$\mathcal{INDALOG}$: A Declarative Deductive Database Language

Jesús M. Almendros-Jiménez* and Antonio Becerra-Terón
Dpto. de Lenguajes y Computación. Universidad de Almería.
email: {jalmen,abecerra}@ual.es

Abstract. In this paper we present the main features of a deductive database language named $\mathcal{INDALOG}$ based on the integration of functional and logic paradigms. As most deductive database systems, $\mathcal{INDALOG}$ allows the handling of negation, higher-order functions, grouping operators, support for aggregation, handling of non-ground facts, and support for indexing structures on both extensional and intensional relations of a database. Moreover, we present the semantic foundations of this language.

1 Introduction

Deductive database systems [16] are database management systems whose query language and storage structure are designed around a logic data model. Deductive database systems offer a rich query language which extends the relational model in many directions (for instance, support for non-first normal form and recursive relations) and they are suited for application in which a large number of data must be accessed and complex queries must be supported (see [16] for applications of deductive systems). With respect to the information management, the deductive database systems split their information into two categories:

- *Facts* represented by *literals* consisting of a predicate applied to *terms*, which are constants, variables, or functors applied to terms. For instance, the fact *parent(mary,peter)* sets that Peter is parent of Mary. This predicate is represented *extensionally*, that is, all tuples for this predicate are stored in a database relation.
- *Rules* which are written in Prolog-style notation as follows $p :- q_1, ..., q_n.$ where p (the *head*) and q_i's (the *subgoal*s) are literals.

The rules are refereed as the *intensional database* (IDB) and the facts as *extensional database* (EDB). The intensional database plays a role similar to views in conventional database systems, although there will be large numbers of intensional predicates w.r.t. the numbers of views defined in typical database applications. A typical example about the expressivity power of deductive databases is as follows:

Extensional Database

```
parent(john,mary).        parent(john,frank).
parent(mary,thomas).      parent(frank,michael).
```

Intensional Database

```
anc(X,Y) :- parent(X,Y).      anc(X,Y) :- parent(X,Z), anc(Z,Y).
```

Query

```
:- anc(john,Y).
```

* The author has been partially supported by the Spanish CICYT (project TIC 98-0445-C03-02 TREND)

Here, there exist two relations, one extensional relation defined by four rules and called `parent`, and other intensional one, named `anc` and defined by a recursive rule. The proposed query will compute all john's ancestors, that is `Y = mary`, `Y = frank`, `Y = thomas`, and `Y = michael`.

The relations of a deductive database can include functors to store the so-called *complex terms*. In a logic programming context, the complex terms are called *functions* because of their syntactic apparency, consisting of a function name (named the *functor*) followed by a list of arguments. In a deductive language, these complex terms are not used to represent functions; rather, they are used as *variable-length subrecords*. For instance, suppose the following complex term `address(Name,City)` which can be used in the base relation `person(DNI,address (Name, City),Job)` in order to structure the stored information.

Most deductive database systems (for instance, *DATALOG* [20], *CORAL* [14], *ADITI* [21], *LOLA* [23]) allow to handle negative literals, increasing their expressive power as query languages. The introduction of negation in logic programming (see [4] for a survey), and thus the study of semantic models for logic programs have been widely studied in the past being most relevant one the *well-founded semantics* [22]. Some deductive database systems have adopted such semantics and operational models [17, 9] based on this semantics. For instance, in the deductive system *DATALOG*, we can write the following database:

Extensional Database

`student(peter, senior).` `took(jim, cs2, 3.3).`
`student(jim, junior).` `took(jim, cs3, 3.0).`
`student(mary, junior).` `took(mary, cs1, 2.0).`
`took(peter, cs1, 3.0).` `took(mary, cs2, 3.0).`

Intensional Database

`hastaken(Name, Course) :- took(Name, Course, Grade).`
`lacks_cs3(Name) :- student(Name, junior), ¬hastaken(Name, cs3).`

Query

`:- lacks_cs3(N).`

Here, the extensional relations `student` and `took` set the name and year of a student, as well as the courses taken by the student with the corresponding grade. The intensional rules are meant to compute junior students who did not take course cs3. The computed answer will be `N = mary`.

In addition, most deductive database systems allow *grouping* and *aggregation* operators to be used with groups of tuples. The idea of grouping operators is to collect the set (multiset) of values for a given attribute and, in the case of aggregation operators, to manage the collected values. For instance, the deductive database system *CORAL* [14, 13] allows to collect values in the form of sets and multisets by means of set ({}) and multiset (<>) grouping constructors. The following rule illustrates the use of the multiset construct:

```
set_of_grades(Class, < Grade >) :- student(Name, Class),
                                   took(Name, Course, Grade).
```

wherein for each `Class` value, a multiset containing all corresponding values for `Grade` is created. Similarly with the set-grouping construct. The use of these grouping primitives can be combined with aggregation operators like `count`, `min`, `max`, `sum`, `product`, `average`, `any` and so on. In *CORAL* we can write the following rule for computing the average grade:

```
max_grade(Class, average(< Grade >)) :- student(Name, Class),
                                        took(Name, Course, Grade).
```

Some of the deductive database systems (for instance, *XSB* [18]) enrich their expressivity power by adding higher-order features. The deductive database system *XSB* adopts *HiLog* [6], a foundation for higher order logic programming, by providing an elegant way to construct and manipulate sets. For instance, the following *XSB* database:

ExtensionalDatabase

```
package1(health_ins, required).      package2(long_vacations, optional).
package1(life_ins, optional).        benefits(john, package1).
package2(health_ins, required).      benefits(bob, package2).
package2(free_car, optional).
```

Query

```
:−benefits(john, P), P(X, Y).
```

defines the extensional relations `package1` and `package2` used to denote the set of john's and bob's benefits, respectively. Benefits are a set of facts indicating the type of benefit and whether it is optional or required. The query binds `P` to the name of the set of john's benefits (`P = package1`), and then retrieves the facts that describe his benefits explicitly, that is `X = health_ins Y = required` and `X = life_ins Y = optional`. Also, this representation can be extended to include set operations. For instance, the following rule computes the intersection of two sets and the query allows to obtain both john's and bob's benefits.

