Database Query Languages and Functional Logic Programming

Jesús M. Almendros-Jiménez and Antonio Becerra-Terón

Dpto. de Lenguajes y Computación. Universidad de Almería. Carretera de Sacramento s/n. 04120-Almería. Spain

jalmen@ual.es,abecerra@ual.es

Abstract Functional logic programming is a paradigm which integrates functional and logic programming. It is based on the use of rewriting rules for defining programs, and rewriting for goal solving. In this context, goals, usually, consist of equality (and, sometimes, inequality) constraints, which are solved in order to obtain answers, represented by means of substitutions. On the other hand, database programming languages involve a data model, a data definition language and, finally, a query language against the data defined according to the data model. To use functional logic programming as a database programming language, (1) we will propose a data model involving the main features adopted from functional logic programming (for instance, handling of partial and infinite data), (2) we will use conditional rewriting rules as data definition language, and finally, (3) we will deal with equality and inequality constraints as query language. Moreover, as most database systems, (4) we will propose an extended relational calculus and algebra, which can be used as alternative query languages in this framework. Finally, (5) we will prove that three alternative query languages are equivalent.

Keywords Logic Programming, Functional-Logic Programming, Deductive Databases.

§1 Introduction

Functional logic programming is a paradigm which integrates *functional* and *logic programming*, widely investigated during the last years. In fact, many languages, such as *CURRY* [14], *BABEL* [25], and *TOY* [21], among others, have been developed around this research area [13].

Functional logic programming is based on the use of *rewriting rules* for programs and *rewriting* for goal solving. Goals, usually, consist of equations (and sometimes inequations) which are solved in order to obtain answers represented by means of substitutions.

On the other hand, it is known that *database technology* is involved in most software applications. For this reason, *programming languages* should include database features in order to cover with 'real world' applications. Therefore, the integration of database technology into functional logic programming may be interesting, in order to increase its application field.

In this sense, we should consider that *database programming languages* consist of a *data model*, a *data definition language*, and a *query language* against the data defined according to the data model.

Relational calculus and *algebra* are traditional formalisms for querying *relational databases* [11]. In fact, they are the basis of a high-level database query languages like SQL, and the simplicity of these formalisms is one of the keys for the wide adoption from database technology.

On the one hand, relational calculus is based on the use of a fragment of the *first-order logic*. In the relational calculus, logic formulas contain *logic predicates*, representing *relations*, and they use *equality* relations, which allow us to compare *attribute values*. In the logic formulas, *free variables* play the same role as *search variables*. The simplest relational calculus handles *conjunctions*, does not support *negation*, and formulas are *existentially quantified*. Moreover, it allows the handling of tuples belonging to the *cross product* and *join* of two or more input relations. However, *disjunctions*, *universal quantifications* and *negation* can be included in order to handle other relations, such as the *union* of two relations, the *complement* of a relation (i.e. tuples which do not belong to a relation), and the *difference* of two relations (i.e. tuples which belong to a relation but not to another one).

On the other hand, relational algebra is based on the use of algebra operators, such as *selection*, *projection*, *cross product*, *join*, *set union* and *set difference*. The *selection* operator selects those tuples which satisfy a given condition. The *projection* operator projects some attribute values from a given set of tuples. The *cross product* operator combines two or more sets of tuples. The *join* operator is a combination of selection operator together with the cross product. Finally, relational algebra incorporates two operators from set theory, such as *set union* and *set difference*, which represent the union and the difference of two sets of tuples, respectively.

1.1 Contributions of the Paper

In order to integrate functional logic programming and databases, we propose: (1) to adapt functional logic programs to databases, by considering a suitable *data model* and a *data definition language*; (2) to *propose different query formalisms which handle the proposed data model*; concretely, a *functional logic query language, an extended relational calculus and an extended relational algebra*; and finally, (3) to provide *semantic foundations to the different query languages.*

With respect to (1), the underlying data model of functional logic programming is *complex* from a database point of view [1, 9, 15, 36)] in a double sense. Firstly, types can be defined by using *recursively defined datatypes*, such as *lists* and *trees*. This means the attributes can be multi-valued (i.e. more than one value for a given attribute corresponds to each set of key attributes), storing complex values built from these datatypes. Secondly, we have adopted a *non-deterministic semantics* from functional logic programming, investigated in the framework *CRWL* [12)]. Under this non-deterministic semantics, values can be *grouped into sets*, representing the output of a non-deterministic function. For instance,

```
(1) edge a := b.
(2) edge a := c.
(3) edge b := c.
```

define a non-deterministic function, named `edge`, in order to represent a graph with three nodes (i.e. `a`, `b` and `c`), and three edges, that is one edge from `a` to `b` (rule `(1)`), one from `a` to `c` (rule `(2)`) and, finally, one from `b` to `c` (rule `(3)`). Here, the values defined by function `edge` for node `a` (rules `(1)` and `(2)`) are represented by the set $\{\texttt{b}, \texttt{c}\}$, and this set can be handled by means of a recursive function, called `path`, which includes the conditions for computing the paths occurring in a graph. The function `path` is defined as follows:

```
(4) path X := edge X.
(5) path X := path (edge X).
```

Therefore, in our case, the fact of adopting the framework *CRWL* assumes that the attributes can be also multi-valued, in the sense of storing complex values grouped into sets.

Moreover, functional logic programming can handle *partial and possibly infinite data*. The undefined value $\perp$ is introduced in *CRWL* in order to give semantics to expressions such as `edge c`, whose set of defined values is $\{\perp\}$. Even more, $\perp$ is used for representing a partial approximation to possibly infinite data; for instance, we could define the function `listpath` as follows:

```
(6) listpath X := [X | listpath (edge X)].
```

wherein $[\mathtt{a}, \mathtt{b}, \mathtt{c}, \perp]$ and $[\mathtt{a}, \mathtt{c}, \perp]$ represent two partial approximations to the two paths defined from node `a` enclosed in a list. With respect to our setting, an attribute can be partially defined or, even, include possibly infinite information. The first case can be interpreted as follows: the database can include undefined information and partially undefined information [19] (i.e. absence or unknown information, and information that is partially known, respectively); the second one indicates that the database can store *infinite information*, allowing infinite database instances [8] (i.e. *infinite attribute values* and *infinite sets of tuples*). In our case, the infinite information will be also handled by means of *partial approximations*.

Furthermore, we have adopted the handling of *negation* from functional logic programming, studied in the framework *CRWLF* [22]. This framework incorporates the notion of "failure of reduction", in such a way that, in the framework *CRWLF* expressions like `edge c` define the set of values $\{\mathtt{F}\}$ instead of $\{\perp\}$. $\{\mathtt{F}\}$ represents, for instance, that the expression `edge c` fails when it is reduced, since none of the rules (i.e. `(1)`, `(2)` and `(3)`) can be applied. In this case, it is when we can state that there exists no edges from node `c`. The failure value, `F`, can be also used in order to build terms with failure, that is terms including `F`. For instance, the expression `listpath c` denotes $[\mathtt{c}, \mathtt{F}, \mathtt{F}, \ldots]$, given that the expression `edge F` denotes `F` too. However, in absence of information, the undefined value $\perp$ incorporated by *CRWL* keeps on being useful, since it can be used with the same role as in functional programming; that is, to define partial approximations to the value of a function, and to provide semantics to functions with a cyclic definition, or even functions with an undefined condition such as:

```
(7) cycle X := cycle X.
(8) cycle2 X := 0 ⇐ cycle X ⋈ 0.
```

where, in order to apply rule `(8)`, the equality constraint `cycle X ⋈ 0` has to be solved. In both rules, `cycle X` and `cycle2 X` define as semantics the unitary set $\{\perp\}$. Finally, let us remark that both values $\perp$ and `F` are used from a semantic point of view, and they can be never used to explicitly provide values to functions. For more details about the frameworks *CRWL* (resp. *CRWLF*), we recommend to the interested readers the papers [12] (resp. [22, 23]).

As a consequence, the proposed data model can also deal with nonexistent information and partially nonexistent information (i.e. information that does not exist, and information that exists partially, respectively).

Finally, we propose a *data definition language* which, basically, consists of *database schema definitions*, *database instance definitions* and *(lazy) function definitions*.

A *database schema definition* includes *relation names* and a sequence of *attributes* for each defined relation name. For a given database schema, the *database instance definitions* define *key* and *non-key attribute values*, by means of *(constructor-based) conditional rewriting rules* [12, 22]. The rewriting rules include conditions which allow us to handle equality and inequality constraints. Moreover, we can define a set of lazy functions to be used by the queries, which allows us to deal with recursively defined datatypes. In a database setting, these functions are also named *interpreted functions*. As a consequence, "pure" functional logic programs (i.e. programs without negation) can be considered as a particular case of our programs.

With respect to (2) (i.e. to *propose query formalisms based on extensions of the relational calculus and algebra which handle the proposed data model*), typically, the query language involved in a functional logic language will be based on the solving of conjunctions of equality and inequality constraints. These constraints are defined w.r.t. some equality and inequality relations defined over terms [12, 22].

In the context of query languages, the proposed extended relational calculus will handle *conjunctions* of *atomic formulas*, which represent *relation predicates*, *equality and inequality relations* over terms, and *approximation equations* used for dealing with interpreted functions. Finally, the logic formulas will be either existentially or universally quantified, depending on whether they include negation or not.

Analogously, and w.r.t. the proposed extended relational algebra, it will deal with equality and inequality constraints, complex values, and interpreted functions. With this aim, we will generalize the *selection* and *projection* operators in a double sense, allowing: (a) to select tuples satisfying a set of equality and inequality constraints; and (b) to restructure complex values by applying *data constructors and destructors*, and *interpreted functions* and their *inverses* over the attribute values of a given schema instance.

Finally, and w.r.t. (3) (i.e. to provide *semantic foundations to the different query languages*), we will prove that our relational calculus and algebra are *equivalent query formalisms*, as well as, equivalent to the query language involved in most existent functional logic languages. Let us remark that this query language will be based on *equality* and *inequality constraints*.

Finally, let us remark that this work goes toward the design of a functional logic deductive language, called $\mathcal{INDALOG}$. The $\mathcal{INDALOG}$ project involves the design of an *operational semantics*, already studied in [3, 6], a *data model*, which has been firstly presented in [5], and two alternative query formalisms: an *extended relational calculus* [5] and an *extended relational algebra* [4]. The relation of this paper with the just mentioned ones is as follows: here, we will compare both alternative query formalisms (i.e. the extended relational calculus and algebra), showing its equivalence, and we will prove the equivalence of both query formalisms w.r.t. the built-in functional logic query language; however, in this paper and w.r.t. [4], we will consider the extension for the handling of negation.

1.2 Benefits of the Approach

The benefits of our approach come from the integration of functional logic programming and databases.

From the point of view of functional-logic programming, programmers in this paradigm can use a functional-logic language for programming databases. Therefore, this integration opens an application field to this kind of languages. In addition, it is known that declarative languages (i.e. logic, functional, functional-logic) are very useful for some specialized tasks and the connectivity of these languages with databases opens new lines of cooperation. In fact, this connectivity can be considered as the main benefit of our approach.

Nevertheless, functional-logic databases are adapted to the philosophy of functional-logic programming. Declarative programmers can define a simple database schema, and define a database instance by means of a set of conditional rewriting rules. Database instances can handle partial and infinite data, natural in functional-logic programming. In addition, they could use the query language based on equality and inequality constraints, to which they can be more habituated, or alternatively, they can use more database oriented languages, such as the extended relational calculus and algebra. For instance, from the relational calculus, the translation to a SQL style syntax is not too complicated, allowing the programmer to use this syntax. In addition, let us remark that the development of an extended relational algebra keeps on being very interesting, since this query formalism is suitable for the design of an operational semantics based on the application of the algebra operators. This aspect is out of the scope of this paper, but we will consider it as future work.

From the point of view of database programmers, we know and assume that functional-logic programming is not the most natural and typical framework. Therefore, we think that our main contribution to this context is the study of complex data models and query languages, which basically handle more sophisticated data (for instance, partial and infinite data, multi-valued attributes and handling of constraints over multi-valued attributes). The need of handling partial data and multi-valued attributes in databases is widely known [19, 30]. Now, the main question is why to use infinite data. Traditional relational databases work with simple data models, and as a consequence the application field of traditional relational databases is limited. Once recognized the importance for the database context of storing less standard data, like spatial and temporal data, it is obvious that we need to handle new data models. A spatial object can be infinite or, at least, its infiniteness should be handled in an efficient way. Similarly with temporal data. Infinite data can be handled as follows: (1) by means of a symbolic representation, like an equation; or (2) by means of (possibly infinite) data structures, which are computed, as much as needed, by using a lazy evaluation. (2) is the representation used by the declarative languages, such as functional and functional-logic languages. In any case, we are convinced that our contribution in this field is more theoretical than practical.

With respect to aspects of efficiency, the functional and logic languages are rich in expressivity once infinite and partial data, and non-determinism are introduced. Obviously, these features can cause a loss of efficiency when, for instance, a query involves infinite data. However, it is important to remark that the infinite data are lazily handled, and thus not all the aspects of efficiency are negative. This means an infinite data is evaluated as far as needed up to obtain the answer. In our approach, we have also taken into account this improvement of efficiency, since our language is lazy. Nevertheless, we have studied some aspects relative to efficiency, which have been considered in the deductive logic languages; concretely, the top-down evaluation, usual in (functional) logic languages, is not a suitable evaluation method from the point of view of disk accesses. For this reason, we have developed an operational semantics [3, 6] based on *magic sets and a goal-directed bottom-up evaluation* for functional-logic programs. Moreover, this evaluation mechanism remains the lazy component of functional logic languages. In fact, the adoption of the operational semantics based on a goal-directed bottom-up evaluation allows us to keep, at least, the same efficiency as other functional-logic languages.

Finally, we will prove that the three alternative query languages (i.e. functional-logic query language, extended relational calculus and extended relational algebra) are equivalent; that is, they have the same expressivity. Here, we are not interested in describing evaluation mechanisms for each of them. In fact, for the functional-logic query language (i.e. based on equality and inequality constraints), an evaluation mechanism has been already studied in [3, 6]. Here, our interest is focused on showing three kinds of equivalent query syntaxis, and the development of operational semantics for the formalisms based on extensions of relational languages is considered as future work.

1.3 Organization of the Paper

The organization of this paper is as follows. Section 2 describes the data model; Section 3 presents a safe functional logic query language based on (in)equality constraints; Section 4 defines the extended safe relational calculus; Section 5 shows the extended relational algebra; Section 6 states the equivalence results between all the query languages; Section 7 shows a precise comparison between the related work and the proposed approach; and finally, Section 8 shows the conclusions and future work. In addition, we have included an Appendix wherein we show the proofs of lemmas needed for the equivalence results presented in Section 6.

§2 The Data Model

Our data model consists of complex values and partial information, which can be handled in a data definition language based on conditional constructor-based rewriting rules.

2.1 Complex Values

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

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

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

Next, we will define a set of equality and inequality relations over these kinds of information. These relations should consider the defined values (i.e. $\perp$, F and totally defined), the defined partial information (i.e. partially undefined and partially nonexistent information); in addition, these relations should assume that the undefined value (i.e. $\perp$) cannot be compared with other values. Lastly, these relations are defined as follows:

(1) $=$ (*syntactic equality*), expressing that *two values are syntactically equal*; for instance, the relation s&b(3000, F) $=$ s&b(3000, F) is satisfied;

(2) $\downarrow$ (*strong equality*), expressing that *two values are equal and totally defined*; for instance, the relation s&b(3000, 25) $\downarrow$ s&b(3000, 25) holds;

(3) $\uparrow$ (*strong inequality*), where *two values are (strongly) different, if they are different in their defined information*; for instance, the relation s&b(3000, $\perp$) $\uparrow$ s&b(2000, 25) is satisfied;

In addition, we will define their corresponding inequality relations; that is, $\neq$, $\not\downarrow$ and $\not\uparrow$, representing a *syntactic inequality*, *weak inequality* and *weak equality* relation, respectively:

(1') $\neq$ (*syntactic inequality*), expressing that *two values are not syntactically equal*; for instance, the relation s&b(3000, 100) $\neq$ s&b(4000, 100) is satisfied;

(2') $\not\downarrow$ (*weak inequality*), expressing that *two values are different in its defined information, or they include* nonexistent information; for instance, the relations s&b(3000, 25) $\not\downarrow$ s&b(4000, 100) and s&b(3000, F) $\not\downarrow$ s&b (3000, 25) are satisfied;

(3') $\not\uparrow$ (*weak equality*), expressing that *two values are equal, although they include* nonexistent information; for instance, the relations s&b(3000, F) $\not\uparrow$ s&b(3000, F) and s&b(3000, F) $\not\uparrow$ s&b(3000, $\perp$) are satisfied.

As we have just mentioned, the relations if *strong equality* and *strong inequality* only compare defined information. However the relations *weak equality* and *weak inequality* take into account the (possible) presence of partially non-existent information. For instance, the relation *weak equality* considers that the value F is equal to any value. Now, the question is: "Which is the fact we want to express?" The answer is the following: values which are not different due to defined information, and thus we can consider them in some sense (i.e. *weak*) "equal".

Let us remark that the negations (i.e. $\neq$, $\not\downarrow$ and $\not\uparrow$) do not express the "logical negation" due to the presence of undefined information (i.e. $\perp$); for instance, given the values $\mathsf{s\&b(3000, 100)}$ and $\mathsf{s\&b}(3000, \perp)$, then we have that neither the relation $\mathsf{s\&b(3000, 100)} \downarrow \mathsf{s\&b}(3000, \perp)$ nor the relation $\mathsf{s\&b(3000, 100)} \not\downarrow \mathsf{s\&b}(3000, \perp)$ are satisfied.

Next, we will show the needed technical preliminaries for defining the above equality and inequality relations.

Assuming a set of *constructor symbols* $c, d, \ldots$ $DC = \cup_n DC^n$, each one with an associated arity, the symbols $\perp$, F as special cases with arity 0 (not included in DC), and a set $\mathcal{V}$ of variables $X, Y, \ldots$, we can build the set of *c-terms with* $\perp$ *and* F, denoted by $CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$. C-terms are complex values, including variables which are, implicitly, universally quantified. We denote by $cterms(t)$ the set of (sub)c-terms occurring in t. Given a set of data constructors S, *we say that two c-terms* t *and* t' *have an* S*-clash if they have different constructor symbols of* S *at the same position.* In the same way, *we say that two c-terms* t *and* t' *have an* $S \cup \{\mathsf{F}\}$*-clash if they have different symbols of* $S \cup \{\mathsf{F}\}$ *at the same position.* In addition, we can use *substitutions* $Subst_{DC,\perp,\mathsf{F}} = \{\eta \mid \eta : \mathcal{V} \to CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})\}$, in the usual way, where the domain of a substitution η, denoted by $Dom(\eta)$, is defined as usual. id denotes the identity substitution. Now, the above (in)equality relations can be formally defined as follows.

Definition 2.1 (Relations over Complex Values [22])
Given two c-terms $t, t' \in CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$, then:

(1) $t = t' \Leftrightarrow_{def} t$ and t' are syntactically equal;

(2) $t \downarrow t' \Leftrightarrow_{def} t = t'$ and $t \in CTerm_{DC}(\mathcal{V})$; that is t is a totally defined c-term;

(3) $t \uparrow t' \Leftrightarrow_{def} t$ and t' have a DC-clash.

In addition, their 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 sub-term, or they have a DC-clash;
(3') $\not\uparrow$ is defined as the least symmetric relation over $CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V})$ 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 $c \in DC^n$.

Given that complex values can be partially defined, a *partial ordering* $\leq$ can be considered. This ordering is 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*, given that $\bot$ represents undefined information; that is, information more refinable can exist. In addition, F is *maximal* under $\leq$ (F satisfies the relations $\bot \leq \mathsf{F}$ and $\mathsf{F} \leq \mathsf{F}$), representing nonexistent information); that is, no further refinable information can be obtained, given that it does not exist. Definitely, the idea that we want to state is that F cannot be more refinable; concretely, it cannot be used as a partial approximation to any c-term.

Now, we can consider (*possibly infinite*) sets of (*partial*) c-terms, denoted by $\mathcal{SET}(CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V}))$. In our framework, these sets are used for representing multi-valued attributes as well as the output from non-deterministic functions. Finally, we denote by $cterms(\mathcal{CV})$ the set of (sub)c-terms of the c-terms of $\mathcal{CV}$, where $\mathcal{CV} \in \mathcal{SET}(CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V}))$.

Given that these sets can be infinite and c-terms can be also infinite, we need to define an order over sets, representing an *approximation ordering* over (possibly infinite) sets of c-terms. This approximation ordering is defined as follows: *given* $\mathcal{CV}_1, \mathcal{CV}_2 \in \mathcal{SET}(CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V}))$*, then* $\mathcal{CV}_1 \sqsubseteq \mathcal{CV}_2$ *iff for all* $t_1 \in \mathcal{CV}_1$ *there exists* $t_2 \in \mathcal{CV}_2$ *such that* $t_1 \leq t_2$*, and, in addition, for all* $t_2 \in \mathcal{CV}_2$ *there exists* $t_1 \in \mathcal{CV}_1$ *such that* $t_1 \leq t_2$.

Let us remark that a multi-valued attribute or a non-deterministic function can represent an infinite set of (infinite) values; however, in our framework, we have that multi-valued attributes and non-deterministic functions will be lazily handled, in such a way that their corresponding values will be approximated by finite partial approximations w.r.t. the given order.

Finally, we need to define the following *equality* and *inequality* relations over sets of c-terms.

Definition 2.2 (Relations over Sets of Complex Values)
Given $\mathcal{CV}_1$ and $\mathcal{CV}_2 \in \mathcal{SET}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$, then:

(1) $\mathcal{CV}_1 \bowtie \mathcal{CV}_2$ holds, iff there exist $t_1 \in \mathcal{CV}_1$ and $t_2 \in \mathcal{CV}_2$ such that t_1 and t_2 are finite and *strongly equal*; and

(2) $\mathcal{CV}_1 <> \mathcal{CV}_2$ holds, iff there exist $t_1 \in \mathcal{CV}_1$ and $t_2 \in \mathcal{CV}_2$ such that t_1 and t_2 are *strongly different*;

and their negations:

(1') $\mathcal{CV}_1 \not\bowtie \mathcal{CV}_2$ holds, iff for all $t_1 \in \mathcal{CV}_1$ and $t_2 \in \mathcal{CV}_2$, we have that t_1 and t_2 are *weakly different*; and

(2') $\mathcal{CV}_1 \not\!<\!> \mathcal{CV}_2$ holds, iff for all $t_1 \in \mathcal{CV}_1$ and $t_2 \in \mathcal{CV}_2$, we have that t_1 and t_2 are *weakly equal*.

The above relations are defined for (possibly infinite) sets of (possibly infinite) values. However, in the case of infinite values and sets, these relations can be still used, taking finite and partial values. Like lazy functional-logic languages, our proposed language will handle partial approximations which, in our case, are built for sets of c-terms, considering the order $\sqsubseteq$. This order ensures the following nice property: *the use of partial approximations is sound*, that is, for every $\mathcal{CV}_1$ and $\mathcal{CV}_2$ if $\mathcal{CV}_1 \bowtie \mathcal{CV}_2$ then $\mathcal{CV}'_1 \bowtie \mathcal{CV}'_2$ for any $\mathcal{CV}_1 \sqsubseteq \mathcal{CV}'_1$ and $\mathcal{CV}_2 \sqsubseteq \mathcal{CV}'_2$; similarly with the rest of relations (i.e. $<>$, $\not\bowtie$ and $\not\!<\!>$).

2.2 Data Definition Language

We propose a *data definition language* which, basically, consists of *database schema definitions*, *extended database schema definitions*, *database schema instance definitions*, and *database instance definitions*.

Briefly, a *database schema definition* includes *relation names*, and a sequence of *attributes* for each relation name. For a given database schema, an *extended database schema definition* includes, in addition, a set of constructor and functional symbols. Next, a *database schema instance* defines a set of tuples including values for the key and non-key attributes. Moreover, a *database instance* defines a database schema instance and a set of interpretations for constructor and functional symbols. The functions, used by queries, allow us to deal with recursively defined datatypes. In a database setting, these functions are called *interpreted functions*. Finally, the values of tuples included by a database instance are defined by means of *(constructor-based) conditional rewriting rules*. The conditional rewriting rules include conditions which handle equality and inequality constraints.

Definition 2.3 (Database Schemas)
Assuming a Milner's [24)] style polymorphic type system, a *database schema* S is a finite sequence of *relation schemas* $R_1, \ldots, R_m$, wherein the *relation names* are $R_1, \ldots, R_m$, each relation schema R_i, $1 \leq i \leq m$, has the form $R_i(\underline{A_1} : T_1, \ldots, \underline{A_k} : T_k, A_{k+1} : T_{k+1}, \ldots, A_n : T_n)$, the *attribute names* are a sequence $A_1, \ldots, A_n$, and, finally, the *attribute types* are $T_1, \ldots, T_n$. In each relation schema R_i, the underlined attributes $A_1, \ldots, A_k$ represent the *key attributes*, and $A_{k+1}, \ldots, A_n$ are the *non-key attributes*, denoted by the sets $Key(R_i)$ and $NonKey(R_i)$, respectively. Finally, the sets of attribute names of any two relation schemas are disjoint. We can assume attribute names qualified with the relation name when the attribute names coincide.

The sequence of values of the key attributes of a tuple is assumed to identify this tuple. Although a relation may include several sequences of key attributes, we have that one of these sequences should identify the tuples of the relation (such as a *primary key*). Finally, given a relation schema $R_i(\underline{A_1} : T_1, \ldots, \underline{A_k} : T_k, A_{k+1} : T_{k+1}, \ldots, A_n : T_n)$, then we denote by $nAtt(R_i) = n$ and $nKey(R_i) = k$, the number of attributes and key attributes defined in the relation schema R_i, respectively.

Definition 2.4 (Extended Database Schemas)
An *extended database schema* D is a triple (S, DC, IF), where S is a *database schema*, $DC = \cup_{n \geq 0} DC^n$ is a set of *constructor symbols*, and $IF = \cup_{n \geq 0} IF^n$ represents a set of *interpreted function symbols*.

