A Safe Relational Calculus for Functional Logic Deductive Databases *

Jesús M. Almendros-Jiménez and Antonio Becerra-Terón

Dpto. de Lenguajes y Computación. Universidad de Almería.
email: {jalmen,abecerra}@ual.es

Abstract. In this paper, we present an extended relational calculus for expressing queries in functional-logic deductive databases. This calculus is based on first-order logic and handles relation predicates, equalities and inequalities over partially defined terms, and approximation equations. For the calculus formulas, we have studied syntactic conditions in order to ensure the domain independence property. Finally, we have studied its equivalence w.r.t. the original query language which is based on equality and inequality constraints.

1 Introduction

Database technology is involved in most software applications. For this reason *functional logic languages* [7] should include database features in order to increase its application field and cover with 'real world' applications. In order to integrate functional logic programming and databases, we propose: (1) to adapt functional logic programs to databases, by considering a suitable *data model* and a *data definition language*; (2) to *consider an extended relational calculus* as query language, which handles the proposed data model; and finally, (3) to provide *semantic foundations* to the new query language.

With respect to (1), the underlying data model of functional logic programming is *complex* from a database point of view [1]. Firstly, types can be defined by using *recursively defined datatypes*, as *lists* and *trees*. Therefore, the attribute values can be *multi-valued*; that is, more than one value (for instance, a set of values enclosed in a list) for a given attribute corresponds to each set of key attributes. In addition, we have adopted *non-deterministic semantics* from functional-logic programming, investigated in the framework *CRWL* [6]. Under non-deterministic semantics, values can be *grouped into sets*, representing the set of values of the output of a non-deterministic function. Therefore, the data model is complex in a double sense, allowing the handling of complex values built from recursively defined datatypes, and complex values grouped into sets.

Moreover, functional logic programming is able to handle *partial and possibly infinite data*. Therefore, in our setting, an attribute can be partially defined or, even, include possibly infinite information. The first case can be interpreted

* This work has been partially supported by the Spanish project of the Ministry of Science and Technology "INDALOG" TIC2002-03968.

as follows: the database includes *unknown information* o *partially defined information*; and the second case indicates that the database can store *infinite information*. In addition, database instances can be infinite (*infinite attribute values* or an *infinite set of tuples*). The infinite information can be handled by means of *partial approximations*. Moreover, we have adopted the handling of *negation* from functional logic programming, studied in the framework *CRWLF* [9]. As a consequence, the data model proposed here also handles *non-existent information*, and *partially non-existent information*.

Finally, we propose a *data definition language* which, basically, consists on *database schema definitions*, *database instance definitions* and *(lazy) function definitions*. A database schema definition includes *relation names*, and a set of *attributes* for each relation. For a given database schema, the *database instances* define *key values* and *non-key attribute values*, by means of *(constructor-based) conditional rewriting rules*, where conditions handle equality and inequality constraints. In addition, we can define a set of functions. These functions will be used by queries in order to handle recursively defined datatypes, also named *interpreted functions* in a database setting. As a consequence, "pure" functional-logic programs can be considered as a particular case of our programs.

With respect to (2), typically the query language of functional logic languages is based on the solving of conjunctions of (in)equality constraints, which are defined w.r.t. some (in)equality relations over terms [6, 9]. Our relational calculus will handle *conjunctions* of *atomic formulas*, which are *relation predicates*, *(in)equality relations* over terms, and *approximation equations* in order to handle interpreted functions. Logic formulas are either existentially or universally quantified, depending on whether they include negation or not.

However, it is known in database theory that a suitable query language must ensure the property of *domain independence* [2]. A query is domain independent, whenever the query satisfies, properly, two conditions: *(a) the query output over a finite relation is also a finite relation; and (b) the output relation only depends on the input relations.* In general, it is undecidable, and therefore syntactic conditions have to be developed in such a way that, only the so-called *safe queries* (satisfying these conditions) ensure the property of domain independence. For instance, [1] and [10] propose syntactic conditions, which allow the building of safe formulas in a relational calculus with complex values and linear constraints, respectively. In this line, we have developed syntactic conditions over our query language, which allow the building of the so-called *safe formulas*.

Extended relational calculi have been studied as alternative query languages for *deductive databases* [1], and *constraint databases* [8, 10]. Our extended relational calculus is in the line of [1], in which deductive databases handle complex values in the form of *set* and *tuple* constructors. In our case, we generalize the mentioned calculus for handling *complex values built from (arbitrary) recursively defined datatypes*. In addition, our calculus is similar to the calculi for constraint databases in the sense of allowing the handling of *infinite databases*. However, in the framework of constraint databases, infinite databases model *infinite objects* by means of *(linear) equations* and *inequations*, and *intervals*, which are handled in a symbolic way. Here, infinite databases are handled by means of *lazyness* and partial approximations. In addition, we handle constraints which consist on equality and inequality relations over complex values.

Finally, and w.r.t. (3), we will show that our relational calculus is *equivalent* to a query language based on *(in)equality constraints*, similar to existent functional logic languages. In addition, we have developed theoretical foundations for the database instances, by defining a *partial order* representing the *approximation ordering on database instances*, and a suitable *fixed point operator* which computes the *least database instance* (w.r.t. the approximation order) induced from a set of conditional rewriting rules.

Finally, remark that this work goes towards the design of a functional logic deductive language for which an operational semantics [3, 5], and a relational algebra [4] have been studied.

The organization of this paper is as follows. Section 2 describes the data model; section 3 presents the relational calculus; section 4 states the equivalence result between the relational calculus and the original query language; and section 5 defines the least database induced from a set or conditional rules.

2 The Data Model

In our framework, we consider two main kinds of partial information: undefined information (ni), represented by $\perp$, which means *information unknown, although it may exist*, and nonexistent information (ne), represented by F, which means *the information does not exist*.

Now, let's suppose a complex value, storing information about job salary and salary bonus, by means of a data constructor (like a *record*) s&b(Salary, Bonus). Then, we can additionally consider the following kinds of partial information:

s&b(3000, 100)	totally defined information, *expressing that a person's salary is* 3000 euros, *and his(her) salary bonus is* 100 euros
s&b($\perp$, 100)	partially undefined information (pni), *expressing that a person's salary bonus is known, that is* 100 euros, *but not his(her) salary*
s&b(3000, F)	partially nonexistent information (pne), *expressing that a person's salary is* 3000 euros, *but (s)he has no salary bonus*

Over these kinds of information, the (in)equality relations can be defined as follows:

- (1) = (*syntactic equality*), expressing that *two values are syntactically equal*; for instance, the relation s&b(3000, $\perp$) = s&b(3000, $\perp$) is satisfied.
- (2) $\downarrow$ (*strong equality*), expressing that *two values are equal and totally defined*; for instance, the relation s&b(3000, 25) $\downarrow$ s&b(3000, 25) holds, and the relations s&b(3000, $\perp$) $\downarrow$ s&b(3000, 25) and s&b(3000, F) $\downarrow$ s&b(3000, 25) do not hold.
- (3) $\uparrow$ (*strong inequality*), where *two values are (strongly) different, if they are different in their defined information*; for instance, the relation s&b(3000, $\perp$) $\uparrow$ s&b(2000, 25) is satisfied, whereas the relation s&b(3000, F) $\uparrow$ s&b(3000, 25) does not hold.

In addition, we will consider their logical negations, that is, $\neq$, $\not\downarrow$ and $\not\uparrow$, which represent a *syntactic inequality*, *(weak) inequality* and *(weak) equality* relation, respectively. Next, we will formally define the above equality and inequality relations.

Assuming *constructor symbols* $c, d, \ldots$ $DC = \cup_n DC^n$ each one with an associated arity, and the symbols $\perp$, F as special cases with arity 0 (not included in DC), and a set $\mathcal{V}$ of variables $X, Y, \ldots$, we can build the set of *c-terms with* $\perp$ *and* F, denoted by $CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$. C-terms are complex values including variables which implicitly are universally quantified. We can use *substitutions* $Subst_{DC,\perp,\mathsf{F}} = \{\theta \mid \theta : \mathcal{V} \rightarrow CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})\}$, in the usual way. The above (in)equality relations can be formally defined as follows.

