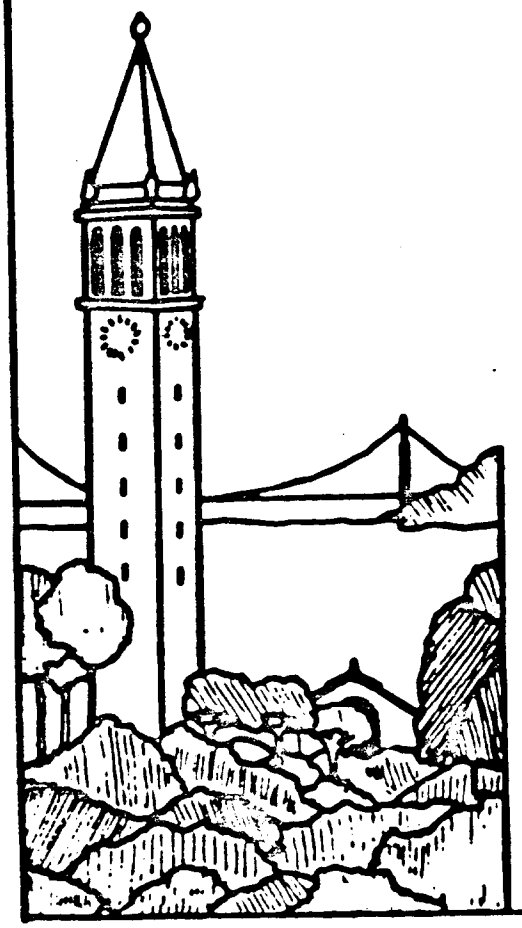
Fuzzy Queries with Linguistic Quantifiers for Information Retrieval from Data Bases

Marc Uszynski



Report No. UCB/CSD 87/333

July 29, 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Fuzzy queries with linguistic quantifiers for information retrieval from data bases

*Marc Uszynski**

Computer Science Division
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

This system is an extension of Prolog toward semantic unification for processing vague queries on a relational data base containing imprecise or uncertain informations. Representation of imprecise and uncertain knowledge is based on fuzzy sets and possibility and necessity measures. A treatment of fuzzy linguistic quantifiers is presented.

1. Introduction

In the past few years, along with the growing importance and use of data bases, there has been a growing interest in information retrieval systems. Not only has become important the ability to store a large amount of data, but we have also been able to retrieve it efficiently. Speed, expressiveness and friendlyness have been among recent considerations when designing such systems.

But one important aspect has somewhat been left apart. This aspect would be the ability to ask questions in a more natural way, that is to say, in a way that would allow general queries, queries that, even though more flexible, could retrieve interesting data (and maybe non-interesting data because of this flexibility but this is not the point). Very often a user has to formulate his query in a very precise manner, translating sometimes a general concept into numerical values, having therefore a doubt to miss interesting data.

This system allows you to stick to your general and imprecise concept. Those queries can be used either with a classic data base or with a data base where data can also be imprecise and expressed in a linguistic way. The system is written in Prolog, which is both a relational data base management system and a programming language. The representation of the imprecision is based on fuzzy logic, theory introduced in 1968 by Pr. Zadeh. Section 2 describes some basic comcepts relative to fuzzy logic, data bases and Prolog in order to set up the background knowledge for a better understanding of section 3, which covers the calculus necessary to handle imprecision in both queries and data.

2.

2.1. Data bases and queries

Basically a relational data base would be constituted of relations containing records. For a single relation, every record has the same structure, i.e. the same number of attributes, arranged in always the same order. A relation is defined by a relation name and attribute names. It could be viewed as a cartesian product of two domains: the first one containing the objects described by the relation and the second one containing the attributes. A column of the table representing the relation stands for an attribute and a row for an object: for instance an EMPLOYEE relation can be represented in the following way:

*Supported in part by NSF Grant NSF IST-8320416 and DARPA Contract NOOO39-84-C-0089.

EMPLOYEE	Name	Age	Manager	Salary
	joe	35	bob	30000
	john	50	joe	20000
	jack	25	joe	16000
	...			

In Prolog the way to represent such a relation is to create a Prolog-clause for each row, whose predicate is "employee" and whose arguments are the values of the attributes. Clauses do not have a right-hand side. They are also called facts. For instance in this case:

employee (joe, 35, bob, 30000).
employee (john, 50, joe, 20000).
employee (jack, 25, joe, 16000).

Then, a typical query in Prolog would be: "Find all records in such relation, such that attribute i has such value". For instance: "Find all employees, whose manager is bob". This query is translated is Prolog in the following way:

employee (X, _, bob, _)?

The variable "_" (underscore), called the anonymous variable, tells that we are not interested in this attribute. The system will go over every record in the employee relation, matching it with the query, that is, matching every instantiated attribute in the query with the record under consideration. Whenever the manager is "bob", the corresponding employee name is retrieved. In a query, there is an implicit *and* between attributes, e.g. "employee (X, _, bob, 30000)?" means: "Find all employees who earn $30000 a year and whose manager is bob". You can also retrieve a single record by asking:

employee (john, Age, Manager, Salary)?

