A Framework for Modelling Real-World Knowledge Capable of Obtaining Answers to Fuzzy and Flexible Searches

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: we need human intervention to determine how the non-fuzzy information stored needs to be combined and processed to answer a fuzzy query. We present a web interface for posing fuzzy and flexible queries and a framework. Our framework allows to represent non-fuzzy concepts, fuzzy concepts and relations between them, giving the programmer the capability to model any real-world knowledge. It is this representation in the framework's language what it uses to (1) determine how to answer the query without any human intervention and (2) provide the search engine with the information it needs to present the user a friendly and easy to use query form. We expect this work contributes to the development of more human-oriented fuzzy search engines.

Keywords Search engine · Fuzzy logic · Framework

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.

V. Pablos-Ceruelo · S. Munoz-Hernandez (✉)
The Babel Research Group, Facultad de Informática,
Universidad Politécnica de Madrid, Madrid, Spain
e-mail: susana@babel.ls.fi.upm.es
url: http://babel.ls.fi.upm.es

V. Pablos-Ceruelo
e-mail: vpablos@babel.ls.fi.upm.es

K. Madani et al. (eds.), *Computational Intelligence*,
Studies in Computational Intelligence 613,
DOI 10.1007/978-3-319-23392-5_16

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 some houses to the center and the user query "I want a house close to the center". Assuming that it is nonsense to teach every search engine user how to translate the (almost always) fuzzy query the user has 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 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 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 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

[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.

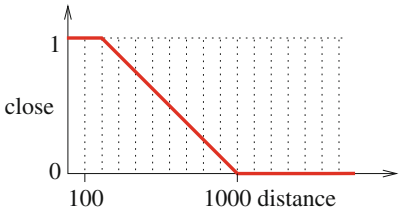

Fig. 1 Restaurants database and close fuzzification function

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, Ni Hao is "hardly" 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.

Name	Distance	Price avg.	Food type
Il_tempietto	100	30	Italian
Tapasbar	300	20	Spanish
Ni Hao	900	10	Chinese
Kenzo	1200	40	Japanese

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 [4], the SQLf language. The Ph.D. thesis of Leonid Tineo [20] and the work of Dubois and Prade [6] 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 [22], we know about the Prolog-Elf system [8], the FRIL Prolog system [1], the F-Prolog language [9], the FuzzyDL reasoner [2], the Fuzzy Logic Programming Environment for Research (FLOPER) [15], the Fuzzy Prolog system [7, 21], or Rfuzzy [17]. All of them implement in some way the fuzzy set theory introduced by Lotfi Zadeh in 1965 [23], 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 Sect. 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 [3, 5, 19]. 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?".

The paper is structured as follows: the syntax needed to understand it goes first (Sect. 2), the description of our framework after (Sect. 3) and conclusions and current work in last place (Sect. 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* $\&_1, \&_2, \ldots, \&_k$, *disjunctions* $\vee_1, \vee_2, \ldots, \vee_l$, *implications* $\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m$, *aggregations* $@_1, @_2, \ldots, @_n$ and tuples of real numbers in the interval $[0, 1]$ represented by (p, 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 (p, v) and $(\hat{\mathsf{p}}, \hat{\mathsf{v}})$ 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 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 ($\hat{F}(x, y) = y \mathrm{if} x > y \mathrm{else} 1$), in product logic ($\hat{F}(x, y) = x \cdot y$) and for aggregation operators[3]: 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 (2.1) the search capabilities and (2.2) the metainformation that the web application needs to present the user the tools he/she needs to pose the query and (3) a web application that (3.1) reads the metainformation, (3.2) determines the framework capabilities, (3.3)

[2] As usually, a n-ary function $\hat{F}$ is called *monotonic in the i* define-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.

generates an easy to use human-oriented interface for posing queries to the search engine and (3.4) shows the answers found by the framework to the user.

