A Safe Calculus and Algebra for Querying 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 and algebra for a functional logic deductive database language. The extended relational calculus is based on the relational first-order logic, by adding constraints in the form of equalities and disequalities over complex (partially defined and possibly infinite) values and interpreted functions. In addition, we propose the notion of safety over calculus formulas in order to guarantee the property of domain independence. In the case of the extended relational algebra, we provide a generalization of the selection and projection operators in order to restructure complex values by means of constructors and destructors, interpreted functions and function inverses, as well as, to consider equality and disequality constraints. Finally, we will state that the two query formalisms (calculus and algebra) are equivalent.
Keywords: *Logic Programming, Functional Logic Programming and Deductive Databases.*

1 Introduction

Deductive databases are database management systems based on *logic programming* and *functional programming* [18, 15]. Many languages, such as DATALOG [21] and FDL [17], among others, have been developed around this research area. In most deductive database systems, the *database model* deals, as the classical *relational model*, with *relations* defined on basic values (for instance, strings, integers, etc.), wherein these relations are in *first normal form* and finite. However, the classical relational model is rather restricted for some applications, and some database paradigms have extended this model. For instance: (1) *deductive database systems which handle complex values* in the form of *sets* and *tuples* [11, 8] violate the first normal form (i.e. they allow *multi-valued attributes*), and permit *nested relations*, although the relations are finite and the querying mechanism deals with finite relations; (2) *constraint databases* [13, 14], which generalize the relational model by considering tuples as quantifier-free conjunctions of constraints over variables, allowing the handling of infinite relations (for instance, representing *spatial and temporal* data), although finitely representable, and the *querying mechanism* deals efficiently with this finite representation.

On the other hand, as in the *relational database model*, different calculi and algebras [1, 12, 19, 7] have been developed as query formalisms for these new database paradigms. These calculi and algebras extend the *relational calculus and algebra*, by adding the handling of complex values and constraints. In the case of complex values [1], the calculi extend the *relational first-order logic*, for instance, with *equality*, *membership* and *inclusion* relations over sets. In the case of constraint

* This work has been partially supported by the Spanish project "INDALOG" TIC2002-03968

databases, the calculi extend relational first-order logic, incorporating the handling of different classes of constraints, such as *equalities* and *disequalities* over integer and real numbers [12], *linear constraints* over real numbers [20], among others.

Relational calculi use a fragment of the first-order logic for expressing queries against databases. The built formulas must ensure the so-called *domain independence property.* A formula is domain independent whenever the query satisfies, properly, two properties: (1) *the query output over a finite relation is also a finite relation*; and (2) *the output relation only depends on the input relations.* However, it is undecidable whether a given calculus formula is domain independent. This has led to develop syntactic conditions over formulas, in such a way that, only the so-called *safe formulas* (satisfying these conditions) ensure the property of domain independence. In constraint deductive databases dealing with possibly infinite relations, the property of domain independence is substituted by the so-called *closed-form evaluation property.* This property means that given a set of input relations and a query expressed in terms of a conjunction of constraints over variables, then the output one can be also finitely expressed in terms of the same kind of constraints.

In both cases (deductive and constraint databases), the finiteness of the output relations from a query plays a key role, in order to take an output relation of another query as input relation. In addition, the fact of the output relation only depends on the input relations, and the same kind of constraints used by the input relations is used for expressing answers, can be intended as follows: *no additional domains (out of control of the database programmer) have to be considered by the querying mechanism.*

In the case of algebras handling complex values [1], the idea is to generalize the algebra operators allowing to restructure the complex values, including the manipulation of deeply nested components. In the case of constraint databases [12, 7], the algebra operators are extended in order to consider geometric and temporal operations, simulating the considered constraints.

In our case, we adopt *functional logic programming* [10] as base language for a deductive database system. On one hand, the integration of database technology into functional logic programming may be interesting, in order to increase its application field. A nice way of interfacing with databases is to provide a *SQL*-like syntax, more adequate and flexible for the database programmers, which allows expressing queries against a database.

On the other hand, the integration of functional logic programming and databases enriches the database model by combining and extending some of the concepts involved on relational, deductive and constraint databases, and, even, proposing some notions not presented in these database paradigms, like the handling of *(partial approximations of) infinite information* and *lazy evaluation.*

In this paper, we present a framework for integrating functional logic programming and databases, by means of a database model with the following features. The database model allows the definition of *database schemas*, which deal with complex values defined by means of (recursive) *type constructors* (such as *records*, *lists* and *trees*). The *database instances* can be infinite or, even, include *multi-valued attributes* with an *infinite set of values* or *infinite values*, and *nested relations.* The difference of our database model w.r.t. deductive databases, is that complex values are built from "arbitrary" user-defined data types, and they can be *partially defined* and *possibly infinite*. With respect to constraint databases, the difference of the database model is the handling of *constraints* over complex values, and the use of *partial approximations* in order to finitely represent the infinite values and relations. Finally, the adoption of functional logic programming allows the handling of relations and *lazy interpreted functions*, which lazily manage possibly infinite databases.

In addition, in order to make easier the interfacing with functional logic databases, we propose an extension of the relational calculus and algebra for our language. Our extended relational calculus will be based on the relational first-order logic, by adding the handling of constraints in the form of *equalities* and *disequalities* over complex (partially defined) values, and *approximation equations* over interpreted functions. As the calculi above mentioned, we propose a notion of *safety* which will ensure the property of domain independence in the following sense: (1) *the output relation of a query against finite input relations is also finite*; and (2) *the output relation of a query only depends on the input relations.* Finally, the queries against infinite input relations can have an infinite output relation.

Our extended relational algebra will analogously deal with equality and disequality constraints, complex values, and interpreted functions. With this aim, we will generalize *selection* and *projection* operators in a double sense: (a) in order to select tuples satisfying the equality and disequality constraints, and (b) in order to restructure complex values by applying data *constructors* and *destructors*, interpreted functions and their *inverses* over the attributes of a given relation.

Finally, a querying mechanism has been already presented in two recent papers [3, 5], based on the use of equalities and disequalities in the line of *CRWL* [9] and *CRWLF* [16]. This querying mechanism proposed a *goal-directed bottom-up evaluation* of a functional logic program based on *program transformation* and *magic sets* [6]. Here, we will state that the three proposed querying mechanisms (the presented one in [3, 5], calculus and algebra) are equivalent, that is, they will obtain the same answers. The work developed in this paper, together with the above quoted papers, provides the basis of the language $\mathcal{INDALOG}$ [4], currently in progress.

The rest of the paper will be organized as follows: section 2 will present the framework for partial and infinite databases; section 3 will show the extended relational calculus; section 4 will describe the relational algebra and, finally, section 5 will present the conclusions and future work. Due to the lack of space, the result proofs are omitted and can be found in a technical report at http://www.ual.es/∼jalmen.

2 A Framework for Partial and Infinite Databases

In this section, we will present a framework for the integration of functional logic programming and database technology. This framework is adapted to functional logic programming in the following sense: the underlying data model is complex, allowing the use of user-defined data types, like lists and records, and handling partial and possibly infinite information. On the other hand, this framework is adapted to the database context, equipping the functional logic programs (usually, constituted by a set of *(conditional) rewriting rules*) with database schema definitions, multi-valued attributes, nested relations, and interpreted functions.

