Databases Project Report:

About First Order Logic as a Query Language:

First Order Logic to Non-recursive Datalog

Iken Omar
Master 2 Data Science, Lille University

November 7, 2021

Contents

1 Introduction

This project is part of the course on databases taught by Mr. Sylvain Salvati, and it's about *first order logic as a query language*. Throughout this project, we will first propose a naive representation of first order formulas, and then, in order to push this approach further, we will look for a deep representation of these formulas. A useful description can be found in [1].

The organization of this report is as follows: we first recall the definition and properties of some common query languages in the section 2, as well as the way to switch from one to the other. Then, the section 3 proposes a representation and an "naive" implementation and evaluation of the first order logic formulas. Finally, we present the first order logic formulas in a more advanced (deep) way and we compile them to a datalog program in the section 4.

2 Querying data

A database query is a request for data from a database in order to retrieve or manipulate it. Writing a query, however, requires a set of well-defined codes and commands for the database to understand the statement. This is also known as a query language. There are several equivalent ways to query data, some of the most known querying languages are:

1. First Order Logic (FOL).
2. Relational Algebra (RA).
3. Non Recursive Datalog.

2.1 First Order Logic (FOL)

Definition: Also called first order predicate logic, first order logic is a formal querying languages to describe and reason about predicates. A predicate is a statement that represent a relation or a property, for example in this formula $P(x)$, P is predicates that applies to variable x.

Roughly speaking first order logic is a propositional logic that deals not only with facts or statements but includes as well objects, relations, and functions. The difference between first order logic and Propositional logic is that, in propositional logic, propositions are interpreted as true or false while first order logic contains predicates, quantifiers and variables and those variables range over individuals.

Syntax & Semantics: There are two main parts of first order logic, the first is the syntax that determines which sequences of symbols are well-formed expressions, and the second part is the semantics which determines the meanings of these expressions.

The syntax of first order logic is composed of:

- *Non-logical symbols* which are specific to the domain, they include terms, relations and atomic formulas.

- *Logical symbols* which unlike non logical ones are independent of the domain, they include Boolean combinations, logic connectives and quantifiers. The most common ones are:
  - Negation: $\neg P$.
  - Conjunction: $P \wedge Q$.
  - Disjunction: $P \vee Q$.
  - Universal quantifier: $\forall x P$.
  - Existential quantifier: $\exists x P$.

For example, the following formulas are first-order formulas:

$$\forall x, \neg artist(x) \vee (artist(x) \wedge (actor(x) \vee director(x))) \quad (1)$$
$$\forall x, \neg actor(x) \vee (actor(x) \wedge \exists y, film(y) \wedge acts(x, y)) \quad (2)$$

The first statement states that each artist is either an actor or a film director. While the second one states that every actor acts in at least one movie.

2.2 Relational Algebra (RA)

Definition: Relational algebra is a procedural querying language, that uses operators to perform queries. It takes as input a set of instances of relations and outputs instances of relations, in other words, it take relations as input and output relations. The particularity of relational algebra is that it's performed recursively on a given relation.

Main operations: The main operators of relational algebra are given bellow:

- *Selection* (σ): to select tuples that satisfy a given predicate: $\sigma_p(r) = \{t |\ t \in r \text{ and } p(t)\}$
- *Projection* (Π): to project columns that satisfy a given predicate: $\prod_{A_1, A_2, \ldots, A_n}(r)$
- *Union* ($\cup$): to perform binary union between two relations: $r \cup s = \{t |\ t \in r \text{ or } t \in s\}$
- *Set different* ($-$): select tuples that are in one relation but not in the other: $r - s = \{t |\ t \in r \text{ and } t \notin s\}$
- *Cartesian product* ($\times$): combines two given relations into a new one: $r \times s = \{tq |\ t \in r \text{ and } q \in s\}$

There are additional operations such as: *Set intersection*, *Natural join*, *Assignment*, *Outer join*, etc.