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

$$I'm\ looking\ for\ a/an\ \boxed{\text{individual}} \quad \left\{ \begin{array}{l} \boxed{\text{not}}\ \boxed{\text{quantifier}}\ \boxed{\text{fuzzy-pred}} \\ whose\ \boxed{\text{non-fuzzy-pred}}\ \boxed{\text{comp-op}}\ \boxed{\text{value}} \end{array} \right\} \boxed{\text{AND}} \tag{1}$$

where *individual* is the element we are looking for (car, skirt, restaurant, …), *quantifier* is a quantifier (quite, rather, very, …), *fuzzy-pred* is a fuzzy predicate (cheap, large, close to the center, …), *non-fuzzy-pred* is a non-fuzzy predicate (price, size, distance to the center, …) and *comp-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). 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). In the examples the empty boxes mean that we do not choose any of the available elements.

$$I'm\ looking\ for\ a/an\ \boxed{\text{restaurant}} \quad \boxed{\text{not}}\ \boxed{\text{very}}\ \boxed{\text{near the city center}}\ \square \tag{2}$$

$$I'm\ looking\ for\ a/an\ \boxed{\text{restaurant}} \quad whose\ \boxed{\text{food type}}\ \boxed{\text{is}}\ \boxed{\text{mediterranean}}\ \square \tag{3}$$

$$I'm\ looking\ for\ a/an\ \boxed{\text{restaurant}} \quad whose\ \boxed{\text{food type}}\ \boxed{\text{is similar to}}\ \boxed{\text{mediterranean}}\ \boxed{\text{AND}} \quad \square\square\ \boxed{\text{near the city center}} \tag{4}$$

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", "quantifier", "fuzzy-pred", "non-fuzzy-pred", "comp-op" 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 [10–14, 16]. 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 [18]. Definitions needed to understand the formal semantics are given in advance, as usually.

In [18] 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 1 ($\preccurlyeq_{\mathbf{KT}}$)

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

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

Definition 2 (*Multi-Adjoint Logic Program*) A multi-adjoint logic program is a set of clauses of the form

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

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 $=, \neq, \geq, \leq, >$ 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 (*Valuation, Interpretation*) A *valuation* or *instantiation* $\sigma : V \to \mathbf{HU}$ is an assignment of ground terms to variables and uniquely constitutes a mapping $\hat{\sigma} : \mathrm{TB}_{\Pi,\Sigma,V} \to \mathbf{HB}$ 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.[4]

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 $\mathcal{I}_{\mathbf{P}}$.

Definition 4 (*The operator* $\circ$) The application of some conjunctor $\bar{\&}$ (resp. implicator $\bar{\leftarrow}$, aggregator $\bar{@}$) to elements $(\mathsf{p},\ \mathsf{v}) \in \mathbf{KT}\backslash\{\bot\}$ 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 5 (*Satisfaction, Model*) Let **P** be a multi-adjoint logic program, $I \in \mathcal{I}_{\mathbf{P}}$ 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

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

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" or "houses" 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 write the lines in Eqs. 8–12 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.

[4] 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 $(\mathsf{p},\ \mathsf{v}) \in \mathbf{KT}\backslash\{\bot\}$ (we omit those atoms whose interpretation is the truth value $\bot$).

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

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

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

$$:-sql\_persistent(rdttcc(integer,\ integer),\ rdttcc(id,\ dttcc),\ rdttcc). \tag{11}$$

$$restaurant(id, ft, dttcc) :- rft(id,\ ft),\ rdttcc(id,\ dttcc). \tag{12}$$

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

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

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,[5] 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 five columns (or the predicate has five arguments), the first for the unique identifier given to each restaurant (its name), the second for the food type served there, the third for the number of years since its opening, the fourth for the restaurant's price average and the last one for the distance to the city center from that restaurant.

$$define\_database(pT/pA, [(pN, pT')]) \tag{15}$$

$$define\_database((id,\ string\_type),\ (food\_type,\ enum\_type),\ (years\_since\_opening,\ integer\_type),\ (price\_average,\ integer\_type),\ (distance\_to\_the\_city\_center,\ integer\_type)]). \tag{16}$$

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. We go first for the setters/getters. For each column defined for a vdbt the framework builds for us a setter/getter to store/access the information in/of each cell in the database. The cell selected gets fully determined by the predicate name (the one given to the column) and its first argument. For example, by writing the line in Eq. 16 the framework defines for us the

[5]Please note that the types in our framework are not the same as the types used in Eqs. 8–14. Nevertheless, our types are subsets of this ones. We justify in the paragraph below this one why we need this fine-grained type control.