Definition 1 (Relations over Complex Values [9]). *Given c-terms t, t': (1) $t = t'$ $\Leftrightarrow_{def}$ t and t' are syntactically equal; (2) $t \downarrow t'$ $\Leftrightarrow_{def}$ $t = t'$ and $t \in CTerm_{DC}(\mathcal{V})$; (3) $t \uparrow t'$ $\Leftrightarrow_{def}$ they have a DC-clash, where t and t' have a DC-clash whether they have different constructor symbols of DC at the same position.*
In addition, their logical negations can be defined as follows: (1') $t \neq t'$ $\Leftrightarrow_{def}$ t and t' have a $DC \cup \{\mathsf{F}\}$-clash; (2') $t \not\downarrow t'$ $\Leftrightarrow_{def}$ t or t' contains F as subterm, or they have a DC-clash; (3') $\not\uparrow$ is defined as the least symmetric relation over $CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$ satisfying: $X \not\uparrow X$ for all $X \in \mathcal{V}$, $\mathsf{F} \not\uparrow t$ for all t, and if $t_1 \not\uparrow t'_1, \ldots, t_n \not\uparrow t'_n$, then $c(t_1, \ldots, t_n) \not\uparrow c(t'_1, \ldots, t'_n)$ for $c \in DC^n$.

Given that complex values can be partially defined, a *partial ordering* $\leq$ can be considered. This ordering is defined as the least one satisfying: $\perp \leq t$, $X \leq X$, and $c(t_1, \ldots, t_n) \leq c(t'_1, \ldots, t'_n)$ if $t_i \leq t'_i$ for all $i \in \{1, \ldots, n\}$ and $c \in DC^n$. The intended meaning of $t \leq t'$ is that t is *less defined or has less information* than t'. In particular, $\perp$ is the *bottom element*, given that $\perp$ represents undefined information (ni), that is, information more refinable can exist. In addition, F is *maximal* under $\leq$ (F satisfies the relations $\perp \leq \mathsf{F}$ and $\mathsf{F} \leq \mathsf{F}$), representing nonexistent information (ne), that is, no further refinable information can be obtained, given that it does not exist. Now, we can build the set of (possibly infinite) cones of c-terms $\mathcal{C}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$, and the set of (possibly infinite) ideals of c-terms $\mathcal{I}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$. Cones and ideals can also be partially ordered under the *set-inclusion* ordering (i.e. $\subseteq$) in such a way that, the set of ideals is a *complete partial order* (*cpo*). Over cones and ideals, we can define the following *equality* and *inequality* relations.

Definition 2 (Relations over Sets of Complex Values). *Given $\mathcal{C}$ and $\mathcal{C}' \in \mathcal{C}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$: (1) $\mathcal{C} \bowtie \mathcal{C}'$ holds, whenever at least one value in $\mathcal{C}$ and $\mathcal{C}'$ is* strongly equal *and (2) $\mathcal{C} \Diamond \mathcal{C}'$ holds, whenever at least one value in $\mathcal{C}$ and $\mathcal{C}'$ is* strongly different*; and their logical negations (1') $\mathcal{C} \not\bowtie \mathcal{C}'$ holds, whenever all values in $\mathcal{C}$ and $\mathcal{C}'$ are* weakly different *and (2') $\mathcal{C} \not\Diamond \mathcal{C}'$ holds, whenever all values in $\mathcal{C}$ and $\mathcal{C}'$ are* weakly equal.

Definition 3 (Database Schemas). *Assuming a Milner's style polymorphic type system, a database schema S is a finite set of relation schemas $R_1, \ldots, R_p$*

in the form: $R(\underline{A_1} : T_1, \ldots, \underline{A_k} : T_k, A_{k+1} : T_{k+1}, \ldots, A_n : T_n)$*, wherein the relation names are a pairwise disjoint set, and the relation schemas* $R_1, \ldots, R_p$ *include a pairwise disjoint set of typed attributes*[1] $(A_1 : T_1, \ldots, A_n : T_n)$*.*

In the relation schema R, $A_1, \ldots, A_k$ are *key attributes* and $A_{k+1}, \ldots, A_n$ are *non-key attributes*, denoted by the sets $Key(R)$ and $NonKey(R)$, respectively. Key values are supposed to identify each tuple of the relation. Finally, we denote by $nAtt(R) = n$ and $nKey(R) = k$.

Definition 4 (Databases). *A database* D *is a triple* (S, DC, IF)*, where* S *is a database schema,* DC *is a set of constructor symbols, and* IF *represents a set of interpreted function symbols, each one with an associated arity.*

We denote the set of *defined schema symbols* (i.e. relation and non-key attribute symbols) by $DSS(D)$, and the set of *defined symbols* by $DS(D)$ (i.e. $DSS(D)$ together with IF). As an example of database, we can consider the following one:

$$
\begin{array}{ll}
\hline
S & \left\{ \begin{array}{l}
\mathtt{person\_job}(\underline{\mathtt{name}} : \mathtt{people}, \mathtt{age} : \mathtt{nat}, \mathtt{address} : \mathtt{dir}, \mathtt{job\_id} : \mathtt{job}, \mathtt{boss} : \mathtt{people}) \\
\mathtt{job\_information}(\underline{\mathtt{job\_name}} : \mathtt{job}, \mathtt{salary} : \mathtt{nat}, \mathtt{bonus} : \mathtt{nat}) \\
\mathtt{person\_boss\_job}(\underline{\mathtt{name}} : \mathtt{people}, \mathtt{boss\_age} : \mathtt{cbossage}, \mathtt{job\_bonus} : \mathtt{cjobbonus}) \\
\mathtt{peter\_workers}(\underline{\mathtt{name}} : \mathtt{people}, \mathtt{work} : \mathtt{job})
\end{array} \right. \\
DC & \left\{ \begin{array}{l}
\mathtt{john} : \mathtt{people},\ \mathtt{mary} : \mathtt{people},\ \mathtt{peter} : \mathtt{people} \\
\mathtt{lecturer} : \mathtt{job},\ \mathtt{associate} : \mathtt{job},\ \mathtt{professor} : \mathtt{job} \\
\mathtt{add} : \mathtt{string} \times \mathtt{nat} \rightarrow \mathtt{dir} \\
\mathtt{b\&a} : \mathtt{people} \times \mathtt{nat} \rightarrow \mathtt{cbossage} \\
\mathtt{j\&b} : \mathtt{job} \times \mathtt{nat} \rightarrow \mathtt{cjobbonus}
\end{array} \right. \\
IF & \left\{ \mathtt{retention\_for\_tax} : \mathtt{nat} \rightarrow \mathtt{nat} \right. \\
\hline
\end{array}
$$

where S includes the schemas `person_job` (storing information about people and their jobs) and `job_information` (storing generic information about jobs), and the *"views"* `person_boss_job`, and `peter_workers`, which will take key values from the set of key values defined for `person_job`. The first view includes, for each person, the pairs in the form of records constituted by: (a) his/her boss and boss' age, by using the complex c-term `b&a(people, nat)`; and (b) his/her job and job salary bonus, by using the complex c-term `j&b(job, nat)`. The second view includes `peter`'s workers. The set DC includes constructor symbols for the types `people`, `job`, `dir`, `cbossage` and `cjobbonus`, and IF defines the interpreted function symbol `retention_for_tax`, which computes the salary free of taxes. In addition, we can consider database schemas involving *(possibly) infinite databases* such as shown in the following:

$$
\begin{array}{ll}
\hline
S & \left\{ \begin{array}{l}
\mathtt{2Dpoint}(\underline{\mathtt{coord}} : \mathtt{cpoint},\ \mathtt{color} : \mathtt{nat}) \\
\mathtt{2Dline}(\underline{\mathtt{origin}} : \mathtt{cpoint},\ \underline{\mathtt{dir}} : \mathtt{orientation}, \mathtt{next} : \mathtt{cpoint}, \mathtt{points} : \mathtt{cpoint}, \\
\mathtt{list\_of\_points} : \mathtt{list}(\mathtt{cpoint}))
\end{array} \right. \\
DC & \left\{ \begin{array}{l}
\mathtt{north} : \mathtt{orientation}, \mathtt{south} : \mathtt{orientation}, \mathtt{east} : \mathtt{orientation}, \mathtt{west} : \mathtt{orientation}, \ldots \\
[\,] : \mathtt{list\ A}, \quad [\,|\,] : \mathtt{A} \times \mathtt{list\ A} \rightarrow \mathtt{list\ A} \\
\mathtt{p} : \mathtt{nat} \times \mathtt{nat} \rightarrow \mathtt{cpoint}
\end{array} \right. \\
IF & \left\{ \mathtt{select} : (\mathtt{list\ A}) \rightarrow \mathtt{A} \right. \\
\hline
\end{array}
$$

[1] We can suppose attributes qualified with the relation name when the names coincide.

wherein the schemas `2Dpoint` and `2Dline` are defined for representing bidimensional points and lines, respectively. `2Dpoint` includes the point coordinates (`coord`) and `color`. Lines represented by `2Dline` are defined by using a starting point (`origin`) and direction (`dir`). Furthermore, `next` indicates the next point to be drawn in the line, `points` stores the *(infinite) set* of points of this line, and `list_of_points` the *(infinite) list* of points of the line. Here, we can see the double use of complex values: (1) a set (which can be implicitly assumed), and (2) a list.

