A Practical Management of Fuzzy Truth-Degrees using FLOPER [⋆]

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. During the last two years, our developments regarding the design of the FLOPER tool ("Fuzzy LOgic Programming Environment for Research"), have been devoted to implant in its core a rule-based, easy representation of lattices representing fuzzy notions of truth degrees beyond the boolean case, in order to work with flexible programs belonging to the so-called *multi-adjoint logic approach*. Now, the system improves its initial running/debugging/tracing capabilities for managing this kind of fuzzy logic programs, with new options for manipulating in a classical Prolog style the mathematical foundations of the enrichment introduced by *multi-adjoint lattices*. In particular, we show that for a given program and query, many different answers can be obtained when changing the assumption of truth in a single work session. The experience related here evidences the expressive power of Prolog rules (i.e., clauses) for implementing rich versions of multi-adjoint lattices in a very easy way, as well as its crucial role in further fuzzy logic computations.

1 Introduction

Research in the fields of *Declarative Programming* and *Fuzzy Logic* have traditionally provided programming languages and techniques with important applications in the areas of AI, rule-based systems, and so on [3, 17, 21]. In particular, *Logic Programming* [16] has been widely used for problem solving and knowledge representation in the past. Nevertheless, traditional logic languages do not incorporate techniques or constructs to explicitly deal with uncertainty and approximate reasoning in a natural way.

To fulfill this gap, *Fuzzy Logic Programming* has emerged as an interesting and still growing research area trying to consolidate the efforts for introducing fuzzy logic into logic programming. During the last decades, several fuzzy logic programming systems have been developed, such as [2, 4, 6, 15, 13, 27] and the many-valued logic programming language of [25, 26], where the classical inference

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

mechanism of SLD–Resolution has been replaced by a fuzzy variant which is able to handle partial truth and to reason with uncertainty.

This is the case of *multi-adjoint logic programming* [20, 18, 19], a powerful and promising approach in the area. In this framework, 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 plus a substitution (initially the empty substitution, denoted by *id*). *Admissible steps* (a generalization of the classical *modus ponens* inference rule) are systematically applied on goals in a similar way to classical resolution steps in pure logic programming, thus returning a state composed by a computed substitution together with an expression where all atoms have been exploited. Next, during the so called interpretive phase (see [10, 22]), this expression is 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 used in pure logic programming.

The main goal of the present paper is to present our last developments performed on the FLOPER system (see [1, 21, 24] and visit `http://www.dsi.uclm.es/investigacion/dect/FLOPERpage.htm`) which enables the introduction of different notions of multi-adjoint lattices for managing truth degrees even in a single work-session without changing a given multi-adjoint logic program and goal. Nowadays, the tool provides facilities for executing and debugging (by generating declarative traces) 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 structure of the paper is as follows. In Section 2, we summarize the main features of multi-adjoint logic programming, both language syntax and procedural semantics. Section 3 presents a discussion on multi-adjoint lattices and their nice representation by using standard Prolog code, in order to facilitate its further assimilation inside the FLOPER tool, as described in Section 4. Finally, in Section 5 we give our conclusions and some lines of future work.

2 Multi-Adjoint Logic Programming

This section summarizes the main features of multi-adjoint logic programming (see [20, 18, 19] for a complete formulation of this framework). In what follows, we will use the abbreviation MALP for referencing programs belonging to this setting.

2.1 MALP Syntax

We work with a first order language, $\mathcal{L}$, containing variables, constants, function symbols, predicate symbols, and several (arbitrary) connectives to increase language expressiveness: implication connectives ($\leftarrow_1, \leftarrow_2, \ldots$); conjunctive operators (denoted by $\&_1, \&_2, \ldots$), disjunctive operators ($\vee_1, \vee_2, \ldots$), and hybrid operators (usually denoted by $@_1, @_2, \ldots$), all of them are grouped under the name of “aggregators”.

Aggregation operators are useful to describe/specify user preferences. An aggregation operator, when interpreted as a truth function, may be an arithmetic mean, a weighted sum or in general any monotone application whose arguments are values of a complete bounded lattice L. For example, if an aggregator @ is interpreted as $[\![@]\!](x,y,z) = (3x+2y+z)/6$, we are giving the highest preference to the first argument, then to the second, being the third argument the least significant.