2.3 Non Recursive Datalog

Definition A Datalog is a logical query language for the relational model. A Datalog program Π is a finite set of rules, each rule has a head and a body ($rule : head \leftarrow body$), where body is a set of relations. Any relation of Π is said to be intentional if it's mentioned in the head of some rule of Π, otherwise it's said to be extensional. A Datalog program is said to be defined over a schema R if the set of extensional relations of Π is a subset of R. A query is non-recursive if no intentional predicate appears in the body of any rule. For example, the following program is a Datalog program:

```
p(x) ← not  actor(x)
q(x, y) ← film(y), acts(x, y),  not  p(x)
```

In this example, $\{p, q\}$ is the set of the intensional relations, while $\{actor, acts, film\}$ is the set of extensional relations. A non-recursive Datalog program is a Datalog program such that there exists a function f which assigns a positive number to each relation in Π in a way that for any rule R of Π, if h is the relation of the head of R and b is the its body, then $f(h) > f(b)$. For example, the example given above is non-recursive as the function f given by: $f(p) = 4,\ f(q) = 5,\ f(actor) = 3,\ f(film) = 2,\ f(acts) = 1$ satisfies the condition given above.

2.4 Transformations

For these querying languages, one can go from one to another by using some transformations. In fact these models are said to be equivalent, and we can transform one to another as follows:

First order logic formula into relational algebra: In order to transform a first order logic into an equivalent term of relational algebra we can use the following transformations:

- Negation: $P(x) \wedge \neg Q(y) \rightsquigarrow$ Difference: $P(x) - Q(y)$
- Conjunction: $P(x, y) \wedge Q(x, y) \rightsquigarrow$ Intersection: $P(x) \cap Q(y)$
- Disjunction: $P(x) \vee Q(y) \rightsquigarrow$ Union: $P(x) \cup Q(y)$

Notice that we cannot directly accommodate $\forall$-quantifiers, so before the translation we need to remove them using *De Morgan identities*, this will be explained in details afterwards.

Relational algebra into non recursive datalog program: To transform a relational algebra into an equivalent non term of non-recursive datalog program, we can use the following properties:

- Intersection $R(x, y) \cap T(x, y) \rightsquigarrow I(x, y) \leftarrow R(x, y), T(x, y)$
- Union $R(x, y) \cup T(x, y) \rightsquigarrow \begin{cases} U(x, y) \leftarrow R(x, y) \\ U(x, y) \leftarrow T(x, y) \end{cases}$
- Difference $R(x, y) - T(x, y) \rightsquigarrow I(x, y) \leftarrow R(x, y), \neg T(x, y)$

- Projection $\Pi_x(R) \rightsquigarrow P(x) \leftarrow R(x, y)$

- Selection $\sigma_{x>5}(R) \rightsquigarrow S(x, y) \leftarrow R(x, y), x > 5$
- Product $R \times T \rightsquigarrow P(x, y, u, v) \leftarrow R(x, y), T(u, v)$

First order logic formula into non recursive datalog program: As for relational algebra, we can transform a first order logic into an equivalent term of datalog program using the following transformations:

- Negation: $\neg P(x) \rightsquigarrow N(x) \leftarrow \neg P(x)$
- Conjunction: $P(x) \wedge Q(y) \rightsquigarrow C(x, y) \leftarrow P(x), Q(y)$
- Disjunction: $P(x) \vee Q(y) \rightsquigarrow \begin{cases} D(x, y) \leftarrow P(x) \\ D(x, y) \leftarrow Q(y) \end{cases}$

3 Naive implementation of the evaluation of first order formulae

3.1 Model representation

We will now move on to the implementation of simple data models. To do so, we will choose Python as our programming language. The data and a complete description are available at this link: https://www.fil.univ-lille1.fr/~salvati/cours/bdd_M2_ds/fol/fo_evaluation.html.

In our case a model is divided into several parts:

- A domain: the set of individuals forming the data. In our case it refers to the set of all actors, artists, film directors and films.

- Relations: sets of tuples of individuals or predicates (characteristic functions of these sets). In our case, this refers to the functions such as acts (actor, film) and directs (director, film), etc. There are several ways of implementing these sets:

  1. Set data-structures (any sort of balanced trees, hash-maps, etc.)
  2. Sequences without repetitions
  3. Sequences possibly with repetitions
  4. Streams (sequence whose elements are produced lazily on demand).

