Fuzzy Computed Answers Collecting Proof Information [⋆]

Pedro J. Morcillo, Ginés Moreno, Jaime Penabad and Carlos Vázquez

University of Castilla-La Mancha
Faculty of Computer Science Engineering
02071, Albacete (Spain)
{pmorcillo,cvazquez}@dsi.uclm.es
{Gines.Moreno,Jaime.Penabad}@uclm.es

Abstract. MALP (i.e., the so-called *Multi-Adjoint Logic Programming approach*) can be seen as a promising fuzzy extension of the popular, pure logic language Prolog, including too a wide repertoire of constructs based on fuzzy logic in order to support uncertainty and approximated reasoning in a natural way. Moreover, the *Fuzzy LOgic Programming Environment for Research*, FLOPER in brief, that we have implemented in our research group, is intended to assists the development of real-world applications written with MALP syntax. Among other capabilities, the system is able to safely translate fuzzy code into Prolog clauses which can be directly executed inside any standard Prolog interpreter in a completely transparent way for the final user. In this fuzzy setting, it is mandatory the use of lattices modeling truth degrees beyond $\{true; false\}$. As described in this paper, FLOPER is able to successfully deal (in a very easy way) with sophisticated lattices modeling truth degrees in the real interval $[0, 1]$, also documenting -via declarative traces- the proof procedures followed when solving queries, without extra computational cost.

Key words: Fuzzy Logic Programming, Logic Proofs, Declarative Debugging

1 Introduction

Logic Programming (LP) [8] has been widely used for problem solving and knowledge representation in the past, with recognized applications in AI and related areas. Nevertheless, traditional LP languages do not incorporate techniques or constructs to treat explicitly with uncertainty and approximated reasoning. To overcome this situation, during the last years, several fuzzy logic programming systems have been developed where the classical inference mechanism of SLD–Resolution has been replaced by a fuzzy variant able to handle partial truth and to reason with uncertainty [3, 1, 10], with promising applications in the fields of Computational Intelligence, Soft-Computing, Semantic Web, etc.

[⋆] This work was supported by the EU (FEDER), and the Spanish Science and Innovation Ministry (MICINN) under grants TIN 2007-65749 and TIN2011-25846, and by the Castilla-La Mancha Administration under grant PII1I09-0117-4481.

Informally speaking, in the MALP framework of [10, 9], a program can be seen as a set of rules each one annotated by a truth degree, and a goal is a query to the system, i.e., a set of atoms linked with connectives called *aggregators*. A *state* is a pair $\langle \mathcal{Q}, \sigma \rangle$ where $\mathcal{Q}$ is a goal and σ a substitution (initially, the identity substitution). States are evaluated in two separate computational phases. Firstly, *admissible steps* (a generalization of the classical *modus ponens* inference rule) are systematically applied by a backward reasoning procedure in a similar way to classical resolution steps in pure logic programming, thus returning a computed substitution together with an expression where all atoms have been exploited. This last expression is then interpreted under a given lattice, hence returning a pair $\langle$*truth degree*; *substitution*$\rangle$ which is the fuzzy counterpart of the classical notion of computed answer traditionally used in LP.

In the present paper, we draw the last developments performed on the FLOPER system (see [11, 12] and visit `http://www.dsi.uclm.es/investigacion/dect/FLOPERpage.htm`), which currently provides facilities for compiling, executing and manipulating such kind of fuzzy programs, by means of two main representation (high/low-level, Prolog-based) ways which are somehow antagonistics regarding simplicity and accuracy features. The main purpose of the present paper is to highlight a collateral effect of the last feature implemented into the tool, regarding the possibility of introducing different notions of multi-adjoint lattices which can be easily defined with a Prolog taste. Only a few number of clauses suffices for modeling rich notions of truth degrees incorporating augmented information about the program rules used in a derivation sequence as well as the set of fuzzy connectives evaluated at execution time when reaching the whole set of solutions for a given program and goal. The most surprising fact reported here is that this kind of "*extra proof information*" can be freely collected on fuzzy computed answers without requiring any additional computational resource.

The outline of this work is as follows. In Section 2 we detail the main features of multi-adjoint logic programming, both syntax and procedural semantics. Section 3 explains the current menu of programming resources implemented into the FLOPER tool, which nowadays is being equipped with new options for performing advanced program manipulation tasks (transformation, specialization, optimization) with a clear fuzzy taste. The benefits of our present approach regarding how to obtain fuzzy computed answers containing debugging information on execution proofs, are highlighted in Section 4. Finally, in Section 5 we present our conclusions and propose some lines of future work.