2.1 Partial and Complex Data

With respect to the adaptation to functional logic programming, we have adopted the semantics *CRWL* (*C*onstructor Based *Re Writing* *L*ogic) [9] and *CRWLF* (*C*onstructor Based *Re Writing* *L*ogic with *F*ailure) [16]. In the context of functional logic programming, these semantics are suited for handling of *partial* and *possibly infinite* information, and *negation*, respectively. In our context, the *data model* adopts, from the cited semantics, the handling of the *partial and possibly infinite information*, and *equality and disequality relations*.

For instance, let's suppose a complex value, storing information about people's addresses (*street name and number*), by using a data constructor (like a *record*) $\mathtt{add(StreetName, StreetNumber)}$. This complex value can be totally or partially defined, as shown in the following example:

$\mathtt{add('5th\ Avenue', 25)}$	totally defined information, *expressing that a given person address is* 5th Avenue, 25
$\perp$	undefined information (ni), *expressing that the address for a given person is unknown, although it may exist*
$\mathtt{add(\perp, 25)}$	partially undefined information (pni), *expressing that we know the street number, but not the name*
F	nonexistent information (ne), *expressing that a given person has no address*
$\mathtt{add('6th\ Avenue', F)}$	partially nonexistent information (pne), *expressing that a given person address is 6th Avenue, unnumbered*

Over these kinds of information, the (dis)equality relations can be defined:

- (1) $=$ (*syntactic equality*), expressing that *two values are syntactically equal*; for instance, $\mathtt{add('5th\ Avenue', \perp)} = \mathtt{add('5th\ Avenue', \perp)}$ holds
- (2) $\downarrow$ (*strong equality*), expressing that *two values are equal and totally defined*; for instance, $\mathtt{add('5th\ Avenue', 25)} \downarrow \mathtt{add('5th\ Avenue', 25)}$ holds, and $\mathtt{add('5th\ Avenue', \perp)} \downarrow \mathtt{add('5th\ Avenue', 25)}$ and $\mathtt{add('5th\ Avenue', \mathsf{F})} \downarrow \mathtt{add}$ $\mathtt{('5th\ Avenue', 25)}$ do not hold
- (3) $\uparrow$ (*strong disequality*), where *two values are (strongly) different, if they are different in their defined information*; for instance, $\mathtt{add('6th\ Avenue', \perp)} \uparrow \mathtt{add('5th\ Avenue', 25)}$ holds, and $\mathtt{add('5th\ Avenue', \mathsf{F})} \uparrow \mathtt{add('5th\ Avenue', 25)}$ does not hold

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

Given a set of *data constructors* $c, d, \ldots$ $DC = \cup_n DC^n$ where DC^n represents the data constructors with arity n, the symbols $\perp$, F as special cases of data constructors with arity 0 (not included in DC), and a set $\mathcal{V}$ of variables $X, Y, \ldots$, then we define $CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$, called *c-terms with* $\perp$ *and* F, as the set of terms built from DC, $\perp$, F and $\mathcal{V}$. In addition, we will use *substitutions* $\theta : \mathcal{V} \rightarrow CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$ in the usual way, and we denote the domain of θ and the set of substitutions by $dom(\theta)$ and $Subst_{DC,\perp,\mathsf{F}} = \{\theta \mid \theta : \mathcal{V} \rightarrow CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})\}$, respectively.

Considering $CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$, the above (dis)equality relations can be defined as follows: (1) $t = t' \Leftrightarrow_{def} t \equiv t', t \in CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$; (2) $t \downarrow t' \Leftrightarrow_{def} t \equiv t'$ 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 $t_1 \not\uparrow t'_1, ..., t_n \not\uparrow t'_n$, then $c(t_1, ..., t_n) \not\uparrow c(t'_1, ..., t'_n)$ for $c \in DC^n$.

2.2 Database Schemas and Instances

With respect to the adaptation to databases, functional logic programs are equipped with *database schema definitions*. Each database schema definition includes a set of *relation schemas*, where every relation schema contains a *relation name*, and a set of *attributes*. Moreover, each attribute is typed either with a user-defined type or a relation name. Relation names as types mean the existence of relations with other relations, i.e. *nested relations*. Finally, each relation and attribute name is associated with a set of rules. Some of the symbols defined by rules in the program may be not associated to a relation or attribute name. These symbols specify the functional part of the program which, in the database context, represents the set of *interpreted functions*. As previously, we have also adopted the semantics *CRWL* and *CRWLF*, which are suited for the handling of *possibly infinite data*, and *non-deterministic* and *lazy functions*. In our case, the adaptation, from the semantics, allows us the handling of *multi-valued* and possibly *infinite attributes*, possibly *infinite database instances*, and *interpreted lazy functions*. Finally, the quoted semantics allow the dealing with *equality* and *disequality constraints*, involving *partial* and *non-deterministic functions*. The constraints, in our case, will be used for handling partially defined multi-valued attributes. Now, we formally define the notions of *database schema and instance* and *conditional rewriting rule*.

A *database schema* S is a finite set of *relation schemas* $R_1, \ldots, R_p$ of 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 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 the *key attributes* and $A_{k+1}, \ldots, A_n$ represent the *non-key attributes*, denoted by the sets $Key(R)$ and $NonKey(R)$, respectively. In addition, we denote by $nAtt(R) = n$ and $nKey(R) = k$.

A *database* D is a triple (S, DC, IF), where S is a *database schema*, DC is a set of *data constructors*, and IF represents a set of *interpreted function symbols* $f, g, \ldots$, 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:

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

where S includes the relation schemas `person_job` (storing information about people and their jobs) and `job_information` (storing generic information about jobs), and the *view* `person_boss_job`. This view includes, for each person, the

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

pairs 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 complement by using the complex c-term `j&c(job, nat)`. In this case, DC includes data constructors for the types `people`, `job`, `dir`, `cbossage` and `cjobcomp`, and IF the interpreted function symbol `retention_for_tax` which, given a full salary, allows computing the salary without tax.

In addition, we can consider databases including relation instances with an *infinite set of tuples*, as well as, an *infinite set of values* for a given attribute. As an example, we propose the following database for modeling spatial data:

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

wherein the relation 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 set of points of this line (i.e. an infinite set of values), and `list_of_points` the infinite list of points of the line. In this case, DC contains data constructors for the types `orientation`, `list` and `cpoint`, and IF the interpreted function symbol `select`, which allows to obtain, in a non-deterministic way, the elements of a list.

Now, 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 $\mathcal{R}$ is a (possibly infinite) set of tuples of the form $(V_1, \ldots, V_n)$, 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 $V_j \in CTerm_{DC}(\mathcal{V})$.

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 $f^{\mathcal{D}} : CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})^n \to \mathcal{C}(CTerm_{DC,\perp,\mathsf{F}}$ $(\mathcal{V}))$ is monotone for each $f \in IF^n$. In particular, the *deterministic functions* satisfy $f^{\mathcal{D}} : CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})^n \to \mathcal{I}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$.

