Getting Answers to Fuzzy and Flexible Searches by Easy Modelling of Real-World Knowledge *

Víctor Pablos-Ceruelo and Susana Munoz-Hernandez
The Babel Research Group, Facultad de Informática, Universidad Politécnica de Madrid, Madrid, Spain

Keywords: Search Engine, Fuzzy Logic, Framework.

Abstract: We present a framework for merging the non-fuzzy real-world information stored in databases with the fuzzy knowledge that we (human beings) have. The interest in this aggregation is providing a (fuzzy and non-fuzzy) search engine able to answer flexible and expressive queries without sacrificing a friendly user interface. We achieve this task by using a new syntax (whose semantics are included too) for modelling the domain knowledge and a flexible and enough general structure to represent any user query. We expect this work contributes to the development of more human-oriented fuzzy search engines.

1 INTRODUCTION

Most of the real-world information is stored in non-fuzzy databases, but most of the queries that we (human beings) wanna pose to a search engine are fuzzy. One example of this is the databases containing the distance of a restaurant to the center and the user query "I want a restaurant close to the center". Assuming that it is nonsense to teach every search engine user how to translate the (almost always) fuzzy query they have in mind into a query that a machine can understand and answer, the problem to be solved has two very different parts: recognition of the query and execution of the recognized query.

The recognition of the query has basically two parts: syntactic and semantic recognition. The first one has to be with the lexicographic form of the set of words that compose the query and tries to find a query similar to the user's one but more commonly used. The objective with this operation is to pre-cache the answers for the most common queries and return them in less time, although sometimes it serves to remove typos in the user queries. An example of this is replacing "cars", "racs", "arcs" or "casr" by "car". The detection of words similar to one in the query is called fuzzy matching and the decision to propose one of them as the "good one" is based on statistics of usage of words and groups of words. The search engines usually ask the user if they want to change the typed word(s) by this one(s).

The semantic recognition is work still in progress and it is sometimes called "natural language processing". In the past search engines were tools to retrieve the web pages containing the words typed in the query, but today they tend to include capabilities to understand the user query. An example is computing 4 plus 5 when the query is "4+5" or presenting a currency converter when we write "euro dollar". This is still far away from our proposal: retrieving web pages containing "fast red cars" instead of the ones containing the words "fast", "red" and "car".

The execution of the recognized query is the second part. Suppose a query like "I want a restaurant close to the center". If we assume that the computer is able to "understand" the query then it will look for a set of restaurants in the database satisfying it and return them as answer, but the database does not contain any information about "close to the center", just the "distance of a restaurant to the center". It needs a mapping between the "distance" and the meaning of "close", and this knowledge must be stored somewhere.

One of the most successful programming languages for representing knowledge in computer science is Prolog, whose main advantage with respect to

*This work is partially supported by research projects DESAFIOS10 (TIN2009-14599-C03-00) funded by Ministerio Ciencia e Innovación of Spain, PROMETIDOS (P2009/TIC-1465) funded by Comunidad Autónoma de Madrid and Research Staff Training Program (BES-2008-008320) funded by the Spanish Ministry of Science and Innovation. It is partially supported too by the Universidad Politécnica de Madrid entities Departamento de Lenguajes Sistemas Informáticos e Ingeniería de Software and Facultad de Informática.

Pablos-Ceruelo V. and Munoz-Hernandez S..
Getting Answers to Fuzzy and Flexible Searches by Easy Modelling of Real-World Knowledge.
DOI: 10.5220/0004555302650272
In *Proceedings of the 5th International Joint Conference on Computational Intelligence* (FCTA-2013), pages 265-272
ISBN: 978-989-8565-77-8

name	distance	price avg.	food type
Il_tempietto	100	30	italian
Tapasbar	300	20	spanish
Ni Hao	900	10	chinese
Kenzo	1200	40	japanese

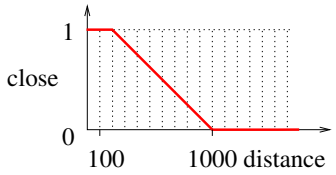

Figure 1: Restaurants database and close fuzzification function.

the other ones is being a more declarative programming language[2]. Prolog is based on logic. It is usual to identify logic with bi-valued logic and assume that the only available values are "yes" and "no" (or "true" and "false"), but logic is much more than bi-valued logic. In fact we use fuzzy logic (FL), a subset of logic that allow us to represent not only if an individual belongs or not to a set, but the grade in which it belongs. Supposing the database contents, the definition for "close" in Fig. 1 and the question "Is restaurant X close to the center?" with FL we can deduce that Il_tempietto is "definitely" close to the center, Tapasbar is "almost" close, Tapasbar is "hardly" close and Kenzo is "not" close to the center. We highlight the words "definitely", "almost", "hardly" and "not" because the usual answers are "1", "0.9", "0.1" and "0" and their humanization is done by defuzzification.