2 Multi-Adjoint Logic Programs

In what follows, we present a short summary of the main features of our language (we refer the reader to [10] 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 to increase language expressiveness. In our fuzzy setting, we use implication connectives $(\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m)$ and also other connectives which are grouped under the name of "aggregators" or "aggregation operators". They are used to combine/propagate truth values through the rules. The general definition of aggregation operators subsumes conjunctive operators (denoted by $\&_1, \&_2, \ldots, \&_k$), disjunctive operators $(\vee_1, \vee_2, \ldots, \vee_l)$, and average and hybrid operators (usually denoted by $@_1, @_2, \ldots, @_n$). Although the connectives $\&_i$, $\vee_i$ and $@_i$ are binary operators, we usually generalize them as functions with an arbitrary number of arguments. By definition, the truth function for an n-ary aggregation operator $[\![@]\!] : L^n \rightarrow L$ is required to be monotone and fulfills $[\![@]\!](\top, \ldots, \top) = \top$, $[\![@]\!](\bot, \ldots, \bot) = \bot$. Additionally, our language $\mathcal{L}$ contains the values of a multi-adjoint lattice, $\langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n \rangle$, 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*. In general, the set of truth values L may be the carrier of any complete bounded lattice but, for simplicity, in this paper we shall select L as the set of real numbers in the interval $[0, 1]$.

A *rule* is a formula $A \leftarrow_i \mathcal{B}$, where A is an atomic formula (usually called the *head*) and $\mathcal{B}$ (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 aggregations. Rules with an empty body are called *facts*. A *goal* is a body submitted as a query to the system. Variables in a rule are assumed to be governed by universal quantifiers. Roughly speaking, a multi-adjoint logic program is a set of pairs $\langle \mathcal{R}; v \rangle$, where $\mathcal{R}$ is a rule and v is a *truth degree* (a value of L) expressing the confidence which the user of the system has in the truth of the rule $\mathcal{R}$. Often, we will write "$\mathcal{R}$ with v" instead of $\langle \mathcal{R}; v \rangle$.

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 sub-expression (usually an atom) which occurs in the –possibly empty– context $\mathcal{C}[]$ whereas $\mathcal{C}[A/A']$ means the replacement of A by A' in context $\mathcal{C}[]$. Moreover, $\mathcal{V}ar(s)$ denotes the set of distinct variables occurring in the syntactic object s, $\theta[\mathcal{V}ar(s)]$ refers to the substitution obtained from θ by restricting its domain to $\mathcal{V}ar(s)$ and $mgu(E)$ denotes the *most general unifier* of an equation set E. In the following definition, we always consider that A is the selected atom in goal $\mathcal{Q}$.

Definition 1 (Admissible Step). *Let $\mathcal{Q}$ be a goal and let σ be a substitution. The pair $\langle \mathcal{Q}; \sigma \rangle$ is a* state. *Given a program $\mathcal{P}$, an* admissible computation *is formalized as a state transition system, whose transition relation $\rightarrow_{AS}$ is the smallest relation satisfying the following* admissible rules*:*

1) $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v\&_i\mathcal{B}])\theta; \sigma\theta \rangle$ *if* $\theta = mgu(\{A' = A\})$, $\langle A' \leftarrow_i \mathcal{B}; v \rangle$ *in* $\mathcal{P}$ *and* $\mathcal{B}$ *is not empty.*
2) $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle$ *if* $\theta = mgu(\{A' = A\})$, *and* $\langle A' \leftarrow_i; v \rangle$ *in* $\mathcal{P}$.

Apart for exploiting atoms by using program rules, in this setting we can also evaluate expressions composed by truth degrees and fuzzy connectives by directly interpreting them w.r.t. lattice L following our definition recasted from [6]:

Definition 2 (Interpretive Step). *Let $\mathcal{P}$ be a program, $\mathcal{Q}$ a goal and σ a substitution. Assume that $[\![@]\!]$ is the truth function of connective $@$ in the lattice*

$\langle L, \preceq \rangle$ *associated to* $\mathcal{P}$*, such that, for values* $r_1, \ldots, r_n, r_{n+1} \in L$*, we have that* $[\![@]\!](r_1, \ldots, r_n) = r_{n+1}$*. Then, we formalize the notion of* interpretive computation *as a state transition system, whose transition relation* $\rightarrow_{IS}$ *is defined as the least one satisfying:* $\langle \mathcal{Q}[@(r_1, \ldots, r_n)]; \sigma \rangle \ \rightarrow_{IS}\ \langle \mathcal{Q}[@(r_1, \ldots, r_n)/r_{n+1}]; \sigma \rangle$