Definition 5 (Schema Instances). *A schema instance $\mathcal{S}$ of a database schema S is a set of relation instances $\mathcal{R}_1, \ldots \mathcal{R}_p$, where each relation instance $\mathcal{R}_j$, $1 \leq j \leq p$, is a (possibly infinite) set of tuples of the form $(V_1, \ldots, V_n)$ for the relation $R_j \in S$, with $n = nAtt(R)$ and $V_i \in \mathcal{C}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$. In particular, each V_j $(j \leq nKey(R))$ satisfies $V_j \in CTerm_{DC,\mathsf{F}}(\mathcal{V})$.*

The last condition forces the key values to be one-valued and without $\perp$. Attribute values can be non-ground, wherein the variables are implicitly universally quantified.

Definition 6 (Database Instances). *A database instance $\mathcal{D}$ of a database $D = (S, DC, IF)$ is a triple $(\mathcal{S}, \mathcal{DC}, \mathcal{IF})$, where $\mathcal{S}$ is a schema instance, $\mathcal{DC} = CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$, and $\mathcal{IF}$ is a set of function interpretations $f^{\mathcal{D}}, g^{\mathcal{D}}, \ldots$ satisfying $f^{\mathcal{D}} : CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})^n \rightarrow \mathcal{C}(CTerm_{DC,\perp,\mathsf{F}}\ (\mathcal{V}))$ is monotone, that is, $f^D(t_1, \ldots, t_n) \subseteq f^D(t'_1, \ldots, t'_n)$ if $t_i \leq t'_i$, $1 \leq i \leq n$, for each $f \in IF^n$.*

Databases can be infinite, although, as a particular case, we can consider finite databases. A schema instance $\mathcal{S}$ is *ground* if tuples only contain ground c-terms. A schema instance $\mathcal{S}$ is *finite* if it contains a finite set of tuples and finite attributes. A database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ is *finite*, whenever $\mathcal{S}$ is ground and finite, and $\mathcal{IF}$ is finite w.r.t. $\mathcal{S}$.

Next, we will show an example of schema instance for the schemas `person_job`, `job_information`, and the views `person_boss_job` and `peter_workers`:

$$
\begin{array}{ll}
\texttt{person\_job} & \left\{ \begin{array}{l} (\texttt{john}, \{\perp\}, \{\texttt{add}('\texttt{6th Avenue}', 5)\}, \{\texttt{lecturer}\}, \{\texttt{mary}, \texttt{peter}\}) \\ (\texttt{mary}, \{\perp\}, \{\texttt{add}('\texttt{7th Avenue}', 2)\}, \{\texttt{associate}\}, \{\texttt{peter}\}) \\ (\texttt{peter}, \{\perp\}, \{\texttt{add}('\texttt{5th Avenue}', 5)\}, \{\texttt{professor}\}, \{\mathsf{F}\}) \end{array} \right. \\
\texttt{job\_information} & \left\{ \begin{array}{l} (\texttt{lecturer}, \{1200\}, \{\mathsf{F}\}) \\ (\texttt{associate}, \{2000\}, \{\mathsf{F}\}) \\ (\texttt{professor}, \{3200\}, \{1500\}) \end{array} \right. \\
\texttt{person\_boss\_job} & \left\{ \begin{array}{l} (\texttt{john}, \{\texttt{b\&a}(\texttt{mary}, \perp), \texttt{b\&a}(\texttt{peter}, \perp)\}, \{\texttt{j\&b}(\texttt{lecturer}, \mathsf{F})\}) \\ (\texttt{mary}, \{\texttt{b\&a}(\texttt{peter}, \perp)\}, \{\texttt{j\&b}(\texttt{associate}, \mathsf{F})\}) \\ (\texttt{peter}, \{\texttt{b\&a}(\mathsf{F}, \perp)\}, \{\texttt{j\&b}(\texttt{professor}, 1500)\}) \end{array} \right. \\
\texttt{peter\_workers} & \left\{ \begin{array}{l} (\texttt{john}, \{\texttt{lecturer}\}) \\ (\texttt{mary}, \{\texttt{associate}\}) \end{array} \right.
\end{array}
$$

With respect to the modeling of (possibly) infinite databases, we can consider the following approximation to the instance of the relation schema `2Dline` including (*possibly infinite*) values in the defined attributes:

$$
\begin{array}{ll}
\texttt{2Dpoint} & \left\{ (\texttt{p}(0,0), \{1\}), (\texttt{p}(0,1), \{2\}), (\texttt{p}(1,0), \{\mathsf{F}\}),\ \ldots \right. \\
\texttt{2Dline} & \left\{ \begin{array}{l} (\texttt{p}(0,0), \texttt{north}, \{\texttt{p}(0,1)\}, \{\texttt{p}(0,1), \texttt{p}(0,2), \perp\}, \{[\texttt{p}(0,0), \texttt{p}(0,1), \texttt{p}(0,2)|\perp]\}),\ \ldots \\ (\texttt{p}(1,1), \texttt{east}, \{\texttt{p}(2,1)\}, \{\texttt{p}(2,1), \texttt{p}(3,1), \perp\}, \{[\texttt{p}(1,1), \texttt{p}(2,1), \texttt{p}(3,1)|\perp]\}),\ \ldots \end{array} \right.
\end{array}
$$

Instances can also be *partially ordered* as follows.

Definition 7 (Approximation Ordering on Databases). *Given a database $D = (S, DC, IF)$ and two instances $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ and $\mathcal{D}' = (\mathcal{S}', \mathcal{DC}, \mathcal{IF}')$, then $\mathcal{D} \sqsubseteq \mathcal{D}'$, if (1) $V_i \subseteq V_i'$ for each $k+1 \leq i \leq n$, $(V_1, \ldots, V_k, V_{k+1}, \ldots, V_n) \in \mathcal{R}$ and $(V_1, \ldots, V_k, V_{k+1}', \ldots, V_n') \in \mathcal{R}'$, where $\mathcal{R} \in \mathcal{S}$ and $\mathcal{R}' \in \mathcal{S}'$, are relation instances of $R \in S$ and $k = nKey(R)$; and (2) $f^{\mathcal{D}}(t_1, \ldots, t_n) \subseteq f^{\mathcal{D}'}(t_1, \ldots, t_n)$ for each $t_1, \ldots, t_n \in \mathcal{DC}$, $f^{\mathcal{D}} \in \mathcal{IF}$ and $f^{\mathcal{D}'} \in \mathcal{IF}'$.*

In particular, the *bottom database* has an empty set of tuples and each interpreted function is undefined. Instances (key and non-key values, and interpreted functions) are defined by means of *constructor-based conditional rewriting rules.*

Definition 8 (Conditional Rewriting Rules). *A constructor-based conditional rewrite rule RW for a symbol $H \in DS(D)$ has the form $H\ t_1 \ldots t_n := r \Leftarrow C$ representing that* r is the value of $H\ t_1 \ldots t_n$, whenever the condition C holds.
In this kind of rule $(t_1, \ldots, t_n)$ is a linear tuple (each variable in it occurs only once) with $t_i \in CTerm_{DC}(\mathcal{V})$, $r \in Term_D(\mathcal{V})$; C is a set of constraints of the form $e \bowtie e', e \Diamond e', e \not\bowtie e', e \not\Diamond e'$, where $e, e' \in Term_D(\mathcal{V})$ and extra variables *are not allowed, i.e. $var(r) \cup var(C) \subseteq var(\overline{t})$.*