Intensional Database

```
intersect(S1, S2)(X, Y) :−S1(X, Y), S2(X, Y).
```

Query

```
:−benefits(john, P1), benefits(bob, P2), intersect(P1, P2)(X, Y).
```

Following with the features of deductive database systems, there exist systems which allow to use non-ground facts. The use of non-ground facts (i.e. facts containing universally quantified variables) is useful in the deductive database context. The deductive system *CORAL* [19] supports efficiently the handling of non-ground facts. For instance, suppose a database about employee salary:

Extensional Database

```
base_salary(john, 1500).      base_salary(marc, 2500).
base_salary(peter, 3500).     complement_salary(X, 1000).
```

wherein there exists one non-ground fact, `complement_salary(X, 1000)`, expressing that all employees will have a salary complement (1000 euros), and thus avoiding the specification of one fact for each employee. Now, we can compute the total salary by means of the following rule:

Intensional Database

```
total_salary(X, Y) :−base_salary(X, T), complement_salary(X, Z), Y = T + Z.
```

With respect to the implementation, the deductive database systems provide a representation of indices. Most deductive database systems use the typical indexing structures used in database applications, like hash and B-trees indices on the extensional and intensional database relations. By means of adding indices, some database operations, like *join*, can be efficiently made for disk-resident data. For instance, *XSB* [18] supports different kinds of indexing. The default is hashing on the first argument of a relation. However, declarations can be used to indicate the desired options. Indices can be constructed on any attribute or set of attributes (multi-attribute indexing). For example, a predicate `p` with arity 5 could have the following index declaration:

```
:−index(p/5, [1, 2, 3 + 5]).
```

which will cause indices on `p/5` in such a way that a retrieval will use the index

on the first argument, if *ground*, otherwise on the second, if *ground*, and otherwise on the third and fifth combined.

Other example of indexing is the deductive language *CORAL* [15] which supports hash-based indices for memory relations and B-trees indices for disk relations. It allows two types of hash-based indices, (1) *argument indices* with the traditional multi-attribute hash index and (2) *pattern indices* which is an index structure more complicated since that such indices are used with complex terms and non-ground facts. For instance, given a relation `employee` with two arguments, `Name` and a complex term `address(Street, City)` in *CORAL*, then the following declaration creates a pattern index to retrieve employees without knowing their street.

`@make_indexemployee(Name, address(Street, City))(Name, City).`

Here, we are interested in the study of a deductive database system based on the integration of the functional and logic paradigms. The integration of functional and logic programming has been widely investigated during the last years. It has led to the design of modern programming languages such as *CURRY* [8] and *TOY* [10]. The aim of such integration is to include features from functional (cfr. determinism, higher order functions, partial, non-strict and lazy functions, possibly infinite data structures) and logic (cfr. logic variables, function inversion, non-determinism, built-in search) languages.

Our idea is to develop a deductive database query language, named $\mathcal{INDALOG}$, which includes these advantages and also allows the efficient management of data like in deductive databases based on logic programming.

The main difference with respect to a "pure" functional-logic language, such as *CURRY* or *TOY*, is that $\mathcal{INDALOG}$ is thought for working with a large volume of data which usually will be stored in secondary memory. Like Prolog, functional-logic languages work efficiently with main-memory resident data, but inefficiently with respect to the disk accesses when data are stored in secondary memory. The main reason for this drawback is that the operational mechanism of such languages works with a tuple (fact) at time, whereas the deductive database systems allow a set of tuples at time.

The basic idea of the operational mechanism of deductive database languages is to use a *bottom-up* evaluation for query solving. This bottom-up mechanism is based on the application of the *immediate consequence operator* defined for logic programs which allows to compute the *Herbrand model* of a logic program. The use of this operator for query solving is very inefficient and thus a program transformation techniques, called *Magic Sets*-based transformations [5, 12], have been studied. The aim of this process is to transform the original program w.r.t. the proposed query in such a way that the application of the immediate consequence operator on the transformed program and query is *goal-oriented* and can be used for query solving. Moreover, this evaluation mechanism is more set-of-tuples-oriented than top-down evaluation mechanism supported by logic languages and therefore adequate for indexing of tuples. In our case, we have proposed a bottom-up evaluation mechanism for functional-logic programming in [1], which has been extended in [2] for the handling of negation in the line of [11], to be considered as the computational model for $\mathcal{INDALOG}$.

The aim of this paper is to present the main features already developed and to be developed in our deductive database language $\mathcal{INDALOG}$. As most deductive database systems, this language will allow the handling of negation, higher-order

functions, grouping operators, support for aggregation, handling of non-ground facts, and support for indexing structures on both extensional and intensional relations of a database. Moreover, we will present the semantic foundations of $\mathcal{INDALOG}$.

2 INDALOG Features

In this section, we present the main features in our deductive language. An $\mathcal{INDALOG}$ deductive database $\mathcal{DB}$ will be constituted by the following set of modules.

- *type definition* which includes the definition of type symbols.
- *base relations* which includes the definition of each extensional relation of the database. Also called *base facts*, typically they will be indexed and stored in secondary memory (see subsection 2.4 for more details).
- *derived relations* which includes the definition of each intensional relation of the database. Facts obtained from derived relations will be also indexed and stored at run-time in secondary memory.
- *function definition* which is a set of manipulating functions to be used in an $\mathcal{INDALOG}$ database which they are neither base nor derived relations. Typically, they represent the functional part of the language which manages data types such as list, trees, etc. It is assumed that indices are not used for the functional part.
- *query definition* which is a set of conditions to be solved.

```
DB := Types; Rules; Query.
Types := type Type_definition_1 ... Type_definition_n endtype.
Type_definition := Data_declaration | ExtRelation_declaration | IntRelation_declaration |
                   Function_declaration.
Data_declaration := Type_constrID Type_varID_1 ... Type_varID_n
                    = Constr_declaration_1 '|' ... '|' Constr_declaration_m.
Constr_declaration = Data_constrID Simple_type_expr_1 ... Simple_type_expr_n.
Simple_type_expr := Type_varID | Simple_type_appl.
Simple_type_appl := Type_constrID Simple_type_expr_1 ... Simple_type_expr_n.
ExtRelation_declaration := Ext_relationID :: Type_expr.
IntRelation_declaration := Int_relationID :: Type_expr.
Function_declaration := FunctionID :: Type_expr.
Type_expr := Type_varID | Type_appl.
Type_appl := Type_constrID Type_expr_1 ... Type_expr_n | (Type_expr → Type_expr) |
             (Type_expr ⇒ Type_expr).
```

In the previous table, we have shown the type syntax which is similar to other languages, such as *CURRY* and *TOY*. In the type declaration, two different notations are used for functional types: $a \rightarrow b$ and $a \Rightarrow b$. The first case is the usual functional type, and the second one is a special functional type denoting that this argument is *flexible*, which means that the actual parameters can be only *rigid* expressions (i.e. left-most symbol cannot be a variable). Our idea is to avoid the extensively search of functional values allowed in some higher order languages, and thus the programmer can note his wishes about the using mode of this functions.