Example 1. In order to illustrate our definitions, consider now the following program $\mathcal{P}$ and lattice $([0,1], \leq)$, where $\leq$ is the usual order on real numbers.

$$\begin{array}{lll} \mathcal{R}_1 : p(X) \leftarrow_{\mathtt{P}} q(X,Y) \&_{\mathtt{G}}\ r(Y) & \mathit{with}\ 0.8 & \mathcal{R}_2 : q(a,Y) \leftarrow_{\mathtt{P}} s(Y)\ \mathit{with}\ 0.7 \\ \mathcal{R}_3 : q(b,Y) \leftarrow_{\mathtt{L}} r(Y) & \mathit{with}\ 0.8 & \mathcal{R}_4 : r(Y) \leftarrow \qquad\quad \mathit{with}\ 0.7 \\ \mathcal{R}_5 : s(b) \leftarrow & \mathit{with}\ 0.9 & \end{array}$$

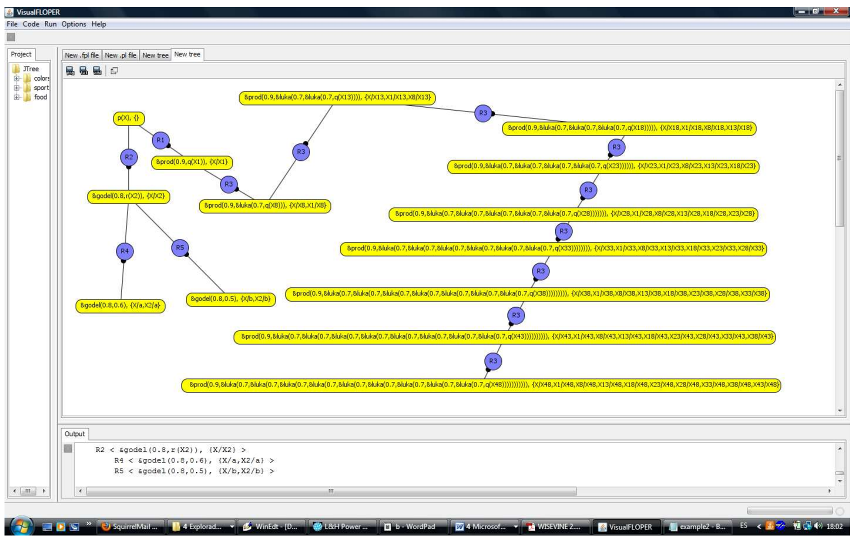
The labels P, G and L mean for *Product logic*, *Gödel intuitionistic logic* and *Łukasiewicz logic*, respectively. That is, $[\![\&_{\mathtt{P}}]\!](x,y) = x \cdot y$, $[\![\&_{\mathtt{G}}]\!](x,y) = min(x,y)$, and $[\![\&_{\mathtt{L}}]\!](x,y) = max(0, x+y-1)$. In the following derivation for the program $\mathcal{P}$ and goal $\leftarrow p(X)$, we underline the selected expression in each computation step, also indicating as a superscript the rule/connective exploited/evaluated in each admissible/interpretive step (as usual, variables of program rules are renamed after being used):

$$\begin{array}{lll} \langle \underline{p(X)}; \{\} \rangle & \rightarrow_{AS1}{}^{\mathcal{R}_1} & \langle 0.8\ \&_{\mathtt{P}}\ (\underline{q(X_1,Y_1)}\ \&_{\mathtt{G}}\ r(Y_1)); \{X/X_1\} \rangle \\ & \rightarrow_{AS1}{}^{\mathcal{R}_2} & \langle 0.8\ \&_{\mathtt{P}}\ ((0.7\ \&_{\mathtt{P}}\ \underline{s(Y_2)})\ \&_{\mathtt{G}}\ r(Y_2)); \{X/a, X_1/a, Y_1/Y_2\} \rangle \\ & \rightarrow_{AS2}{}^{\mathcal{R}_5} & \langle 0.8\ \&_{\mathtt{P}}\ (\underline{(0.7\ \&_{\mathtt{P}}\ 0.9)}\ \&_{\mathtt{G}}\ r(b)); \{X/a, X_1/a, Y_1/b, Y_2/b\} \rangle \\ & \rightarrow_{IS}{}^{\&_{\mathtt{P}}} & \langle 0.8\ \&_{\mathtt{P}}\ (0.63\ \&_{\mathtt{G}}\ \underline{r(b)}); \{X/a, X_1/a, Y_1/b, Y_2/b\} \rangle \\ & \rightarrow_{AS2}{}^{\mathcal{R}_4} & \langle 0.8\ \&_{\mathtt{P}}\ \underline{(0.63\ \&_{\mathtt{G}}\ 0.7)}; \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b\} \rangle \\ & \rightarrow_{IS}{}^{\&_{\mathtt{G}}} & \langle \underline{0.8\ \&_{\mathtt{P}}\ 0.63}; \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b\} \rangle \\ & \rightarrow_{IS}{}^{\&_{\mathtt{P}}} & \langle 0.504; \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b\} \rangle \end{array}$$