The system will answer: Age = 50, Manager = joe, Salary = 20000. But the mecanism is quite the same except that, when matching the query against the file, only one record will succeed (except if there are more than one employee named john).

2.2. Imprecise data and queries

This is how would work a classical data base and a query session in Prolog. Let's now describe how we can introduce imprecision in both data and queries and what type of query the system can now handle.

We won't go over the theoretical background about fuzzy logic in great detail. See references for that purpose. Let's just recall some basic ideas in order to set up the notation. A fuzzy set F in X is represented by its membership function $\mu_F: X \rightarrow [0,1]$ taking its values in the [0,1] interval. Suppose that A is an attribute, i.e. a column of our simple data base, taking its values in a certain domain X. Imprecision and uncertainty in the knowledge about A can be represented by what is called a possibility distribution over the domain X, $\Pi_{A(e)}$, e standing for a specific employee. Consider a data base, which could be a criminal file and contain informations held by the police that could be imprecise or incertain. This data base would have such attributes as name, age, height, weight. For a single attribute, various types of information can be stored. Here is a few examples and the possibility distributions associated:

* The suspect age is unknown:

  $\Pi_{Age(suspect)}(x) = 1$ for all x in [0,100]

* The age is known with certainty to be between 20 and 30, but no further information is available:

$\Pi_{Age(suspect)}(x)=1$ for $20 \leq x \leq 30$

$\Pi_{Age(suspect)}(x)=0$ for $x<20$ and $x>30$.

* The age is best described by a linguistic term such as *young*:

  $\Pi_{Age(suspect)}(x)=\mu_{young}(x)$ for all x in [0,100], where *young* is a fuzzy set and μ_{young} its membership function.

A portion of the data base could have the following aspect:

SUSPECT	Name	Age	Height	Weigh
	john	24	177	160
	joe	[25,30]	around(180)	around(200)
	jack	young	[170,190]	≥(200)
	jim	unknown	tall	unknown
	...			

young and *tall* are fuzzy set labels, *around* and ≥ are fuzzy set modifiers. All data in this relation are converted into possibility distributions, even though they are crisp values. All data are assumed to have only one single truth value; when the data is not crisp but described by a possibility distribution, all the possible values are mutually exclusive.

This type of data base can be stored in Prolog in exactly the same way as shown previously and the same kind of queries can be treated after having modified the matching mecanism. The matching is no longer syntactic but now a match has to be conducted between the two semantic representations of apparently different atoms. Queries as:

"Find all suspects whose age is around 25, whose height is around 170 cms and whose weight is around 180 pounds"

or "Find all suspects that are young, tall and weight between 170 and 200 pounds"

are typical queries for this system and show much more flexibility and expressiveness than traditional queries. Two matching degrees are computed with each record retrieved so that they can be ranked.

In addition to these capabilities, quantification can be specified. In the original query, there is an implicit *and* between attributes and all attributes must be satisfied. Now queries such as:

"Find all records where Age is young or height is tall or weight is around 180"

or "Find all records where most of the attributes {age is young, height is tall, weight is around 180} are true"

are valid queries. *most* can be replaced by quantifiers like *almost_all*, *all*, *half*, *around_half*, and so on. The next section will describe the computation of the two matching degrees.

3.

3.1. Possibility and Necessity

Whenever the data stored is uncertain or vague, or when the query is vague itself, a match between a unit of data and the query cannot be determined with complete sureness because the exact values of both data and query are not known with precision. One way of getting, in spite of this, some information is to consider successively the most favourable case and the less favourable case and in this way retrieve two matching degrees called respectively Possibility and Necessity.

Let's consider an object *s* of a data base (in our example *s* is a suspect like john). A(s) is one of the attributes (age, height,...) and Q is a unit of query dealing with the same attribute (age is young). Q must be viewed as a union of possible solutions for achieving the match, the union (let's call it α) of all ages that can fit the description *young*. In our

example "age is young", *is* stands for an equality. This is the simplest form of relation that can be requested between the attribute and the atom *young*. But other relations can be specified like: *much bigger than*, *around* or *not*. These relations are also described by a membership function μ_r. The general form of a query focused on only one attribute is the following:

"Retrieve all records s such that $A(s)$ r a"

a corresponds to *young* in our case.