We denote the set of *defined schema symbols* of an extended database schema D (i.e. relation and non-key attribute symbols of D) by $DSS(D)$, and the set of *defined symbols* of D by $DS(D)$ (i.e. $DSS(D)$ together with IF). Let us remark that the instances for an extended database schema $D = (S, DC, IF)$ (defined above) will require tuples for S, including sets of c-terms built from DC, and the interpreted functions of IF are defined for terms built from DC. As an example of extended database schema, we can consider the following one:

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

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 *"views"* `person_boss_job` and `peter_workers`, considered as such, since they will take key attribute values from the set of key values defined for the relation schema `person_job`.

The first view, `person_boss_job`, will include, for each person, pairs in the form of records constituted by: (a) his/her boss and boss' age, by using the data constructor `b&a` (attribute `bos_age`); and (b) his/her job and job salary bonus, by using the data constructor `j&b` (attribute `job_bonus`). The second view, `peter_workers`, will include workers whose boss is `peter`. The set DC includes constructor symbols for the types `people`, `job`, `dir`, `cbossage` and `cjobbonus`, and IF defines the interpreted function symbol `retention_for_tax`, which will compute the 'tax free' salary.

In addition, we can consider extended database schemas involving *(possibly) infinite database instances (i.e. infinite tuples and tuples with infinite values)*. For instance, we can consider the following extended database schema, which shows how to handle simple spatial objects.

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

Here, the relation schemas `2Dpoint` and `2Dline` are defined in order to represent bidimensional points and lines, respectively. `2Dpoint` includes the point coordinates (attribute `coord`) and `color`. Lines represented by `2Dline` are defined by using a starting point (attribute `origin`) and direction (attribute `dir`). Furthermore, `next` indicates the next point to be drawn in the line, `points` stores the *(infinite) set* of points of this line, and `list_of_points` the *(infinite) list* of points of the line. Here, we can see the double use of complex values: (1) attribute `points` as a set (which can be implicitly assumed); and (2) attribute `list_of_points` as a list of values. Let us remark that the attribute `points` represents the infinite set of points of a line. However, we can handle a finite approximation to this set, concretely, a subset of the points in such a way that we can use the relations $\bowtie$ and $<>$ for comparing this attribute with another finite set of values. Obviously, it only works in lucky cases: the relations $\bowtie$ and

$<>$ can deliver neither true nor false, when one of the sets is infinite, and the relations cannot be checked for finite and partial approximations. This is the case in which laziness cannot be useful.

Definition 2.5 (Schema Instances)
Given a database schema S with sequence of relation names $R_1, \ldots, R_p$, then a *schema instance* $\mathcal{S}$ of S is a sequence of *relation instances* $\mathcal{R}_1, \ldots \mathcal{R}_p$, where each $\mathcal{R}_i$ is an instance of the relation R_i $(1 \le i \le p)$. Now, each $\mathcal{R}_i$ includes a (possibly infinite) set of tuples of the form $(V_1, \ldots, V_n)$, where:
(1) $n = nAtt(R_i)$;
(2) each V_j $(1 \le j \le nKey(R_i))$ satisfies that $V_j \in CTerm_{DC,\mathsf{F}}(\mathcal{V})$; and
(3) $V_l \in \mathcal{SET}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$ for each $nKey(R_i) + 1 \le l \le n$.

With respect to the above Definition, we have that the key attribute values have to be one-valued and they cannot include $\perp$. As will be seen later, the form of the rules that define the key attribute values will justify this restriction. However, non-key attributes can be multi-valued with an infinite set of values and infinite values. In this case, the non-deterministic nature of the rules that define the non-key attribute values will justify this restriction. In addition, the attribute values can be non-ground (i.e. including variables), wherein the variables are implicitly universally quantified. From now on, and in order to simplify the notation, we assume that the instance of a relation name R, will be denoted by calligraphic style, such as $\mathcal{R}$. Finally, let us remark that we can assume that attribute values are typed in the corresponding attribute types; however types are not necessary for the correctness results of our approach.

Definition 2.6 (Database Instances)
A *database instance* $\mathcal{D}$ for an extended database schema $D = (S, DC, IF)$ is a triple

$$(\mathcal{S}, \mathcal{DC}, \mathcal{IF})$$

where $\mathcal{S}$ is a schema instance of the database schema S, $\mathcal{DC} = CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$, and $\mathcal{IF}$ is a set of *function interpretations* defined as $f^{\mathcal{D}} : CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})^n \to \mathcal{SET}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$ for each $f \in IF^n$, where each $f^{\mathcal{D}}$ is monotonic; that is, $f^{\mathcal{D}}(t_1, \ldots, t_n) \sqsubseteq f^{\mathcal{D}}(t'_1, \ldots, t'_n)$ if $t_i \le t'_i$, $1 \le i \le n$.

Functions are monotonic w.r.t. the approximation ordering defined over c-terms (i.e. $\le$). Deterministic functions define a unitary set; otherwise they represent non-deterministic functions, which can represent a set of c-terms.

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

$$
\begin{array}{ll}
\texttt{person\_job} & \left\{ \begin{array}{l} (\texttt{john}, \{\bot\}, \{\texttt{add}('\texttt{6th Avenue}', 5)\}, \{\texttt{lecturer}\}, \{\texttt{mary}, \texttt{peter}\}) \\ (\texttt{mary}, \{\bot\}, \{\texttt{add}('\texttt{7th Avenue}', 2)\}, \{\texttt{associate}\}, \{\texttt{peter}\}) \\ (\texttt{peter}, \{\bot\}, \{\texttt{add}('\texttt{5th Avenue}', 5)\}, \{\texttt{professor}\}, \{\mathsf{F}\}) \end{array} \right. \\
\texttt{job\_information} & \left\{ \begin{array}{l} (\texttt{lecturer}, \{1500\}, \{\mathsf{F}\}) \\ (\texttt{associate}, \{2500\}, \{\mathsf{F}\}) \\ (\texttt{professor}, \{4000\}, \{1500\}) \end{array} \right. \\
\texttt{person\_boss\_job} & \left\{ \begin{array}{l} (\texttt{john}, \{\texttt{b\&a}(\texttt{mary}, \bot), \texttt{b\&a}(\texttt{peter}, \bot)\}, \{\texttt{j\&b}(\texttt{lecturer}, \mathsf{F})\}) \\ (\texttt{mary}, \{\texttt{b\&a}(\texttt{peter}, \bot)\}, \{\texttt{j\&b}(\texttt{associate}, \mathsf{F})\}) \\ (\texttt{peter}, \{\texttt{b\&a}(\mathsf{F}, \bot)\}, \{\texttt{j\&b}(\texttt{professor}, 1500)\}) \end{array} \right. \\
\texttt{peter\_workers} & \left\{ \begin{array}{l} (\texttt{john}, \{\texttt{lecturer}\}) \\ (\texttt{mary}, \{\texttt{associate}\}) \end{array} \right.
\end{array}
$$

As can be seen, each instance tuple includes the key attribute values, as well as the non-key attribute values grouped by sets of c-terms. Firstly, in the instance of the relation schema `person_job`, all the tuples include the value $\bot$ for the attribute `age`, representing undefined information; that is, information which has not been included in the database, although it can exist. Secondly, the third tuple in the instance of the relation `person_job` includes the symbol F in the attribute `boss`, representing nonexistent information; that is, `peter` has no boss. In the same way, the first and second tuple in the instance of the relation `job_information` also include nonexistent information, expressing that jobs `lecturer` and `associate` have no salary bonus.

Finally, the presence of partially undefined and partially nonexistent information occurs in the instance of the "view" `person_boss_job`. For instance, the first tuple includes partially undefined, $\texttt{b\&a}(\texttt{mary}, \bot)$, and partially nonexistent, $\texttt{j\&b}(\texttt{lecturer}, \mathsf{F})$, information; in the first case, $\texttt{b\&a}(\texttt{mary}, \bot)$ expresses that `john`'s boss is known (i.e. `mary`), but `mary`'s age is undefined (there can exist the age, but it is unknown). In the second case, $\texttt{j\&b}(\texttt{lecturer}, \mathsf{F})$ expresses that `john`'s job is `lecturer`, but `john` has no salary bonus (F).

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

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

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

$$\mathtt{2Dline}\ \left\{\begin{array}{l} (\mathtt{p}(0,0), \mathtt{north}, \{\mathtt{p}(0,1)\}, \{\mathtt{p}(0,0), \mathtt{p}(0,1) \mid \bot\}, \{[\mathtt{p}(0,0), \mathtt{p}(0,1), \mathtt{p}(0,2) \mid \bot]\}) \\ (\mathtt{p}(1,0), \mathtt{north}, \{\mathtt{p}(1,1)\}, \{\mathtt{p}(1,0), \mathtt{p}(1,1) \mid \bot\}, \{[\mathtt{p}(1,0), \mathtt{p}(1,1), \mathtt{p}(1,2) \mid \bot]\}),\ \ldots \\ (\mathtt{p}(1,1), \mathtt{east}, \{\mathtt{p}(2,1)\}, \{\mathtt{p}(1,1), \mathtt{p}(2,1) \mid \bot\}, \{[\mathtt{p}(1,1), \mathtt{p}(2,1), \mathtt{p}(3,1) \mid \bot]\}),\ \ldots \end{array}\right.$$

By using this partial approximation, we can compare attribute `points` with the relations $\bowtie$ or $\Diamond$; for instance, consider the lines with key attribute values $(\mathtt{p}(1,0), \mathtt{north})$ and $(\mathtt{p}(1,1), \mathtt{east})$. In this case, the relations $\mathtt{points}\ \mathtt{p}(1,0)\ \mathtt{north} \bowtie \mathtt{points}\ \mathtt{p}(1,1)\ \mathtt{east}$ and $\mathtt{points}\ \mathtt{p}(1,0)\ \mathtt{north} \Diamond \mathtt{points}\ \mathtt{p}(1,1)\ \mathtt{east}$ are true, given that both lines intersect (i.e. $\bowtie$ is satisfied), but they are different lines (i.e. $\Diamond$ holds). In this way, we can handle these infinite sets in our approach.

On the other hand, the values included by a database instance (i.e. key and non-key attribute values, and interpreted function values) for an extended database schema are stated by means of *constructor-based conditional rewriting rules.* Next, we formally define the conditional rewriting rules.

Definition 2.7 (Conditional Rewriting Rules)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$, then a *constructor-based conditional rewriting* rule RW for a symbol $H \in DS(D)$ has the form $H\ t_1 \ldots t_n := r \Leftarrow C$, representing that r *is the value of* $H\ t_1 \ldots t_n$*, whenever the condition* C *is satisfied.* In this kind of rule, we have that:

(1) $(t_1, \ldots, t_n)$ is a linear tuple (i.e. each variable in it occurs only once) with $t_i \in CTerm_{DC}(\mathcal{V})$;

(2) $r \in Term_D(\mathcal{V})$, where $Term_D(\mathcal{V})$ represents the *terms* or *expressions* built from D (i.e. terms or expressions built from DC, $DS(D)$ and variables of $\mathcal{V}$);

(3) C is a set of constraints of the form $e \bowtie e', e \Diamond e', e \not\bowtie e', e \not\Diamond e'$, where $e, e' \in Term_D(\mathcal{V})$; and

(4) *extra variables* are not allowed, i.e. $var(r) \cup var(C) \subseteq var(t_1, \ldots, t_n)$.

Let us remark that both $\bot$ and F are only used at the semantic level, and thus they are not included in $Term_D(\mathcal{V})$. However, each term or expression e represents a set of c-terms (i.e. an element of $\mathcal{SET}(CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V}))$), in such a way that the set of constraints C allows us to compare sets of c-terms accordingly to the semantics of the relations defined over sets of complex values; that is, $\bowtie, \Diamond, \not\bowtie, \not\Diamond$ (see Definition 2.2).

Finally, rules cannot include extra variables in conditions. This is due to the handling of negation in functional-logic programming. Extra variables in negative conditions are universally quantified and they would have a sophisticated and costly operational behavior.

Definition 2.8 (Database Instance Values)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ for an extended database schema $D = (S, DC, IF)$, then the *database instance values* of $\mathcal{D}$ are defined by means of the following set of conditional rewriting rules:

- Rules of the form $R\ t_1\ \dots\ t_k := r \Leftarrow C$, where r is a term of type `typeok` which consists of a unique special value `ok` (representing a shorthand of *object key*), and $nKey(R) = k$; this kind of rules allows us to define a new tuple with key attribute values $t_1, \dots, t_k$ in the instance $\mathcal{R}$ of $\mathcal{S}$ for the relation R of S;
- Rules $A\ t_1\ \dots\ t_k := r \Leftarrow C$, where $A \in NonKey(R)$, $R \in S$, $nKey(R) = k$, set r as the value of the non-key attribute A in the tuple with key values $t_1, \dots, t_k$ of the instance $\mathcal{R}$ of $\mathcal{S}$ for the relation R of S, whenever the set of constraints C holds.
- Rules $f\ t_1\ \dots\ t_n := r \Leftarrow C$, where $f \in IF^n$, set r as the value of $f\ t_1\ \dots\ t_n$ whenever the set of constraints C holds.

In all the kinds of rules, $t_1, \dots, t_n, r$ can be non-ground values, and thus the key and non-key attribute values can also represent non-ground values. Rules for the non-key attributes $A\ t_1\ \dots\ t_k := r \Leftarrow C$ are implicitly constrained to the form $A\ t_1\ \dots\ t_k := r \Leftarrow R\ t_1\ \dots\ t_k \bowtie \texttt{ok},\ C$, in order to guarantee that $t_1, \dots, t_k$ are key values defined in a tuple of R. For instance, the values of the above database instance relative to people and job information can be defined by the following rules:

```
person_job
    person_job john := ok.                 person_job mary := ok.
    person_job peter := ok.
    address john := add('6th Avenue', 5).
    address mary := add('7th Avenue', 2).
    address peter := add('5th Avenue', 5).
    job_id john := lecturer.               job_id mary := associate.
    job_id peter := professor.
    boss john := mary.                     boss john := peter.
    boss mary := peter.
job_information
    job_information lecturer := ok.
    job_information associate := ok.
    job_information professor := ok.
    salary lecturer := retention_for_tax 1500.
    salary associate := retention_for_tax 2500.
    salary professor := retention_for_tax 4000.
    bonus professor := 1500.
```

person_boss_job	$\begin{cases} \texttt{person\_boss\_job Name} := \texttt{ok} \Leftarrow \texttt{person\_job Name} \bowtie \texttt{ok}. \\ \texttt{boss\_age Name} := \texttt{b\&a(boss Name, address (boss Name))}. \\ \texttt{job\_bonus Name} := \texttt{j\&b(job\_id (Name), bonus (job\_id (Name)))}. \end{cases}$
peter_workers	$\begin{cases} \texttt{peter\_workers Name} := \texttt{ok} \Leftarrow \texttt{person\_job Name} \bowtie \texttt{ok}, \texttt{boss Name} \bowtie \texttt{peter}. \\ \texttt{work Name} := \texttt{job\_id Name}. \end{cases}$
retention_for_tax	$\{ \texttt{retention\_for\_tax Fullsalary} := \texttt{Fullsalary} - (0.2 * \texttt{Fullsalary}).$

Let us remark that the condition C can be used in order to define *views*, such as shown in the rule that defines the key attribute values for person_boss_job (i.e. person_boss_job Name := ok ⇐ person_job Name ⋈ ok). Here, this rule indicates that the key attribute values defined for person_job are also valid for the view person_boss_job.

Furthermore, it is important to remark that undefined information is interpreted, whenever *there are no rules* for a given non-key attribute (for instance, attribute age in relation person_job; see the set of values $\{\perp\}$ in the previous presented instance of the relation *person_job* for all defined key values). However, whenever a non-key attribute is defined by *at least one rule*, it is assumed that the tuples for which either the attribute is not defined or the constraints of the rule are not satisfied, include nonexistent information as value (for instance, attribute boss in relation person_job for the key value peter; that is, we set boss values for john and mary, but not for peter). This behavior fits with the failure of reduction of conditional rewriting rules proposed in [22]. Once $\perp$ and F are introduced as special cases of attribute values, the view person_boss_job will include partially undefined and partially nonexistent information. Finally, as previously mentioned, and due to the form of defining the key attribute values, we have that person_boss_job and peter_workers can be considered as "views" defined from the relation schema person_job.

Comparing our approach with other kinds of database languages based on declarative programming (i.e. logic and functional languages), we have that our data model enriches the data models proposed by functional and logic programming due to, mainly, the presence of multi-valued attributes in the form of (*possibly infinite*) sets of (*possibly infinite*) c-terms; for instance, we can consider the attribute points with the (infinite) set of values $\{\mathtt{p}(0,0), \mathtt{p}(0,1), \mathtt{p}(0,2), \ldots\}$. As far as we know, none of above approaches (i.e. functional and logic data model) is able to handle this kind of information. And even more, the use of F in order to explicitly represent the non-existence of values for a given attribute introduces a new mechanism for the handling of negation in deductive databases. Finally, let us remark that the proposed query languages in the following sections will handle nicely both aspects (i.e. (possibly infinite) sets of (possibly infinite)

Table 1 Examples of (Functional-Logic) Queries

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

c-terms and nonexistent information).

§3 Safe Functional Logic Query Language

Now, we can consider a *(functional logic) query language*, involving *queries* similar to the condition of a conditional rewriting rule. For instance, the (functional logic) query $\mathcal{Q}_{\mathtt{s}} \equiv$ `retention_for_tax X` $\bowtie$ `salary (job_id peter)` w.r.t the instances of the relation schemas `person_job` and `job_information`, requests `peter`'s full salary, obtaining as answer {X/4000}. Table 1 shows some other examples, with their corresponding meanings and expected answers.

On the other hand, in database theory it is known that any query language must ensure the so-called property of *domain independence* [2]. A query is *domain independent*, whenever the query satisfies, properly, two conditions: *(a) the query output over a finite relation is also a finite relation; and (b) the output relation only depends on the input relations.* In general, it is undecidable, and thus syntactic conditions have to be developed in such a way that, only the so-called *safe queries* (satisfying these conditions) ensure the property of domain independence. For example, in [2] the variables occurring in queries must be *range restricted*. In our case, we generalize the notion of *range restriction* to c-terms. In addition, the safety conditions should ensure the equivalence between the functional logic query language and the proposed alternative query formalisms, and based on extensions from relational calculus and algebra. Now, in order to define the safety conditions, we need the following previous definitions.

Definition 3.1 (Query Keys)
Given an extended database schema $D = (S, DC, IF)$, then the set of *query keys* of a key attribute $A_i \in Key(R)$ $(R \in S)$ occurring in a term $e \in Term_D(\mathcal{V})$, denoted by $query\_key(e, A_i)$, is defined as follows:

$$query\_key(e, A_i) =_{def} \begin{array}{l} \{t_i \in CTerm_{DC,\mathsf{F}}(\mathcal{V}) \mid there\ exists\ an\ expression \\ of\ the\ form\ H\ e_1 \ldots t_i \ldots e_k\ occurring\ in\ e\ and \\ H \in \{R\} \cup NonKey(R)\} \end{array}$$

Now, the set of query keys in a query $\mathcal{Q}$ w.r.t. an extended database schema $D = (S, DC, IF)$ is defined as follows:

$$query\_key(\mathcal{Q}, D) =_{def} \bigcup_{A_i \in Key(R), R \in S} query\_key(\mathcal{Q}, A_i) \text{ where}$$

$$query\_key(\mathcal{Q}, A_i) =_{def} \bigcup_{e \Diamond_q e' \in \mathcal{Q}} (query\_key(e, A_i) \cup query\_key(e', A_i))$$

with $\Diamond_q \in \{\bowtie, <>, \not\bowtie, \not<>\}$.

The underlying meaning of this Definition is that a query will select tuples of a database instance from the defined key attribute values. In fact, the query keys of a query represents the set of the selected key attribute values. Then, the range restricted condition will ensure that each c-term occurring in a query is either a query key or depends on a query key. In this way, we ensure that variables are always used for retrieving (sub-terms of) key or non-key attributes values.

Definition 3.2 (Range Restricted C-Terms of Queries)
Given an extended database schema $D = (S, DC, IF)$ and a query $\mathcal{Q}$, then a c-term t is *range restricted* in $\mathcal{Q}$ w.r.t. D, if either:

(1) t belongs to $\bigcup_{s \in query\_key(\mathcal{Q}, D)} cterms(s)$,or

(2) there exists a constraint $e \Diamond_q e'$, $\Diamond_q \in \{\bowtie, <>, \not\bowtie, \not<>\}$, such that t belongs to $cterms(e)$ (resp. $cterms(e')$) and every c-term occurring in e' (resp. e) is *range restricted* in $\mathcal{Q}$.

In the above Definition, $cterms(e)$ denotes the set of (sub)c-terms of a term $e \in Term_D(\mathcal{V})$. Range restricted c-terms occur in the scope of a relation symbol or a non-key attribute symbol, or they are compared (by means of equality and inequality constraints) with c-terms in the scope of a relation symbol or a non-key attribute symbol. Therefore, we have that all of them will take values from a defined schema instance.

Definition 3.3 (Safe Queries)
Given an extended database schema $D = (S, DC, IF)$ and a query $\mathcal{Q}$, then $\mathcal{Q}$ is *safe* w.r.t D if all c-terms occurring in $\mathcal{Q}$ are range restricted in $\mathcal{Q}$ w.r.t D.

For instance, let us consider the above query: $\mathcal{Q}_s \equiv$ `retention_for_tax X` $\bowtie$ `salary(job_id peter)`. $\mathcal{Q}_s$ is *safe*, since the constant `peter` is a *query key* (and thus *range restricted*); finally, the variable `X` is also *range restricted*, since `X` occurs in the right-hand of the query and in the left-hand side there are only range restricted c-terms (i.e. `peter`). Now, we will provide semantic foundations to the query language based on equality and inequality constraints. With this aim, firstly, we need to define the *denoted values* and the *active domain* of a database term in a functional logic query.

Definition 3.4 (Denotation of Database Terms)
Given a term $e \in Term_D(\mathcal{V})$ and a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$, then *the denotation of* e in $\mathcal{D}$ under a substitution η, represented by $[\![e]\!]^{\mathcal{D}}\eta$, is defined as follows:

(1) $[\![R\ e_1\ \ldots\ e_k\ ]\!]^{\mathcal{D}}\eta$ for all $R \in S$:

- $\star$ $[\![R\ e_1\ \ldots\ e_k\ ]\!]^{\mathcal{D}}\eta =_{def} \{\mathsf{ok}\}$, if there exist a tuple $(V_1, \ldots, V_k, V_{k+1}, \ldots, V_n) \in \mathcal{R}$, where $\mathcal{R}$ is a schema instance of $\mathcal{S}$ for the relation R of S and $k = nKey(R)$, and a substitution $\psi \in Subst_{DC,\perp,\mathsf{F}}$, such that $([\![e_1]\!]^{\mathcal{D}}\eta,\ \ldots\ , [\![e_k]\!]^{\mathcal{D}}\eta) = (V_1\psi, \ldots, V_k\psi)$ and $V_i\psi \in CTerm_{DC,\mathsf{F}}(\mathcal{V})$ with $1 \le i \le k$;
- $\star$ $[\![R\ e_1\ \ldots\ e_k\ ]\!]^{\mathcal{D}}\eta =_{def} \{\mathsf{F}\}$, if for all tuple $(V_1, \ldots, V_k, V_{k+1}, \ldots, V_n) \in \mathcal{R}$, with $\mathcal{R}$ instance of the relation R and $k = nKey(R)$, and substitution $\psi \in Subst_{DC,\perp,\mathsf{F}}$, then $[\![e_i]\!]^{\mathcal{D}}\eta \neq V_i\psi$ for some i, $1 \le i \le k$. However, there exist tuples $(W_1, \ldots, Z_i, \ldots, W_k, \ldots, W_n) \in \mathcal{R}$ and substitutions $\psi_i \in Subst_{DC,\perp,\mathsf{F}}$, such that $[\![e_i]\!]^{\mathcal{D}}\eta = Z_i\psi_i$ with $Z_i\psi_i \in CTerm_{DC,\mathsf{F}}(\mathcal{V})$ with $1 \le i \le k$;
- $\star$ $[\![R\ e_1\ \ldots\ e_k\ ]\!]^{\mathcal{D}}\eta =_{def} \{\mathsf{F}\}$, if $\eta = id$ and for all tuple $(V_1, \ldots, V_k, V_{k+1}, \ldots, V_n) \in \mathcal{R}$, with $\mathcal{R}$ instance of the relation R and $k = nKey(R)$, and substitution $\psi \in Subst_{DC,\perp,\mathsf{F}}$, then $[\![e_i]\!]^{\mathcal{D}}\eta \neq V_i\psi$ for some i, $1 \le i \le k$;
- $\star$ $[\![R\ e_1\ \ldots\ e_k\ ]\!]^{\mathcal{D}}\eta =_{def} \{\perp\}$ otherwise;

(2) $[\![A_i\ e_1\ \ldots\ e_k\ ]\!]^{\mathcal{D}}\eta$ for all $A_i \in NonKey(R)$:

- $\star$ $[\![A_i\ e_1\ \ldots\ e_k\ ]\!]^{\mathcal{D}}\eta =_{def}\ V_i\psi$, if there exist a tuple $(V_1, \ldots, V_k, V_{k+1}, \ldots, V_i, \ldots, V_n) \in \mathcal{R}$, with $\mathcal{R}$ instance of the relation R and $i > nKey(R) = k$, and a substitution $\psi \in Subst_{DC,\perp,\mathsf{F}}$, such that $([\![e_1]\!]^{\mathcal{D}}\eta, \ldots, [\![e_k]\!]^{\mathcal{D}}\eta)$

$= (V_1\psi, \ldots, V_k\psi)$ and $V_j\psi \in CTerm_{DC,\mathsf{F}}(\mathcal{V})$ with $1 \leq j \leq k$;

- ⋆ $[\![A_i\ e_1\ \ldots\ e_k\ ]\!]^{\mathcal{D}}\eta =_{def} \{\mathsf{F}\}$, if $[\![R\ e_1\ \ldots\ e_k\ ]\!]^{\mathcal{D}}\eta = \{\mathsf{F}\}$;
- ⋆ $[\![A_i\ e_1\ \ldots\ e_k\ ]\!]^{\mathcal{D}}\eta =_{def} \{\bot\}$ otherwise;

(3) $[\![X]\!]^{\mathcal{D}}\eta =_{def} \{X\eta\}$, for all $X \in \mathcal{V}$;

(4) $[\![c]\!]^{\mathcal{D}}\eta =_{def} \{c\}$, for all $c \in DC^0$;

(5) $[\![c(e_1, \ldots, e_n)]\!]^{\mathcal{D}}\eta =_{def} c([\![e_1]\!]^{\mathcal{D}}\eta, \ldots, [\![e_n]\!]^{\mathcal{D}}\eta)^{*1}$, for all $c \in DC^n$;

(6) $[\![f\ e_1\ \ldots\ e_n\ ]\!]^{\mathcal{D}}\eta =_{def} f^{\mathcal{D}}\ [\![e_1]\!]^{\mathcal{D}}\eta\ \ldots\ [\![e_n]\!]^{\mathcal{D}}\eta$, for all $f \in IF^n$.