$Term_D(\mathcal{V})$ represents the set of *terms* or expressions over a database D, and they are built from DC, $DS(D)$ and variables of $\mathcal{V}$. Each term e represents a cone (or an ideal), in such a way that, the constraints allow to compare the cones of e and e', accordingly to the semantics of the defined operators (i.e. $\bowtie, \Diamond, \not\bowtie, \not\Diamond$). For instance, the above mentioned instances can be defined by the following rules:

```
person_job        { person_job john := ok.                         person_job mary := ok.
                  { person_job peter := ok.
                  { address john := add('6th Avenue', 5).          address mary := add('7th Avenue', 2).
                  { address peter := add('5th Avenue', 5).
                  { job_id john := lecturer.                       job_id mary := associate.
                  { job_id peter := professor.
                  { boss john := mary.                             boss john := peter.
                  { boss mary := peter.
job_information   { job_information lecturer := ok.             job_information associate := ok.
                  { job_information professor := ok.
                  { salary lecturer := retention_for_tax 1500.
                  { salary associate := retention_for_tax 2500.
                  { salary professor := retention_for_tax 4000.
                  { bonus professor := 1500.
person_boss_job   { person_boss_job Name := ok ⇐ person_job Name ⋈ ok.
                  { boss_age Name := b&a(boss Name, address (boss Name)).
                  { job_bonus Name := j&b(job_id (Name), bonus (job_id (Name))).
peter_workers     { peter_workers Name := ok ⇐ person_job Name ⋈ ok, boss Name ⋈ peter.
                  { work Name := job_id Name.
retention_for_tax { retention_for_tax Fullsalary := Fullsalary − (0.2 ∗ Fullsalary).
```

The rules $R\ t_1\ \ldots\ t_k := r \Leftarrow C$, where r is a term of type `typeok`, allow the setting of $t_1, \ldots, t_k$ as key values of the relation R. `typeok` consists of a unique special value `ok` (`ok` is a shorthand of *object key*). The rules $A\ t_1\ \ldots\ t_k := r \Leftarrow C$, where $A \in NonKey(R)$, set r as the value of A for the tuple of R with key values $t_1, \ldots, t_k$. In these kinds of rules, $t_1, \ldots, t_k$ (and r) can be non-ground

Table 1. Examples of (Functional-Logic) Queries

Query	Description	Answer
	Handling of Multi-valued Attributes	
boss X $\bowtie$ peter.	*who has* peter *as boss?*	$\{$ Y/john Y/mary
address (boss X) $\bowtie$ Y, job_id X $\not\bowtie$ lecturer.	*To obtain non-*lecturer *people and their bosses' address*	$\{$ X/mary Y/add('5th Avenue', 5)
	Handling of Partial Information	
job_bonus X $\not\Diamond$ j&b(associate, Y).	*To obtain people whose all jobs are equal to* associate, *and their salary bonuses, although they do not exist*	$\{$ X/mary, Y/F
	Handling of Infinite Databases	
select (list_of_points p(0,0) Z) $\bowtie$ p(0,2).	*To obtain the orientation of the line from* p(0,0) *to* p(0,2)	$\{$ Z/north

values, and thus the key and non-key values are so too. Rules for the non-key attributes $A\ t_1\ \ldots\ t_k\ := r \Leftarrow C$ are implicitly constrained to the form $A\ t_1\ \ldots\ t_k := r \Leftarrow R\ t_1\ \ldots\ t_k \bowtie \mathsf{ok},\ C$, in order to guarantee that $t_1, \ldots, t_k$ are key values defined in a tuple of R.

As can be seen in the rules, undefined information (ni) is interpreted, whenever *there are no rules* for a given attribute. In addition, whenever the attribute is defined by rules, it is assumed that the attribute does not exist for the keys for which either the attribute is not defined or the rule conditions do not hold (i.e. nonexistent information (ne)). It fits with the failure of reduction of conditional rewriting rules [9]. Once $\perp$ and F are introduced as special cases of attribute values, the view person_boss_job will include partially undefined (pni) and partially nonexistent (pne) information. In addition, from the form of the rules for the key values of person_boss_job and peter_workers, we can consider them as views defined from person_job.

Now, we can consider (functional-logic) *queries*, which are similar to the condition of a conditional rewriting rule. For instance, table 1 shows some examples, with their corresponding meanings and expected answers.

3 Extended Relational Calculus

Next, we present the *extension of the relational calculus*, by showing its syntax, safety conditions, and, finally, its semantics.

Definition 9 (Atomic Formulas). *Given a database $D = (S, DC, IF)$, the* atomic formulas *are expressions of the form:*

1. *$R(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$, where R is a schema of S, the variables x_i's are pairwise distinct, $k = nKey(R)$, and $n = nAtt(R)$*
2. *$x = t$, where $x \in \mathcal{V}$ and $t \in CTerm_{DC}(\mathcal{V})$*
3. *$t \Downarrow t'$ or $t \Uparrow t'$, where $t, t' \in CTerm_{DC}(\mathcal{V})$*
4. *$e \triangleleft x$, where $e \in Term_{DC,IF}(\mathcal{V})^2$, and $x \in \mathcal{V}$*

[2] Terms used in the calculus equations do not include schema symbols.

(1) represents *relation predicates*, (2) the *syntactic equality*, (3) the *(strong) equality and inequality equations*, which have the same meaning as the corresponding relations (see section 2, definition 1). Finally, (4) is an *approximation equation*, representing approximation values obtained from interpreted functions.

Definition 10 (Calculus Formulas). *A calculus formula φ against an instance $\mathcal{D}$ has the form $\{x_1, \ldots, x_n \mid \phi\}$, such that ϕ is a conjunction of the form $\phi_1 \wedge \ldots \wedge \phi_n$ where each ϕ_i has the form ψ or $\neg\psi$, and each ψ is an existentially quantified conjunction of atomic formulas. Variables x_i's are the* free variables *of ϕ, denoted by $free(\phi)$. Finally, variables x_i's occurring in all $R(\bar{x})$ are distinct and the same happens to variables x's occurring in equations $e \lhd x$.*

Formulas can be built from $\forall, \rightarrow, \vee, \leftrightarrow$ whenever they are logically equivalent to the defined calculus formulas. For instance, the (functional-logic) query $\mathcal{Q}_s \equiv$ `retention_for_tax X` $\bowtie$ `salary (job_id peter)` w.r.t the database schemas `person_job` and `job_information`, requests `peter`'s full salary, and obtains `X/4000` as answer. This query can be written as follows:

$$\varphi_s \equiv \{\mathtt{x} \mid (\exists \mathtt{y_1}.\exists \mathtt{y_2}.\exists \mathtt{y_3}.\exists \mathtt{y_4}.\exists \mathtt{y_5}.\ \mathtt{person\_job}(\mathtt{y_1}, \mathtt{y_2}, \mathtt{y_3}, \mathtt{y_4}, \mathtt{y_5})\ \wedge \mathtt{y_1} = \mathtt{peter}\ \wedge \exists \mathtt{z_1}.\exists \mathtt{z_2}.\ \exists \mathtt{z_3}.\ \mathtt{job\_information}(\mathtt{z_1}, \mathtt{z_2}, \mathtt{z_3})\ \wedge\ \mathtt{z_1} = \mathtt{y_4}\ \wedge\ \exists \mathtt{u}.\ \mathtt{retention\_for\_tax\ x} \lhd \mathtt{u}\ \wedge\ \mathtt{z_2} \Downarrow \mathtt{u})\}$$

In this case, φ_s expresses to obtain the full salary, that is, `retention_for_tax x` $\lhd$ `u` and $\exists \mathtt{z_1}.\exists \mathtt{z_2}.\exists \mathtt{z_3}.\mathtt{job\_information}(\mathtt{z_1}, \mathtt{z_2}, \mathtt{z_3})\ \wedge\ \mathtt{z_2} \Downarrow \mathtt{u}$, for `peter`, that is, $\exists \mathtt{y_1}. \ldots \exists \mathtt{y_5}.\ \mathtt{person\_job}(\mathtt{y_1}, \ldots, \mathtt{y_5}) \wedge \mathtt{y_1} = \mathtt{peter} \wedge \mathtt{z_1} = \mathtt{y_4}$

In database theory, it is known that any query language must ensure the property of *domain independence* [2]. This has led to define syntactic conditions, called *safety conditions*, over the queries in such a way that the so-called *safe queries* guarantee this property. For example, in [2], the variables occurring in formulas must be *range restricted.* In our case, we generalize the notion of *range restricted* to c-terms. In addition, we require *safety conditions over atomic formulas*, and *conditions over bounded variables.*

Now, given a calculus formula φ against a database D, we define the sets: $formula\_key(\varphi) = \{x_i \mid there\ \ exists\ R(x_1, \ldots, x_i, \ldots, x_n)\ occurring\ in\ \varphi\ and\ 1 \leq i \leq nKey(R)\}$, $formula\_nonkey(\varphi) = \{x_j \mid there\ exists\ \ R(x_1, \ldots, x_j, \ldots, x_n)\ \ occurring\ in\ \ \varphi\ and\ nKey(R){+}1 \leq j \leq n\}$, and $approx(\varphi) = \{x \mid there\ exis\ ts\ \ e\ \lhd\ x\ occurring\ in\ \varphi\}$.

