (provided the domain contains every function symbol in $\mathcal{Z}$). A discussion of similar and related notions can be found in [Kif88].

The following examples should clarify this notion.

*Domain-independent rules:*

P(s) $\rightarrow$P(**f**(s)).
P(s), R(x) $\rightarrow$P(**g**(s,x)).

*Domain-dependent rules:*

R(x).
P(s) $\rightarrow$P(**g**(s,x)).

Domain-independence of functional rules can be syntactically tested, because it is equivalent to range-restrictedness [GMN84]: every variable in a head of a rule has to appear also in the body.

**Proposition 2.1** *The least fixpoint $LFP(\mathcal{Z}, \mathcal{D})$ of a domain-independent set of rules $\mathcal{Z}$ can contain only the function symbols (including constants) appearing in $\mathcal{Z}$ or $\mathcal{D}$.*

## 2.4 Restrictions

A functional rule (query) $r$ is *normal* if $r$ contains only one functional variable and non-ground functional terms in $r$ are of depth at most 1.

For every functional rule, there is a set of normal rules (obtained through the introduction of additional predicates and rules) which is equivalent to the original set with respect to the original predicates. An example of this transformation is given in the Appendix.

When a set of rules is domain-independent, there is a simple transformation from the case of mixed function symbols to the case of pure (unary) function symbols. Take a term $\mathbf{g}(\mathbf{s},\bar{x})$ and a vector $\bar{a}$ of non-functional constants appearing in the database or in the rules(lengths of $\bar{x}$ and $\bar{a}$ are equal). If any components of $\bar{x}$ are constants, choose only such $\bar{a}$ that agree with $\bar{x}$ on those components. Create a new unary function symbol $\mathbf{f_a}$ and a new instance of every rule $r$ in $\mathcal{Z}$ where $\mathbf{g}(\mathbf{s},\bar{x})$ is replaced by $\mathbf{f_a}(\mathbf{s})$ and the occurrences of elements of $\bar{x}$ in $r$ - by the corresponding elements of $\bar{a}$. As the result of this transformation, the number and the arity of predicates will not change. The number of new rules will be a polynomial function of the size of the database. Also, this transformation preserves the normal property of rules. In section 3, we give an example of this transformation.

In the following, we will assume that rules and queries are *normal*, and that mixed function symbols have been removed by the above transformation. These assumptions have no effect on the generality of the results obtained in this paper.

## 2.5 Data complexity

We will study the impact of the database size on the size of least fixpoints. This type of complexity is termed *data complexity* [CH82].

We will identify the following parameters of a functional deductive database $\mathcal{D} \wedge \mathcal{Z}$:

- $s$ - the number of predicates in $\mathcal{Z}$ and $\mathcal{D}$.
- $k$ - the maximal arity of a predicate in $\mathcal{Z}$ and $\mathcal{D}$.
- $d$ - the number of different non-functional constants in $\mathcal{Z}$ and $\mathcal{D}$.
- $c$ - the depth of the largest functional term in $\mathcal{Z}$ and $\mathcal{D}$ (or 0 if there is no such term).

As discussed above, $s$ and $k$ will be fixed, while $c$ and $d$ will vary.

For given database $\mathcal{D}$ and a domain-independent set of rules $\mathcal{Z}$, define a *generalized* database $\mathcal{G}$ as a set of all possible tuples belonging to the extensions of the predicates from $\mathcal{Z}$ and built out of ground terms appearing in $\mathcal{D}$ and $\mathcal{Z}$. Assume that the size of the database $\mathcal{D}$ is $n$. It is clear that the size of $\mathcal{G}$ is at most $(s+1)n^{k+1}$ and is therefore polynomially related to the size of $\mathcal{D}$. If we replace the original set of rules by its normalized version, the size *gsize* of the generalized database will still be a polynomial function of the size of $\mathcal{D}$. We will use *gsize* instead of the size of $\mathcal{D}$ throughout the paper.

# 3 Finite representation

After introducing the basic notions, we are ready now to present the main result of this paper: a method to finitely represent infinite least fixpoints of functional rules as relational specifications. Subsequently, we will show that also query answers can be represented in the same way.

It should be noted that a finite and explicit representation of least fixpoints and query answers is possible only because we restrict our attention to functional rules. Least fixpoints of functional rules have a repetitive structure, but broader classes of rules don't have to share this property. We formalize repetitiveness with the notions of *state*, *state equivalence* relation and *state congruence* relation.