The simplicity of the previous example introduces a question that the curious reader might have in mind: "Does adding a column "close" of type float to the database and computing its value for each row solves the problem?". The answer is yes, but only if our query is not modifiable: It does not help if we can change our question to "I want a very close to the center restaurant" or to "I want a not very close to the center restaurant". Adding a column for each possible question results into a storage problem, and in some sense it is unnecessary: all this values can be computed from the distance value.

Getting fuzzy answers for fuzzy queries from non-fuzzy information stored in non-fuzzy databases has been studied in some works, for example in (Bosc and Pivert, 1995), the SQLf language. The PhD. thesis of Leonid Tineo (Rodriguez, 2005) and the work (Dubois and Prade, 1997) are good revisions, although maybe a little bit outdated. Most of the works mentioned in this papers focus in improving the efficiency of the existing procedures, in including new syntactic constructions or in allowing to introduce the conversion between the non-fuzzy values needed to execute the query and the fuzzy values in the query, for which they use a syntax rather similar to SQL (reflected into the name of the one mentioned before). The advantages of using a syntax similar to SQL are many (removal of the necessity to retrieve all the entries in the database, SQL programmers can learn the new syntax easily, ...) but there is an important disadvantage: the user needs to teach the search engine how to obtain the fuzzy results from the non-fuzzy values stored in the database to get answers to his/her queries and this includes that he/she must know the syntax and semantics of the language and the structure of the database tables. This task is the one we try to remove by including in the representation of the problem the knowledge needed to link the fuzzy knowledge with the non-fuzzy one.

To include the links between fuzzy and non-fuzzy concepts we could use any of the existing frameworks for representing fuzzy knowledge. Leaving apart the theoretical frameworks, as (Vojtáš, 2001), we know about the Prolog-Elf system (Ishizuka and Kanai, 1985), the FRIL Prolog system (Baldwin et al., 1995), the F-Prolog language (Li and Liu, 1990), the FuzzyDL reasoner (Bobillo and Straccia, 2008), the Fuzzy Logic Programming Environment for Research (FLOPER) (Morcillo and Moreno, 2008) the Fuzzy Prolog system (Vaucheret et al., 2002; Guadarrama et al., 2004), or Rfuzzy (Muñoz-Hernández et al., 2011). All of them implement in some way the fuzzy set theory introduced by Lotfi Zadeh in 1965 ((Zadeh, 1965)), and all of them let you implement the connectors needed to retrieve the non-fuzzy information stored in databases, but we needed more meta-information than the one they provide.

Retrieving the information needed to ask the query is part of the problem but, as introduced before, it is needed to determine what the query is asking for before answering it. Instead of providing a free-text search field and recognize the query we do it in the other way: we did an in-depth study on which are all the questions that we can answer from the knowledge stored in our system and we created a general query form that allows to introduce any of this questions. This is why in sec. 3 we do not only present the semantics of our syntactic constructions, but the information that helps us to instantiate the general query form for each domain.

To our knowledge, the works similar to ours are

[2]We say that it is a more declarative programming language because it removes the necessity to specify the flow control in most cases, but the programmer still needs to know if the interpreter or compiler implements depth or breadth-first search strategy and left-to-right or any other literal selection rule.

(Ribeiro and Moreira, 2003), (Bosc and Pivert, 2011) and (Bordogna and Pasi, 1994). While the last two seem to be theoretical descriptions with no implementation associated the first one does not appear to be a search engine project. They provided a natural language interface that answers queries of the types (1) does X (some individual) have some fuzzy property, for example "Is it true that IBM is productive?", and (2) do an amount of elements have some fuzzy property, for example "Do most companies in central Portugal have sales_profitability?".

The paper is structured as follows: the syntax needed to understand it goes first (sec. 2), the description of our framework after (sec. 3) and conclusions and current work in last place (sec. 4), as usual.

2 SYNTAX

We will use a signature Σ of function symbols and a set of variables V to "build" the *term universe* $\mathrm{TU}_{\Sigma,V}$ (whose elements are the *terms*). It is the minimal set such that each variable is a term and terms are closed under Σ-operations. In particular, constant symbols are terms. Similarly, we use a signature Π of predicate symbols to define the *term base* $\mathrm{TB}_{\Pi,\Sigma,V}$ (whose elements are called *atoms*). Atoms are predicates whose arguments are elements of $\mathrm{TU}_{\Sigma,V}$. Atoms and terms are called *ground* if they do not contain variables. As usual, the *Herbrand universe* **HU** is the set of all ground terms, and the *Herbrand base* **HB** is the set of all atoms with arguments from the Herbrand universe. A substitution σ or ξ is (as usual) a mapping from variables from V to terms from $\mathrm{TU}_{\Sigma,V}$ and can be represented in suffix ($(Term)\sigma$) or in prefix notation ($\sigma(Term)$).

To capture different interdependencies between predicates, we will make use of a signature Ω of *many-valued connectives*formed by *conjunctions* $\&_1,\&_2,...,\&_k$, *disjunctions* $\vee_1,\vee_2,...,\vee_l$, *implications* $\leftarrow_1,\leftarrow_2,...,\leftarrow_m$, *aggregations* $@_1,@_2,...,@_n$ and tuples of real numbers in the interval $[0,\ 1]$ represented by $(\mathsf{p},\ \mathsf{v})$.