Definition 11 (Safe Atomic Formulas). *An atomic formula is safe in φ in the following cases:*

- *$R(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$ is safe, if the variables $x_1, \ldots, x_n$ are bound in φ, and for each x_i, $i \leq nKey(R)$, there exists one equation $x_i = t_i$ in φ*
- *$x = t$ is safe, if the variables occurring in t are distinct from the variables of $formula\_key(\varphi)$, and $x \in formula\_key(\varphi)$*
- *$t \Downarrow t'$ and $t \Uparrow t'$ are safe, if the variables occurring in t and t' are distinct from the variables of $formula\_key(\varphi)$*

Table 2. Examples of Calculus Formulas

Query	Calculus Formula
boss X $\bowtie$ peter.	$\{\mathtt{x} \mid (\exists \mathtt{y}_1.\exists \mathtt{y}_2.\exists \mathtt{y}_3.\exists \mathtt{y}_4.\exists \mathtt{y}_5.\ \mathtt{person\_job}(\mathtt{y}_1,\mathtt{y}_2,\mathtt{y}_3,\mathtt{y}_4,\mathtt{y}_5) \wedge \mathtt{y}_1 = \mathtt{x} \wedge \mathtt{y}_5 \Downarrow \mathtt{peter})\}$
address (boss X) $\bowtie$ Y, job_id X $\not\bowtie$ lecturer.	$\{\mathtt{x},\mathtt{y} \mid (\exists \mathtt{y}_1.\exists \mathtt{y}_2.\exists \mathtt{y}_3.\exists \mathtt{y}_4.\exists \mathtt{y}_5.\ \mathtt{person\_job}(\mathtt{y}_1,\mathtt{y}_2,\mathtt{y}_3,\mathtt{y}_4,\mathtt{y}_5) \wedge \mathtt{y}_1 = \mathtt{x} \wedge \exists \mathtt{z}_1. \exists \mathtt{z}_2.\exists \mathtt{z}_3.\exists \mathtt{z}_4.\exists \mathtt{z}_5.\mathtt{person\_job}(\mathtt{z}_1,\mathtt{z}_2,\mathtt{z}_3,\mathtt{z}_4,\mathtt{z}_5) \wedge \mathtt{z}_1 = \mathtt{y}_5 \wedge \mathtt{z}_3 \Downarrow \mathtt{y}) \wedge (\forall \mathtt{v}_4. ((\exists \mathtt{v}_1.\exists \mathtt{v}_2.\exists \mathtt{v}_3.\exists \mathtt{v}_5.\ \mathtt{person\_job}(\mathtt{v}_1,\mathtt{v}_2,\mathtt{v}_3,\mathtt{v}_4,\mathtt{v}_5) \wedge \mathtt{v}_1 = \mathtt{x}) \rightarrow \neg \mathtt{v}_4 \Downarrow \mathtt{lec-turer}))\}$
job_bonus X $\not\Diamond$ j&b(associate, Y).	$\{\mathtt{x},\mathtt{y} \mid (\forall \mathtt{y}_3.(\exists \mathtt{y}_1.\exists \mathtt{y}_2.\ \mathtt{person\_boss\_job}(\mathtt{y}_1,\mathtt{y}_2,\mathtt{y}_3) \wedge \mathtt{y}_1 = \mathtt{x}) \rightarrow \neg \mathtt{y}_3 \Uparrow \mathtt{j\&b}(\mathtt{associate},\mathtt{y}))\}$
select (list_of_points p(0,0) Z) $\bowtie$ p(0,2).	$\{\mathtt{z} \mid (\exists \mathtt{y}_1.\exists \mathtt{y}_2.\exists \mathtt{y}_3.\exists \mathtt{y}_4.\exists \mathtt{y}_5.\ \mathtt{2Dline}(\mathtt{y}_1,\mathtt{y}_2,\mathtt{y}_3,\mathtt{y}_4,\mathtt{y}_5) \wedge \mathtt{y}_1 = \mathtt{p}(0,0) \wedge \mathtt{y}_2 = \mathtt{z} \wedge \exists \mathtt{u}.\mathtt{select}\ \mathtt{y}_5 \lhd \mathtt{u} \wedge \mathtt{u} \Downarrow \mathtt{p}(0,2))\}$

– *$e \lhd x$ is safe, if the variables occurring in e are distinct from the variables of $formula\_key(\varphi)$, and x is bound in φ*

Definition 12 (Ranged Restricted C-Terms of Calculus Formulas). *A c-term is range restricted in a calculus formula φ if: either (a) it occurs in $formula\_key(\varphi) \cup formula\_nonkey(\varphi)$, or (b) there exists one equation $e \Diamond_c e'$ ($\Diamond_c \equiv =, \Uparrow, \Downarrow$, or $\lhd$) in φ, such that it belongs to $cterms(e)$ (resp. $cterms(e')$) and every c-term of e' (resp. e) is ranged restricted in φ.*

In the above definition, $cterms(e)$ denotes the set of c-terms occurring in e. Range restricted c-terms are variables occurring in the scope of a relation predicate or c-terms compared (by means of syntactic and strong (in)equalities, and approximation equations) with variables in the scope of a relation predicate. Therefore, all of them take values from the schema instance.

Definition 13 (Safe Formulas). *A calculus formula φ against a database D is safe, if all c-terms and atomic formulas occurring in φ are range restricted and safe, respectively and, in addition, the only bounded variables are variables of $formula\_key(\varphi) \cup formula\_nonkey(\varphi) \cup approx(\varphi)$.*

For instance, the previous φ_s is *safe*, given that the constant peter is *range restricted* (by means of $\mathtt{y}_1 = \mathtt{peter}$), and the variables u, x are also *range restricted* (by means of $\mathtt{retention\_for\_tax}\ \mathtt{x} \lhd \mathtt{u}$ and $\mathtt{z}_2 \Downarrow \mathtt{u}$). Once we have defined the conditions over the built formulas, we guarantee that they represent "queries" against a database. Negation can be used in combination with strong (in)equality relations; for instance, the calculus formula

$$\varphi_0 \equiv \neg \exists \mathtt{x}_1.\mathtt{x}_2.\mathtt{x}_3.\mathtt{x}_4.\mathtt{x}_5.\mathtt{person\_job}(\mathtt{x}_1,\ldots,\mathtt{x}_5) \wedge \mathtt{x}_1 = \mathtt{mary} \wedge \mathtt{x}_5 \Downarrow \mathtt{y}$$

requests people who are not a mary's boss. In this case, y is restricted to be a value of the column boss of the relation person_job. In this case, the answers are {y/mary} and {y/F}. Table 2 shows *(safe) calculus formulas* built from the *queries* presented in table 1. Now, we define the answers of a calculus formula. With this aim, we need to define the following notions.

Definition 14 (Denotation of Terms). *The denoted values for $e \in Term_{DC,IF}(\mathcal{V})$ in an instance $\mathcal{D}$ of a database $D = (S, DC, IF)$ w.r.t. a substitution θ, represented by $\llbracket e \rrbracket^{\mathcal{D}}\theta$, are defined as follows: $\llbracket X \rrbracket^{\mathcal{D}}\theta =_{def} < X\theta >$, for $X \in \mathcal{V}$;*

$\llbracket c(e_1, \ldots, e_n) \rrbracket^{\mathcal{D}}\theta =_{def} < c(\llbracket e_1 \rrbracket^{\mathcal{D}}\theta, \ldots, \llbracket e_n \rrbracket^{\mathcal{D}}\theta) >$[3], *for all* $c \in DC^n$; *and*
$\llbracket f\ e_1 \ \ldots\ e_n \rrbracket^{\mathcal{D}}\theta =_{def} f^{\mathcal{D}}\ \llbracket e_1 \rrbracket^{\mathcal{D}}\theta \ \ldots\ \llbracket e_n \rrbracket^{\mathcal{D}}\theta$, *for all* $f \in IF^n$

The denoted values for a term represent a cone (resp. ideal), containing the set of values which defines a non-deterministic (resp. deterministic) interpreted function.

