A Relational Algebra 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 study the integration of functional logic programming and databases by presenting a data model, and a query and data definition language. The query and data definition language is based on the use of a set of algebra operators over an extended relational algebra. The algebra expressions built from the extended algebra are used for expressing queries and rules. In addition, algebra expressions can be used for defining functions, typical in a functional logic program.

1 Introduction

Database technology is involved in most software applications. For this reason, *functional logic languages* [9] should include database features in order to cover with 'real world' applications and increase its application field.

Relational algebra [7] is a formalism for defining and querying *relational databases* [6]. Relational algebra is based on the use of the operators, such as *selection*, *projection*, *cross product*, *join*, *set union* and *set difference*. Database *views* and *queries* can be defined, in an intuitive way, by using these operators.

In order to integrate functional logic programming and databases, our idea is: (1) to adapt *functional logic programming to databases*, by considering a suitable *data model*; (2) to adapt *the algebra operators* in order to handle the proposed *data model*; (3) to propose a *data definition language* which consists on *expressing conditional rewriting rules by means of the algebra operators* and consider a *query language* based on algebra operators; and finally, (4) to provide *semantic foundations* to this integration.

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) corresponds to each set of key attributes. In addition, we have adopted *non-deterministic semantics* from functional-logic programming, recently investigated in the framework CRWL [8]. Under non-deterministic semantics, values can be *grouped into sets*, representing the set of values of the output of a non-deterministic function. Therefore, our 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. In addition, functional logic programming handles *partial and non-strict* functions, and *possibly infinite*

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

data. Therefore, in our setting, an attribute can be partially defined or, even, include possibly infinite information.

With respect to (2), and in order to handle recursively defined datatypes, we *generalize the algebra operators* in such a way that, for instance, we can project the elements of a list. In addition, goals and rule conditions in functional logic programming consist on *equality constraints*, which have to be solved in order to solve a goal, or apply a rule. Therefore, we will generalize *selection* and *projection* operators in a double sense: (a) in order to restructure complex values, by applying data *constructors* and *destructors* over the attributes of a given relation, and (b) in order to select tuples whose *totally defined values* satisfy *equality constraints*.

With respect to (3), we propose a *data definition language* which basically consists on *database schema definitions*, *database instance definitions* and *(lazy) function definitions*. For a given database schema, we can define an instance (which consists on *key* and *non-key* attribute values) by means of rules in the form of *algebra expressions*, called *algebraic rules*. In addition, a set of functions can be defined by means of algebraic rules in order to be used by queries for handling recursively defined datatypes. The *query language* will be also based on algebra expressions.

Finally, and w.r.t. (4), we will study the *semantics* of the algebra operators, and conditions in order to ensure that they have the same *expressivity power* as equality constraints. In addition, we have studied a *fixed point operator*, which allows computing the *least instance induced* from a set of algebraic rules. Due to lack of space, these results will be omitted here, but they will be included in the full version of this paper.

Extended relational algebras have been studied as alternative query languages for *deductive databases* [1], and *constraint databases*[10, 5]. Our extended relational algebra 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 algebra in order to handle *complex values built from arbitrary recursively defined datatypes*. In addition, our algebra is similar to the algebras for constraint databases in the sense of dealing with *(equality) constraints*, but here for comparing sets of complex values.

Finally, remark that this works goes towards the design of a functional logic deductive language whose operational semantics [2, 4], and an extended relation calculus [3] have also been studied.

The organization of this paper is as follows. In section 2 we will present a short introduction to the syntax of algebraic rules; section 3 will show the data model; section 4 will define the extended relational algebra, and finally, section 5 will show the query and data definition language.

2 Preliminaries

Logic programs and *goals* can be translated into algebra expressions following three basic ideas: *logic variables* are translated into projections, *bounded arguments* into selections and *shared variables* into joins. For instance:

(1)	$\mathtt{p(a,a)}.$	**(2)**	$\mathtt{p(b,c)}.$
(3)	$\mathtt{q(b,a)}.$	**(4)**	$\mathtt{r(X,Y)} : -\mathtt{p(X,c)}, \mathtt{q(X,Y)}.$
	can be translated into		
$(\mathbf{1})'$	$\mathtt{p} = (\mathtt{a},\mathtt{a}).$	$(\mathbf{2})'$	$\mathtt{p} = (\mathtt{b},\mathtt{c}).$
$(\mathbf{3})'$	$\mathtt{q} = (\mathtt{b},\mathtt{a}).$	$(\mathbf{4})'$	$\mathtt{r} = \pi_{\mathtt{p.1,q.2}}(\sigma_{\mathtt{p.2=c}}(\mathtt{p}) \bowtie_{\mathtt{p.1=q.1}} \mathtt{q})$