## 3.1 State equivalence

Define the *state equivalence* relation w.r.t. $L = LFP(\mathcal{Z}, \mathcal{D})$ (where $\mathcal{Z}$ is a domain-independent set of functional rules and $\mathcal{D}$ is a functional database):

$$\mathbf{t1} \sim \mathbf{t2} \text{ iff } \forall \bar{x}\ \mathrm{P}(\mathbf{t1}, \bar{x}) \in L \equiv \mathrm{P}(\mathbf{t2}, \bar{x}) \in L$$

where **t1** and **t2** are ground functional terms, and $\bar{x}$ is a vector of non-functional constants present in $\mathcal{Z}$ or $\mathcal{D}$.

The state-equivalence relation is indeed an equivalence relation: reflexive, symmetric and transitive. Moreover, it has only a finite number of equivalence classes for a given $L$ ($\mathcal{Z}$ is domain-independent). Once the functional component is fixed, the remaining components of predicates can assume only a finite number of values (the constants in $\mathcal{Z}$ and $\mathcal{D}$).

An equivalence class of $\sim$ will be called a *state*. Intuitively, a state behaves like a function-free database. It should be clear that knowing for each functional term its state equivalence class, one can reconstruct the entire least fixpoint $L$.

The equivalence scope, $scope_{\sim}(L)$, of $L$ is the number of equivalence classes of $\sim$ . For a given $L$, the number of tuples in a state of $L$ can not be greater than *gsize*. We can conclude that

$$scope_{\sim}(L) \leq 2^{gsize}.$$

It follows from Theorem 4.1 that

**Proposition 3.1** *$\sim$ is decidable in DEXPTIME (data complexity).*

## 3.2 State congruence

Intuitively, we would like to collapse equivalent states of the least fixpoint and obtain a finite model of $\mathcal{Z}$ and $\mathcal{D}$. But this is only possible if function symbols in $\mathcal{Z}$ can be interpreted as mapping equivalence classes to equivalence classes, i.e. when the equivalence relation is a *congruence*. This leads to the definition of finite state congruence.

A *finite state congruence* of $L = LFP(\mathcal{Z}, \mathcal{D})$ is a congruence relation $\cong$ between ground functional terms such that:

- $\cong \subseteq \sim$ (congruent terms correspond to equivalent terms).
- $\cong$ has a finite number of congruence classes (finite representation).

We will call the congruence classes of $\cong$ *clusters*.

It is clear that using a finite state congruence the least fixpoint $L$ can be finitely represented. Every cluster $\alpha$ corresponds to exactly one state $State(\alpha)$ (a state may correspond to many clusters). Because $\cong$ is a congruence, for every cluster $\alpha$, the successors $f(\alpha)$ are well defined for every function symbol **f**. Therefore it is enough to list clusters with their corresponding states and list the successor functions which map clusters to clusters. To verify whether a given tuple $\mathrm{P}(\mathbf{t}, \bar{x})$ belongs to $L$, we have to find the cluster $\alpha$ of **t** and check whether the tuple is in $State(\alpha)$ (ignoring the functional component).

The relation $\cong$ is defined as:

$\mathbf{t} \cong \mathbf{t}$ if $depth(\mathbf{t}) \leq c$.
$\mathbf{t1} \cong \mathbf{t2}$ if $depth(\mathbf{t1}) \geq c+1$, $depth(\mathbf{t2}) \geq c+1$ and $\mathbf{t1} \sim \mathbf{t2}$.

where $c$ is depth of the largest functional term in the database $\mathcal{D}$ and the rules $\mathcal{Z}$.

**Lemma 3.1** *$\cong$ defined above is a congruence.*

The number of congruence classes (clusters) of $\cong$ will be called the congruence scope $scope_{\cong}(L)$ of the least fixpoint $L$. Because the congruence $\cong$ is a refinement of the equivalence $\sim$ , we have that:

$$scope_{\sim}(L) \leq scope_{\cong}(L).$$

Note that the congruence scope, like the equivalence scope, is well defined only for domain-independent rules. Below, we show that $scope_{\cong}(L)$ is finite and at most exponential in the database size.

**Lemma 3.2** *The congruence scope of the least fixpoint $L$ of a set of domain-independent functional rules applied to a functional database*

$$scope_{\cong}(L) \leq 1 + m^c + 2^{gsize}.$$

*where $m$ is the number of successors of any state (fixed if only unary function symbols, bounded by gsize if also mixed function symbols).*

We summarize the above results in the following theorem.