So, after focusing our interest in variables belonging to the original goal, the final fuzzy computed answer (f.c.a., in brief) is $\langle 0.504; \{X/a\} \rangle$, with the obvious meaning that the original goal is true at a 50.4% when X be a.

3 The FLOPER System

As detailed in [11, 12], our parser 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 (in our case, Sicstus Prolog v.3.12.5), it shows a menu which includes options for loading, parsing, listing and saving fuzzy programs, as well as for executing fuzzy goals. All these actions are based in the translation of the fuzzy code into standard Prolog code. The key point is to extend each atom with an extra argument, called *truth variable* of the form `_TVi`, which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. For instance, the first clause in our target program is translated into: "`p(X, _TV0) : −q(X, Y, _TV1), r(Y, _TV2), and_godel(_TV1, _TV2, _TV3), and_prod(0.8, _TV3, _TV0).` ", where the definition of the "aggregator predicates" are: "`and_prod(X, Y, Z) : −Z is X ∗ Y.`" and "`and_godel(X, Y, Z) : −(X =<`


VisualFLOPER
File Code Run Options Help
Project
JTree
colors
sport
food
New .fpl file
New .pl file
New tree
New tree
p(X), {}
R1
R2
&prod(0.9,q(X1)), {X/X1}
&godel(0.8,r(X2)), {X/X2}
R3
&prod(0.9,&luka(0.7,q(X8))), {X/X8,X1/X8}
R3
&prod(0.9,&luka(0.7,&luka(0.7,q(X13)))), {X/X13,X1/X13,X8/X13}
R3
&prod(0.9,&luka(0.7,&luka(0.7,&luka(0.7,q(X18))))), {X/X18,X1/X18,X8/X18,X13/X18}
R3
&prod(0.9,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,q(X23)))))), {X/X23,X1/X23,X8/X23,X13/X23,X18/X23}
R3
&prod(0.9,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,q(X28))))))), {X/X28,X1/X28,X8/X28,X13/X28,X18/X28,X23/X28}
R3
&prod(0.9,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,q(X33)))))))), {X/X33,X1/X33,X8/X33,X13/X33,X18/X33,X23/X33,X28/X33}
R3
&prod(0.9,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,q(X38))))))))), {X/X38,X1/X38,X8/X38,X13/X38,X18/X38,X23/X38,X28/X38,X33/X38}
R3
&prod(0.9,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,q(X43)))))))))), {X/X43,X1/X43,X8/X43,X13/X43,X18/X43,X23/X43,X28/X43,X33/X43,X38/X43}
R3
&prod(0.9,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,&luka(0.7,q(X48))))))))))), {X/X48,X1/X48,X8/X48,X13/X48,X18/X48,X23/X48,X28/X48,X33/X48,X38/X48,X43/X48}
R4
R5
&godel(0.8,0.6), {X/a,X2/a}
&godel(0.8,0.5), {X/b,X2/b}
Output
SquirrelMail ...
4 Explorad...
WinEdt - [D...
L&H Power ...
b - WordPad
4 Microsof...
WISEVINE 2....
VisualFLOPER
example2 - B...
ES
18:02

```
R2 < &godel(0.8,r(X2)), {X/X2} >
    R4 < &godel(0.8,0.6), {X/a,X2/a} >
    R5 < &godel(0.8,0.5), {X/b,X2/b} >
```

Fig. 1. Building a graphical interface for FLOPER.

$\mathtt{Y}, \mathtt{Z} = \mathtt{X}; \mathtt{X} > \mathtt{Y}, \mathtt{Z} = \mathtt{Y}$).". The last clause in the program, becomes the pure Prolog fact "s(b, 0.9)." while 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 the Prolog interpreter returns the two desired fuzzy computed answers `[Truth_degree=0.504,X=a]` and `[Truth_degree=0.4,X=b]`.