However for the moment we will simply use sequences possibly with repetitions. We represent the following model within Python as follows:

- Domain: can be represented as a Python list including all individuals of actors, artists, movies, etc.

- Relations: can be represented either as lists of individuals or as lists of tuples as follows.

Relation	Type of predicate	Python representation
artist	unary predicate	list
actor	unary predicate	list
film	unary predicate	list
film director	unary predicate	list
acts	binary predicate	list of tuples
directs	binary predicate	list of tuples

Table 1: Representation of relations in Python

3.2 Naive implementation of the evaluation

In this subsection we will try our first ('naive') approach to represent and evaluate first order formulas. In order to do that, we will use the fact that most of modern languages are higher-order in the sense that they allow functions to take functions as argument. Therefore, we will implement first order quantifiers using this feature so that we can write in our programming language first order formulae as follow (this method is called shallow embedding; the example is in python) :

```
forall(lambda x: not actor(x) and not director(x) or film(x))
```

We defined `forall` and `exists` methods as well as other predicates like `acted` to check whether an actor is acted in a movie and `directed` to check whether a director directed a given movie or not. Our proposal for `forall` and `exists` is given bellow:

```
def forall(pred:callable):
    """Universal quantifier"""
    return all([pred(x) for x in domain])

def exists(pred:callable):
    """Existential quantifier"""
    return any([pred(x) for x in domain])
```

The other predicates can be found in the Python script `naive_implementation.py`. We test our proposal by evaluating the first order formula that correspond to the following sentences:

1. Each artist is either an actor or a film director.
2. Every actor acts in at least one movie.
3. Every film director directs at least one movie.
4. No actor acts in every movie.
5. For each movie, at least two actors are involved.
6. For every pairs of actors, each of them has played in a movie of the same director (not necessarily the same movie).

7. Write a predicate that corresponds to those artists who are both actors and film directors.

8. Write a predicate that corresponds to those film director who directed only one film.

9. Write a binary relation which contains the pairs of actors and of film directors so that the actors is acting in each movie directed by the director.

For example, the first query refers to the following:

```
forall(lambda x:not artist(x) or (artist(x) and (actor(x)or director(x))))
```

and this yields false, which means that there is an artist who is neither an actor nor a film director.

4 Deep representation of first order formulae

Our previous approach being "naive", we would like to have a more "clever" one. Therefore, in this section, we will try to push our approach forward by trying another representation of the structure of first-order formulas. In fact, we will represent the syntactic structure of first order formulas as a data structure. Then, we will implement an evaluator of these formulas with respect to a model, and finally we will test our proposal on different formulas.

4.1 Deep representation

Rather than using simple function, the elements of the model will be represented as Python classes, in other words, each of the elements in our model will be a Python object. Each one of these objects will have a method `accept`, that will accept a *visitor*. In fact, we will be using *Design - Visitor Pattern* to evaluate first order formulas as well as in other processes, which we will see later, such as finding *Free variables* or *Range-restricted variables* of a given formula. Each one of those visitors can run visitor operations over any set of elements without figuring out their concrete classes. The accept operation directs a call to the appropriate operation in the visitor object.

In the Visitor pattern, we employ a visitor class that alters the execution algorithm of an elementary class. In this way, the execution algorithm of the element can vary according to the variation of the visitor. This pattern belongs to the category of behavioral patterns. Depending on the pattern, the element object must accept the visitor object so that the latter can manage the operation on the element object. For example, to check whether an individual in the domain is an actor or not, the class `actor` must accept the visitor that will verify that the given object is an actor.

The First Order Visitor has different instances and methods that it uses when it visits different predicates. The first instance is an object *model* which contains various predicates (unary and binary) as well as the individuals of the model. The second instance is the *environment* which allows to associate a variable to each individual, in Python we represent it as a dictionary. The last instance is the stack, and as its name indicates, it is used to manage the variables, notably when visiting the quantifiers *forall* and *exists* , and in Python it's represented by a simple list.