**Theorem 3.1** *The relation $\cong$ is a finite state congruence.*

## 3.3 Quotient models

One way of representing $\cong$ has been already suggested: congruent terms may be collapsed. We make it precise here by defining the *quotient interpretation* of $\mathcal{Z} \wedge \mathcal{D}$. Its universe consists of congruence classes (clusters) of $\cong$ and non-functional constants from the original domain. Non-constant function symbols are interpreted as finite functions mapping clusters to clusters. We will assume that a function symbol **f** is interpreted as the function $f$. Non-functional terms are interpreted as themselves.

**Proposition 3.2** *The quotient interpretation is a non-Herbrand model of* $\mathcal{Z} \wedge \mathcal{D}$.

Thus we will call it the *quotient model* and denote by $L_{\cong}$ because it preserves the value of atomic facts in $L$.

For all functional predicates P, we have that

$$\forall\, \mathbf{t}, \bar{x},\ \mathrm{P}(\mathbf{t},\bar{x}) \in L \text{ iff } \mathrm{P}(t,\bar{x}) \in L_{\cong}$$

where the term **t** is interpreted as $t$ in $L_{\cong}$. For all non-functional predicates S, we have that

$$\forall \bar{x},\ \mathrm{S}(\bar{x}) \in L \text{ iff } \mathrm{S}(\bar{x}) \in L_{\cong}.$$

## 3.4 Graph specification

The quotient model $L_{\cong}$ will be finitely represented by choosing for every cluster a *representative* term which is the smallest of all congruent terms in the precedence ordering $\prec$. If we picture the set of functional terms as a tree, the precedence ordering corresponds to a breadth-first traversal of the tree.

*Example:* Assume there are two function symbols **f** and **g**. The precedence ordering may be as follows:

$$\mathbf{0} \prec \mathbf{f(0)} \prec \mathbf{g(0)} \prec \mathbf{f(f(0))} \prec \mathbf{g(f(0))} \prec \mathbf{f(g(0))} \ldots$$

*End of example.*

We note that a specification of the quotient model consists of two kinds of elements. First, for terms whose depth is at most $c$, clusters will consist of exactly one term. Second, for other terms clusters may (but don't have to) consist of infinitely many terms. We call the second part of the specification *repetitive*.

For every function symbol **f** we construct mappings $\mathbf{successor}_f$ from representative terms to representative terms such that

> **t** represents the cluster $\alpha$
> iff $\mathbf{successor}_f(\mathbf{t})$ represents the cluster $f(\alpha)$.

Note that $\mathbf{successor}_f(\mathbf{t})$=**f(t)** only if **f(t)** is a representative term. Denote the entire (finite) graph of successor mappings by $\mathcal{F}$.

Representative terms (and their corresponding states) form nodes and successor mappings - edges of the *graph* specification of the quotient model. In Figure 1, we show an algorithm which computes the graph specification. We should note that it computes only the *repetitive* part of the specification.

We use a Prolog-like notation. The second and fourth clauses should be repeated for any non-constant function symbol **f** appearing in $\mathcal{Z}$.

Potential(**u**) :- $depth(\mathbf{u}) = c + 1$.
Potential(**f(u)**) :- Active(**u**).
Active(**u**) :- Potential(**u**), ¬(∃ **v** Active(**v**), **v**≺**u**, **v**∼**u**).
$\mathbf{successor}_f$(**u**)=**v** :- Potential(**f(u)**), Active(**v**), **v**∼**f(u)**.

Figure 1: Algorithm **Q**: quotient model.

The algorithm **Q** traverses the least fixpoint breadth-first, starting from terms of depth $c+1$ [3]. For every cluster, there is exactly one term in Active. It is the representative term of this cluster. There is no term of smaller depth that is state-equivalent to it (clause 3). The successor mappings are pointing to the appropriate representative term(clause 4). Only the branches whose tip is in Active are further extended (clause 2). That's what we want: other branches are subsumed by those.

Define the predicate Repr(**t**) which is true only of representative terms. Notice that

Repr(**t**) iff Active(**t**) or $depth(\mathbf{t}) \leq c$.

Algorithm **Q** computes the successor mappings for all representative terms **t** such that $depth(\mathbf{t}) \geq c$. For the remaining (finitely many) representative terms, we can assume:

$\mathbf{successor}_f$(**t**)=**f(t)**.

Now for every term **t** such that **t** is a representative of the cluster $\alpha$, we can represent $State(\alpha)$ as $L[\mathbf{t}]$ (where the *slice* $L[\mathbf{t}]$ contains all the tuples in $L$ whose functional component is equal to **t**). Under the Closed World Assumption, we don't have to explicitly represent tuples *not* in the least fixpoint. Slices are effectively computable, because the yes-no query processing problem is decidable for functional rules (see Section 4). Moreover, the state equivalence relation $\sim$ is also decidable (Proposition 3.1).