The syntax of the rest of modules is shown in the following table. The base and derived relations can use complex terms in its definition in the form of *patterns*. A pattern is an expression without total applications of extensional, intensional and functional symbols. *Type_constrID*, *Type_varID*, *Data_constrID*, *Data_varID*, *Ext_relationID*, *Int_relationID* and *FunctionID* are identifiers (starting with upper letters in the case of variables), and the symbols in bold-font constitute the set of the *keywords*.

```
Rules := Base_relations [Derived_relations] [Functions].
Base_relations := base_relations Base_fact_1 ... Base_fact_n end_base_relations.
Base_fact := Ext_relationID Expression_1 ... Expression_n ':=' Expression.
Derived_relations := derived_relations Derived_fact_1 ... Derived_fact_n
                     end_derived_relations.
Derived_fact := Int_relationID Expression_1 ... Expression_n ':=' Expression [⇐ Conditions].
Expression := Data_varID | Data_constrID Expression_1 ... Expression_n |
              FunctionID Expression_1 ... Expression_n |
              Ext_relationID Expression_1 ... Expression_n |
              Int_relationID Expression_1 ... Expression_n.
Conditions := Condition_1 ... Condition_n.
Condition := Expression ⋈ Expression | Expression ⋈̸ Expression | Expression <> Expression |
             Expression </> Expression.
Functions := functions Function_def_1 ... Function_def_n end_functions.
Function_def := FunctionID Expression_1 ... Expression_n ':=' Expression [⇐ Conditions].
Query := query Conditions end_query.
```

Next, we show the typical example of the "ancestor" and, as you can see, $\mathcal{INDALOG}$ has a functional syntax and thus base and derived relations can be represented by means of functions. This functions can have conditions in the form of *joinability* equalities [7] whose semantics is to represent the same constructor term. Moreover, the joinability equality can be also used in the queries like in this example.

```
type                    person = frank | john | mary | michael | thomas
                        parent :: person → person.
                        anc :: person → person.
endtype.
base_relations          parent john := mary.        parent john := frank.
                        parent mary := thomas.      parent frank := michael.
end_base_relation.
derived_relations       anc X := parent X.          anc X := anc(parent X).
end_derived_relations.
query                   anc john ⋈ X.
end_query.
```

In the rest of sections, we will explain more in detail the $\mathcal{INDALOG}$ features, but now we want to explain briefly why three kind of declarations are considered: base, derived relations and functions. It is assumed that for a given query, the language will generate, at compile-time and automatically, indices in order to obtain a more efficient evaluation. These indices will be only generated for the base and derived relations. The base relations will be stored at compile-time and the derived relations at run-time, both in secondary memory. The indexing criteria will be based on the query and the form of the rules to be used in the query solving. In the previous example, we requested the john's ancestors, and thus the relation `anc` will be indexed by its unique argument in order to improve the retrieval of such values. Moreover, the function `anc` uses the function `parent` in its definition and thus `parent` will be also indexed by its (only) argument. Intuitively, for each call to the function `anc`, then the corresponding value for `parent` and the ancestors of the value obtained from `parent` must be accessed.

2.1 Handling of Negation in INDALOG

The incorporation of negation supposes to study its semantics foundations. In [11], a framework called *Constructor Based ReWriting Logic with Failure* (*CRWLF*) has been presented, extending the *CRWL* semantics [7], and allowing to handle negative information in functional-logic programming. In this framework, the negation is intended as '*finite failure*' of reduction. *CRWLF* provides four kinds of operators: (a) $\bowtie$ (joinability constraint) and (a') $\not\bowtie$ (its logical negation), and (b) $<>$ (divergence constraint) and (b') $\not<>$ (its logical negation). As semantics,

we adopt *CRWLF* and thus these constraints can be included in any database. An operational semantics based on bottom-up evaluation extended for the handling of negation has been studied in [2], according to [11].

Next, we show an example of database which includes researching heads, `boss`, and departments, `dept`, as extensional database, and a recursive rule `superboss` as intensional one which allows to compute each hierarchy line for a given researcher.

```
type                  person = john | mary | peter | thomas.
                      depart = cs | elect.
                      boss :: person → person.
                      dept :: person → depart.
                      superboss :: person → person.
endtype.
base_relations        boss john := mary.        boss john := thomas.
                      boss mary := peter.
                      dept john := cs.          dept mary := elect.
                      dept peter := cs.         dept thomas := cs.
end_base_relation.
derived_relations     superboss P := boss P .
                      superboss P := superboss boss P.
end_derived_relations.
query                 superboss X ⋪ john, dept X ⋪ cs.
end_query.
```

In the query, the obtained values for `superboss` are compared with `john` and one value `X=mary` (not belonging to the department `cs`) satisfying the query is found.

2.2 Higher Order

Other feature of the language $\mathcal{INDALOG}$ is the use of higher-order. Higher order programming has been widely studied in the functional logic paradigm offering a rich expressivity power by allowing, among others, more abstraction and reuse in the code. The higher-order example for *XSB* presented in the introduction can be also written in our language, given that we can use higher-order patterns in the head and body of both intensional and extensional part:

```
type                  benefit_name = free_car | health_inst | life_ins | long_vacations.
                      benefit_type = optional | required.
                      person = bob | john.
                      package1 :: benefit_name → benefit_type.
                      package2 :: benefit_name → benefit_type.
                      benefits :: person → (benefit_name → benefit_type).
end_type.
base_relations        package1 health_ins := required.      package1 life_ins := optional.
                      package2 health_ins := required.      package2 free_car := optional.
                      package2 long_vacations := optional.
                      benefits john := package1.            benefits bob := package2.
end_base_relations.
query                 benefits john ⋈ F, F X ⋈ Y.
end_query.
```

The query will obtain the following answers `F = package1`, `X = health_ins`, `Y = required` and `F = package1`, `X = life_ins`, `Y = optional`.

2.3 Grouping Primitives

Our idea is to include primitives in $\mathcal{INDALOG}$ for collecting answers for a given query which can be either a set or a multiset. For this reason we consider two kinds of primitives: one considering the answers as a set and the other one as a multiset. The set primitives are `set` and `set_at_least?` and the syntax is as follows:

`set n var query` `set_at_least? n var query`

wherein `n` is a natural number, `var` is a variable, and `query` is a set of conditions. The primitive `set` is a function which obtains the *n-th* answer (as set) for the

variable `var` of `query`. There is a restriction: the variable specified by the argument `var` can only appear in `query`. Otherwise a renaming of variables will be accomplished. Moreover, this argument cannot be instantiated, only indicates the variable for which values must be computed. This primitive is similar to most Prolog system's primitive `setof(var,goal,list)`, but with a relevant difference. In order to leave to the programmer the decision about what kind of data structure is used to store the answers of the query (list, tree, etc), the $\mathcal{INDALOG}$ primitive `set` returns the solutions once at time. By iterating the counter `n` of type `nat`, we can obtain the (partial or complete) set of solutions for the `query` to be stored in the wished structure. This structure can represent a possibly infinite data in the case of infinite solutions, which can nicely be managed in functional logic programming.

For instance, w.r.t. the ancestor database, the following primitive call `set 2 X` (`anc john` $\bowtie$ `X`) will compute the second answer for the variable `X` in the query `anc john` $\bowtie$ `X`. In this case, the answer will be `thomas` or `michael` depending on the order which the answer is computed in. In addition, we will need other set primitive, named `set_at_least?`, which is a boolean function and returns `true` whenever there exist at least `n` solutions for `var` in the `query`. For instance, the primitive call `set_at_least? 2 X` (`anc john` $\bowtie$ `X`) will compute `true`.