predicates $id(R, Id)$, $food\_type(R, FoodType)$, $years\_since\_opening(R, Years)$, $price\_average(R, Price)$ and $distance\_to\_the\_city\_center(R, Dttcc)$. Each one serves to set/obtain the value in/from the database cell corresponding to the row of restaurant R and the column with the same name as the predicate used (id, $food\_type$, $years\_since\_opening$, $price\_average$ and $distance\_to\_the\_city\_center$). With respect to the web interface, the framework notifies to it that we have a new value for the field "individual" (the value in pT, restaurant in the example), a list of values for *non-fuzzy-pred* (id, food type, years since opening, price average and distance to the city center) and their types (string_type, enum_type, integer_type, integer_type, integer_type). In addition to this explicit information the web interface itself is capable of deriving from the type of each column the values that it can show in *comp-op*. We show them in the table below. It is even able to detect in some cases that it must ask the framework for the values of some field, as in the case of the selection for the field *comp-op* the value "is similar to".

Type	Values for *comp-op*
$string\_type$	"is equal to" and "is different from"
$enum\_type$	"is equal to", "is different from" and "is similar to"
$interger\_type$	"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 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 between 0 and 1) 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 0.7 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 appear too as tails of the constructions in Eqs. 17, 23, 27, 31, 32 and 36, we dedicate some paragraphs (just 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 p, v, $\&_i$ and *COND* we have by default[7] the values 0.8, 1, *product* and *true*. We show in Eq. 22 the translation of the example in Eq. 20 for the reader to see how the

[6] Be careful, we are not saying that the spanish food is 0.7 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 paragraphs after this one.

translation is done in practice. This construction does not provide any information to the web interface.

$$similarity\_between(pT,\ pN(V1),\ pN(V2),\ TV) \tag{17}$$

$$with\_credibility(credOp,\ credVal) \tag{18}$$

$$only\_for\_user\ 'UserName' \tag{19}$$

$$similarity\_between(restaurant,\ food\_type(mediterranean),\ food\_type(spanish), 0.7) \tag{20}$$

$$similarity(pT(pN(V1, V2))) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} TV \ \text{if}\ COND \tag{21}$$

$$similarity(restaurant(food\_type(mediterranean, spanish))) \xleftarrow{(0.8,\ 1),\ prod} 0.7 \ \text{if}\ true \tag{22}$$

As outlined in the previous paragraph, the constructions in Eqs. 18 and 19 can be used as tails for the constructions in Eqs. 17, 23, 27, 31, 32 and 36. There is another construction that can be used as tail, the one in Eq. 24, but only for the constructions in Eqs. 23, 27, 31, 32 and 36. This three constructions are meant to change slightly the semantics of the original constructions when they appear as their tails, which is done by modifying at least one of the values given "by default" to the variables p, v, $\&_i$ and *COND*. We explain each case separately.

The tail in Eq. 18 serves to (re)define the credibility of a clause, together with the operator needed to combine it with its truth value. In its syntactic definition in Eq. 18 *credVal* is the credibility, a number of float type, and *credOp* is the operator, any conjunctor having the properties defined in Sect. 2. When we use it the values for v and $\&_i$ (usually 1 and *product*) are changed by the values given to the variables *credVal* and *credOp*.

The tail in Eq. 19 is aimed at defining personalized rules, rules that only apply when the name of the user logged in and the user name in the rule are the same one. In the construction *Username* is the name of any user, a string. When we use it the value of *COND* is replaced[8] by $COND \wedge currentUser(Me) \wedge Me = 'UserName'$[9] and the value for p gets increased by 0.1. While the first change is to ensure that the rule is only used when the logged user is the selected user, the second one is to ensure that, when the logged user is the selected user, this rule (considered to be more specialized for the selected user) is chosen before another rule not having this specialization.

The tail in Eq. 24 (not applicable to the construction in Eq. 17) 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. 15, *cond* can take the values is_equal_to,

[8]Please note that we not remove the original condition, so we can combine conditions introduced by the semantics of a clause with the conditions introduced by one or more tails.

[9]We use indistinctively ',' and $\wedge$ because the first one is the Prolog symbol for conjunction.

$is\_different\_from$, 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 $cond$ are is_equal_to and $is\_different\_from$. When we use this tail construction the value of $COND$ is changed by $COND \wedge (pN(Individual)\ condvalue)$, where $Individual$ is basically a vdbt row (of type pT), and the value for p gets increased by 0.05.

