2014 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)
July 6-11, 2014, Beijing, China

On Modelling Real-world Knowledge to Get Answers to Fuzzy and Flexible Searches without Human Intervention

Víctor Pablos-Ceruelo and Susana Munoz-Hernandez

***Abstract*—The Internet has become a place where massive amounts of information and data are being generated every day. This information is most of the times stored in a non-structured way, but the times it is structured in databases it cannot be retrieved by using easy fuzzy queries. Being the information in the database the distance to the city center of some restaurants (and their names) by easy fuzzy queries we mean queries like "I want a restaurant close to the center". Since the computer does not have knowledge about the relation between being close to the center and the distance to the center (of a restaurant) it does not know how to answer this query by itself. We need human intervention to tell the computer from which database column it needs to retrieve data (the one with the restaurant's distance to the center), and how this non-fuzzy information is fuzzified (applying the close function to the retrieved value). Once this is done it can give an answer, just ordering the database elements by this new computed attribute. This example is very simple, but there are others not so simple, as "I want a restaurant close to the center, not very expensive and whose food type is mediterranean". Doing this for each existing attribute does not seem to be a very good idea. We present a web interface for posing fuzzy and flexible queries and a search engine capable of answering them without human intervention, just from the knowledge modelled by using the framework's syntax. We expect this work contributes to the development of more human-oriented fuzzy search engines.**

I. Introduction

Most of the real-world information is stored in a non-structured way, but the times it is kept in regular databases the retrieval cannot be done in an humanized (i.e. flexible and fuzzy) way. Take, for example, a database containing the distance of some restaurants 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 he/she has in his/her 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 deal 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 he/she wants 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 used 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 to know the link between the concepts "distance to the center" and "close to the center" and how to obtain the second from the first one. This is what we call representing or modelling the real-world knowledge.

One of the most successful programming languages for representing knowledge in computer science is Prolog, whose main advantage with respect to the other ones is being a more declarative programming language[1]. 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 and 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, Ni Hao is "hardly"

Pablos-Ceruelo and Munoz-Hernandez are with the Babel Research Group at the Facultad de Informática (Universidad Politécnica de Madrid, Spain). Email: { vpablos, susana }@babel.ls.fi.upm.es

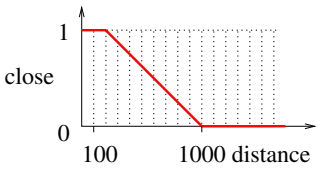
This work is partially supported by research projects DESAFIOS10 (TIN2009-14599-C03-00) funded by Ministerio Ciencia e Innovación de Spain, PROMETIDOS (P2009/TIC-1465) funded by Comunidad Autónoma de Madrid and Research Staff Training Program (BES-2008-008320) funded by the Spanish Ministry of Science 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.

[1]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.

978-1-4799-2072-3/14/$31.00 ©2014 IEEE

name	distance	price avg.	food type
Il_tempietto	100	30	italian
Tapasbar	300	20	spanish
Ni Hao	900	10	chinese
Kenzo	1200	40	japanese
Zalacain	2000		
Don_Jamon			spanish

(a) Restaurants' database contents



(b) Close fuzzification function

Fig. 1: Restaurants database and close fuzzification function.

close and Kenzo is "not" close to the center. We highlight the words "definitely", "almost", "hardly" and "not" because the usual answers for the query are "1", "0.9", "0.1" and "0" for the individuals Il tempietto, Tapasbar, Ni Hao and Kenzo and the humanization of the crisp values is done in a subsequent step by defuzzification.

The simplicity of the previous example introduces a question that the curious reader might have in mind: "Does adding a column "close to the center" 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 these 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 [1], the SQLf language. The PhD. thesis of Leonid Tineo [2] and the work of Dubois & Prade [3] are good revisions, although maybe a little bit outdated. Most of the works mentioned in these 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 [4], we know about the Prolog-Elf system [5], the FRIL Prolog system [6], the F-Prolog language [7], the FuzzyDL reasoner [8], the Fuzzy Logic Programming Environment for Research (FLOPER) [9], the Fuzzy Prolog system [10], [11], or Rfuzzy [12]. All of them implement in some way the fuzzy set theory introduced by Lotfi Zadeh in 1965 ([13]), 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. This is why we propose a new one here.

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 in 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 these questions. The interest in this query form is that we can study from it what information our framework needs to feed its variable fields and what relations it needs to answer the questions it allows to represent. We do this in Sec. III.