We tested our proposal on different queries given in the section above and it seems to be working perfectly. However, we would like to push our approach forwards, since it has some limitations. In fact, our approach does not take into account whether the query is safe or not, i.e., if the query is range-restricted or not. For instance, the following formula *p(x)*$\leftarrow \neg$ *actor(x)* is unsafe, since the variable x ranges over an infinite set. In fact it's the same whenever a formula is negated. Therefore, we can say that queries can become unsafe in case the negation is allowed.

Hence, we need to handle negation, i.e., push negation as low as possible in a formula, then represent a datalog program and compile the first order logic to a datalog.

Definition[Safe Query, Range-Restricted Query]: In literature, a query is said to be safe ("range-restricted") if it returns finite results over all (finite) databases.

Unfortunately the safety of a query is undecidable (There is non algorithm to check whether a query is safe or not). Notice that from now on we will refer to a safe query by *range-restricted* query.

The main steps that we need to do are:

1. Implement Range-Restricted Interpretation and Free Variables visitors.
2. Renoval of forall and handling Negation (Not): put Not as low as possible in a formula!
3. Represent Datalog programs.
4. Compile FOL to datalog.
5. Unfolding in datalog (unfold rules that have variables that are not range restricted)
6. Test range restriction for datalog rules.
7. Compute the right datalog program.
8. Execution of the datalog program.

4.2 Range Restriction Criterion

As mentioned before, we need formulas to be range-restricted before evaluating them. To do so, we will be using the range-restriction criterion. In other words, we will check whether the variables in a formula are bounded or not. Roughly speaking, we say that a formula is range-restricted is every variable on the left is bounded to a table or a constant on the right. Therefore a variable is bounded if:

a. It's the argument of a "positive" predicate (no negation on that predicate). e.g.: $q(x) \leftarrow actor(x)$

b. The variable is equal to one constant. e.g.: $q(x, y) \leftarrow actor(x), y = "Gladiator"$

c. The variable is equivalent (via the symmetry transitive closure of $=$) to some bound variable. e.g.: $q(x, y) \leftarrow actor(x), film(u), u = y$

We would like to represent a datalog program and compile the first order logic to a datalog. However, the datalog program should be range-restricted:

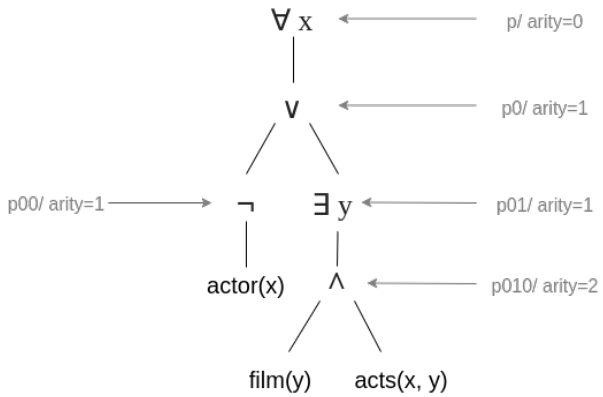

Figure 1: Syntax tree of FOL formula given above.

Definition: We define the fact that a "non-recursive" datalog program is restricted, when all its rules are range-restricted. By extension, a FO-formula is range-restricted when it compiles to such program.

Would we be more liberal in our definition by saying that an FO-formula is range-restricted when it has an "equivalent" term in relational algebra ?, we would have more range-restricted FO-formula. For example, with our criterion, the formula $(a(x) \vee \neg a(x)) \wedge b(x)$ is not range-restricted, while this formula being equivalent to $b(x)$ is range-restricted in the liberal sense. However, the more liberal criterion is undecidable as, taking any closed FO-formula φ, the formula $\varphi \iff a(x) \vee \neg a(x)$ is range-restricted if and only if φ is a contradiction ($\neg\varphi$ is an tautology) determining whether φ is a contradiction or undecidable.