The denoted values of a term or expression represent the set of values that a multi-valued (resp. one-valued) attribute or a non-deterministic (resp. deterministic) interpreted function defines. Whenever the schema instance includes variables, we need to instantiate it, in order to obtain the complete set of values represented by an attribute.

Expressions $R\ e_1 \ldots e_k$ denotes $\{\mathtt{ok}\}$, whenever $e_1, \ldots, e_k$ represent the set of key attribute values in a tuple of the instance $\mathcal{R}$ of relation R. On the other hand, $R\ e_1 \ldots e_k$ denotes $\{\mathsf{F}\}$ (i.e. fail), whenever $e_1, \ldots, e_k$ are not the key attribute values of any tuple of the instance $\mathcal{R}$ of relation R, although $e_1, \ldots, e_k$ must be key values defined in the instance $\mathcal{R}$. Therefore, $e_1, \ldots, e_k$ should represent combinations obtained from key attribute values defined in the instance $\mathcal{R}$. Otherwise, $R\ e_1 \ldots e_k$ denotes $\{\bot\}$. Expressions $A_i\ e_1 \ldots e_k$ denote the values of the non-key attribute A_i ($A_i \in R$) for the tuple with key attribute values $e_1, \ldots, e_k$ of the instance $\mathcal{R}$ of relation R. Moreover, $A_i\ e_1 \ldots e_k$ denotes $\{\mathsf{F}\}$ and $\{\bot\}$ in the same cases as $R\ e_1 \ldots e_k$. For instance, considering the non-key attribute `boss`, we have that `boss mary` denotes $\{$`peter`$\}$, `boss robert` denotes $\{\mathsf{F}\}$, and, finally, `boss X` denotes $\{\mathsf{F}\}$.

Definition 3.5 (Active Domain of Database Terms)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$ and a term $e \in Term_D(\mathcal{V})$, then the *active domain* e w.r.t $\mathcal{D}$ and a query $\mathcal{Q}$, denoted by $adom(e, \mathcal{D})$, is defined as follows:

(1) $adom(R\ e_1 \ldots e_k, \mathcal{D}) =_{def} \{\mathtt{ok}, \mathsf{F}, \bot\}$, for all $R \in S$;

(2) $adom(A_i\ e_1 \ldots e_k, \mathcal{D}) =_{def} \bigcup_{\{\psi \in Subst_{DC,\bot,\mathsf{F}},\ (V_1,\ldots,V_i,\ldots,V_n) \in \mathcal{R}\}} V_i\psi$, for all $A_i \in NonKey(R)$, where $\mathcal{R}$ is an instance of relation R;

(3) $adom(t, \mathcal{D}) =_{def} \{\ t \mid t \in cterms(V_i\psi)$, where $\psi \in Subst_{DC,\bot,\mathsf{F}}$ and $(V_1, \ldots, V_i, \ldots, V_n) \in \mathcal{R}\}$, with $\mathcal{R}$ schema instance of $\mathcal{S}$, if $t \in cterms(s)$,

---

*1 To simplify denotation, we write $\{c(t_1, \ldots, t_n) \mid t_i \in S_i\}$ as $c(S_1, \ldots, S_n)$ and $\{f(t_1, \ldots, t_n) \mid t_i \in S_i\}$ as $f(S_1, \ldots, S_n)$ where $S_i's$ are certain sets.

with $s \in query\_key(\mathcal{Q}, A_i)$ and $A_i \in Key(R)$; otherwise $\{\perp\}$, for all $t \in CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$;

(4) $adom(c(e_1, \ldots, e_n), \mathcal{D}) =_{def} c(adom(e_1, \mathcal{D}), \ldots, adom(e_n, \mathcal{D}))$, if $c(e_1, \ldots, e_n)$ is not a c-term, for all $c \in DC^n, n > 0$;

(5) $adom(f\ e_1 \ldots e_n, \mathcal{D}) =_{def} f^{\mathcal{D}} adom(e_1, \mathcal{D}) \ldots adom(e_n, \mathcal{D})$, for all $f \in IF^n$.

The active domain of expressions involving non-key attributes represents the complete set of values defined in the schema instance for the corresponding attribute (see case 2 in the above Definition). For instance, $\texttt{adom}(\texttt{boss mary}, \mathcal{D}) = \{\texttt{mary}, \texttt{peter}, \mathsf{F}\}$. Similarly with the query keys, whose active domain includes the complete set of c-terms defined in the schema instance for the corresponding key attribute (see case 3 in the above Definition). As can be seen in the next Definition, both sets are used for defining the set of query answers.

Definition 3.6 (Query Answers)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$ and a query $\mathcal{Q}$, then η is an *answer* of $\mathcal{Q}$ w.r.t. $\mathcal{D}$ (in symbols $(\mathcal{D}, \eta) \models_Q \mathcal{Q}$) in the following cases:

(1) $(\mathcal{D}, \eta) \models_Q e \bowtie e'$, if there exist $t \in \llbracket e \rrbracket^{\mathcal{D}} \eta$ and $t' \in \llbracket e' \rrbracket^{\mathcal{D}} \eta$, such that $t \downarrow t'$ and $t, t' \in adom(e, \mathcal{D}) \cup adom(e', \mathcal{D})$;

(2) $(\mathcal{D}, \eta) \models_Q e <> e'$, if there exist $t \in \llbracket e \rrbracket^{\mathcal{D}} \eta$ and $t' \in \llbracket e' \rrbracket^{\mathcal{D}} \eta$, such that $t \uparrow t'$ and $t, t' \in adom(e, \mathcal{D}) \cup adom(e', \mathcal{D})$;

(3) $(\mathcal{D}, \eta) \models_Q e \not\bowtie e'$ if for all $t \in \llbracket e \rrbracket^{\mathcal{D}} \eta$ and $t' \in \llbracket e' \rrbracket^{\mathcal{D}} \eta$, then $t \not\downarrow t'$ and $t, t' \in adom(e, \mathcal{D}) \cup adom(e', \mathcal{D})$;

(4) $(\mathcal{D}, \eta) \models_Q e \not<> e'$, if for all $t \in \llbracket e \rrbracket^{\mathcal{D}} \eta$ and $t' \in \llbracket e' \rrbracket^{\mathcal{D}} \eta$, then $t \not\uparrow t'$ and $t, t' \in adom(e, \mathcal{D}) \cup adom(e', \mathcal{D})$.

The active domain is used in order to restrict the answers obtained from a query against a given database instance. In fact, the use of the active domain allows us to ensure the property of domain independence in the following sense: *the query output will only depend on the input relation instances.* For instance, let us consider the query $\mathcal{Q}_0 \equiv \texttt{boss mary} \not\bowtie \texttt{Y}$. $\mathcal{Q}_0$ has as answers $\eta_1 = \{\texttt{Y/mary}\}$ and $\eta_2 = \{\texttt{Y/}\mathsf{F}\}$, since the variable Y takes values from the active domain of boss mary, defined as $\texttt{adom}(\texttt{boss mary}, \mathcal{D}) = \{\texttt{mary}, \texttt{peter}, \mathsf{F}\}$. Now, we have that boss mary denotes peter, and, from $\texttt{adom}(\texttt{boss mary}, \mathcal{D})$, we can conclude that $\texttt{peter} \not\downarrow \texttt{mary}$ and $\texttt{peter} \not\downarrow \mathsf{F}$ are satisfied. Note that we could have considered the answer $\{\texttt{Y/lecturer}\}$, since the relation $\texttt{peter} \not\downarrow \texttt{lecturer}$ is also satisfied. However, this answer causes the fact of not ensuring the property of domain independence, since, now, the query output does not depend on the input relation instances. In fact, if we consider $\{\texttt{Y/lecturer}\}$ as a valid answer, then we could consider as many answers as we wish, and thus, obtain an infinite set of answers when the database is finite; definitely, the domain independence is not satisfied. Therefore, in our case, we must restrict the answers to values considered in the active domain; this means that the value `lecturer` does not belong to $\texttt{adom}(\texttt{boss mary}, \mathcal{D})$, and thus $\{\texttt{Y/lecturer}\}$ is not a valid answer for the query $\texttt{boss mary} \not\bowtie \texttt{Y}$. In this way, equality and inequality constraints will be solved by using only values defined in the database. But, what happens to infinite database instances? How do they affect to the domain independence property? the answer is the following: given that the database instance is infinite, then we can obtain infinite answers from the input relation instance, and thus this does not mean that the domain independence property is not satisfied.

Definition 3.7 (Set of Query Answers)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ and a safe query $\mathcal{Q}$, then the *set of answers* of $\mathcal{Q}$ w.r.t. $\mathcal{D}$, denoted by $Ans(\mathcal{D}, \mathcal{Q})$, is defined as follows: $Ans(\mathcal{D}, \mathcal{Q}) =_{def} \{(X_1\eta, \ldots, X_n\eta) \mid Dom(\eta) \subseteq var(\mathcal{Q}) = \{X_1, \ldots, X_n\}, (\mathcal{D}, \eta) \models_Q \mathcal{Q}\}$.

It can be proved that each safe query is domain independent. Therefore, a query is domain independent, whenever the query satisfies, properly, two conditions: *(a) the query output over a finite relation is also a finite relation; and (b) the output relation only depends on the input relations.* The interested reader can check in [5)] the proofs of such results. The safety conditions are not only required for ensuring the domain independence property, but also to state the equivalence of the functional logic query language with the extended relational calculus and algebra, presented here.

§4 An Extended Relational Calculus

In this section, we present the proposed *extension of the relational calculus*, by showing its *syntax*, *safety conditions*, and, finally, its *semantics*.

Definition 4.1 (Atomic Formulas)
Given an extended database schema $D = (S, DC, IF)$, then *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 schema of S, the variables x_i 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 $e \in Term_{DC,IF}(\mathcal{V})$[*2] and $x \in \mathcal{V}$.

In the above Definition, (1) represents *relation predicates*, (2) *syntactic equality equations*, (3) *(strong) equality and inequality equations* with the same meaning as the corresponding relations $\downarrow$ and $\uparrow$ (see Section 2.1; concretely, Definition 2.1). Finally, (4) is an *approximation equation*, representing approximation values obtained from interpreted functions.

Definition 4.2 (Calculus Formulas)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$, then a *calculus formula* φ against $\mathcal{D}$ has the form:

$$\{x_1, \ldots, x_n \mid \phi\}$$

such that

- ϕ is a conjunction of the form $\phi_1 \wedge \ldots \wedge \phi_n$ where each ϕ_i has the form ψ or $\neg\psi$, and each ψ is an *existentially quantified conjunction of atomic formulas.*
- variables x_i are the *free variables* of ϕ, denoted by $free(\phi)$
- and finally, variables x_i occurring in all atomic formulas of the form $R(\bar{x})$ are distinct; similarly with the variables x occurring in the approximation equations $e \lhd x$.

Formulas can be built from other logical connectives, such as $\forall, \rightarrow, \vee, \leftrightarrow$, whenever they are logically equivalent to the calculus formulas defined in Definition 4.2. As an example of calculus formula, we have that the previous functional logic query $\mathcal{Q}_\mathsf{s} \equiv$ `retention_for_tax X` $\bowtie$ `salary` (`job_id peter`) can be written in the proposed relational calculus as follows:

$$\begin{array}{ll}\varphi_\mathsf{s} \equiv & \{\mathtt{x} \mid (\exists \mathtt{y_1}.\exists \mathtt{y_2}.\exists \mathtt{y_3}.\exists \mathtt{y_4}.\exists \mathtt{y_5}.\ \mathtt{person\_job}(\mathtt{y_1}, \mathtt{y_2}, \mathtt{y_3}, \mathtt{y_4}, \mathtt{y_5})\ \wedge \mathtt{y_1} = \mathtt{peter}\ \wedge \\ & \quad \exists \mathtt{z_1}.\exists \mathtt{z_2}.\exists \mathtt{z_3}.\ \mathtt{job\_information}(\mathtt{z_1}, \mathtt{z_2}, \mathtt{z_3})\ \wedge\ \mathtt{z_1} = \mathtt{y_4}\ \wedge\ \exists \mathtt{u}. \\ & \quad \mathtt{retention\_for\_tax}\ \mathtt{x} \lhd \mathtt{u}\ \wedge\ \mathtt{z_2} \Downarrow \mathtt{u})\}\end{array}$$

In this case, φ_s expresses the following meaning: to obtain the full salary,

[*2] Terms which do not include schema symbols (i.e. relation symbols and non-key attribute symbols).

that is, $\texttt{retention\_for\_tax x} \lhd \texttt{u}$ and $\exists \texttt{z}_1.\exists \texttt{z}_2.\exists \texttt{z}_3.\texttt{job\_information}(\texttt{z}_1, \texttt{z}_2, \texttt{z}_3) \wedge \texttt{z}_2 \Downarrow \texttt{u}$, for $\texttt{peter}$, that is, $\exists \texttt{y}_1. \ldots \exists \texttt{y}_5.\ \texttt{person\_job}(\texttt{y}_1, \ldots, \texttt{y}_5) \wedge \texttt{y}_1 = \texttt{peter} \wedge \texttt{z}_1 = \texttt{y}_4$.

Like the functional logic query language, and as usual in the database query formalisms, we need to ensure the property of domain independence in the proposed calculus. In fact, the domain independence property in the extended relational calculus will be preserved by ensuring *safety conditions over atomic formulas*, and *safety conditions over bounded variables.* With this aim, firstly, we need to define the following sets of variables occurring in a calculus formula φ and w.r.t. an extended database schema $D = (S, DC, IF)$:

(1) *Key variables.*
$formula\_key(\varphi) = \{x_i \mid$ *there exists* $R(x_1, \ldots, x_i, \ldots, x_n)$ *occurring in* φ *and* $1 \leq i \leq nKey(R)\}$, where R is a relation schema of S;

(2) *Non-key variables.*
$formula\_nonkey(\varphi) = \{x_j \mid$ *there exists* $R(x_1, \ldots, x_j, \ldots, x_n)$ *occurring in* φ *and* $nKey(R) + 1 \leq j \leq n\}$, where R is a relation schema of S; and

(3) *Approximation variables.*
$approx(\varphi) = \{x \mid$ *there exists* $e \lhd x$ *occurring in* $\varphi\}$.

Definition 4.3 (Safe Atomic Formulas)
An atomic formula occurring in a calculus formula φ is *safe* in the following cases:

(1) $R(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$ is *safe*, if the variables $x_1, \ldots, x_n$ are bounded in φ and for each x_i, $i \leq nKey(R)$, there exists one syntactic equality equation $x_i = t_i$ occurring in φ;

(2) $x = t$ is *safe*, if the variables occurring in t are distinct from the variables of $formula\_key(\varphi)$, and x is a variable of $formula\_key(\varphi)$;

(3) $t \Downarrow t'$ and $t \Uparrow t'$ are *safe*, if the variables occurring in t and t' are distinct from the variables of $formula\_key(\varphi)$;

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

Definition 4.4 (Range Restricted C-Terms of Calculus Formulas)
A c-term t is *range restricted* in a calculus formula φ, if either:

(1) t occurs in $formula\_key(\varphi) \cup formula\_nonkey(\varphi)$; or

(2) there exists one equation $e \Diamond_c e'$ ($\Diamond_c \in \{=, \Uparrow, \Downarrow, \lhd\}$) in φ, such that t belongs to *cterms*(e) (resp. *cterms*(e')) and every c-term of e' (resp. e) is range restricted in φ w.r.t. D.

Table 2 Examples of Calculus Formulas

Query	Calculus Formula
$\texttt{boss X} \bowtie \texttt{peter.}$	$\{\texttt{x} \mid (\exists \texttt{y}_1.\exists \texttt{y}_2.\exists \texttt{y}_3.\exists \texttt{y}_4.\exists \texttt{y}_5.\ \texttt{person\_job}(\texttt{y}_1,\texttt{y}_2,\texttt{y}_3,\texttt{y}_4,\texttt{y}_5) \wedge \texttt{y}_1 = \texttt{x} \wedge \texttt{y}_5 \Downarrow \texttt{peter})\}$
$\texttt{address (boss X)} \bowtie \texttt{Y,}$ $\texttt{job\_id X} \not\bowtie \texttt{lecturer.}$	$\{\texttt{x},\texttt{y} \mid (\exists \texttt{y}_1.\exists \texttt{y}_2.\exists \texttt{y}_3.\exists \texttt{y}_4.\exists \texttt{y}_5.\ \texttt{person\_job}(\texttt{y}_1,\texttt{y}_2,\texttt{y}_3,\texttt{y}_4,\texttt{y}_5) \wedge \texttt{y}_1 = \texttt{x} \wedge \exists \texttt{z}_1.\exists \texttt{z}_2.\exists \texttt{z}_3.\exists \texttt{z}_4.\exists \texttt{z}_5.\texttt{person\_job}(\texttt{z}_1,\texttt{z}_2,\texttt{z}_3,\texttt{z}_4,\texttt{z}_5) \ \wedge\ \texttt{z}_1 = \texttt{y}_5 \wedge \texttt{z}_3 \Downarrow \texttt{y}) \wedge (\forall \texttt{v}_4.((\exists \texttt{v}_1.\exists \texttt{v}_2.\exists \texttt{v}_3.\exists \texttt{v}_5.\ \texttt{person\_job}(\texttt{v}_1,\texttt{v}_2,\texttt{v}_3,\texttt{v}_4,\texttt{v}_5) \wedge \texttt{v}_1 = \texttt{x}) \rightarrow \neg \texttt{v}_4 \Downarrow \texttt{lecturer}))\}$
$\texttt{job\_bonus X} \not\lhd\!\rhd$ $\texttt{j\&b(associate, Y).}$	$\{\texttt{x},\texttt{y} \mid (\forall \texttt{y}_3.(\exists \texttt{y}_1.\exists \texttt{y}_2.\ \texttt{person\_boss\_job}(\texttt{y}_1,\texttt{y}_2,\texttt{y}_3) \wedge \texttt{y}_1 = \texttt{x}) \rightarrow \neg \texttt{y}_3 \Uparrow \texttt{j\&b}(\texttt{associate},\texttt{y}))\}$
$\texttt{select (list\_of\_points p(0,0) Z)} \bowtie \texttt{p(0,2).}$	$\{\texttt{z} \mid (\exists \texttt{y}_1.\exists \texttt{y}_2.\exists \texttt{y}_3.\exists \texttt{y}_4.\exists \texttt{y}_5.\ \texttt{2Dline}(\texttt{y}_1,\texttt{y}_2,\texttt{y}_3,\texttt{y}_4,\texttt{y}_5) \wedge \texttt{y}_1 = \texttt{p}(0,0) \wedge \texttt{y}_2 = \texttt{z} \ \wedge\ \exists \texttt{u}.\texttt{select}\ \texttt{y}_5 \lhd \texttt{u} \ \wedge\ \texttt{u} \Downarrow \texttt{p}(0,2))\}$

Range restricted c-terms are variables occurring in the scope of a relation predicate, or c-terms compared (by means of syntactic, strong (in)equality, and approximation equations) with variables in the scope of a relation predicate. Therefore, all of them take values from the database instance.

Definition 4.5 (Safe Formulas)
A calculus formula φ is *safe*, if the following conditions are satisfied:

(1) all c-terms and atomic formulas occurring in φ are range restricted and safe, respectively; and,

(2) the only bounded variables occurring in φ are variables of $formula\_key(\varphi) \cup formula\_nonkey(\varphi) \cup approx(\varphi)$.

For instance, the previous calculus formula

$$\varphi_s \equiv \{\texttt{x} \mid (\exists \texttt{y}_1.\exists \texttt{y}_2.\exists \texttt{y}_3.\exists \texttt{y}_4.\exists \texttt{y}_5.\ \texttt{person\_job}(\texttt{y}_1,\texttt{y}_2,\texttt{y}_3,\texttt{y}_4,\texttt{y}_5)\ \wedge \texttt{y}_1 = \texttt{peter} \wedge \exists \texttt{z}_1.\exists \texttt{z}_2.\exists \texttt{z}_3.\ \texttt{job\_information}(\texttt{z}_1,\texttt{z}_2,\texttt{z}_3)\ \wedge\ \texttt{z}_1 = \texttt{y}_4\ \wedge\ \exists \texttt{u}.\ \texttt{retention\_for\_tax x} \lhd \texttt{u}\ \wedge\ \texttt{z}_2 \Downarrow \texttt{u})\}$$

is *safe*, since:

(1) the c-term $\texttt{peter}$ is *range restricted* (by means of $\texttt{y}_1 = \texttt{peter}$), and the variables $\texttt{u},\texttt{x}$ are also *range restricted* (by means of $\texttt{z}_2 \Downarrow \texttt{u}$ and $\texttt{retention\_for\_tax x} \lhd \texttt{u}$); in addition, the atomic formulas $\texttt{person\_job}(\texttt{y}_1,\texttt{y}_2,\texttt{y}_3,\texttt{y}_4,\texttt{y}_5)$ and $\texttt{job\_information}(\texttt{z}_1,\texttt{z}_2,\texttt{z}_3)$ are safe, since the variables $\texttt{y}_1,\texttt{y}_2,\texttt{y}_3,\texttt{y}_4,\texttt{y}_5$ and $\texttt{z}_1,\texttt{z}_2,\texttt{z}_3$ are bounded, and there exist $\texttt{y}_1 = \texttt{peter}$ and $\texttt{z}_1 = \texttt{y}_4$; next, the atomic formulas $\texttt{y}_1 = \texttt{peter}$ and $\texttt{z}_1 = \texttt{y}_4$ are safe, since $\texttt{y}_1$ and $\texttt{z}_1$ are variables of $formula\_key(\varphi_s)$ and $\texttt{peter}$ and $\texttt{y}_4$ are distinct from variables of $formula\_key(\varphi_s)$; finally, $\texttt{z}_2 \Downarrow \texttt{u}$ is safe, since $\texttt{z}_2$ and $\texttt{u}$ are distinct from $formula\_key(\varphi_s)$, and $\texttt{retention\_for\_tax x} \lhd \texttt{u}$ is also safe, since $\texttt{x}$ is distinct from $formula\_key(\varphi_s)$ and u is bounded;

(2) the only bounded variables are $\texttt{y}_1,\texttt{y}_2,\texttt{y}_3,\texttt{y}_4,\texttt{y}_5,\ \texttt{z}_1,\texttt{z}_2,\texttt{z}_3$ and $\texttt{u}$; that is $formula\_key(\varphi_s) \cup formula\_nokey(\varphi_s) \cup approx(\varphi_s)$.

Table 2 shows *(safe) calculus formulas* built from the *(safe) functional logic queries* presented in table 1.

Now, we define the proposed *semantics* for our relational calculus. With this aim, firstly, we have to consider the *calculus terms*, which are defined as terms or expressions built from DC, IF and variables of $\mathcal{V}$, and they are represented by $Term_{DC,IF}\ (\mathcal{V})$. Secondly, we need to define the *denoted values* and the *active domain* of the calculus terms; in fact, the same as we did for the functional logic query language.

Definition 4.6 (Denotation of Calculus Terms)
The *denoted values* of a calculus term $e \in Term_{DC,IF}(\mathcal{V})$ in a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$ w.r.t. a substitution η, represented by $[\![e]\!]^{\mathcal{D}}\eta$, are defined as follows:

(1) $[\![X]\!]^{\mathcal{D}}\eta =_{def} \{X\eta\}$, for $X \in \mathcal{V}$;
(2) $[\![c]\!]^{\mathcal{D}}\eta =_{def} \{c\}$, for $c \in DC^0$;
(3) $[\![c(e_1, \ldots, e_n)]\!]^{\mathcal{D}}\eta =_{def} c([\![e_1]\!]^{\mathcal{D}}\eta, \ldots, [\![e_n]\!]^{\mathcal{D}}\eta)$, for all $c \in DC^n$, $n > 0$;
(4) $[\![f\ e_1\ \ldots\ e_n\ ]\!]^{\mathcal{D}}\eta =_{def} f^{\mathcal{D}}\ [\![e_1]\!]^{\mathcal{D}}\eta\ \ldots\ [\![e_n]\!]^{\mathcal{D}}\eta$, for all $f \in IF^n$.

Definition 4.7 (Active Domain of Calculus Terms)
The *active domain* of a calculus term $e \in Term_{DC,IF}(\mathcal{V})$ in a calculus formula φ w.r.t a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$, denoted by $adom(e, \mathcal{D})$, is defined as follows:

(1) $adom(x, \mathcal{D}) =_{def} \bigcup_{\{\psi \in Subst_{DC,\perp,\mathsf{F}}, (V_1,\ldots,V_i,\ldots,V_n) \in \mathcal{R}\}} V_i\psi$, if there exists an atomic formula $R(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n)$ in φ, where $\mathcal{R}$ is a schema instance of $\mathcal{S}$ for the relation R of S;
(2) $adom(x, \mathcal{D}) =_{def} adom(e, \mathcal{D})$, if $e \lhd x$ occurs in φ;
(3) $adom(x, \mathcal{D}) =_{def} \{\perp\}$, otherwise;
(4) $adom(c, \mathcal{D}) =_{def} \{\perp\}$, if $c \in DC^0$;
(5) $adom(c(e_1, \ldots, e_n), \mathcal{D}) =_{def} c(adom(e_1, \mathcal{D}), \ldots, adom(e_n, \mathcal{D}))$, if $c \in DC^n$, $n > 0$ and $c(e_1, \ldots, e_n)$ contains variables of the set $formula\_key(\varphi) \cup formula\_nonkey(\varphi) \cup approx(\varphi)$;
(6) $adom(f\ e_1 \ldots e_n, \mathcal{D}) =_{def} f^{\mathcal{D}} adom(e_1, \mathcal{D}) \ldots adom(e_n, \mathcal{D})$, if $f \in IF^n$.

In the case of key and non-key variables, the active domain contains the complete set of values defined in the database instance for the corresponding key and non-key attribute. In the case of approximation variables, the active domain contains the complete set of values defined for the interpreted function.