The work we present here is, as far as we know, novel in the idea of getting a fuzzy and flexible search engine from the representation of the real-world knowledge in the syntax understood by the framework, but includes facilities present in other works. With respect to the search engine we know the works [14], [15] and [16] are similar to ours. 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 provide 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?". Some facilities we include and are present in most of the frameworks cited before are the definitions of the fuzzy truth value for predicates, fuzzifications, rules or default values. Other ones included and, as far as we now, present only in [17] [18] are priorities and personalization of clauses (or rules) per user. And the one that we have not found in any other framework is the representation of similarity. In contrast with this, there are many works about similarity in fuzzy logic, as [19], [20], [21], [22]. There are many differences between these proposals and ours, but the most important are (1) that we do not force the similarity relation to be reflexive, symmetric and transitive, i.e., an equivalence relation. As some of them mention, this is too restrictive for real-world applications. And (2) that we do not try to measure the closeness (or similarity) between two fuzzy propositions. The facilities we provide go in the other direction: we take the similarity value computed and return the elements considered to be similar to the one we are looking for.

The paper is structured as follows: the syntax needed

to understand it goes first (Sec. II), the description of our framework after (Sec. III) and conclusions and current work in last place (Sec. IV), as usual.

II. 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[2] 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, disjunctors and implicators are shown in Fig. 2, where Ł stands for Łukasiewicz logic, Gö for Gödel logic and prod for product logic. For aggregation operators[3] we have arithmetic mean, weighted sum or a monotone function learned from data.

III. The framework in detail

As mentioned in the introduction, we present a search engine able to answer fuzzy and flexible queries introduced by a user-friendly interface. The most important part of the interface is how the user introduces his/her queries. For that purpose we present him/her the following syntactical structure, which has been defined after studying multiple user queries. It allows to introduce all of them (sometimes with small modifications) and has the form

	conjunctor	disjunctor	implicator
Ł	$max(0, x+y-1)$	$min(1, x\ +\ y)$	$min(1, 1-x+y)$
Gö	$min(x,y)$	$max(x,y)$	y if $x\ >\ y$ else 1
prod	$x \cdot y$	$x \cdot y$	$x \cdot y$

Fig. 2: Examples of conjunctors, disjunctors and implicators

$$I'm\ \ looking\ \ for\ \ a/an\ \ \boxed{\text{individual}}$$
$$\left\{ \begin{array}{l} \boxed{\text{not}}\ \boxed{\text{m}}\ \ \boxed{\text{f-pred}} \\ \quad whose\ \ \boxed{\text{nf-pred}}\ \ \boxed{\text{co-op}}\ \ \boxed{\text{val}} \end{array} \right\} \boxed{\text{ADD}} \qquad (1)$$

where *individual* is the element we are looking for (car, skirt, restaurant, ...), *not* is a negation mechanism, *m* is a modifier (quite, rather, very, ...), $f-pred$ is a fuzzy predicate (cheap, large, close to the center, ...), $nf-pred$ is a non-fuzzy predicate (price, size, distance to the center, ...), $co-op$ 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") and *val* is a crisp value (a number if $co-op$ serves to compare numbers or a string if it takes the values "is equal to", "is different from" or "is similar to"). The elements in boxes can be modified (and in some cases left blank) and the brackets symbolize choosing between the first line (a fuzzy predicate query) or the second one (a comparison between values). The "ADD" serves to add more lines to the query, to combine multiple conditions. By default they are combining using the minimum operand, which in FL is considered as AND, but we can choose max (considered as OR), product, ... Some examples of use are "I'm looking for a restaurant close to the city center or not very expensive" (Eq. 2) and "I'm looking for a restaurant whose food type is similar to mediterranean and not close to the city center" (Eq. 3). In the examples the empty boxes mean that we do not choose any of the available elements.

$$I'm\ \ looking\ \ for\ \ a/an\ \ \boxed{\text{restaurant}}$$
$$\square\ \square\ \ \boxed{\text{close to the city center}}\ \ \boxed{\text{OR}}$$
$$\boxed{\text{not}}\ \ \boxed{\text{very}}\ \ \boxed{\text{expensive}} \qquad (2)$$

$$I'm\ \ looking\ \ for\ \ a/an\ \ \boxed{\text{restaurant}}$$
$$whose\ \ \boxed{\text{food type}}\ \ \boxed{\text{is similar to}}$$
$$\boxed{\text{mediterranean}}\ \ \boxed{\text{AND}} \qquad (3)$$
$$\boxed{\text{not}}\ \square\ \boxed{\text{close to the city center}}$$

For most of the elements in boxes the web interface presents lists of values from which we can choose one. These lists values are obtained from the real-world knowledge modelled using the framework syntax, so the framework only receives queries that it can answer. This is why when we provide the semantics of the framework syntactical constructions we do it from the point of view of what queries

[2] 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.

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

the user can pose to the search engine. 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 [23], [24], [25], [26], [27], [28]. 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 nor in the existence of other mechanisms for this purpose, as the one proposed in [29]. We simply highlight this fact so the reader knows why this structure and not some other one. Since the semantics details can be found in the papers cited, we only focus here in what matters to understand our contribution.