Here, $\mathcal{C}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$ denotes the *cones* of (partial) c-terms. A cone is a set of c-terms containing all possible partial approximations of each c-term belonging to the cone. As a special case, an *ideal* is a *directed cone*, denoted by $\mathcal{I}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$. Ideals are used to provide semantics to one-valued attributes and deterministic functions, and the cones to multi-valued attributes and non-deterministic functions.

Formally, the set of c-terms $CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$ is a *partially ordered set* (in short, *poset*) with a least element *bottom* $\perp$ w.r.t. the *approximation ordering* $\leq$. This approximation ordering is defined as the least partial ordering satisfying: $\perp \leq t$, $X \leq X$, and $c(t_1, ..., t_n) \leq c(t'_1, ..., t'_n)$ if $t_i \leq t'_i$ for all $i \in \{1, ..., n\}$, $c \in DC^n$. The intended meaning of $t \leq t'$ is that t is less defined or has less information than t'; in particular $\perp$ is the *least 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. On the other hand, we can build $\mathcal{C}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$ and $\mathcal{I}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$, and they define a partial order with $\perp$ under the *set-inclusion* ordering. There is a natural mapping for each $t \in CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$ into the *principal ideal* generated by t and defined as $< t >=_{def} \{t' : t' \leq t\}$. In addition, $\mathcal{I}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$ is a *cpo* (i.e. every directed subset D has a least upper bound). Next, we show an example of schema instance for the schemas `person_job`, `job_information` and the view `person_boss_job`:

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

As can be seen, each instance tuple includes the key attribute values, and the non-key attribute values grouped by sets [2] since they can be multi-valued. The key attribute values must be *total*.

With respect to the spatial modeling example, we can consider the following (*infinite*) instances for the relation schemas `2Dpoint` and `2Dline` with (*possibly infinite*) values in their attributes:

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

In order to handle *infinite database instances*, we deal with a *finite representation* of these possibly infinite sets, considering *finite subsets of the database instance* and *partial approximations* of infinite values. For example, a subset of the instance `2Dline` with partial approximations is of the form:

$$
(\texttt{p}(0,0), \texttt{north}, \{\texttt{p}(0,1)\}, \{\texttt{p}(0,1), \texttt{p}(0,2), \perp\}, \{[\texttt{p}(0,0), \texttt{p}(0,1), \texttt{p}(0,2)|\perp]\})
$$

representing partially defined values of multi-valued and infinite attributes.

2.3 Rules and Queries

A *conditional rewrite rule* RW for a symbol $H \in DS(D)$ is of the form $H\ t_1 \ldots t_n := r \Leftarrow C$, where $(t_1, \ldots, t_n)$ is a linear tuple (each variable in it occurs only once) with $t_1, \ldots, t_n \in CTerm_{DC}(\mathcal{V})$. C is a set of constraints of the form $e \bowtie e', e \diamond e', e \not\bowtie e', e \not\diamond e'$, where $r, e, e' \in Term_D(\mathcal{V})$. $Term_D(\mathcal{V})$ defines the set of *terms* over a database D, and represents the terms built from DC, $DS(D)$ and variables of $\mathcal{V}$. Finally, *extra variables* are not allowed, i.e. $var(r) \cup var(C) \subseteq var(\bar{t})$. The reading of these rules is as follows: *r is the value of $H\ t_1 \ldots t_n$, whenever the condition C holds.*

Constraints can handle the set of values and approximations for a given (multi-valued) attribute and interpreted function. In general, they compare terms over the database. The meaning of the constraints is as follows: (1) $e \bowtie e'$ holds, whenever at least one value of e and e' are *strongly equal* and (2) $e \diamond e'$ holds, whenever at least one value of e and e' are *strongly different*; and their logical negations (1')

[2] These sets can be ideals or cones, but we will summarize their contents in the examples.

$e \not\bowtie e'$ holds, whenever all values of e and e' are not *strongly equal* and (2') $e \not\mathrel{<\!\!>} e'$ holds, whenever all values of e and e' are not *strongly different*.

For instance, the above mentioned instances for the relations `person_job`, `job_information` and the view `person_boss_job` can be defined by the 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.`	
	`complement professor := 1500.`	
`person_boss_job`	`person_boss_job Name := ok.`	
	`boss_age Name := b&a(boss Name, address (boss Name)).`	
	`job_complement Name := j&c(job_id (Name), complement (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` taking a special value `ok` (`ok` is a shorthand of *object key*), allow the setting of $t_1, \ldots, t_k$ as key values of the relation R. The rules $A\ t_1, \ldots, t_k := r \Leftarrow C$, where $A \in NonKey(R)$, set r as value of A for the tuple of R with keys $t_1, \ldots, t_k$.

As can be seen in the rules, firstly, there are *no rules for setting* the value of the attribute `age` for any key value, that is *undefined information (ni)* is being represented. In our framework, undefined information will be interpreted whenever there are no rules for a given attribute. However, if there is, at least, one rule for the attribute (for instance, in `person_job`, the attribute `boss` has rules for the key values `john` and `mary`), then the key values which have no rules, will represent *nonexistent information (ne)* in this attribute (for instance, attribute `boss` for key value `peter`, representing that `peter` has no boss). The *partially undefined (pni)* and *partially nonexistent (pne)* information is obtained from the rules for the attributes `boss_age` and `job_complement`.

Next, consider the following rules in order to obtain the above instances of relation schemas `2Dpoint` and `2Dline`:

`2Dpoint`	`2Dpoint p(0, 0) := ok.`	
	`2Dpoint p(X + 1, Y) := 2Dpoint p(X, Y).`	`2Dpoint p(X, Y + 1) := 2Dpoint p(X, Y).`
	`color p(0, 0) := 1.`	`color p(0, 1) := 2.`
`2Dline`	`2Dline X north := ok.`	`2Dline X south := ok.`
	`2Dline X east := ok.`	`2Dline X west := ok. ...`
	`next p(X, Y) north := p(X, Y + 1).`	`next p(X, Y) south := p(X, Y − 1).`
	`next p(X, Y) east := p(X + 1, Y).`	`next p(X, Y) west := p(X − 1, Y).`
`2Dline`	`2Dline X north := ok.`	`2Dline X south := ok.`
	`2Dline X east := ok.`	`2Dline X west := ok. ...`
	`next p(X, Y) north := p(X, Y + 1).`	`next p(X, Y) south := p(X, Y − 1).`
	`next p(X, Y) east := p(X + 1, Y).`	`next p(X, Y) west := p(X − 1, Y). ...`
	`points X Z := next X Z.`	`points X Z := points (next X Z) Z.`
	`list_of_points X Z := [X | list_of_points (next X Z) Z].`	
`select`	`select [X|L] := X.`	`select [X|L] := select L.`

Now, given an instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of the database $D = (S, DC, IF)$, a *query* $\mathcal{Q}$ against $\mathcal{D}$ is a set of constraints of the form $e \bowtie e'$, $e \mathrel{<\!\!>} e'$, $e \not\bowtie e'$, $e \not\mathrel{<\!\!>} e'$, where $e, e' \in Term_D(\mathcal{V})$.