For example, in the atomic formula $\mathtt{person\_job}(\mathtt{x}_1, \ldots, \mathtt{x}_5)$, the active domain of $\mathtt{x}_5$ (where $\mathtt{x}_5$ is a non-key variable, representing the non-key attribute `boss`) is $adom(\mathtt{x}_5, \mathcal{D}) = \{\mathtt{mary}, \mathtt{peter}, \mathsf{F}\}$, corresponding to the set of values included in the database instance for the attribute `boss`. Like the functional logic query language, the active domain is used in order to restrict the answers obtained from a calculus formula.

For instance, the following calculus formula φ_0:

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

corresponding to the query $\mathcal{Q}_0 \equiv \mathtt{boss\ mary} \not\bowtie \mathtt{Y}$, requests people who are not `mary`'s boss. In this case, the variable `y` in the calculus formula is restricted to take values from the attribute `boss` of the relation `person_job`; that is, from the active domain of $\mathtt{x}_5$, defined as $\mathtt{adom}(\mathtt{x}_5, \mathcal{D}) = \{\mathtt{peter}, \mathtt{mary}, \mathsf{F}\}$. Therefore, the obtained answers are $\{\mathtt{y}/\mathtt{mary}\}$ and $\{\mathtt{y}/\mathsf{F}\}$; in fact, the same answers as computed for the query $\mathcal{Q}_0 \equiv \mathtt{boss\ mary} \not\bowtie \mathtt{Y}$.

Definition 4.8 (Calculus Formula Answers)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$ and a calculus formula $\{\bar{x} \mid \phi\}$, then η is an *answer* of ϕ w.r.t. $\mathcal{D}$ such that $dom(\eta) \subseteq free(\phi)$ (in symbols $(\mathcal{D}, \eta) \models_C \phi$) in the following cases:

(1) $(\mathcal{D}, \eta) \models_C R(x_1, \ldots, x_n)$, if there exists a tuple $(V_1, \ldots, V_n) \in \mathcal{R}$ ($\mathcal{R}$ is a schema instance of $\mathcal{S}$ for the relation R of S) and a substitution $\psi \in Subst_{DC,\perp,\mathsf{F}}$, such that $x_i\eta \in V_i\psi$ for every $1 \leq i \leq n$, and $V_j\psi \in CTerm_{DC,\mathsf{F}}(\mathcal{V})$ for every $1 \leq j \leq k$;

(2) $(\mathcal{D}, \eta) \models_C x = t$, if $x\eta = t\eta$, and $t\eta \in adom(x, \mathcal{D})$;

(3) $(\mathcal{D}, \eta) \models_C t \Downarrow t'$, if $t\eta \downarrow t'\eta$, and $t\eta, t'\eta \in adom(t, \mathcal{D}) \cup adom(t', \mathcal{D})$;

(4) $(\mathcal{D}, \eta) \models_C t \Uparrow t'$, if $t\eta \uparrow t'\eta$, and $t\eta, t'\eta \in adom(t, \mathcal{D}) \cup adom(t', \mathcal{D})$;

(5) $(\mathcal{D}, \eta) \models_C e \lhd x$, if $x\eta \in [\![e]\!]^{\mathcal{D}}\eta$, and $x\eta \in adom(e, \mathcal{D})$;

(6) $(\mathcal{D}, \eta) \models_C \phi_1 \wedge \phi_2$, if $\mathcal{D}$ satisfies ϕ_1 and ϕ_2 under η;

(7) $(\mathcal{D}, \eta) \models_C \exists x.\phi$, if there exists v, such that $\mathcal{D}$ satisfies ϕ under $\eta \circ \{x/v\}$;

(8) $(\mathcal{D}, \eta) \models_C \neg\phi$, if $(\mathcal{D}, \eta) \not\models_C \phi$, where:

(8.1) $(\mathcal{D}, \eta) \not\models_C R(x_1, \ldots, x_n)$, if for all tuple $(V_1, \ldots, V_n) \in \mathcal{R}$ ($\mathcal{R}$ is a schema instance of $\mathcal{S}$ for the relation R of S) and substitution $\psi \in Subst_{DC,\perp,\mathsf{F}}$, then $x_i\eta \neq V_i\psi$ for some i with $1 \leq i \leq n$, but there exist tuples $(W_1, \ldots, Z_i, \ldots, W_n) \in \mathcal{R}$ and substitutions $\psi_i \in Subst_{DC,\perp,\mathsf{F}}$, such that $x_i\eta \in Z_i\psi_i$;

(8.2) $(\mathcal{D}, \eta) \not\models_C x = t$, if $x\eta \neq t\eta$ and $t\eta \in adom(x, \mathcal{D}) \cup \{t\}$;

(8.3) $(\mathcal{D},\eta) \not\models_C t \Downarrow t'$, if $t\eta \not\downarrow t'\eta$, and $t\eta, t'\eta \in adom(t,\mathcal{D}) \cup adom(t',\mathcal{D})$;
(8.4) $(\mathcal{D},\eta) \not\models_C t \Uparrow t'$, if $t\eta \not\uparrow t'\eta$, and $t\eta, t'\eta \in adom(t,\mathcal{D}) \cup adom(t',\mathcal{D})$;
(8.5) $(\mathcal{D},\eta) \not\models_C e \lhd x$, if $x\eta \notin \llbracket e \rrbracket^{\mathcal{D}}\eta$, and $x\eta \in adom(e,\mathcal{D})$ or $\eta = id$;
(8.6) $(\mathcal{D},\eta) \not\models_C \phi_1 \wedge \phi_2$, if $(\mathcal{D},\eta) \models_C \phi_1$ or $(\mathcal{D},\eta) \models_C \phi_2$;
(8.7) $(\mathcal{D},\eta) \not\models_C \exists x.\phi$, if for all v, then $(\mathcal{D}, \eta \circ \{x/v\}) \not\models_C \phi$;
(8.8) $(\mathcal{D},\eta) \not\models_C \neg\phi$, if $(\mathcal{D},\eta) \models_C \phi$.

With regard to the use of both denotation and active domain in the Definition of calculus formula answers, for instance, in the previous formula φ_0 and w.r.t. the formula $\neg \mathtt{x}_5 \Downarrow \mathtt{y}$, we have that $\mathtt{adom}(\mathtt{x}_5,\mathcal{D}) = \{\mathtt{peter}, \mathtt{mary}, \mathsf{F}\}$ and $\mathtt{adom}(\mathtt{y},\mathcal{D}) = \{\perp\}$ (remember that $\mathtt{x}_5$ is non-key variable, representing the non-key attribute $\mathtt{boss}$). Moreover, $\eta_1 = \{\mathtt{x}_5/\mathtt{peter}, \mathtt{y}/\mathtt{mary}\}$ and $\eta_2 = \{\mathtt{x}_5/\mathtt{peter}, \mathtt{y}/\mathsf{F}\}$ satisfy $\mathtt{y}\eta_1$, $\mathtt{y}\eta_2 \in \mathtt{adom}(\mathtt{x}_5,\mathcal{D}) \cup \mathtt{adom}(\mathtt{y},\mathcal{D})$, and thus $\mathtt{x}_5\eta_1 \not\downarrow \mathtt{y}\eta_1$ and $\mathtt{x}_5\eta_2 \not\downarrow \mathtt{y}\eta_2$ are satisfied. Finally, no more values for the variable $\mathtt{y}$ can be used for satisfying the constraint $\neg \mathtt{x}_5 \Downarrow \mathtt{y}$. Therefore, as in the functional logic query language, we will consider the domain of the variables (in general, the *active domain of c-terms*) in order to obtain the answers from a given calculus formula.

With respect to the negation, we have to explicitly define the meaning of the negated formulas, since, as previously mentioned, $\neq$, $\not\downarrow$ and $\not\uparrow$ are not the "logical" negation of the corresponding relations $=$, $\downarrow$ and $\uparrow$. Similarly with the atomic formulas of the form $R(x_1, \ldots, x_n)$.

Definition 4.9 (Set of Calculus Formula Answers)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$ and a calculus formula $\varphi \equiv \{x_1, \ldots, x_n \mid \phi\}$, then the *set of answers* of φ w.r.t. $\mathcal{D}$, denoted by $Ans(\mathcal{D},\varphi)$, is defined as follows: $Ans(\mathcal{D}, \{x_1, \ldots, x_n \mid \phi\}) = \{(x_1\eta, \ldots, x_n\eta) \mid \eta \in Subst_{DC,\perp,\mathsf{F}}$ *and* $(\mathcal{D},\eta) \models_C \phi\}$.

§5 An Extended Relational Algebra

Next, we will present an *extended relational algebra*, equivalent to the previously presented query formalisms (i.e. functional logic query language and extended relational calculus). The proposed algebra is based on the use of a small set of operators which encapsulate operations over relations, and they can be composed in order to express queries. Our proposal will generalize the classical *selection* and *projection operators* of the relational algebra, by using *equality* and *inequality constraints*, *data constructors* and *destructors*, and, finally, *interpreted functions* and their *inverses*. Let us start with some preliminary definitions.

Definition 5.1 (Data Destructors and Function Inverses)

(1) Given a set of data constructors DC, we define the set of *data destructors* DD induced from DC, as the set of symbols $c.idx : T_0 \rightarrow T_{idx}$, for each $c : T_1 \times \ldots \times T_n \rightarrow T_0 \in DC$ and $1 \leq idx \leq n$, where $c.idx(t) =_{def} t_{idx}$ if t has the form $c(t_1, \ldots, t_n)$; and $\perp$ otherwise;

(2) Given a set of interpreted functions IF, we define the set of *function inverses* FI induced from IF, as the set of symbols $f.idx : T_0 \rightarrow T_{idx}$, for each $f : T_1 \times \ldots \times T_n \rightarrow T_0 \in IF$ and $1 \leq idx \leq n$, where $f.idx\ t =_{def} \{\ t_{idx} \mid t \in f^{\mathcal{D}}\ t_1 \ldots t_n\}$.

Now, given an extended database schema $D = (S, DC, IF)$, the set of *algebra terms* defined over D, denoted by $Term_{DC \cup DD, IF \cup FI}(Att(D))$, are terms built from the attributes defined in D, data constructors of DC, data destructors of DD induced from DC, interpreted function symbols of IF, and, finally, function inverses FI induced from IF.

Definition 5.2 (Denotation of Algebra Terms)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$, then the *denotation* of an algebra term $e \in Term_{DC \cup DD, IF \cup FI}(Att(D))$ in a tuple $V = (V_1, \ldots, V_n) \in \mathcal{R}$, where $\mathcal{R}$ is a schema instance of $\mathcal{S}$, denoted by $\| e \|_V^{\mathcal{D}}$, is defined as follows:

(1) $\| A_i \|_V^{\mathcal{D}} =_{def} \bigcup_{\{\eta \in Subst_{DC,\mathsf{F}}\}} V_i\eta$, for all $A_i \in Key(D)$;

(2) $\| A_i \|_V^{\mathcal{D}} =_{def} \bigcup_{\{\eta \in Subst_{DC,\perp,\mathsf{F}}\}} V_i\eta$, for all $A_i \in NonKey(D)$;

(3) $\| c \|_V^{\mathcal{D}} =_{def} \{c\}$, for all $c \in DC^0$;

(4) $\| c(e_1, \ \ldots \ , e_n) \|_V^{\mathcal{D}} =_{def} c(\| e_1 \|_V^{\mathcal{D}}, \ \ldots \ , \| e_n \|_V^{\mathcal{D}})$, for all $c \in DC^n$, $n > 0$;

(5) $\| c.idx(e) \|_V^{\mathcal{D}} =_{def} c.idx(\| e \|_V^{\mathcal{D}})$, for all $c.idx \in DD$ induced from DC;

(6) $\| f\ e_1\ \ldots\ e_n\ \|_V^{\mathcal{D}} =_{def} f^{\mathcal{D}}\ \| e_1 \|_V^{\mathcal{D}} \ldots \| e_n \|_V^{\mathcal{D}}$, for all $f \in IF^n$;

(7) $\| f.idx\ e \|_V^{\mathcal{D}} =_{def} f.idx\ \| e \|_V^{\mathcal{D}}$, for all $f.idx \in FI$ induced from IF.

For instance, with respect to the relation schema `job_information`, we can build the algebra term `j&b`(`job_name`, `bonus`), representing the values of the attributes `job_name` and `bonus` encapsulated by the data constructor `j&b`. In addition, by considering the view `person_boss_job`, we can build the algebra term `j&b.1`(`job_bonus`), which 'destructs' the values of the attribute `job_bonus`; for example, w.r.t. the value `j&b`(`professor`, `1500`) in the attribute `job_bonus`, we have that `j&b.1`(`job_bonus`) denotes `professor`. As an example of lists, let us consider the relation schema `2Dline`. Then, [_|_].`1`(`list_of_points`) and

[_|_].2(`list_of_points`) represent the first point and the rest of points of the attribute `list_of_points`, respectively.

Furthermore, given an extended database schema $D = (S, DC, IF)$, we also need to define the so-called *projection terms* as follows:

(a) terms of the form $\overset{=}{p}$ and $\overset{\neq}{p}$, where p is a term built from a key attribute (i.e. attribute of $Key(D)$), data destructors of DD induced from DC, and inverses of FI induced from IF; or

(b) terms of the form $\overset{\bowtie}{p}$, $\overset{<\!\!>}{p}$, $\overset{\not\bowtie}{p}$ and $\overset{\not<\!\!>}{p}$, where p is an algebra term (i.e. a term of the set $Term_{DC \cup DD, IF \cup FI}(Att(D))$).

The projection terms $\overset{=}{p}$, $\overset{\bowtie}{p}$, $\overset{<\!\!>}{p}$ represent the set of values (obtained from a database instance), which are syntactically equal, strongly equal, and strongly different to p, respectively. Analogously, $\overset{\neq}{p}$ represents the set of values (obtained from a database instance), which are not syntactically equal to p; finally, $\overset{\not\bowtie}{p}$ and $\overset{\not<\!\!>}{p}$ represent the set of values (obtained from a database instance), which are weakly different and weakly equal to p, respectively.

For instance, $\overset{\bowtie}{\texttt{job\_bonus}}$ w.r.t the instance of "view" `person_boss_job` requests the values which are strongly equal to the values of the attribute `job_bonus`; that is, totally defined values in this attribute, obtaining the value {`j&b(professor, 1500)`}. Now, if we consider $\overset{\not\bowtie}{\texttt{job\_bonus}}$, then we are demanding those values which are strongly different to the values of the attribute `job_bonus`; that is, {`j&b(lecturer, F)`, `j&b(associate, F)`}. Finally, `j&b.2` ($\overset{\not\bowtie}{\texttt{job\_bonus}}$) represents {`F`}.

Definition 5.3 (Denotation of Projection Terms)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$, then the *denotation of a projection term* $\overset{\Diamond_a}{p}$ in a tuple $V = (V_1, \ldots, V_n) \in \mathcal{R}$, where $\mathcal{R}$ is a schema instance of $\mathcal{S}$ and $\Diamond_a \in \{=, \bowtie, <\!\!>, \neq, \not\bowtie, \not<\!\!>\}$, is represented by $\| \overset{\Diamond_a}{p} \|^{\mathcal{D}}_V$ and defined as follows:

(1) $\| \overset{=}{p} \|^{\mathcal{D}}_V =_{def} \|p\|^{\mathcal{D}}_V$;

(2) $\| \overset{\bowtie}{p} \|^{\mathcal{D}}_V =_{def} \{t' |\ t' \in \|p\|^{\mathcal{D}}_{V'}, V' \in \mathcal{R}$ and there exists $t \in \|p\|^{\mathcal{D}}_V$ and $t \downarrow t'\}$;

(3) $\| \overset{<\!\!>}{p} \|^{\mathcal{D}}_V =_{def} \{t' |\ t' \in \|p\|^{\mathcal{D}}_{V'}, V' \in \mathcal{R}$ and there exists $t \in \|p\|^{\mathcal{D}}_V$ and $t \uparrow t'\}$;

(4) $\| \overset{\neq}{p} \|_V^{\mathcal{D}} =_{def} \{t' |\ t' \in \|p\|_{V'}^{\mathcal{D}}, V' \in \mathcal{R}$ and for all $t \in \|p\|_V^{\mathcal{D}}$ then $t \neq t'\}$;

(5) $\| \overset{\not\bowtie}{p} \|_V^{\mathcal{D}} =_{def} \{t' |\ t' \in \|p\|_{V'}^{\mathcal{D}}, V' \in \mathcal{R}$ and for all $t \in \|p\|_V^{\mathcal{D}}$ then $t \not\downarrow t'\}$;

(6) $\| \overset{\not\diamond}{p} \|_V^{\mathcal{D}} =_{def} \{t' |\ t' \in \|p\|_{V'}^{\mathcal{D}}, V' \in \mathcal{R}$ and for all $t \in \|p\|_V^{\mathcal{D}}$ then $t \not\uparrow t'\}$;

(7) $\| c.\overset{\diamond_a}{idx}(p) \|_V^{\mathcal{D}} =_{def} c.idx(\| \overset{\diamond_a}{p} \|_V^{\mathcal{D}})$;

(8) $\| f.\overset{\diamond_a}{idx}\ p \|_V^{\mathcal{D}} =_{def} f.idx \| \overset{\diamond_a}{p} \|_V^{\mathcal{D}}$.

Definition 5.4 (Algebra Formulas)
Given an extended database schema $D = (S, DC, IF)$, then *algebra formulas* are defined as expressions of the form:

(1) $e = e'$ and $e \neq e'$, where e is a term built from a key attribute (i.e. attribute of $Key(D)$), data destructors of DD induced from DC, and inverses of FI induced from IF, and e' is an algebra term; or
(2) $e \bowtie e'$, $e <> e'$, $e \not\bowtie e'$ and $e \not<> e'$, where e and e' are algebra terms.

Definition 5.5 (Algebra Formula Answers)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$ and an algebra formula F, then a tuple $V \in \mathcal{R}$ (where $\mathcal{R}$ is a schema instance of $\mathcal{S}$) is an *answer* of F w.r.t. $\mathcal{D}$ (in symbols $V \models_A F$) in the following cases:

(1) $V \models_A e = e'$, if there exist $t \in \|e\|_V^{\mathcal{D}}$ and $t' \in \|e'\|_V^{\mathcal{D}}$, such that $t = t'$;
(2) $V \models_A e \bowtie e'$, if there exist $t \in \|e\|_V^{\mathcal{D}}$ and $t' \in \|e'\|_V^{\mathcal{D}}$, such that $t \downarrow t'$;
(3) $V \models_A e <> e'$, if there exist $t \in \|e\|_V^{\mathcal{D}}$ and $t' \in \|e'\|_V^{\mathcal{D}}$, such that $t \uparrow t'$;
(4) $V \models_A e \neq e'$, if for all $t \in \|e\|_V^{\mathcal{D}}$ and $t' \in \|e'\|_V^{\mathcal{D}}$, then $t \neq t'$ holds
(5) $V \models_A e \not\bowtie e'$, if for all $t \in \|e\|_V^{\mathcal{D}}$ and $t' \in \|e'\|_V^{\mathcal{D}}$, then $t \not\downarrow t'$ holds;
(6) $V \models_A e \not<> e'$, if for all $t \in \|e\|_V^{\mathcal{D}}$ and $t' \in \|e'\|_V^{\mathcal{D}}$, then $t \not\uparrow t'$ holds.

Definition 5.6 (Algebra Operators)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$, and let $\mathcal{R}, \mathcal{Q}$ be two schema instances of $\mathcal{S}$ for the relations R and Q of S, then the *algebra operators* are defined as follows:

Selection (σ):
$\mathcal{R}' = \sigma_{F_1,\ldots,F_n}(\mathcal{R}) =_{def} \{V \in \mathcal{R} \mid V \models_A F_1, \ldots, F_n\}$
It denotes the tuple selection over $\mathcal{R}$ according to the algebra formulas $F_1, \ldots, F_n$; where:

- $Key(R') = Key(R)$;
- $NonKey(R') = NonKey(R)$;

Projection (π):

$\mathcal{R}' = \pi_{\overset{\Diamond_k}{p_1},\ldots,\overset{\Diamond_k}{p_k},\overset{\Diamond_{nk}}{p_{k+1}},\ldots,\overset{\Diamond_{nk}}{p_n}}(\mathcal{R})$

where $\Diamond_k \in \{=, \neq\}$ and $\Diamond_{nk} \in \{\bowtie, \Diamond, \not\bowtie, \not\Diamond\}$. Now, we need to consider two cases; that is, a positive and a negative case:

(a) Positive Case:

$\mathcal{R}' = \pi_{\overset{=}{p_1},\ldots,\overset{=}{p_k},\overset{\Diamond_{nk}}{p_{k+1}},\ldots,\overset{\Diamond_{nk}}{p_n}}(\mathcal{R}) =_{def}$

$\{(W_1, \ldots, W_k, \llbracket \overset{\Diamond_{nk}}{p_{k+1}} \rrbracket^{\mathcal{D}}_V, \ldots, \llbracket \overset{\Diamond_{nk}}{p_n} \rrbracket^{\mathcal{D}}_V) \mid W_i \in \llbracket \overset{=}{p_i} \rrbracket^{\mathcal{D}}_V, V \in \mathcal{R}\}$

(b) Negative Case:

$\mathcal{R}' = \pi_{\overset{\neq}{p_1},\ldots,\overset{\neq}{p_k},\overset{\Diamond_{nk}}{p_{k+1}},\ldots,\overset{\Diamond_{nk}}{p_n}}(\mathcal{R}) =_{def}$

$\{W = (W_1, \ldots, W_k, \llbracket \overset{\Diamond_{nk}}{p_{k+1}} \rrbracket^{\mathcal{D}}_V, \ldots, \llbracket \overset{\Diamond_{nk}}{p_n} \rrbracket^{\mathcal{D}}_V) \mid W_i \in \llbracket \overset{=}{p_i} \rrbracket^{\mathcal{D}}_V, V \in \mathcal{R}, W \notin \mathcal{R}\} \cup$
$\{W = (W_1, \ldots, W_k, \bigcup_{V \in \mathcal{R}} \llbracket \overset{=}{p_{k+1}} \rrbracket^{\mathcal{D}}_V, \ldots, \bigcup_{V \in \mathcal{R}} \llbracket \overset{=}{p_n} \rrbracket^{\mathcal{D}}_V) \mid W_i \in \llbracket \overset{\neq}{p_i} \rrbracket^{\mathcal{D}}_V, V \in \mathcal{R}, W \notin \mathcal{R}\}$

It denotes the projection over $\mathcal{R}$ according to the projection terms $\overset{\Diamond_k}{p_1}, \ldots, \overset{\Diamond_k}{p_k}$, $\overset{\Diamond_{nk}}{p_{k+1}}, \ldots, \overset{\Diamond_{nk}}{p_n}$, where $\mathcal{R}'$ is the instance of the relation schema R' defined as follows:

- $Key(R') = \{\overset{\Diamond_k}{p_1}, \ldots, \overset{\Diamond_k}{p_k}\}$;
- $NonKey(R') = \{\overset{\Diamond_{nk}}{p_{k+1}}, \ldots, \overset{\Diamond_{nk}}{p_n}\}$.

Cross Product ($\times$):

$\mathcal{P} = \mathcal{R} \times \mathcal{Q} =_{def}$
$\{(V_1, \ldots, V_k, W_1, \ldots, W_{k'}, V_{k+1}, \ldots, V_n, W_{k'+1}, \ldots, W_m) \mid V = (V_1, \ldots, V_n) \in \mathcal{R}, W = (W_1, \ldots, W_m) \in \mathcal{Q}\}$

It denotes the cross product of the two schema instances $\mathcal{R}$ and $\mathcal{Q}$; where $\mathcal{P}$ is the instance of the relation schema P defined as follows:

- $Key(P) = Key(R) \cup Key(Q)$;
- $NonKey(P) = NonKey(R) \cup NonKey(Q)$;
- $k = nKey(R)$, $n = nAtt(R)$, $k' = nKey(Q)$ and $m = nAtt(Q)$.

Join ($\bowtie$):

$\mathcal{R} \bowtie_{F_1,\ldots,F_n} \mathcal{Q} =_{def} \sigma_{F_1,\ldots,F_n}(\mathcal{R} \times \mathcal{Q})$

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 selection operator.

Renaming (δ_ρ):

$\mathcal{R}' = \delta_\rho(\mathcal{R})$
It denotes an attribute renaming of the relation R of the form $A_1A_2 \ldots A_m \rightarrow B_1B_2 \ldots B_m$; 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 of relation R' is $R'(\rho(A_1), \ldots, \rho(A_n))$, whenever relation R has as schema $R(A_1, \ldots, A_n)$.

The most relevant operator is the projection one, since π takes: (a) $k \geq 0$ projection terms, considered as the key values in the instance of the new output relation; and (b) $n - k$ projection terms (where n is the number of projection terms in the operator), considered as the non-key values in the instance of the new output relation. In addition, we need to consider two cases:
(Positive Case) - π projects tuples occurring in $\mathcal{R}$; for instance, the query $\mathcal{Q}_s \equiv$ `retention_for_tax X` $\bowtie$ `salary` (`job_id peter`) can be written as the following algebra expression $\mathcal{A}_s$:

$$\mathcal{A}_s \equiv \pi_{\overset{\bowtie}{\texttt{retention\_for\_tax.1(salary)}}}(\sigma_{\texttt{name=peter}}(\texttt{job\_information} \;\overline{\bowtie}_{\texttt{job\_name=job\_id}}\; \texttt{person\_job}))$$

In this case, $\mathcal{A}_s$ 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 $\pi_{\overset{\bowtie}{\texttt{retention\_for\_tax.1(salary)}}}$.
(Negative Case) - π projects tuples which are not in $\mathcal{R}$, but they are obtained from combinations of key and non-key values occurring in $\mathcal{R}$; for instance, the query $\mathcal{Q} \equiv$ `next X Y` $\not\bowtie$ `Z` w.r.t. the relation schema `2Dline` can be written by the following algebra expression:

$$\mathcal{A} \equiv \pi_{\overset{\neq}{\texttt{origin}},\overset{\neq}{\texttt{dir}},\overset{\not\bowtie}{\texttt{next}}}(\texttt{2Dline})$$

In this case, $\mathcal{A}$ expresses the following meaning: to request those lines (i.e. `2Dline`) which are not in the database (i.e. $\pi_{\overset{\neq}{\texttt{origin}},\overset{\neq}{\texttt{dir}}}$), or points which are not the next of any line origin occurring in the database (i.e. $\pi_{\overset{\not\bowtie}{\texttt{next}}}$).