Facts are translated into tuple definitions (see $(\mathbf{1})'$, $(\mathbf{2})'$ and $(\mathbf{3})'$). With respect to the *rules*, and assuming that attribute names are $p.1, \ldots$ for each p, then in rule **(4)**, $\mathtt{p(X,c)}$ selects the tuples of $\mathtt{p}$ with the second attribute equal to $\mathtt{c}$; that is, $\sigma_{\mathtt{p.2=c}}(\mathtt{p})$. Here "=" is intended as the values unifying with $\mathtt{c}$. In addition, the condition $\mathtt{p(X,c)}, \mathtt{q(X,Y)}$ can be intended as a join operation between $\mathtt{p}$ and $\mathtt{q}$ w.r.t the first attribute of $\mathtt{p}$ and $\mathtt{q}$ (represented by the shared variable $\mathtt{X}$); that is, $\sigma_{\mathtt{p.2=c}}(\mathtt{p}) \bowtie_{\mathtt{p.1=q.1}} \mathtt{q}$. Finally, $\mathtt{r}$ projects the first attribute of $\mathtt{p}$ (or $\mathtt{q}$) and the second attribute of $\mathtt{q}$, that is, $\pi_{\mathtt{p.1,q.2}}(\sigma_{\mathtt{p.2=c}}(\mathtt{p}) \bowtie_{\mathtt{p.1=q.1}} \mathtt{q})$. Now, a goal $: -\mathtt{r(b,X)}$ can be translated into $\pi_{\mathtt{r.2}}(\sigma_{\mathtt{r.1=b}}(\mathtt{r}))$, obtaining as answer the tuple $(\mathtt{a})$. Now, we can consider the following (recursive) program, which succeeds for each natural number built from $\mathtt{0}$ and $\mathtt{s}$:

(1)	$\mathtt{p(0)}.$	*can be translated into*	$(\mathbf{1})'$	$\mathtt{p} = (\mathtt{0}).$
(2)	$\mathtt{p(s(X))} : -\mathtt{p(X)}.$		$(\mathbf{2})'$	$\mathtt{p} = \pi_{\mathtt{s(p.1)}}(\mathtt{p}).$

In order to write the original program as an algebra expression, we need to consider projection operators over *complex values*, such as the above translation shows. The meaning of $\pi_{\mathtt{s(p.1)}}(\mathtt{p})$ is to project the "successor" of the first attribute of $\mathtt{p}$. This projection operation is even recursive and defines the same relation as the corresponding logic program. Finally, the projection operation also needs to accomplish the destruction of terms. For instance:

(1)	$\mathtt{q(s(0))}.$	*can be translated into*	**(1)**	$\mathtt{q} = (\mathtt{s(0)}).$
(2)	$\mathtt{p(X)} : -\mathtt{q(s(X))}.$		**(2)**	$\mathtt{p} = \pi_{\mathtt{s.1(q.1)}}(\mathtt{q})$

where $\pi_{\mathtt{s.1(q.1)}}(\mathtt{q})$ projects the destruction of the first attribute of $\mathtt{q}$ (i.e. $\mathtt{s.1(q.1)}$), by obtaining n whenever the first attribute of $\mathtt{q}$ is $\mathtt{s(n)}$.

In order to generalize this formalism to functional logic programming, we should assume the following convention. A function f defines a $n+1$-tuple, whenever the arity of f is n (i.e. $ar(f) = n$). For instance:

(1) $\mathtt{append([\,],L)} := \mathtt{L}.$
(2) $\mathtt{append([X|L],L1)} := \mathtt{[X|append(L,L1)]}.$
(3) $\mathtt{member(X,L)} := \mathtt{true} \Leftarrow \mathtt{append(L1,[X|L2])} == \mathtt{L}.$
can be translated into
$(\mathbf{1})'$ $\mathtt{append} := (\mathtt{[\,]},\mathtt{L},\mathtt{L}).$
$(\mathbf{2})'$ $\mathtt{append} := \pi_{\mathtt{[X|append.1]},\mathtt{append.2},\mathtt{[X|append.3]}}(\mathtt{append}).$
$(\mathbf{3})'$ $\mathtt{member} := \pi_{\mathtt{[]}.1(\mathtt{append}.2),\mathtt{append}.3,\mathtt{true}}(\pi_{\mathtt{append}.1,\mathtt{[]}.1(\mathtt{append}.2),\mathtt{[]}.2(\mathtt{append}.2),\mathtt{append}.3}(\mathtt{append} == \mathtt{append}) == \mathtt{append}).$

As can be seen, we consider the functions `append` and `member` as relations, including *non-ground tuples*; that is, tuples with variables where: (a) these ones are universally quantified (and thus representing an *infinite relation*), and (b) they can be instantiated. For instance, the tuples $([\,],[\mathtt{0}],[\mathtt{0}])$ and $([\mathtt{1}],[\mathtt{0}],[\mathtt{1},\mathtt{0}])$ are (instantiated) tuples of the relation `append`.

Finally, functions can be either *non-strict* or represent *possibly infinite values*, where $\perp$ is used for representing *non-strictness* and *partial approximations* of infinite values. For instance:

(1)	$\mathtt{f(X)} := \mathtt{0}$.	*can be translated into*	(1)′ $\mathtt{f} := (\mathtt{X}, \mathtt{0})$.
(2)	$\mathtt{g(X)} := \mathtt{s(g(X))}$		(2)′ $\mathtt{g} := \pi_{\mathtt{g.1,s(g.2)}}(\mathtt{g})$

where g represents the tuples $(\mathtt{X}, \{\bot, \mathtt{s}(\bot), \mathtt{s}(\mathtt{s}(\bot)), \ldots\})$. Finally, the equality constraint $\mathtt{f}(\mathtt{g}(\mathtt{X})) == \mathtt{0}$, which can be written as $\pi_{\mathtt{g.1}}(\mathtt{f} \bowtie_{\mathtt{f.1=g.2,f.2==0}} \mathtt{g})$, defines as answer the tuple (X), where X is universally quantified.