Define the *primary database* $\mathcal{B}$ as consisting of all the slices $L[\mathbf{t}]$ such that **t** is a representative term of some cluster. We have that for every functional predicate P and every non-functional predicate S in $\mathcal{Z}$ and $\mathcal{D}$:

$\forall \mathbf{t}, \bar{x}$, $\mathrm{P}(\mathbf{t},\bar{x}) \in L$ iff $\mathrm{P}(\mathbf{t'},\bar{x}) \in \mathcal{B}$
and **t'** is a representative term such that $\mathbf{t} \cong \mathbf{t'}$.
$\forall \bar{x},\ \mathrm{S}(\bar{x}) \in L$ iff $\mathrm{S}(\bar{x}) \in \mathcal{B}$.

Summarizing, the *graph specification* of the least fixpoint $L$ is a pair $(\mathcal{B}, \mathcal{F})$ where $\mathcal{B}$ is the primary database and $\mathcal{F}$ is the

[3] This can be improved to $c$ for temporal rules.

entire graph of successor mappings defined on representative terms.

Once the graph specification of $L$ is computed, to verify whether $P(\mathbf{t0},\bar{a}) \in L$, we have to find the term **t** which is the representative of (the congruence class) of **t0** and check whether $P(\mathbf{t},\bar{a}) \in \mathcal{B}$. A pair $\mathbf{successor}_f(\mathbf{t1})=\mathbf{t2}$ can be seen as a tuple $S_f(\mathbf{t1},\mathbf{t2})$. In the following rules, $\mathbf{Link(t0,t)}$ is true if **t** is the representative term of **t0** (the second clause should be repeated for every non-constant function symbol in $\mathcal{Z}$):

$$\mathrm{Repr}(\mathbf{t}) \rightarrow \mathrm{Link}(\mathbf{t},\mathbf{t}).$$
$$\mathrm{Link}(\mathbf{t},\mathbf{s}),\ S_f(\mathbf{s},\mathbf{s'}) \rightarrow \mathrm{Link}(\mathbf{f(t)},\mathbf{s'}).$$

where $\mathrm{Repr}(\mathbf{t})$ holds of representative terms only and was defined earlier. The above rules will be used solely to find the representative term of **t0** and this computation always terminates.

We will follow now the work of the algorithm on the example of list construction from Section 2.

*Example:* The set of rules $\mathcal{Z}$:

$$P(x) \rightarrow \mathrm{Member}(\mathbf{ext(0,x)},x).$$
$$P(y),\ \mathrm{Member}(s,x) \rightarrow \mathrm{Member}(\mathbf{ext(s,y)},y).$$
$$P(y),\ \mathrm{Member}(s,x) \rightarrow \mathrm{Member}(\mathbf{ext(s,y)},x).$$

and the database $\mathcal{D}$:

$$P(a). \qquad P(b).$$

Note that we will have now two new function symbols $\mathbf{ext}_a$ and $\mathbf{ext}_b$ instead of **ext** (transformation from mixed to pure function symbols). The transformed set of rules:

$$P(a) \rightarrow \mathrm{Member}(\mathbf{ext}_a\mathbf{(0)},a).$$
$$P(b) \rightarrow \mathrm{Member}(\mathbf{ext}_b\mathbf{(0)},b).$$
$$P(a),\ \mathrm{Member}(s,x) \rightarrow \mathrm{Member}(\mathbf{ext}_a\mathbf{(s)},a).$$
$$P(b),\ \mathrm{Member}(s,x) \rightarrow \mathrm{Member}(\mathbf{ext}_b\mathbf{(s)},b).$$
$$P(a),\ \mathrm{Member}(s,x) \rightarrow \mathrm{Member}(\mathbf{ext}_a\mathbf{(s)},x).$$
$$P(b),\ \mathrm{Member}(s,x) \rightarrow \mathrm{Member}(\mathbf{ext}_b\mathbf{(s)},x).$$

We will use the notation $\mathbf{a1}\cdots\mathbf{ak}$ for the term $\mathbf{ext}_{a1}(\cdots(\mathbf{ext}_{ak}\mathbf{(0)})\cdots)$, i.e. the list consisting of a1,...,ak.