While Ω denotes the set of connective symbols, $\hat{\Omega}$ denotes the set of their respective associated truth functions. Instances of connective symbols and truth functions are denoted by $\&_i$ and $\hat{\&}_i$ for conjunctors, $\vee_i$ and $\hat{\vee}_i$ for disjunctors, $\leftarrow_i$ and $\hat{\leftarrow}_i$ for implicators, $@_i$ and $\hat{@}_i$ for aggregators and $(\mathsf{p},\ \mathsf{v})$ and $(\mathsf{p},\hat{}\ \mathsf{v})$ for the tuples.

Truth functions for the connectives are then defined as $\hat{\&} : [0,1]^2 \to [0,1]$ monotone[3] and non-decreasing in both coordinates, $\hat{\vee} : [0,1]^2 \to [0,1]$ monotone in both coordinates, $\hat{\leftarrow} : [0,1]^2 \to [0,1]$ non-increasing in the first and non-decreasing in the second coordinate, $\hat{@} : [0,1]^n \to [0,1]$ as a function that verifies $\hat{@}(0,\ \ldots,\ 0) = 0$ and $\hat{@}(1,\ \ldots,\ 1) = 1$ and $(\mathsf{p},\ \mathsf{v}) \in \Omega^{(0)}$ are functions of arity 0 (constants) that coincide with the connectives.

Immediate examples for connectives that come to mind for conjunctors are: in Łukasiewicz logic ($\hat{F}(x,y) = max(0,x+y-1)$), in Gödel logic ($\hat{F}(x,y) = min(x,y)$), in product logic ($\hat{F}(x,y) = x \cdot y$), for disjunctors: in Łukasiewicz logic ($\hat{F}(x,y) = min(1,x+y)$), in Gödel logic ($\hat{F}(x,y) = max(x,y)$), in product logic ($\hat{F}(x,y) = x \cdot y$), for implicators: in Łukasiewicz logic ($\hat{F}(x,y) = min(1,1-x+y)$), in Gödel logic ($\hat{F}(x,y) = y$ if $x > y$ else 1), in product logic ($\hat{F}(x,y) = x \cdot y$) and for aggregation operators[4]: arithmetic mean, weighted sum or a monotone function learned from data.

3 THE FRAMEWORK IN DETAIL

As stated in the introduction, the framework we present provides (1) the syntax needed to model any knowledge domain and (2) an enough expressive syntactical structure for representing any query we can answer with the information stored in the system. We can view it as the sum of three parts: (1) a configuration file (CF) that defines the fuzzy and non-fuzzy concepts of our domain and the relations between them, (2) a framework that understands the CF and provides the search capabilities and (3) a web application that understands the CF, knows the framework capabilities and generates an easy to use human-oriented interface for posing queries to the search engine and show the answers to the user.

The syntactical structure we use to query the search engine has been defined after studying multiple user queries. In comprises all of them (sometimes with small modifications) while trying to be as expressive as possible and has the form

$$I'm\ looking\ for\ a/an\ \boxed{\text{individual}} \quad \left\{ \begin{array}{l} \boxed{\text{not}}\ \boxed{\text{q}}\ \ \boxed{\text{fp}} \\ \quad whose\ \ \boxed{\text{nfp}}\ \ \boxed{\text{co}}\ \boxed{\text{value}} \end{array} \right\} \boxed{\text{AND}} \tag{1}$$

[3] 1 As usually, a n-ary function $\hat{F}$ is called *monotonic in the i-th argument* ($i \leq n$), if $x \leq x'$ implies $\hat{F}(x_1,\ \ldots,\ x_{i-1},\ x,\ x_{i+1},\ \ldots,\ x_n) \leq \hat{F}(x_1,\ \ldots,\ x_{i-1},\ x',\ x_{i+1},\ \ldots,\ x_n)$ and a function is called *monotonic* if it is monotonic in all arguments.

[4] Note that the above definition of aggregation operators subsumes all kinds of minimum, maximum or mean operators.

where *individual* is the element we are looking for (car, skirt, restaurant, ...), q is a quantifier (quite, rather, very, ...), fp is a fuzzy predicate (cheap, large, close to the center, ...), nfp is a non-fuzzy predicate (price, size, distance to the center, ...) and co is a comparison operand (is equal to, is different from, is bigger than, is lower than, is bigger than or equal to, is lower than or equal to and is similar to). The elements in boxes can be modified and the brackets symbolize choosing between a fuzzy predicate query or a comparison between non-fuzzy values (which can be a fuzzy comparison). The "AND" serves to add more lines to the query, to combine multiple conditions. Some examples of use are "I'm looking for a restaurant not very near the city center" (eq. 2), "I'm looking for a restaurant whose food type is mediterranean" (eq. 3) and "I'm looking for a restaurant whose food type is similar to mediterranean and near the city center" (eq. 4).