In this framework then, the possibility measure *Pos* is the possibility that the value of the attribute A for the object s is in the set of elements that are in relation r with at least one element of a. And the necessity measure *Nec* is the necessity of the same event, that is the necessity that the value of the attribute A for the object s is in the set of elements that are in relation r with at least one element of a.

$$Pos(r(a)|A(s)) = \underset{x\ in\ X}{Sup}\ Min\ (\mu_{r(a)}(x),\ \Pi_{A(s)}(x))$$

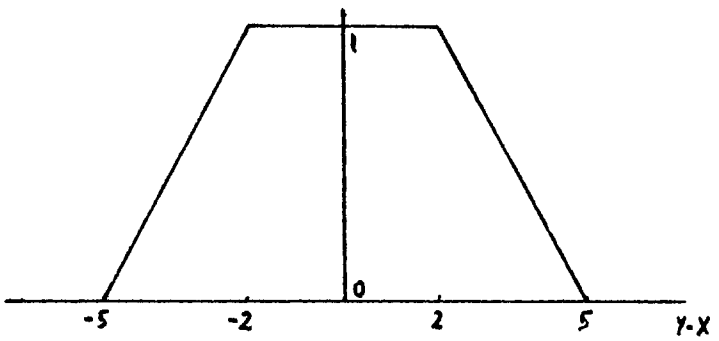
and

$$Nec(r(a)|A(s)) = \underset{x\ in\ X}{Inf}\ Max\ (\mu_{r(a)}(x),\ 1-\Pi_{A(s)}(x))$$

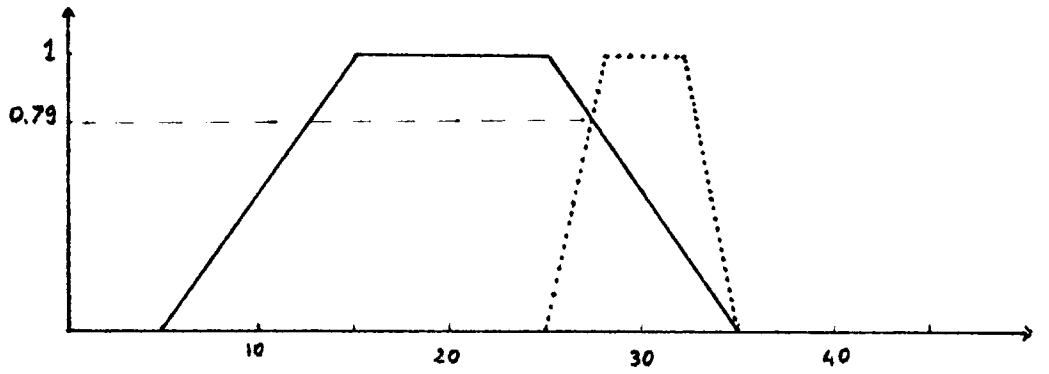
with

$$\mu_{r(a)}(x) = \underset{y\ in\ X}{Sup}\ Min\ (\mu_{r(x,y)},\ \mu_a(y))$$

In the case where the information stored in the data base $A(s)$ is precise, that is, has a single value, then the two matching degrees are equal. In other cases they are not. μ_a is the membership function describing the atom a (a = *young* in our example) and μ_r is the membership function of the relation r, which can be fuzzy, defined over the cross product XxX, where X is the universe of discourse for the attribute A. For instance, *around* can be represented in the following way:



As an example of those two matching degrees, suppose that we have in the data base the assertion "john is young" and that we are looking for people whose age is around 30. Both concepts can be represented as fuzzy numbers and drawn on the same diagram.



We can then conclude that the possibility that john is around 30 is 0.79 and that the necessity that john is around 30 is 0.

3.2. Agregation of single queries. Fuzzy quantifiers.

So far, we have only considered queries dealing with only one argument. But very often we need to specify queries where more than one argument is implied. The simplest example of compound queries is when there is an implicit *and* between arguments ("...where age is young and height is around 170 and ..."). This case and the symetric one, corresponding with *or* between arguments, are solved by taking respectively the minimum and the maximum of atomic degrees.

$$Pos(Q_1 \text{ and } Q_2|s) = Min\ (Pos(Q_1|s),\ Pos(Q_2|s))$$

$$Nec(Q_1 \text{ and } Q_2|s) = Min\ (Nec(Q_1|s),\ Nec(Q_2|s))$$

$$Pos(Q_1 \text{ or } Q_2|s) = Max\ (Pos(Q_1|s),\ Pos(Q_2|s))$$

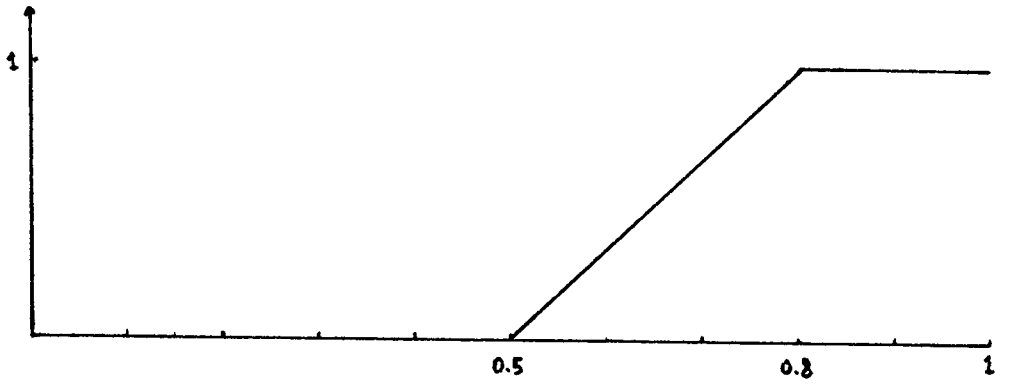
$$Nec(Q_1 \text{ or } Q_2|s) = Max\ (Nec(Q_1|s),\ Nec(Q_2|s))$$

