Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 282 (2012) 3–18

www.elsevier.com/locate/entcs

Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language [1]

Jesús M. Almendros-Jiménez [2]

Dep. of Languages and Computation,
University of Almería, Spain

Alejandro Luna and Ginés Moreno [3]

Dep. of Computing Systems,
University of Castilla-La Mancha, Spain

Abstract

FLOPER is the "Fuzzy LOgic Programming Environment for Research" designed in our research group for assisting the development of real-world applications where fuzzy logic might play an important role. This is the case of our recently proposed extension for the popular XPath query language in order to handle flexible queries which provide ranked answers, fuzzy variants of operators *and*, *or* and *avg* for XPath conditions, as well as two structural constraints, called *down* and *deep*, for which a certain degree of relevance can be associated.

Keywords: Fuzzy Logic Programming, XPath Query Language, Software Tools

1 Introduction

The *XPath* language [7] has been proposed as a standard for XML querying and it is based on the description of the path in the XML tree to be retrieved.

[1] This work has been partially supported by the EU, under FEDER, and the Spanish Science and Innovation Ministry (MICINN) under grants TIN2008-06622-C03-03, TIN 2007-65749 and TIN2011-25846, as well as by the Castilla-La Mancha Administration under grant PII1I09-0117-4481.

[2] Email: jalmen@ual.es

[3] Email: {gines.moreno,alejandro.luna}@uclm.es

doi:10.1016/j.entcs.2011.12.002

XPath allows to specify the name of nodes (i.e., tags) and attributes to be present in the XML tree together with boolean conditions about the content of nodes and attributes. XPath querying mechanism is based on a boolean logic: the nodes retrieved from an XPath expression are those matching the path of the XML tree. Therefore, the user should know the *XML schema* in order to specify queries. However, even when the XML schema exists, it may not be available for users. Moreover, XML documents with the same XML schema can be very different in structure. Let us suppose the case of XML documents containing the curriculum vitae of a certain group of persons. Although they can share the same schema, each one can decide to include studies, jobs, training, etc. organized in several ways: by year, by relevance, and with different nesting degree.

Therefore, in the context of semi-structured databases, the need for *flexible query languages* arises, in which the user can formulate queries without taking into account a rigid schema database. In addition, they should be equipped with a mechanism for obtaining a certain *ranked list* of answers. The ranking of answers can provide *satisfaction degree* depending on several factors. In a XPath-based structural query, the main criteria to provide a certain degree of satisfaction are the *hierarchical deepness* and *document order*. Therefore the query language should provide mechanisms for assigning *priority* to answers when they occur in different parts of the document.

In this paper we focus on implementation issues based on fuzzy logic programming regarding our extension of the XPath query language initially presented in [5] for the handling of flexible queries. Our approach proposes two structural constraints called *down* and *deep* for which a certain degree of relevance can be associated. So, whereas *down* provides a ranked set of answers depending on the path they are found from "top to down" in the XML document, *deep* provides a ranked set of answers depending on the path they are found from "left to right" in the XML document. Both structural constraints can be combined. In addition, we provide fuzzy operators *and*, *or* and *avg* for XPath conditions. In this way, users can express the priority they give to answers. Such fuzzy operators can be combined to provide ranked answers. Our approach has been implemented by means of multi-adjoint logic programming and the FLOPER tool [16,17,18].

The need for providing flexibility to XPath has recently motivated the investigation of extensions of the XPath language. The most relevant ones are [8,9] in which authors introduce in XPath flexible matching by means of fuzzy constraints called *close* and *similar* for node content, together with *below* and *near* for path structure. In addition, they have studied *deep-similar* notion for tree matching. In order to provide ranked answers they adopt a *Fuzzy set theory*-based approach in which each answer has an associated numeric value (the membership degree). The numeric value represents the *Retrieval Status Value (RSV)* of the associated item. In the work of [11], they propose a satisfaction degree for XPath expressions based on associating a degree of importance to XPath nodes, and they study how to compute the best k answers. In both cases, authors allow the user to specify in the query the degree to which the answers will be penalized.

Our work is similar to the proposed by [8,9]. The *below* operator of [8,9] is equivalent to our proposed *down*: both extract elements that are direct descendants of the current node, and the penalization is proportional to the distance. The *near* operator of [8,9], which is defined as a generalization of *below*, ranks answers depending on the distance to the required node, in any XPath axis. Our proposed *deep* ranks answers depending of the distance to the current node, but the considered nodes can be direct and non direct descendants. Therefore our proposed *deep* combined with *down* is a particular case of *near*. However, in the future we intend to extend the number of constraints and fuzzy operators of our approach thanks to the expressivity power of our framework based on fuzzy logic programming. The so-called *multi-adjoint logic programming* approach [15], MALP in brief, is an extension of logic programming to support fuzzy logic. Such framework provides theoretical basis for defining flexibility to XPath in many directions. In addition, the framework provides a mechanism for customizing ranked answers by assigning priorities to solutions independently of their occurrences.