3 The Data Model

Let's consider job_bonus as an attribute, represented by a complex value of the form j&b(job_name, bonus). Now, the attribute can include the following kinds of information:

$\bot$	undefined information (ni)	*expressing that both person's job name and salary bonus are unknown*
j&b(lecturer, 100)	totally defined information (tdi)	*expressing that a person's job name is* lecturer *and his(her) salary bonus is* 100 €
j&b(associate, $\bot$)	partially undefined information (pni)	*expressing that we know the job name, but not the salary bonus*

Over these kinds of information, the following *equality relations* can be defined: (1) $=$ (*syntactic equality*); for instance, j&b(associate, $\bot$) = j&b(associate, $\bot$) holds; and (2) $==$ (*strict equality*), expressing that *two values are syntactically equal and totally defined*; for instance, j&b(associate, 250) == j&b (associate, 250) holds, and j&b(associate, $\bot$) == j&b(associate, 250) does not satisfy.

Assuming a set of *data constructors* $c, d, \ldots$ $DC = \cup_{n \geq 0} DC^n$ each one with a given arity, the special symbol $\bot$ with arity 0 (not included in DC), and a set $\mathcal{V}$ of variables $X, Y, \ldots$, we can build the set of *partial c-terms* $CTerm_{DC,\bot}(\mathcal{V})$. In particular, $CTerm_{DC}(\mathcal{V})$ represents the set of *totally defined c-terms*. We can build substitutions $Subst_\bot = \{\theta : \mathcal{V} \to CTerm_{DC,\bot}(\mathcal{V})\}$ in the usual way.

A *partial ordering* $\leq$ on $CTerm_{DC,\bot}(\mathcal{V})$ can be defined as the least one satisfying: $\bot \leq t$, $X \leq X$, and $c(t_1, ..., t_n) \leq c(t'_1, ..., t'_n)$ if $t_i \leq t'_i$ for all $i \in \{1, ..., n\}$ 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 $\bot$ is the *bottom element*. Now, we can build the set of (possibly infinite) cones of c-terms $\mathcal{C}(CTerm_{DC,\bot}(\mathcal{V}))$, and the set of (possibly infinite) ideals of c-terms $\mathcal{I}(CTerm_{DC,\bot}(\mathcal{V}))$. Cones and ideals can also be partially ordered under the *set-inclusion* ordering (i.e $\subseteq$).

Over cones and ideals, we can define an *equality relation*, $\mathcal{C} == \mathcal{C}'$, which holds whenever *any value* in $\mathcal{C}$ and $\mathcal{C}'$ is *strictly equal*. Finally, we need to consider the *greatest subcone* containing the totally defined values of a cone $\mathcal{C}$, denoted by $Total(\mathcal{C})$, and defined as $Total(\mathcal{C}) = \{\bot\} \cup_{t \in \mathcal{C} \cap CTerm_{DC}(\mathcal{V})} < t >$

Definition 1 (Database Schemas). *Assuming a Milner's style polymorphic type system, a* database schema S *is a finite set of* relation schemas $R_1, \ldots, R_p$ *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 *key attributes* and $A_{k+1} \ldots A_n$ represent *non-key attributes*, denoted by the sets $Key(R)$ and $NonKey(R)$, respectively. We denote by $Att(R)$ the attributes of R, $nAtt(R) = n$ and $nKey(R) = k$.

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

Definition 2 (Databases). *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 by $DS(D)$ the set of *defined symbols* of D, which contains relation names, non-key attributes and interpreted function symbols. Interpreted functions can be considered as relations, assuming the following convention. For each $f : T_1 \times \ldots \times T_n \rightarrow T_0 \in IF$, we can include in S a relation schema $f(f.1 : T_1, \ldots, f.n : T_n, f.n + 1 : T_0)$. As an example of database, we can consider the following one:

$$S \left\{ \begin{array}{l} \texttt{person\_job}(\underline{\texttt{name}} : \texttt{people}, \texttt{age} : \texttt{nat}, \texttt{address} : \texttt{dir}, \texttt{job\_id} : \texttt{job}, \texttt{boss} : \texttt{people}) \\ \texttt{job\_information}(\underline{\texttt{job\_name}} : \texttt{job}, \texttt{salary} : \texttt{nat}, \texttt{bonus} : \texttt{nat}) \\ \texttt{person\_boss\_job}(\underline{\texttt{name}} : \texttt{people}, \texttt{boss\_age} : \texttt{cbossage}, \texttt{job\_bonus} : \texttt{cjobbonus}) \\ \texttt{peter\_workers}(\underline{\texttt{name}} : \texttt{people}, \texttt{work} : \texttt{job}) \end{array} \right.$$

$$DC \left\{ \begin{array}{l} \texttt{john} : \texttt{people},\ \texttt{mary} : \texttt{people},\ \texttt{peter} : \texttt{people} \\ \texttt{lecturer} : \texttt{job},\ \texttt{associate} : \texttt{job},\ \texttt{professor} : \texttt{job} \\ \texttt{add} : \texttt{string} \times \texttt{nat} \rightarrow \texttt{dir} \\ \texttt{b\&a} : \texttt{people} \times \texttt{nat} \rightarrow \texttt{cbossage} \\ \texttt{j\&b} : \texttt{job} \times \texttt{nat} \rightarrow \texttt{cjobbonus} \end{array} \right. \qquad IF \left\{ \texttt{retention} : \texttt{nat} \rightarrow \texttt{nat} \right.$$