We obtain the following slices:

$$L[\mathbf{0}]=\emptyset.$$
$$L[\mathbf{a}]=\{\mathrm{Member}(\mathbf{a},a)\}.$$
$$L[\mathbf{aa}]=\{\mathrm{Member}(\mathbf{aa},a)\}.$$
$$L[\mathbf{b}]=\{\mathrm{Member}(\mathbf{b},b)\}.$$
$$L[\mathbf{bb}]=\{\mathrm{Member}(\mathbf{bb},b)\}.$$
$$L[\mathbf{ab}]=\{\mathrm{Member}(\mathbf{ab},a),\mathrm{Member}(\mathbf{ab},b)\}.$$
$$L[\mathbf{ba}]=\{\mathrm{Member}(\mathbf{ba},a),\mathrm{Member}(\mathbf{ba},b)\}.$$
$$L[\mathbf{aba}]=\{\mathrm{Member}(\mathbf{aba},a),\mathrm{Member}(\mathbf{aba},b)\}.$$
$$L[\mathbf{abb}]=\{\mathrm{Member}(\mathbf{abb},a),\mathrm{Member}(\mathbf{abb},b)\}.$$

Therefore $\mathbf{a}\sim\mathbf{aa}$, $\mathbf{b}\sim\mathbf{bb}$, $\mathbf{ab}\sim\mathbf{ba}$, $\mathbf{ab}\sim\mathbf{aba}$ and $\mathbf{ab}\sim\mathbf{abb}$.

It will be clear that those slices suffice for the computation of the graph specification. The precedence ordering $\prec$ between the terms [4] is, for example, as follows:

$$\mathbf{0}\prec\mathbf{a}\prec\mathbf{b}\prec\mathbf{aa}\prec\mathbf{ab}\prec\mathbf{ba}\prec\mathbf{bb}\prec\mathbf{aba}\prec\mathbf{abb}.$$

The algorithm will compute:

**Active:** a,b,ab.
**Potential:** a,b,aa,ab,ba,bb, aba and abb.
$\mathbf{successor}_{f_a}\mathbf{(a)=a}$, $\mathbf{successor}_{f_b}\mathbf{(b)=b}$
$\mathbf{successor}_{f_a}\mathbf{(b)=ab}$, $\mathbf{successor}_{f_b}\mathbf{(a)=ab}$,
$\mathbf{successor}_{f_a}\mathbf{(ab)=ab}$, $\mathbf{successor}_{f_b}\mathbf{(ab)=ab}$.

The representative terms are: **0**, **a**, **b** and **ab**. They correspond to four clusters - elements of the quotient model $L_{\cong}$. The slices corresponding to the representative terms have been determined in the course of the algorithm and together with the successor mappings constitute a finite graph specification of the least fixpoint. *End of example.*

## 3.5 Equational specification

We've shown one way of representing quotient models (and consequently least fixpoints). It was based on explicit representation of finite successor mappings. Here we show an alternative method which makes more significant use of $\cong$ being a congruence.

$\cong$ is an infinite relation. However, it can be represented as a closure $Cl(\mathcal{R})$ of a finite relation $\mathcal{R}$ defined in the following way:

- *Initialization:* $\mathcal{R}(\mathbf{t},\mathbf{t'}) \Rightarrow (\mathbf{t},\mathbf{t'}) \in Cl(\mathcal{R})$.
- *Reflexivity:* $(\mathbf{t},\mathbf{t}) \in Cl(\mathcal{R})$.
- *Symmetry:* $(\mathbf{t},\mathbf{t'}) \in Cl(\mathcal{R}) \Rightarrow (\mathbf{t'},\mathbf{t}) \in Cl(\mathcal{R})$.
- *Transitivity:* $(\mathbf{t},\mathbf{t'}) \in Cl(\mathcal{R})$ and $(\mathbf{t'},\mathbf{t''}) \in Cl(\mathcal{R}) \Rightarrow (\mathbf{t},\mathbf{t''}) \in Cl(\mathcal{R})$.
- *Congruence:* $(\mathbf{t},\mathbf{t'}) \in Cl(\mathcal{R}) \Rightarrow (\mathbf{f(t)},\mathbf{f(t')}) \in Cl(\mathcal{R})$ where **f** is a pure function symbol appearing in $\mathcal{Z}$.

Our notion of closure is the infinite version of (finite) *congruence closure* which is used in the congruence closure procedure [DST80]. Instead of a finite graph, we use an infinite one.

The *equational specification* of the least fixpoint $L$ is a pair $(\mathcal{B},\mathcal{R})$ where $\mathcal{B}$ is the primary database defined earlier in this section and $\mathcal{R}$ is a finite relation such that $Cl(\mathcal{R})$is equal to $\cong$ .