With respect to *similar* and *close* operators proposed in [8,9], our framework lacks similarity relations and rather focuses on structural (i.e. path-based) flexibility. With regard to tree matching, the operator *deep-similar* defined in [8,9] can be simulated by means of *deep* and *down* operators. We believe that we could also work in the future in adapting our framework for working with degree of importance to XPath nodes along the lines of [11], and relaxing XPath expressions by rewriting along the lines of [10]. In both cases, our framework could provide ranked answers w.r.t. the degree of importance, and degree of matching. Our proposal makes use of the multi-adjoint logic programming framework for defining new fuzzy operators for XPath: *and*, *or* and *avg*. Such operators are used in XPath conditions on nodes and attribute values. They provide fuzzy combinations for ranking answers.

Finally, let us remark that our work is an extension of previous works about the implementation of XPath by means of logic programming [4], which has been extended to XQuery in [1]. The proposed extension follows the same encoding proposed in [1]. There, a predicate called *xpath* is defined by means of Prolog rules, which basically traverse the Prolog representation of the XML tree by means of a Prolog list. In order to implement the flexible extension of XPath, we proceed similarly to the Prolog implementation of XPath, but proposing a new (fuzzy) predicate called *fuzzyXPath* implemented in MALP, using the «*Fuzzy LOgic Programming Environment for Research*» FLOPER system [16,17,18]. The new query language returns a set of ranked answers each one with an associated RSV. Such RSV is computed by easily using MALP rules. The notion of *RSV* is modeled inside a multi-adjoint lattice, and usual fuzzy connectives of the MALP language act as ideal resources to represent new flexible XPath operators. The main predicate *fuzzyXPath* uses as parameters the user-proposed values for *deep* and *down*, whereas new operators to model flexible XPath conditions admit a natural, elegant and direct representation via standard fuzzy connectives of MALP.

The structure of the paper is as follows. Whereas in Section 2 we present our fuzzy extension of XPath, Section 3 is devoted to MALP and the FLOPER environment, together a description of the integration of fuzzy XPath and MALP. Next, Section 4 discusses the implementation issues and finally, Section 5 concludes planning future work.

2 Flexible XPath

Our flexible XPath is defined by means of the following rules:

xpath :=	[deepdown]path
path :=	literal | text() | node | @att |
	node/path | node//path
node :=	QName | QName[cond]
cond :=	path op path
deepdown :=	DEEP=degree,DOWN=degree
op :=	> | = | < | and | or | avg

Basically, our proposal extends XPath as follows:

- A given XPath expression can be adorned with «[DEEP $= r_1$, DOWN $= r_2$]» which means that the *deepness* of elements is penalized by r_1 and that the *order* of elements is penalized by r_2, and such penalization is proportional to the distance (i.e., the length of the branch and the weight of the tree, respectively). In particular, «[DEEP $=$ 1, DOWN $= r_2$]» can be used for penalizing only w.r.t. document order. *DEEP* works for //, that is, the deepness in the XML tree is only computed when descendant nodes are explored, while *DOWN* works for both / and //. Let us remark that currently *DEEP* and *DOWN* can be only used at the beginning of the XPath expression but they are computed for each occurrence of / and //.
- Moreover, the classical *and* and *or* connectives admit here a fuzzy behavior

Fig. 1. Input XML document in our examples

```
<bib>
   <book year="2001" price="45.95">
      <title>Don Quijote de la Mancha</title>
      <author>Miguel de Cervantes Saavedra</author>
      <publications> <book year="1997" price="35.99">
                          <title>La Galatea</title>
                          <author>Miguel de Cervantes Saavedra</author>
                          <publications>
                                <book year="1994" price="25.99">
                                   <title>Los trabajos de Persiles y
                                       Segismunda</title>
                                   <author>Miguel de Cervantes Saavedra</
                                       author></book>
                         </publications></book>
      </publications></book>
   <book year="1999" price="25.65">
      <title>La Celestina</title>
      <author>Fernando de Rojas</author></book>
   <book  year="2005" price="29.95">
      <title>Hamlet</title>
      <author>William Shakespeare</author>
      <publications>
         <book year="2000" price="22.5">
            <title>Romeo y Julieta</title>
            <author>William Shakespeare</author></book>
      </publications></book>
   <book  year="2007" price="22.95">
      <title>Las ferias de Madrid</title>
      <author>Felix Lope de Vega y Carpio</author>
      <publications>
         <book year="1996" price="27.5">
            <title>El remedio en la desdicha</title>
            <author>Felix Lope de Vega y Carpio</author> </book>
         <book year="1998" price="12.5">
            <title>La Dragontea</title>
            <author>Felix Lope de Vega y Carpio</author></book>
      </publications></book>
</bib>
```

based on fuzzy logic, i.e., assuming two given *RSV*'s r_1 and r_2, operator *and* is defined as $r_3 = r_1 * r_2$ and operator *or* returns $r_3 = r_1 + r_2 - (r_1 * r_2)$. In addition, the *avg* operator is defined as $r_3 = (r_1 + r_2)/2$.

In general, an extended XPath expression defines, w.r.t. an XML document, a sequence of subtrees of the XML document where each subtree has an associated RSV. XPath conditions, which are defined as fuzzy operators applied to XPath expressions, compute a new RSV from the RSVs of the involved XPath expressions, which at the same time, provides a RSV to the node. In order to illustrate these explanations, let us see some examples of our proposed fuzzy version of XPath according to the XML document shown of Figure 1, whose *skeleton* is depicted in Figure 2.