Although these connectives are binary operators, we usually generalize them as functions with an arbitrary number of arguments. So, we often write $@(x_1,\ldots,x_n)$ instead of $@(x_1,\ldots,@(x_{n-1},x_n),\ldots)$. By definition, the truth function for an n-ary aggregation operator $[\![@]\!] : L^n \to L$ is required to be monotonous 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 which is intended to the evaluation of *modus ponens* [20]. More exactly, in this setting the following items must be satisfied:

- $\langle L, \preceq \rangle$ is a bounded lattice, i.e. it has bottom and top elements, denoted by $\bot$ and $\top$, respectively.
- Each operation $\&_i$ is increasing in both arguments.
- Each operation $\leftarrow_i$ is increasing in the first argument and decreasing in the second.
- If $\langle \&_i, \leftarrow_i \rangle$ is an *adjoint pair* in $\langle L, \preceq \rangle$ then, for any $x, y, z \in L$, we have that: $x \preceq (y \leftarrow_i z)$ if and only if $(x \&_i z) \preceq y$.

This last condition, called *adjoint property*, could be considered the most important feature of the framework (in contrast with many other approaches) which justifies most of its properties regarding crucial results for soundness, completeness, applicability, etc.

In general, L may be the carrier of any complete bounded lattice where a L-expression is a well-formed expression composed by values and connectives of L, as well as variable symbols and *primitive operators* (i.e., arithmetic symbols such as $*, +, min$, etc...).

In what follows, we assume that the truth function of any connective @ in L is given by its corresponding *connective definition*, that is, an equation of the form $@(x_1,\ldots,x_n) \triangleq E$, where E is a L-expression not containing variable symbols apart from $x_1,\ldots,x_n$. For instance, in what follows we will be mainly concerned with the following classical set of adjoint pairs (conjunctors and implications) in $\langle [0,1], \leq \rangle$, where labels `L`, `G` and `P` mean respectively *Łukasiewicz logic*, *Gödel intuitionistic logic* and *product logic* (which different capabilities for modeling *pessimist*, *optimist* and *realistic scenarios*, respectively):

$\&_{\mathtt{P}}(x,y) \triangleq x * y$	$\leftarrow_{\mathtt{P}} (x,y) \triangleq \min(1, x/y)$	*Product*
$\&_{\mathtt{G}}(x,y) \triangleq \min(x,y)$	$\leftarrow_{\mathtt{G}} (x,y) \triangleq \begin{cases} 1 & \text{if } y \leq x \\ x & \text{otherwise} \end{cases}$	*Gödel*
$\&_{\mathtt{L}}(x,y) \triangleq \max(0, x+y-1)$	$\leftarrow_{\mathtt{L}} (x,y) \triangleq \min\{x-y+1, 1\}$	*Łukasiewicz*

A *rule* is a formula $H \leftarrow_i \mathcal{B}$, where H 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, conjunctions, disjunctions and aggregations. A *goal* is a body submitted as a query to the system. Roughly speaking, a multi-adjoint logic program is a set of pairs $\langle \mathcal{R}; \alpha \rangle$ (we often write "$\mathcal{R}$ *with* α"), where $\mathcal{R}$ is a rule and α is a *truth degree* (a value of L) expressing the confidence of a programmer in the truth of rule $\mathcal{R}$. By abuse of language, we sometimes refer a tuple $\langle \mathcal{R}; \alpha \rangle$ as a "rule".

2.2 MALP Procedural Semantics

The procedural semantics of the multi–adjoint logic language $\mathcal{L}$ can be thought of as an operational phase (based on admissible steps) followed by an interpretive one. In the following, $\mathcal{C}[A]$ denotes a formula where A is a sub-expression which occurs in the –possibly empty– context $\mathcal{C}[]$. Moreover, $\mathcal{C}[A/A']$ means the replacement of A by A' in context $\mathcal{C}[]$, whereas $\mathcal{V}ar(s)$ refers to the set of distinct variables occurring in the syntactic object s, and $\theta[\mathcal{V}ar(s)]$ denotes the substitution obtained from θ by restricting its domain to $\mathcal{V}ar(s)$.