These formulas are only true when the attributes implied in queries Q_1 and Q_2 are independent, that is, that their values are not related with one another's.

But it is also possible to quantify a query in many other ways, according for instance, only a certain number of arguments to be satisfied, e.g.

> "Find all records where most of the following conditions are satisfied: age is young, height is around 170, weight between 170 and 200, hair color is dark, and so on."

so that, if for a suspect, only one attribute doesn't match at all, while all others do, this record is still retrieved. This gives an even greater flexibility to queries. Examples of quantifiers would be: *most, almost_all, more than 50%, at least a few...* Quantifiers can be represented also by fuzzy sets in the interval [0,1]. Very often the meaning of such linguistic quantifiers is not easy to capture and can depend on the domain but, for instance, *most* can be described by:



Once we have those representations, the possibility and necessity degrees for a query like:

"Find all records where q out of the following conditions are satisfied $\{A_1(s)\ r_1\ a_1, A_2(s)\ r_2\ a_2, ..., A_n(s)\ r_n\ a_n\}$."

is computed in the following manner: after computing the values of the two matching degrees (possibility and necessity) for each single query $A_i(s)\ r_i\ a_i$, we have membership degrees of the two fuzzy sets of attributes being possibly matched and necessarily matched. Then the ratio of the two sigma-counts of these sets over the number of arguments n is computed:

$$r_P = \frac{1}{n} \sum_{i=1}^{n} Pos(r_i a_i | A_i(s))$$

$$r_N = \frac{1}{n} \sum_{i=1}^{n} Nec(r_i a_i | A_i(s))$$

Those two ratios are themselves matched against the definition of the quantifier q. So, eventually, the two matching degrees for this query Q are:

$$Pos(Q|s) = \mu_q(r_P) \quad \text{and} \quad Nec(Q|s) = \mu_q(r_N)$$

3.3. Requirements and Implementation

The data is stored in a Prolog-fact style, with a predicate (in this case the relation name) and arguments. The third line of our table is converted to:

suspect (jack, young, [170,190], ≥(200)).

To keep it general, let's call the relation name (here *suspect*) *rel*. The general form of a question with no quantifiers would be as in classical Prolog except for adding an *f* before *rel*: frel(.,.,...,.)? Each argument of the query could be either an uninstantiated variable in order to retrieve a unit of data or an atom with optional relation specifications. The implicit one is equality. This is the *r a* form we talked about earlier. Examples of relations are: *around, at least* (≥), *at most* (≤),... The atom can be a number, an interval ([20,35]), an object name (with no fuzzy definition), or a fuzzy concept (young, tall,...). When computing the matching degrees, every *r a* is converted into a fuzzy number, except for the object name that has no fuzzy description. In this last case, the match either completely succeed or completely fail as in a classical Prolog query.

Every fuzzy concept has to be defined by a fuzzy number. This fuzzy number can be specified for a precise attribute inside a precise relation. Those descriptions are context dependent. There can very possibly be two descriptions of the same atom depending on the context (*good* can be viewed very differently for a student grade or for an economic ratio). But not only concepts that are used in the data base have to be described by fuzzy numbers but also concepts that are likely to be use by a potential user. If the semantic counterpart is not present, the match will most of the time fail.

With a quantifier, the general form of a query is the following:

q (Q, [liste of queries])?

Q is a quantifier whose description has also to be stored. Q can also be *and* or *or*. The list of queries is a list of queries of the above form frel(.,.,...,.).

4. Conclusion

This work is only a starting point that intends to demonstrate the possibility of storing imprecise and uncertain knowledge and using it in a way that is closer to our not so rigorous commonsense understanding. Once a data base is complete and every possible concept expressed by fuzzy numbers, it becomes no longer useful to remember in what precise way the data is stored. And so, the data base can be used very efficiently by first-time users.

The two matching degrees provide then a ranking of data. The user can decide himself whether he wants to be very strict and consider only completely relevant data or on the contrary allows only a very slight match. Mechanisms of thresholds can also be introduced in order to retrieved data relevant beyond a certain point.

There are possible extensions in different directions that can be added to this system: allowing relations between attributes inside queries, or more possible interactions between relations, and also storing and manipulating rules besides facts.