From a theoretical point of view (implementation details apart), `set_at_least?` will return true whether there exists the *n-th* answer, or fail, whenever there not exist more than n answers, and `set` will return the *n-th* answer or will be undefined, respectively. In order to know if a query has no more than n answers, we have to proceed as follows: we have to find all the solutions of the query (which must be less than n) and to refute the query for the rest of values. A query is refuted whether the complementary of any constraint can be proved. For instance, `set 4` (`anc john` $\bowtie$ `X`) fails because `anc john` $\bowtie$ `mary`, `anc john` $\bowtie$ `peter`, `anc john` $\bowtie$ `thomas`, `anc john` $\not\bowtie$ `john`. In some cases, there can be neither proved nor refuted the satisfiability of the query for some value, and thus `set` and `set_at_least?` can remain undefined. In the case of removing `anc X` := `parent X` of the ancestor example, neither `anc john` $\bowtie$ `mary` nor `anc john` $\not\bowtie$ `mary` can be proved and the same for the rest of values, due to `anc` is undefined, and thus `set 1` (`anc john` $\bowtie$ `X`), `set_at_least? 1` (`anc john` $\bowtie$ `X`) are also undefined.

As a nice example of its use, in the above ancestor database is the definition of the function `ancestors` which allows to collect all ancestors of a person in a possibly infinite list:

```
type                    list(A) = [] | [A|list(A)].
                        collect_list_anc :: person → nat → list(person).
                        ancestors :: person → list(person).
end_type.
derived_relations       collect_list_anc Y N := [set N X (anc Y ⋈ X) | collect_list_anc Y (N + 1)]
                                  ⇐ set_at_least? N X (anc Y ⋈ X) ⋈ true.
                        collect_list_anc Y N := [] ⇐ set_at_least? N X (anc Y ⋈ X) ⋪ true.
                        ancestors X := collect_list_anc X 1.
end_derived_relations.
query                   ancestors john ⋈ L.
end_query.
```

The control established by the primitive `set_at_least?` in the definition rule of `collect_list_anc Y N` will allow us to control the number of solutions of the query and therefore the end of the list to be build. In such a way that all computed answers are collected in the list and the query will obtain all john's ancestors, that is `L` = [`mary`, `frank`, `thomas`, `michael`] (or even `L` = [`frank`, `mary`, `michael`, `thomas`]).

With respect to the multiset primitives, we will include two primitives, called `bag` and `bag_at_least`?, with the syntax: `bag n var query` and `bag_at_least? n var query` wherein `n` is natural number, `var` is a variable, and `query` is a set of conditions. The meaning of these primitives is similar to the primitives `set` and `set_at_least?` but considering the solutions as a multiset.

Finally, we will show the use of these four primitives together with the higher-order features to generate aggregation operations. For instance, suppose the following $\mathcal{INDALOG}$ database:

```
type                  person = frank | john | mary | michael | thomas.
                      list(A) = [] | [A|list(A)].
                      parent :: person → person.
                      salary :: person → nat.
                      anc :: person → person.
                      collect_salary_list :: person → nat → list(nat).
                      salary_sum :: person → nat.
                      fold :: (A → B → A) → A → list(B) → A.
                      sum_list :: list(nat) → nat.
endtype.
base_relations        parent john := mary.          parent mary := thomas.
                      parent john := frank.         parent frank := michael.
                      salary john := 2500.          salary mary := 3000.
                      salary thomas := 1200.        salary frank := 2500.
                      salary michael := 1200.
end_base_relation.
derived_relations     anc X := parent X.            anc X := anc(parent X).
                      collect_salary_list X N :=
                              [bag N Y (salary (anc X) ⋈ Y) | collect_salary_list X (N + 1)]
                              ⇐ bag_at_least? N Y (salary (anc X) ⋈ Y) ⋈ true.
                      collect_salary_list X N := []
                              ⇐ bag_at_least? N Y (salary (anc X) ⋈ Y) ⋪ true.
                      salary_sum Y := sum_list (collect_salary_list Y 1).
end_derived_relations.
functions             fold F Z [] := Z.          fold F Z [X|L] := fold F (F Z X) L.
                      sum_list L := fold + 0 L.
end_functions.
query                 salary_sum john ⋈ Y.
end_query.
```

This example allows us to compute the sum of all salaries of john's ancestors. Firstly, we define, in a recursive way, a collector, named `collect_salary_list Z N`, which collects all salaries in a list. Secondly, we use a manipulating function, named `sum_list`, defined by using the higher-order function `fold`. Finally, in order to obtain the accumulated sum, we define the function `salary_sum Y` by applying `sum_list` to the list obtained from `collect_salary_list Y 1`. The proposed query `salary_sum john` $\bowtie$ `Y` will obtain as answer `Y` = 7900 euros.

2.4 Indexing

With respect to the indexing process, our idea is, given an $\mathcal{INDALOG}$ database, to generate automatically the set of indices necessary for the query solving. These indices will be created on base and derived relations of the database taking into account the query. With this aim, the indexing process will detect the set of 'join' operations necessary for the query solving. Next, consider a database as follows:

```
type                  person = john | peter | rose.
                      job_name = computer | teacher.
                      age :: nat → person → nat.
                      job :: nat → job_name.
                      eq_23 :: nat → person → job_name.
end_type.
base_relations        age 10 john := 23.          age 11 peter := 24.
                      age 20 john := 23.          age 21 rose := 21.
                      age 22 peter := 25.         job 10 := computer.
                      job 11 := teacher.          job 20 := teacher.
                      job 21 := computer.         job 22 := teacher.
end_base_relations.
```

derived_relations eq_23 ID Name := Job ⇐ age ID Name ⋈ 23, job ID ⋈ Job.
end_derived_relations.
query eq_23 X Y ⋈ Z.
end_query.

Taking into account this query, the indexing process will start analyzing the definition rules for the function eq_23, and it will detect a 'join' operation in the conditions due to the variable ID shared by the functions age and job. Therefore, the indexing process will generate the following compilation directives @index_age(ID, Name, **Result**)(ID) and @index_job(ID, **Result**)(ID) where **Result** denotes the result of the relation. Here, two indices have been created, one for the function age indexing by the argument ID and other one for the function job with the argument ID.

Now, suppose the query eq_23 X john ⋈ Z wherein the second argument of the function eq_23 is instantiated by the value john. In this case, the indexing process will generate the compilation directives @index_age(ID, Name, **Result**)(Name, ID) and @index_job(ID, **Result**)(ID). A multi-attribute (i.e. Name and ID) index is generated for the function age due to the instantiated variable in the query, and therefore also instantiated in the function age, as well as the 'join' operation occurring in the conditions of the rule for the function eq_23. The index for the function job is the same than the previous one.