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 *and we denote by $\mathcal{E}$ the set of states. Given a program $\mathcal{P}$, an* admissible computation *is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathcal{E} \times \mathcal{E})$ is the smallest relation satisfying the following* admissible rules *(where we always consider that A is the selected atom in $\mathcal{Q}$ and $mgu(E)$ denotes the* most general unifier *of an equation set E [14]):*

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

2) $\langle \mathcal{Q}[A]; \sigma \rangle \quad \rightarrow_{AS} \quad \langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle$,
   if $\theta = mgu(\{A' = A\})$ and $\langle A' \leftarrow_i; v \rangle$ in $\mathcal{P}$.

Note that the second case could be subsumed by the first one, after expressing each fact $\langle A' \leftarrow_i; v \rangle$ as a program rule of the form $\langle A' \leftarrow_i \top; v \rangle$. As usual, rules are taken renamed apart. We shall use the symbols $\rightarrow_{AS1}$ and $\rightarrow_{AS2}$ to distinguish between computation steps performed by applying one of the specific admissible rules. Also, the application of a rule on a step will be annotated as a superscript of the $\rightarrow_{AS}$ symbol.

Definition 2. *Let $\mathcal{P}$ be a program, $\mathcal{Q}$ a goal and "id" the empty substitution. An* admissible derivation *is a sequence $\langle \mathcal{Q}; id \rangle \rightarrow_{AS} \ldots \rightarrow_{AS} \langle \mathcal{Q}'; \theta \rangle$. When $\mathcal{Q}'$ is a formula not containing atoms (i.e., a L-expression), the pair $\langle \mathcal{Q}'; \sigma \rangle$, where $\sigma = \theta[\mathcal{V}ar(\mathcal{Q})]$, is called an* admissible computed answer *(a.c.a.) for that derivation.*

Example 1. Let $\mathcal{P}$ be the multi-adjoint fuzzy logic program described in Figure 1 where the equation defining the average aggregator $@_{\texttt{aver}}$ must obviously has the form: $@_{\texttt{aver}}(x_1, x_2) \triangleq (x_1 + x_2)/2$. Now, we can generate the admissible

Multi-adjoint logic program $\mathcal{P}$:

$$\begin{array}{llllll}
\mathcal{R}_1 : & p(X) & \leftarrow_{\mathtt{P}} & \&_{\mathtt{G}}(q(X), @_{\mathtt{aver}}(r(X), s(X))) & \textit{with} & 0.9 \\
\mathcal{R}_2 : & q(a) & \leftarrow & & \textit{with} & 0.8 \\
\mathcal{R}_3 : & r(X) & \leftarrow & & \textit{with} & 0.7 \\
\mathcal{R}_4 : & s(X) & \leftarrow & & \textit{with} & 0.5
\end{array}$$

Admissible derivation:

$$\begin{array}{ll}
\langle \underline{p(X)};\ id \rangle & \rightarrow_{AS1}{}^{\mathcal{R}_1} \\
\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\underline{q(X_1)}, @_{\mathtt{aver}}(r(X1), s(X1)))); \{X/X_1\} \rangle & \rightarrow_{AS2}{}^{\mathcal{R}_2} \\
\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(\underline{r(a)}, s(a)))); \{X/a, X_1/a\} \rangle & \rightarrow_{AS2}{}^{\mathcal{R}_3} \\
\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(0.7, \underline{s(a)}))); \{X/a, X_1/a, X_2/a\} \rangle & \rightarrow_{AS2}{}^{\mathcal{R}_4} \\
\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(0.7, 0.5))); \{X/a, X_1/a, X_2/a, X_3/a\} \rangle &
\end{array}$$