Suppose a program in the syntax the papers cited define and a query. In order to give an answer to the query the program is instantiated or grounded, the atoms are given an interpretation and this interpretation is extended to the formulas in the language. In bi-valued logic the interpretations are just $\{ true, false \}$ while here it is a number $\mathsf{v} \in [0,1]$, but in both cases the expected result is the maximum of the values obtained by the different rules. So, we select all the rules whose head unify with the query, take their respective interpretations (numbers between 0 and 1) and compute the maximum. This is the result for the query.

The multi-adjoint algebra is an impeccable theoretical work but, as all theoretical works, needs adaption for modelling the real world. In our case, the necessity for modelling real-world knowledge is the capability to say that we prefer the results of some rules to the results of some other ones.

In [17], [18] the authors extend the multi-adjoint semantics. The main difference with the original works is that 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 changed by $(\mathsf{p}, \mathsf{v}) \in \Omega^{(0)}$, a tuple of real numbers between 0 and 1 where $\mathsf{p} \in [0,1]$ denotes the (accumulated) priority. In order to give meaning to the new programs' syntax the authors redefine the maximum operator and extend the connectives so they can work with tuples instead of only with truth values. The new definition for the maximum is derived from the ordering definition in Def. III.1 and the extension of the connectives is done by applying the original connectives only to the truth value v and the connectives $\circ_{\&}$, $\circ_{\leftarrow}$ and $\circ_{\&}$ to the priority value, depending if the connective applied to the tuple is a conjunctor, an implicator or an aggregator[4]. We copy the definitions Def. III.1 and Def. III.2 from [17]. The only symbol not explained yet is **KT**, the set of all possible values for $(\mathsf{p}, \mathsf{v}) \in \Omega^{(0)}$.

Definition III.1 ($\preccurlyeq_{\mathbf{KT}}$)**.**

$$\begin{aligned} \bot \preccurlyeq_{\mathbf{KT}} \bot \preccurlyeq_{\mathbf{KT}} (\mathsf{p}, \mathsf{v}) \\ (\mathsf{p}_1, \mathsf{v}_1) \preccurlyeq_{\mathbf{KT}} (\mathsf{p}_2, \mathsf{v}_2) \leftrightarrow (\mathsf{p}_1 < \mathsf{p}_2) \quad or \\ (\mathsf{p}_1 = \mathsf{p}_2 \quad and \quad \mathsf{v}_1 \leq \mathsf{v}_2) \end{aligned} \tag{4}$$

[4] $\circ_{\&}$ is used for conjunctors and for aggregators. It is not a typo.

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

Definition III.2 (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

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

Now that we have introduced the basics of our formal semantics we introduce the syntax and semantics of the constructions with which we can model the world knowledge. Since our main achievement is providing a search engine capable of answering fuzzy and flexible queries we do it from the point of view of the interface: from its use to the way we define it in our framework syntax.

The first selection that the user has to do is what he/she is looking for. He/she does that by selecting a value for the field *individual*. Each value for this field is obtained from the definition of a database, which can be done by using the construction in Eq. 5. In the construction in Eq. 5 pT is the name of the database table or virtual database table (vdbt)[5] that will appear in the selection box of the field individual, pA is the arity of the vdbt, pN is the name assigned to a column of the vdbt pT and pT' is the type of the information stored in the column, (a basic type, one of $\{ boolean\_type, enum\_type, integer\_type, float\_type, string\_type \}$). We provide an example in Eq. 6 to clarify. In the example we define the restaurant database with four columns, the first for its name, the second for the food type served there, the third for the restaurant's price average and the last one for the distance to the city center from that restaurant.

$$define\_database(pT/pA, [(pN, pT')]) \tag{5}$$

$$\begin{aligned} &define\_database(\ restaurant/4, \\ &\qquad (name,\ string\_type), \\ &\qquad (food\_type,\ enum\_type), \\ &\qquad (price\_average,\ integer\_type), \\ &\qquad (distance\_to\_the\_city\_center,\ integer\_type)]). \end{aligned} \tag{6}$$

After the user chooses what he/she is looking for he/she needs to impose conditions on the search, for which the web interface presents a combo with all the available predicates. In Eq. 1 we differentiate when the predicate is fuzzy and when it is not, but the web interface does not force the user to know that. The user just has to choose one predicate and, depending if the predicate is fuzzy or not, the search engine will show him/her the fields *not* and *m* or the fields $co-op$ and *val*. The predicates shown in the combo of predicates come from different syntactical structures. We see each one of them in detail.

[5] We usually name the database "virtual database table" (vdbt) because the database that we define can be mapped to more than one database by using Prolog to database libraries. We do not enter here into these low-level details.