*Example:* Assume $\mathcal{D} = \{\mathrm{Even}(\mathbf{0})\}$ and $\mathcal{Z}$ contains one rule (where the function symbol is $+1$, i.e. $\mathbf{t+2}=(\mathbf{t+1})+1$):

$$\mathrm{Even}(\mathbf{t}) \rightarrow \mathrm{Even}(\mathbf{t+2}).$$

[4] Other terms are not listed in the ordering, because they will be ignored by the algorithm.

We will have $\mathcal{B} = \mathcal{D}$ and $\mathcal{R} = \{(\mathbf{0},\mathbf{2})\}$. $Cl(\mathcal{R})$ contains all pairs (u,v) such that |u-v|=2*i* for some $i = 0, 1, \ldots$. However, only those pairs where both u and v are even will be used in the derivation of the elements of $LFP(\mathcal{Z},\mathcal{D})$. In particular, every tuple Even(u) such that $(\mathbf{u},\mathbf{0}) \in Cl(\mathcal{R})$ belongs to $LFP(\mathcal{Z},\mathcal{D})$ (soundness). The opposite is also true: for every tuple Even(u)$\in LFP(\mathcal{Z},\mathcal{D})$, $(\mathbf{u},\mathbf{0}) \in Cl(\mathcal{R})$ (completeness). *End of example.*

We'd like to stress several points here.

First, $Cl(\mathcal{R})$ is not explicitly computed but implicitly present in $\mathcal{R}$. Only when after having computed $(\mathcal{B},\mathcal{R})$ we want to verify whether P($\mathbf{t0}$,$\bar{a}$)$\in L$, we compute the (finite) set $T = \{\mathbf{t} : \mathrm{P}(\mathbf{t},\bar{a}) \in \mathcal{B}\}$. Then we verify whether for some $\mathbf{t} \in T$, $(\mathbf{t0},\mathbf{t}) \in Cl(\mathcal{R})$. The last test is performed by the congruence closure procedure [DST80]. Although the entire $Cl(\mathcal{R})$ is infinite, the test $(\mathbf{t0},\mathbf{t}) \in Cl(\mathcal{R})$ needs to examine only finitely many terms, because of the finiteness of $\mathcal{B}$ and $\mathcal{R}$.

*Example:* In the last example, try to verify whether Even(**4**) and Even(**3**), i.e. whether $(\mathbf{0},\mathbf{4}) \in Cl(\mathcal{R})$ and $(\mathbf{0},\mathbf{3}) \in Cl(\mathcal{R})$. We obtain $(\mathbf{1},\mathbf{3}) \in Cl(\mathcal{R})$ and $(\mathbf{0},\mathbf{4}) \in Cl(\mathcal{R})$, but not $(\mathbf{0},\mathbf{3}) \in Cl(\mathcal{R})$. *End of example.*

Second, $(\mathcal{B},\mathcal{R})$ implicitly contains all (and only) tuples in $L$. Consequently both $\mathcal{Z}$ and $\mathcal{D}$ can be forgotten and queries answered solely on the basis of $(\mathcal{B},\mathcal{R})$.

We will now show how to obtain the relation $\mathcal{R}$. Consider Algorithm **Q** (Figure 1). We define:

$\mathcal{R}$(**t1**,**t2**) iff Active(**t1**) and Potential(**t2**) and **t1**$\sim$**t2**.

## 3.6 Canonical form

The results on finite representability in this section have another interesting interpretation: they provide a canonical form for any set of functional rules. This form is not only independent of the database but also the same for any set of original rules. Define a set of rules $CONGR$ consisting of:

- recursive rules describing the closure $\cong$ of the relation $\mathcal{R}$ between terms.
- recursive rules, one for each predicate P, of the form:

  P(s,$\bar{x}$), s$\cong$**t** $\rightarrow$P(**t**,$\bar{x}$).

The rules in $CONGR$ are not functional: the predicate $\cong$ contains two functional components.

We have that [5]

$$LFP(\mathcal{Z},\mathcal{D}) = LFP(CONGR,\mathcal{C}).$$

where the database $\mathcal{C}$ consists of $\mathcal{B}$ and $\mathcal{R}$ (both are finite sets of tuples) and is effectively computable.

[5] In $LFP(CONGR,\mathcal{C})$ we consider only the extensions of predicates from $\mathcal{Z}$ and $\mathcal{D}$.

This interesting property of functional rules, which can not be generalized to rules with arbitrary function symbols, makes them analogous to DATALOG rules. Any set of DATALOG rules has a canonical form - the least fixpoint which is always a finite relational database. Here, any set of functional rules has also a canonical representation which is always finite (although it is no longer a relational database!)