Interpretive derivation:

$$\begin{array}{ll}
\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, \underline{@_{\mathtt{aver}}(0.7, 0.5)})); \{X/a\} \rangle & \rightarrow_{IS} \\
\langle \&_{\mathtt{P}}(0.9, \underline{\&_{\mathtt{G}}(0.8, 0.6)}); \{X/a\} \rangle & \rightarrow_{IS} \\
\langle \underline{\&_{\mathtt{P}}(0.9, 0.6)}; \{X/a\} \rangle & \rightarrow_{IS} \\
\langle 0.54; \{X/a\} \rangle. &
\end{array}$$

Fig. 1. MALP program $\mathcal{P}$ with admissible/interpretive derivations for goal $p(X)$.

derivation shown in Figure 1 (we underline the selected atom in each step). So, the admissible computed answer (a.c.a.) in this case is composed by the pair: $\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(0.7, 0.5))); \theta \rangle$, where θ only refers to bindings related with variables in the goal, i.e., $\theta = \{X/a, X_1/a, X_2/a, X_3/a\}[\mathcal{V}ar(p(X))] = \{X/a\}$.

If we exploit all atoms of a given goal, by applying admissible steps as much as needed during the operational phase, then it becomes a formula with no atoms (a L-expression) which can be then directly interpreted w.r.t. lattice L by applying the following definition we initially presented in [10]:

Definition 3 (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 $\rightarrow_{IS} \subseteq (\mathcal{E} \times \mathcal{E})$ is defined as the least one satisfying:*

$$\langle \mathcal{Q}[@(r_1, \ldots, r_n)]; \sigma \rangle \quad \rightarrow_{IS} \quad \langle \mathcal{Q}[@(r_1, \ldots, r_n)/r_{n+1}]; \sigma \rangle$$

Definition 4. *Let $\mathcal{P}$ be a program and $\langle \mathcal{Q}; \sigma \rangle$ an a.c.a., that is, $\mathcal{Q}$ is a goal not containing atoms (i.e., a L-expression). An* interpretive derivation *is a sequence $\langle \mathcal{Q}; \sigma \rangle \rightarrow_{IS} \ldots \rightarrow_{IS} \langle \mathcal{Q}'; \sigma \rangle$. When $\mathcal{Q}' = r \in L$, being $\langle L, \preceq \rangle$ the lattice associated to $\mathcal{P}$, the state $\langle r; \sigma \rangle$ is called a* fuzzy computed answer *(f.c.a.) for that derivation.*

Example 2. If we complete the previous derivation of Example 1 by applying 3 interpretive steps in order to obtain the final f.c.a. $\langle 0.54; \{X/a\} \rangle$, we generate the interpretive derivation shown in Figure 1.

3 Truth-Degrees and Multi-adjoint Lattices in Practice

We have recently conceived a very easy way to model truth-degree lattices for being included into the FLOPER tool. 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 (including special mentions to the "`top`" and "`bottom`" ones), the full or partial ordering established among them, as well as the repertoire of fuzzy connectives which can be used for their subsequent manipulation. In order to simplify our explanation, assume that file "bool.pl" refers to the simplest notion of (a binary) adjoint lattice, thus implementing the following set of predicates:

– `member/1` which is satisfied when being called with a parameter representing a valid truth degree. In the case of finite lattices, it is also recommend to implement `members/1` which returns in one go a list containing the whole set of truth degrees. For instance, in the Boolean case, both predicates can be simply modeled by the Prolog facts: `member(0).`, `member(1).` and `members([0,1]).`

– `bot/1` and `top/1` obviously answer with the top and bottom element of the lattice, respectively. Both are implemented into "bool.pl" as `bot(0).` and `top(1).`

– `leq/2` models the ordering relation among all the possible pairs of truth degrees, and obviously it is only satisfied when it is invoked with two elements verifying that the first parameter is equal or smaller than the second one. So, in our example it suffices with including into "bool.pl" the facts: `leq(0,X).` and `leq(X,1).`