Example: Let's compile the following FO-formula (its syntax tree is given in figure 1):

$$\forall x, \neg actor(x) \vee \exists y, film(y) \wedge acts(x, y)$$

to non-recursive datalog, and check whether it is range restricted or not. To do so, we will be using the following *De Morgan identities*:

1. $\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2$
2. $\neg(\varphi_1 \vee \varphi_2) = \neg\varphi_1 \wedge \neg\varphi_2$
3. $\forall x\varphi = \neg\exists x\neg\varphi$
4. $\exists x\varphi = \neg\forall x\neg\varphi$

Using these identities, we get the following formula (the syntax tree is given in figure 2):

$$\begin{aligned}
\forall x, \neg actor(x) \vee \exists y, film(y) \wedge acts(x, y) &\iff \neg\exists x\neg(\neg actor(x) \vee \exists y, film(y) \wedge acts(x, y)) \\
&\iff \neg\exists x actor(x) \wedge \neg(\exists y, film(y) \wedge acts(x, y)) \\
&\iff \neg\exists x actor(x) \wedge (\neg\exists y, film(y) \vee \neg acts(x, y)
\end{aligned}$$

This new FO-formula is neither range-restricted. But still, by using these identities, we might transform non-range-restricted formulas to range-restricted ones.

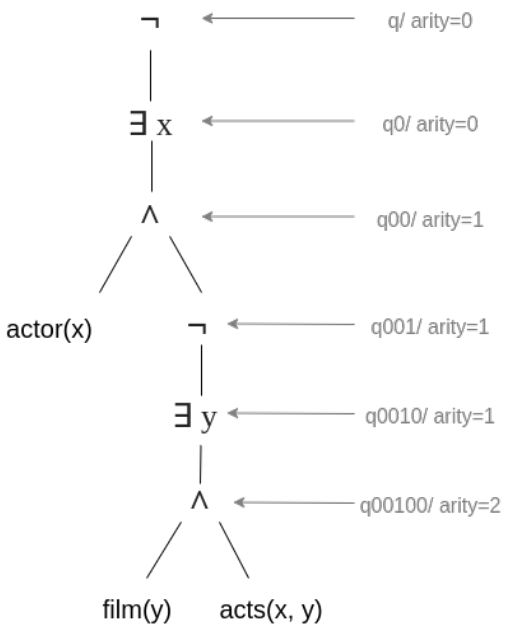

Figure 2: Syntax tree of new FOL formula.

For the implementation, we will define the Range-Restriction Interpretation as a visitor that visits different methods using the following properties:

$$
\begin{aligned}
&rri(\varphi_1 \vee \varphi_2) = rri(\varphi_1) \cap rri(\varphi_2) \\
&rri(\varphi_1 \wedge \varphi_2) = rri(\varphi_1) \cup rri(\varphi_2) \\
&rri(\neg\varphi) = \emptyset \\
&rri(\exists x\varphi) = \begin{cases} rri(\varphi) \setminus \{x\} \text{ if } x \in rri(\varphi) \\ failure, \text{otherwise} \end{cases}
\end{aligned}
$$

Where $rri(\varphi)$ is the set of range-restricted variables of the formula φ. The first formula states that, the set of range-restricted variables of a node $\vee$ is the intersection of the sets of range-restricted variables of its children. While the set of range-restricted variables of a node $\wedge$ is the union of the sets of range-restricted variables of its children. For a node $\neg$, the set of range-restricted variables is just the empty set. And finally, for the quantifier $\exists$, the set of restricted variables is the set of restricted variables of its predicate except its variable if its variable is part of the set of restricted variables of the predicate, otherwise the program fails.

For example,

$$
\begin{aligned}
&rri(\neg actor(x)) = \emptyset \\
&rri(director(x) \wedge film(y)) = \{x, y\} \\
&rri(actor(x) \vee artist(x)) = \{x\}
\end{aligned}
$$

4.3 Free variables

We still need to define *free variables*, this will help us decide whether a formula is range-restricted or not. Let's consider a formula φ as a query, free variables play an important

role:

- A variable x in the formula φ is bound if it is in the scope of a quantifier $\exists x$ or $\forall x$.
- A variable in φ is free if it is not bound.
- A variable in φ is free if it has a free occurrence.
- Free variables go into the output of a query.