Definition 15 (Active Domain of Terms). *The active domain of a term* $e \in Term_{DC,IF}(\mathcal{V})$ *in a calculus formula* φ *w.r.t. an instance* $\mathcal{D}$ *of database* $D = (S, DC, IF)$, *which is denoted by* $adom(e, \mathcal{D})$, *is defined as follows:*

1. $adom(x, \mathcal{D}) =_{def} \bigcup_{\psi \in Subst_{DC,\perp,\mathsf{F}}, (V_1,\ldots,V_i,\ldots,V_n) \in \mathcal{R}} V_i\psi$, *if there exists an atomic formula* $R(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n)$ *in* φ; $adom(x, \mathcal{D}) =_{def} adom(e, \mathcal{D})$ *if there exists an approximation equation* $e \triangleleft x$ *in* φ; *and* $< \perp >$, *otherwise*
2. $adom(c(e_1, \ldots, e_n), \mathcal{D}) =_{def} < c(adom(e_1, \mathcal{D}), \ldots, adom(e_n, \mathcal{D})) >$, *if* $c \in DC^n$
3. $adom(f\ e_1 \ldots e_n, \mathcal{D}) =_{def} f^{\mathcal{D}} adom(e_1, \mathcal{D}) \ldots adom(e_n, \mathcal{D})$, *if* $f \in IF^n$

The active domain of key and non-key variables contains the complete set of values of the column representing the corresponding key and non-key attributes. In the case of approximation variables, the active domain contains the complete set of values of the interpreted function. For example, the active domain of $\mathtt{x_5}$ in $\mathtt{person\_job}(\mathtt{x_1}, \ldots, \mathtt{x_5})$ is $\{\mathtt{mary}, \mathtt{peter}, \mathsf{F}\}$. The active domain is used in order to restrict the answers of a calculus formula w.r.t the schema instance. For instance the previous formula φ_0 restricts $\mathtt{y}$ to be valued in the active domain of $\mathtt{x_5}$, which is $\{\mathtt{peter}, \mathtt{mary}, \mathsf{F}\}$, and, therefore, obtaining as answers $\theta_1 = \{\mathtt{y}/\mathtt{mary}\}$ and $\theta_2 = \{\mathtt{y}/\mathsf{F}\}$. Remark that the isolated equation $\neg \mathtt{x_5} \Downarrow \mathtt{y}$ holds for $\{\mathtt{x_5}/\mathtt{peter}, \mathtt{y}/\mathtt{lecturer}\}$, w.r.t. $\not\downarrow$. However the value $\mathtt{lecturer}$ is not in the active domain of $\mathtt{x_5}$.

Remark that we have to instantiate the schema instance, whenever it includes variables (see case (1) of the above definition).

Definition 16 (Satisfiability). *Given a calculus formula* $\{\bar{x} \mid \phi\}$, *the* satisfiability *of* ϕ *in an instance* $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ *under a substitution* θ, *such that* $dom(\theta) \subseteq free(\phi)$, *(in symbols* $(\mathcal{D}, \theta) \models_C \phi$*) is defined as follows:*

- $(\mathcal{D}, \theta) \models_C R(x_1, \ldots, x_n)$, *if there exists* $(V_1, \ldots, V_n) \in \mathcal{R}$ $(\mathcal{R} \in \mathcal{S})$, *such that* $x_i\theta \in V_i\psi$ *for every* $1 \leq i \leq n$, *where* $\psi \in Subst_{DC,\perp,\mathsf{F}}$.
- $(\mathcal{D}, \theta) \models_C x = t$, *if* $x\theta \equiv t\theta$; $(\mathcal{D}, \theta) \models_C t \Downarrow t'$, *if* $t\theta \downarrow t'\theta$ *and,* $t\theta, t'\theta \in adom(t, \mathcal{D}) \cup adom(t', \mathcal{D})$; $(\mathcal{D}, \theta) \models_C t \Uparrow t'$, *if* $t\theta \uparrow t'\theta$ *and,* $t\theta, t'\theta \in adom(t, \mathcal{D}) \cup adom(t', \mathcal{D})$; *and* $(\mathcal{D}, \theta) \models_C e \triangleleft x$, *if* $x\theta \in \llbracket e \rrbracket^{\mathcal{D}}\theta$
- $(\mathcal{D}, \theta) \models_C \phi_1 \wedge \phi_2$, *if* $\mathcal{D}$ *satisfies* ϕ_1 *and* ϕ_2 *under* θ
- $(\mathcal{D}, \theta) \models_C \exists x.\phi$, *if there exists* v, *such that* $\mathcal{D}$ *satisfies* ϕ *under* $\theta \cdot \{x/v\}$
- $(\mathcal{D}, \theta) \models_C \neg\phi$, *if* $\mathcal{D}$ *does not satisfy* ϕ *under the substitution* θ

[3] To simplify denotation, we write $\{c(t_1, \ldots, t_n) \mid t_i \in C_i\}$ as $c(C_1, \ldots, C_n)$ and $\{f(t_1, \ldots, t_n) \mid t_i \in C_i\}$ as $f(C_1, \ldots, C_n)$ where $C_i's$ are certain cones.

In the formula φ_0, $\mathtt{adom}(\mathtt{x_5}, \mathcal{D}) = \{\mathtt{peter}, \mathtt{mary}, \mathsf{F}\}$ and $\mathtt{adom}(\mathtt{y}, \mathcal{D}) = \{\perp\}$. Moreover, $\theta_1 = \{\mathtt{y/mary},\ \mathtt{x_5/peter}\}$ and $\theta_2 = \{\mathtt{y}/\mathsf{F}, \mathtt{x_5/peter}\}$ holds $\mathtt{Y}\theta_1, \mathtt{Y}\theta_2 \in \mathtt{adom}(\mathtt{x_5}, \mathcal{D}) \cup \mathtt{adom}(\mathtt{y}, \mathcal{D})$; and $\mathtt{x_5}\theta_1 \not\Downarrow \mathtt{y}\theta_1$ and $\mathtt{x_5}\theta_2 \not\Downarrow \mathtt{y}\theta_2$ are satisfied.

Given a calculus formula $\varphi \equiv \{x_1, \ldots, x_n \mid \phi\}$, we define the *set of answers* of φ w.r.t. an instance $\mathcal{D}$, denoted by $Ans(\mathcal{D}, \varphi)$, as follows: $Ans(\mathcal{D}, \{x_1, \ldots, x_n | \phi\}) = \{(x_1\theta, \ldots, x_n\theta) | \theta \in Subst_{DC,\perp,\mathsf{F}}\ and\ \ (\mathcal{D}, \theta) \models_C \phi\}$. Finally, the property of *domain independence* is defined as follows.

Definition 17 (Domain Independence). *A calculus formula φ is domain independent whenever: (a) if the instance $\mathcal{D}$ is finite, then $Ans(\mathcal{D}, \varphi)$ is finite and (b) given two ground instances $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ and $\mathcal{D}' = (\mathcal{S}, \mathcal{DC}', \mathcal{IF}')$ such that $\mathcal{DC}' \supseteq \mathcal{DC}$, and $\mathcal{IF}' \supseteq \mathcal{IF}$ w.r.t. $\mathcal{S}$, then $Ans(\mathcal{D}, \varphi) = Ans(\mathcal{D}', \varphi)$.*

The case (a) establishes that the set of answers is finite whenever $\mathcal{S}$ is finite; and (b) states that the output relation (i.e. answers) depends on the input schema instance $\mathcal{S}$, and not on the domain; that is, data constructors (i.e. $\mathcal{DC}$) and interpreted functions (i.e. $\mathcal{IF}$).

Theorem 1 (Domain Independence of Calculus Formulas). *Safe calculus formulas are domain independent.*

4 Calculus Formulas and Functional Logic Queries Equivalence

In this section, we establish the equivalence between the relational calculus and the functional-logic query language. With this aim, we need to define analogous safety conditions over functional-logic queries. The set of *query keys* of a key attribute $A_i \in Key(R)$ ($R \in S$) occurring in a term $e \in Term_D(\mathcal{V})$ and denoted by $query\_key(e, A_i)$, is defined as $\{t_i \mid H\ e_1 \ldots t_i \ldots e_k\ occurs\ in\ e, H \in \{R\} \cup NonKey(R)\}$. Now, $query\_key(\mathcal{Q}) = \cup_{A_i \in Key(R)} query\_key(\mathcal{Q}, A_i)$ where $query\_key(\mathcal{Q}, A_i) = \cup_{e \Diamond_q e' \in \mathcal{Q}}\ (query\_key(e, A_i) \cup query\_key(e', A_i))$ ($\Diamond_q \equiv \bowtie$, $\Diamond$, $\not\bowtie$, or $\not\Diamond$).