The first tail construction, the one in Eq. 18, is aimed at changing the clause credibility. This is why it only changes the credibility value and the credibility operator in the "by default" semantics (of the clause in which it appears as tail). On the contrary the tails constructions in Eqs. 19 and 24 have as purpose increasing the specialization of the clause. The first one defines that the user prefers the results of this clause to the results of any other clause and the second one defines that, for the subset of individuals of our clause domain delimited by the condition, we prefer the results provided by this clause to the results provided by any other clause. This justifies in part the increasing of the value of p by 0.1 when the clause contains the tail in Eq. 19 and by 0.05 when it is the one in Eq. 24. The missing part, the cause of defining different values for each, comes from a design decision: when choosing between the results of a personalized clause and the ones of a clause defined for a subset of individuals we prefer the first ones. Furthermore, the use of one of the tails' constructions does not disallow the use of the other ones, so we can have personalized rules for a subset of individuals of the clause's domain. And with a defined credibility.

The third construction (shown in Eq. 23) is the one used to define the result of a fuzzy predicate (*fPredName*) when this one is applied to an individual in the selected vdbt (*pT*). 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. 24 appears as its tail, for subsets of the set of individuals in the vdbt. In Eq. 23 the variables *pT* and *TV* mean the same as in Eqs. 15 and 17 and *fPredName* is the fuzzy predicate we are defining. Equation 25 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. 26, where *Individual* is a variable representing the vdbt individual for which the clause will be computed (a restaurant in the example). The default values for p, v, $\&_i$ and *COND* are the values 0.8, 1, $product$ 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 the field *fuzzy-pred* (see Eq. 1).

$$fPredName(pT) :\sim value(TV) \tag{23}$$

$$if(pN(pT)\ \ cond\ \ value). \tag{24}$$

$$cheap(restaurant) :\sim value(0.1)$$

$$if(id(restaurant)\ is\_equal\_to\ zalacain). \tag{25}$$

$$fPredName(Individual) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} TV \ \text{ if } \ COND \tag{26}$$

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. 27, where *pN* and *pT* mean the same as in Eq. 15, *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 and $valOut$ to its image.[10] An example in which we compute how much traditional is a restaurant from the number of years since its opening is presented in Eq. 28. The formal semantics for this construction are shown in Eq. 29, but only for one sequence of two contiguous points[11] $(valIn1, valOut1)(valIn2, valOut2)$ in Eq. 27. The default values for p, v, $\&_i$ and *COND* are the values 0.6, 1, $product$ and the $COND'$ in Eq. 30. This value for $COND, COND'$, serves to limit the domain of the generated clause. Since we generate one clause for each piece of the piecewise function we use $COND'$ to ensure that we use the clause designated for the piece our input value belongs to. The web interface assumes that fuzzification functions are 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.

$$fPredName(pT) :\sim function(pN(pT), [(valIn, valOut)]) \tag{27}$$

$$traditional(restaurant) :\sim function(years\_since\_opening(restaurant), [(0, 0), (5, 0.1), (10, 0.4), (15, 1), (100, 1)]). \tag{28}$$

$$fPredName(Individual) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} pN(Individual) * \frac{(valOut\_2 - valOut\_1)}{(valIn\_2 - valIn\_1)} \quad \text{if} \quad COND \tag{29}$$

$$COND' = (valIn\_1 < pN(Individual) \leq valIn\_2) \tag{30}$$

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 depends in 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 (and the vdbt they are associated to, represented by *pT*), while in Eq. 32 *simplexBody* 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. 27. We show an example in Eq. 35. The formal semantics for this constructions are respectively shown in Eqs. 33 and 34 and the default values for p, v, $\&_i$ and *COND* are the values 0.4, 1, *product* and *true*. With respect to what the web interface receives from this syntactic structure, it considers fuzzy rules as fuzzy predicates, and it always includes fuzzy predicates in the list of available predicates for the field *fp* (see Eq. 1) when they are not there yet.