The previous set of options suffices for running fuzzy programs: all internal computations (including compiling and executing) are pure Prolog derivations whereas inputs (fuzzy programs and goals) and outputs (fuzzy computed answers) have always a fuzzy taste, which produces the illusion on the final user of being working with a purely fuzzy logic programming tool.

Moreover, it is also possible to select into the FLOPER's goal menu, options "`tree`" and "`depth`", which are useful for tracing execution trees and fixing the maximum length allowed for their branches (initially 3), respectively. Working with these options is crucial when the "`run`" choice fails: remember that this last option is based on the generation of pure logic SLD-derivations which might fall in loop or directly fail in some cases as the experiments of [11] show, in contrast with the traces (based on finite, non-failed, admissible derivations) that the "`tree`" option displays. By using the graphical interface we are implementing for FLOPER, Figure 1 shows a tree evidencing an infinite branch where states are colored in yellow and program rules exploited in admissible steps are enclosed in circles.

4 Fuzzy Computed Answers with Extended Information

Strongly related with the last paragraph of the previous section and also connecting with the results we plan to explain in what follows, the "`ismode`" choice is useful for deciding among three levels of detail when visualizing the interpretive computations performed during the generation of "evaluation trees". This last option, together with the possibility of loading new lattices into the system, represents our last developments performed on FLOPER, as reported in [12].

```
member(X) :- number(X),0=<X,X=<1.

bot(0).                    top(1).

leq(X,Y) :- X=<Y.

and\_luka(X,Y,Z):- pri_add(X,Y,U1),pri_sub(U1,1,U2),pri_max(0,U2,Z).
and_godel(X,Y,Z):- pri_min(X,Y,Z).
and_prod(X,Y,Z) :- pri_prod(X,Y,Z).

or_luka(X,Y,Z)  :- pri_add(X,Y,U1),pri_min(U1,1,Z).
or_godel(X,Y,Z) :- pri_max(X,Y,Z).
or_prod(X,Y,Z)  :- pri_prod(X,Y,U1),pri_add(X,Y,U2),pri_sub(U2,U1,Z).

agr_aver(X,Y,Z) :- pri_add(X,Y,U),pri_div(U,2,Z).

pri_add(X,Y,Z)  :- Z is X+Y.    pri_min(X,Y,Z) :- (X=<Y,Z=X;X>Y,Z=Y).
pri_sub(X,Y,Z)  :- Z is X-Y.    pri_max(X,Y,Z) :- (X=<Y,Z=Y;X>Y,Z=X).
pri_prod(X,Y,Z) :- Z is X * Y.  pri_div(X,Y,Z) :- Z is X/Y.
```

Fig. 2. Multi-adjoint lattice modeling truth degrees in the real interval [0,1].

We have recently conceived a very easy way to model truth-degree lattices for being included into the FLOPER tool by using the "`lat/show`" options. All relevant components of each lattice can be encapsulated inside a Prolog file which must necessarily contain the definitions of a minimal set of predicates defining the set of valid elements (predicate `member`), including special mentions to the "`top`" and "`bottom`" ones, the full or partial ordering established among them (predicate `leq`), as well as the repertoire of fuzzy connectives which can be used for their subsequent manipulation.

For instance, in Figure 2 we have modeled the lattice that we used in our examples, which enables the possibility of working with truth degrees in the infinite space of the real numbers between 0 and 1, allowing too the possibility of using conjunction and disjunction operators recasted from the three typical fuzzy logics proposals described before (i.e., the *Łukasiewicz*, *Gödel* and *product* logics), as well as a useful description for the hybrid aggregator *average*. Note also that we have included definitions for auxiliary predicates, whose names always begin with the prefix "pri_". All of them are intended to describe primitive/arithmetic operators (in our case $+$, $-$, $*$, $/$, *min* and *max*) in a Prolog style, for being appropriately called from the bodies of clauses defining predicates with higher levels of expressivity (this is the case for instance, of the three kinds of fuzzy connectives we are considering: conjuntions, disjunctions and agreggations).