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 *"views"* `person_boss_job`, and `peter_workers`, which will take key values from the relation `person_job`. The first view includes, for each person, pairs in the form of records constituted by: (a) his/her boss and boss' age, by using the complex c-term $\texttt{b\&a}(\texttt{people}, \texttt{nat})$, and (b) his/her job and job bonus, by using the complex c-term $\texttt{j\&b}(\texttt{job}, \texttt{nat})$. The second view includes workers whose boss is `peter`. In addition, DC includes data constructors for the types `people`, `job`, `dir`, `cbossage` and `cjobbonus`, and IF the interpreted function symbol `retention` which computes the salary without tax. The function `retention` can be considered as a relation of the form $\texttt{retention}(\underline{\texttt{retention.1}} : \texttt{nat}, \texttt{retention.2} : \texttt{nat})$. In addition, we can consider database schemas involving *(possibly) infinite databases* as follows:

$$S \left\{ \begin{array}{l} \texttt{2Dpoint}(\underline{\texttt{coord}} : \texttt{cpoint},\ \texttt{color} : \texttt{nat}) \\ \texttt{2Dline}(\underline{\texttt{origin}} : \texttt{cpoint},\ \underline{\texttt{dir}} : \texttt{orientation}, \texttt{next} : \texttt{cpoint}, \texttt{points} : \texttt{cpoint}, \\ \texttt{list\_of\_points} : \texttt{list}(\texttt{cpoint})) \end{array} \right.$$

$$DC \left\{ \begin{array}{l} \texttt{north} : \texttt{orientation}, \texttt{south} : \texttt{orientation}, \texttt{east} : \texttt{orientation}, \\ \quad \texttt{west} : \texttt{orientation},\ \texttt{northeast} : \texttt{orientation}\ldots \\ [\,] : \texttt{list A},\quad [\,|\,] : \texttt{A} \times \texttt{list A} \rightarrow \texttt{list A} \\ \texttt{p} : \texttt{nat} \times \texttt{nat} \rightarrow \texttt{cpoint} \end{array} \right. \qquad IF \left\{ \texttt{select} : (\texttt{list A}) \rightarrow \texttt{A} \right.$$

wherein the relation schemas `2Dpoint` and `2Dline` are defined for representing bidimensional points and lines, respectively. The attribute `points` stores the *(infinite) set* of points of the line, and `list_of_points` the *(infinite) list* of points of the line. Here, we can see the double use of complex values: as a set (which can be implicitly assumed), and a list.

Definition 3 (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, with $n = nAtt(R)$ and $V_i \in \mathcal{C}(CTerm_{DC,\perp}(\mathcal{V}))$. In particular, each V_j ($j \leq nKey(R)$) satisfies $V_j \in CTerm_{DC}(\mathcal{V})$[2].

The last condition forces the key values to be one-valued and totally defined. Values can be non-ground where variables are implicitly universally quantified.

Definition 4 (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, $\mathcal{DC} = CTerm_{DC,\perp}(\mathcal{V})$, and $\mathcal{IF}$ is a set of function interpretations $f^{\mathcal{D}}, g^{\mathcal{D}}, \ldots$ satisfying $f^{\mathcal{D}} : CTerm_{DC,\perp}(\mathcal{V})^n \rightarrow \mathcal{C}(CTerm_{DC,\perp,F}(\mathcal{V}))$ is monotone for each $f \in IF^n$, that is, $f^D\ t_1, \ldots, t_n \subseteq f^D\ t'_1, \ldots, t'_n$ if $t_i \leq t'_i$, with $1 \leq i \leq n$.*

If we consider interpreted functions as relations, $\mathcal{IF}$ defines a set of tuples for each $f^{\mathcal{D}} \in \mathcal{IF}$ as follows: $(t_1, \ldots, t_n, V) \in f$ if $V = \{t \mid f^{\mathcal{D}} t_1, \ldots, t_n = t,\ and\ t_i \in CTerm_{DC}(\mathcal{V})\}$[3]. Next, we show an example of instance for the example of people and jobs:

`person_job`	$\left\{\begin{array}{l}(\texttt{john}, \{\perp\}, \{\texttt{add}('\texttt{6th Avenue}', 5)\}, \{\texttt{lecturer}\}, \{\texttt{mary}, \texttt{peter}\}) \\ (\texttt{mary}, \{\perp\}, \{\texttt{add}('\texttt{7th Avenue}', 2)\}, \{\texttt{associate}\}, \{\texttt{peter}\}) \\ (\texttt{peter}, \{\perp\}, \{\texttt{add}('\texttt{5th Avenue}', 5)\}, \{\texttt{professor}\}, \{\perp\})\end{array}\right.$
`job_information`	$\left\{\begin{array}{l}(\texttt{lecturer}, \{1200\}, \{\perp\}) \\ (\texttt{associate}, \{2000\}, \{\perp\}) \\ (\texttt{professor}, \{3200\}, \{1500\})\end{array}\right.$
`person_boss_job`	$\left\{\begin{array}{l}(\texttt{john}, \{\texttt{b\&a}(\texttt{mary}, \perp), \texttt{b\&a}(\texttt{peter}, \perp)\}, \{\texttt{j\&b}(\texttt{lecturer}, \perp)\}) \\ (\texttt{mary}, \{\texttt{b\&a}(\texttt{peter}, \perp)\}, \{\texttt{j\&b}(\texttt{associate}, \perp)\}) \\ (\texttt{peter}, \{\texttt{b\&a}(\perp, \perp)\}, \{\texttt{j\&b}(\texttt{professor}, 1500)\})\end{array}\right.$
`peter_workers`	$\left\{\begin{array}{l}(\texttt{john}, \{\texttt{lecturer}\}) \\ (\texttt{mary}, \{\texttt{associate}\})\end{array}\right.$
`retention`	$\{(0, \{0\}) \ldots..(10, \{8\}), \ldots$

As can be seen, each tuple instance includes key and non-key attribute values grouped by sets[4]. Firstly, in the instance of the relation `person_job`, all tuples include the value $\perp$ for the attribute `age`, indicating *undefined information* (*ni*). Secondly, the instance of the view `person_boss_job` includes *partially undefined (pni)*, `b&a(mary`, $\perp$), expressing that `john`'s boss is known (i.e. `mary`), but `mary`'s age is undefined. With respect to the modeling of a (possibly) infinite database, we can consider the following approximation to the instance for the relation schema `2Dline` with (*possibly infinite*) values in their defined attributes:

`2Dline`	$\left\{\begin{array}{l}(\texttt{p}(0,0), \texttt{north}, \{\texttt{p}(0,1)\}, \{\texttt{p}(0,1), \texttt{p}(0,2), \perp\}, \{[\texttt{p}(0,0), \texttt{p}(0,1), \texttt{p}(0,2)\vert\perp]\}), \ \ldots \\ (\texttt{p}(1,1), \texttt{east}, \{\texttt{p}(2,1)\}, \{\texttt{p}(2,1), \texttt{p}(3,1), \perp\}, \{[\texttt{p}(1,1), \texttt{p}(2,1), \texttt{p}(3,1)\vert\perp]\}), \ \ldots\end{array}\right.$

4 Extended Relational Algebra

The projection operator is based on the so-called *projection c-terms*, wherein a projection c-term is a *(labeled) algebra c-term*. Now, we will define both concepts, although, firstly, we need to define the notion of *data destructors*.

[2] Each key value V_j can be considered as an ideal by means of the mapping $V_j \rightarrow < V_j >$.

[3] Functions can define partially defined values, but we can lift to totally defined values, by using variables instead of $\perp$ without losing generality.

[4] These sets can be ideals or cones but, here, in the examples we will summarize their contents with the maximal elements.

Definition 5 (Data Destructors). *Given a set of data constructors DC, we define the set of* data destructors *DD induced from DC as the set $c.idx : T_0 \to T_{idx}$, whenever $c : T_1 \times \ldots \times T_n \to T_0 \in DC$ and $1 \leq idx \leq n$. The semantics of each $c.idx$ is defined as follows: $c.idx(c(t_1, \ldots, t_n)) =_{def} t_{idx}$, and $\perp$ otherwise.*

Now, the set of *algebra c-terms* is defined as the set of c-terms built from DC, DD, attributes of D, and variables of $\mathcal{V}$. The *projection operator* projects two kinds of elements: (a) algebra c-terms, and (b) totally defined algebra c-terms.

Definition 6 (Projection C-Terms). *A* projection c-term *$p, q, \ldots$ has the form t_A and $\overset{==}{t_A}$, where t_A is an algebra c-term.*

$\overset{==}{t_A}$ is a labeled algebra c-term, representing the *set of totally defined c-terms represented by t_A*. A projection c-term denotes a cone or ideal w.r.t. a tuple, depending on whether it combines one-valued or multi-valued attributes.

Definition 7 (Denotation of Projection C-Terms). *The* denotation *of a projection c-term p in a tuple $V = (V_1, \ldots, V_n) \in \mathcal{R}$ $(R \in S)$, w.r.t. a database $D = (S, DC, IF)$ and instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$, denoted by $[\![p]\!]^{\mathcal{D}}_V$, is defined as follows:*
$[\![X]\!]^{\mathcal{D}}_V =_{def} < X >$, *if* $X \in \mathcal{V}$; $[\![A_i]\!]^{\mathcal{D}}_V =_{def} \cup_{\psi \in Subst_\perp} V_i\psi$, *for all* $A_i \in Key(R) \cup NonKey(R)$, *and* $< \perp >$, *otherwise;* $[\![c(t_1^A, \ \ldots \ , t_n^A)]\!]^{\mathcal{D}}_V =_{def} < c([\![t_1^A]\!]^{\mathcal{D}}_V, \ \ldots \ , [\![t_n^A]\!]^{\mathcal{D}}_V) >$[5], *for all* $c \in DC^n$; $[\![c.idx(t_A)]\!]^{\mathcal{D}}_V =_{def} c.idx([\![t_A]\!]^{\mathcal{D}}_V)$, *for all* $c.idx \in DD$; *and* $[\![\overset{==}{t_A}]\!]^{\mathcal{D}}_V =_{def} Total([\![t_A]\!]^{\mathcal{D}}_V)$.