Table 1. Examples of Queries

Query	Description	Answer
	Handling of Multi-valued Attributes	
boss john $\bowtie$ mary.	*Is one of* john*'s bosses* mary*?*	{ Yes
boss peter $\not\bowtie$ mary.	*Are all* peter*'s bosses different from* mary*?*	{ Yes (*Given that* peter *has no bosses, then all his bosses are different from* 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	
boss_age X $\bowtie$ b&a(mary, 14).	*To obtain people with boss and age equal to* mary, 14	{ No answer
boss_age X $\not\bowtie$ b&a(mary, 14).	*To obtain people whose all bosses and ages are different from* mary *or* 14*, although they do not exist*	{ X/mary X/peter (john *is not an answer, since one of his bosses is* mary)
job_complement X $<>$ j&c(lecturer, 1000).	*To obtain people with job and complement different from* lecturer, 1000	{ X/mary X/peter
job_complement X $\not<>$ j&c(associate, Y).	*To obtain people whose all jobs are equal to* associate*, and its salary complement, although it does 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

However, this query definition allows requesting information *out of the database domain*; that is, the queries can be *domain dependent* [2]. For instance, consider the following query $\mathcal{Q}_{ns} \equiv$ `X` $\bowtie$ `select [Y, next p(0, 1) Z]` against the schema instance `2Dline`. Here, the answers from $\mathcal{Q}_{ns}$ are {`X/Y`}, {`X/p(0, 2)`, `Z/north`}, ... However, the answer `X/Y` *depends on the domain of* `Y`; that is `X` does not range in the database domain, since `Y` does neither.

In order to ensure domain independence, [2] forces the variables to be ranged restricted. In our case, we need to generalize the range restriction to c-terms.

Now, given a query $\mathcal{Q}$ against an instance $\mathcal{D}$ of the database D, we define the set of *key c-terms (variables, constants and complex values) of* $\mathcal{Q}$, denoted by *query_key*($\mathcal{Q}$), as those ones appearing as argument of a symbol of $DSS(D)$. A c-term is *range restricted* in $\mathcal{Q}$, if it either belongs to *query_key*($\mathcal{Q}$), or there exists a constraint $e \Diamond_q e'$ ($\Diamond_q \equiv \bowtie, <>, \not\bowtie$, or $\not<>$), such that it belongs to $var(e)$ ($var(e')$) and e' (e) only includes *range restricted* c-terms. Finally, a query $\mathcal{Q}$ is *safe* w.r.t. a database D, if all c-terms occurring in $\mathcal{Q}$ are range restricted.

For instance, consider the following query $\mathcal{Q}_s \equiv$ `retention_for_tax X` $\bowtie$ `salary (job_id peter)`, against the schema instances `person_job` and `job_information`. $\mathcal{Q}_s$ requests `peter`'s full salary (i.e. salary without considering the tax), obtaining as answer `X/4000 Y/peter`. In this case, $\mathcal{Q}_s$ is *safe*, given that the constant `peter` is *range restricted* (i.e. it occurs in the scope of the defined schema symbol `job_id`), and thus the variable `X` is also *ranged restricted.* The table 1 shows different examples of *safe queries.*

In order to define the *set of answers* from a query, we need to consider the *active domain* of each term occurring in the query. The domain active is the set of values which a term can range in, and represents a subset of the schema instances and the range of interpreted functions.

Given an instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of a database $D = (S, DC, IF)$, we define the *active domain* of $e \in Term_D(\mathcal{V})$ w.r.t. $\mathcal{D}$ as follows:

- $adom(X, \mathcal{D}) =_{def} < X >$ (*principal ideal*), if $X \in \mathcal{V}$
- $adom(c(e_1, \dots, e_n), \mathcal{D}) =_{def} <c(adom(e_1, \mathcal{D}), \dots, adom(e_n, \mathcal{D}))>^3$, if $c \in DC^n$
- $adom(f\ e_1 \dots e_n, \mathcal{D}) =_{def} f^{\mathcal{D}} adom(e_1, \mathcal{D}) \dots adom(e_n, \mathcal{D})$, if $f \in IF^n$
- $adom(R\ e_1 \dots e_k, \mathcal{D}) =_{def} < \mathbf{ok} >$, if $R \in S$
- $adom(A_i\ e_1 \dots e_k, \mathcal{D}) =_{def} \cup_{(V_1, \dots, V_i, \dots, V_n) \in \mathcal{R}} V_i$, where $A_i \in NonKey(R)$

Now, given a query $\mathcal{Q}$ and a database instance $\mathcal{D}$, the *satisfiability* of $\mathcal{Q}$ in $\mathcal{D}$ under a substitution θ (in symbols $(\mathcal{D}, \theta) \models_Q \mathcal{Q}$) is defined as follows:

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

In the previous definition, $[\![e]\!]^{\mathcal{D}}\theta$ represents the *denotation* of $e \in Term_D(\mathcal{V})$ in the database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ w.r.t. a substitution θ. The *denotation* is defined as follows:

- $[\![R\ e_1\ \dots\ e_k\ ]\!]^{\mathcal{D}}\theta =_{def} < \mathbf{ok} >$, if $([\![e_1]\!]^{\mathcal{D}}\theta, \ \dots\ , [\![e_k]\!]^{\mathcal{D}}\theta) = (V_1, \dots, V_k)$ and there exists a tuple $(V_1, \dots, V_k, V_{k+1}, \dots, V_n) \in \mathcal{R}$, where $\mathcal{R} \in \mathcal{S}$, $R(A_1, \dots, A_n) \in S$ and $k = nKey(R)$; $< \perp >$ otherwise
- $[\![A_i\ e_1\ \dots\ e_k\ ]\!]^{\mathcal{D}}\theta =_{def} V_i$, if $([\![e_1]\!]^{\mathcal{D}}\theta, \ \dots\ , [\![e_k]\!]^{\mathcal{D}}\theta) = (V_1, \dots, V_k)$ and there exists a tuple $(V_1, \dots, V_k, V_{k+1}, \dots, V_i, \dots, V_n) \in \mathcal{R}$, where $\mathcal{R} \in \mathcal{S}$, $R(A_1, \dots, A_n) \in S$ and $i > nKey(R) = k$; $< \perp >$ otherwise
- $[\![X]\!]^{\mathcal{D}}\theta =_{def} < X\theta >$, for $X \in \mathcal{V}$
- $[\![c(e_1, \dots, e_n)]\!]^{\mathcal{D}}\theta =_{def} < c([\![e_1]\!]^{\mathcal{D}}\theta, \dots, [\![e_n]\!]^{\mathcal{D}}\theta) >$, for all $c \in DC^n$
- $[\![f\ e_1\ \dots\ e_n\ ]\!]^{\mathcal{D}}\theta =_{def} f^{\mathcal{D}}\ [\![e_1]\!]^{\mathcal{D}}\theta\ \dots\ [\![e_n]\!]^{\mathcal{D}}\theta$, for all $f \in IF^n$

Finally, given a query $\mathcal{Q}$, we define the *set of answers* from $\mathcal{Q}$ w.r.t. a database instance $\mathcal{D}$, denoted by $Ans(\mathcal{D}, \mathcal{Q})$, as follows: $Ans(\mathcal{D}, \mathcal{Q}) = \{(X_1\theta, \dots, X_n\theta) \mid \theta \in Subst_{DC,\perp,\mathsf{F}}(\mathcal{V})\ and\ (\mathcal{D}, \theta) \models_Q \mathcal{Q}\}$, where $\{X_1, \dots, X_n\} = var(\mathcal{Q})$.