It is important to note that the set of rules $CONGR$ depends on the set of predicates in $\mathcal{Z}$, but not on the actual rules in $\mathcal{Z}$. Therefore it might be possible to develop optimization techniques for functional rules that will optimize $CONGR$ rules independently of the set of rules $\mathcal{Z}$. Techniques for optimizing the database $\mathcal{C}$ are also necessary.

# 4 Computational complexity

We will discuss here the *data* complexity of processing queries and constructing relational specifications.

**Theorem 4.1** *Yes-no query processing is complete for DEXPTIME in the case of functional rules [Fur81] and complete for PSPACE in the case of temporal rules [Pla84].*

**Theorem 4.2** *Graph specification of the quotient model $L_{\cong}$ can be computed in DEXPTIME. The upper and lower bounds on the the size of the specification are exponential.*

Note that for functional rules, it is not harder (in the sense of complexity class) to compute a finite representation of the least fixpoint than to answer a single yes-no query! For temporal rules, it is harder only if PSPACE $\subset$ DEXPTIME (as yet undetermined).

**Theorem 4.3** *Equational specification of the quotient model $L_{\cong}$ can be computed in D2EXPTIME (in DEXPTIME for temporal rules). The upper and lower bounds on the size of the specification are double exponential (single exponential for temporal rules).*

In the case of temporal terms, the relation $\mathcal{R}$ contains just one pair capturing the periodicity of the least fixpoint. The set of tuples $B$ can be, however, exponentially sized.

Note that *finite* least fixpoints can be of double exponential size. In this case, a graph specification provides a more economical representation.

# 5 Query answers

So far we have dealt with finite representations of least fixpoints. Now it is time to see whether similar techniques can be used to finitely represent query answers.

The *functional* queries that we consider in this paper are positive and contain at most one functional variable. The

variables (functional and non-functional) in a query are either free or existentially quantified. Such a query can be written as an additional functional query rule [6] i.e. rule whose body contains the definition of the query and head - a new predicate QUERY of appropriate arity.

Assume $(\mathcal{B}, \mathcal{F})$ is the graph specification [7] of $L = LFP(\mathcal{Z}, \mathcal{D})$ $(\mathcal{Z})$. There are two ways of constructing the graph specification of a query answer.

First, the query rule may be added to $\mathcal{Z}$ yielding a new set of rules $\mathcal{Z}'$ and the fixpoint $L' = LFP(\mathcal{Z}', \mathcal{D})$. The graph specification $(\mathcal{B}', \mathcal{F}')$ of $L'$ may then be constructed. Note that if a ground functional term is present in the query, it has to be replaced by the representative term from the same cluster. The extension of the predicate QUERY in $\mathcal{B}'$ and the mappings $\mathcal{F}'$ constitute the graph specification of the answer to the query.

Second, it can be observed that for many queries the query answer is finitely represented as the result of the query *applied* to $\mathcal{B}$ (call it $\mathcal{Q}(\mathcal{B})$) and $\mathcal{F}$, i.e. $(\mathcal{Q}(\mathcal{B}), \mathcal{F})$. We will call such specification *incremental.*

The second approach is preferable, because to process new queries we don't have to recompute the specification of the least fixpoint.

*Example:* In the list processing example of the previous section, assume the query is formulated as:

Member(s,a) →QUERY(s).

The incremental graph specification of the query contains the same representative terms as before: **0**, **a**, **b** and **ab**. The successor mappings are unchanged. However, the primary database is now:

QUERY(a).
QUERY(ab).

*End of example.*

We say that a functional query is *uniform* if the only non-ground functional term in it is a variable.

**Theorem 5.1** *All uniform functional queries have incremental graph specifications.*

It is possible that the answer to a query is domain-independent, while the least fixpoint is not. A method to deal with such queries would significantly broaden the application scope of our approach.

# 6 Related work and conclusions

There are many possible definitions of a query answer in deductive databases. In this paper, we start from the concept of a *ground* answer. In logic programming, the answers are not necessarily ground, but rather *most general.* However, for domain-independent rules and ground databases, ground and most general answers coincide.

Several researchers have recognized the need for higher-level, intensional, implicit query answers. However, the motivation and the applicability of various methods differ. In [Imi87b, CD87], the goal is to get an intensional representation, in terms of formulas or rules, for answers to particular queries. The approach of [CD87] tries to find a (first-order) formula that will express the query answer. It is clear that no such formula exists when recursive rules are allowed, and consequently this approach has limited applicability. The approach of [Imi87b] is more general as it can generate recursive rules in query answers. Still, the class of queries and rules which can be handled by this method is rather limited. In [CI88], we suggested that infinite least fixpoints (of temporal deductive databases) should be represented as sets of tuples with *infinite objects*: elements of the extended Herbrand universe. So every query answer was, in a sense, explicit. However, not every temporal deductive database can be handled in this way and the restrictions imposed by this method are rather arbitrary (the introductory example can not be handled). Moreover, even for simple rules, the resulting infinite objects are complex.