Finally, as other example of algebra expression, we can consider the query $\mathcal{Q}_0 \equiv$ `boss mary` $\not\bowtie$ `Y` which can be written by the following algebra expression $\mathcal{A}_0$:

Table 3 Examples of Algebra Expressions

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

ρ_1 : `name age ... name age ...` $\rightarrow$ `name age ... name' age' ...`

$$\mathcal{A}_0 \equiv \pi_{\overset{\not\bowtie}{\mathtt{boss}}}(\sigma_{\mathtt{name=mary}}(\mathtt{person\_job}))$$

representing, like query language (i.e. $\mathcal{Q}_0$) and relational calculus (i.e. φ_0), the tuples (F) and (`mary`). Remark that projection terms play the role of attribute names in the output relation. For instance, the output relation of the algebra expression $\mathcal{A}_0$ has a unique non-key attribute, whose attribute name is $\overset{\not\bowtie}{\mathtt{boss}}$. Next, we formally define the so-called *algebra expressions.*

Definition 5.7 (Algebra Expressions)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$, then *algebra expressions* Ψ are defined as expressions built from a composition of algebra operators over a sub-sequence of schema instances $\mathcal{R}_i, \ldots, \mathcal{R}_j$ of $\mathcal{S}$ for the relation names $R_i, \ldots, R_j$ of S, satisfying the following conditions:

(1) Ψ must be *closed w.r.t. key values*; that is, $Key(\Psi) = \cup_{R \in Rel(\Psi)} Key(R)$, where $Key(\Psi)$ and $Rel(\Psi)$ represent the key attribute names and relation names corresponding to schema instances occurring in Ψ, respectively;

(2) Ψ must be *closed w.r.t. data destructors and function inverses*; that is, whenever $\pi_{\overset{\Diamond_a}{c.index(e)}}$ or $\sigma_{c.index(e) \Diamond_a e^*}$ (resp. $\pi_{\overset{\Diamond_a}{f.index(e)}}$ or $\sigma_{f.index(e) \Diamond_a e^*}$) occurs in Ψ, then $\pi_{\overset{\Diamond_a}{c.i(e)}}$ or $\sigma_{c.i(e) \Diamond_a e^*}$ (resp. $\pi_{\overset{\Diamond_a}{f.i(e)}}$ or $\sigma_{f.i(e) \Diamond_a e^*}$) must occur in Ψ, for every $1 \leq i \leq n$ with $c \in DC^n$ (resp. $f \in IF^n$).

Table 3 shows the *algebra expressions* built from the *safe queries* presented in table 1.

§6 Query Formalism Equivalence

In this section, we will state the equivalence between all the query formalisms, that is the functional logic query language, the extended relational

Table 4 Calculus and Query Transformation Rules

$$\textbf{(1)}\quad \frac{\phi \wedge \exists \bar{z}.\psi \oplus e_1 \bowtie e_2, \mathcal{Q}}{\phi \wedge \exists \bar{z}.\exists x.\exists y.\psi \wedge e_1 \lhd x \;\wedge\; e_2 \lhd y \;\wedge\; x \Downarrow y \oplus \mathcal{Q}}$$

$$\textbf{(2)}\quad \frac{\phi \wedge \neg\exists \bar{z}.\psi \oplus e_1 \not\bowtie e_2, \mathcal{Q}}{\phi \wedge \neg\exists \bar{z}.\exists x.\exists y.\psi \wedge e_1 \lhd x \;\wedge\; e_2 \lhd y \;\wedge\; x \Downarrow y \oplus \mathcal{Q}}$$

$$\textbf{(3)}\quad \frac{\phi \wedge \exists \bar{z}.\psi \oplus e_1 \diamond e_2, \mathcal{Q}}{\phi \wedge \exists \bar{z}.\exists x.\exists y.\psi \wedge e_1 \lhd x \;\wedge\; e_2 \lhd y \;\wedge\; x \Uparrow y \oplus \mathcal{Q}}$$

$$\textbf{(4)}\quad \frac{\phi \wedge \neg\exists \bar{z}.\psi \oplus e_1 \not\diamond e_2, \mathcal{Q}}{\phi \wedge \neg\exists \bar{z}.\exists x.\exists y.\psi \wedge e_1 \lhd x \;\wedge\; e_2 \lhd y \;\wedge\; x \Uparrow y \oplus \mathcal{Q}}$$

$$\textbf{(5)}\quad \frac{\phi \wedge (\neg)\exists \bar{z}.\exists x.\psi \wedge R\ e_1 \ldots e_k \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1 \ldots \exists y_n.\psi \wedge R(y_1, \ldots, y_k, \ldots, y_n) \wedge e_1 \lhd y_1 \wedge \ldots \wedge e_k \lhd y_k\{x/\mathtt{ok}\} \oplus \mathcal{Q}}$$

$\star$ $\mathtt{R} \in \mathtt{S}$, *where* $\mathtt{D} = (\mathtt{S}, \mathtt{DC}, \mathtt{IF})$ *is an extended database schema.*

$$\textbf{(6)}\quad \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge A_i\ e_1 \ldots e_k \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1 \ldots \exists y_n.\psi \wedge R(y_1, \ldots, y_k, \ldots, y_i, \ldots, y_n) \wedge e_1 \lhd y_1 \wedge \ldots \wedge e_k \lhd y_k \wedge y_i \lhd x \oplus \mathcal{Q}}$$

$\star$ $\mathtt{A_i} \in \mathtt{NonKey(R)}$ *and* $\mathtt{R} \in \mathtt{S}$, *where* $\mathtt{D} = (\mathtt{S}, \mathtt{DC}, \mathtt{IF})$ *is an extended database schema.*

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

$\star$ $\mathtt{f\ e_1 \ldots e_n} \notin \mathtt{Term}_{\mathtt{DC,IF}}(\mathcal{V})$, *where* $\mathtt{D} = (\mathtt{S}, \mathtt{DC}, \mathtt{IF})$ *is an extended database schema.*

$$\textbf{(8)}\quad \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge c(e_1, \ldots, e_n) \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1 \ldots y_n.\psi \wedge c(y_1, \ldots, y_n) \lhd x \wedge e_1 \lhd y_1 \wedge \ldots \wedge e_n \lhd y_n \oplus \mathcal{Q}}$$

$\star$ $\mathtt{c}(\mathtt{e_1 \ldots e_n}) \notin \mathtt{Term}_{\mathtt{DC,IF}}(\mathcal{V})$, *where* $\mathtt{D} = (\mathtt{S}, \mathtt{DC}, \mathtt{IF})$ *is an extended database schema.*

$$\textbf{(9)}\quad \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge t \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge x = t \oplus \mathcal{Q}}$$

$\star$ $\mathtt{x} \in \mathtt{formula\_key}(\phi \wedge (\neg)\exists \bar{z}.\psi \wedge \mathtt{t} \lhd \mathtt{x})$

$$\textbf{(10)}\quad \frac{\phi \wedge (\neg)\exists \bar{z}.\exists x.\psi \wedge t \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\psi\{x/t\} \oplus \mathcal{Q}}$$

$\star$ $\mathtt{x} \notin \mathtt{formula\_key}(\phi \wedge (\neg)\exists \bar{z}.\exists x.\psi \wedge \mathtt{t} \lhd \mathtt{x})$

calculus and the extended relational algebra.

6.1 Query and Calculus Equivalence

Firstly, we will show the equivalence between the functional logic query language and the extended relational calculus. With this aim, we will define a set of *transformation rules*, which allow us to transform a *safe query* into a *safe calculus formula* and viceversa. In order to prove the equivalence result, we will use two additional results (concretely, Lemma 10.1 (Answers in Calculus and Query Transformation Rules) and Lemma 10.2 (Safety in Calculus and Query Transformation Rules)) shown in Appendix subsection Query and Calculus Equivalence, since there are quite a lot technical subtle in these results.

Definition 6.1 (Calculus and Query Transformation Rules)
Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of an extended database schema $D = (S, DC, IF)$, we define a set of *calculus and query transformation rules* of the form:

$$\frac{\phi \oplus \ \mathcal{Q}}{\phi^* \oplus \mathcal{Q}^*}$$

The calculus and query transformation rules are shown in Table 4, and they transform pairs $(\phi \oplus \mathcal{Q})$ into pairs $(\phi^* \oplus \mathcal{Q}^*)$, where ϕ and ϕ^* are *calculus formulas*, and $\mathcal{Q}$ and $\mathcal{Q}^*$ are *queries*. Note that the rules can be applied in a top-down and a bottom-up way. In fact, in order to transform a *safe query* $\mathcal{Q}$ into a *safe calculus formula* ϕ, we start from[*3] $(\emptyset \oplus \mathcal{Q})$ and apply the transformation rules in a top-down way up to obtain the safe calculus formula ϕ. Analogously, to transform a *safe calculus formula* ϕ into a *safe query* $\mathcal{Q}$, we start from $(\phi \oplus \emptyset)$ and apply the transformation rules in a bottom-up way up to obtain the safe query $\mathcal{Q}$.

Theorem 6.1 (Query and Calculus Equivalence)
Let $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ be a database instance of an extended database schema $D = (S, DC, IF)$, then:

(1) given a safe query $\mathcal{Q}^\star$ against $\mathcal{D}$, there exists a safe calculus formula $\phi_{\mathcal{Q}^\star}$ such that $Ans(\mathcal{D}, \mathcal{Q}^\star) = Ans(\mathcal{D}, \phi_{\mathcal{Q}^\star})$

(2) given a safe calculus formula $\phi^\star$ against $\mathcal{D}$, there exists a safe query $\mathcal{Q}_{\phi^\star}$ such that $Ans(\mathcal{D}, \phi^\star) = Ans(\mathcal{D}, \mathcal{Q}_{\phi^\star})$

Proof
In order to prove the theorem, we should prove the following:

(1) if $(\emptyset \oplus \mathcal{Q}^\star) \rightarrow^n (\phi_{\mathcal{Q}^\star} \oplus \emptyset)$ (i.e. starting from query $\mathcal{Q}^\star$ and applying the calculus and query transformation rules n times, the calculus formula $\phi_{\mathcal{Q}^\star}$ is obtained), then:

(1.1) $\bar{x}\eta \in Ans(\mathcal{D}, \mathcal{Q}^\star)$, iff there exists a substitution η^* such that $\bar{x}\eta^* \in Ans(\mathcal{D},\ \phi_{\mathcal{Q}^\star})$, where $\eta^* = \eta|_{var(\mathcal{Q}^\star)}$

(1.2) $\mathcal{Q}^\star$ is safe, iff $\phi_{\mathcal{Q}^\star}$ is safe

(2) if $(\phi^\star \oplus \emptyset) \rightarrow^n (\emptyset \oplus \mathcal{Q}_{\phi^\star})$ (i.e. starting from calculus formula $\phi^\star$ and applying the calculus and query transformation rules n times, the query $\mathcal{Q}_{\phi^\star}$ is obtained) then:

(2.1) $\bar{x}\eta \in Ans(\mathcal{D}, \phi^\star)$, iff there exists a substitution η^* such that $\bar{x}\eta^* \in Ans(\mathcal{D}, \mathcal{Q}_{\phi^\star})$, where $\eta^* = \eta|_{free(\phi)}$

(2.2) $\phi^\star$ is safe, iff $\mathcal{Q}_{\phi^\star}$ is safe

Let us start proving (1), that is $(\emptyset \oplus \mathcal{Q}^\star) \rightarrow^n (\phi_{\mathcal{Q}^\star} \oplus \emptyset)$:
For each transformation step, applying the rule,

[*3] $\emptyset$ denotes an empty sequence

$$\frac{\phi \oplus \mathcal{Q}}{\phi^* \oplus \mathcal{Q}^*}$$

(1.1) η is a substitution such that $\bar{x}\eta \in Ans(\mathcal{D}, \phi) \cap Ans(\mathcal{D}, \mathcal{Q})$ with $\bar{x} = var(\mathcal{Q}) \cup free(\phi)$ iff, by Lemma 10.1, there exists a substitution $\eta^* = \eta|_{var(\mathcal{Q}) \cup free(\phi)}$, such that $\bar{x}\eta^* \in Ans(\mathcal{D}, \phi^*) \cap Ans(\mathcal{D}, \mathcal{Q}^*)$ with $\bar{x} = var(\mathcal{Q}^*) \cup free(\phi^*)$. Therefore, starting from $\mathcal{Q}^\star$ and iterating transformation steps, then there exists a substitution η such that $\bar{x}\eta \in Ans(\mathcal{D}, \mathcal{Q}^\star)$ with $\bar{x} = var(\mathcal{Q}^\star)$, iff, by Lemma 10.1, there exists a substitution η^* such that $\bar{x}\eta^* \in Ans(\mathcal{D}, \phi_{\mathcal{Q}^\star})$ with $\bar{x} = free(\phi_{\mathcal{Q}^\star})$;

(1.2) the calculus formula ϕ and query $\mathcal{Q}$ are safe iff, by Lemma 10.2, the calculus formula ϕ^* and query $\mathcal{Q}^*$ are safe. Now, if $\mathcal{Q}^\star$ is safe w.r.t. Lemma 10.2, then, in particular, $\mathcal{Q}^\star$ is safe w.r.t. Definition 3.3 (*Safe Queries*). Therefore, by iterating transformation steps and by Lemma 10.2, $\phi_{\mathcal{Q}^\star}$ is safe w.r.t. Lemma 10.2 and, in particular, w.r.t. the Definition 4.5 (*Safe Calculus Formulas*).

Analogously, we can prove (2), that is, $(\phi^\star \oplus \emptyset) \to^n (\emptyset \oplus \mathcal{Q}_{\phi^\star})$. ■

6.2 Calculus and Algebra Equivalence

In this subsection, we will show the equivalence among the proposed extended relational calculus and algebra. As previously, we will define a set of *transformation rules*, which allow us to transform a *safe calculus formula* into a *closed algebra expression* and viceversa. In addition, in order to prove the equivalence result, we will use two additional results (concretely, Lemma 10.3 (Answers in Calculus and Algebra Transformation Rules) and Lemma 10.4 (Safety in Calculus and Algebra Transformation Rules)) shown in Appendix subsection Calculus and Algebra Equivalence.

Definition 6.2 (Calculus and Algebra Transformation Rules) Given a database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ where $\mathcal{S}$ is a sequence of schema instances $\mathcal{R}_1, \ldots, \mathcal{R}_n$. In addition, given an extended database schema $D = (S, DC, IF)$, where S is a sequence of relation names $R_1, \ldots, R_n$ and each $\mathcal{R}_i$ is a schema instance of R_i $(1 \leq i \leq n)$, then we define a set of *calculus and algebra transformation rules* of the form:

$$\frac{\phi \oplus \Delta \oplus (Rel|Select|Proj|Ren)}{\phi^* \oplus \Delta^* \oplus (Rel^*|Select^*|Proj^*|Ren^*)}$$

The calculus and algebra transformation rules are shown in Table 5, and they transform triples $(\phi \oplus \Delta \oplus (Rel|Select|Proj|Ren))$ into triples $(\phi^* \oplus \Delta^* \oplus (Rel^*|Select^*|\ Proj^*|Ren^*))$, where:

- ϕ and ϕ^* are calculus formulas;
- Δ and Δ^* represent substitutions of key and non-key variables by attribute names, and variables by terms of $Term_{DC\cup DD,IF\cup FI}(Att(D))$;
- Rel and Rel^* are sequences of schema instances of database instance $\mathcal{D}$;
- $Select$ and $Select^*$ are sequences of selection formulas;
- $Proj$ and $Proj^*$ are sequences of projection terms;
- Ren and Ren^* are sequences of renamings.

Finally, Ψ and Ψ^* are algebra expressions, defined as follows:

- $\Psi \equiv \pi_{Proj}(\sigma_{Select}(\delta_{Ren}(\mathcal{R}_i\times \ldots \times\mathcal{R}_j)))$, whenever Rel is the sequence $\mathcal{R}_i,\ldots,\mathcal{R}_j$ and each $\mathcal{R}_p$ is a schema instance of $\mathcal{S}$ with $i \leq p \leq j$;
- $\Psi^* \equiv \pi_{Proj^*}(\sigma_{Select^*}(\delta_{Ren^*}(\mathcal{R}^*_k\times \ldots \times\mathcal{R}^*_l)))$, whenever Rel^* is the sequence $\mathcal{R}^*_k,\ldots,\mathcal{R}^*_l$ and each $\mathcal{R}^*_p$ is a schema instance of $\mathcal{S}$ with $k \leq p \leq l$.

As the previous transformation rules, these rules can be also applied in a top-down and bottom-up way. In order to transform a *safe calculus formula* ϕ into a *closed algebra expression* Ψ, we start from $(\phi \oplus id \oplus \emptyset)$ and apply the transformation rules in a top-down way up to obtain the closed algebra expression Ψ. Analogously, in order to transform a *closed algebra expression* $\pi_{Proj}(\sigma_{Select}(\delta_{Ren}(\mathcal{R}_i\times \ldots \times\mathcal{R}_j)))$ into a *safe calculus formula* ϕ, we start from $(\emptyset \oplus id \oplus (\mathcal{R}_i,\ldots,\mathcal{R}_j|Select|Proj|Ren))$ and apply the transformation rules in a bottom-up way up to obtain the safe calculus formula ϕ.

Theorem 6.2 (Calculus and Algebra Equivalence)
Let $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ be a database instance, where $\mathcal{S}$ is a sequence of schema instances $\mathcal{R}_1,\ldots,\mathcal{R}_n$. In addition, given an extended database schema $D = (S, DC, IF)$, where S is a sequence of relation names $R_1,\ldots,R_n$ such that $\mathcal{R}_i$ is a schema instance of R_i $(1 \leq i \leq n)$, then:

(1) given a safe calculus formula $\phi^\star$, then there exists a closed algebra expression $\Psi_{\phi^\star}(\mathcal{R}_i,\ldots,\mathcal{R}_j)$ ($\mathcal{R}_p$ is a schema instance of $\mathcal{S}$ with $i \leq p \leq j$), such that $Ans(\mathcal{D}, \phi^\star) = \Psi_{\phi^\star}(\mathcal{R}_i,\ldots,\mathcal{R}_j)$

(2) given a closed algebra expression $\Psi^\star(\mathcal{R}_i,\ldots,\mathcal{R}_j)$ ($\mathcal{R}_p$ is a schema instance of $\mathcal{S}$ with $i \leq p \leq j$), then there exists a safe calculus formula $\phi_{\Psi^\star}$ such that $\Psi^\star(\mathcal{R}_i,\ldots,\mathcal{R}_j) = Ans(\mathcal{D}, \phi_{\Psi^\star})$

Proof

Table 5 Calculus and Algebra Transformation Rules

(1) $$\frac{\phi \wedge (\neg)\exists \bar{z}.\exists y_1 \ldots \exists y_n.\psi \wedge R_i(y_1,\ldots,y_n) \oplus \Delta \oplus (Rel|Select|Proj|Ren)}{\phi \wedge (\neg)\exists \bar{z}.\psi \oplus \Delta \circ \{y_j/\rho(A_j)\} \oplus (Rel, \mathcal{R}_i|Select|Proj|Ren,\rho)}$$

- ⋆ *$R_i \in S$, $\mathcal{R}_i \in \mathcal{S}$ is a schema instance of R_i and $A_1,\ldots,A_n$ is the sequence of attributes defined for R_i, where $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ is a database instance of an extended database schema $D = (S, DC, IF)$*
- ⋆ $n = nAtt(R_i)$
- ⋆ $\rho : A_1 \ldots A_n \rightarrow B_1 \ldots B_n$

(2) $$\frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge y_i = t_i \oplus \Delta \oplus (Rel|Select|Proj|Ren)}{\phi \wedge (\neg)\exists \bar{z}.\psi \oplus \Delta^* \oplus (Rel|Select^*|Proj^*|Ren)}$$

- ⋆ *ϕ and ψ contain neither formulas $R(y_1,\ldots,y_n)$, nor $e \lhd x$*
- ⋆ $Decomp((\neg)y_i\Delta = t_i\Delta|Select|Proj|\Delta) = (Select^*|Proj^*|\Delta^*)$

(3) $$\frac{\phi \wedge (\neg)\exists \bar{z}.\exists x.\psi \wedge e \lhd x \oplus \Delta \oplus (Rel|Select|Proj|Ren)}{\phi \wedge (\neg)\exists \bar{z}.\psi \oplus \Delta \circ \{x/e\} \oplus (Rel|Select|Proj|Ren)}$$

- ⋆ *ϕ and ψ contain no formulas $R(y_1,\ldots,y_n)$*

(4) $$\frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge t_1 \Downarrow t_2 \oplus \Delta \oplus (Rel|Select|Proj|Ren)}{\phi \wedge (\neg)\exists \bar{z}.\psi \oplus \Delta^* \oplus (Rel|Select^*|Proj^*|Ren)}$$

- ⋆ *ϕ and ψ contain neither formulas $R(y_1,\ldots,y_n)$, $e \lhd x$, nor $y = t$*
- ⋆ *$t_1\Delta$ or $t_2\Delta$ contains no free variables in $\phi \wedge (\neg)\exists \bar{z}.\ \psi \wedge t_1 \Downarrow t_2$*
- ⋆ $Decomp((\neg)t_1\Delta \bowtie t_2\Delta|Select|Proj|\Delta) = (Select^*|Proj^*|\Delta^*)$

(5) $$\frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge t_1 \Uparrow t_2 \oplus \Delta \oplus (Rel|Select|Proj|Ren)}{\phi \wedge (\neg)\exists \bar{z}.\psi \oplus \Delta^* \oplus (Rel|Select^*|Proj^*|Ren)}$$

- ⋆ *ϕ and ψ contain neither formulas $R(y_1,\ldots,y_n)$, $e \lhd x$, nor $y = t$*
- ⋆ *$t_1\Delta$ or $t_2\Delta$ contains no free variables in $\phi \wedge (\neg)\exists \bar{z}.\ \psi \wedge t_1 \Uparrow t_2$*
- ⋆ $Decomp((\neg)t_1\Delta <> t_2\Delta|Select|Proj|\Delta) = (Select^*|Proj^*|\Delta^*)$

In the above transformation rules, we consider the following cases:
$Decomp((\neg)e_1 = e_2|Select|Proj|\Delta) = Decomp(e_1 \neq e_2|Select|Proj|\Delta)$
$Decomp((\neg)e_1 \bowtie e_2|Select|Proj|\Delta) = Decomp(e_1 \not\bowtie e_2|Select|Proj|\Delta)$
$Decomp((\neg)e_1 <> e_2|Select|Proj|\Delta) = Decomp(e_1 \not<> e_2|Select|Proj|\Delta)$
Finally, $Decomp(e_1 \diamondsuit_a e_2|Select|Proj|\Delta) = (Select^*|Proj^*|\Delta^*)$,
with $\diamondsuit_a \in \{=, \bowtie, <>, \neq, \not\bowtie, \not<>\}$, is computed as follows:

⋆ $e \diamondsuit_a y$ and $var(e) = \emptyset$, then
$Proj^* = Proj, \overset{\diamondsuit_a}{e} \mid Select^* = Select \mid \Delta^* = \Delta \circ \{y/\ \overset{\diamondsuit_a}{e}\}$;

⋆ $e \diamondsuit_a A$, $var(e) = \emptyset$ and $A \in Key(\delta_{Ren}(Rel)) \cup NonKey(\delta_{Ren}(Rel))$, then
$Proj^* = Proj \mid Select^* = Select, e \diamondsuit_a A \mid \Delta^* = \Delta$;

⋆ $e \diamondsuit_a c(e_1,\ldots,e_n)$, $var(e) = \emptyset$, with $c \in DC$, then
if $var(c(e_1,\ldots,e_n)) = \emptyset$, then
$Proj^* = Proj \mid Select^* = Select, e \diamondsuit_a c(e_1,\ldots,e_n) \mid \Delta^* = \Delta$;
else
$Proj^* = Proj, Proj_1,\ldots,Proj_n \mid Select^* = Select, Select_1,\ldots,Select_n \mid$
$\Delta^* = \Delta_n$;
where $Decomp(c.i(e) \diamondsuit_a e_i|Select|Proj|\Delta_i) = (Select_i|Proj_i|\Delta_{i+1})$, $\Delta_0 = \Delta$;

⋆ $e \diamondsuit_a f\ e_1 \ldots e_n$, $var(e) = \emptyset$, with $f \in IF$, then
if $var(f\ e_1 \ldots e_n) = \emptyset$, then
$Proj^* = Proj \mid Select^* = Select, e \diamondsuit_a f\ e_1 \ldots e_n \mid \Delta^* = \Delta$;
else
$Proj^* = Proj, Proj_1,\ldots,Proj_n \mid Select^* = Select, Select_1,\ldots,Select_n$
$\Delta^* = \Delta_n$;
where $Decomp(f.i\ e \diamondsuit_a e_i|Select|Proj|\Delta_i) = (Select_i|Proj_i|\Delta_{i+1})$, $\Delta_0 = \Delta$.