Safe queries guarantee the property of *domain independence*. In our context, the *query domain independence* means that: *given a database instance*, (a) *if it is finite then the set of answers is finite*, and (b) *the set of answers only depends on the schema instance*. Formally, the property of *independence domain* is defined as follows: given an instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of a database $D = (S, DC, IF)$ and a query $\mathcal{Q}$, then $\mathcal{Q}$ is *domain independent* whenever:

(a) if $\mathcal{D}$ is finite, then $Ans(\mathcal{D}, \mathcal{Q})$ *is finite* and
(b) for every instance $\mathcal{D}' = (\mathcal{S}, \mathcal{DC}', \mathcal{IF}')$ of a database $D' = (S, DC', IF')$, such that $\mathcal{DC}' \supseteq \mathcal{DC}$, $\mathcal{IF}' \supseteq \mathcal{IF}$, then $Ans(\mathcal{D}, \mathcal{Q}) = Ans(\mathcal{D}', \mathcal{Q})$.

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

Theorem 1 (Query Domain Independence). *Safe queries are domain independent.*

For instance, the non-safe query $\mathcal{Q}_{ns} \equiv$ `X` $\bowtie$ `select` [`Y`, `next p(0,1) Z`] has different answers depending on the data constructors DC and interpreted function symbols IF. For example, considering the data constructors $DC = \{$`0`, `s`, [ ], [_|_]$\}$, then an answer would be `X/0 Y/0`, but the modification of $DC = \{$`john`, `0`, `s`, [ ], [_|_]$\}$ allows computing the answer `X/john Y/john`.

3 Extended Relational Calculus

Here, we present the *extension of the relational calculus*. Firstly we will show its syntax, next its safety conditions, and finally its semantics. 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 relation scheme 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. $e \lhd x$, where $e \in Term_{DC \cup IF}(\mathcal{V})$[4], and $x \in \mathcal{V}$

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

A *calculus formula* φ against a database instance $\mathcal{D}$ is of the form $\{x_1, \ldots, x_n \mid \phi\}$ such that ϕ is a conjunction of the form $\phi_1 \wedge \cdots \wedge \phi_n$, where each ϕ_i is an *existentially quantified conjunction of atomic formulas* or its logical negation, and x_i's are the free variables of ϕ, denoted by $free(\phi)$. Formulas can be built from $\forall, \rightarrow, \vee, \leftrightarrow$ whenever are logically equivalent to this kind of formulas.
For instance, the query:
$\mathcal{Q}_s \equiv$ `retention_for_tax X` $\bowtie$ `salary (job_id peter)`.
can be written as:
$\varphi_s \equiv \{\mathtt{x} \mid (\exists \mathtt{y_1}.\exists \mathtt{y_2}.\exists \mathtt{y_3}.\exists \mathtt{y_4}.\exists \mathtt{y_5}.\ \mathtt{person\_job}(\mathtt{y_1}, \mathtt{y_2}, \mathtt{y_3}, \mathtt{y_4}, \mathtt{y_5}) \wedge \mathtt{y_1} = \mathtt{peter} \wedge \exists \mathtt{z_1}.\exists \mathtt{z_2}.\exists \mathtt{z_3}.\ \mathtt{job\_information}(\mathtt{z_1}, \mathtt{z_2}, \mathtt{z_3}) \wedge \mathtt{z_1} = \mathtt{y_4} \wedge \exists \mathtt{u}.\ \mathtt{retention\_for\_tax}\ \mathtt{x} \lhd \mathtt{u} \wedge \mathtt{z_2} \Downarrow \mathtt{u})\}$.
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* $\exists \mathtt{y_1}. \ldots \exists \mathtt{y_5}.\mathtt{person\_job}(\mathtt{y_1}, \ldots, \mathtt{y_5}) \wedge \mathtt{y_1} = \mathtt{peter} \wedge \mathtt{z_1} = \mathtt{y_4}$. The table 2 shows the *calculus formulas* built from some *safe queries* presented in table 1.

Similarly to [2], in order to ensure domain independence, the *safe formulas* require: (1) *atomic formulas must be safe*, and (2) *c-terms must be range restricted.*

(1) Given a calculus formula φ against a database D, we define $formula\_key(\varphi) = \{x_i \mid$ *there exists* $R(x_1, \ldots, x_n)$ *occurring in* φ *and* $1 \leq i \leq nKey(R)\}$, and $formula\_nonkey(\varphi) = \{x_j \mid$ *there exists* $R(x_1, \ldots, x_n)$ *occurring in* φ *and* $nKey(R) + 1 \leq j \leq n\}$. Moreover we define the notion of *safe atomic formula* occurring in φ as follows:

– $R(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$ 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 $x_i = t_i \in \varphi$

[4] Terms used in the calculus equations do not include scheme symbols.

Table 2. Examples of Calculus Formulas

Safe Query	Calculus Formula
$\mathtt{boss\ john} \bowtie \mathtt{mary.}$	$\{(\exists y_1.\exists y_2.\exists y_3.\exists y_4.\exists y_5.\ \mathtt{person\_job}(y_1,y_2,y_3,y_4,y_5) \wedge y_1 = \mathtt{john} \wedge y_5 \Downarrow \mathtt{mary})\}$
$\mathtt{address\ (boss\ X)} \bowtie \mathtt{Y},$ $\mathtt{job\_id\ X} \not\bowtie \mathtt{lecturer.}$	$\{x, y \mid (\exists y_1.\exists y_2.\exists y_3.\exists y_4.\exists y_5.\ \mathtt{person\_job}(y_1,y_2,y_3,y_4,y_5) \wedge y_1 = x \wedge \exists z_1.\exists z_2.\exists z_3.\exists z_4.\exists z_5.\ \mathtt{person\_job}(z_1,z_2,z_3,z_4,z_5) \wedge z_1 = y_5 \wedge z_3 \Downarrow y) \wedge (\forall v_4.((\exists v_1.\exists v_2.\exists v_3.\exists v_5.\ \mathtt{person\_job}(v_1,v_2,v_3,v_4,v_5) \wedge v_1 = x) \rightarrow \neg v_4 \Downarrow \mathtt{lecturer}))\}$
$\mathtt{job\_complement\ X} \not\Diamond$ $\mathtt{j\&c(associate, Y).}$	$\{x, y \mid (\forall y_3.(\exists y_1.\exists y_2.\ \mathtt{person\_boss\_job}(y_1,y_2,y_3) \wedge y_1 = x) \rightarrow \neg y_3 \Uparrow \mathtt{j\&c}(\mathtt{associate}, y))\}$
$\mathtt{select\ (list\_of\_points}$ $\mathtt{p(0,0)\ Z)} \bowtie \mathtt{p(0,2).}$	$\{z \mid (\exists y_1.\exists y_2.\exists y_3.\exists y_4.\exists y_5.\ \mathtt{2Dline}(y_1,y_2,y_3,y_4,y_5) \wedge y_1 = \mathtt{p}(0,0) \wedge y_2 = z \ \wedge\ \exists u.\mathtt{select}\ y_5 \lhd u \ \wedge\ u \Downarrow \mathtt{p}(0,2))\}$

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

(2) A c-term is *range restricted* in a calculus formula φ against a database D, if either it occurs in $formula\_key(\varphi) \cup formula\_nonkey(\varphi)$, or there exists one equation $e \Diamond_c e'$ ($\Diamond_c \equiv =, \Uparrow, \Downarrow$, or $\lhd$) in φ, such that it occurs in $var(e)$ ($var(e')$) and e' (e) only includes range restricted c-terms of φ. Now, 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.