Example 2.1 Consider the XPath query: « [DEEP=0.9,DOWN=0.8]//title

Fig. 2. XML skeleton represented as a tree

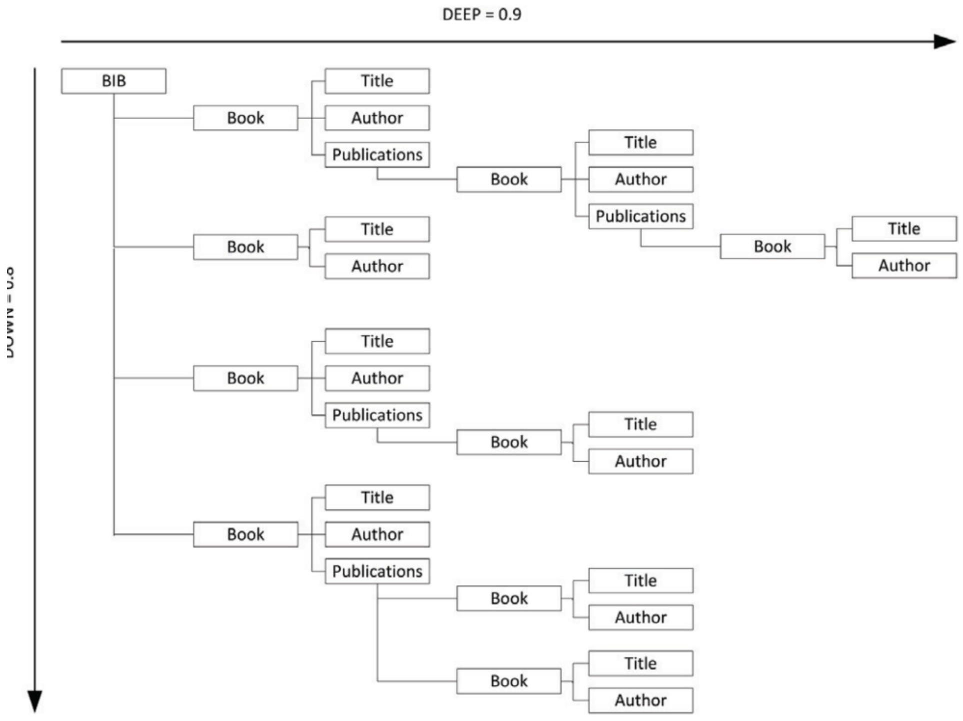

Fig. 3. Output of a query using *DEEP/DOWN*

Document

```
<result>
  <title rsv="0.81">Don Quijote de la Mancha</title>
  <title rsv="0.52488">La Galatea</title>
  <title rsv="0.340122">Los trabajos de Persiles y Sigismunda</title>
  <title rsv="0.648">La Celestina</title>
  <title rsv="0.5184">Hamlet</title>
  <title rsv="0.335923">Romeo y Julieta</title>
  <title rsv="0.41472">Las ferias de Madrid</title>
  <title rsv="0.268739">El remedio en la desdicha</title>
  <title rsv="0.214991">La Dragontea</title>
</result>
```

RSV computation

$0.81 = 0.9^2$

$0.52488 = 0.9^4 * 0.8$

$0.340122 = 0.9^6 * 0.8^2$

$0.648 = 0.9^2 * 0.8$

$0.5184 = 0.9^2 * 0.8^2$

$0.335923 = 0.9^4 * 0.8^3$

$0.41472 = 0.9^2 * 0.8^3$

$0.268739 = 0.9^4 * 0.8^4$

$0.214991 = 0.9^4 * 0.8^5$

Fig. 4. Output of a query using the *average* operator *AVG*

Document

```
<result>
  <book rsv="0.5" ...> <title>Don Quijote ...</title> ...</book>
  <book rsv="1.0"...><title>La Celestina</title> ...</book>
  <book rsv="1.0" ...><title>Hamlet</title> ...</book>
  <book rsv="0.5" ...><title>Las ferias de Madrid</title> ...</book>
</result>
```

RSV computation

$0.5 = (0 + 1)/2$

$1 = (1 + 1)/2$

$1 = (1 + 1)/2$

$0.5 = (1 + 0)/2$

Fig. 5. Output of a query using all operators

Document

```
<result>
  <title rsv="0.3645">La Galatea</title>
  <title rsv="0.59049">Los trabajos de Persiles y Sigismunda</title>
  <title rsv="0.72">La Celestina</title>
  <title rsv="0.288">Hamlet</title>
  <title rsv="0.2304">Las ferias de Madrid</title>
  <title rsv="0.373248">El remedio en la desdicha</title>
  <title rsv="0.149299">La Dragontea</title>
</result>
```

RSV computation

$0.3645 = 0.9^3 * 1/2$

$0.59049 = 0.9^5 * 1$

$0.72 = 0.9 * 0.8 * 1$

$0.288 = 0.9 * 0.8^2 * 1/2$

$0.2304 = 0.9 * 0.8^3 * 1/2$

$0.373248 = 0.9^3 * 0.8^3 * 1$

$0.149299 = 0.9^3 * 0.8^4 * 1/2$

», that requests *title*'s penalizing the occurrences from the document root by a proportion of 0.9 and 0.8 by nesting and ordering, respectively, and for which we obtain the document listed in Figure 3. In such document we have included as attribute of each subtree, its corresponding RSV. The highest RSVs correspond to the main *books* of the document, and the lowest RSVs represent the *books* occurring in nested positions (those annotated as related *publication*'s).