By considering the query eq_23 X Y ⋈ teacher, the instantiated value is referred to the result from the function eq_23. Then, there exist two 'join' operations: (1) variable Job shared by the functions eq_23 and job and (2) variable ID shared by the function age and job. Therefore, two indices will be generated by means of the directives @index_age(ID, Name, **Result**)(ID) and @index_job(ID, **Result**)(**Result**, ID). Here a multi-attribute index (primary key is **Result** and secondary one is ID) will be generated for the function job in order to retrieve each ID for every teacher.

Finally and with respect to the complex terms, the indexing process will allow to generate indices on the arguments of a complex term but never on non basic types. For instance, given the complex term address in the following function person(ID, address(Name, City), Age), then the indexing process could generate compilation directives for indices on Name or City, or create a multi-attribute index on both Name and City.

To put an end, remark that in order to implement the indexing process, we will use the typical indexing structures implemented in the database context, that is hash index for main memory relations and B-trees for relations stored in secondary memory.

3 Foundations of INDALOG

In this section we will present the semantic foundations of our language. We have adopted the *Constructor based Rewriting Logic with Failure* (*CRWLF*) presented in [11] as semantic framework of $\mathcal{INDALOG}$ given that the cited semantics allows to provide meaning to functional logic programming with negative constraints. *CRWLF* extends the Constructor Based Rewriting Logic presented in [7], wherein the negation is intended as 'finite failure' of reduction. The conditions that are provable in *CRWL* can also be proved in *CRWLF* but, in addition, *CRWLF* provides 'proofs of unprovability' within *CRWL*. In general, the unprovability is not computable which means that *CRWLF* can only give an approximation to failure in *CRWL* that corresponds to the cases in which unprovability refers to 'finite

failure' of reduction. Here, we are interested in the presentation of an alternative characterization of the *CRWLF* semantics (and therefore equivalent) by using *CRWLF* Herbrand algebras and models. From now on we restrict our presentation to the first-order fragment of the language.

3.1 Basis

We assume a signature $\Sigma = DC \cup DS$ where $DC = \bigcup_{n \in \mathbb{N}} DC^n$ is a set of *constructor* symbols $c, d, \ldots$ and $DS = \bigcup_{n \in \mathbb{N}} DS^n$ is a set of *defined* symbols $f, g, \ldots$, all of them with associated arity and such that $DC \cap DS = \emptyset$. Defined symbols consist of three sets *ES* of extensional symbols, *IS* of intensional symbols and *FS* of function symbols: $DS = ES \cup IS \cup FS$. We also assume a countable set $\mathcal{V}$ of *variable* symbols $X, Y, \ldots$. We write *Term* for the set of (total) *terms* e, e' (also called *expressions*) built up with Σ and $\mathcal{V}$ in the usual way, and we distinguish the subset *CTerm* of (total) constructor terms or (total) *c-terms* $t, s, \ldots$, built up only with symbols of DC and $\mathcal{V}$. Terms intend to represent possibly reducible expressions, whereas c-terms represent data values, not further reducible. Terms correspond with first-order expressions and CTerms with first-order patterns in the first-order fragment of the language. We extend the signature Σ by adding two new constants: the constant $\bot$ that plays the role of undefined value and the new constant symbol F that will be used as an explicit representation of failure of reduction. The set $Term_\bot$ of *partial* terms and the set $CTerm_\bot$ of *partial* c-terms are defined in a natural way. Partial c-terms represent the result of partially evaluated expressions, and thus they can be considered as approximations to the value of expressions. Moreover, we will consider the corresponding sets $Term_{\bot,\mathsf{F}}$ and $CTerm_{\bot,\mathsf{F}}$. A natural *approximation ordering* $\leq$ over $CTerm_{\bot,\mathsf{F}}$ can be defined as the least partial ordering satisfying: $\bot \leq t$, $X \leq X$ and $h(t_1, ..., t_n) \leq h(t'_1, ..., t'_n)$, if $t_i \leq t'_i$ for all $i \in \{1, ..., n\}$, $h \in DC \cup DS$. The intended meaning of $t \leq t'$ is that t is less defined or has less information than t'. Note that the only relations satisfied by F are $\bot \leq \mathsf{F}$ and $\mathsf{F} \leq \mathsf{F}$. In particular, F is maximal. This is reasonable, since F represents 'failure of reduction' and this gives a no further refinable information about the result of the evaluation of an expression.

3.2 INDALOG Semantics

A conditional rewrite rule for a defined symbol $f \in DS^n$ is of the form:

$$\underbrace{f(t_1, ..., t_n)}_{head} := \underbrace{r}_{body} \Leftarrow \underbrace{C}_{condition}$$

where $(t_1, ..., t_n)$ is a linear tuple (each variable in it occurs only once) with $t_1, ..., t_n \in CTerm$; C is a set of constraints of the form $e' \bowtie e''$ (*joinability*), $e' \diamond e''$ (*divergence*), $e' \not\bowtie e''$ (*failure of joinability*) or $e' \not\diamond e''$ (*failure of divergence*); r, e and e' belong to the set $\mathcal{T}$ which consists of terms built from $\Sigma, \mathcal{V}$, $set(n, X, C)$, $set\_at\_least?(n, X, C)$, $bag(n, X, C)$ and $bag\_at\_least?(n, X, C)$ where $n = 0, suc^k(0)$, $k > 0$, $X \in \mathcal{V}$ and C is a condition. We implicitly suppose that $0, suc, true \in DC$. *Extra variables* are not allowed, i.e. $var(r) \cup var(C) \subseteq var(\bar{t})$. The reading of the rule is: $f(t_1, ..., t_n)$ reduces to r if the condition C is satisfied.

The meaning of the conditions is as follows: $e \bowtie e'$ (joinability): e and e' can be both reduced to some $t \in CTerm$;$e \diamond e'$ (divergence): e and e' can be reduced to some (possibly partial) c-terms t and t' having a DC-clash; $e \not\bowtie e'$: failure of $e \bowtie e'$ and $e \not\diamond e'$: failure of $e \diamond e'$, where given set of constructor symbols S,

we say that the c-terms t and t' have a *S-clash* if they have different constructor symbols of S at the same position.

We will use the symbol $\diamondsuit$ to refer to any of the constraints $\bowtie, \lhd\rhd, \not\bowtie, \not\lhd\rhd$. The constraints $\not\bowtie$ and $\bowtie$ are called the *complementary* of each other; the same holds for $\not\lhd\rhd$ and $\lhd\rhd$, and we write $\widetilde{\diamondsuit}$ for the complementary of $\diamondsuit$. The meaning of the constraint $e \diamondsuit e'$ depends on certain syntactic (hence decidable) relations between the corresponding approximations for e and e' and is defined as follows.

Definition 1 (Relations over $CTerm_{\perp,\mathsf{F}}$ [11]).