One step beyond, we can also conceive a more complex lattice whose elements could have two components, coping with truth degrees and "labels" collecting information about the program rules and fuzzy connectives used when executing programs. In order to be loaded into FLOPER, we must define in Prolog the new lattice, whose elements could be expressed, for instance, as data terms of the form "info(Fuzzy_Truth_Degree, Label)". Moreover, the clauses defining some predicates required for managing them are:

```
member(info(X,_)):-number(X),0=<X,X=<1.

bot(info(0,_)).         top(info(1,_)).

leq(info(X1,_),info(X2,_)) :- X1 =< X2.

and_prod(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-
             pri_prod(X1,Y1,Z1,DatPROD),pri_app(X2,Y2,Dat1),
             pri_app(Dat1,'&PROD.',Dat2),pri_app(Dat2,DatPROD,Z2).

pri_app(X,Y,Z):-name(X,L1),name(Y,L2),append(L1,L2,L3),name(Z,L3).

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

Here, we have seen that when implementing for instance the conjunction operator of the Product Logic, in the second component of our extended notion of "truth degree", we have *appended* the labels of its arguments with the label '&PROD.' (see clauses defining and_prod, pri_app and append). Of course, in the fuzzy program to be run, we must also take into account the use of labels associated to the program rules. For instance, in our example the first rule must have the form:

```
   p(X) <prod q(X,Y) &godel r(Y) with info(0.8,'RULE1.').
```

And now, after executing goal p(X) we obtain the two desired computed answers (including the sequence of program rules exploited and connective definitions evaluated till finding each solution):

```
 [ Truth_degree=info(0.504,    RULE1.RULE2.RULE5.&PROD.
                               RULE4.&GODEL.&PROD.}),         X=a]

 [ Truth_degree=info(0.4,      RULE1.RULE3.RULE4.&LUKA.
                               RULE4.&GODEL.&PROD.),          X=b]
```

5 Conclusions and Future Work

The experience acquired in our research group regarding the design of techniques and methods based on fuzzy logic in close relationship with the so-called multi-adjoint logic programming approach ([2, 5, 7, 4]), has motivated our interest for putting in practice all our developments around the design of the FLOPER environment [11, 12]. Our philosophy is to friendly connect this fuzzy framework with Prolog programmers: our system, apart for being implemented in Prolog, also translates the fuzzy code to classical clauses and, what is more, in this paper we have also shown that a wide range of lattices modeling powerful and flexible notions of truth degrees can be easily used into FLOPER for augmenting fuzzy computed answers with proof traces without requiring additional cost.

Apart for our ongoing efforts devoted to providing FLOPER with a graphical interface as illustrated in Figure 1, nowadays we are especially interested in implementing all the manipulation tasks developed in our group on fold/unfold transformations [2, 5], partial evaluation [7] and thresholded tabulation [4].

References

1. J. F. Baldwin, T. P. Martin, and B. W. Pilsworth. *Fril- Fuzzy and Evidential Reasoning in Artificial Intelligence.* John Wiley & Sons, Inc., 1995.
2. J.A. Guerrero and G. Moreno. Optimizing fuzzy logic programs by unfolding, aggregation and folding. *Electronic Notes in Theoretical Computer Science*, 219:19–34, 2008.
3. M. Ishizuka and N. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In Aravind K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI'85)*, pages 701–703. Morgan Kaufmann, 1985.
4. P. Julián, J. Medina, G. Moreno, and M. Ojeda. Efficient thresholded tabulation for fuzzy query answering. *Studies in Fuzziness and Soft Computing (Foundations of Reasoning under Uncertainty)*, 249:125–141, 2010.
5. P. Julián, G. Moreno, and J. Penabad. On Fuzzy Unfolding. A Multi-adjoint Approach. *Fuzzy Sets and Systems, Elsevier*, 154:16–33, 2005.
6. P. Julián, G. Moreno, and J. Penabad. Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. *Journal of Universal Computer Science*, 12(11):1679–1699, 2006.
7. P. Julián, G. Moreno, and J. Penabad. An Improved Reductant Calculus using Fuzzy Partial Evaluation Techniques. *Fuzzy Sets and Systems*, 160:162–181, 2009.
8. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987.
9. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. A procedural semantics for multi-adjoint logic programing. *Progress in Artificial Intelligence, EPIA'01, Springer-Verlag, Lecture Notes in Artificial Intelligence*, 2258(1):290–297, 2001.
10. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146:43–62, 2004.
11. P. J. Morcillo and G. Moreno. Programming with Fuzzy Logic Rules by using the FLOPER Tool. In N. Bassiliades et al., editors, *Proc. of RuleML'08*, pages 119-126. Springer Verlag, LNCS 3521, 2008.
12. P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In M. Dean et al., editors, *Proc. of RuleML'10*, pages 20-34. Springer Verlag, LNCS 6403, 2010.