Example 2.2 Figure 4 shows the answer associated to the XPath expression: « /bib/book[@price<30 avg @year<2006] ». Here we show that books satisfying a *price* under 30 and a *year* before 2006 have the highest RSV.

Example 2.3 Finally, in Figure 5 we combine all operators (thus obtaining more scattered RSV values) in query: «[DEEP=0.9,DOWN=0.8] //book [(@price>25 and @price<30) avg (@year<2000 or @year>2006)]/title».

3 Multi-Adjoint Logic Programming and FLOPER

Logic Programming (LP) [14] has been widely used as a formal method for problem solving and knowledge representation in the past. Nevertheless, traditional LP languages do not incorporate techniques or constructs to deal explicitly with uncertainty and approximated reasoning. To overcome this situation, during the last decades several fuzzy logic programming systems have been developed where the classical inference mechanism of SLD–Resolution has been replaced with a fuzzy variant able to handle partial truth and to reason with uncertainty. Most of these systems implement the fuzzy resolution principle introduced by Lee in [13], such as languages Prolog-Elf [12], Fril [6], the QLP scheme of [19], as well as many-valued logic programming languages [21,20] and MALP [15]. In this paper we are mainly concerned with this last framework, which uses a syntax close to Prolog but enjoys higher levels of flexibility and for which we are developing the FLOPER tool (see [16,17,18] and visit `http://dectau.uclm.es/floper`). In what follows, we present a short summary of the main features of MALP (we refer the reader to [15] for a complete formulation).

We work with a first order language, $\mathcal{L}$, containing variables, function symbols, predicate symbols, constants, quantifiers ($\forall$ and $\exists$), and several arbitrary connectives such as implications ($\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m$), conjunctions ($\&_1, \&_2, \ldots, \&_k$), disjunctions ($\vee_1, \vee_2, \ldots, \vee_l$), and general hybrid operators ("aggregators" $@_1, @_2, \ldots, @_n$), used for combining/propagating truth values through the rules, and thus increasing the language expressiveness. Additionally, our language $\mathcal{L}$ contains the values of a multi-adjoint lattice in the form $\langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n \rangle$, equipped with a collection of adjoint pairs $\langle \leftarrow_i, \&_i \rangle$ where each $\&_i$ is a conjunctor intended to the evaluation of *modus ponens*.

A *rule* is a formula "$A \leftarrow_i \mathcal{B}$ with α", where A is an atomic formula (usually called the *head*), $\mathcal{B}$ (which is called the *body*) is a formula built from atomic formulas $B_1, \ldots, B_n$ ($n \geq 0$), truth values of L and conjunctions, disjunctions and general aggregations, and finally $\alpha \in L$ is the "weight" or *truth degree* of the rule. The set of truth values L may be the carrier of any complete bounded lattice, as for instance occurs with the set of real numbers in the interval $[0, 1]$ with their corresponding ordering $\leq$. Consider, for instance, the following program $\mathcal{P}$ composed of three rules with associated multi-adjoint lattice $\langle [0, 1], \leq, \leftarrow_{\mathtt{P}}, \&_{\mathtt{P}} \rangle$, where label `P` mean for *Product logic* with the following connective definitions (for implication, conjunction and disjunction symbols, respectively): "$\leftarrow_{\mathtt{P}} (x, y) = \min(1, x/y)$", "$\&_{\mathtt{P}}(x, y) = x * y$" and "$|_{\mathtt{P}}(x, y) = x + y - x * y$".

$$\begin{array}{llllll}
\mathcal{R}_1 : & p(X) & \leftarrow_{\mathtt{P}} & q(X, Y) \mid_{\mathtt{P}} r(Y) & \textit{with} & 0.8 \\
\mathcal{R}_2 : & q(a, Y) & \leftarrow & & \textit{with} & 0.9 \\
\mathcal{R}_3 : & r(b) & \leftarrow & & \textit{with} & 0.7
\end{array}$$

In order to describe the procedural semantics of the multi–adjoint logic language, in the following we denote by $\mathcal{C}[A]$ a formula where A is a subexpression (usually an atom) which occurs in the –possibly empty– one hole context $\mathcal{C}[]$ whereas $\mathcal{C}[A/A']$ means the replacement of A by A' in context $\mathcal{C}[]$, and $mgu(E)$ is the *most general unifier* of an equation set E. The pair $\langle \mathcal{Q}; \sigma \rangle$ composed of a goal and a substitution is called a *state*. So, given a program $\mathcal{P}$, an *admissible computation* is formalized as a state transition system, whose transition relation $\overset{\text{AS}}{\leadsto}$ is the smallest relation satisfying the following *admissible rules*:

1) $\langle \mathcal{Q}[A]; \sigma \rangle \overset{\text{AS}}{\leadsto} \langle (\mathcal{Q}[A/v \&_i \mathcal{B}])\theta; \sigma\theta \rangle$ if A is the selected atom in goal $\mathcal{Q}$,