We denote by $Att(p)$ (resp. $var(p)$) the attribute names (resp. variables) occurring in a projection c-term p. For instance $\pi_{\overset{==}{\texttt{name,b\&a(boss,age)}}}(\texttt{person\_job})$ contains the tuples $(\texttt{john}, \{\perp\})$, $(\texttt{mary}, \{\perp\})$ and $(\texttt{peter}, \{\perp\})$, given that $\texttt{b\&a}(\texttt{boss}, \texttt{age})$ represents a partially defined value in the original relation `person_job`.

With respect to *selection operator*, we need to define when a tuple satisfies two kinds of *equality relations*: (1) *syntactic equality* (i.e. $=$) and (2) *strict equality* (i.e. $==$).

Definition 8 (Selection Formulas). *A* selection formula *F has the form $A = t_A$, where $A \in Key(R)$, $R \in S$, and t_A is an algebra c-term, and $t_A == t'_A$, where t_A and t'_A are algebra c-terms*

Definition 9 (Satisfiability). *A tuple $V \in \mathcal{R}$ $(\mathcal{R} \in \mathcal{S})$* satisfies a selection formula *F w.r.t. a database $D = (S, DC, IF)$ and instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ if $V \models_A F$, where $\models_A$ is defined as follows: $V \models_A A = t_A$, if $[\![t_A]\!]^{\mathcal{D}}_V \in [\![A]\!]^{\mathcal{D}}_V$; and $V \models_A t_A == t'_A$, if $[\![t_A]\!]^{\mathcal{D}}_V == [\![t'_A]\!]^{\mathcal{D}}_V$*

For instance, let's consider the tuple $(\texttt{john}, \{\perp\}, \{\texttt{add}('\texttt{6th Avenue}', 5)\}, \{\texttt{lecturer}\}, \{\texttt{mary}, \texttt{peter}\})$ in the instance of the relation schema `person_job`. Now, the selection formulas $\texttt{name} = \texttt{john}$, $\texttt{add.2}(\texttt{address}) == 5$, and $\texttt{boss} == \texttt{mary}$ are satisfied, but $\texttt{age} == 35$, and $\texttt{job\_id} == \texttt{associate}$ are not.

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

Definition 10 (Algebra Operators). *Let $D = (S, DC, IF)$ and $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ be a database and an instance, and let R, Q be relation names of S (or IF), then the* algebra operators *are defined as follows:*

Selection (σ):
$\mathcal{R}' = \sigma_{F_1,\ldots,F_n}(R) =_{def} \{V \in \mathcal{R} \mid V \models_A F_1, \ldots, F_n\}$ *where* $Key(R') = Key(R)$ *and* $NonKey(R') = NonKey(R)$.

Projection (π):
$\mathcal{R}' = \pi_{t_1^A,\ldots,t_k^A,\overset{==}{t_{k+1}^A},\ldots,\overset{==}{t_n^A}}(R) =_{def} \{(\llbracket t_1^A \rrbracket_V^{\mathcal{D}}, \ldots, \llbracket t_k^A \rrbracket_V^{\mathcal{D}}, \llbracket \overset{==}{t_{k+1}^A} \rrbracket_V^{\mathcal{D}}, \ldots, \llbracket \overset{==}{t_n^A} \rrbracket_V^{\mathcal{D}}) | V \in \mathcal{R}\}$
where $t_1^A, \ldots, t_k^A$ *are all (possibly* destructed*) key attributes of* R *and* $t_{k+1}^A, \ldots, t_n^A$ *are algebra c-terms, and* $Key(R') = \{t_1^A, \ldots, t_k^A\}$ *and* $NonKey(R') = \{\overset{==}{t_{k+1}^A}, \ldots, \overset{==}{t_n^A}\}$.

Cross Product ($\times$):
$\mathcal{P} = R \times Q =_{def} \{(V_1, \ldots, V_k, W_1, \ldots, W_{k'}, V_{k+1}, \ldots, V_n, W_{k'+1}, \ldots, W_m) \mid (V_1, \ldots, V_k, V_{k+1}, \ldots, V_n) \in \mathcal{R}, (W_1, \ldots, W_{k'}, W_{k'+1}, \ldots, W_m) \in \mathcal{Q}\}$
where $k = nKey(R)$, $k' = nKey(Q)$, $n = nAtt(R)$ *and* $m = nAtt(Q)$, $Key(P) = Key(R) \cup Key(Q)$, *and* $NonKey(P) = NonKey(R) \cup NonKey(Q)$.

Join ($\bowtie$):
$R \bowtie_{F_1,\ldots,F_n} Q =_{def} \sigma_{F_1,\ldots,F_n}(R \times Q)$ *with the same conditions as the selection operator.*

Renaming (δ_ρ):
$\mathcal{R}' = \delta_\rho(R)$ *where* $\rho : A_1 A_2 \ldots A_n \to B_1 B_2 \ldots B_n$, $n = nAtt(R)$, *and* $B_i \neq B_j$ *if* $i \neq j$. $\mathcal{R}'$ *contains the same tuples as* $\mathcal{R}$ *and its schema is* $R'(\rho(A_1), \ldots, \rho(A_k), \rho(A_{k+1}), \ldots, \rho(A_n))$, *whenever the schema of* R *is* $R(\underline{A_1}, \ldots, \underline{A_k}, A_{k+1}, \ldots, A_n)$.