The first syntactical construction from which we obtain predicates for the list of predicates is the one presented in Eq. 5. In this structure we define a name and a type for each column in the vdbt and the framework interprets that we want to allow the user to impose conditions depending of the values of those columns. These predicates are all marked as non-fuzzy predicates.

The second syntactical construction from which we obtain predicates for the list of predicates is presented in Eq. 7. It serves to define the rare situation in which for all the individuals in the vdbt we have the same result and, when the construction in Eq. 8 appears as its tail, for subsets of the set of individuals in the vdbt. In Eq. 7 the variable pT mean the same as in Eq. 5, TV is the truth value (a float number between 0 and 1) and $fPredName$ is name of the fuzzy predicate we are defining. In Eq. 11 we present an example in which we say that all the restaurants are cheap with a truth value of 0.5.

$$fPredName(pT) :\sim value(TV) \quad (7)$$
$$if(\ pN(pT)\ \ comp\ \ value). \quad (8)$$
$$with\_credibility(credOp,\ credVal) \quad (9)$$
$$only\_for\_user\ 'UserName' \quad (10)$$
$$cheap(restaurant) :\sim value(0.5) \quad (11)$$

The syntactical construction in Eq. 7 is not intended to appear alone in programs, but in conjunction with one or more of the syntactical constructions that appear in Eqs. 8, 9 and 10. These constructions serve as tails for the constructions in Eqs. 7, 15, 17, 18, 20, 23, 24 and 29. These three constructions are meant to change slightly the semantics of the original constructions when they appear as their tails. We explain each case separately.

The tail in Eq. 8 (not applicable to the construction in Eq. 29) serves 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. 5, $comp$ 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 of type $integer\_type$, $enum\_type$ or $string\_type$. The only restrictions are that the type of $value$ must be the same as the one given to to the column pN of pT and that if they are of type $enum\_type$ or $string\_type$ the only available values for $comp$ are "is_equal_to" and "$is\_different\_from$". We show an example in Eq. 12 in which we say that the restaurant Zalacain is cheap with a truth value of 0.1.

$$cheap(restaurant) :\sim value(0.1)$$
$$if(name(restaurant)\ is\_equal\_to\ zalacain). \quad (12)$$

The tail in Eq. 9 serves to define a credibility for a clause, together with the operator needed to combine it with its truth value. In its syntactic definition in Eq. 9 $credVal$ is the credibility, a number of float type, and $credOp$ is the operator, any conjunctor having the properties defined in Sec. II. We show an example in Eq. 13 in which we say that the restaurant Don Jamon is cheap with a truth value of 0.3 but this rule has a credibility of 0.8 and the operator that must be used to combine the credibility with the truth value is the minimum.

$$cheap(restaurant) :\sim value(0.3)$$
$$if(name(restaurant)\ is\_equal\_to\ don\_jamon)$$
$$with\_credibility(min,\ 0.8). \quad (13)$$

The tail in Eq. 10 is aimed at defining 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, a string. We show an example in Eq. 14 in which we say that Lara considers that the restaurant Zalacain is not close to the center. So, if it is she who poses a query to the system asking for restaurants close to the city center she will obtain that the Zalacain restaurant is not.

$$close\_to\_the\_city\_center(restaurant) :\sim value(0)$$
$$if(name(restaurant)\ is\_equal\_to\ zalacain)$$
$$only\_for\_user\ 'Lara' \quad (14)$$

The last example opens up an unresolved question: how does the system knows that the result provided by the rule personalized for Lara must be chosen before the result of any other clause when the user that poses the query is she? We use for that the priorities assigned to the rules. We see this later, once all the constructions have been revised.

The third syntactical construction from which we obtain predicates for the list of predicates is presented in Eq. 15. It serves to define fuzzification functions, functions (predicates) that allow us to know how much satisfied is a fuzzy predicate for some individual stored in our database, from a non-fuzzy value that we have in the database for that individual. In Eq. 15 pN and pT mean the same as in Eq. 5, $fPredName$ is the name of the fuzzy predicate that we are defining (the fuzzification), $[(valIn,\ valOut)]$ is a list of pairs of values such that $valIn$ belongs to the domain of the fuzzification function and $valOut$ to its image[6]. An example in which we compute how cheap is a restaurant from its average price is presented in Eq. 16. The graphical representation corresponding to this example is in Fig. 3.

$$fPredName(pT) :\sim function(\ pN(\ pT\ ),$$
$$[\ (valIn,\ valOut)\ ]). \quad (15)$$
$$cheap(restaurant) :\sim function($$
$$price\_average(restaurant),$$
$$[(0,\ 1),(10,\ 1),(20,\ 0.9),(50,\ 0),(200,0)]). \quad (16)$$

The fourth syntactical construction from which we obtain predicates for the list of predicates is the one we use to define rules. Rules allow us to define the satisfaction of a fuzzy predicate from the satisfaction of other fuzzy predicates. We have two syntactical forms for defining rules, the first used