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

[11] This "only for one sequence of two contiguous points" means that we generate one clause of the form in Eq. 29 for each piece defined by two contiguous points.

$$fPredName(pT) :\sim rule(aggr, complexBody) \tag{31}$$

$$fPredName(pT) :\sim rule(simpleBody) \tag{32}$$

$$fPredName(Individual) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} aggr(complexBody) \text{ if } COND \tag{33}$$

$$fPredName(Individual) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} simplexBody \text{ if } COND \tag{34}$$

$$tempting\_restaurant(restaurant) :\sim rule(min, (near\_the\_city\_center(restaurant), cheap(restaurant))) \tag{35}$$

The sixth syntactical construction 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. The syntactic form is presented in Eq. 36, where pT means the same as in Eq. 15 and $fPredName$ the same as in Eq. 27. We provide two examples in Eqs. 37 and 38 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.[12] The formal semantics for this constructions are shown in Eq. 39 and the default values for p, v, $\&_i$ and *COND* are the values 0, 1, *product* and *true*. With respect to what the web interface receives from this syntactic structure, the syntactic construction for defining default values is translated as a fuzzy predicate 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.

$$fPredName(pT) :\sim defaults\_to(TV) \tag{36}$$

$$near\_the\_city\_center(restaurant) :\sim defaults\_to(0). \tag{37}$$

$$cheap(restaurant) :\sim defaults\_to(0.5). \tag{38}$$

$$fPredName(Individual) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} TV \text{ if } COND \tag{39}$$

The six constructions defined before and their semantics orchestrate the intended meaning we wanna give to our programs. We summarize in the table below the values given to the variables p, v, $\&_i$ and *COND* when no tail is attached to the no-tail constructions and explain now the reasons for choosing the values that appear there. For the variables v and $\&_i$ we assign by default the values 1 and *product*. We have chosen this values due to the fact that multiplying by one a number (the rule's truth value) results always in the same number: it does not affect the clause's result. Their value is only changed when the construction in Eq. 18 is used as tail, which means that the programmer wants to change the default credibility of the clause. The variable *COND* has as goal avoiding the clause from obtaining results when the

[12] We include two examples here so if one builds a program by taking all the examples in the contribution the rule in Eq. 35 does not fail to obtain answers because it has not enough information to infer results.

condition is not satisfied. The only construction that needs by default this behaviour is the fuzzification, as explained before. For the other constructions we assign by default the value *true* and modify it when any of the constructions in Eqs. 19 and 24 appears as tails, which affects too to the value of the variable p. The value given to the variable p is used to decide when more than one rule returns results which ones are the valid ones. The only restriction in the selection of its default value is that its range of values is [0, 1]. Since the tails' constructions in Eqs. 19 and 24 can add to it a maximum of 0.15 our default value for p must be always below or equal to 0.85 to ensure that we satisfy the restriction.

Construction	p	v	$\&_i$	*COND*
Similarity between individuals	0.8	1	Product	True
Fuzzy value	0.8	1	Product	True
Fuzzification function	0.6	1	Product	$COND'$
Fuzzy rule	0.4	1	Product	True
Default fuzzy value	0	1	Product	True

4 Conclusions

We present a framework for modelling the real world knowledge and a web interface for posing fuzzy and flexible queries. As introduced before, 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 (inclusive the linking of information from databases with real-world fuzzy concepts) and feed the search engine with the information it needs to provide a friendly and easy to use user interface. The search engine main advantage over the existing ones just derives from this: we avoid the necessity to learn a complex syntax to just pose (fuzzy) queries. This, joint with the possibility to include Prolog code (for complex tasks) makes our framework a very powerful tool for representing the real world and answering questions about it. A link to a beta version of our flexible search engine (with example programs, the possibility to upload new ones, etc.) is available at our web page.

Our current research focus on deriving similarity relations from the modelization of a problem in our framework's language. In this way we could, for example, derive from the RGB composition of two colors their similarity relation.