I'm looking for a/an [restaurant]
[not] [very] [near the city center] [□] (2)

I'm looking for a/an [restaurant]
whose [food type] [is] [mediterranean] [□] (3)

I'm looking for a/an [restaurant]
whose [food type] [is similar to]
[mediterranean] [AND] (4)
[□] [□] [near the city center]

The syntax that we provide to model any knowledge domain is highly coupled to the information that we need to retrieve for providing the values for "individual", "not", "q" (quantifier), "fp" (fuzzy predicate), "nfp" (non-fuzzy predicate), "co" (comparison operand) and "value", and to present the answers in a human-readable way. This is why when we provide its semantics we do it in two ways: by providing the formal ones and by providing what the web interface understands from them. We present first a brief but, for our purposes, complete introduction to the multi-adjoint semantics with priorities that we use to give formal semantics to our syntactical constructions. For a more complete description we recommend reading the papers cited below.

The structure used to give semantics to our programs is the multi-adjoint algebra, presented in (Medina et al., 2002; Medina et al., 2001a; Medina et al., 2001b; Medina et al., 2001c; Medina et al., 2004; Moreno and Ojeda-Aciego, 2002). The interest in using this structure is that we can obtain the credibility for the rules that we write from real-world data, although this time we do not focus in that advantage. We simply highlight this fact so the reader knows why this structure and not some other one.

This structure provides us with the basis, but for our purposes we need that the maximum operator used to decide between multiple rules results the valid one chooses the value of the less generic rule instead of just the higher value. This is why we take as point of departure the work (Pablos-Ceruelo and Muñoz-Hernández, 2011). Definitions needed to understand the formal sematics are given in advance, as usually.

In (Pablos-Ceruelo and Muñoz-Hernández, 2011) the meaning of a fuzzy logic program gets conditioned by the combination of a truth value and a priority value. So, the usual truth value $\mathsf{v} \in [0,1]$ is converted into $(\mathsf{p},\ \mathsf{v}) \in \Omega^{(0)}$, a tuple of real numbers between 0 and 1 where $\mathsf{p} \in [0,1]$ denotes the (accumulated) priority. The usual representation (p,v) is sometimes changed into (pv) to highlight that the variable is only one and it can take the value $\bot$. The set of all possible values is symbolized by **KT** and the ordering between its elements is defined as follows:

Definition 3.1 ($\preccurlyeq_{\mathbf{KT}}$)**.**

$$\begin{array}{c} \bot \preccurlyeq_{\mathbf{KT}} \bot \preccurlyeq_{\mathbf{KT}} (\mathsf{p},\ \mathsf{v}) \\ (\mathsf{p}_1,\ \mathsf{v}_1) \preccurlyeq_{\mathbf{KT}} (\mathsf{p}_2,\ \mathsf{v}_2) \quad \leftrightarrow \quad (\ \mathsf{p}_1 < \mathsf{p}_2\ ) \ \ or \\ (\ \mathsf{p}_1 = \mathsf{p}_2 \ \ and \ \ \mathsf{v}_1 \leq \mathsf{v}_2\ ) \end{array} \quad (5)$$

where $<$ is defined as usually (v_i and p_j are just real numbers between 0 and 1).

Definition 3.2 (Multi-Adjoint Logic Program)**.** A multi-adjoint logic program is a set of clauses of the form

$$A \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} @_j\ (B_1,\ldots,B_n) \ \text{ if } \ \text{COND} \qquad (6)$$

where $(\mathsf{p},\ \mathsf{v}) \in \mathbf{KT}$, $\&_i$ is a conjunctor, $@_j$ an aggregator (unnecessary if $k \in [1..1]$), A and B_k, $k \in [1..n]$, are atoms and *COND* is a first-order formula (basically a bi-valued condition) formed by the predicates in $\mathrm{TB}_{\Pi,\Sigma,V}$, the predicates $=, \geq, \leq, >$ and $<$ restricted to terms from $\mathrm{TU}_{\Sigma,V}$, the symbol true and the conjunction $\wedge$ and disjunction $\vee$ in their usual meaning.

Definition 3.3 (Valuation, Interpretation)**.** A *valuation* or *instantiation* $\sigma : V \rightarrow \mathbf{HU}$ is an assignment of ground terms to variables and uniquely constitutes a mapping $\hat{\sigma} : \mathrm{TB}_{\Pi,\Sigma,V} \rightarrow \mathbf{HB}$ that is defined in the obvious way.

A *fuzzy Herbrand interpretation* (or short, *interpretation*) of a fuzzy logic program is a mapping $I : \mathbf{HB} \rightarrow \mathbf{KT}$ that assigns an element in our lattice

to ground atoms[5].

It is possible to extend uniquely the mapping I defined on **HB** to the set of all ground formulas of the language by using the unique homomorphic extension. This extension is denoted $\hat{I}$ and the set of all interpretations of the formulas in a program **P** is denoted $I_{\mathbf{P}}$.