- $t \downarrow t' \Leftrightarrow_{def} t = t', t \in CTerm$
- $t \uparrow t' \Leftrightarrow_{def}$ *t and t' have a DC-clash*
- $t \not\curlyvee t' \Leftrightarrow_{def}$ *t or t' contain F as subterm, or they have a DC-clash*
- *$\curlyvee$ is defined as the least symmetric relation over $CTerm_{\perp,\mathsf{F}}$ satisfying:*
  - *i)* $X \curlyvee X$, *for all* $X \in \mathcal{V}$
  - *ii)* $\mathsf{F} \curlyvee t$, *for all* $t \in CTerm_{\perp,\mathsf{F}}$
  - *iii) if* $t_1 \curlyvee t'_1, ..., t_n \curlyvee t'_n$ *then* $c(t_1, ..., t_n) \curlyvee c(t'_1, ..., t'_n)$, *for* $c \in DC^n$

When using rules to derive statements, we will need to use what are called *c-instances* of such rules. The set of c-instances of a rule R is defined as $[R]_{\perp,\mathsf{F}} = \{R\theta \mid \theta \in CSubst_{\perp,\mathsf{F}}\}$, where we use substitutions $CSubst_{\perp,\mathsf{F}} = \{\theta : \mathcal{V} \to CTerm_{\perp,\mathsf{F}}\}$.

Now, we present *CRWLF*-Herbrand algebras and models. We assume the reader has familiarity with basic concepts of model theory on logic programming and functional-logic programming (see [3, 7] for more details). Now, we point up some of the notions used in this paper.

Given S, a *partially ordered set* (in short, *poset*) with *bottom* $\perp$ (equipped with a partial order $\leq$ and a least element $\perp$), the set of all totally defined elements of S will be noted *Def(S)*. We write $\mathcal{C}(S)$, $\mathcal{I}(S)$ for the sets of cones and ideals of S, respectively. The set $\bar{S} =_{def} \mathcal{I}(S)$ denotes the *ideal completion* of S, which is also a *poset* under the *set-inclusion* ordering $\subseteq$, and there is a natural mapping for each $x \in S$ into the principal ideal generated by x, $< x >=_{def} \{y \in S : y \leq x\} \in \bar{S}$. Furthermore, $\bar{S}$ is a *cpo* (i.e. every directed set $D \subseteq \bar{S}$ has a least upper bound) whose finite elements are precisely the principal ideals $< x >$, $x \in S$.

Definition 2 (Herbrand Algebras). *For any given signature Σ, a Herbrand algebra $\mathcal{H}$ is an algebraic structure of the form $\mathcal{H} = (CTerm_{\perp,\mathsf{F}}, \{f^{\mathcal{H}}\}_{\{f \in DS\}})$ where $CTerm_{\perp,\mathsf{F}}$ is a poset with the approximation ordering $\leq$ and $f^{\mathcal{H}} \in [CTerm^l_{\perp,\mathsf{F}} \to_n CTerm_{\perp,\mathsf{F}}]$ for $f \in DS^l$, where $[D \to_n E] =_{def} \{f : D \to \mathcal{C}(E) | \ \forall\ u, u' \in D : (u \leq u' \Rightarrow f(u) \subseteq f(u'))\}$. From the set $\{f^{\mathcal{H}}\}_{\{f \in DS\}}$, we can distinguish the deterministic functions $f \in DS^n$, holding that $f^{\mathcal{H}} \in [CTerm^n_{\perp,\mathsf{F}} \to_d CTerm_{\perp,\mathsf{F}}]$ where $[D \to_d E] =_{def} \{f \in [D \to_n E] | \ \forall\ u \in D : f(u) \in \mathcal{I}(E)\}$. The elements of $Def(\mathcal{H})$ are the elements of $CTerm_{\mathsf{F}}$.*

Given a Herbrand algebra $\mathcal{H}$, a valuation over $\mathcal{H}$ is any mapping $\eta : \mathcal{V} \to CTerm_{\perp,\mathsf{F}}$, and we say that η is totally defined iff $\eta(X) \in Def(\mathcal{H})$ for all $X \in \mathcal{V}$. We denote by $Val(\mathcal{H})$ the set of all valuations, and by $DefVal(\mathcal{H})$ the set of all totally defined valuations.

Definition 3 (Satisfactibility). *Let $\mathcal{H}$ be a Herbrand algebra, we say that:*

1. *$\mathcal{H}$ satisfies a joinability $e \bowtie e'$ under a valuation η (in symbols, $(\mathcal{H}, \eta) \models e \bowtie e'$) iff there exist $t \in [\![e]\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ and $t' \in [\![e']\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ such that $t \downarrow t'$.*

2. $\mathcal{H}$ *satisfies a divergence* $e \mathrel{<>} e'$ *under a valuation* η *(in symbols,* $(\mathcal{H},\eta) \models e \mathrel{<>} e'$*) iff there exist* $t \in [\![e]\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ *and* $t' \in [\![e']\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ *such that* $t \uparrow t'$.
3. $\mathcal{H}$ *satisfies a failure of joinability* $e \not\bowtie e'$ *under a valuation* η *(in symbols,* $(\mathcal{H},\eta) \models e \not\bowtie e'$*) iff for every* $t \in [\![e]\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ *and* $t' \in [\![e']\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$*, then* $t \not\downarrow t'$ *holds..*
4. $\mathcal{H}$ *satisfies a failure of divergence* $e \not<> e'$ *under a valuation* η *(in symbols,* $(\mathcal{H},\eta) \models e \not<> e'$*) iff for every* $t \in [\![e]\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$ *and* $t' \in [\![e']\!]^{\mathcal{H}}\eta \cap CTerm_{\perp,\mathsf{F}}$*, then* $t \not\uparrow t'$ *holds..*
5. $\mathcal{H}$ *satisfies a condition* C *under a valuation* η *(in symbols,* $(\mathcal{H},\eta) \models C$*) if* $(\mathcal{H},\eta) \models e \Diamond e'$ *for every* $e \Diamond e' \in C$.
6. $\mathcal{H}$ *satisfies the failure of a condition* C *under a valuation* η *(in symbols,* $(\mathcal{H},\eta) \models \neg C$*) if* $(\mathcal{H},\eta) \models e \tilde{\Diamond} e'$ *for any* $e \Diamond e' \in C$.

Satisfactibility of constraints (cases from (1) to (4)) is expressed by means of the relations over terms defined in definition 1. Cases (5) and (6) express satisfactibility of a condition C and the failure of a condition C, respectively, which take into account that $\neg C \equiv e_1 \widetilde{\Diamond} e'_1 \vee \ldots \vee e_n \widetilde{\Diamond} e'_n$, whenever $C \equiv e_1 \Diamond e'_1 \wedge \ldots \wedge e_n \Diamond e'_n$.