In order to prove the theorem, we should prove the following:

(1) if $(\phi^\star \ \oplus\ id \oplus \emptyset) \rightarrow^n (\emptyset \oplus \Delta \oplus (Rel_{\phi^\star}|Select_{\phi^\star}|Proj_{\phi^\star}|Ren_{\phi^\star}))$, where $Rel_{\phi^\star} = \mathcal{R}_i, \ldots, \mathcal{R}_j$ (i.e. starting from calculus formula $\phi^\star$ and applying the calculus and algebra transformation rules n times, the algebra expression $\Psi_{\phi^\star} \equiv \pi_{Proj_{\phi^\star}}(\sigma_{Select_{\phi^\star}}(\delta_{Ren_{\phi^\star}}(Rel_{\phi^\star})))$ is obtained), then:

(1.1) there exists a tuple V such that $V = \bar{x}\eta \in Ans(\mathcal{D}, \phi^\star)$, iff there exists a permutation V^* of tuple V such that $V^* \in \Psi_{\phi^\star}(\mathcal{R}_i, \ldots, \mathcal{R}_j)$

(1.2) $\phi^\star$ is a safe calculus formula, iff $\Psi_{\phi^\star}(\mathcal{R}_i, \ldots, \mathcal{R}_j)$ is a closed algebra expression

(2) if $(\emptyset \ \oplus\ id \oplus (Rel^\star|Select^\star|Proj^\star|Ren^\star)) \rightarrow^n (\phi_{\Psi^\star} \oplus \Delta \oplus \emptyset)$, where $Rel^\star = \mathcal{R}_i, \ldots, \mathcal{R}_j$ (i.e. starting from the algebra expression $\Psi^\star \equiv \pi_{Proj^\star}(\sigma_{Select^\star}$ $(\delta_{Ren^\star}(Rel^\star)))$ and applying the calculus and algebra transformation rules n times, the calculus formula $\phi_{\Psi^\star}$ is obtained), then:

(2.1) there exists a tuple V such that $V \in \Psi^\star(\mathcal{R}_i, \ldots, \mathcal{R}_j)$, iff there exists a permutation V^* of tuple V such that $V^* = \bar{x}\eta \in Ans(\mathcal{D}, \phi_{\Psi^\star})$

(2.2) $\Psi^\star(\mathcal{R}_i, \ldots, \mathcal{R}_j)$ is a closed algebra expression, iff $\phi_{\Psi^\star}$ is a safe calculus formula

Let us start proving (1), that is $(\phi^\star \ \oplus id \oplus \emptyset) \rightarrow^n (\emptyset \oplus \Delta \oplus (Rel_{\phi^\star}|Select_{\phi^\star}|Proj_{\phi^\star}|$ $Ren_{\phi^\star}))$:

For each transformation step, applying the rule,

$$\frac{\phi \ \oplus\ \Delta \ \oplus\ (Rel|Select|Proj|Ren)}{\phi^* \ \oplus\ \Delta^* \ \oplus\ (Rel^*|Select^*|Proj^*|Ren^*)}$$

(1.1) η is a substitution and $W_1 \times W_2$ is a permutation of a tuple $(V_1, \ldots, V_n)$, such that $W_1 = \bar{y}\eta$ with $\bar{x}\eta \in Ans(\mathcal{D}, \phi)$ and $\bar{y} = \bar{x} \setminus dom(\Delta)$, and $W_2 = \pi_{Proj}(W_3)$ with $W_3 \in \sigma_{Select}(\delta_{Ren}(Rel))$, iff by Lemma 10.3, there exists a substitution η^* and a tuple $W_1^* \times W_2^*$ (i.e. a permutation of the tuple $(V_1, \ldots, V_n)$), such that $W_1^* = \bar{z}\eta^*$ with $\bar{u}\eta^* \in Ans(\mathcal{D}, \phi^*)$ and $\bar{z} = \bar{u} \setminus dom(\Delta^*)$, and $W_2^* = \pi_{Proj^*}(W_3^*)$ with $W_3^* \in \sigma_{Select^*}$ $(\delta_{Ren^*}(Rel^*))$. Therefore, starting from $\phi^\star$, a substitution η and $\Delta = id$, then, by iterating transformation steps, there exists a tuple $V = \bar{x}\eta \in Ans(\mathcal{D}, \phi^\star)$ iff, by Lemma 10.3, we can find a permutation V^* of tuple V such that $V^* \in \Psi_{\phi^\star}(\mathcal{R}_i, \ldots, \mathcal{R}_j) \equiv \pi_{Proj_{\phi^\star}}(\sigma_{Select_{\phi^\star}}(\delta_{Ren_{\phi^\star}}(Rel_{\phi^\star})))$;

(1.2) ϕ is a safe calculus formula and $\pi_{Proj}(\sigma_{Select}(\delta_{Ren}(Rel)))$ is a closed algebra expression, iff, by Lemma 10.4, ϕ^* is a safe calculus formula and $\pi_{Proj^*}(\sigma_{Select^*}(\delta_{Ren^*}(Rel^*)))$ is a closed algebra expression. Now, if $\phi^\star$ is safe w.r.t. Lemma 10.4, then, in particular, $\phi^\star$ is safe w.r.t. Definition 4.5

(Safe Calculus Formulas). Therefore, by iterating transformation steps and by Lemma 10.4, $\Psi_{\phi^\star}(\mathcal{R}_i, \ldots, \mathcal{R}_j) \equiv \pi_{Proj_{\phi^\star}}(\sigma_{Select_{\phi^\star}}(\delta_{Ren_{\phi^\star}}(Rel_{\phi^\star})))$ is a closed algebra expression w.r.t. Lemma 10.4, and, in particular, w.r.t. Definition 5.7 (Algebra Expressions).

Analogously, we can prove (2), that is, $(\emptyset \oplus id \oplus (Rel^\star | Select^\star | Proj^\star | Ren^\star)) \rightarrow^n (\phi_{\Psi^\star} \oplus \Delta \oplus \emptyset)$. ■

6.3 Query and Algebra Equivalence

Finally, in this subsection, we will show the result of equivalence between the proposed functional logic query language and the extended relational algebra. In order to present this result, we will use the previous equivalence results.

Corollary 6.1 (Query and Algebra Equivalence)
Let $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ be a database instance, where $\mathcal{S}$ is a sequence of schema instances $\mathcal{R}_1, \ldots, \mathcal{R}_n$. In addition, given an extended database schema $D = (S, DC, IF)$, where S is a sequence of relation names $R_1, \ldots, R_n$ such that $\mathcal{R}_i$ is a schema instance of R_i $(1 \leq i \leq n)$, then:

(1) given a safe query $\mathcal{Q}^\star$ against $\mathcal{D}$, then there exists a closed algebra expression $\Psi_{\mathcal{Q}^\star}(\mathcal{R}_i, \ldots, \mathcal{R}_j)$ ($\mathcal{R}_p$ is schema instance of $\mathcal{S}$ with $i \leq p \leq j$), such that $Ans(\mathcal{D}, \mathcal{Q}^\star) = \Psi_{\mathcal{Q}^\star}(\mathcal{R}_i, \ldots, \mathcal{R}_j)$

(2) given a closed algebra expression $\Psi^\star(\mathcal{R}_i, \ldots, \mathcal{R}_j)$ ($\mathcal{R}_p$ is a schema instance of $\mathcal{S}$ with $i \leq p \leq j$), then there exists a safe query $\mathcal{Q}_{\Psi^\star}$ against $\mathcal{D}$, such that $\Psi^\star(\mathcal{R}_i, \ldots, \mathcal{R}_j) = Ans(\mathcal{D}, \mathcal{Q}_{\Psi^\star})$

Proof

(1) Trivial; from the safe query $\mathcal{Q}^\star$, by Theorem 6.1, we can obtain a safe calculus formula $\phi_{\mathcal{Q}^\star}$ such that $Ans(\mathcal{D}, \mathcal{Q}^\star) = Ans(\mathcal{D}, \phi_{\mathcal{Q}^\star})$; now, from $\phi_{\mathcal{Q}^\star}$ and by Theorem 6.2, we can obtain the corresponding closed algebra expression $\Psi_{\mathcal{Q}^\star}(\mathcal{R}_i, \ldots, \mathcal{R}_j)$ such that $Ans(\mathcal{D}, \phi_{\mathcal{Q}^\star}) = \Psi_{\mathcal{Q}^\star}(\mathcal{R}_i, \ldots, \mathcal{R}_j)$; therefore, we can obtain $\Psi_{\mathcal{Q}^\star}(\mathcal{R}_i, \ldots, \mathcal{R}_j)$ such that $Ans(\mathcal{D}, \mathcal{Q}^\star) = \Psi_{\mathcal{Q}^\star}(\mathcal{R}_i, \ldots, \mathcal{R}_j)$;

(2) Trivial; from a closed algebra expression $\Psi^\star(\mathcal{R}_i, \ldots, \mathcal{R}_j)$, by Theorem 6.2, we can obtain a safe calculus formula $\phi_{\Psi^\star}$ such that $\Psi^\star(\mathcal{R}_i, \ldots, \mathcal{R}_j) = Ans(\mathcal{D}, \phi_{\Psi^\star})$; finally, from $\phi_{\Psi^\star}$ and by Theorem 6.1, we can obtain the corresponding safe query $\mathcal{Q}_{\Psi^\star}$ such that $Ans(\mathcal{D}, \phi_{\Psi^\star}) = Ans(\mathcal{D}, \mathcal{Q}_{\Psi^\star})$; therefore, we can obtain $\mathcal{Q}_{\Psi^\star}$ such that $\Psi^\star(\mathcal{R}_i, \ldots, \mathcal{R}_j) = Ans(\mathcal{D}, \mathcal{Q}_{\Psi^\star})$.

■

§7 A Comparison with the Related Work

Data models and query languages have been studied for *functional deductive languages*, such as *FDL* [28], *PFL* [37], among others, for *logic deductive languages*, like *CORAL* [31], *ADITI* [39] and *LDL* [10], among others, and for *constraint databases* such as *DEDALE* [34]. In our case, we consider a data model and a query language, which combine and enrich some aspects of the mentioned models and languages in a uniform way.

Functional models [26, 27] are usually based on the data model proposed by Shipman [35]. Taking as an example [26], we have that this data model nicely manages the following notions:

(1) *Schema definitions* which consist of *relation definitions*; then, these relations define a sequence of *key* names, as well as *attribute definitions* by means of type definitions, allowing relation names as types;

(2) *Instances* are defined from the key names included in the relation definitions, and *rules* in the form of *rewriting rules* defining attribute values. Attributes can be *multi-valued*, in the sense that they can represent a *set of values* in the form of a *record*. In addition, the model can handle *default values* [29] for attributes, and *partial information* in the form of *null values* [19, 26, 27].

(3) The query language is based on *list comprehension* syntax [38]. List comprehension is a high-level formalism similar to the relational calculus which allows *encapsulated search*. Queries can handle *lazy functions* in order to manage the collected values by means of the list comprehension syntax.

In *Deductive databases with complex values*, *attributes* can be *multi-valued* built from *set and tuple constructors* [15, 10]. Instances are defined by means of Prolog-style facts and (*recursive*) rules. However, like Prolog, relations are defined over finite values, the relations are finite, and the querying mechanism deals with these finite relations. In addition, the query language is based on the solving of Prolog-style queries, although alternative query languages, like extensions from relational calculus and algebra, have been studied [1, 2].

In *Constraint databases* [17, 18], the relational model is generalized by considering tuples as *quantifier-free conjunctions of constraints over variables.* Instances include tuples defined by means of Prolog-style rules enriched with constraints. Constraint databases allow the handling of *infinite relations*, although *finitely (symbolically) representable* by using, for example, linear constraints [32]. Moreover, the database querying mechanisms deal efficiently with this finite representation. The query language is based on the solving of Prolog-style queries enriched with constraints. Finally, in this paradigm, alternative query languages, based on extensions of relational calculus and algebra, have been also studied [17, 33].

Now, w.r.t. the query formalisms based on extensions of relational languages, as previously mentioned, extended relational calculi have been studied as alternative query formalisms for *deductive databases* [1, 20] and *constraint databases* [8, 16, 17, 18, 33]. Our extended relational calculus is in the line of [1], in which deductive databases can handle complex values built from the *set* and *tuple* constructors. In our case, we generalize the mentioned calculus, allowing to deal with *complex values built from (arbitrary) recursively defined datatypes*.

In addition, our calculus also follows the proposed line by the calculi developed for constraint databases [17, 33], in the sense of allowing the handling of *infinite database instances*. However, in the framework of constraint databases, infinite database instances model *infinite objects*, represented by *(linear) equations and inequations*, and *intervals* handled in a symbolic way. In our framework, infinite database instances are handled by means of *laziness* and *partial approximations*. Moreover, our calculus handles constraints defined from equality and inequality constraints over complex values.

Like extended relational calculi, extended relational algebras have been also studied as alternative query formalisms for *deductive databases* [1] and *constraint databases*[16, 7]. Our extended relational algebra is in the line of [1], although, in our case, we generalize the mentioned algebra, allowing to handle *complex values built from arbitrary recursively defined datatypes*. In addition, our algebra is also in the line of algebra proposed for constraint databases [16], in the sense of dealing with *equality and inequality constraints*, but, here, used for comparing sets of complex values.

§8 Conclusions and Future Work

In this paper, we have studied how to express queries in a framework integrating the context of databases and functional logic programming. We have proposed three alternative query formalisms; that is, a functional logic query language, an extended relational calculus, and an extended relational algebra, showing that all of them are alternative equivalent ways of expressing queries in our framework. As future work, we will propose two main lines of research:

(a) the study of our extended relational formalisms as data definition languages, as well as the development of operational mechanisms for such languages; and (b) the implementation of both query formalisms in the current prototype of $\mathcal{INDALOG}$.

§9 Acknowledgements

We would like to thank anonymous referees for their useful comments. This work has been partially supported by the Spanish project of the Ministry of Science and Technology "INDALOG" TIC2002-03968 under FEDER funds.

References

1) S. Abiteboul and C. Beeri. The Power of Languages for the Manipulation of Complex Values. *The VLDB Journal*, 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 International Symposium on Functional and Logic Programming, FLOPS*, LNCS 2024, pages 153–169. Springer, 2001.

4) J. M. Almendros-Jiménez and A. Becerra-Terón. A Relational Algebra for Functional Logic Deductive Databases. In *Procs. of Perspectives of System Informatics, PSI*, LNCS 2890, pages 494–508. Springer, 2003.

5) J. M. Almendros-Jiménez and A. Becerra-Terón. A Safe Relational Calculus for Functional Logic Deductive Databases, Selected Papers of the International Workshop on Functional and (Constraint) Logic Programming, WFLP. *Electronic Notes on Theoretical Computer Science, ENTCS*, 86(3), 2003.

6) J. M. Almendros-Jiménez, A. Becerra-Terón, and J. Sánchez-Hernández. A Computational Model for Funtional Logic Deductive Databases. In *Proc. of International Conference on Logic Programming, ICLP*, LNCS 2237, pages 331–347. Springer, 2001.

7) A. Belussi, E. Bertino, and B. Catania. An Extended Algebra for Constraint Databases. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 10(5):686–705, 1998.

8) M. Benedikt and L. Libkin. *Constraint Databases*, chapter Query Safety with Constraints, pages 109–129. Springer, 2000.

9) P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science, TCS*, 149(1):3–48, 1995.

10) D. Chimenti, R. Gamboa, R. Krishnamurthy, S. A. Naqvi, S. Tsur, and C. Zaniolo. The LDL System Prototype. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 2(1):76–90, 1990.

11) E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM, CACM*, 13(6):377–387, 1970.

12) 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. *Journal of Logic Programming, JLP*, 1(40):47–87, 1999.

13) M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming, JLP*, 19,20:583–628, 1994.

14) M. Hanus. Curry: An Integrated Functional Logic Language, Version 0.8. Technical report, University of Kiel, Germany, 2003.

15) R. Hull and J. Su. Deductive Query Language for Recursively Typed Complex Objects. *Journal of Logic Programming, JLP*, 35(3):231–261, 1998.

16) P. Kanellakis and D. Goldin. Constraint Query Algebras. *Constraints*, 1(1–2):45–83, 1996.

17) P. Kanellakis, G. Kuper, and P. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences, JCSS*, 51(1):26–52, 1995.

18) G. M. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases.* Springer, 2000.

19) L. Libkin. A Semantics-based Approach to Design of Query Languages for Partial Information. In *Proc. of Semantics in Databases*, LNCS 1358, pages 170–208. Springer, 1995.

20) M. Liu. Deductive Database Languages: Problems and Solutions. *ACM Computing Surveys*, 31(1):27–62, 1999.

21) F. J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In *Procs. of Conference on Rewriting Techniques and Applications, RTA*, LNCS 1631, pages 244–247. Springer, 1999.

22) F. J. López-Fraguas and J. Sánchez-Hernández. Proving Failure in Functional Logic Programs. In *Proc. of the International Conference on Computational Logic, CL*, LNCS 1861, pages 179–193. Springer, 2000.

23) F. J. López-Fraguas and J. Sánchez-Hernández. A Proof Theoretic Approach to Failure in Functional Logic Programming. *Theory and Practice of Logic Programming, TPLP*, 4(1-2):41–74, 2004.

24) R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

25) J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming, JLP*, 12(3):191–223, 1992.

26) N. Paton, R. Cooper, H. Williams, and P. Trinder. *Database Programming Languages.* C.A.R. Hoare Series. Prentice Hall, 1996.

27) A. Poulovassilis. FDL: An Integration of the Functional Data Model and the Functional Computation Model. In *Proc. of the British National Conference on Databases, BNCOD*, pages 215–236. Cambridge University Press, 1988.

28) A. Poulovassilis. The Implementation of FDL, a Functional Database Language. *The Computer Journal*, 35(2):119–128, 1992.

29) A. Poulovassilis and P. King. Extending the Functional Data Model to Computational Completeness. In *Proc. of International Conference on Extending Database Technology, EDBT*, LNCS 416, pages 75–91. Springer, 1990.

30) A. Poulovassilis and C. Small. A Functional Programming Approach to Deductive Databases. In *Proc. of Very Large Data Bases Conference, VLDB*, pages 491–500. Morgan Kaufmann, 1991.

31) R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL Deductive Database System. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 167–176. ACM Press, 1993.

32) P. Z. Revesz. Safe Datalog Queries with Linear Constraints. In *Proc. of International Conferente on Principles and Practice of Constraint Programming, CP*, LNCS 1520, pages 355–369. Springer, 1998.

33) P. Z. Revesz. Safe Query Languages for Constraint Databases. *ACM Transactions on Database Systems, TODS*, 23(1):58–99, 1998.

34) P. Rigaux, M. Scholl, L. Segoufin, and S. Grumbach. Building a Constraint-based Spatial Database System: Model, Languages, and Implementation. *Information Systems*, 28(6):563–595, 2003.

35) D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems, TODS*, 6(1):140–173, 1981.

36) O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of Set Terms in the Logic Data Language (LDL). *Journal of Logic Programming, JLP*, 12(1–2):89–119, 1992.

37) C. Small and A. Poulovassilis. An Overview of PFL. In *Proc. of International Workshop on Database Programming Languages, DBPL*, pages 96–110. Morgan Kaufmann, 1991.

38) P. W. Trinder. Comprehensions, A Query Notation for DBPL. In *Proc. of International Workshop on Database Programming Languages, DBPL*, pages 55–68. Morgan Kaufman, 1991.

39) J. Vaghani, K. Ramamohanarao, D. B. Kemp, Z. Somogyi, P. J. Stuckey, T. S. Leask, and J. Harland. The Aditi Deductive Database System. *The VLDB Journal*, 3(2):245–288, 1994.

§10 Appendix. Proofs of the Lemmas

Query and Calculus Equivalence

In this subsection, we will show two additional results in two Lemmas which will prove the *equivalence of the calculus and query transformation rules*, and are used in the proof of Theorem 6.1. From a non-formal point of view, the first result states the following: *once applied a given transformation rule, then*

- *the set of answers obtained from ϕ and $\mathcal{Q}$ is the same as one obtained from ϕ^* and $\mathcal{Q}^*$.*

Now, we formally state this result in the following Lemma.

Lemma 10.1 (Answers in Calculus and Query Transformation Rules) Given a calculus and query transformation rule,

$$\frac{\phi \oplus \ \mathcal{Q}}{\phi^* \oplus \mathcal{Q}^*}$$

and let $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ be a database instance of an extended database schema $D = (S, DC, IF)$, then:

there exists a substitution η such that $\bar{x}\eta \in Ans(\mathcal{D}, \phi) \cap Ans(\mathcal{D}, \mathcal{Q})$, with $\bar{x} = free(\phi) \cup var(\mathcal{Q})$,
iff
there exists a substitution η^* such that $\bar{x}\eta^* \in Ans(\mathcal{D}, \phi^*) \cap Ans(\mathcal{D}, \mathcal{Q}^*)$, with $\bar{x} = free(\phi^*) \cup var(\mathcal{Q}^*)$ and $\eta = \eta^*|_{free(\phi)\cup var(\mathcal{Q})}$,

where:

- $\bar{x}\eta$ denotes a tuple $(x_1\eta, \ldots, x_n\eta)$ and, in addition, $\bar{x}\eta \in Ans(\mathcal{D}, \phi) \cap Ans(\mathcal{D}, \mathcal{Q})$ ($\bar{x} = free(\phi) \cup var(\mathcal{Q})$) whenever $(\mathcal{D}, \eta) \models_C \phi$ and $(\mathcal{D}, \eta) \models_Q \mathcal{Q}$;
- $\eta^*|_{free(\phi)\cup var(\mathcal{Q})}$ denotes the substitution restricted to the variables of $\mathcal{Q}$ and the free variables of ϕ.

Proof

Let us see the proof for the main calculus and query transformation rules.

$$\textbf{(1)}\ \frac{\phi \wedge \exists \bar{\mathsf{z}}.\ \psi\ \oplus\ \mathsf{e}_1 \bowtie \mathsf{e}_2,\ \ \mathcal{Q}}{\phi \wedge \exists \bar{\mathsf{z}}.\exists \mathsf{x}.\exists \mathsf{y}.\ \psi \wedge \mathsf{e}_1 \triangleleft \mathsf{x}\ \ \wedge\ \ \mathsf{e}_2 \triangleleft \mathsf{y}\ \ \wedge\ \ \mathsf{x} \Downarrow \mathsf{y}\ \oplus\ \mathcal{Q}}$$

$(\mathcal{D}, \eta) \models_Q e_1 \bowtie e_2$ iff there exist $t_1 \in [\![e_1]\!]^{\mathcal{D}}\eta$ and $t_2 \in [\![e_2]\!]^{\mathcal{D}}\eta$, such that $t_1 \downarrow t_2$ and $t_1, t_2 \in adom(e_1, \mathcal{D}) \cup adom(e_2, \mathcal{D})$. Now, let η^* be a substitution such that $\eta^* = \eta \circ \{x/t_1,\, y/t_2\}$, then $x\eta^* \in [\![e_1]\!]^{\mathcal{D}}\eta^*$ and $y\eta^* \in [\![e_2]\!]^{\mathcal{D}}\eta^*$; thus iff

$(\mathcal{D},\eta^*) \models_C e_1 \lhd x \wedge e_2 \lhd y$. In addition, by Definition 4.7 (*Active Domain of Calculus Terms*), $adom(x,\mathcal{D}) = adom(e_1,\mathcal{D})$ and $adom(y,\mathcal{D}) = adom(e_2,\mathcal{D})$, and since $x\eta^* \downarrow y\eta^*$, and $x\eta^*, y\eta^* \in adom(x,\mathcal{D}) \cup adom(y,\mathcal{D})$, and thus iff $(\mathcal{D},\eta^*) \models_C x \Downarrow y$. Therefore, $(\mathcal{D},\eta^*) \models_C (e_1 \lhd x \wedge e_2 \lhd y \wedge x \Downarrow y)$ and, finally, $(\mathcal{D},\eta) \models_C \phi$, $(\mathcal{D},\eta) \models_C (\exists\bar{z}.\exists x.\exists y.\ \psi \wedge e_1 \lhd x \wedge e_2 \lhd y \wedge x \Downarrow y)$ and $(\mathcal{D},\eta) \models_Q \mathcal{Q}$.

$$\textbf{(2)}\quad \frac{\phi \wedge \neg\exists\bar{\mathsf{z}}.\psi \oplus \mathsf{e_1} \not\bowtie \mathsf{e_2},\ \mathcal{Q}}{\phi \wedge \neg\exists\bar{\mathsf{z}}.\exists \mathsf{x}.\exists \mathsf{y}.\psi \wedge \mathsf{e_1} \lhd \mathsf{x}\ \wedge\ \mathsf{e_2} \lhd \mathsf{y}\ \wedge\ \mathsf{x} \Downarrow \mathsf{y} \oplus \mathcal{Q}}$$

$(\mathcal{D},\eta) \models_Q e_1 \not\bowtie e_2$ iff for all $t_1 \in [\![e_1]\!]^{\mathcal{D}}\eta$ and $t_2 \in [\![e_2]\!]^{\mathcal{D}}\eta$, then $t_1 \not\downarrow t_2$ and $t_1, t_2 \in adom(e_1,\mathcal{D}) \cup adom(e_2,\mathcal{D})$. Now, let η^* be substitutions such that $\eta^* = \eta \circ \{x/t_1,\ y/t_2\}$, then $x\eta^* \in [\![e_1]\!]^{\mathcal{D}}\eta^*$ and $y\eta^* \in [\![e_2]\!]^{\mathcal{D}}\eta^*$; thus iff $(\mathcal{D},\eta^*) \models_C e_1 \lhd x \wedge e_2 \lhd y$. In addition, by Definition 4.7 (*Active Domain of Calculus Terms*), $adom(x,\mathcal{D}) = adom(e_1,\mathcal{D})$ and $adom(y,\mathcal{D}) = adom(e_2,\mathcal{D})$, and since $\eta^* = \eta \circ \{x/t_1,\ y/t_2\}$, $t_1 \not\downarrow t_2$ and $t_1, t_2 \in adom(e_1,\mathcal{D}) \cup adom(e_2,\mathcal{D})$, then, in particular, $x\eta^* \not\downarrow y\eta^*$ and $x\eta^*, y\eta^* \in adom(x,\mathcal{D}) \cup adom(y,\mathcal{D})$; then $(\mathcal{D},\eta^*) \models_C \neg x \Downarrow y$; now, if η^* is such that $x\eta^* \downarrow y\eta^*$ then $x\eta^* \notin [\![e_1]\!]^{\mathcal{D}}\eta^*$ or $y\eta^* \notin [\![e_2]\!]^{\mathcal{D}}\eta^*$, by hypothesis; thus $(\mathcal{D},\eta) \models_C \phi$, $(\mathcal{D},\eta) \models_C \neg(\exists\bar{z}.\exists x.\exists y.\ \psi \wedge e_1 \lhd x \wedge e_2 \lhd y \wedge x \Downarrow y)$ and $(\mathcal{D},\eta) \models_Q \mathcal{Q}$.

(3) and **(4)** are similar.