Definition 3.4 (The operator $\circ$)**.** The application of some conjunctor $\bar{\&}$ (resp. implicator $\bar{\leftarrow}$, aggregator $\bar{@}$) to elements $(\mathsf{p}, \mathsf{v}) \in \mathbf{KT} \setminus \{\bot\}$ refers to the application of the truth function $\hat{\&}$ (resp. $\hat{\leftarrow}$, $\hat{@}$) to the second elements of the tuples while $\circ_{\&}$ (resp. $\circ_{\leftarrow}$, $\circ_{\&}$) is the one applied to the first ones. The operator $\circ$ is defined by

$$x \circ_{\&} y = \frac{x+y}{2} \quad \text{and} \quad z \circ_{\leftarrow} y = 2 * z - y\,.$$

Definition 3.5 (Satisfaction, Model)**.** Let **P** be a multi-adjoint logic program, $I \in I_{\mathbf{P}}$ an interpretation and $A \in \mathbf{HB}$ a ground atom. We say that a clause $Cl_i \in \mathbf{P}$ of the form shown in eq. 6 is satisfied by I or *I is a model of the clause Cl_i* ($I \Vdash Cl_i$) if and only if (iff) for all ground atoms $A \in \mathbf{HB}$ and for all instantiations σ for which $B\sigma \in \mathbf{HB}$ (note that σ can be the empty substitution) it is true that

$$\hat{I}(A) \succcurlyeq_{\mathbf{KT}} (\mathsf{p}, \mathsf{v}) \,\bar{\&}_i\, \bar{@}_i(\, \hat{I}(B_1\sigma),\ \ldots,\ \hat{I}(B_n\sigma)\,) \quad (7)$$

whenever *COND* is satisfied (true). Finally, we say that *I is a model of the program* **P** and write $I \Vdash \mathbf{P}$ iff $I \Vdash Cl_i$ for all clauses in our multi-adjoint logic program **P**.

Now that we have introduced the basics of our formal semantics we introduce the syntax, semantics and what the web interface interprets from them.

The first and most important syntactic structure is the one used to define the individuals we can play with, as "restaurants" in the previous examples. Since the database tables storing the information of an individual can be more than one we decided to allow the programmer to use the Prolog facilities for mixing all the information into a predicate and we depart from this predicate. This means that if we have two tables for storing the information of a restaurant, one for the "food type" (ft) and another for the "distance to the city center" ($dttcc$) we can do the operations in eqs. 8, 9, 10, 11 and 12 to obtain all the information about a restaurant. If instead of that we have all the information of a restaurant in just one table we can make use of the code in eqs. 13 and 14.

[5]The *domain* of an interpretation is the set of all atoms in the Herbrand Base (interpretations are total functions), although for readability reasons we present interpretations as sets of pairs $(A, (\mathsf{p}, \mathsf{v}))$ where $A \in \mathbf{HB}$ and $(\mathsf{p}, \mathsf{v}) \in \mathbf{KT} \setminus \{\bot\}$ (we omit those atoms whose interpretation is the truth value $\bot$).

$$sql\_persistent\_location(rft, db('SQL', user, pass, 'host' : port)). \quad (8)$$

$$:- sql\_persistent(rft(integer, string), rft(id, ft), rft). \quad (9)$$

$$sql\_persistent\_location(rdttcc, db('SQL', user, pass, 'host' : port)). \quad (10)$$

$$:- sql\_persistent(rdttcc(integer, integer), rdttcc(id, dttcc), rdttcc). \quad (11)$$

$$restaurant(id, ft, dttcc) :- rft(id, ft), rdttcc(id, dttcc). \quad (12)$$

$$sql\_persistent\_location(restaurant, db('SQL', user, pass, 'host' : port)). \quad (13)$$

$$:- sql\_persistent(restaurant(integer, string, integer, integer), restaurant(id, ft, yso, dttcc), restaurant). \quad (14)$$

Once we have all the information accessible we use the syntactical structure in eq. 15 to define our virtual database table (vdbt), where pT is the name of the vdbt (the individual or subject of our searches), pA is the arity of the predicate or the vdbt, pN is the name assigned to a column of the vdbt pT and pT' is a basic type, one of $\{boolean\_type,\ enum\_type,\ integer\_type,\ float\_type,\ string\_type\}$. We provide an example in eq. 16 to clarify, in which the restaurant vdbt has four columns (or the predicate has four arguments), the first for the name of the restaurant (the id, of string type), the second for the food type served in the restaurant, the third for the number of years since its opening and the last one for the distance to the city center from that restaurant.

$$define\_database(pT/pA, [(pN, pT')]) \quad (15)$$

$$define\_database(restaurant/4, [(id, string\_type), (food\_type, enum\_type), (years\_since\_opening, integer\_type), (distance\_to\_the\_city\_center, integer\_type)]). \quad (16)$$

This syntactical construction has no formal semantics because it is just for defining the input data, but it provides a lot of information to the web interface and setters/getters that can be used in the programs. First, it provides an instance value for the query field "individual", pT (restaurant). Secondly, a list of values for nfp (id, food type, years since opening and distance to the city center) and their types (string_type, enum_type, integer_type, integer_type) which means that in *co* we can show "is equal to" and "is different from" if it of *string_type*,

"is equal to", "is different from" and "is similar to" if it is of *enum_type* or "is equal to", "is different from", "is bigger than", "is lower than", "is bigger than or equal to" and "is lower than or equal to" if it is of *interger_type*. Third, for each column we have a setter/getter so that for the example in eq. 16 we get by free the predicates $food\_type(R,FT)$ and $distance\_to\_the\_city\_center(R,dttcc)$ that set/obtain in their second argument the respective value in the database column for the restaurant R.