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

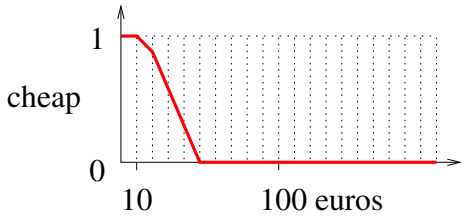


Fig. 3: Cheap function (for restaurant).

when the body depends on more than one fuzzy predicate, shown in Eq. 17, and the second one when it depends in just one, shown in Eq. 18. In Eq. 17 *aggr* is the aggregator used to combine the truth values of the fuzzy predicates in *complexBody*, which is just a conjunction of names of fuzzy predicates (and the vdbt they are associated to, represented by pT), while in Eq. 18 *simplexBody* is just the name of a fuzzy predicate (and the vbdt it is associated to). In both of them pT means the same as in Eq. 5 and $fPredName$ the same as in Eq. 15. We show an example in Eq. 19 in which we say that a restaurant is a tempting restaurant depending on the worst value it has between being close to the center and being cheap, which means that a restaurant must be close to the center and cheap at the same time to consider it a tempting restaurant.

$$fPredName(pT) :\sim rule(aggr, complexBody) \quad (17)$$

$$fPredName(pT) :\sim rule(simpleBody) \quad (18)$$

$$tempting\_restaurant(restaurant) :\sim rule(\ min, (\ close\_to\_the\_city\_center(restaurant), cheap(restaurant)\ )) \quad (19)$$

The fifth syntactical construction from which we obtain predicates for the list of predicates is presented in Eq. 20. It is the one used to define default values for fuzzy computations. Its main goal is to avoid that the inference process stops when a needed value is missing and it is really useful when a database can have null values. In Eq. 20 pT means the same as in Eq. 5, $fPredName$ the same as in Eq. 15 and TV the same as in Eq. 7. We provide two examples in Eqs. 21 and 22 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) and that, in absence of information, a restaurant is considered to be medium cheap[7].

$$fPredName(pT) :\sim defaults\_to(TV) \quad (20)$$

$$close\_to\_the\_city\_center(restaurant) :\sim defaults\_to(0). \quad (21)$$

$$cheap(restaurant) :\sim defaults\_to(0.5). \quad (22)$$

The last two constructions from which we obtain predicates for the list of predicates are for defining the satisfaction of a fuzzy predicate from the satisfaction of another that is considered to be a synonym or an antonym of the first one. The syntax for defining a fuzzy predicate from a synonym is shown in Eq. 23 and the one for defining it from an antonym in Eq. 24. In Eqs. 23 and 24 pT means the same as in Eq. 5 and $fPredName$ the same as in Eq. 15, while $fPredName2$ is the fuzzy predicate from which we are defining the synonym or antonym. Its name must be different from $fPredName$. In the examples in Eqs. 25 and 26 we define an unexpensive restaurant as a cheap restaurant and an expensive one as the opposite of a cheap one.

$$fPredName(pT) :\sim synonym\_of(fPredName2(pT)) \quad (23)$$

$$fPredName(pT) :\sim antonym\_of(fPredName2(pT)) \quad (24)$$

$$unexpensive(restaurant) : synonym\_of(cheap(restaurant), prod, 1). \quad (25)$$

$$expensive(restaurant) : antonym\_of(cheap(restaurant), prod, 1). \quad (26)$$

Once the user has chosen between the available predicates the one[8] he wants to use to impose a condition to his/her search the application determines if this predicate is fuzzy or not. If it is a non-fuzzy predicate the web interface shows him/her the fields for entering the values for the $co-op$ and *val* variables. As introduced before, the available values for $co-op$ depend on the type of the predicate chosen. If it is of type *enum_type* or *string_type* they are "is equal to", "is different from" and "is similar to". If it is of *boolean_type* they are "is equal to" and "is different from". And if it is of *integer_type* of *float_type* they are "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". The available values for *val* are determined too when the predicate type is *boolean_type* or *enum_type*. In that cases the interface shows a selection box instead of a free text input field. When the predicate that the user chooses is a fuzzy predicate the web interface presents him/her the selection fields for *not* and *m*, from which he can select to negate the result obtained by the predicate, apply to it a modifier (very, rather, ...), both or none.

By selecting or introducing the values for $co-op$ and *val* or for *not* and *m* the query line is complete and the user has to press the search button and wait for the framework answers. Before entering into how the framework decides which is the result when two or more rules give answers to the query we introduce two more syntactical constructions. These two syntactical constructions complement the knowledge introduced by the programmer by using the previous ones. The first one is for defining new modifiers (the ones pre-defined in the framework are "rather", "very" and "a few") and the other one is for defining the similarity between values of type *enum_type*. The syntax of the first one is shown in Eq. 27, where *predName* is the name of the modifier, *TV_In* the truth value computed by the fuzzy predicate selected by the user and *TV_Out* the resultant truth value after applying the modifier. Eq. 28 is an example of use in which we define the truth value obtained by the modifier "too much"

[7]We include two examples here so if one builds a program by taking all the examples in the contribution the rule in Eq. 19 the framework is able to obtain results for all the restaurants in our database.