As in the case of queries, in order to define the answers from a calculus formula, we need to define the *active domain* of a term $e \in Term_{DC,IF}(\mathcal{V})$ in a calculus formula $\varphi = \{\bar{x} \mid \phi\}$ w.r.t. a database instance $\mathcal{D}$, which is denoted by $adom(e, \mathcal{D})$ and defined as follows:

- $adom(x, \mathcal{D}) =_{def} \cup_{(V_1,\ldots,V_{i-1},V_i,V_{i+1},\ldots,V_n)\in\mathcal{R}} V_i$, if $x \in formula\_key(\varphi) \cup formula\_nonkey(\varphi)$ and there exists an atomic formula $R(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n)$ in φ; $< x >$ otherwise
- $adom(c(e_1, \ldots, e_n), \mathcal{D}) =_{def} < c(adom(e_1, \mathcal{D}), \ldots, adom(e_n, \mathcal{D})) >$, if $c \in DC^n$
- $adom(f\ e_1 \ldots e_n, \mathcal{D}) =_{def} f^{\mathcal{D}} adom(e_1, \mathcal{D}) \ldots adom(e_n, \mathcal{D})$, if $f \in IF^n$

Given a calculus formula $\{\bar{x} \mid \phi\}$, the *satisfiability* of ϕ in a database instance $\mathcal{D}$ 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 $x_i\theta \in V_i$ for every $1 \leq i \leq n$
- $(\mathcal{D}, \theta) \models_C x = t$, if $x\theta \equiv t\theta$ and $x\theta \in adom(t\theta, \mathcal{D})$
- $(\mathcal{D}, \theta) \models_C t \Downarrow t'$, if $t\theta \downarrow t'\theta$, $t\theta \in adom(t'\theta, \mathcal{D})$ and $t'\theta \in adom(t\theta, \mathcal{D})$
- $(\mathcal{D}, \theta) \models_C t \Uparrow t'$, if $t\theta \uparrow t'\theta$ and, $t\theta \in adom(t'\theta, \mathcal{D})$ or $t'\theta \in adom(t\theta, \mathcal{D})$
- $(\mathcal{D}, \theta) \models_C e \lhd x$, if $x\theta \in [\![e]\!]^{\mathcal{D}}\theta \cap adom(e\theta, \mathcal{D})$
- $(\mathcal{D}, \theta) \models_C \phi_1 \wedge \phi_2$, 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

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

We finish this section with the following result establishing the equivalence between the relational calculus and query language.

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

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

As a consequence of this theorem, the safe calculus formulas ensure domain independence in the same way as the safe queries.

4 Extended Relational Algebra

Next, we present the extension of the relational algebra, which generalizes *selection and projection operators* by using *equalities and disequalities*, and *data constructors and destructors*, as well as, *interpreted functions* and their *inverses*, respectively. Let's start with some definitions:

(1) Given a set of data constructors DC, we define the set of *data destructors* DD induced from DC as the set $c.idx : T_0 \rightarrow T_{idx}$ (whenever $c : T_1 \times \cdots \times T_n \rightarrow T_0 \in DC$ and $1 \leq idx \leq n$)

(2) Given a set of (non-deterministic) interpreted functions IF, we define the set of *function inverses* FI induced from IF as the set $f.idx : T_0 \rightarrow T_{idx}$ (whenever $f : T_1 \times \cdots \times T_n \rightarrow T_0 \in IF$ and $1 \leq idx \leq n$)

From now on, given a database $D = (S, DC, IF)$ and an instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$, we will consider the *induced database and instance* as follows:

- *induced database*: $D^A = (S, DC \cup DD, IF \cup FI)$
- *induced database instance*: $\mathcal{D}^{\mathcal{A}} = (\mathcal{S}, \mathcal{DC} \cup \mathcal{DD}, \mathcal{IF} \cup \mathcal{FI})$, where $\mathcal{DD}$ is a set of interpretations $c.idx^{\mathcal{D}^{\mathcal{A}}}$, such that $c.idx^{\mathcal{D}^{\mathcal{A}}}(c(t_1, \ldots, t_n)) =< t_{idx} >$, *with* $c \in DC^n$ *and* $1 \leq idx \leq n$ and $\mathcal{FI}$ is a set of interpretations $f.idx^{\mathcal{D}^{\mathcal{A}}}$, such that $f.idx^{\mathcal{D}^{\mathcal{A}}}\ t = \{t_{idx} \mid t \in f^{\mathcal{D}}\ t_1, \ldots, t_n\}$, *with* $f \in IF^n$ *and* $1 \leq idx \leq n$.

In addition, *terms in an induced database* D^A (denoted by $Term_{DC \cup DD, IF \cup FI}(Att(D))$) are defined as the terms built from the attributes defined in D (denoted by $Att(D)$) instead of variables. Now, the *denotation* of such terms in a tuple $V = (V_1, \ldots, V_n) \in \mathcal{R}$ ($\mathcal{R} \in \mathcal{S}$), denoted by $[\![e]\!]_V^{\mathcal{D}^{\mathcal{A}}}$, is defined as follows:

- $[\![A_i]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} V_i$, if $A_i \in Att(D)$
- $[\![c(e_1, \ \ldots \ , e_n)]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} < c([\![e_1]\!]_V^{\mathcal{D}^{\mathcal{A}}}, \ \ldots \ , [\![e_n]\!]_V^{\mathcal{D}^{\mathcal{A}}}) >$, for all $c \in DC^n$, and $[\![c.idx(e)]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} c.idx^{\mathcal{D}^{\mathcal{A}}}([\![e]\!]_V^{\mathcal{D}^{\mathcal{A}}})$, for all $c.idx \in DD$
- $[\![f\ e_1\ \ldots\ e_n]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} f^{\mathcal{D}^{\mathcal{A}}}\ [\![e_1]\!]_V^{\mathcal{D}^{\mathcal{A}}}\ \ldots\ [\![e_n]\!]_V^{\mathcal{D}^{\mathcal{A}}}$, for all $f \in IF^n$, and $[\![f.idx\ e]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} f.idx^{\mathcal{D}^{\mathcal{A}}}\ [\![e]\!]_V^{\mathcal{D}^{\mathcal{A}}}$, for all $f.idx \in FI$

Moreover, we will consider the so-called *algebra terms*, which are defined as terms of the form: $\overset{=}{e}$, $\overset{\bowtie}{e}$, $\overset{\diamond}{e}$, $\overset{\neq}{e}$, $\overset{\not\bowtie}{e}$ and $\overset{\not\diamond}{e}$, where $e \in Term_{DC \cup DD, IF \cup FI}(Att(D))$. Now, given an induced instance $\mathcal{D}^{\mathcal{A}} = (\mathcal{S}, \mathcal{DC} \cup \mathcal{DD}, \mathcal{IF} \cup \mathcal{FI})$ and an algebra term e, the *denotation* of such algebra terms in a tuple $V = (V_1, \ldots, V_n) \in \mathcal{R}$ ($\mathcal{R} \in \mathcal{S}$), denoted by $[\![e]\!]_V^{\mathcal{D}^{\mathcal{A}}}$, is defined as follows:

- $[\![\overset{=}{e}]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} \{t \mid \textit{there exists } V' \in \mathcal{R}, \textit{ such that } t' \in [\![e]\!]_{V'}^{\mathcal{D}^{\mathcal{A}}} \textit{ and } t = t'\}$
- $[\![\overset{\bowtie}{e}]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} \{t \mid \textit{there exists } V' \in \mathcal{R}, \textit{ such that } t' \in [\![e]\!]_{V'}^{\mathcal{D}^{\mathcal{A}}} \textit{ and } t \downarrow t'\}$

– $[\![\overset{\Diamond}{e}]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} \{t \mid \textit{there exists } V' \in \mathcal{R} \textit{ such that } t' \in [\![e]\!]_{V'}^{\mathcal{D}^{\mathcal{A}}}, \textit{ and } t \uparrow t'\}$
– $[\![\overset{\neq}{e}]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} \{t \mid \textit{for all } V' \in \mathcal{R} \textit{ such that } t' \in [\![e]\!]_{V'}^{\mathcal{D}^{\mathcal{A}}}, \textit{ then } t \neq t' \textit{ holds}\}$
– $[\![\overset{\not\bowtie}{e}]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} \{t \mid \textit{for all } V' \in \mathcal{R} \textit{ such that } t' \in [\![e]\!]_{V'}^{\mathcal{D}^{\mathcal{A}}}, \textit{ then } t \not\downarrow t' \textit{ holds}\}$
– $[\![\overset{\not\Diamond}{e}]\!]_V^{\mathcal{D}^{\mathcal{A}}} =_{def} \{t \mid \textit{for all } V' \in \mathcal{R} \textit{ such that } t' \in [\![e]\!]_{V'}^{\mathcal{D}^{\mathcal{A}}} \textit{ then } t \not\uparrow t' \textit{ holds}\}$

Now, we can define an *algebra formula* F as follows: (1) $A = e$, $A \neq e$ where $A \in Key(D)$ (i.e. key attributes of D) and $e \in Term_{DC\cup DD,IF\cup FI}(NonKey(D))$ (i.e. terms built from non-key attributes of D); (2) $e \bowtie e'$, $e \Diamond e'$, $e \not\bowtie e'$ and $e \not\Diamond e'$, where $e,\ e' \in Term_{DC\cup DD,IF\cup FI}(Att(D))$.

Given a database $D = (S, DC, IF)$ and an instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$, a tuple $V \in \mathcal{R}$, $\mathcal{R} \in \mathcal{S}$ *satisfies an algebra formula* F if $V \models_A F$, where $\models_A$ is defined as follows:

– $V \models_A A = e$ if $[\![e]\!]_V^{\mathcal{D}^{\mathcal{A}}} \in [\![A]\!]_V^{\mathcal{D}^{\mathcal{A}}}$
– $V \models_A A \neq e$ if $[\![e]\!]_V^{\mathcal{D}^{\mathcal{A}}} \notin [\![A]\!]_V^{\mathcal{D}^{\mathcal{A}}}$
– $V \models_A e \bowtie e'$, if there exist $t \in [\![e]\!]_V^{\mathcal{D}^{\mathcal{A}}}$ and $t' \in [\![e']\!]_V^{\mathcal{D}^{\mathcal{A}}}$, such that $t \downarrow t'$
– $V \models_A e \Diamond e'$, if there exist $t \in [\![e]\!]_V^{\mathcal{D}^{\mathcal{A}}}$ and $t' \in [\![e']\!]_V^{\mathcal{D}^{\mathcal{A}}}$, such that $t \uparrow t'$
– $V \models_A e \not\bowtie e'$, if for all $t \in [\![e]\!]_V^{\mathcal{D}^{\mathcal{A}}}$ and $t' \in [\![e']\!]_V^{\mathcal{D}^{\mathcal{A}}}$, then $t \not\downarrow t'$ holds
– $V \models_A e \not\Diamond e'$, if for all $t \in [\![e]\!]_V^{\mathcal{D}^{\mathcal{A}}}$ and $t' \in [\![e']\!]_V^{\mathcal{D}^{\mathcal{A}}}$, then $t \not\uparrow t'$ holds

For instance, the query:
$\mathcal{Q}_{\mathbf{s}} \equiv$ `retention_for_tax X` $\bowtie$ `salary (job_id peter)`.
can be written as:
$\pi_{\overset{\bowtie}{\texttt{retention\_for\_tax.1(salary)}}}(\sigma_{\texttt{name=peter}}(\texttt{job\_information} \,\overline{\bowtie}_{\texttt{job\_name=job\_id}}\, \texttt{person\_job}))$

In this case, the above algebra expression $\mathcal{A}_s$ expresses the following meaning: *to join the relations* `job_information` *and* `person_job` *w.r.t. the attributes* `job_name` *and* `job_id` *(i.e.* `job_information` $\overline{\bowtie}_{\texttt{job\_name=job\_id}}$ `person_job`*), and to project* `peter`*'s (i.e.* $\sigma_{\texttt{name = peter}}$*) full salary, that is* $\pi_{\overset{\bowtie}{\texttt{retention\_for\_tax.1(salary)}}}$. Let's remark that `retention_for_tax.1` denotes the inverse in the first argument.

Now, let $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ be a database instance and $\mathcal{R}, \mathcal{Q}$ be two relation instances of $\mathcal{S}$, then the *algebra operators* are defined as follows:

Selection (σ): $\mathcal{R}' = \sigma_{F_1,\ldots,F_n}(\mathcal{R}) = \{t \in \mathcal{R} \mid t \models_A F_1, \ldots, F_n\}$
It denotes the tuple selection over $\mathcal{R}$ according to the algebra formulas $F_1, \ldots, F_n$, where:

– $F_i \cap (Key(R) \cup NonKey(R)) \neq \emptyset$
– $Key(R') = Key(R) - \cup_{1\leq i\leq n} Key(F_i)$
– $NonKey(R') = NonKey(R) \cup_{1\leq i\leq n} Key(F_i)$, ($Key(F_i)$ denotes the key attributes occurring in F_i)

Projection (π): $\mathcal{R}' = \pi_{e_1,\ldots,e_n}(\mathcal{R}) = \{([\![e_1]\!]_V^{\mathcal{D}^{\mathcal{A}}}, \ldots, [\![e_n]\!]_V^{\mathcal{D}^{\mathcal{A}}}) \mid V \in \mathcal{R}\}$
It denotes the projection over $\mathcal{R}$ according to the algebra terms $e_1, \ldots, e_n$, where:

– $e_i \cap (Key(R) \cup NonKey(R)) \neq \emptyset$
– $\{e_1, \ldots, e_n\} \supseteq Key(R)$
– $Key(R') = Key(R)$ and $NonKey(R') = \{e_1, \ldots, e_n\} - Key(R)$