Definition 4 (Herbrand Denotation). *The evaluation of an* $e \in \mathcal{T}_{\perp,\mathsf{F}}$ *in* $\mathcal{H}$ *under* η *yields* $[\![e]\!]^{\mathcal{H}}\eta \in \mathcal{C}(CTerm_{\perp,\mathsf{F}})$ *which is defined recursively as follows:*

1. $[\![\perp]\!]^{\mathcal{H}}\eta =_{def} < \perp >$, $[\![\mathsf{F}]\!]^{\mathcal{H}}\eta =_{def} < \mathsf{F} >$ *and* $[\![X]\!]^{\mathcal{H}}\eta =_{def} < \eta(X) >$*, for* $X \in \mathcal{V}$.
2. $[\![c(e_1, \ \ldots \ , e_n)]\!]^{\mathcal{H}}\eta =_{def} < c([\![e_1]\!]^{\mathcal{H}}\eta, \ \ldots \ , [\![e_n]\!]^{\mathcal{H}}\eta) >$ *for all* $c \in DC^n$.
3. $[\![f(e_1, \ \ldots \ , e_n)]\!]^{\mathcal{H}}\eta =_{def} f^{\mathcal{H}}([\![e_1]\!]^{\mathcal{H}}\eta, \ \ldots \ , [\![e_n]\!]^{\mathcal{H}}\eta)$*, for all* $f \in DS^n$.
4. $[\![set(n,X,C)]\!]^{\mathcal{H}}\eta =_{def} < \theta_n(X) >$ *if* $card(Ans) \geq [n]$*, otherwise* $[\![set(n,X,C)]\!]^{\mathcal{H}}\eta =_{def} < \perp >$*, where* $Ans = \{\theta_i(X) \mid (\mathcal{H}, \eta \cdot \theta_i) \models C\}$.
5. $[\![bag(n,X,C)]\!]^{\mathcal{H}}\eta =_{def} < \theta_n(X) >$ *if* $card(Ans) \geq [n]$*, otherwise* $[\![bag(n,X,C)]\!]^{\mathcal{H}}\eta =_{def} < \perp >$*, where* $Ans = \{\{\theta_i(X) \mid (\mathcal{H}, \eta \cdot \theta_i) \models C\}\}$.
6. $[\![set\_at\_least?(n,X,C)]\!]^{\mathcal{H}}\eta =_{def} < true >$ *whenever* $card(Ans) \geq [n]$*; and* $[\![set\_at\_least?(n,X,C)]\!]^{\mathcal{H}}\eta =_{def} < \mathsf{F} >$ *whenever* $card(Ans) < [n]$ *and for every valuation* θ *either holds* $(\mathcal{H}, \eta \cdot \theta) \models C$ *or* $(\mathcal{H}, \eta \cdot \theta) \models \neg C$*, otherwise* $[\![set\_at\_least?(n,X,C)]\!]^{\mathcal{H}}\eta =_{def} < \perp >$ *where* $Ans = \{\theta_i(X) | (\mathcal{H}, \eta \cdot \theta_i) \models C\}$
7. $[\![bag\_at\_least?(n,X,C)]\!]^{\mathcal{H}}\eta =_{def} < true >$ *whenever* $card(Ans) \geq [n]$*; and* $[\![bag\_at\_least?(n,X,C)]\!]^{\mathcal{H}}\eta =_{def} < \mathsf{F} >$ *whenever* $card(Ans) < [n]$ *and for every valuation* θ *either holds* $(\mathcal{H}, \eta \cdot \theta) \models C$ *or* $(\mathcal{H}, \eta \cdot \theta) \models \neg C$*, otherwise* $[\![bag\_at\_least?(n,X,C)]\!]^{\mathcal{H}}\eta =_{def} < \perp >$ *where* $Ans = \{\{\theta_i(X) | (\mathcal{H}, \eta \cdot \theta_i) \models C\}\}$

where $\{\{\_\}\}$ *denotes a multiset of elements, card the cardinal of a set (multiset),* $[n]$ *the value of* n *as natural number.*

Due to non-determinism the evaluation of an expression yields a cone rather than an element. It can be proved that given a Herbrand algebra $\mathcal{H}$, for any $e \in Term_{\perp,\mathsf{F}}$ and $\eta \in Val(\mathcal{H})$, then $[\![e]\!]^{\mathcal{H}}\eta \in \mathcal{I}(CTerm_{\perp,\mathsf{F}})$ if $f^{\mathcal{H}}$ is deterministic for every defined function symbol f occurring in e, and $[\![e]\!]^{\mathcal{H}}\eta \in \mathcal{I}(CTerm_{\mathsf{F}})$ if $e \in Term_{\mathsf{F}}$ and $\eta \in DefVal(\mathcal{H})$.

Set and *bag* primitives denote the *n-th* answer of the condition C if there exists, otherwise are undefined. *Set_at_least?* and *bag_at_least?* denote true whenever there are at least n answers of the condition C, and failure whether there are less than n answers which can only be ensured (cases (6) and (7)) whenever every valuation either satisfies the condition or the failure of the condition.

Definition 5 (Poset of Herbrand Algebras). *We can define a poset with bottom over the Herbrand algebras as follows: given $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A} \leq \mathcal{B}$ iff $f^{\mathcal{A}}(t_1, \ldots, t_n) \subseteq f^{\mathcal{B}}(t_1, \ldots, t_n)$ for every $f \in DS^n$ and $t_i \in CTerm_{\perp,\mathsf{F}}, 1 \leq i \leq n$.*

It can be proved that the ideal completion of this poset is a cpo, called $\mathcal{HALG}$, and $[\![\ ]\!]$ is continuous w.r.t. $\mathcal{HALG}$.

Definition 6 (Herbrand Models). *Let $\mathcal{H}$ be a Herbrand algebra:*

- *$\mathcal{H}$ satisfies a rule $f(\bar{t}) := r \Leftarrow C$ iff*
  1. *every valuation η such that $(\mathcal{H}, \eta) \models C$ verifies $[\![f(\bar{t})]\!]^{\mathcal{H}}\eta \supseteq [\![r]\!]^{\mathcal{H}}\eta$*
  2. *every valuation η such that, for some $i \in \{1, \ldots, n\}$, l_i and t_i have a $DC \cup \{\mathsf{F}\}$-clash, where $l_i \in [\![s_i]\!]^{\mathcal{H}}\eta$, verifies $\mathsf{F} \in [\![f(\bar{s})]\!]^{\mathcal{H}}\eta$*
  3. *every valuation η such that $(\mathcal{H}, \eta) \models \neg C$ verifies $\mathsf{F} \in [\![f(\bar{t})]\!]^{\mathcal{H}}\eta$*
- *$\mathcal{H}$ is a model of a set of rules $R_1, \ldots, R_n$ (in symbols, $\mathcal{H} \models R_1, \ldots, R_n$) iff $\mathcal{H}$ satisfies every R_i.*

Rules can either provide approximations to the value of a function (case (1)) or fail values (cases (2) and (3)). A rule provides fail values whenever either a unification (case (2)) or a condition (case (3)) failure occurs.