(5) and **(6)** are analogous, let us see **(6)**:

$$\textbf{(6)}\quad \frac{\phi \wedge (\neg)\exists\bar{\mathsf{z}}.\psi \wedge \mathsf{A_i}\ \mathsf{e_1} \ldots \mathsf{e_k} \lhd \mathsf{x} \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists\bar{\mathsf{z}}.\exists \mathsf{y_1} \ldots \exists \mathsf{y_n}.\psi \wedge \mathsf{R}(\mathsf{y_1},\ldots,\mathsf{y_k},\ldots,\mathsf{y_i},\ldots,\mathsf{y_n})\ \wedge\ \mathsf{e_1} \lhd \mathsf{y_1} \wedge \ldots \wedge \mathsf{e_k} \lhd \mathsf{y_k} \wedge \mathsf{y_i} \lhd \mathsf{x} \oplus \mathcal{Q}}$$

$\star$ $\mathtt{A_i} \in \mathtt{NonKey(R)}$ *and* $\mathtt{R} \in \mathtt{S}$*, where* $\mathtt{D} = (\mathtt{S}, \mathtt{DC}, \mathtt{IF})$ *is an extended database schema.*

In the positive case, $(\mathcal{D},\eta) \models_C \exists\bar{z}.\ \psi \wedge A_i\ e_1 \ldots e_k \lhd x$ iff there exists a substitution η' such that $(\mathcal{D},\eta') \models_C A_i\ e_1 \ldots e_k \lhd x$. Therefore, iff $x\eta' \in [\![A_i\ e_1 \ldots e_k]\!]^{\mathcal{D}}\eta'$; that is, $v_i = x\eta' \in V_i\eta_V$ for a given substitution η_V, whenever $([\![e_1]\!]^{\mathcal{D}}\eta', \ldots, [\![e_k]\!]^{\mathcal{D}}\eta') = (V_1\eta_V, \ldots, V_k\eta_V)$ and there exists a tuple $(V_1, \ldots, V_k, \ldots, V_i, \ldots, V_n) \in \mathcal{R}$, where $\mathcal{R} \in \mathcal{S}$ is an instance of the relation $R \in S$ and $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ is a database instance of an extended database schema $D = (S, DC, IF)$. Now, let η^* be a substitution, such that $\eta^* = \eta' \circ \{y_1/v_1, \ldots, y_n/v_n\}$ and $v_1 \in V_1\eta_V, \ldots, v_n \in V_n\eta_V$. Therefore, $(\mathcal{D},\eta^*) \models_C R(y_1, \ldots, y_n)$ and since $y_1\eta^* \in [\![e_1]\!]^{\mathcal{D}}\eta^* \ldots y_k\eta^* \in [\![e_k]\!]^{\mathcal{D}}\eta^*$, then $(\mathcal{D},\eta^*) \models_C e_i \lhd y_i$. Now, given that $y_i\eta^* = x\eta$, then $(\mathcal{D},\eta^*) \models_C y_i \lhd x$ and we can prove $(\mathcal{D},\eta^*) \models_C R(y_1, \ldots, y_k, \ldots, y_i, \ldots, y_n) \wedge e_1 \lhd y_1 \wedge \ldots \wedge e_k \lhd y_k \wedge y_i \lhd x$. Finally, $(\mathcal{D},\eta) \models_C \phi$, $(\mathcal{D},\eta) \models_C$

$\exists\bar{z}.\exists y_1 \ldots \exists y_n.\ \psi\ \wedge\ R(y_1, \ldots, y_k, \ldots, y_i, \ldots, y_n) \wedge\ e_1 \triangleleft y_1 \wedge \ldots \wedge e_k \triangleleft y_k\ \wedge\ y_i \triangleleft x$ and $(\mathcal{D}, \eta) \models_Q \mathcal{Q}$.

In the negative case, if $x\eta^* \notin [\![A_i\ e_1\ \ldots\ e_k]\!]^{\mathcal{D}}\eta^*$ for every $\eta^* = \eta \circ \{x/v\}$, since $x\eta^* \in adom(A\ e_1, \ldots, e_k, \mathcal{D})$ then $x\eta^* \in \cup_{(V_1,\ldots,V_i,\ldots,V_n)\in\mathcal{R},\lambda\in Subst_{DC,\perp,\mathsf{F}}} V_i\lambda$; Let be susbtitutions $\eta^{**} = \eta \circ \{x/v\} \circ \{y_j/v_j\}$, where $v_j \in [\![e_j]\!]^{\mathcal{D}}\eta^* = V_j\lambda^*$; then if $(\mathcal{D}, \eta^{**}) \models_C R(y_1, \ldots, y_i, \ldots, y_n)$ and $(\mathcal{D}, \eta^{**}) \models_C e_1 \triangleleft y_1, \ldots, e_k \triangleleft y_k$ then $(\mathcal{D}, \eta^{**}) \not\models_C y_i \triangleleft x$ since $x\eta^* \notin V_i\lambda^*$ and $x\eta^* \in adom(A\ e_1, \ldots, e_k)$; analogously, if $(\mathcal{D}, \eta^{**}) \models_C e_1 \triangleleft y_1, \ldots, e_k \triangleleft y_k$ and $(\mathcal{D}, \eta^{**}) \models_C y_i \triangleleft x$ for a given η^{**}, then $(\mathcal{D}, \eta^{**}) \not\models_C R(y_1, \ldots, y_i, \ldots, y_n)$ since $x\eta^* \notin [\![A_i\ e_1\ \ldots\ e_k]\!]^{\mathcal{D}}\eta^*$; finally, if $(\mathcal{D}, \eta^{**}) \models_C R(y_1, \ldots, y_i, \ldots, y_n), y_i \triangleleft x$ for a given η^{**}, then $(\mathcal{D}, \eta^{**}) \not\models_C e_1 \triangleleft y_1, \ldots, e_k \triangleleft y_k$ by the same reason.

The cases **(7)** and **(8)** are similar. Let us see **(7)** in the positive case, the negative case can be analougously proved.

$$\textbf{(7)}\quad \frac{\phi \wedge (\neg)\exists \bar{\mathtt{z}}.\psi \wedge \mathtt{f}\ \mathtt{e_1} \ldots \mathtt{e_n} \triangleleft \mathtt{x} \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{\mathtt{z}}.\exists \mathtt{y_1} \ldots \mathtt{y_n}.\psi \wedge \mathtt{f}\ \mathtt{y_1} \ldots \mathtt{y_n} \triangleleft \mathtt{x} \wedge \mathtt{e_1} \triangleleft \mathtt{y_1} \wedge \ldots \wedge \mathtt{e_n} \triangleleft \mathtt{y_n} \oplus \mathcal{Q}}$$
$\star$ $\mathtt{f}\ \mathtt{e_1} \ldots \mathtt{e_n} \notin \mathtt{Term_{DC,IF}}(\mathcal{V})$, *where* $\mathtt{D} = (\mathtt{S}, \mathtt{DC}, \mathtt{IF})$ *is an extended database schema.*

$(\mathcal{D}, \eta) \models_C \exists\bar{z}.\psi\ \wedge\ f\ e_1 \ldots e_n \triangleleft x$ iff there exists a substitution η' such that $(\mathcal{D}, \eta') \models_C f\ e_1 \ldots e_n \triangleleft x$. Therefore, $x\eta' \in [\![f\ e_1 \ldots e_n]\!]^{\mathcal{D}}\eta'$, iff $x\eta' \in f^{\mathcal{D}}\ [\![e_1]\!]^{\mathcal{D}}\eta' \ldots [\![e_n]\!]^{\mathcal{D}}\eta'$. Now, there exist c-terms $t_1, \ldots, t_n$ such that $t_1 \in [\![e_1]\!]^{\mathcal{D}}\eta' \ldots t_n \in [\![e_n]\!]^{\mathcal{D}}\eta'$, and thus, iff $x\eta' \in f^{\mathcal{D}}\ t_1 \ldots t_n$. Now, let η^* be a substitution such that $\eta^* = \eta \circ \{y_1/t_1, \ldots, y_n/t_n\}$. Then $y_1\eta^* \in [\![e_1]\!]^{\mathcal{D}}\eta^* \ldots y_n\eta^* \in [\![e_n]\!]^{\mathcal{D}}\eta^*$, and $x\eta^* \in f^{\mathcal{D}}\ [\![y_1]\!]^{\mathcal{D}}\eta^* \ldots [\![e_n]\!]^{\mathcal{D}}\eta^*$; thus, $x\eta^* \in [\![f\ y_1 \ldots y_n]\!]^{\mathcal{D}}\eta^*$ iff $(\mathcal{D}, \eta) \models_C \phi$, $(\mathcal{D}, \eta) \models_C \exists\bar{z}.\exists y_1 \ldots \exists y_n.\psi \wedge f\ y_1 \ldots y_n \triangleleft x \wedge e_1 \triangleleft y_1 \wedge \ldots \wedge e_n \triangleleft y_n$ and $(\mathcal{D}, \eta) \models_Q \mathcal{Q}$.

The cases **(9)** and **(10)** are similar. Let us see the positive of **(9)**, the negative case is analogous.

$$\textbf{(9)}\quad \frac{\phi \wedge (\neg)\exists \bar{\mathtt{z}}.\psi \wedge \mathtt{t} \triangleleft \mathtt{x} \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{\mathtt{z}}.\psi \wedge \mathtt{x} = \mathtt{t} \oplus \mathcal{Q}}$$
$\star$ $\mathtt{x} \in \mathtt{formula\_key}(\phi \wedge (\neg)\exists \bar{\mathtt{z}}.\psi \wedge \mathtt{t} \triangleleft \mathtt{x})$

$(\mathcal{D}, \eta) \models_C \exists\bar{z}.\ \psi\ \wedge\ t \triangleleft x$ iff there exists a substitution η', such that $(\mathcal{D}, \eta') \models_C t \triangleleft x$. Therefore, $x\eta' \in [\![t]\!]^{\mathcal{D}}\eta' = \{t\eta'\}$, and then $x\eta' = t\eta'$. In adition $adom(x, \mathcal{D}) = adom(t, \mathcal{D})$ and therefore, iff $(\mathcal{D}, \eta) \models_C \exists\bar{z}.\ \psi\ \wedge\ x = t$, and we have that $(\mathcal{D}, \eta) \models_C \phi$, $(\mathcal{D}, \eta) \models_C \exists\bar{z}.\ \psi\ \wedge\ x = t$ and $(\mathcal{D}, \eta) \models_Q \mathcal{Q}$. ■

Now, the additional second result states the following: *once applied a given transformation rule, then*

- *ϕ and $\mathcal{Q}$ are safe iff ϕ^* and $\mathcal{Q}^*$ are safe*;

where the Definitions 3.3 (*Safe Queries*) and 4.5 (*Safe Calculus Formulas*) state the safety conditions for the queries and calculus formulas, respectively. However, the Definitions of the required conditions by the above Definitions (i.e. *range restricted in queries*, *range restricted in calculus formulas*, and *safe atomic formulas*) are now modified as follows:

(a) both *Range Restricted C-Terms of Queries* (Definition 3.2) and *Range Restricted C-Terms of Calculus Formulas* (Definition 4.4) are replaced as follows: a c-term t occurring in a query $\mathcal{Q}$ or calculus formula ϕ is *range restricted*, if either:

(1) t belongs to $\bigcup_{s \in query\_key(\mathcal{Q})} cterms(s)$, or

(2) there exists a constraint $e \diamondsuit_q e'$ ($\diamondsuit_q \in \{\bowtie, <>, \not\bowtie, \not<>\}$), such that t belongs to $cterms(e)$ (resp. $cterms(e')$) and every c-term occurring in e' (resp. e) is *range restricted* in $\mathcal{Q}$ or ϕ, or

(4) t occurs in $formula\_key(\phi) \cup formula\_nonkey(\phi)$, or

(5) there exists one equation $e \diamondsuit_c e'$ ($\diamondsuit_c \in \{=, \Uparrow, \Downarrow, \lhd\}$) in ϕ, such that t belongs to $cterms(e)$ (resp. $cterms(e')$) and every c-term of e' (resp. e) is range restricted in $\mathcal{Q}$ or ϕ.

(b) *Safe Atomic formulas* (Definition 4.3) is replaced by:

(1) $R(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$ is safe, if the variables $x_1, \ldots, x_n$ are bounded in ϕ, and for each x_i ($i \leq nKey(R)$) there exists one equation $e_i \lhd x_i$ or $x_i = t_i$ occurring in ϕ;

(2) $x = t$ is *safe*, if the variables occurring in t are distinct from the variables of $formula\_key(\phi)$ and $\bigcup_{s \in query\_key(\mathcal{Q})} cterms(s)$, and x is a variable of $formula\_key(\phi)$ or $\bigcup_{s \in query\_key(\mathcal{Q})} cterms(s)$;

(3) $t_1 \Downarrow t_2$, $t_1 \Uparrow t_2$, and $e_1 \diamondsuit_q e_2$ are safe if the variables occurring in t_1, t_2, e_1 and e_2 are distinct from the variables of $formula\_key(\phi)$ and $\bigcup_{s \in query\_key(\mathcal{Q})} cterms(s)$;

(4) $e \lhd x$ is *safe*, if the variables occurring in e are distinct from the variables of $formula\_key(\phi)$ and $\bigcup_{s \in query\_key(\mathcal{Q})} cterms(s)$, and x is bounded in ϕ.

Note that this safety definition is more general than the original one. In fact, if $\mathcal{Q} = \emptyset$ or $\phi = \emptyset$, then the above conditions coincide with the original ones (i.e. Definition 3.2 for queries, and Definitions 4.3 and 4.4 for calculus formulas). As previously, we formally state this result in the following Lemma.

Lemma 10.2 (Safety in Calculus and Query Transformation Rules) Given a calculus and query transformation rule,

$$\frac{\phi \oplus \ \mathcal{Q}}{\phi^* \oplus \mathcal{Q}^*}$$

and let $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ be a database instance of an extended database schema $D = (S, DC, IF)$, then:

ϕ is a safe calculus formula and $\mathcal{Q}$ is a safe query,
iff
ϕ^* is a safe calculus formula and $\mathcal{Q}^*$ is a safe query

Proof
Let us see the proof for the main calculus and query transformation rules.

Case **(1)**. It can be reasoned that:

(a) those range restricted c-terms in ϕ, ψ and $\mathcal{Q}$ by means of $e_1 \bowtie e_2$, now they are range restricted by means of $e_1 \triangleleft x$, $e_2 \triangleleft y$, $x \Downarrow y$; in addition, by hypothesis, the c-terms of e_1 and e_2 are range restricted and, thus, the variables x and y are range restricted too.

(b) the atomic formulas of ϕ and ψ are safe by hypothesis; in addition, the atomic formulas $e_1 \ \triangleleft \ x$, $e_2 \ \triangleleft \ y$, $x \Downarrow y$ are safe, since e_1 and e_2 do not contain, by hypothesis, key variables, and the variables x and y are variables distinct from key variables due to the renaming of quantified variables.

The cases **(2)**, **(3)** and **(4)** are similar.
The cases **(5)** and **(6)** are similar. Let us see the case **(6)**. It can be reasoned that:

(a) those range restricted c-terms in ϕ, ψ and $\mathcal{Q}$ by means of $A_i\ e_1 \ldots e_k \triangleleft x$, now they are range restricted by means of $R(y_1, \ldots, y_n)$, $e_1 \triangleleft y_1, \ldots, e_k \triangleleft y_k$, $y_i \triangleleft x$; in addition, by hypothesis, the c-terms of $e_1, \ldots, e_k$ are range restricted and, thus the variables $y_1, \ldots, y_n, x$ are range restricted too.

(b) the atomic formulas of ϕ and ψ are safe by hypothesis; in addition, the atomic formula $R(y_1, \ldots, y_k, \ldots, y_i, \ldots, y_n)$ is safe, since it contains new variables by the renaming of quantified variables and they are bounded, and for each y_i $(1 \leq j \leq k)$, there exists one $e_i \triangleleft y_i$; finally, the atomic formulas $e_1 \triangleleft y_1 \wedge \ldots \wedge e_k \triangleleft y_k \wedge y_i \triangleleft x$ are safe, since, by hypothesis, the variable x is bounded and $e_1, \ldots, e_k$ do not contain key variables; in addition, the variables $y_1, \ldots, y_k, y_i$ are also bounded and the variable y_i

is not a key variable.

The cases **(7)** and **(8)** are similar. Let us see the case **(7)**. It can be reasoned that:

(a) those range restricted c-terms in ϕ, ψ and $\mathcal{Q}$ by means of $f\ e_1 \ldots e_n \lhd x$, now they are range restricted by means of $f\ y_1 \ldots y_n \lhd x$, $e_1 \lhd y_1, \ldots, e_n \lhd y_n$; in addition, by hypothesis, the c-terms of $e_1, \ldots, e_k$ are range restricted and, thus the variables $y_1, \ldots, y_n$ are range restricted too.

(b) the atomic formulas of ϕ and ψ are safe by hypothesis; in addition, the atomic formula $f\ y_1 \ldots y_n \lhd x$ is safe, since $y_1, \ldots, y_n$ are new variables distinct from key variables and, by hypothesis, the variable x is bounded; finally, the atomic formulas $e_1 \lhd y_1, \ldots, e_n \lhd y_n$ are safe, since, by hypothesis, $e_1, \ldots, e_n$ do not contain key variables and the variables $y_1, \ldots, y_n$ are bounded.

Case **(9)**. It can be reasoned that:

(a) those range restricted c-terms in ϕ, ψ and $\mathcal{Q}$ by means of $t \lhd x$, now they are range restricted by means of $x = t$; in addition, by hypothesis, the c-terms of t and the variable x are range restricted;

(b) the atomic formulas of ϕ and ψ are safe by hypothesis; in addition, the atomic formula $x = t$ is safe, since, by hypothesis, x is a key variable and t contains no key variables.

Therefore, the calculus formulas ϕ and ψ are safe, the query $\mathcal{Q}$ is safe, and, finally, the calculus formula $\exists\bar{z}.\psi \wedge x = t$ is safe.

Case **(10)**. It can be reasoned that:

(a) The elimination of $t \lhd x$ does not affect to the range restricted condition since $x \notin formula\_key(\phi \wedge (\neg)\exists\bar{z}.\psi \wedge t \lhd x)$

(b) The formulas in $\phi \wedge (\neg)\exists\bar{z}.\psi\{x/t\}$ are safe since $t \lhd x$ is eliminated but $x \notin formula\_key(\phi \wedge (\neg)\exists\bar{z}.\psi \wedge t \lhd x)$.

■

Calculus and Algebra Equivalence

In this subsection, we will show two additional results in two Lemmas which will prove the *equivalence of the calculus and algebra transformation rules*, and are used in the proof of Theorem 6.2. From a non-formal point of view, the first result states the following: *once applied a given transformation rule, then*

- *each answer of* ϕ *and each tuple represented by* $\pi_{Proj}\ (\sigma_{Select}(\delta_{Ren}(\mathcal{R}_i \times \ldots \times \mathcal{R}_j)))$ *is a permutation of an answer of* ϕ^* *and a tuple represented by*

$\pi_{Proj^*}(\sigma_{Select^*}(\delta_{Ren^*}(\mathcal{R}_i^* \times \ \ldots \times \mathcal{R}_j^*)))$.
Now, we formally state this results in the following Lemma.

Lemma 10.3 (Answers in Calculus and Algebra Transformation Rules) Given a calculus and algebra transformation rule,

$$\frac{\phi \oplus \Delta \oplus (Rel|Select|Proj|Ren)}{\phi^* \oplus \Delta^* \oplus (Rel^*|Select^*|Proj^*|Ren^*)}$$

and let $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ be a database instance of an extended database schema $D = (S, DC, IF)$, then:

there exists $W_1 \times W_2$, which is a permutation of a tuple $(V_1, \ldots, V_n)$, such that:

- *$W_1 = \bar{y}\eta$ for a given substitution η, with $\bar{x}\eta \in Ans(\mathcal{D}, \phi)$ and $\bar{y} = \bar{x} \setminus dom(\Delta)$ (i.e. variables of $\bar{x}$ and not occurring in the domain of Δ)*; and
- *$W_2 = \pi_{Proj}(W_3)$ and $W_3 \in \sigma_{Select}(\delta_{Ren}(Rel))$*;

where, for each $x_i \in dom(\Delta)$, we have to consider the following:

- *if $\{x_i/A_i\} \in \Delta$, then $x_i\eta \in \|A_i\|^{\mathcal{D}}_{W_3}$*;
- *if $\{x_i/e_i\} \in \Delta$, then there exists a substitution η' such that $\eta' = \eta \circ \lambda$ and $x_i\eta \in \|e_i\|^{\mathcal{D}}\eta'$, for a given substitution λ*;

iff

there exists a tuple $W_1^* \times W_2^*$, which is a permutation $(V_1, \ldots, V_n)$, such that:

- *$W_1^* = \bar{z}\eta^*$ for a given substitution η^*, with $\bar{u}\eta^* \in Ans(\mathcal{D}, \phi^*)$ and $\bar{z} = \bar{u} \setminus dom(\Delta^*)$*; and, finally,
- *$W_2^* = \pi_{Proj^*}(W_3^*)$ and $W_3^* \in \sigma_{Select^*}(\delta_{Ren^*}(Rel^*))$*;

where, for each $u_i \in dom(\Delta^*)$, we have to consider the following:

- *if $\{u_i/A_i\} \in \Delta^*$, then $u_i\eta^* \in \|A_i\|^{\mathcal{D}}_{W_3^*}$*;
- *if $\{u_i/e_i\} \in \Delta^*$, then there exists a substitution η'' such that $\eta'' = \eta^* \circ \lambda^*$ and $u_i\eta^* \in \|e_i\|^{\mathcal{D}}\eta''$, for a given substitution λ^*.*

Proof

(1) $$\frac{\phi \wedge (\neg)\exists \bar{\mathsf{z}}.\exists \mathsf{y}_1 \ldots \exists \mathsf{y}_n.\psi \wedge \mathsf{R}_i(\mathsf{y}_1, \ldots, \mathsf{y}_n) \oplus \Delta \oplus (Rel|Select|Proj|Ren)}{\phi \wedge (\neg)\exists \bar{\mathsf{z}}.\psi \oplus \Delta \circ \{\mathsf{y}_j/\rho(\mathsf{A}_j)\} \oplus (Rel, \mathcal{R}_i|Select|Proj|Ren, \rho)}$$

Let us see the positive case. Assume a substitution η, such that $\bar{x}\eta \in Ans(\mathcal{D}, \phi \wedge \exists\bar{z}.\exists y_1. \ldots . \exists y_n.\psi \wedge R_i(y_1, \ldots, y_n))$. Then, there exists a substitution $\eta^* = \eta \circ \{z_k/s_k, y_j/l_j\}$ such that $(\mathcal{D}, \eta^*) \models_C R_i(y_1, \ldots, y_n)$ and, thus, $y_j\eta^* \in V_j\lambda$, where $\lambda \in Subst_{\perp,\mathsf{F}}$ and $(V_1, \ldots, V_n) \in \mathcal{R}_i$, which is a schema instance of the relation R_i. In this case, we have to prove that if y_j is free in $\phi \wedge \exists\bar{z}.\psi$, then $y_j\eta^* \in \|\rho(A_j)\|^{\mathcal{D}}_{W_3^*}$, since $\{y_j/\rho(A_j)\} \in \Delta^*$. However, y_j is free since, by the safety condition in the calculus formulas, there exists an equation $y_j = t_j$ in $\phi \wedge \exists \bar{z}.\psi$. Now, it is enough to consider $W_1^* = W_1$, $W_2^* = W_2$ and $W_3^* = (V_1, \ldots, V_n) \times W_3$, in such a way that $y_j\eta^* \in V_j\lambda \subseteq \llbracket \rho(A_j) \rrbracket^{\mathcal{D}}_{W_3^*}$, and $W_3^* \in \sigma_{Select}(\delta_{Ren^*}(Rel^*))$. In addition, $z\eta^* = z\eta$ if $z \not\equiv y_j$, and $z\eta^* = y_j\eta^*$ if $z \equiv y_j$, satisfying that $\bar{z}\eta^* \in Ans(\mathcal{D}, \psi)$ and $W_1 = W_1^* = \bar{u}\eta^*$ with $\bar{u} = \bar{z} \setminus dom(\Delta^*)$.

Let us see the negative case. It can be reasoned as previously, assuming $(W_1, \ldots, V_j, \ldots, W_n) \in \mathcal{R}_i$ such that $y_j\eta^* \in V_j\lambda$; and $W_3^* = (V_1, \ldots, V_n) \times W_3$.

$$\textbf{(2)}\ \frac{\phi \wedge (\neg)\exists \bar{\mathsf{z}}.\psi \wedge \mathsf{y_i} = \mathsf{t_i} \oplus \Delta \oplus (Rel|Select|Proj|Ren)}{\phi \wedge (\neg)\exists \bar{\mathsf{z}}.\psi \oplus \Delta^* \oplus (Rel|Select^*|Proj^*|Ren)}$$

- $\star$ ϕ *and* ψ *contain neither formulas* $\mathsf{R(y_1, \ldots, y_n)}$, *nor* $\mathsf{e \lhd x}$
- $\star$ $Decomp((\neg)\mathsf{y_i}\Delta = \mathsf{t_i}\Delta|Select|Proj|\Delta) = (Select^*|Proj^*|\Delta^*)$

Let us see the positive case. There exists a substitution $\eta^* = \eta \circ \{z_i/s_i\}$, such that $(\mathcal{D}, \eta^*) \models_C y_i = t_i$, and thus $y_i\eta^* = t_i\eta^*$. Now, by the safety condition, we have that $\{y_i/A_i\} \in \Delta$ and, in addition, y_i is free in $\phi \wedge \exists \bar{z}.\psi \wedge y_i = t_i$. Therefore, $y_i\eta^* = y_i\eta$ and $y_i\eta \in \llbracket A_i \rrbracket^{\mathcal{D}}_{W_3}$. Now, we need to distinguish cases by considering the form of $t_i\Delta$:

- $t_i\Delta$ is a variable z, and thus t_i is a variable u. In this case, we need to consider two subcases:
  - $\star$ u is free in $\phi \wedge \exists \bar{z}.\psi \wedge y_i = t_i$. In this case $u \equiv t_i$ and thus $u\eta^* = u\eta = y_i\eta$, ensuring that $u\eta \in \llbracket A_i \rrbracket^{\mathcal{D}}_{W_3}$. Therefore, we can take $W_1^* = (V_1, \ldots, V_{i-1}, V_{i+1}, \ldots, V_n)$ whenever $W_1 = (V_1, \ldots, V_i, \ldots, V_n)$ and $u\eta \in V_i\lambda$, $W_2^* = \pi_{Proj, y_i \overset{=}{\Delta}}(W_3^*)$, and $W_3^* = W_3 \times V_i$, in such a way that $W_1^* \times W_2^*$ and $W_1 \times W_2$ contain the same elements. In addition, by hypothesis (i.e. u is free in $\phi \wedge \exists \bar{z}.\psi \wedge y_i = t_i$), we have that $u\eta \in \llbracket \overset{=}{A_i} \rrbracket^{\mathcal{D}}_{W_3^*}$, where $\{u/\overset{=}{A_i}\} \in \Delta$;
  - $\star$ u is not free in $\phi \wedge \exists \bar{z}.\psi \wedge y_i = t_i$, then either u is a key or non-key variable or it occurs in an approximation equation $e \lhd u$. However, given that $u\Delta$ is variable, this contradicts the safety condition and the condition of the transformation rule.
- $t_i\Delta$ is an attribute B_i. Therefore $t_i \equiv u_i$ where u_i is a variable and, in addition, $\{u_i/B_i\} \in \Delta$. Then, on one hand, we have that $y_i\eta \in \llbracket A_i \rrbracket^{\mathcal{D}}_{W_3}$, where $\{y_i/A_i\} \in \Delta$, and, on the other hand, $u_i\eta \in \llbracket B_i \rrbracket^{\mathcal{D}}_{W_3}$ with $y_i\eta = u_i\eta$. Therefore, we have that $W_3 \models_A A_i = B_i$. Now, we can take $W_3^* = W_3$, $W_2^* = W_2$, where $W_3^* \in \sigma_{Select,\, y_i\Delta = t_i\Delta}(\delta_{Ren}(Rel))$, $W_2^* = \pi_{Proj}(W_3^*)$ and $W_1^* = W_1$, since, by the safety condition and the condition of the

transformation rule, u_i is none of the variables of $\bar{y}$.