The second syntactical construction is the one used to define similarity between the individuals of *enum_type*. It is shown in eq. 17, where pT and pN mean the same as in eq. 15, $V1$ and $V2$ are possible values for the column pN of the vdbt pT, column that must be of type *enum_type*, and TV is the truth value (a float number) we assign to the similarity between $V1$ and $V2$. We show an example in eq.20, in which we say that the food type mediterranean is 0.8 similar to the spanish food[6]. The syntactical constructions in eqs. 18 and 19 are optional tails for the syntactical construction in eq. 17. Since they can be used as tails when using any of the syntactical constructions that follow this one, we dedicate the paragraph after this one to explain how the semantics of the constructions change when they are used. With respect to the semantics of eq. 17, we show them in eq.21. For the variables in common we take the values written using the new syntax, while for v, $\&_i$, p and *COND* we have by default[7] the values 1, *product*, 0.8 and *true*. This construction does not provide any information to the web interface.

$$similarity\_between(pT, pN(V1), pN(V2), TV) \quad (17)$$
$$with\_credibility(credOp, credVal) \quad (18)$$
$$only\_for\_user\ 'UserName' \quad (19)$$
$$similarity\_between(restaurant, food\_type(mediterranean), food\_type(spanish), 0.8) \quad (20)$$
$$similarity(pT(pN(V1,V2))) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} TV \text{ if } COND \quad (21)$$

We explain now the changes that the use of the tails' constructions in eqs. 18, 19 and 23 introduce in the semantics of the constructions in eqs. 17, 22, 26, 31, 32 and 35. The "by default" values for the variables v, $\&_i$, p and *COND* in the semantics any of this clauses are the values given to those variables when no tail is appended to them and they are used as initial values when the tails used are the ones in eqs. 19 and 23. The tail in eq. 18 serves to define a credibility for a rule together with the operator needed to combine it with the rule truth value. In the construction *credVal* is the credibility, a number of float type, and *credOp* is the operator, any conjunctor having the properties defined in Sec. 2. When we use it the values for v and $\&_i$ (usually 1 and *product*) are changed by *credVal* and *credOp*. The tail in eq. 19 serves to write personalized rules, rules that only apply when the user logged in and the user in the rule are the same one. In the construction *Username* is the name of any user, any string. When we use it the value of *COND* is changed by $COND, currentUser(Me), Me = 'UserName'$[8] and the value for p gets increased by 0.1 because the rule is considered to be more specialized than before and it should be chosen before another rule not having this specialization. The tail in eq. 23 serves (not applicable to the construction in eq. 17) to limit the individuals for which we wanna use the fuzzy clause or rule. In the construction *pN* and *pT* mean the same as in eq. 15, *cond* can take the values *is_equal_to*, *is_different_from*, *is_bigger_than*, *is_lower_than*, *is_bigger_than_or_equal_to* and *is_lower_than_or_equal_to* and *value* can be an integer or a string. When we use it the value of *COND* is changed by $COND, pN(pT)$ *cond value* and the value for p gets increased by 0.05. The reason to increase p in 0.05 when the tail is the one in eq. 23 and to do it in 0.1 when it is the one in eq. 19 is because we want to give to the personalized rules more importance than to the conditioned ones. For example, if eq. 20 had a tail of the form in eq. 19 then the value for p would be 0.85 instead of 0.8.

The third construction (shown in eq. 22) is the one used to define a fuzzy value for all the individuals in a vdbt, and is most of the times used in conjunction with the tail in eq. 23 to limit the assignment to a subset of individuals. In eq. 22 pT and TV mean the same as in eqs. 15 and 17 and $fPredName$ is the fuzzy predicate we are defining. Eq. 24 is an example of use in which we say that the restaurant with id Zalacain is cheap with a truth value of 0.1. The formal semantics for this construction are shown in eq. 25 and the default values for v, $\&_i$, p and *COND* are the values 1, *product*, 0.8 and *true*. From the point of view of the interface, the inclusion of a new fuzzy predicate is taken into account and a new predicate appears in the list of predicates from which we can choose one for

[6]Be careful, we are not saying that the spanish food is 0.8 similar to the mediterranean one. You need to add another clause with that information if you wanna say that too.

[7]The meaning of this "by default" is explained too in the paragraph after this one.