References

1. Baldwin, J.F., Martin, T.P., Pilsworth, B.W.: Fril—Fuzzy and Evidential Reasoning in Artificial Intelligence. Wiley, New York (1995)
2. Bobillo, F., Straccia, U.: FuzzyDL: an expressive fuzzy description logic reasoner. In: 2008 International Conference on Fuzzy Systems (FUZZ-08), pp. 923–930. IEEE Computer Society (2008)
3. Bordogna, G. Pasi, G.: A fuzzy query language with a linguistic hierarchical aggregator. In: Proceedings of the 1994 ACM Symposium on Applied computing, SAC'94, pp. 184–187. ACM, New York (1994)
4. Bosc, P., Pivert, O.: SQLF: a relational database language for fuzzy querying. IEEE Trans. Fuzzy Syst. **3**(1), 1–17 (1995)
5. Bosc, P., Pivert, O.: On a strengthening connective for flexible database querying. In: 2011 IEEE International Conference on Fuzzy Systems (FUZZ), pp. 1233–1238 (2011)
6. Dubois, D., Prade, H.: Using fuzzy sets in flexible querying: why and how? In: Andreasen, T., Christiansen, H., Larsen, H.L. (eds.) Flexible Query Answering Systems, pp. 45–60. Kluwer Academic Publishers, Norwell (1997)
7. Guadarrama, S., Muñoz-Hernández, S., Vaucheret, C.: Fuzzy prolog: a new approach using soft constraints propagation. Fuzzy Sets Syst. (FSS) **144**(1), 127–150 (2004). Possibilistic Logic and Related Issues
8. Ishizuka, M., Kanai, N.: Prolog-ELF incorporating fuzzy logic. In: Proceedings of the 9th International Joint Conference on Artificial Intelligence, IJCAI'85, pp. 701–703. Morgan Kaufmann Publishers Inc, San Francisco (1985)
9. Li, D., Liu, D.: A Fuzzy Prolog Database System. Wiley, New York (1990)
10. Medina, J., Ojeda-Aciego, M., Vojtáš, P.: A completeness theorem for multi-adjoint logic programming. In: FUZZ, pp. 1031–1034. IEEE (2001)
11. Medina, J., Ojeda-Aciego, M., Vojtáš, P.: Multi-adjoint logic programming with continuous semantics. In: Eiter, T., Faber, W., Truszczynski, M. (eds.) LPNMR. Lecture Notes in Computer Science, vol. 2173, pp. 351–364. Springer, Berlin (2001)
12. Medina, J., Ojeda-Aciego, M., Vojtáš, P.: A procedural semantics for multi-adjoint logic programming. In: Brazdil, P., Jorge, A. (eds.) EPIA. Lecture Notes in Computer Science, vol. 2258, pp. 290–297. Springer, Berlin (2001)
13. Medina, J., Ojeda-Aciego, M., Vojtáš, P.: A multi-adjoint approach to similarity-based unification. Electron. Notes Theor. Comput. Sci. **66**(5):70–85 (2002). UNCL'2002, Unification in Non-Classical Logics (ICALP 2002 Satellite Workshop)
14. Medina, J., Ojeda-Aciego, M., Vojtáš, P.: Similarity-based unification: a multi-adjoint approach. Fuzzy Sets Syst. **146**(1), 43–62 (2004)
15. Morcillo, P.J., Moreno, G.: Floper, a fuzzy logic programming environment for research. In: Fundación Universidad de Oviedo (ed.) Proceedings of VIII Jornadas sobre Programación y Lenguajes (PROLE'08), pp. 259–263. Gijón, October 2008
16. Moreno, J.M., Ojeda-Aciego, M.: On first-order multi-adjoint logic programming. In: 11th Spanish Congress on Fuzzy Logic and Technology (2002)
17. Muñoz-Hernández, S., Pablos-Ceruelo, V., Strass, H.: RFuzzy: syntax, semantics and implementation details of a simple and expressive fuzzy tool over prolog. Inf. Sci. **181**(10), 1951–1970 (2011). Special Issue on Information Engineering Applications Based on Lattices
18. Pablos-Ceruelo, V., Muñoz-Hernández, S.: Introducing priorities in rfuzzy: syntax and semantics. In: Proceedings of the 11th International Conference on Mathematical Methods in Science and Engineering, CMMSE 2011, vol. 3, pp. 918–929, Benidorm (Alicante), June 2011
19. Ribeiro, R.A., Moreira, A.M.: Fuzzy query interface for a business database. Int. J. Hum.-Comput. Stud. **58**(4), 363–391 (2003)
20. Rodriguez, L.J.T.: A contribution to database flexible querying: fuzzy quantified queries evaluation, P.hD. thesis, November 2005

21. Vaucheret, C., Guadarrama, S., Muñoz-Hernández, S.: Fuzzy prolog: a simple general implementation using CLP(R). In: Baaz, M., Voronkov, A. (eds.) LPAR. Lecture Notes in Artificial Intelligence, vol. 2514, pp. 450–464. Springer, Berlin (2002)
22. Vojtáš, P.: Fuzzy logic programming. Fuzzy Sets Syst. **124**(3), 361–370 (2001)
23. Zadeh, L.A.: Fuzzy sets. Inf. Control **8**(3), 338–353 (1965)