- The rest of cases of $t_i\Delta$ can be proved by structural induction.

Let us see the negative case. It can be reasoned as before. In the case of $t_i\Delta$ is variable, $u\eta \notin [\![A_i]\!]^{\mathcal{D}}_{W_3}$. Assuming $[\![A_i]\!]^{\mathcal{D}}_{W_3} = V_i$ then we can take as before $W_1^* = (V_1, \ldots, V_{i-1}, V_{i+1}, \ldots, V_n)$, $W_3^* = W_3 \times V_i$ in such a way that $W_2^* = \pi_{\substack{\neq \\ Proj, y_i\Delta}}(W_3^*)$ and $W_1^* \times W_2^*$ and $W_1 \times W_2$ contain the same elements. The case of $t_i\Delta$ is an attribute can be reasoned as before proving that $W_3 \models_A A_i \neq B_i$.

(3) $\dfrac{\phi \wedge (\neg)\exists \bar{\mathsf{z}}.\exists \mathsf{x}.\psi \ \wedge \ \mathsf{e} \lhd \mathsf{x} \ \oplus \ \Delta \ \oplus \ (Rel|Select|Proj|Ren)}{\phi \ \wedge \ (\neg)\exists \bar{\mathsf{z}}.\psi \ \oplus \ \Delta \circ \{\mathsf{x}/\mathsf{e}\} \ \oplus \ (Rel|Select|Proj|Ren)}$

$\star$ *ϕ and ψ contain no formulas* $\mathsf{R}(\mathsf{y}_1, \ldots, \mathsf{y}_n)$

Let us see the positive case. Let η be a substitution such that $\bar{x}\eta \in Ans(\mathcal{D}, \exists\bar{z}.\exists x. \psi \wedge e \lhd x)$. Then, we can consider a substitution $\eta^* = \eta \circ \{x/s, z_i/s_i\}$ such that $(\mathcal{D}, \eta^*) \models_C e \lhd x$. Therefore, $x\eta^* \in [\![e]\!]^{\mathcal{D}}\eta^*$. Now, it is enough to consider $W_1^* = W_1$, $W_2^* = W_2$, $W_3^* = W_3$, and $u\eta^{**} = u\eta^*$ if u occurs in e but is not free in $\exists\bar{z}.\psi$. With this choice, $x\eta^* \in [\![e]\!]^{\mathcal{D}}\eta^{**}$ if $\{x/e\} \in \Delta$.

Let us see the negative case. It can be reasoned as before, where η^* must be taken such that $x\eta^* \in [\![e]\!]^{\mathcal{D}}\eta^*$; it ensures the result since, by the safety condition, x is not free in $\psi \wedge \neg\exists.\bar{z}.\phi$, and therefore $\eta^* = \eta|_{free(\psi \wedge \neg\exists.\bar{z}.\phi) \cup var(\mathcal{Q})}$.

The cases **(4)** and **(5)** are similar. The negative case is similar also to the positive one. Let us see the positive case.

(4) $\dfrac{\phi \wedge (\neg)\exists \bar{\mathsf{z}}.\psi \ \wedge \ \mathsf{t}_1 \Downarrow \mathsf{t}_2 \ \oplus \ \Delta \ \oplus \ (Rel|Select|Proj|Ren)}{\phi \ \wedge \ (\neg)\exists \bar{\mathsf{z}}.\psi \ \oplus \ \Delta^* \ \oplus \ (Rel|Select^*|Proj^*|Ren)}$

$\star$ *ϕ and ψ contain neither formulas* $\mathsf{R}(\mathsf{y}_1, \ldots, \mathsf{y}_n)$, $\mathsf{e} \lhd \mathsf{x}$, *nor* $\mathsf{y} = \mathsf{t}$
$\star$ $\mathsf{t}_1\Delta$ *or* $\mathsf{t}_2\Delta$ *contains no free variables in* $\phi \wedge (\neg)\exists\bar{\mathsf{z}}.\ \psi \ \wedge \ \mathsf{t}_1 \Downarrow \mathsf{t}_2$
$\star$ *Decomp*$((\neg)\mathsf{t}_1\Delta \bowtie \mathsf{t}_2\Delta|Select|Proj|\Delta) = (Select^*|Proj^*|\Delta^*)$

Let η be a substitution, such that $\bar{x}\eta \in Ans(\mathcal{D}, \exists\bar{z}.\psi \wedge t_1 \Downarrow t_2)$. Then, we can find a substitution $\eta^* = \eta \circ \{z_i/s_i\}$ such that $(\mathcal{D}, \eta^*) \models_C t_1 \Downarrow t_2$. This means $t_1\eta^* \downarrow t_2\eta^*$ and $t_1\eta^*, t_2\eta^* \in adom(t_1, \mathcal{D}) \cup adom(t_2, \mathcal{D})$. Now, we can distinguish the following cases for $t_1\Delta$ (similarly with $t_2\Delta$):

- $t_1\Delta$ is a variable y. Then, we have that t_1 is a variable u. In this case, $t_2\Delta$ contains no free variables. Otherwise, it contradicts the safety condition and the condition of the transformation rule. Therefore, since $x_i\eta \in [\![A_i]\!]^{\mathcal{D}}_{W_3}$ for each $\{x_i/A_i\} \in \Delta$ with x_i variable of t_2, then we have that $t_2\eta = t_2\eta^* \in [\![t_2\Delta]\!]^{\mathcal{D}}_{W_3}$, and, by hypothesis, $t_1\eta^* \in adom(t_1, \mathcal{D}) \cup$

$adom(t_2, \mathcal{D})$ and $t_2\eta^* \in adom(t_1, \mathcal{D}) \cup adom(t_2, \mathcal{D})$. In addition, $t_1\eta^* \downarrow t_2\eta^*$ is satisfied, therefore $t_1\eta^* \in \| \overset{\bowtie}{t_2\Delta} \|^{\mathcal{D}}_{W_3}$. On the other hand, since t_1 is a variable, by the safety condition, we can reason that it is free; thus $t_1\eta^* = u\eta^* \in V_i$ where $W_1 = (V_1, \ldots, V_i, \ldots, V_n)$. Now, as previous cases, it is enough to consider $W_1^* = (V_1, \ldots, V_{i-1}, V_{i+1}, \ldots, V_n)$, where $W_2^* = \pi_{Proj,\, \overset{\bowtie}{t_2\Delta}}(W_3^*)$, and $W_3^* = W_3 \times V_i$, where $W_1^* \times W_2^*$ and $W_1 \times W_2$ contain the same elements; in addition, we have that $u\eta^* \in \| \overset{\bowtie}{t_2\Delta} \|^{\mathcal{D}}_{W_3^*}$ and $\{u / \overset{\bowtie}{t_2\Delta}\} \in \Delta^*$;

- $t_1\Delta$ is an attribute name, for instance, B. Now, we have that $t_1 \equiv u$, where u is a variable, and $\{u/B\} \in \Delta$. In addition, we have, by hypothesis, that $u\eta \in \|B\|^{\mathcal{D}}_{W_3}$. On the other hand, $t_1\eta \equiv u\eta$ and $u\eta \downarrow t_2\eta^*$; as previously, t_2 contains no free variables, and thus $t_2\eta^* \in \|t_2\Delta\|^{\mathcal{D}}_{W_3}$. Therefore, we have that $W_3 \models_A t_1\Delta \bowtie t_2\Delta$, and taking $W_1^* = W_1$, $W_2^* = W_2$ and $W_3^* = W_3$, we conclude the result.
- The rest of cases of $t_1\Delta$ can be proved by structural induction.

■

Now, the additional second result states the following: *once applied a given transformation rule, then*

- *ϕ is a safe calculus formula and π_{Proj} $(\sigma_{Select}(\delta_{Ren}(\mathcal{R}_i \times \ldots \times \mathcal{R}_j)))$ is a closed algebra expression iff ϕ^* is a safe calculus formula and $\pi_{Proj^*}(\sigma_{Select^*}$ $(\delta_{Ren^*}(\mathcal{R}_i^* \times \ \ldots \times \mathcal{R}_j^*)))$ is a closed algebra expression*;

where the Definitions 4.5 (*Safe Calculus Formulas*) and 5.7 (*Algebra Expressions*), which state the safety conditions for the calculus formulas and the closed conditions for the algebra expressions, respectively, are modified as follows:

(a) *Safe Calculus Formula* (Definition 4.5) is replaced by:

(1) all the c-terms and atomic formulas occurring in ϕ are range restricted and safe, respectively; and,

(2) the only bounded variables occurring in ϕ are variables of *formula_key*(ϕ) $\cup$ *formula_nonkey*(ϕ) $\cup$ *approx*(ϕ), or variables of $Dom(\Delta)$.

Now, the definitions of the safety conditions for the calculus formulas (i.e. *range restricted in calculus formulas* and *safe atomic formulas*) are modified as follows:

(a.1) *Range Restricted C-Terms of Calculus Formulas* (Definition 4.4) is replaced as follows:

(1) t occurs in *formula_key*(ϕ) $\cup$ *formula_nonkey*(ϕ), or $\{t/A\} \in \Delta$ where

$A \in Key(\delta_{Ren}(Rel)) \cup NonKey(\delta_{Ren}(Rel))$;

(2) there exists one equation $e \diamondsuit_c e'$ ($\diamondsuit_c \in \{=, \Uparrow, \Downarrow, \triangleleft\}$) in ϕ, such that t belongs to $cterms(e)$ (resp. $cterms(e')$) and every c-term of e' (resp. e) is range restricted in ϕ.

(a.2) *Safe Atomic Formulas* (Definition 4.3) is replaced as follows:

(1) $R(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$ is safe, if the variables $x_1, \ldots, x_n$ are bound in ϕ, or they are free variables with $\{x_i/A_i\} \in \Delta$ and $A_i \in Key(\delta_{Ren}(Rel)) \cup NonKey(\delta_{Ren}(Rel))$; in addition, for each x_i, $i \leq nKey(R)$, there exists one equation $x_i = t_i$ in ϕ, or $\{x_i/A_i\} \in \Delta$ where $A_i \in Key(\delta_{Ren}(Rel))$;

(2) $x = t$ is safe, if the variables occurring in t are distinct from the variables of $formula\_key(\phi)$, and x is a variable of $formula\_key(\phi)$ or $\{x/A\} \in \Delta$ where $A \in Key(\delta_{Ren}(Rel))$;

(3) $t \Downarrow t'$ and $t \Uparrow t'$ are safe, if the variables occurring in t and t' are distinct from the variables of $formula\_key(\phi)$ and $\{y \mid y\Delta \in Key(\delta_{Ren}(Rel))\}$;

(4) $e \triangleleft x$ is safe, if either the variables occurring in e are distinct from the variables of $formula\_key(\phi)$ and $\{y \mid y\Delta \in Key(\delta_{Ren}(Rel))\}$; and x is bounded in ϕ, or x is free in ϕ and $\{x/e\} \in \Delta$.

(b) *Algebra Expressions* (Definition 5.7) is replaced by:

(1) Ψ must be *closed w.r.t. key values*; that is, $Key(\Psi) \cup \{y_i\Delta \mid \{y_i/A_i\} \in \Delta,\ y_i\ free\ in\ \phi\ and\ A_i \in Key(\delta_{Ren}(Rel))\} = \bigcup_{R \in Rel(\Psi)} Key(\delta_{Ren}(Rel))$, where $Key(\Psi)$ and $Rel(\Psi)$ represent the set of key attribute names and relation names occurring in Ψ, respectively;

(2) Ψ must be *closed w.r.t. data destructors and function inverses*; that is, whenever $\pi_{\substack{\diamondsuit_a \\ c.index(e)}}$ or $\sigma_{c.index(e)\diamondsuit_a e^*}$ (resp. $\pi_{\substack{\diamondsuit_a \\ f.index(e)}}$ or $\sigma_{f.index(e)\diamondsuit_a e^*}$) occurs in Ψ, then $\pi_{\substack{\diamondsuit_a \\ c.i(e)}}$ or $\sigma_{c.i(e)\diamondsuit_a e^*}$ (resp. $\pi_{\substack{\diamondsuit_a \\ f.i(e)}}$ or $\sigma_{f.i(e)\diamondsuit_a e^*}$) must occur in Ψ, for every $1 \leq i \leq n$ with $c \in DC^n$ (resp. $f \in IF^n$).

Note that the new safety and closed conditions coincide with the original ones (i.e. Definitions 4.3, 4.4 and 4.5 for calculus formulas, and Definition 5.7 for algebra expressions), whenever $\Delta = id$ and $\phi = \emptyset$, respectively. Finally, the previous mentioned safety conditions and the rule conditions expressed in transformation rules **(4)** and **(5)** (i.e. $t_1\Delta$ *or* $t_2\Delta$ *contains no free variables*), allow us to progress in the transformation. As previously, we formally state this result in the following Lemma.

Lemma 10.4 (Safety in Calculus and Algebra Transformation Rules) Given a calculus and algebra transformation rule,

$$\frac{\phi \oplus \Delta \oplus (Rel|Select|Proj|Ren)}{\phi^* \oplus \Delta^* \oplus (Rel^*|Select^*|Proj^*|Ren^*)}$$

and let $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ be a database instance of an extended database schema $D = (S, DC, IF)$, then:

ϕ is a safe calculus formula and $\pi_{Proj}(\sigma_{Select}(\delta_{Ren}(Rel)))$ is a closed algebra expression,
iff
ϕ^* is a safe calculus formula and $\pi_{Proj^*}(\sigma_{Select^*}(\delta_{\ Ren^*}(Rel^*)))$ is a closed algebra expression.

Proof
Case **(1)**. It can be reasoned that:

(a)

(a.1) those range restricted c-terms in ϕ and ψ by the variables $y_1, \ldots, y_n$, now they are range restricted by means of Δ^*, since $\Delta^* = \Delta \circ \{y_i/\rho(A_i)\}$ and $\rho(A_i) \in Key(\delta_{Ren^*}(Rel^*)) \cup NonKey(\delta_{Ren^*}(Rel^*))$; the rest of c-terms occurring in ϕ and ψ are range restricted by hypothesis.

(a.2) those atomic formulas in ϕ and ψ that are safe by the variables $y_1, \ldots, y_n$, now they are safe by Δ^*; the rest of atomic formulas occurring in ϕ and ψ are safe by hypothesis.

Therefore, the calculus formulas ϕ and ψ are safe, since the c-terms and atomic formulas occurring in ϕ and ψ are range restricted and safe, respectively, and, in addition, $\Delta^* = \Delta \circ \{y_i/\rho(A_i)\}$ where $\rho(A_i) \in Key(\delta_{Ren^*}(Rel^*)) \cup NonKey(\delta_{Ren^*}(Rel^*))$.

(b) $\Psi \equiv \pi_{Proj}(\sigma_{Select}(\delta_{Ren^*}(Rel^*)))$ is closed w.r.t. key values, since $\Delta^* = \Delta \circ \{y_i/\rho(A_i)\}$ where $\rho(A_i) \in Key(\delta_{Ren^*}(Rel^*)) \cup NonKey(\delta_{Ren^*}\ (Rel^*))$; in addition, Ψ is closed w.r.t. data destructors and function inverses by hypothesis.

Case **(2)**. It can be reasoned that:

(a)

(a.1) the c-terms occurring in ϕ and ψ are range restricted by hypothesis;

(a.2) the atomic formulas occurring in ϕ and ψ are safe by hypothesis;

Therefore, ϕ and ψ are safe by hypothesis; in fact, the elimination of $y_i = t_i$ does not affect the safety and range restricted conditions, since both ϕ and ψ do not contain formulas $R(y_1, \ldots, y_n)$ or $e \lhd x$;

(b) In this case, since $y_i = t_i$ is safe and by the condition of the rule (i.e. ϕ and ψ contain neither formulas $R(y_1, \ldots, y_n)$ nor $e \lhd x$), then $\{y_i/A_i\} \in \Delta$ with $A_i \in Key(\delta_{Ren}(Rel))$; now, we need to consider the following cases w.r.t. the form of $t_i\Delta$:

- ⋆ a variable, then $\Psi \equiv \pi_{Proj,\, \overset{=}{y_i\Delta}}(\sigma_{Select}(\delta_{Ren}(Rel)))$ is closed w.r.t. key values, since $\{y_i/A_i\} \in \Delta$ with $A_i \in Key(\delta_{Ren}(Rel))$ and $t_i\Delta$ does not affect the key variables; in addition, Ψ is closed w.r.t. data destructors and function inverses by hypothesis.
- ⋆ an attribute name, then $\Psi \equiv \pi_{Proj}(\sigma_{Select,\, y_i\Delta = t_i\Delta}(\delta_{Ren}(Rel)))$ is a closed expression w.r.t. key values, since $\{y_i/A_i\} \in \Delta$ with $A_i \in Key(\delta_{Ren}(Rel))$; in addition, t_i contains neither key variables x_i, nor Δ includes substitutions of the form $\{x_i/A_i\}$ with A_i key attribute; finally, Ψ is closed w.r.t. data destructors and function inverses by hypothesis.
- ⋆ $c(e_1, \ldots, e_n)$, with $c \in DC^n$, and $var(e_1, \ldots, e_n) = \emptyset$, then $\Psi \equiv \pi_{Proj}$ $(\sigma_{Select,\, y_i\Delta = c(e_1,\ldots,e_n)}(\delta_{Ren}(Rel)))$ is a closed algebra expression w.r.t. key values, since $\{y_i/A_i\} \in \Delta$ with $A_i \in Key(\delta_{Ren}(Rel))$; in addition, t_i contains neither key variables x_i, nor Δ includes substitutions of the form $\{x_i/A_i\}$ with A_i key attribute; finally, Ψ is closed w.r.t. data destructors and function inverses by hypothesis.
- ⋆ The case $f\ e_1\ \ldots\ e_n$, with $f \in IF^n$, and $var(e_1, \ldots, e_n) = \emptyset$ is similar.
- ⋆ $c(e_1, \ldots, e_n)$ (with $c \in DC^n$) or $f\ e_1\ \ldots\ e_n$ (with $f \in IF^n$), and $var(e_1, \ldots, e_n) \neq \emptyset$, then we have to check the data destructors and function inverses; that is, $c.i(y_i\Delta) = e_i$ and $f.i(y_i\Delta) = e_i$ with $1 \leq i \leq n$. Now, as previously, we need to consider the following subcases:
  - – e_i is a variable, then $Proj^* = Proj, c.i(\overset{=}{y_i}\Delta)$ or $Proj^* = Proj, f.i(\overset{=}{y_i}\Delta)$; otherwise, $Select^* = Select, c.i(y_i\Delta) = e_i$ or $Select^* = Select, f.i(y_i\Delta) = e_i$ with $1 \leq i \leq n$; therefore, $\Psi \equiv \pi_{Proj^*}(\sigma_{Select^*}(\delta_{Ren}(Rel)))$ is a closed algebra expression w.r.t. data destructors and function inverses; finally, Ψ is a closed algebra expression w.r.t. key values, since $\{y_i/A_i\} \in \Delta$ with $A_i \in Key(\delta_{Ren}(Rel))$;
  - – The rest of cases can be proved by structural induction.

Case **(3)**. It can be reasoned that:

(a)

(a.1) those range restricted c-terms in ϕ and ψ by the variable x, now they are range restricted by means of Δ^*, since $\Delta^* = \Delta \circ \{x/e\}$;

(a.2) those atomic formulas in ϕ and ψ that are safe by the variable x, now they are safe by Δ^*;

Therefore, the calculus formulas ϕ and ψ are safe, since the c-terms and atomic formulas occurring in ϕ and ψ are range restricted and safe, respectively; in addition, the elimination of $\exists x.e \lhd x$ does not affect the safety condition, since $\Delta^* = \Delta \circ \{x/e\}$;

(b) $\Psi \equiv \pi_{Proj}(\sigma_{Select}(\delta_{Ren}(Rel)))$ is a closed algebra expression w.r.t. key values, and data destructors and function inverses by hypothesis.

The cases **(4)** and **(5)** are similar. Let us see the case **(4)**. It can be reasoned that:

(a)

(a.1) the c-terms occurring in ϕ and ψ are range restricted by hypothesis;

(a.2) the atomic formulas occurring in ϕ and ψ are safe by hypothesis;

Therefore, ϕ and ψ are safe by hypothesis; in fact, the elimination of $t_1 \Downarrow t_2$ does not affect the safety and range restricted conditions, since both ϕ and ψ do not contain formulas $R(y_1, \ldots, y_n)$, $e \lhd x$, or $y = t$;

(b) In this case, given that $t_1 \Downarrow t_2$ is safe, the c-terms of t_1 and t_2 are range restricted, and the condition of the rule (i.e. ϕ and ψ contain neither formulas $R(y_1, \ldots, y_n)$, $e \lhd x$, nor $y = t$), then we need to consider the following cases:

- ⋆ the variables of t_1 are variables x_i such that $\{x_i/A_i\} \in \Delta$, where $A_i \in Key(\delta_{Ren}(Rel)) \cup NonKey(\delta_{Ren}(Rel))$; now, we have to consider the following subcases w.r.t. the form of $t_2\Delta$:
  - a variable, then: $\Psi \equiv \pi_{\underset{Proj,\, t_1\Delta}{\bowtie}}(\sigma_{Select}(\delta_{Ren}(Rel)))$ is a closed algebra expression w.r.t. key values, since $\{x_i/A_i\} \in \Delta$, where $A_i \in Key(\delta_{Ren}(Rel)) \cup NonKey(\delta_{Ren}(Rel))$, and $t_2\Delta$ do not affect the key variables; in addition, Ψ is a closed algebra expression w.r.t. data destructors and function inverses by hypothesis;
  - a key attribute name A_j, then: $\Psi \equiv \pi_{Proj}(\sigma_{Select,\, t_1\Delta \bowtie t_2\Delta}(\delta_{Ren}$ $(Rel)))$ is a closed algebra expression w.r.t. key values, since $\{x_i/A_i\}$ $\in \Delta$, where $A_i \in Key(\delta_{Ren}(Rel)) \cup NonKey(\delta_{Ren}(Rel))$, and, by the

condition of the rule, $\{t_2/A_j\} \in \Delta$ with $A_j \in Key(\delta_{Ren}\ (Rel))$; in addition, Ψ is a closed algebra expression w.r.t. data destructors and function inverses by hypothesis;

– a non-key attribute name A_j, then: $\Psi \equiv \pi_{Proj}(\sigma_{Select,\, t_1\Delta \bowtie t_2\Delta}(\delta_{Ren}\ (Rel)))$ is a closed algebra expression w.r.t. key values, since $\{x_i/A_i\} \in \Delta$, where $A_i \in Key(\delta_{Ren}(Rel)) \cup NonKey(\delta_{Ren}(Rel))$, and $t_2\Delta$ does not affect the key variables; in addition, Ψ is a closed algebra expression w.r.t. data destructors and function inverses by hypothesis;

– $c(e_1, \ldots, e_n)$, with $c \in DC^n$, and $var(e_1, \ldots, e_n) = \emptyset$, then $\Psi \equiv \pi_{Proj}(\sigma_{Select,\, t_1\Delta \bowtie c(e_1,\ldots,e_n)}(\delta_{Ren}(Rel)))$ is a closed algebra expression w.r.t. key values, since $\{x_i/A_i\} \in \Delta$, where $A_i \in Key(\delta_{Ren}\ (Rel)) \cup NonKey(\delta_{Ren}(Rel))$, and $t_2\Delta$ does not affect the key variables; in addition, Ψ is a closed algebra expression w.r.t. data destructors and function inverses by hypothesis;

– The case $f\ e_1 \ldots e_n$, with $f \in IF^n$, and $var(e_1, \ldots, e_n) = \emptyset$ is similar.

– $c(e_1, \ldots, e_n)$ (with $c \in DC^n$) or $f\ e_1 \ldots e_n$ (with $f \in IF^n$), and $var(e_1, \ldots, e_n) \neq \emptyset$, then we have to check the data destructors and function inverses; that is, $c.i(t_1\Delta) \bowtie e_i$ or $f.i(t_1\Delta) \bowtie e_i$, with $1 \leq i \leq n$. Now, we have the following cases:

  · e_i is a variable, then $Proj^* = Proj,\ c.i(\overset{\bowtie}{t_1}\Delta)$ or $Proj^* = Proj,\ f.i(\overset{\bowtie}{t_1}\Delta)$; otherwise $Select^* = Select, c.i(t_1\Delta) \bowtie e_i$ or $Select^* = Select, f.i(t_1\Delta) \bowtie e_i$, with $1 \leq i \leq n$; therefore, $\Psi \equiv \pi_{Proj^*}(\sigma_{Select^*}\ (\delta_{Ren}(Rel)))$ is a closed algebra expression w.r.t. data destructors and function inverses; finally, Ψ is a closed algebra expression w.r.t. key values, since $\{x_i/A_i\} \in \Delta$, where $A_i \in Key(\delta_{Ren}\ (Rel)) \cup NonKey(\delta_{Ren}(Rel))$;

  · the rest of cases can be proved by structural induction.

⋆ the variables of t_2 are variables x_i such that $\{x_i/A_i\} \in \Delta$, where $A_i \in Key(\delta_{Ren}(Rel)) \cup NonKey(\delta_{Ren}(Rel))$; analogously to previous case.

⋆ t_1 and t_2 contain variables not belonging to Δ; however, it contradicts the condition of the transformation rule, where $t_1\Delta$ or $t_2\Delta$ contains no free variables of $\phi \wedge \exists \bar{z}, \psi \wedge t_1 \Downarrow t_2$.

■