[8]Please remember that ',' is the Prolog symbol for conjunction ($\wedge$).

the field fp (see eq. 1).

$$fPredName(pT) :\sim value(TV) \quad (22)$$
$$if(pN(pT)\ \ cond\ \ value). \quad (23)$$
$$cheap(restaurant):\ value(0.1)$$
$$if(id(restaurant)\ is\_equal\_to\ zalacain). \quad (24)$$
$$fPredName \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} TV\ \text{ if }\ COND \quad (25)$$

The fourth construction serves to define fuzzifications, the computation of fuzzy values for fuzzy predicates from the non-fuzzy value that the individual has in some column in the database. The syntax is presented in eq. 26, where pN and pT mean the same as in eq. 15, $fPredName$ is the name of the fuzzy predicate that is going to be a fuzzification, $[(valIn, valOut)]$ is a list of pairs of values such that $valIn$ belongs to the domain of the fuzzification and $valOut$ to its image[9]. An example in which we compute if a restaurant is traditional or not from the number of years since its opening is presented in eq. 27. The formal semantics for this construction are shown in eq. 28, but only for one sequence of two contiguous points $(valIn1, valOut1)$ $(valIn2, valOut2)$ in 26 (we need to generate one of this for each piece), and the default values for v, $\&_i$, p and $COND$ are the values 1, $product$, 0.6 and the $COND'$ in eq. 30, where OP is the formula in eq. 29. The web interface takes fuzzification functions as fuzzy predicates, so it includes them in the list of available predicates for the field fp (see eq. 1) when they are not there yet.

$$fPredName(pT) :\sim function(pN(pT), [(valIn, valOut)]) \quad (26)$$
$$traditional(restaurant):\ function(years\_since\_opening(restaurant), [(0,0),(5,0.1),(10,0.4),(15,1),(100,1)]). \quad (27)$$
$$fPredName(valIn) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} OP\ \text{ if }\ COND \quad (28)$$
$$OP = valIn\ *\ \frac{(valOut2 - valIn1)}{(valIn2 - valIn1)} \quad (29)$$
$$COND' = (valIn1 < valIn < valIn2) \quad (30)$$

The fifth syntactical construction is for defining rules and has two forms, one used when the body depends on more than one subgoal, shown in eq. 31, and one used when it is just one subgoal, shown in eq. 32. In eq. 31 $aggr$ is the aggregator used to combine the truth values of the subgoals in $complexBody$, which is just a conjunction of names of fuzzy predicates, while in eq. 32 $simplexBody$ it is just the name of a fuzzy predicate. In both of them pT means the same as in eq. 15 and $fPredName$ the same as in eq. 26. The formal semantics for this constructions are respectively shown in eqs. 33 and 34 and the default values for v, $\&_i$, p and $COND$ are the values 1, $product$, 0.4 and $true$. With respect to what the web interface receives from this syntactic structure, it always includes fuzzy predicates in the list of available predicates for the field fp (see eq. 1) when they are not there yet.

$$fPredName(pT) :\sim rule(aggr, complexBody) \quad (31)$$
$$fPredName(pT) :\sim rule(simpleBody) \quad (32)$$
$$fPredName \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} aggr(complexBody)\ \text{ if }\ COND \quad (33)$$
$$fPredName \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} simplexBody\ \text{ if }\ COND \quad (34)$$

The sixth syntactical construction is the one used to define default values for fuzzy computations. Its main goal is not to stop a derivation when a value is missing, and it is really useful when a database can have null values. The syntactic form is presented in eq. 35, where pT means the same as in eq. 15 and $fPredName$ the same as in eq. 26, and we provide an example in eq. 36 in which we say that in absence of information we consider that a restaurant will not be close to the city center (this is what the zero value means). The formal semantics for this constructions are shown in eq. 37 and the default values for v, $\&_i$, p and $COND$ are the values 1, $product$, 0 and $true$. With respect to what the web interface receives from this syntactic structure, this structure serves to define default values for fuzzy predicates and the web interface always includes fuzzy predicates in the list of available predicates for the field fp (see eq. 1) when they are not there yet.

$$fPredName(pT) :\sim defaults\_to(TV) \quad (35)$$
$$near\_the\_city\_center(restaurant) :\sim defaults\_to(0). \quad (36)$$
$$fPredName \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} TV\ \text{ if }\ COND \quad (37)$$

4 CONCLUSIONS

The framework presented is a fuzzy and flexible search engine whose main advantage over the existing ones is providing an easy to use and friendly user interface, avoiding the necessity to learn the complex syntax used in the existing ones. For that purpose it has a syntax (and its semantics) to capture the relations between the fuzzy and non-fuzzy knowledge of

[9] $[(valIn, valOut)]$ is basically a piecewise function definition, where each two contiguous points represent a piece.

any domain (linking information from databases with real-world concepts) and the definition of a general query structure. In this way the user interface is generated automatically from the world representation introduced in the configuration file. This, joint with the possibility to include Prolog code in our configuration file for complex tasks makes our framework a very powerful tool for representing the real world and answering questions about it. A beta version of our framework FleSe is available at our web page.

Our current research focus on deriving similarity relations from the information in the database and not only from the knowledge hard-coded in the program. In this way we could, for example, derive from the RGB composition of colors if they are similar or not.