[8]He/she can use more than one predicate for imposing conditions, but each predicate must be in its own query line. The "ADD" button serves to add new lines to the query.

as three times the truth value it receives as input. The syntax for defining similarity between values of type *enum_type* is shown in Eq. 29, where pT and pT' mean the same as in Eq. 5, TV the same as in Eq. 7 and $value1$ and $value2$ are two values for the vdbt column pT' of the vdbt pT. In the example in Eq. 30 we say that the food type mediterranean is 0.6 similar to the spanish one (but not in the other way. If we want to say that the spanish food is 0.6 similar to the mediterranean one we need to add another line of code saying that).

$$modifier(predName/2,\ TV\_In,\ TV\_Out)\ :- \\ <<\ Prologcode\ >> \quad (27)$$

$$modifier(too\_much/2,\ TV\_In,\ TV\_Out)\ :- \\ TV\_Out\ .=.\ (TV\_In * TV\_In * TV\_In). \quad (28)$$

$$similarity\_between(pT,\ pT'(value1),\ pT'(value2),\ TV). \quad (29)$$

$$similarity\_between(\ restaurant, \\ food\_type(mediterranean),\ food\_type(spanish), \\ 0.6). \quad (30)$$

The framework interprets the constructions presented and provides the web interface with the information it needs to present the user the query form in Eq. 1. As told before, the framework is able to answer any user query introduced by using this query form. The remaining question is how the framework is able to decide when two or more rules (or clauses) provide answers to the query which one is the expected one. In the introduction to the multi-adjoint semantics with priorities we talk about the inclusion of a parameter p for managing priority values, a new maximum operator that takes into account this new parameter and the extension of the connectives to manage it. This maximum operator is what the framework uses to decide between two tuples $(\mathsf{p},\ \mathsf{v}) \in \Omega^{(0)}$ the expected one. The value of the variable v can be constant or computed at execution time from the information introduced by the programmer, but it is not the same for the value of the priority variable p. The value of the priority variable p is fixed for each construction but modifiable when the programmer uses the tails' constructions in Eqs. 8 and 10. The values given to it depending on the construction used are summarized in the table in Fig. 4. Without going into too much detail, we want to give more importance to rules for computing fuzzy values, followed by fuzzifications, rules and default values in last place. The reason for assigning the lowest value for synonyms and antonyms is that we want the capability to define a fuzzy predicate as a synonym (or antonym) to some other one, but overwrite the values returned by this other predicate whenever we need to.

The tails' constructions in Eqs. 8 and 10 slightly modify the values for the priority variable p shown in Fig. 4. The modifications consist in increasing the value of the priority variable p in 0.05 and 0.1 respectively. The interest in doing this is giving more importance to the clauses or rules conditioned, and even more to the personalized ones, so we can define a general behaviour and specialize it whenever we need to do it. We can even define conditioned and personalized rules and they will be chosen before the ones just personalized.

construction	p
fuzzy value	0.8
fuzzification function	0.6
fuzzy rule	0.4
default fuzzy value	0.2
synonyms/antonyms	0

Fig. 4: Values for the priority variable p.

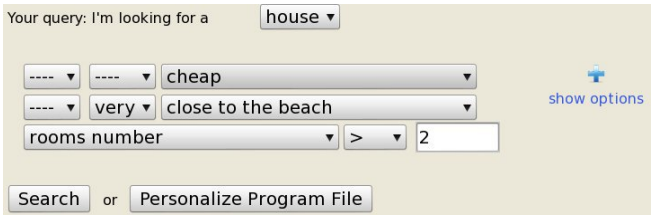
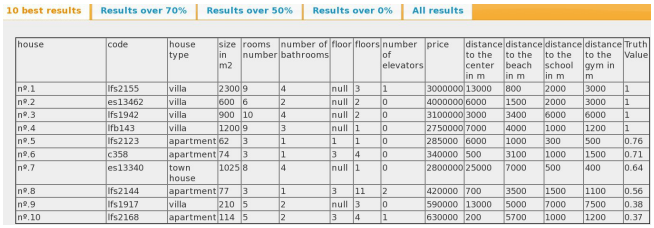


Fig. 5: Query example.