5. Example

The following session, even though very simple and not very realistic, shows what types of queries the system can now handle. The first part is the data stored including the description of the fuzzy numbers. The second part is a sample session on this data base.

5.1.

```
suspect(john, 24, 177, 160).
suspect(joe, [25,30], around(180), around(200)).
suspect(jack, young, [170,190], >=(200)).
suspect(jim, unknown, tall, unknown).
suspect(joan, around(35), small, <=(140)).
suspect(julianne, young, tall, unknown).
```

The fuzzy numbers are stored using a fnum predicate whose arguments are respectively the fuzzy set label, a relation name and an index inside the relation to make it specific to a precise attribute, and four numbers representing the fuzzy number. The fuzzy number (Mi, Ms, Ei, Es) takes the value 1 between Mi and Ms and the value 0 before Mi-Ei and after Ms+Es. The shape of the curb is trapezoidal. Only the two last can take an infinite value.

```
fnum(old, suspect, 2, 70, 90, 10, inf).
fnum(young, suspect, 2, 15, 25, 10, 10).
fnum(very_tall, suspect, 3, 195, 200, 10, inf).
fnum(tall, suspect, 3, 175, 190, 10, 10).
fnum(medium, suspect, 3, 165, 180, 10, 10).
fnum(small, suspect, 3, 120, 155, inf, 15).
fnum(unknown, suspect, Ind, 10, 10, inf, inf).
```

The following fuzzy numbers are modifiers and quantifiers. Quantifiers can be either relative or absolute.

```
fnum(around, suspect, 2, -2, 2, 3, 3) :- !.
fnum(around, Rel, abs, -10, 10, 20, 20) :- !.
fnum(around, Rel, Ind, -5, 5, 8, 8) :- Ind /= rel.
fnum(>=, Rel, Ind, 0, 0, 0, inf).
```

fnum(<=, Rel, Ind, 0, 0, inf, 0).
fnum(much_greater, Rel, Ind, 4, 4, 2, inf).
fnum(most, Rel, rel, 80, 100, 50, inf).
fnum(almost_all, Rel, rel, 90, 100, 20, inf).
fnum(half, Rel, rel, 45, 55, 10, 10).
fnum(all, Rel, rel, 100, 100, 0, inf).

5.2.

fsuspect(jack, Age, Height, Weight)?
Nec = 100, Pos = 100
Age= young
Height= [170, 190]
Weight= >=(200)

fsuspect(X, around(25), around(170), _)?
Nec = 75, Pos = 75
X= john

Nec = 0, Pos = 100
X= joe

Nec = 0, Pos = 100
X= jack

Nec = 0, Pos = 100
X= jim

Nec = 0, Pos = 0
X= joan

Nec = 0, Pos = 100
X= julianne

fsuspect(X, [20, 30], medium, >=(170))?
Nec = 0, Pos = 0
X= john

Nec = 27, Pos = 100
X= joe

Nec = 0, Pos = 100
X= jack

Nec = 0, Pos = 100
X= jim

Nec = 0, Pos = 0
X= joan

Nec = 0, Pos = 100
X= julianne

quest(and, [fsuspect(X, [20, 30], _, _), fsuspect(X, _, medium, _), fsuspect(X, _, _, >=(170))])?

Nec= 0, Pos= 0
X= john

Nec= 27, Pos= 100
X= joe

Nec= 0, Pos= 100
X= jack

Nec= 0, Pos= 100
X= jim

Nec= 0, Pos= 0
X= joan

Nec= 0, Pos= 100
X= julianne

quest(most, [fsuspect(X, [20, 30], _, _), fsuspect(X, _, medium, _), fsuspect(X, _, _, >=(170))])?
Nec= 72, Pos= 72
X= john

Nec= 90, Pos= 100
X= joe

Nec= 6, Pos= 100
X= jack

Nec= 0, Pos= 100
X= jim

Nec= 0, Pos= 0
X= joan

Nec= 0, Pos= 100
X= julianne

quest(almost_all, [fsuspect(X, [20, 30], _, _), fsuspect(X, _, medium, _), fsuspect(X, _, _, >=(170))])?
Nec= 0, Pos= 0
X= john

Nec= 25, Pos= 100
X= joe

Nec= 0, Pos= 100
X= jack

Nec= 0, Pos= 100
X= jim

Nec= 0, Pos= 0
X= joan

Nec= 0, Pos= 100
X= julianne

fsuspect(X, young, tall, around(180))?
Nec= 0, Pos= 0
X= john

Nec= 0, Pos= 38
X= joe

Nec= 0, Pos= 0
X= jack

Nec= 0, Pos= 100
X= jim

Nec= 0, Pos= 0
X= joan

Nec= 0, Pos= 100
X= julianne

quest(most, [fsuspect(X, young, _, _), fsuspect(X, _, tall, _), fsuspect(X, _, _, around(180))])?
Nec= 72, Pos= 72
X= john

Nec= 10, Pos= 98
X= joe

Nec= 40, Pos= 72
X= jack

Nec= 6, Pos= 100
X= jim

Nec= 0, Pos= 0
X= joan

Nec= 72, Pos= 100
X= julianne

6. References

[1] J. Kacprzyk and A. Ziolkowski, Retrieval from data bases using queries with fuzzy linguistic quantifiers, Polish Academy of Sciences, 1986.