REFERENCES

Baldwin, J. F., Martin, T. P., and Pilsworth, B. W. (1995). *Fril- Fuzzy and Evidential Reasoning in Artificial Intelligence*. John Wiley & Sons, Inc., New York, NY, USA.

Bobillo, F. and Straccia, U. (2008). fuzzydl: An expressive fuzzy description logic reasoner. In *2008 International Conference on Fuzzy Systems (FUZZ-08)*, pages 923–930. IEEE Computer Society.

Bordogna, G. and Pasi, G. (1994). A fuzzy query language with a linguistic hierarchical aggregator. In *Proceedings of the 1994 ACM symposium on Applied computing*, SAC '94, pages 184–187, New York, NY, USA. ACM.

Bosc, P. and Pivert, O. (1995). Sqlf: a relational database language for fuzzy querying. *Fuzzy Systems, IEEE Transactions on*, 3(1):1 –17.

Bosc, P. and Pivert, O. (2011). On a strengthening connective for flexible database querying. In *Fuzzy Systems (FUZZ), 2011 IEEE International Conference on*, pages 1233–1238.

Dubois, D. and Prade, H. (1997). Using fuzzy sets in flexible querying: why and how? In Andreasen, T., Christiansen, H., and Larsen, H. L., editors, *Flexible query answering systems*, pages 45–60. Kluwer Academic Publishers, Norwell, MA, USA.

Guadarrama, S., Muñoz-Hernández, S., and Vaucheret, C. (2004). Fuzzy prolog: a new approach using soft constraints propagation. *Fuzzy Sets and Systems (FSS)*, 144(1):127 – 150. Possibilistic Logic and Related Issues.

Ishizuka, M. and Kanai, N. (1985). Prolog-elf incorporating fuzzy logic. In *IJCAI'85: Proceedings of the 9th international joint conference on Artificial intelligence*, pages 701–703, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Li, D. and Liu, D. (1990). *A fuzzy Prolog database system*. John Wiley & Sons, Inc., New York, NY, USA.

Medina, J., Ojeda-Aciego, M., and Vojtáš, P. (2001a). A completeness theorem for multi-adjoint logic programming. In *FUZZ-IEEE*, pages 1031–1034.

Medina, J., Ojeda-Aciego, M., and Vojtáš, P. (2001b). Multi-adjoint logic programming with continuous semantics. In Eiter, T., Faber, W., and Truszczynski, M., editors, *LPNMR*, volume 2173 of *Lecture Notes in Computer Science*, pages 351–364. Springer.

Medina, J., Ojeda-Aciego, M., and Vojtáš, P. (2001c). A procedural semantics for multi-adjoint logic programming. In Brazdil, P. and Jorge, A., editors, *EPIA*, volume 2258 of *Lecture Notes in Computer Science*, pages 290–297. Springer.

Medina, J., Ojeda-Aciego, M., and Vojtáš, P. (2002). A multi-adjoint approach to similarity-based unification. *Electronic Notes in Theoretical Computer Science*, 66(5):70 – 85. UNCL'2002, Unification in Non-Classical Logics (ICALP 2002 Satellite Workshop).

Medina, J., Ojeda-Aciego, M., and Vojtáš, P. (2004). Similarity-based unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146(1):43–62.

Morcillo, P. J. and Moreno, G. (2008). Floper, a fuzzy logic programming environment for research. In de Oviedo, F. U., editor, *Proceedings of VIII Jornadas sobre Programación y Lenguajes (PROLE'08)*, pages 259–263, Gijón, Spain.

Moreno, J. M. and Ojeda-Aciego, M. (2002). On first-order multi-adjoint logic programming. In *11th Spanish Congress on Fuzzy Logic and Technology*.

Muñoz-Hernández, S., Pablos-Ceruelo, V., and Strass, H. (2011). Rfuzzy: Syntax, semantics and implementation details of a simple and expressive fuzzy tool over prolog. *Information Sciences*, 181(10):1951 – 1970. Special Issue on Information Engineering Applications Based on Lattices.

Pablos-Ceruelo, V. and Muñoz-Hernández, S. (2011). Introducing priorities in rfuzzy: Syntax and semantics. In *Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE*. to be published.

Ribeiro, R. A. and Moreira, A. M. (2003). Fuzzy query interface for a business database. *International Journal of Human-Computer Studies*, 58(4):363 – 391.

Rodriguez, L. J. T. (2005). (phd. thesis) a contribution to database flexible querying: Fuzzy quantified queries evaluation.

Vaucheret, C., Guadarrama, S., and Muñoz-Hernández, S. (2002). Fuzzy prolog: A simple general implementation using CLP(R). In Baaz, M. and Voronkov, A., editors, *LPAR*, volume 2514 of *Lecture Notes in Artificial Intelligence*, pages 450–464. Springer.

Vojtáš, P. (2001). Fuzzy logic programming. *Fuzzy Sets and Systems*, 124(3):361–370.

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*, 8(3):338–353.