$\langle A' \leftarrow_i \mathcal{B}\ with\ v\rangle$ in $\mathcal{P}$, where $\mathcal{B}$ is not empty, and $\theta = mgu(\{A' = A\})$.
2) $\langle \mathcal{Q}[A]; \sigma\rangle \overset{\text{AS}}{\leadsto} \langle(\mathcal{Q}[A/v])\theta; \sigma\theta\rangle$ if $\langle A' \leftarrow\ with\ v\rangle$ in $\mathcal{P}$, $\theta = mgu(\{A' = A\})$.
The following derivation illustrates our definition (note that the exact program rule used -after being renamed- in the corresponding step is annotated as a super–index symbol, whereas exploited atoms appear underlined):

$$\begin{array}{ll}
\langle \underline{p(X)}; \{\}\rangle & \overset{\text{AS}}{\leadsto} \mathcal{R}_1 \\
\langle 0.8 \ \&_{\mathtt{P}}\ (\underline{q(X_1, Y_1)}\ |_{\mathtt{P}}\ r(Y_1)); \{X/X_1\}\rangle & \overset{\text{AS}}{\leadsto} \mathcal{R}_2 \\
\langle 0.8 \ \&_{\mathtt{P}}\ (0.9\ |_{\mathtt{P}}\ \underline{r(Y_2)}); \{X/a, X_1/a, Y_1/Y_2\}\rangle & \overset{\text{AS}}{\leadsto} \mathcal{R}_3 \\
\langle 0.8 \ \&_{\mathtt{P}}\ (0.9\ |_{\mathtt{P}}\ 0.7); \{X/a, X_1/a, Y_1/b, Y_2/b\}\rangle &
\end{array}$$

The final formula can be directly interpreted in the lattice L to obtain the final *fuzzy computed answer*. So, since $0.8\ \&_{\mathtt{P}}\ (0.9\ |_{\mathtt{P}}\ 0.7) = 0.8 * (0.9 + 0.7 - (0.9 * 0.7)) = 0.776$, we say that goal $p(X)$ is true at a 77.6 % when X is a.

Now, we would like to summarize the main elements of the FLOPER tool which manages MALP programs. The parser of our FLOPER tool [16,17,18] has been implemented by using the classical DCG's (*Definite Clause Grammars*) resource of the Prolog language, since it is a convenient notation for expressing grammar rules. Once the application is loaded inside a Prolog interpreter, it shows a menu which includes options for loading/compiling, parsing, listing and saving fuzzy programs, as well as for executing/debugging fuzzy goals. All these actions are based on the compilation of the fuzzy code into standard Prolog code. The key point of the compilation is to extend each atom with an extra argument, called *truth variable* of the form "`_TV`$_{\mathtt{i}}$", which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. For instance, the first clause in our example is translated into:

```
p(X,_TV0):-q(X,Y,_TV1),r(Y,_TV2),
            or_prod(_TV1,_TV2,_TV3),and_prod(0.8,_TV3,_TV0).
```

Moreover, the remaining rules in our example, become the pure Prolog facts "`q(a,Y,0.9)`" and "`r(b,0.7)`", whereas the corresponding lattice is expressed by the clauses of Figure 6, where the meaning of the mandatory predicates `member`, `top`, `bot` and `leq` is obvious.

Finally, a fuzzy goal like "`p(X)`", is translated into the pure Prolog goal: "`p(X, Truth_degree)`" (note that the last truth degree variable is not anonymous now) for which, after choosing option "`run`", the Prolog interpreter returns the desired fuzzy computed answer [`Truth_degree` = `0.776`, `X` = `a`]. Note that all internal computations (including compiling and executing) are pure Prolog derivations, whereas inputs (fuzzy programs and goals) and out-

Fig. 6. Example of Lattice

```
member(X):- number(X), 0=<X, X=<1.
bot(0).  top(1).  leq(X,Y):- X=<Y.
or_prod(X,Y,Z):- pri_prod(X,Y,U1), pri_add(X,Y,U2), pri_sub(U2,U1,Z).
and_prod(X,Y,Z):- pri_prod(X,Y,Z).
pri_prod(X,Y,Z):- Z is X * Y.
pri_add(X,Y,Z):- Z is X+Y.
pri_sub(X,Y,Z):- Z is X-Y.
```

puts (fuzzy computed answers) have always a fuzzy taste, thus producing the illusion on the final user of being working with a purely fuzzy logic programming tool. By using option "`lat`" ("`show`") of FLOPER, we can associate (visualize) a new lattice to a given program. As seen before, such lattices must be expressed by means of a set of Prolog clauses (defining predicates `member`, `top`, `bot`, `leq` and the ones associated to fuzzy connectives) in order to be loaded into FLOPER.

3.1 MALP and XPath

MALP can be used as basis for our proposed flexible extension of XPath as follows. The idea is to implement XPath by means of MALP rules. With this aim, firstly, we have to consider a complete lattice L_{tv} to be used for representing RSVs associated to elements of an XML tree. L_{tv} contains trees, which we will call *tv trees*, of the form: "$tv(rsv, [root, tvch, tvsib])$", where *rsv* is a value taken from $[0, 1]$, *root* is the root of the tree to which *rsv* is associated, and *tvch*, *tvsib* are the *tv trees* of the children and sibling nodes. Such *tv trees* are the answers of a goal associated to a XPath expression. In this way, MALP is able to compute the RSV associated to each node of the input XML tree. Using such lattice, the MALP rules have the form "$A \leftarrow \mathcal{B}$ *with tvtree*". In addition, the XPath fuzzy operators *and* and *or* and *avg* can be mapped to MALP connectives in lattice L_{tv}. Finally, we have to consider an auxiliary aggregator in L_{tv} called *@fuse*, which builds the *tv tree* of a certain node from the tv trees of the sibling and children nodes. In Figure 7[4], we can see the lattice defined by means of Prolog syntax (to be loaded into FLOPER). Predicates `and_pro`, `or_prod` and `agr_aver` represent the XPath operators *and*, *or* and *avg*, respectively. Let us remark that they are defined for *tv trees* and use in their definitions the operators *and*, *or* and *avg* of the lattice $[0, 1]$.