[2] L. A. Zadeh, A computational approach to fuzzy quantifiers in natural languages, *Comp. & Maths. with Appls.* Vol 9, No. 1, pp. 149-184, 1983.

[3] D. Dubois, H. Prade, Manipulation d'informations incomplètes ou incertaines et traitement de questions vagues dans une base de données, *Théorie des possibilités*, Masson, Paris, 1985.

[4] L. Bolc, A. Kowalski, M. Kozlowska, T. Strzalkowski, A natural language information retrieval system with extensions towards fuzzy reasoning, *Int. J. Man-Machine Studies*, **23**, 335-367, 1985.

[5] A. Gilles, M. Uszynski, Logique Floue, Seminar, Ecole Centrale des Arts et Manufactures, 1985.

[6] L. A. Zadeh, The role of Fuzzy Logic in the management of uncertainty in Expert Systems, North-Holland, 1983.

[7] C. J. Date, An Introduction to Database Systems, *The systems programming series*, Third Edition, Vol. 1, Addison-Wesley, 1982.

[8] J. F. Baldwin, Support Logic Programming, University of Bristol, England, 1986.

[9] L. A. Zadeh, Fuzzy sets as a basis for a theory of possibility, *Fuzzy Sets and Systems*, 3-28, 1978.

[10] L. A. Zadeh, Test-score semantics for natural languages and meaning representation via PRUF, *Tech. Note 247*, AI Center, SRI International, Menlo Park, California, 1981.

[11] A. Zimmer, Some experiments concerning the fuzzy meaning of logical quantifiers, *General Surveys of Systems Methodology*, Edited by L. Troncoli, 435-441, 1982.

[12] B. P. Buckles and F. E. Petry, Extension of the fuzzy database with fuzzy arithmetic, *Proc. IFAC Symposium, Fuzzy Information, Knowledge Representation and Decision Processes*, 409-414, Marseille, July 19-21, 1983.