Definition 7 (Fix Point Operator). *Given a Herbrand algebra $\mathcal{A}$, and $f \in DS$, we define the fix point operator as:*

$$
\begin{aligned}
T_{\mathcal{P}}(\mathcal{A}, f)(s_1, \ldots, s_n) =_{def} \{\ & [\![r]\!]^{\mathcal{A}}_{\eta} \mid \textit{if there exist } f(\bar{t}) := r \Leftarrow C, \textit{and } \eta \in Val(\mathcal{A}) \\
& \qquad \textit{such that } s_i \in [\![t_i]\!]^{\mathcal{A}}_{\eta} \textit{ and } (\mathcal{A}, \eta) \models C\} \\
& \cup \{\ \mathsf{F} \mid \textit{if there exists } f(\bar{t}) := r \Leftarrow C, \\
& \qquad \textit{such that for some } i \in \{1, \ldots, n\}\ , \\
& \qquad s_i \textit{ and } t_i \textit{ have a } DC \cup \{\mathsf{F}\} - \textit{clash}\} \\
& \cup \{\ \mathsf{F} \mid \textit{if there exist } f(\bar{t}) := r \Leftarrow C, \textit{and } \eta \in Val(\mathcal{A}) \\
& \qquad \textit{such that } s_i \in [\![t_i]\!]^{\mathcal{A}}_{\eta} \textit{ and } (\mathcal{A}, \eta) \models \neg C\} \\
& \cup \{\ \perp \mid \textit{otherwise}\}
\end{aligned}
$$

In each step of the fix point operator application a set of approximation values (due to the non-determinism) is computed: including $\perp$ when the rule cannot be used and F when unification or condition failures occur.

Given $\mathcal{A} \in \mathcal{HALG}$, there exists a unique $\mathcal{B} \in \mathcal{HALG}$ denoted by $T_{\mathcal{P}}(\mathcal{A})$ such that $f^{\mathcal{B}}(t_1, \ldots, t_n) = T_P(\mathcal{A}, f)(t_1, \ldots, t_n)$ for every $f \in DS^n$ and $t_i \in CTerm_{\perp,\mathsf{F}}, 1 \leq i \leq n$. With these definitions, we can ensure the following result which characterizes the least Herbrand model of a set of rules.

Theorem 1. *The fix point operator $T_{\mathcal{P}}$ is continuous and satisfies:*

1. *For every $\mathcal{A} \in \mathcal{HALG}$: $\mathcal{A} \models R_1, \ldots, R_n$ iff $T_{\mathcal{P}}(\mathcal{A}) \leq \mathcal{A}$.*
2. *$T_{\mathcal{P}}$ has a least fix point $\mathcal{M} = \mathcal{H}^{\omega}$ where $\mathcal{H}^0$ is the bottom in $\mathcal{HALG}$ and $\mathcal{H}^{k+1} = T_{\mathcal{P}}(\mathcal{H}^k)$*
3. *$\mathcal{M}$ is the least Herbrand model of $R_1, \ldots, R_n$.*

4 Conclusions and Future Work

In this paper we have presented the main features of a declarative deductive database language based on the integration of functional and logic paradigms. This language includes most relevant features considered in deductive logic languages. In addition, we have shown that this language extends this class of languages by adding the typical advantages of functional logic languages like laziness and possibly infinite data. Finally, we have shown how to use primitives in this language for collecting answers and how indexing is used for the management of large volume of data. As future work, on one hand, we will go towards the implementation of the language, and on the other hand, to the study of an extension of the relational algebra for expressing the semantics of this language as a database query language.

References

1. J. M. Almendros-Jiménez and A. Becerra-Terón. A Framework for Goal-Directed Bottom-Up Evaluation of Functional Logic Programs. In *Proc. of FLOPS*, LNCS 2024, pages 153–169. Springer, 2001.
2. J. M. Almendros-Jiménez, A. Becerra-Terón, and J. Sánchez-Hernández. A Computational Model for Funtional Logic Deductive Databases. In *Proc. of ICLP. To appear*, LNCS. Springer, 2001.
3. K. R. Apt. Logic programming. In *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 10, pages 493–574. MIT Press, 1990.
4. K. R. Apt and R. N. Bol. Logic Programming and Negation: A Survey. *JLP*, 19,20:9–71, 1994.
5. C. Beeri and R. Ramakrishnan. On the Power of Magic. *JLP*, 10(3,4):255–299, 1991.
6. W. Chen, M. Kifer, and D. S. Warren. HiLog: A Foundation for Higher-Order Logic Programming. *JLP*, 15:187–230, 1993.
7. J.C. González-Moreno, M.T.R. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *JLP*, 1(40):47–87, 1999.
8. M. Hanus. Curry, An Integrated Functional Logic Language, Version 0.7. Technical report, University of Kiel, Germany, 2000.
9. D. Kemp, D. Srivastava, and P. Stuckey. Bottom-up Evaluation and Query Optimization of Well-Founded Models. *TCS*, 146:145–184, 1995.
10. F. J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In *Procs. of RTA*, LNCS 1631, pages 244–247. Springer, 1999.
11. F. J. López-Fraguas and J. Sánchez-Hernández. Proving Failure in Functional Logic Programs. In *Proc. of the CL*, LNCS 1861, pages 179–193. Springer, 2000.
12. R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. *JLP*, 11(1,2):189–216, 1991.
13. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL - Control, Relations and Logic. In *Proc. of VLDB*, pages 238–250. Morgan Kaufmann, 1992.
14. R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadir. The CORAL Deductive Database System. *VLDB*, 3(2):161–210, 1994.
15. 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.
16. R. Ramakrishnan and J. Ullman. A Survey of Deductive Database Systems. *JLP*, 23:126–149, 1995.
17. K. Ross. Modular Stratification and Magic Sets for Datalog Programs with Negation. *ACM*, 41(6):1216–1266, 1994.
18. K. F. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 442–453. ACM Press, 1994.
19. S. Sudarshan and R. Ramakrishnan. Optimizations of Bottom-up Evaluation with Non-Ground Terms. In *Proc. of ILPS*, pages 557–574. MIT Press, 1993.
20. J. D. Ullman. Bottom-up Beats Top-down for Datalog. In *Proc. of PODS*, pages 140–149. ACM Press, 1989.
21. J. Vaghani, K. Ramamohanarao, D. B. Kemp, Z. Somogyi, P. J. Stuckey, T. S. Leask, and J. Harland. The Aditi Deductive Database System. *VLDB*, 3(2):245–288, 1994.
22. A. van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *ACM*, 38(3):620–650, 1991.
23. U. Zukowski and B. Freitag. The Deductive Database System LOLA. In *Proc. of LPNMR*, LNAI 1265, pages 375–386. Springer, 1997.