[4] Elements of L_{tv} are equivalence classes with regard to the first argument of *tv*. The supremum of the lattice is the equivalence class *tv(1,_)*, and the bottom is the equivalence class *tv(0,_)*.

```
member(tv(N,L)):-number(N),0=<N,N=<1,(L=[];L=[_|_]).

bot(tv(0,_)).          top(tv(1,_)).         leq(tv(X1,_),tv(X2,_)):- X1 =< X2.

and_prod(tv(X1,X2),tv(Y1,Y2),tv(Z1,Z2)):-
                        pri_prod(X1,Y1,Z1),pri_app(X2,Y2,Z2).

or_prod(tv(X1,X2),tv(Y1,Y2),tv(Z1,Z2)):-  pri_prod(X1,Y1,U1),
                        pri_add(X1,Y1,U2),pri_sub(U2,U1,Z1),pri_app(X2,Y2,Z2)
                            .

agr_aver(tv(X1,X2),tv(Y1,Y2),tv(Z1,Z2)):-  pri_add(X1,Y1,Aux),
                        pri_div(Aux,2,Z1),pri_app(X2,Y2,Z2).

pri_add(X,Y,Z)  :- Z is X+Y.                pri_sub(X,Y,Z) :-Z is X-Y.

pri_prod(X,Y,Z) :- Z is X * Y.              pri_div(X,Y,Z) :- Z is X/Y.

pri_app([],X,X).                pri_app([A|B],C,[A|D]):-pri_app(B,C,D).
```

Fig. 7. Multi-adjoint lattice for "fuzzyXPath" (file "tv.pl")

4 Implementation

Although the core of our implementation is written with (fuzzy) MALP rules, we have reused/adapted several modules of our previous Prolog-based implementation of (crisp) XPath described in [1,2,3,4], which make use of the SWI-Prolog library for loading XML files in order to store each XML document by means of a Prolog term[5] representing a tree. The clever idea is that each tag is represented as a data-term of the form:

$$\texttt{element}(\texttt{Tag}, \texttt{Attributes}, \texttt{Subelements})$$

where `Tag` is the name of the XML tag, `Attributes` is a list containing the attributes, and `Subelements` is a list containing the subelements (i.e. subtrees) of the tag. For instance, let us consider the XML document of Figure 1, represented in SWI-Prolog like in Figure 8. Moreover, for loading XML documents in our implementation we can use the predicate `load_xml(+File,-Term)`. Similarly, we have a predicate `write_xml(+File,+Term)` for writing a data-term representing an XML document into a file. And, of course, the parser of our application has been extended to recognize the new keywords *deep, down, avg*, etc... with their proper arguments.

Now, we would like to show how the new «`fuzzyXPath`» predicate admits an elegant definition by means of fuzzy MALP rules which, after being compiled into clauses using FLOPER, can be safely executed in any standard

[5] The notion of *term* (i.e., data structure) is just the same in MALP and Prolog.

Fig. 8. A data-term representing an XML document

```
[element(bib, [],
   [element(book, [year='2001',price='45.95'],
      [element(title,[],['Don Quijote de la Mancha']),
        element(author, [], ['Miguel de Cervantes Saavedra']),
        element(publications, [],
          [element(book, [year='1997',price='35.99'],
             [element(title, [], ['La Galatea']),
               element(author, [], [ 'Miguel de Cervantes Saavedra']),
               element(publications, [],
           ...])
   ...]),
])])
```

Fig. 9. Example of a *tv* structure

```
tv(1.0,[[],
   tv(0.9,[[],
      tv(0.9,[element(title, [], [Don Quijote de la Mancha]), [],
      tv(0.8,[[],
         tv(0.9,[[],
            tv(0.9,[element(title, [], [La Galatea]), [],
            tv(0.8,[[],
               tv(0.9,[[],
                  tv(0.9,[element(title, [], [Los trabajos de Persiles y Sigismunda]),
                      [], []]),
               []]),
            []])]),
         []]),
      []])]),
   tv(0.8,[[],
      tv(0.9,[element(title, [], [La Celestina]), [], []]),
   tv(0.8,[[],
      tv(0.9,[element(title, [], [Hamlet]), [],
      tv(0.8,[[],
         tv(0.9,[[],
            tv(0.9,[element(title, [], [Romeo y Julieta]), [], []]),
         []]),
      []])]),
   tv(0.8,[[],
      tv(0.9,[element(title, [], [Las ferias de Madrid]), [],
      tv(0.8,[[],
         tv(0.9,[[],
            tv(0.9,[element(title, [], [El remedio en la desdicha]), [], []]),
         tv(0.8,[[],
            tv(0.9,[element(title, [], [La Dragontea]), [], []]),
         []])]),
      []])]),
   []])])])]),
[]])
```