The methods of [Imi87b, CI88] mentioned above are syntactic in nature: they seek to *rewrite* the original set of rules $\mathcal{Z}$ as a safe set of rules $\mathcal{Z}_1$ (maybe in a different Herbrand universe). The transformation leading from $\mathcal{Z}$ to $\mathcal{Z}_1$ is performed independently of the database $\mathcal{D}$. Our method is database-dependent and semantic: we finitely encode the least fixpoint $LFP(\mathcal{Z}, \mathcal{D})$.

An interesting idea [Cor87, SM88] is to use additional information, present e.g.as taxonomies of concepts, to return a more compact and more useful answer to a query. So far, this idea has been applied only to function-free deductive databases.

Our method of finite representation of infinite answers may be seen as an analogue of *domain abstraction* [Imi87a]. Instead of dealing with infinite sets, we deal with their representatives. This abstraction is precise: atomic membership queries preserve their logical values. There is another important difference. In [Imi87a] the domain abstraction was given a priori, here it is constructed anew for each database.

Concluding, we have presented a method to finitely specify infinite query answers for an extension of DATALOG: a large and well-motivated class of rules with function symbols. The relational specifications are:

- always finite.
- effectively computable.
- explicit: individual elements of the answer can be obtained from the specification without using the original deductive rules.

Our method handles every domain-independent set of rules in this class.

[6] Or a set of rules.

[7] The same analysis can be applied to the equational specification.

# References


[CD87] L. Cholvy and R. Demolombe. Querying a rule base. In L. Kerschberg, editor, *Expert Database Systems*, pages 477–485, Benjamin/Cummings, 1987.

[CH82] A.K. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[CH85] A.K. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2(1):1–16, April 1985.

[CI88] J. Chomicki and T. Imielinski. Temporal deductive databases and infinite objects. In *ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, 1988.

[Cor87] F. Corella. Semantic retrieval and levels of abstraction. In L. Kerschberg, editor, *Expert Database Systems*, pages 477–485, Benjamin/Cummings, 1987.

[DST80] P. Downey, R. Sethi, and R.E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 1980.

[Fur81] M. Furer. Alternation and the Ackermann case of the decision problem. *L'Enseignement Mathematique*, 1981.

[GMN84] H. Gallaire, J. Minker, and J-M. Nicolas. Logic and databases: a deductive approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[Gre69] C. Green. Application of theorem proving to problem solving. In *International Joint Conference on Artificial Intelligence*, 1969.

[Imi87a] T. Imielinski. Domain abstraction and limited reasoning. In *International Joint Conference on Artificial Intelligence*, 1987.

[Imi87b] T. Imielinski. Intelligent query answering in rule based systems. *Journal of Logic Programming*, 4(2), 1987.

[Kif88] M. Kifer. On safety, domain independence and capturability of database queries. In *Data and Knowledge Base Conference*, Jerusalem, Israel, 1988.

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

[Pla84] D.A. Plaisted. Complete problems in the first-order predicate calculus. *Journal of Computer and System Sciences*, 29:8–35, 1984.

[RBS87] R. Ramakrishnan, F. Bancilhon, and A. Silberschatz. Safety of recursive Horn clauses with infinite relations. In *ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 328–339, 1987.

[SM88] C-D. Shum and R. Muntz. Implicit representation for extensional answers. In L. Kerschberg, editor, *International Conference on Expert Database Systems*, pages 257–273, George Mason University, Tysons Corner, Va., April 1988.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems.* Volume 1, Computer Science Press, 1988.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.


# APPENDIX: rule normalization

The following example should give the idea of *rule normalization*. The rule:

$$P(s),\ W(t,x) \rightarrow P(g(f(s),x)).$$

will be replaced by the following set of normal rules (P1, W1 and P12 are new predicate symbols):

$$P(s),\ W1(x) \rightarrow P12(s,x).$$
$$P1(f(s),x) \rightarrow P12(s,x).$$
$$P12(s,x) \rightarrow P1(f(s),x).$$
$$P(g(s,x)) \rightarrow P1(s,x).$$
$$P1(s,x) \rightarrow P(g(s,x)).$$
$$W(t,x) \rightarrow W1(x).$$

The *normalization* of rules is database-independent, and preserves domain-independence of rules. It yields a set of rules which is equivalent to the original one with respect to the original predicates.