Definition 11 (Algebra Expressions). *Given a database* $D = (S, DC, IF)$ *and instance* $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$, *then the* algebra expressions $\Psi(D)^{\mathcal{D}}$ *are defined as expressions built from a composition of algebra operators over a subset of relation names* $\{R_1, \ldots, R_p\}$ *of* S *(and a subset* $\{f_1, \ldots, f_r\}$ *of* IF*). In addition, the following conditions must be satisfied:*

(a) Ψ *must be closed w.r.t. key values; that is,* $Key(\Psi) = \cup_{R \in Rel(\Psi)} Key(R)$

(b) Ψ *must be closed w.r.t. data destructors; that is,*

- *(b.1) data destructors* $c.idx(t_A)$ *occurs in the form:* $\pi_{c.idx(t_A)}$, $\pi_{c.idx(t_A)=s_A}$, $\pi_{\overset{==}{c.idx(t_A)}}$, *and* $\pi_{c.idx(t_A)==s_A}$;
- *(b.2) whenever* $\pi_{c.idx(t_A)}$ *(or* $\sigma_{s_{idx}^A = c.idx(t_A)}$*) occurs in* Ψ, *then* $\pi_{c.i(t_A)}$ *or* $\sigma_{s_i^A = c.i(t_A)}$ *must occur in* Ψ *for each* $1 \leq i \leq n$, *where* $c \in DC^n$; *and*
- *(b.3) whenever* $\pi_{\overset{==}{c.idx(t_A)}}$ *(or* $\sigma_{s_{idx}^A == c.idx(t_A)}$*) occurs in* Ψ, *then* $\pi_{\overset{==}{c.i(t_A)}}$ *or* $\sigma_{s_i^A == c.i(t_A)}$ *must occur in* Ψ *for each* $1 \leq i \leq n$, *where* $c \in DC^n$

where $Key(\Psi)$ *(resp.* $Rel(\Psi)$*) are the key attribute (resp. relation) names occurring in* Ψ.

Algebra expressions are used for expressing *queries* and *algebraic rules*. For instance, the table 1 shows several algebra expressions representing queries against the instances of `person_job` and `person_boss_job`.

Table 1. Examples of Algebra Expressions

Query	Algebra Expression
To obtain `lecturer` *people and their boss's address whenever it is defined*	$\pi_{\substack{== \\ \texttt{name,address}'}}(\sigma_{\texttt{name}'=\texttt{boss, job\_id} == \texttt{lecturer}}(\delta_{(\rho_1)}(\texttt{person\_job} \times \texttt{person\_job})))$
To obtain `associate`*-people's name, and salary bonus whenever it is defined*	$\pi_{\substack{== \\ \texttt{name,j\&b.2(job\_bonus)}}}(\sigma_{\texttt{j\&b.1(job\_bonus)}==\texttt{associate}}(\texttt{person\_boss\_job}))$
	$\rho_1 \equiv \{\texttt{name age} \ldots \texttt{name age} \ldots \rightarrow \texttt{name age} \ldots \texttt{name}' \texttt{ age}' \ldots\}$

5 The Query and Data Definition Language

Definition 12 (Queries and Query Answers). *A* query *is an algebra expression $\Psi(D)^{\mathcal{D}}$, and $(V_1, \ldots, V_n)$ is an* answer *of $\Psi(D)^{\mathcal{D}}$ if $(V_1, \ldots, V_n) \in \Psi(D)^{\mathcal{D}}$ and $V_i \neq < \perp >$ for all V_i.*

Instances (key and non-key attribute values, and interpreted functions) are defined by means of *algebraic rules*.

Definition 13 (Algebraic Rules). *An* algebraic rule *AR w.r.t. a database D, for a symbol $H \in DS(D)$ of arity n, has the form $H := \pi_{p_1,\ldots,p_n,p}(\Psi(D))$, indicating that H* represents the tuple $(p_1, \ldots, p_n, p)$ obtained from $\Psi(D)$. *In this rule, $\Psi(D)$ is an algebra expression, and $(p_1, \ldots, p_n, p)$ can be any kind of projection c-terms; in addition, $(p_1, \ldots, p_n)$ is a* linear tuple *of projection c-terms (each variable and attribute symbol occurs only once), and* extra variables and attributes *are not allowed, i.e. $var(p) \subseteq \cup_{1 \leq i \leq n} var(p_i)$ and $(Att(p) \cap Att(\Psi(D))) \setminus \cup_{1 \leq i \leq n} Att(p_i) = \emptyset$.*