Prolog Code

```
% f is a question in regular Prolog. Here the program will look for
% solutions trough a fuzzy pattern matching.
% this generates all clauses of the data base that are of the same
% kind as the one in the question.

f(Q) :- fq(Q, Nec, Pos),
      prin("0, "Nec = ", Nec, ", Pos = ", Pos).

fq(Q, Nec, Pos) :- functor(Q, F, N),
                   functor(Q1, F, N),
                   Q1,
                   unif(Q, Q1, Nec, Pos, F).

% quest is a question including a quantificator and several atomic questions
% of the above form, ie f followed by the name of the relation.
% This quantifiation allows the user to specify if he wants all the atomic
% questions true at the same time or only almost all, half of them, and so on..
% abs and rel specify whether the quantifier is of the first kind (absolute
% count) or of the second kind (relative count).

q(Quant, Liste) :- quest(Quant, Liste, Nec, Pos),
                   prin("0, "Nec = ", Nec, ", Pos = ", Pos).

quest(Quant, Liste, Nec, Pos) :- explicit(Liste, Liste_expl, Lu),
                                 quest1(Quant, Liste_expl, Nec, Pos).

% explicit allows the user to specify a same query for more than one
% argument of the relation in a compact way.
% For instance: arguments(fpers, 2, 4, >=(good), L) will be understood in
% the following way: [fpers(_,>=(good),_,_,_,_),fpers(_,_,>=(good),_,_,_),
% fpers(_,_,_,>=(good),_,_)], the connective being specified as usually.
% The first index is 1.
% un is a function that bound together variables whose indexes are given
% in the list L.

explicit([], [], L).

explicit([arguments(Frel, I1, I2, Cl, Lau)|L], [El1|Lcons], Lu) :-
        I1 <= I2,
        !,
        imax(Frel, Imax),
        functor(El1, Frel, Imax),
        arg(I1, El1, Cl),
        un(El1, Lau, Lu),
        I1p is I1+1,
        explicit([arguments(Frel, I1p, I2, Cl, Lau)|L], Lcons, Lu).

explicit([arguments(Frel, I1, I2, Cl, Lau)|L], L1, Lu) :-
        I1 > I2,
        !,
        explicit(L, L1, X).

explicit([A|L], [A|L1], Lu) :- explicit(L, L1, Lu).
```

```
un(El, [], []).
un(El, [A|R], [A1|R1]) :- arg(A, El, X), A1 = X, un(El, R, R1).

quest1(or, Liste, Nec, Pos) :- maxdeg(Liste, Nec, Pos).

quest1(and, Liste, Nec, Pos) :- mindeg(Liste, Nec, Pos).

quest1(Quant, Liste, Nec, Pos) :- fnum(Quant, Rel, abs, Mi, Ms, Ei, Es),
                                  !,
                                  sumdeg(Liste, N, P),
                                  nec(Mi, Ms, Ei, Es, N, N, 0, 0, Nec),
                                  pos(Mi, Ms, Ei, Es, P, P, 0, 0, Pos).

quest1(Quant, Liste, Nec, Pos) :- fnum(Quant, Rel, rel, Mi, Ms, Ei, Es),
                                  !,
                                  agreg(Liste, N, P),
                                  nec(Mi, Ms, Ei, Es, N, N, 0, 0, Nec),
                                  pos(Mi, Ms, Ei, Es, P, P, 0, 0, Pos).

% agreg computes the fuzzy cardinality.

agreg(Liste, N, P) :- nbel(Liste, No),
                      sumdeg(Liste, Nec, Pos),
                      N is Nec/No,
                      P is Pos/No.
nbel([], 0).
nbel([X|Y], N) :- nbel(Y, N1), N is N1+1.

% sumdeg realizes the summation of the matching degrees of terms of a list.
% maxdeg computes the greatest degrees.

sumdeg([], 0, 0).
sumdeg([X|Y], Nec, Pos) :- sumdeg(Y, N, P),
                           X =.. [F|A],
                           name(F, Aux),
                           Aux = [Fu|R],
                           name(Rel, R),
                           Q =.. [Rel|A],
                           FQ =.. [fq|[Q, N1, P1]],
                           FQ,
                           Nec is N+N1,
                           Pos is P+P1.

maxdeg([], 0, 0).
maxdeg([X|Y], Nec, Pos) :- maxdeg(Y, N, P),
                           X =.. [F|A],
                           name(F, Aux),
                           Aux = [Fu|R],
                           name(Rel, R),
                           Q =.. [Rel|A],
                           FQ =.. [fq|[Q, N1, P1]],
                           FQ,
                           max(N, N1, Nec),
                           max(P, P1, Pos).
```

```
mindeg([], 100, 100).
mindeg([X|Y], Nec, Pos) :- mindeg(Y, N, P),
                    X =.. [F|A],
                    name(F, Aux),
                    Aux = [Fu|R],
                    name(Rel, R),
                    Q =.. [Rel|A],
                    FQ =.. [fq|[Q, N1, P1]],
                    FQ,
                    min(N, N1, Nec),
                    min(P, P1, Pos).

% these clauses are for producing listings of execution. Use tell(filename)
% if you want to store this in a file.

doc(Q) :- prin("0, "0, Q), Q =.. L, uninarg(L, Luni),
       Q, print(""), prinarg(Luni,0).

uninarg([], []).
uninarg([X|Y], L) :- var(X), !, uninarg(Y, L1), append([X], L1, L).
uninarg([X|Y], L) :- X = [_|_], !,
                uninarg(X, L1),