For example, let's have a look at some queries and figure out whether they are "bad" or "good" queries, here by "bad" queries we mean not range-restricted queries and obviously by "good" range restricted ones:

1. $p(x) :- \neg actor(x)$, this query is "bad" since x is not bound to any table.
2. $p(x) :- \neg actor(x) \wedge artist(x)$ this query is "good" since x is bound to the table of artists.
3. $p(x, y) :- film(y), directs(x, z)$, this query is "bad", since the variable y is not bounded to any table.
4. $p(x, y, z) :- film(x), directs(x, u), actor(u), actor(y), y = u, z =$ `"Gladiator"`, this query is good since all variables are bounded either to specific tables or constants:
   - u is bounded to some table / relation
   - y is equal to u which means that y is bounded to some table.
   - z is bounded to some table.
   - x is bounded to some table.

In conclusion:

$$\varphi \text{ is range-restricted} \iff rri(\varphi) = FV(\varphi)$$

That is, a formula is range-restricted if and only if the set of its range-restricted variables is the same as the set of its free variables.

For the implementation, we will define a Free Variables visitor to get the free variable of different formulas using the following properties:

$$FV(\varphi_1 \vee \varphi_2) = FV(\varphi_1) \cup FV(\varphi_2)$$
$$FV(\varphi_1 \wedge \varphi_2) = FV(\varphi_1) \cup FV(\varphi_2)$$
$$FV(\neg\varphi(x)) = x$$
$$FV(\exists x\varphi) = \begin{cases} FV(\varphi) \setminus \{x\} \text{ if } x \in FV(\varphi) \\ failure, \text{otherwise} \end{cases}$$

4.4 Unfolding rules

In order to compile a First order program to a datalog, we need to *unfold rules*. Unfolding rules means associating a predicate to each sub-formula in FOL formulas. The process is as follows:

1. Associate a fresh predicate to each sub-formula except for the actual predicates.
2. The arity of these predicates is the number of free variables in the sub-formula.
3. We construct rules for our predicates Q using the predicates of the nodes bellow as follows:
   - $\vee$ node
     - $Q(x) :- Q_1(x_1)$
     - $Q(x) :- Q_2(x_2)$

     In this formula, Q is the predicate associated to $\vee$ node and x is the free variable of that node. While Q_1 is the predicate associated to its left child and x_1 is the free variable of the left child.
   - $\wedge$ node
     - $Q(x) :- Q_1(x_1), Q_2(x_2)$
   - $\exists$ node
     - $Q(x) :- Q_1(x, y)$

     in this formula, y is bounded by $\exists$.

Notice that we cannot directly accommodate $\forall$-quantifiers, so before the translation we need to remove them using *De Morgan* identities given above. Notice as well that the third rule of *Morgan* is problematic for "range restriction" as x must not come from a particular table.

If we consider the following formula from before:

$$\forall x, \neg actor(x) \vee \exists y, film(y) \wedge acts(x, y)$$

We can extract the following rules:

$$\begin{aligned}
&P_{010}(x, y) :- film(y), acts(x, y)\\
&P_{01}(x) :- P_{010}(x, y)\\
&P_{00} :- \neg actor(x)\\
&P_0 :- P_{00}(x)\\
&P_0(x) :- P_{01}(x)\\
&P :- P_0(x)
\end{aligned}$$

The main idea behind unfolding is to compile a formula to a non-recursive datalog, and as long as there is a non range-restricted rule unfold the head of the rule (if possible), and if the resulting program is range-restricted then the formula is range-restricted. Therefore we need to unfold those predicates which have a rule which is not range-restricted and check that every variable in the head has a positive occurrence in the body.

For instance, consider the following non-range-restricted program (figure 3):

$$q(x, y) : -a(x)$$
$$q(x, y) : -b(y)$$

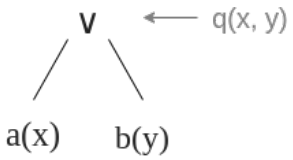

Figure 3: Example of non-range-restricted program

However, we would like the formula in the figure 4 to be range restricted.