Prolog platform. Each rule defining predicate:

$$\texttt{fuzzyXPath}(\texttt{ListXPath}, \texttt{Tree}, \texttt{Deep}, \texttt{Down})$$

Fig. 10. MALP rule for traversing `element` structures representing XML documents

```
fuzzyXPath([Label|LabelRest], [element(Label,_,Children)|Siblings],Deep, Down) <prod
       @fuse(
              tv(1,[element(Label, Attr, Children), [], []]),
              &prod(Deep, fuzzyXPath(LabelRest, Children, Deep, Down))
              &prod(Down, fuzzyXPath([Label|LabelRest], Siblings, Deep, Down))
       ) with tv(1,[])
```

Fig. 11. Prolog clause obtained by FLOPER after compiling a MALP rule

```
fuzzyXPath([Label|LabelRest], [element(Label,_,Children)|Siblings],Deep, Down, TV_Iam):-
       fuzzyXPath(LabelRest, Children, Deep, Down, TV_Son),
       and_prod(Deep, TV_Son, TV_Son0),
       fuzzyXPath([Label|LabelRest], Siblings, Deep, Down, TV_Bro),
       and_prod(Down, TV_Bro, TV_Bro0),
       agr_fuse(tv(1,[element(Label, Attr, Children), [], []]), TV_Son0, TV_Bro0, TV_body),
       and_prod(tv(1,[]), TV_body, TV_Iam).
```

receives four arguments: (1) `ListXPath` is the Prolog representation of an XPath expression, (2) `Tree` is the term representing an input XML document and (3) `Deep/Down` which have the obvious meaning -their default values are `tv(1,[])-`[6]. A call to this predicate returns after being executed a truth-value (i.e., a *tv tree*) like the one depicted in Figure 9.

For instance, the query «[DEEP=0.9,DOWN=0.8]/Path » on a given XML term, would generate a call of the form

$$\texttt{fuzzyXPath}(\mathit{Path}, \mathit{XML}, \texttt{tv}(0.9, []), \texttt{tv}(0.8, []))$$

whose further execution will return the resulting *tv tree*.

Basically, the fuzzyXPath predicate traverses the Prolog tree representing an XML document and extracts in the returned tv tree the subtrees occurring in the given path, also annotating into the *nested* tv trees the corresponding *deep/down* values according to the movements performed (in the horizontal and vertical axis, respectively) when navigating on the XML tree.

The definition of such predicate includes several rules for distinguishing cases in the form of the input document and the XPath expression. As an example, we can see the rule of Figure 10, whose translation to Prolog is shown in Figure 11.

Let us now explain in detail the fuzzy code of Figure 10. After performing

[6] These parameters could be avoided if we declare `deep` and `down` as *constants* in the lattice (in a similar way as done, for instance, with `bot` and `top` in Figure 7) but in that case we would need to redefine them at the beginning of each query evaluation. So, we prefer our present option which is easy to understand and safe, in the sense that in the Prolog code generated by FLOPER when compiling MALP programs, the notions of truth-degree and fuzzy connectives are assimilated to data-terms and predicates, respectively.

Fig. 12. Schema of MALP rules for evaluating conditions (with `avg`)

```
fuzzyXPath([Label,tree(A,B,C)],[element(Label,Attr,Children)|Siblings], Deep, Down) <prod
       @fuse(
               execute_fcond(Label, tree(A,B,C), element(Label,Attr,Children)),
               &prod(Down, fuzzyXPath([Label, tree(A,B,C)], Siblings, Deep, Down))
           ) with tv(1,[])


execute_fcond(Label, tree(avg, T1, T2), element(Label,Attr,Children)) <prod
       @avg(
                   execute_fcond(Label, T1, element(Label,Attr,Children)),
                   execute_fcond(Label, T2, element(Label,Attr,Children)),
       ) with tv(1,[])
```

a recursive call to compute the solutions associated to the children of a given node (i.e., `fuzzyXPath(LabelRest, Children, Deep,Down)`), we use connective `&prod` to muffle the resulting tv tree according to `Deep`, which is represented by `&prod(Deep,fuzzyXPath(LabelRest,Children,Deep,Down))`. A similar operation is next performed on the siblings of the node, whose result is penalized now according to `Down`, that is, `&prod(Down, fuzzyXPath([Label| LabelRest], Siblings, Deep, Down))`. Finally, both tv trees are combined (*fused*) with the content of the current node.

On the other hand, when considering queries with conditions, the `fuzzyXPath` predicate calls to `execute_fcond` predicate, as shown in the MALP rule listed in Figure 12. Here it is remarkable the direct use of connective `avg` when defining the recursive `execute_fcond` predicate.

Finally, we have defined a predicate `tv_to_elem` to show the result in a pretty way (see Figures 3 and 4), which transforms the returned *tv* tree to an XML tree.