                uninarg(Y, L2),
                append(L1, L2, L).
uninarg([X|Y], L) :- X =.. XE,
                XE /= [_], !,
                uninarg(XE, L1),
                uninarg(Y, L2), append(L1, L2, L).
uninarg([X|Y], L) :- uninarg(Y, L), !.

prinarg([],X).
prinarg([X|Y],I) :- print("_", I, "= ", X, " "), I1 is I+1, prinarg(Y,I1).
append([], L, L).
append([X|Y], L1, [X|L2]) :- append(Y, L1, L2).

% make the two lists of arguments match together.

unif(C, C1, Nec, Pos, F) :- C =.. L,
                    C1 =.. L1,
                    L = [_|LA],
                    L1 = [_|LA1],
                    unifl(LA, LA1, Nec, Pos, F, 1).

% unifl is the unification of the two lists of arguments
% unifa is the unification of two arguments
% the agregation of the necessity and possibility degrees are made
% by the min operation, assuming the independance of the arguments

unifl([], [], 100, 100, F, I).
unifl([A1|L1], [A2|L2], Nec, Pos, F, I) :- unifa(A1, A2, N, P, F, I),
                                I1 is I+1,
                                unifl(L1, L2, Ne, Po, F, I1),
                                min(N, Ne, Nec),
                                min(P, Po, Pos).
```

```
min(X, Y, X) :- X <= Y, !.
min(X, Y, Y).

max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).

% fnum is the fuzzy number attached to the argument inside the relation

unifa(A, A, 100, 100, F, I) :- !.
unifa(A1, A2, Nec, Pos, F, I) :- fnum(A1, F, I, Mi1, Ms1, Ei1, Es1),
                                 fnum(A2, F, I, Mi2, Ms2, Ei2, Es2),
                                 !,
                                 pos(Mi1, Ms1, Ei1, Es1,
                                    Mi2, Ms2, Ei2, Es2, Pos),
                                 nec(Mi1, Ms1, Ei1, Es1,
                                    Mi2, Ms2, Ei2, Es2, Nec).

% The possibility and necessity that the fuzzy event F (Mi1, Ms1, Ei1, Es1)
% is true knowing C (Mi2, Ms2, Ei2, Es2) is defined by
% Pos(F) = sup min (uF(w), uC(w)) and
% Nec(F) = inf max (uF(w), 1-uC(w)).

pos(Mi1, M, Ei1, Es1,
   M, Ms2, Ei2, Es2, 100) :- !.
pos(Mi1, Ms1, Ei1, Es1,
   Mi2, Ms2, Ei2, Es2, 0) :- Es1 /= inf,
                             Ei2 /= inf,
                             Ms1+Es1 <= Mi2-Ei2, !.
pos(Mi1, Ms1, Ei1, 0,
   Mi2, Ms2, 0, Es2, 0) :- Ms1 < Mi2, !.
pos(Mi1, Ms1, Ei1, inf,
   Mi2, Ms2, Ei2, Es2, 100) :- Ms1 < Mi2, !.
pos(Mi1, Ms1, Ei1, Es1,
   Mi2, Ms2, inf, Es2, 100) :- Ms1 < Mi2, !.
pos(Mi1, Ms1, Ei1, Es1,
   Mi2, Ms2, Ei2, Es2, Pos) :- Ms1 <= Mi2,
                               Pos is 100-(100*Mi2-100*Ms1)/(Es1+Ei2),
                               !.
pos(Mi1, Ms1, Ei1, Es1,
   Mi2, Ms2, Ei2, Es2, Pos) :- Ms2 < Mi1,
                               pos(Mi2, Ms2, Ei2, Es2,
                                  Mi1, Ms1, Ei1, Es1, Pos),
                               !.
pos(Mi1, Ms1, Ei1, Es1,
   Mi2, Ms2, Ei2, Es2, 100) :- !.

nec(Mi1, Ms1, Ei1, Es1,
   Mi2, Ms2, Ei2, Es2, Nec) :- ng(Mi1, Ei1, Mi2, Ei2, Ng),
                               nd(Ms1, Es1, Ms2, Es2, Nd),
                               min(Ng, Nd, Nec), !.

ng(Mi1, inf, Mi2, Ei2, 100) :- !.
ng(Mi1, Ei1, Mi2, Ei2, 0) :- Mi2 < Mi1-Ei1, !.
ng(Mi1, Ei1, Mi2, inf, 0) :- !.
ng(Mi1, Ei1, Mi2, Ei2, 100) :- Mi1 <= Mi2-Ei2, !.
```

```
ng(Mi1, Ei1, Mi2, Ei2, Ng) :- Ng is 100*(Mi2-Mi1+Ei1)/(Ei1+Ei2), !.

nd(Ms1, inf, Ms2, Es2, 100) :- !.
nd(Ms1, Es1, Ms2, Es2, 0) :- Ms1+Es1 < Ms2, !.
nd(Ms1, Es1, Ms2, inf, 0) :- !.
nd(Ms1, Es1, Ms2, Es2, 100) :- Ms2+Es2 <= Ms1, !.
nd(Ms1, Es1, Ms2, Es2, Nd) :- Nd is 100*(Ms1-Ms2+Es1)/(Es1+Es2), !.

% The fuzzy numbers are defined by four numbers. Only the two last can
% take an infinite value.

% this line is necessary for the system to be able to answer a
% imprecise question.

fsuspect(X1, X2, X3, X4) :- f(suspect(X1, X2, X3, X4)).

imax(fsuspect, 4).

% fnum is the fuzzy number attached to the argument inside the relation
% Ind is the rank of the argument.

% The fuzzy number (Mi, Ms, Ei, Es) takes the value 1 between Mi and Ms
% and the value 0 between Mi-Ei and Ms+Es. The shape of the curb is
% trapezoidal.

fnum(Arg, Rel, abs, X, X, 0, 0) :- integer(Arg), !, X = 100*Arg.
fnum(Arg, Rel, Ind, X, X, 0, 0) :- integer(Arg), Ind /= rel, !, X = Arg.
fnum(Comp(Pred), Rel, I, X, Y, Z, T) :-
                        fnum(Comp, Rel, I, Mi1, Ms1, Ei1, Es1),
                        fnum(Pred, Rel, I, Mi2, Ms2, Ei2, Es2),
                        !,
                        X is Mi1+Mi2,
                        Y is Ms1+Ms2,
                        genadd(Ei1, Ei2, Z),
                        genadd(Es1, Es2, T).
fnum([X,Y], Rel, abs, X1, Y1, 0, 0) :- !, X1 = 100*X, Y1 = 100*Y.
fnum([X,Y], Rel, Ind, X, Y, 0, 0) :- Ind /= rel.

% genadd allows the two arguments to take infinite values (inf).

genadd(inf, B, inf) :- !.
genadd(A, inf, inf) :- !.
genadd(A, B, X) :- X is A+B.
```