A c-term t is *range restricted* in $\mathcal{Q}$, if: either (a) t belongs to $\cup_{s \in query\_key(\mathcal{Q})}$ *cterms*(s) or (b) there exists a constraint $e \Diamond_q e'$, such that t belongs to *cterms*(e) (resp. *cterms*(e')) and every c-term occurring in e' (resp. e) is *range restricted*.

Definition 18 (Safe Queries). *A query $\mathcal{Q}$ is safe if* all c-terms occurring in $\mathcal{Q}$ are range restricted.

For instance, let's consider the following query (corresponding to φ_s previously mentioned): $\mathcal{Q}_s \equiv$ `retention_for_tax X` $\bowtie$ `salary(job_id peter)`. $\mathcal{Q}_s$ is *safe*, given that the constant `peter` is *range restricted*, and thus the variable `X` is also *ranged restricted*. Analogously to the calculus formulas, we need to define the denoted values and the active domain of a database term (which includes relation names and non-key attributes) in a functional-logic query.

Definition 19 (Denotation of Database Terms). *The denotation of $e \in Term_D(\mathcal{V})$ in an instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of database $D = (S, DC, IF)$ under θ, is defined as follows:*

1. $[\![R\ e_1\ \dots\ e_k\ ]\!]^{\mathcal{D}}\theta =_{def} <\mathbf{ok}>$, *if* $([\![e_1]\!]^{\mathcal{D}}\theta,\ \dots\ ,[\![e_k]\!]^{\mathcal{D}}\theta) = (V_1\psi, \dots, V_k\psi)$ *and there exists a tuple* $(V_1, \dots, V_k, V_{k+1}, \dots, V_n) \in \mathcal{R}$, *where* $\psi \in Subst_{DC,\perp,\mathsf{F}}$, $\mathcal{R} \in \mathcal{S}$, $k = nKey(R)$; $<\perp>$ *otherwise, for all* $R \in S$;
2. $[\![A_i\ e_1\ \dots\ e_k\ ]\!]^{\mathcal{D}}\theta =_{def}\ V_i\psi$, *if* $([\![e_1]\!]^{\mathcal{D}}\theta,\ \dots\ ,[\![e_k]\!]^{\mathcal{D}}\theta) = (V_1\psi, \dots, V_k\psi)$ *and there exists a tuple* $(V_1, \dots, V_k, V_{k+1}, \dots, V_i, \dots, V_n) \in \mathcal{R}$, *where* $\psi \in Subst_{DC,\perp,\mathsf{F}}$, $\mathcal{R} \in \mathcal{S}$, *and* $i > nKey(R) = k$; $<\perp>$ *otherwise, for all* $A_i \in NonKey(R)$;
3. *And the rest of cases as in definition 14.*

Definition 20 (Active Domain of Database Terms). *The active domain of* $e \in Term_D(\mathcal{V})$ *w.r.t. an instance* $\mathcal{D}$, *and a query* $\mathcal{Q}$, *denoted by* $adom(e, \mathcal{D})$, *is defined as follows:*

- $adom(t, \mathcal{D}) =_{def} \bigcup_{\psi \in Subst_{DC,\perp,\mathsf{F}}, (V_1,\dots,V_i,\dots,V_n)\in\mathcal{R}} V_i\psi$, *if* $t \in query\_key(\mathcal{Q}, A_i)$, $A_i \in Key(R)$; *and* $<\perp>$ *otherwise, for all* $t \in CTerm_{\perp,\mathsf{F}}(\mathcal{V})$
- $adom(c(e_1, \dots, e_n), \mathcal{D}) =_{def} <c(adom(e_1, \mathcal{D}), \dots, adom(e_n, \mathcal{D}))>$, *for all* $c \in DC^n$
- $adom(f\ e_1 \dots e_n, \mathcal{D}) =_{def} f^{\mathcal{D}} adom(e_1, \mathcal{D}) \dots adom(e_n, \mathcal{D})$, *for all* $f \in IF^n$
- $adom(R\ e_1 \dots e_k, \mathcal{D}) =_{def} <\mathbf{ok}>$, *for all* $R \in S$
- $adom(A_i\ e_1 \dots e_k, \mathcal{D}) =_{def} \bigcup_{\psi \in Subst_{DC,\perp,\mathsf{F}}, (V_1,\dots,V_i,\dots,V_n)\in\mathcal{R}} V_i\psi$, *for all* $A_i \in NonKey(R)$

Both sets are also used for defining the set of query answers.

Definition 21 (Query Answers). θ *is an answer of* $\mathcal{Q}$ *w.r.t.* $\mathcal{D}$ *(in symbols* $(\mathcal{D}, \theta) \models_Q \mathcal{Q}$*) in the following cases:*

- $(\mathcal{D}, \theta) \models_Q e \bowtie e'$, *if there exist* $t \in [\![e]\!]^{\mathcal{D}}\theta$ *and* $t' \in [\![e']\!]^{\mathcal{D}}\theta$, *such that* $t \downarrow t'$, *and* $t, t' \in adom(e, \mathcal{D}) \cup adom(e', \mathcal{D})$.
- $(\mathcal{D}, \theta) \models_Q e <> e'$, *if there exist* $t \in [\![e]\!]^{\mathcal{D}}\theta$ *and* $t' \in [\![e']\!]^{\mathcal{D}}\theta$, *such that* $t \uparrow t'$, *and* $t, t' \in adom(e, \mathcal{D}) \cup adom(e', \mathcal{D})$.
- $(\mathcal{D}, \theta) \models_Q e \not\bowtie e'$ *if* $(\mathcal{D}, \theta) \not\models_Q e \bowtie e'$; *and* $(\mathcal{D}, \theta) \models_Q e \not<> e'$, *if* $(\mathcal{D}, \theta) \not\models_Q e <> e'$.

Now, the set of answers of a safe query $\mathcal{Q}$ w.r.t. an instance $\mathcal{D}$, denoted by $Ans(\mathcal{D}, \mathcal{Q})$, is defined as follows: $Ans(\mathcal{D}, \mathcal{Q}) =_{def} \{(X_1\theta, \dots, X_n\theta) \mid Dom(\theta) \subseteq var(\mathcal{Q}), (\mathcal{D}, \theta) \models_Q \mathcal{Q}, var(\mathcal{Q}) = \{X_1, \dots, X_n\}\}$. With the previous definitions, we can state the following result.

Theorem 2 (Queries and Calculus Formulas Equivalence). *Let* $\mathcal{D}$ *be an instance, then:*

- *given a safe query* $\mathcal{Q}$ *against* $\mathcal{D}$, *there exists a safe calculus formula* φ_Q *such that* $Ans(\mathcal{D}, \mathcal{Q}) = Ans(\mathcal{D}, \varphi_Q)$
- *given a safe calculus formula* φ *against* $\mathcal{D}$, *there exists a safe query* $\mathcal{Q}_\varphi$ *such that* $Ans(\mathcal{D}, \varphi) = Ans(\mathcal{D}, \mathcal{Q}_\varphi)$

Proof Sketch:
The idea is to transform a safe query into the corresponding safe calculus formula, and viceversa, by applying the set of rules of table 3. The rules distinguish two parts $\varphi \oplus \mathcal{Q}$, *where* φ *is a calculus formula and* $\mathcal{Q}$ *the query.*