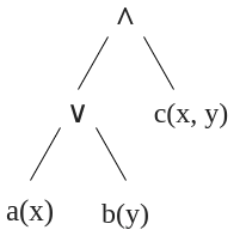

Figure 4: The range-restricted program associated to the program given in figure 3

Unfortunately, it compiles to be a program like:

$$q(x, y) : -q'(x, y), c(x, y)$$
$$q'(x, y) : -a(x)$$
$$q'(x, y) : -b(x)$$

Which is not range restricted ! By unfolding the rules for q' we obtain the following program:

$$q(x, y) : -a(x), c(k, y)$$
$$q(x, y) : -b(y), c(x, y)$$

Which it's range-restricted.

At the level of the formula, this amounts to use distributivity:

$$((\varphi_1 \vee \varphi_2) \wedge \varphi_3) = (\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$$

The program in figure 4 is equivalent to the following program:

4.5 Compile First Order Logic to Datalog

Once the rules are unfolded, we would like to compile first order logic to datalog. However, in datalog program there are two types of predicates:

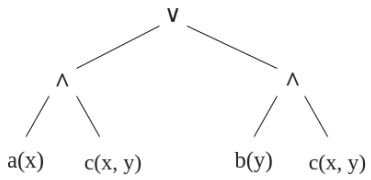

Figure 5: An equivalent program associated to the program given in figure 4

- Existensional predicates: correspond to tables (here to logical predicates).
- Intensional predicates: predicates defined by rules.

Notice also that the process of evaluation in first order logic and datalog program are quite different. In fact, in datalog program the evaluation is done bottom-up, and from left to right, starting with the intensional predicates that are the lowest in the dependency order. In our case, this dependency order is the same as the sub-formula order. So, evaluating a predicate amounts to evaluating each of its rules and taking the union of the results.

We are going to evaluate rules from left to right. To do this, we may need to reorder the predicates in the rules, so as to make sure that negated predicates variables are instantiated when the evaluation reaches them.

A rule $h(x) : -p_1 A_1(x_1), \ldots, p_n A_n(x_n)$ is evaluated as the RA term. Where p_i is the polarity (negation or not) of the predicate A_i. In our case, we suppose that p_1 is positive, meaning that the first predicate is never negated. The projection π_x removes useless variables for the result.

It's possible to project variables on the way when they are useless for the result and don't appear in the right. One way to do it is to use *join via stream fusion* of Cartesian product and selection. However, this is not what we want. For instance, suppose we want to compute $a(x, y)b(y, z)$. If we produce pairs of elements coming from a and coming from b, we do not obtain what we are expecting ! i.e., $tuples(e, f, g)$ so that $(e, f) \in a$ and $(f, g) \in b$.

So as to improve the efficiency of the joins, we want to use *indexes*. For instance, if we are working with an n-array relation, we can construct n indexes for this relation (1 for each dimension).

So as to transform predicates to relation, one idea is to associate number to predicates using the syntax tree of the program so that the function that associate number to predicates verifies the conditions given in the section 2.3. Notice also that between compilation and evaluation we have to maintain the order of the intensional predicates and also reorder right parts of the rules, and concerning the negative predicates, they are fully initiated. The main steps to do in this section are:

1. Design an interface for the relation.
2. Implement relations without indexes.
3. Implement rule evaluation.
4. Implement program evaluation.
5. Implement relations with indexes.

5 Conclusion

Throughout this project, we discovered a naive and deep representation of first-order formulas, as well as interesting concepts such as rule unfolding, free variables, restricted queries, etc. But the approach of using the visitor pattern is relevant even if it makes things more difficult, especially in Python. Nevertheless, we managed to define and implement most of the visitors we need. But unfortunately, due to time constraints, we were not able to test the complete model on some queries.

References

[1] About First Order Logic as a query language

https://www.fil.univ-lille1.fr/~salvati/cours/bdd_M2_ds/fol/fo_evaluation.html

[2] Foundations of Databases

http://webdam.inria.fr/Alice/