Cross Product ($\times$): $\mathcal{P} = \mathcal{R} \times \mathcal{Q} = \{(t, s) \mid t \in \mathcal{R}, s \in \mathcal{Q}\}$
It denotes the cross product of the two relation instances $\mathcal{R}$ and $\mathcal{Q}$, where:

– $Key(P) = Key(R) \cup Key(Q)$

Table 3. Examples of Algebra Expressions

Safe Query	Algebra Expression
`boss john` $\bowtie$ `mary.`	$\sigma_{\texttt{name=john, boss} \bowtie \texttt{mary}}(\texttt{person\_job})$
`address (boss X)` $\bowtie$ `Y,` `job_id X` $\not\bowtie$ `lecturer.`	$\pi_{\overset{=}{\texttt{name}},\overset{\bowtie}{\texttt{address}'}}(\sigma_{\texttt{name}'=\texttt{boss, job\_id} \not\bowtie \texttt{lecturer}}(\delta_{(\rho_1)}(\texttt{person\_job} \times \texttt{person\_job})))$
`job_complement X` $<\!\!/\!\!>$ `j&c(associate,Y).`	$\pi_{\overset{=}{\texttt{name}},\overset{<\!/\!>}{\texttt{j\&c.2(job\_complement)}}}(\sigma_{\texttt{j\&c.1(job\_complement)} <\!/\!> \texttt{associate}}(\texttt{person\_boss\_job}))$
`select(list_of_points p(0,0) Z)` $\bowtie$ `p(0,2).`	$\pi_{\overset{=}{\texttt{orientation}}}(\sigma_{\texttt{origin=p(0,0), select(list\_of\_points)} \bowtie \texttt{p(0,2)}}(\texttt{2Dline}))$

$\rho_1 \equiv \{\texttt{name age} \ \ldots \ \texttt{name age} \cdots \rightarrow \texttt{name age} \ \ldots \ \texttt{name}' \ \texttt{age}' \ \ldots\}$

– $NonKey(P) = NonKey(R) \cup NonKey(Q)$

Join ($\overline{\bowtie}$)***:*** $\quad \mathcal{R} \,\overline{\bowtie}_{F_1,\ldots,F_n}\, \mathcal{Q} = \sigma_{F_1,\ldots,F_n}(\mathcal{R} \times \mathcal{Q})$

It denotes the join of the relation instances $\mathcal{R}$ and $\mathcal{Q}$ according to the algebra formulas $F_1, \ldots, F_n$ with the same conditions as the selection

Renaming (δ_ρ)***:*** $\quad \mathcal{R}' = \delta_\rho(\mathcal{R})$

It denotes an attribute renaming of the relation R of the form $A_1 A_2 \ldots A_m \rightarrow B_1 B_2 \ldots B_m$, where:

– $\rho(A_i) = B_i$, and $\rho(C) = C$ if $C \not\equiv A_i$
– $\mathcal{R}'$ contains the same tuples as $\mathcal{R}$ and the schema $R'(\rho(A_1), \ldots, \rho(A_n))$, if R has as schema $R(A_1, \ldots, A_n)$

Now, given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$, where $\mathcal{S} = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$, the *algebra expressions* Ψ are defined as expressions built by means of an algebra operator composition over a subset of the relation instances $\{\mathcal{R}_1, \ldots, \mathcal{R}_p\}$ of $\mathcal{S}$, and denoted by $\Psi(\mathcal{R}_1, \ldots, \mathcal{R}_p)$. Remark that Ψ must be *closed* w.r.t. data destructors and function inverses; that is, whenever a $c.idx$ (resp. $f.idx$) occurs in Ψ, then $c.i$ (resp. $f.i$) must occur in Ψ for each $1 \leq i \leq n$, where n is the arity of the data constructor c (resp. function symbol f). The table 3 shows the *algebra expressions* built from some *safe queries* presented in table 1.

Finally, we present the following result establishing the equivalence between the relational algebra and calculus, and therefore between the query language and the algebra.

Theorem 3 (Algebra and Calculus Equivalence). *Let* $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ *be a database instance, where* $\mathcal{S} = \{\mathcal{R}_1, \ldots, \mathcal{R}_p\}$*, then:*

– *given an algebra expression* $\Psi(\mathcal{R}_1, \ldots, \mathcal{R}_p)$ *against* $\mathcal{D}$*, then there exists a safe calculus formula* φ_Ψ *against* $\mathcal{D}$ *such that* $\Psi(\mathcal{R}_1, \ldots, \mathcal{R}_p) = Ans(\mathcal{D}, \varphi_\Psi)$*.*
– *given a safe calculus formula* φ *against* $\mathcal{D}$*, then there exists an algebra expression* $\Psi_\varphi(\mathcal{R}_1, \ldots, \mathcal{R}_p)$ *such that* $Ans(\mathcal{D}, \varphi) = \Psi_\varphi(\mathcal{R}_1, \ldots, \mathcal{R}_p)$*.*

5 Conclusions and Future Work

In this paper, we have presented an extended relational algebra and a safe calculus for a deductive database language based on functional logic programming, stating the equivalence of both formalisms with the underlying querying mechanism of the language. As future work, we will develop a high level language based on these formalisms to be included in the implementation of our language $\mathcal{INDALOG}$, and we will study an extension of the proposed calculus and algebra in order to handle aggregation.

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. INDALOG: A Declarative Deductive Database Language. In *Proc. of PROLE*, pages 113–128. Universidad de Castilla-La Mancha, 2001.
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. C. Beeri and R. Ramakrishnan. On the Power of Magic. *JLP*, 10(3,4):255–299, 1991.
7. A. Belussi, E. Bertino, and B. Catania. An Extended Algebra for Constraint Databases. *TKDE*, 10(5):686–705, 1998.
8. D. Chimenti, R. Gamboa, R. Krishnamurthy, S. A. Naqvi, S. Tsur, and C. Zaniolo. The LDL System Prototype. *TKDE*, 2(1):76–90, 1990.
9. 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.
10. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *JLP*, 19,20:583–628, 1994.
11. R. Hull and J. Su. Deductive Query Language for Recursively Typed Complex Objects. *JLP*, 35(3):231–261, 1998.
12. P. Kanellakis and D. Goldin. Constraint Query Algebras. *Constraints*, 1(1–2):45–83, 1996.
13. P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint Query Languages. In *Proc. of the ACM SIGACT-SIGMOD-SIGART PODS*, pages 299–313. ACM Press, 1990.
14. G. M. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 2000.
15. M. Liu. Deductive Database Languages: Problems and Solutions. *ACM Computing Surveys*, 31(1):27–62, 1999.
16. 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.
17. A. Poulovassilis. The Implementation of FDL, a Functional Database Language. *The Computer Journal*, 35(2):119–128, 1992.
18. R. Ramakrishnan and J. Ullman. A Survey of Deductive Database Systems. *JLP*, 23:126–149, 1995.
19. P. Z. Revesz. Safe Datalog Queries with Linear Constraints. In *Proc. of CP*, LNCS 1520, pages 355–369. Springer, 1998.
20. P. Z. Revesz. Safe Query Languages for Constraint Databases. *TODS*, 23(1):58–99, 1998.
21. J. D. Ullman. Bottom-up Beats Top-down for Datalog. In *Proc. of PODS*, pages 140–149. ACM Press, 1989.