Table 3. Transformation Rules

$$(1)\ \frac{\phi \wedge \exists \bar{z}.\psi \oplus e \bowtie e', \mathcal{Q}}{\phi \wedge \exists \bar{z}.\exists x.\exists y.\psi \wedge e \lhd x \ \wedge\ e' \lhd y \ \wedge\ x \Downarrow y \oplus \mathcal{Q}}$$

$$(2)\ \frac{\phi \wedge \neg\exists \bar{z}.\psi \oplus e \not\bowtie e', \mathcal{Q}}{\phi \wedge \neg\exists \bar{z}.\exists x.\exists y.\psi \wedge e \lhd x \ \wedge\ e' \lhd y \ \wedge\ x \Downarrow y \oplus \mathcal{Q}}$$

$$(3)\ \frac{\phi \wedge \exists \bar{z}.\psi \oplus e \Diamond e', \mathcal{Q}}{\phi \wedge \exists \bar{z}.\exists x.\exists y.\psi \wedge e \lhd x \ \wedge\ e' \lhd y \ \wedge\ x \Uparrow y \oplus \mathcal{Q}}$$

$$(4)\ \frac{\phi \wedge \neg\exists \bar{z}.\psi \oplus e \not\Diamond e', \mathcal{Q}}{\phi \wedge \neg\exists \bar{z}.\exists x.\exists y.\psi \wedge e \lhd x \ \wedge\ e' \lhd y \ \wedge\ x \Uparrow y \oplus \mathcal{Q}}$$

$$(5)\ \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge R\ e_1 \ldots e_k \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1.\ldots.\exists y_k.\psi \wedge R(y_1, \ldots, y_k, \ldots, y_n) \wedge e_1 \lhd y_1 \wedge \ldots \wedge e_k \lhd y_k[x|ok] \oplus \mathcal{Q}}$$
$\%\ \texttt{R} \in \texttt{S}$

$$(6)\ \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge A_i\ e_1 \ldots e_k \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1.\ldots.\exists y_k.\psi \wedge R(y_1, \ldots, y_k, \ldots, y_i, \ldots, y_n) \ \wedge\ e_1 \lhd y_1 \wedge \ldots \wedge e_k \lhd y_k \wedge y_i \lhd x \oplus \mathcal{Q}}$$
$\%\ \texttt{A}_\texttt{i} \in \texttt{NonKey(R)}$

$$(7)\ \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge f\ e_1 \ldots e_n \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1 \ldots y_n.\psi \wedge f\ y_1 \ldots y_n \lhd x \wedge e_1 \lhd y_1 \wedge \ldots \wedge e_n \lhd y_n \oplus \mathcal{Q}}$$

$$(8)\ \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge c(e_1, \ldots, e_n) \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1 \ldots y_n.\psi \wedge c(y_1, \ldots, y_n) \lhd x \wedge e_1 \lhd y_1 \wedge \ldots \wedge e_n \lhd y_n \oplus \mathcal{Q}}$$
$\%\ \texttt{c}(\texttt{e}_1, \ldots, \texttt{e}_\texttt{n})\ \texttt{is not a cterm}$

$$(9)\ \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge t \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge x = t \oplus \mathcal{Q}}$$
$\%\ \texttt{x} \in \texttt{formula\_key}(\phi \wedge (\neg)\exists \bar{z}.\psi \wedge \texttt{t} \lhd \texttt{x})$

$$(10)\ \frac{\phi \wedge (\neg)\exists \bar{z}.\exists x.\psi \wedge t \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\psi[x|t] \oplus \mathcal{Q}}$$
$\%\ \texttt{x} \notin \texttt{formula\_key}(\phi \wedge (\neg)\exists \bar{z}.\exists x.\psi \wedge \texttt{t} \lhd \texttt{x})$

5 Least Induced Database

To put an end, we will show how instances can be obtained from a set of conditional rewriting rules, by means of a *fixed point operator*, which computes the *least database induced* induced from a set of rules.

Definition 22 (Fixed Point Operator). *Given $\mathcal{A} = (\mathcal{S}^A, \mathcal{DC}^A, \mathcal{IF}^A)$ instance of a database schema $D = (S, DC, IF)$, we define a fixed point operator $T_{\mathcal{P}}(\mathcal{A}) = \mathcal{B} = (\mathcal{S}^B, \mathcal{DC}^A, \mathcal{IF}^B)$ as follows:*

- *For each schema $R(A_1, \ldots, A_n)$: $(V_1, \ldots, V_k, V_{k+1}, \ldots, V_n) \in \mathcal{R}^B, \mathcal{R}^B \in \mathcal{S}^{\mathcal{B}}$, iff $<$ **ok** $> \in T_{\mathcal{P}}(\mathcal{A}, R)\ (V_1, \ \ldots, V_k)$, $k = nKey(R)$,and $V_i = T_{\mathcal{P}}(\mathcal{A}, A_i)$ $(V_1, \ldots, V_k)$ for every $i \geq nKey(R) + 1$,*
- *For each $f \in IF$ and $t_1, \ldots, t_n \in CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$, $f^{\mathcal{B}}(t_1, \ldots, t_n) = T_{\mathcal{P}}(\mathcal{A}, f)$ $(t_1, \ldots, t_n)$, $f^{\mathcal{B}} \in \mathcal{IF}^{\mathcal{B}}$*

where given a symbol $H \in DS(D)$ and $s_1, \ldots s_n \in CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$, we define:

$$\begin{array}{rl} T_{\mathcal{P}}(\mathcal{D}, H)(s_1, \ldots, s_n) =_{def} & \{\ [\![r]\!]^{\mathcal{D}}\theta \mid \textit{if there exist } H\ \bar{t}\ := r \Leftarrow C \textit{ and } \theta, \\ & \quad \textit{such that } s_i \in [\![t_i]\!]^{\mathcal{D}}\theta \textit{ and } (\mathcal{D}, \theta) \models_Q C\} \\ & \cup \{\ \mathsf{F} \mid \textit{if there exists } H\ \bar{t}\ := r \Leftarrow C, \textit{ such that} \\ & \quad \textit{for some } i \in \{1, \ldots, n\},\ s_i \neq t_i\} \\ & \cup \{\ \mathsf{F} \mid \textit{if there exist } H\ \bar{t}\ := r \Leftarrow C \textit{ and } \theta, \\ & \quad \textit{such that } s_i \in [\![t_i]\!]^{\mathcal{D}}\theta \textit{ and } (\mathcal{D}, \theta) \not\models_Q C\} \\ & \cup \{\ \perp \mid \textit{otherwise}\} \end{array}$$

Starting from the bottom instance, then the fixed point operator computes a chain of database instances $A \sqsubseteq A' \sqsubseteq A'', \ldots$ such that the fixed point is the least instance induced by a set of rules. Finally, the following result establishes that the instance computed by means of the proposed fixed point operator is the least one satisfying the set of conditional rewriting rules.

Theorem 3 (Least Induced Database).

- *The fixed point operator* $T_{\mathcal{P}}$ *has a least fixed point* $\mathcal{L} = \mathcal{D}^{\omega}$ *where* $\mathcal{D}^0$ *is the bottom instance and* $\mathcal{D}^{k+1} = T_{\mathcal{P}}(\mathcal{D}^k)$
- *For each safe query* $\mathcal{Q}$ *and* θ*:* $(\mathcal{L}, \theta) \models_Q \mathcal{Q}$ *iff* $(\mathcal{D}, \theta) \models_Q \mathcal{Q}$ *for each* $\mathcal{D}$ *satisfying the set of rules.*

6 Conclusions and Future Work

We have studied here how to express queries by means of an (extended) relational calculus in a functional logic language integrating databases. We have shown suitable properties for such language, which are summarized in the domain independence property. As future work, we propose two main lines of research: the study of an extension of our relation calculus to be used, also, as data definition language, and the implementation of the language.

References

1. S. Abiteboul and C. Beeri. The Power of Languages for the Manipulation of Complex Values. *VLDB*, 4(4):727–794, 1995.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
3. J. M. Almendros-Jiménez and A. Becerra-Terón. A Framework for Goal-Directed Bottom-Up Evaluation of Functional Logic Programs. In *Proc. of FLOPS*, LNCS 2024, pages 153–169. Springer, 2001.
4. J. M. Almendros-Jiménez and A. Becerra-Terón. A Relational Algebra for Functional Logic Deductive Databases. In *To Appear in Procs. of Perspectives of System Informatics*, LNCS. Springer, 2003.
5. J. M. Almendros-Jiménez, A. Becerra-Terón, and J. Sánchez-Hernández. A Computational Model for Funtional Logic Deductive Databases. In *Proc. of ICLP*, LNCS 2237, pages 331–347. Springer, 2001.
6. J. C. González-Moreno, M. T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *JLP*, 1(40):47–87, 1999.
7. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *JLP*, 19,20:583–628, 1994.
8. P. Kanellakis and D. Goldin. Constraint Query Algebras. *Constraints*, 1(1–2):45–83, 1996.
9. F. J. López-Fraguas and J. Sánchez-Hernández. Proving Failure in Functional Logic Programs. In *Proc. of the CL*, LNCS 1861, pages 179–193. Springer, 2000.
10. P. Z. Revesz. Safe Query Languages for Constraint Databases. *TODS*, 23(1):58–99, 1998.