house	code	house type	size in m2	rooms number	number of bathrooms	floor	floors	number of elevators	price	distance to the center in m	distance to the beach in m	distance to the school in m	distance to the gym in m	Truth Value
nº.1	lfs2155	villa	2300	9	4	null	3	1	3000000	13000	800	2000	3000	1
nº.2	es13462	villa	600	6	2	null	2	0	4000000	6000	1500	2000	3000	1
nº.3	lfs1942	villa	900	10	4	null	2	0	3100000	3000	3400	6000	6000	1
nº.4	lfb143	villa	1200	9	3	null	1	0	2750000	7000	4000	1000	1200	1
nº.5	lfs2123	apartment	62	3	1	1	1	0	285000	6000	1000	300	500	0.76
nº.6	c358	apartment	74	3	1	3	4	0	340000	500	3100	1000	1500	0.71
nº.7	es13340	town house	1025	8	4	null	1	0	2800000	25000	7000	500	400	0.64
nº.8	lfs2144	apartment	77	3	1	3	11	2	420000	700	3500	1500	1100	0.56
nº.9	lfs1917	villa	210	5	2	null	3	0	590000	13000	5000	7000	7500	0.38
nº.10	lfs2168	apartment	114	5	2	3	4	1	630000	200	5700	1000	1200	0.37

Fig. 6: Answers returned for the query example in Fig. 5.

IV. Conclusions

We present a framework for modelling the real world knowledge and a fuzzy and flexible search engine. The first one has a syntax (and its semantics) with which we can capture the relations between the fuzzy and non-fuzzy knowledge of any domain and feed the search engine with the information it needs to provide a friendly and easy to use user interface.

On one hand we provide syntactical constructions for representing fuzzy and non-fuzzy knowledge about the world, with the possibility to represent the existing links between the satisfaction of fuzzy predicates and the non-fuzzy information stored in databases. This includes the assignment of truth values to individuals in our database that satisfy some condition, the fuzzification of values in the database, the use of rules to obtain truth values when they depend on some other fuzzy predicates, the assignment of default truth values to individuals (mainly to avoid that a nonexistent or null value in the database stops the computation), the definition of fuzzy predicates from some others considered to be synonyms or antonyms of the first ones and the representation of the (almost always fuzzy) similarity between values in our database (so the user can ask for individuals with a value

similar to the one he/she is looking for). These syntactical structures, joint with the possibility to include pure Prolog code in the programs (for performing complex tasks) makes our framework a very powerful tool for representing the real world and answering questions about it.

On the other one we present a search engine that takes advantage of all the knowledge introduced by using the framework syntax. Its main advantage over the existing ones is that we avoid the necessity to learn a complex syntax to just pose (fuzzy) queries, without the cost of not being able to answer some queries because the framework is not able to understand them. An example is shown in Figs. 5 and 6.

A link to a beta version of our flexible search engine (with example programs, the possibility to upload new ones, etc) is available at *https://moises.ls.fi.upm.es/java-apps/flese/*

Our current research focus on deriving similarity relations from the information in the database. In this way we could, for example, determine from the RGB composition of two colors if they are similar or not and remove the necessity to code this knowledge in the programs.

REFERENCES

[1] P. Bosc and O. Pivert, "Sqlf: a relational database language for fuzzy querying," *Fuzzy Systems, IEEE Transactions on*, vol. 3, no. 1, pp. 1 –17, feb 1995.

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

[3] D. Dubois and H. Prade, "Using fuzzy sets in flexible querying: why and how?" in *Flexible query answering systems*, T. Andreasen, H. Christiansen, and H. L. Larsen, Eds. Norwell, MA, USA: Kluwer Academic Publishers, 1997, pp. 45–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=285506.285510

[4] P. Vojtáš, "Fuzzy logic programming," *Fuzzy Sets and Systems*, vol. 124, no. 3, pp. 361–370, 2001.

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

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

[7] D. Li and D. Liu, *A fuzzy Prolog database system*. New York, NY, USA: John Wiley & Sons, Inc., 1990.

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

[9] P. Morcillo and G. Moreno, "Floper, a fuzzy logic programming environment for research," in *Proceedings of VIII Jornadas sobre Programación y Lenguajes (PROLE'08)*, F. U. de Oviedo, Ed., Gijón, Spain, october 2008, pp. 259–263.

[10] C. Vaucheret, S. Guadarrama, and S. Muñoz-Hernández, "Fuzzy prolog: A simple general implementation using CLP(R)," in *LPAR*, ser. Lecture Notes in Artificial Intelligence, M. Baaz and A. Voronkov, Eds., vol. 2514. Springer, 2002, pp. 450–464.

[11] S. Guadarrama, S. Muñoz-Hernández, and C. Vaucheret, "Fuzzy prolog: a new approach using soft constraints propagation," *Fuzzy Sets and Systems (FSS)*, vol. 144, no. 1, pp. 127 – 150, 2004, possibilistic Logic and Related Issues.