Projection operator has been defined for the case $\pi_{\substack{== \\ t_1^A,\ldots t_k^A, t_{k+1}^A, \ldots t_n^A}}$, where each t_i^A is (destructed) key attributes. However in algebraic rules, $p_1, \ldots, p_n, p$ can be any kind of projection c-terms, given they can build new tuples. Whenever the tuple is explicitly declared, we can write rules of the form $H := (p_1, \ldots, p_n, p)$ where p_i are c-terms. In particular, assuming R of arity k, the rules $R := \pi_{p_1,\ldots,p_k}(\Psi(D))$ $(R := (p_1, \ldots, p_k))$ allow the setting of $p_1, \ldots, p_k$ as key values of the relation R. Analogously, assuming A of arity $k+1$, the rules $A := \pi_{p_1,\ldots,p_k,p}(\Psi(D))$ (resp. $A := (p_1, \ldots, p_k, p)$), where $A \in NonKey(R)$, set p as value of A for the tuple of R with key values $p_1, \ldots, p_k$. Finally, interpreted function symbols are represented by rules $f := \pi_{p_1,\ldots,p_n,p}(\Psi(D))$, whenever $f \in IF$ is of arity n, where $p_1, \ldots, p_n$ are the arguments of the function, and p is the result. For instance, the instances for the relations `person_job`, `job_information`, and views `person_boss_job` and `peter_workers`, can be defined by the following rules:

$$\texttt{person\_job} \begin{cases} \texttt{person\_job} := (\texttt{john}). & \texttt{person\_job} := (\texttt{mary}). \\ \texttt{person\_job} := (\texttt{peter}). & \\ \texttt{address} := (\texttt{john}, \texttt{add}('\texttt{6th Avenue}', 5)). & \texttt{address} := (\texttt{mary}, \texttt{add}('\texttt{7th Avenue}', 2)). \\ \texttt{address} := (\texttt{peter}, \texttt{add}('\texttt{5th Avenue}', 5)). & \\ \texttt{job\_id} := (\texttt{john}, \texttt{lecturer}). & \texttt{job\_id} := (\texttt{mary}, \texttt{associate}). \\ \texttt{job\_id} := (\texttt{peter}, \texttt{professor}). & \\ \texttt{boss} := (\texttt{john}, \texttt{mary}). & \texttt{boss} := (\texttt{john}, \texttt{peter}). \\ \texttt{boss} := (\texttt{mary}, \texttt{peter}). & \end{cases}$$

job_information	$\left\{\begin{array}{l}\texttt{job\_information} := (\texttt{lecturer}). \qquad \texttt{job\_information} := (\texttt{associate}).\\ \texttt{job\_information} := (\texttt{professor}).\\ \texttt{salary} := \pi_{\texttt{lecturer,retention.2}}(\sigma_{\texttt{retention.1}=1500}(\texttt{retention})).\\ \texttt{salary} := \pi_{\texttt{associate,retention.2}}(\sigma_{\texttt{retention.1}=2500}(\texttt{retention})).\\ \texttt{salary} := \pi_{\texttt{professor,retention.2}}(\sigma_{\texttt{retention.1}=4000}(\texttt{retention})).\\ \texttt{bonus} := (\texttt{professor}, 1500).\end{array}\right.$
person_boss_job	$\left\{\begin{array}{l}\texttt{person\_boss\_job} := \pi_{\texttt{name}}(\texttt{person\_job}).\\ \texttt{boss\_age} := \pi_{\texttt{name,b\&a(boss,age')}}(\sigma_{\texttt{boss}=\texttt{name}'}(\delta_{\texttt{name}\ldots\texttt{name}\ldots\rightarrow\texttt{name}\ldots\texttt{name}'\ldots}(\texttt{person\_job}\\ \times\texttt{person\_job}))).\\ \texttt{job\_bonus} := \pi_{\texttt{name,j\&b(job\_name,bonus)}}(\sigma_{\texttt{job\_id}=\texttt{job\_name}}(\texttt{person\_job}\\ \times\texttt{job\_information})).\end{array}\right.$
peter_workers	$\left\{\begin{array}{l}\texttt{peter\_workers} := \pi_{\texttt{name}}(\sigma_{\texttt{boss}==\texttt{peter}}(\texttt{person\_job})).\\ \texttt{work} := \pi_{\texttt{name,job\_id}}(\texttt{person\_job}).\end{array}\right.$

6 Conclusions and Future Work

We have shown how to integrate functional logic programming and databases by means of a unified syntax and semantics. However, this framework opens new research topics, such as to provide a suitable operational mechanism (and the study of evaluation strategies) in order to efficiently solve queries against a database. On the other hand, we would like to study the extension of our framework for the handling of negative information in the line of [11].

References

1. S. Abiteboul and C. Beeri. The Power of Languages for the Manipulation of Complex Values. *VLDB*, 4(4):727–794, 1995.
2. 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.
3. J. M. Almendros-Jiménez and A. Becerra-Terón. A Safe Relational Calculus for Functional Logic Deductive Databases. In *Proc. of Workshop on Functional and (Constraint) Logic Programming*, 2003.
4. 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.
5. A. Belussi, E. Bertino, and B. Catania. An Extended Algebra for Constraint Databases. *TKDE*, 10(5):686–705, 1998.
6. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM, CACM*, 13(6):377–387, 1970.
7. E. F. Codd. Relational Completeness of Data Base Sublanguages. In *R. Rustin (ed.), Database Systems*, pages 65–98. Prentice Hall, 1972.
8. 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.
9. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *JLP*, 19,20:583–628, 1994.
10. P. Kanellakis and D. Goldin. Constraint Query Algebras. *Constraints*, 1(1–2):45–83, 1996.
11. 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.