5 Conclusions and Future Work

In this paper we have described the implementation of a fuzzy extension of XPath. We have recently proposed such XPath extension [5]. Here we have focused on the implementation of the proposal, by using the FLOPER tool and MALP rules. As a result of the implementation, a prototype is publicly available from `http://dectau.uclm.es/fuzzyXPath/`. The material presented here represents the first real-world application developed with the fuzzy logic language MALP (using too our FLOPER tool), by showing its capabilities for easily modeling scenarios where concepts somehow based on fuzzy logic play a crucial role. In particular, we have shown the ability of the MALP language for easily coding the new constructs (both structural -*deep* and *down*-

and constraints -*avg* and fuzzy versions of classical *or/and* operators-) of the enriched dialect of XPath, in order to flexibly query XML documents.

We are currently working on some extensions suggested by the power of MALP regarding two main implementation lines: a) defining commands for "reverse axes" such as *up*, i.e. the counterpart of *down* (sometimes the more nested information is the more basic and should receive a better relevance), thus connecting with the classical *near* operator and b) using a whole family of fuzzy connectives (belonging, for instance, to the well known *Gödel* and *Łukasiewicz* fuzzy logics for describing scenarios with a somehow optimistic/pesimistic taste) in order to express more flexible conditions on *fuzzyXPath* queries. We think that this research line promises fruitful developments in the near future by reinforcing the power of fuzzy XPath commands, extensions to cope with XQuery and the semantic web, etc.

References

[1] J. M. Almendros-Jiménez. An Encoding of XQuery in Prolog. In *Proceedings of the Sixth International XML Database Symposium XSym'09*, pages 145–155, Heildelberg,Germany, 2009. Springer, LNCS 5679.

[2] J. M. Almendros-Jiménez, A. Becerra-Terón, and F. J. Enciso-Baños. Integrating XQuery and Logic Programming. In *Procs of the International Conference on Applications of Declarative Programming and Knowledge Management, INAP-WLP'07*, pages 117–135, Heidelberg, Germany, 2009. Springer LNAI, 5437.

[3] J. M. Almendros-Jiménez, A. Becerra-Terón, and Francisco J. Enciso-Baños. Magic sets for the XPath language. *Journal of Universal Computer Science*, 12(11):1651–1678, 2006.

[4] J. M. Almendros-Jiménez, A. Becerra-Terón, and Francisco J. Enciso-Baños. Querying XML documents in logic programming. *TPLP*, 8(3):323–361, 2008.

[5] J.M. Almendros-Jiménez, A. Luna, and G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In *Proc. of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11. Barcelona, Spain, July 19–21*, pages 186–193, Heidelberg,Germany, 2011. Springer Verlag, LNCS 6826.

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

[7] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0. *W3C*, 2007.

[8] A. Campi, E. Damiani, S. Guinea, S. Marrara, G. Pasi, and P. Spoletini. A fuzzy extension of the XPath query language. *Journal of Intelligent Information Systems*, 33(3):285–305, 2009.

[9] E. Damiani, S. Marrara, and G. Pasi. FuzzyXPath: Using fuzzy logic an IR features to approximately query XML documents. *Foundations of Fuzzy Logic and Soft Computing*, pages 199–208, 2007.

[10] B. Fazzinga, S. Flesca, and F. Furfaro. On the expressiveness of generalization rules for XPath query relaxation. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*, pages 157–168. ACM, 2010.

[11] B. Fazzinga, S. Flesca, and A. Pugliese. Top-k Answers to Fuzzy XPath Queries. In *Database and Expert Systems Applications*, pages 822–829. Springer, 2009.

[12] M. Ishizuka and N. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In Aravind K. Joshi, editor, *Proceedings of the 9th Int. Joint Conference on Artificial Intelligence, IJCAI'85*, pages 701–703. Morgan Kaufmann, 1985.

[13] R.C.T. Lee. Fuzzy Logic and the Resolution Principle. *Journal of the ACM*, 19(1):119–129, 1972.

[14] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[15] J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146:43–62, 2004.

[16] P.J. Morcillo and G. Moreno. Programming with Fuzzy Logic Rules by using the FLOPER Tool. In Nick Bassiliades et al., editor, *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML'08*, pages 119–126. Springer Verlag, LNCS 3521, 2008.

[17] P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In M. Dean et al., editor, *Proc. of 4nd Intl Symposium on Rule Interchange and Applications, RuleML'10*, pages 20–34. Springer Verlag, LNCS 6403, 2010.

[18] P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. Fuzzy Computed Answers Collecting Proof Information. In J. Cabestany et al., editor, *Advances in Computational Intelligence - Proc of the 11th International Work-Conference on Artificial Neural Networks, IWANN 2011*, pages 445–452. Springer Verlag, LNCS 6692, 2011.

[19] M. Rodríguez-Artalejo and C. Romero-Díaz. Quantitative logic programming revisited. In J. Garrigue and M. Hermenegildo, editors, *Functional and Logic Programming (FLOPS'08)*, pages 272–288. Springer LNCS 4989, 2008.

[20] U. Straccia. Managing uncertainty and vagueness in description logics, logic programs and description logic programs. In *Reasoning Web, 4th International Summer School, Tutorial Lectures*, number 5224 in Lecture Notes in Computer Science, pages 54–103. Springer Verlag, 2008.

[21] P. Vojtáš and L. Paulík. Query answering in normal logic programs under uncertainty. In L. Godó, editor, *Proc. of 8th. European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-05), Barcelona, Spain*, pages 687–700. Lecture Notes in Computer Science 3571, Springer Verlag, 2005.