[12] S. Muñoz-Hernández, V. Pablos-Ceruelo, and H. Strass, "Rfuzzy: Syntax, semantics and implementation details of a simple and expressive fuzzy tool over prolog," *Information Sciences*, vol. 181, no. 10, pp. 1951 – 1970, 2011, special Issue on Information Engineering Applications Based on Lattices. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0C-50PJWYR-2/2/26d8ff890f0effc98aa1c12225a5fb87

[13] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965.

[14] R. A. Ribeiro and A. M. Moreira, "Fuzzy query interface for a business database," *International Journal of Human-Computer Studies*, vol. 58, no. 4, pp. 363 – 391, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1071581903000107

[15] P. Bosc and O. Pivert, "On a strengthening connective for flexible database querying," in *Fuzzy Systems (FUZZ), 2011 IEEE International Conference on*, 2011, pp. 1233–1238.

[16] G. Bordogna and G. Pasi, "A fuzzy query language with a linguistic hierarchical aggregator," in *Proceedings of the 1994 ACM symposium on Applied computing*, ser. SAC '94. New York, NY, USA: ACM, 1994, pp. 184–187. [Online]. Available: http://doi.acm.org/10.1145/326619.326693

[17] V. Pablos-Ceruelo and S. Muñoz-Hernández, "Introducing priorities in rfuzzy: Syntax and semantics," in *CMMSE 2011 : Proceedings of the 11th International Conference on Mathematical Methods in Science and Engineering*, vol. 3, Benidorm (Alicante), Spain, June 2011, pp. 918–929. [Online]. Available: http://gsii.usal.es/~CMMSE/index.php?option=com_content\&task=view\&id=15\&Itemid=16

[18] ——, "Getting answers to fuzzy and flexible searches by easy modelling of real-world knowledge," in *FCTA'2013: Proceedings of the 5th International Conference on Fuzzy Computation Theory and Applications*, 2013.

[19] J.-B. Wang, Z.-Q. Xu, and N.-C. Wang, "A fuzzy logic with similarity," in *Proceedings of the 2002 International Conference on Machine Learning and Cybernetics*, vol. 3, 2002, pp. 1178 – 1183 vol.3. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1167386&tag=1

[20] L. Godo and R. O. Rodriguez, "A fuzzy modal logic for similarity reasoning," in *Fuzzy Logic and Soft Computing*, K.-Y. C. Guoqing Chen, Mingsheng Ying, Ed. Kluwer Academic, 1999. [Online]. Available: http://publicaciones.dc.uba.ar/Publications/1999/GR99

[21] D. Dubois and H. Prade, "Comparison of two fuzzy set-based logics: similarity logic and possibilistic logic," in *Fuzzy Systems, 1995. International Joint Conference of the Fourth IEEE International Conference on Fuzzy Systems and The Second International Fuzzy Engineering Symposium., Proceedings of 1995 IEEE Int*, vol. 3, 1995, pp. 1219–1226.

[22] F. Esteva, P. Garcia, L. Godo, E. Ruspini, and L. Valverde, "On similarity logic and the generalized modus ponens," in *Fuzzy Systems, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the Third IEEE Conference on*, 1994, pp. 1423–1427 vol.2.

[23] J. Medina, M. Ojeda-Aciego, and P. Vojtáš, "A multi-adjoint approach to similarity-based unification," *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 5, pp. 70 – 85, 2002, uNCL'2002, Unification in Non-Classical Logics (ICALP 2002 Satellite Workshop). [Online]. Available: http://www.sciencedirect.com/science/article/B75H1-4DDWJ13-37/2/bdc92744d6ddc8e888ea314efb711107

[24] ——, "A completeness theorem for multi-adjoint logic programming," in *FUZZ-IEEE*, 2001, pp. 1031–1034.

[25] ——, "Multi-adjoint logic programming with continuous semantics," in *LPNMR*, ser. Lecture Notes in Computer Science, T. Eiter, W. Faber, and M. Truszczynski, Eds., vol. 2173. Springer, 2001, pp. 351–364.

[26] ——, "A procedural semantics for multi-adjoint logic programming," in *EPIA*, ser. Lecture Notes in Computer Science, P. Brazdil and A. Jorge, Eds., vol. 2258. Springer, 2001, pp. 290–297.

[27] ——, "Similarity-based unification: a multi-adjoint approach," *Fuzzy Sets and Systems*, vol. 146, no. 1, pp. 43–62, 2004.

[28] J. Medina Moreno and M. Ojeda-Aciego, "On first-order multi-adjoint logic programming," in *11th Spanish Congress on Fuzzy Logic and Technology*, 2002. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.3800

[29] A. M. Palacios, M. J. Gacto, and J. Alcalá-Fdez, "Mining fuzzy association rules from low-quality data," *Soft Comput.*, vol. 16, no. 5, pp. 883–901, 2012.