– Finally, given some fuzzy connectives of the form $\&_{label_1}$ (conjunction), $\vee_{label_2}$ (disjunction) or $@_{label_3}$ (aggregation) with arities n_1, n_2 and n_3 respectively, we must provide clauses defining the *connective predicates* "`and_`*label*$_1$`/(n`$_1$`+1)`", "`or_`*label*$_2$`/(n`$_2$`+1)`" and "`agr_`*label*$_3$`/(n`$_3$`+1)`", where the extra argument of each predicate is intended to contain the result achieved after the evaluation of the proper connective. For instance, in the Boolean case, the following two facts model in a very easy way the behaviour of the classical conjunction operation: `and_bool(0,_,0). and_bool(1,X,X).`

```
member(X) :- number(X),0=<X,X=<1.  %% no members/1 (infinite lattice)

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] ("num.pl").

The reader can easily check that the use of lattice "bool.pl" when working with MALP programs whose rules have the form:

"$A \quad \leftarrow_{bool} \quad \&_{bool}(B_1, \ldots, B_n) \quad with \quad 1$"

.... being A and B_i typical atoms[1], successfully mimics the behaviour of classical Prolog programs where clauses accomplish with the shape "$A \quad :- \quad B_1, \ldots, B_n$". As a novelty in the fuzzy setting, when evaluating goals according to the procedural semantics described in Section 2, each output will contain the corresponding Prolog's substitution (i.e., the *crisp* notion of computed answer obtained by means of classical SLD-resolution) together with the maximum truth degree 1.

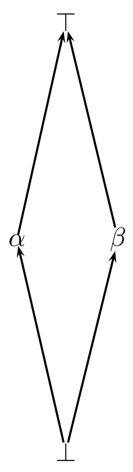
On the other hand and following the Prolog style regulated by the previous guidelines, in file "num.lat" we have included the clauses shown in Figure 2. Here, we have modeled the more flexible lattice (that we will mainly use in our examples, beyond the boolean case) which enables the possibility of working with truth degrees in the infinite space (note that this condition disables the implementation of the consulting predicate "`members/1`") 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

[1] Here we also assume that several versions of the classical conjunction operation have been implemented with different arities.

three kinds of fuzzy connectives we are considering: conjuntions, disjunctions and agreggations).

Since till now we have considered two classical, fully ordered lattices (with a finite and infinite number of elements, collected in files "bool.pl" and "num.pl", respectively), we wish now to introduce a different case coping with a very simple lattice where not always any pair of truth degrees are comparable. So, consider the following partially ordered multi-adjoint lattice in the diagram below for which the conjunction and implication connectives based on the *Gödel* intuitionistic logic described in Section 2 conform an adjoint pair.... but with the particularity now that, in the general case, the *Gödel*'s conjunction must be expressed as $\&_{\mathtt{G}}(x, y) \triangleq inf(x, y)$, where it is important to note that we must replace the use of "*min*" by "*inf*" in the connective definition.



```
member(bottom).  member(alpha).
member(beta).     member(top).

members([bottom,alpha,beta,top]).

leq(bottom,X). leq(alpha,alpha). leq(alpha,top).
leq(beta,beta). leq(beta,top). leq(X,top).

and_godel(X,Y,Z) :- pri_inf(X,Y,Z).

pri_inf(bottom,X,bottom):-!.
pri_inf(alpha,X,alpha):-leq(alpha,X),!.
pri_inf(beta,X,beta):-leq(beta,X),!.
pri_inf(top,X,X):-!.
pri_inf(X,Y,bottom).
```

To this end, observe in the Prolog code accompanying the figure above that we have introduced five clauses defining the new primitive operator "`pri_inf/3`" which is intended to return the *infimum* of two elements. Related with this fact, we must point out the following aspects:

- Note that since truth degrees α and β (or their corresponding representations as Prolog terms "`alpha`" and "`beta`" used for instance in the definition(s) of "`members(s)/1`") are incomparable then, any call to goals of the form "`?- leq(alpha,beta).`" or "`?- leq(beta,alpha).`" will always fail.

- Fortunately, a goal of the form "`?- pri_inf(alpha,beta,X).`", or alternatively "`?- pri_inf(beta,alpha,X).`", instead of failing, successfully produces the desired result "`X=bottom`".

- Note anyway that the implementation of the "`pri_inf/1`" predicate is mandatory for coding the general definition of "`and_godel/3`".

4 The FLOPER System in Action

As detailed in [1, 21], 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 (see Figure 3).

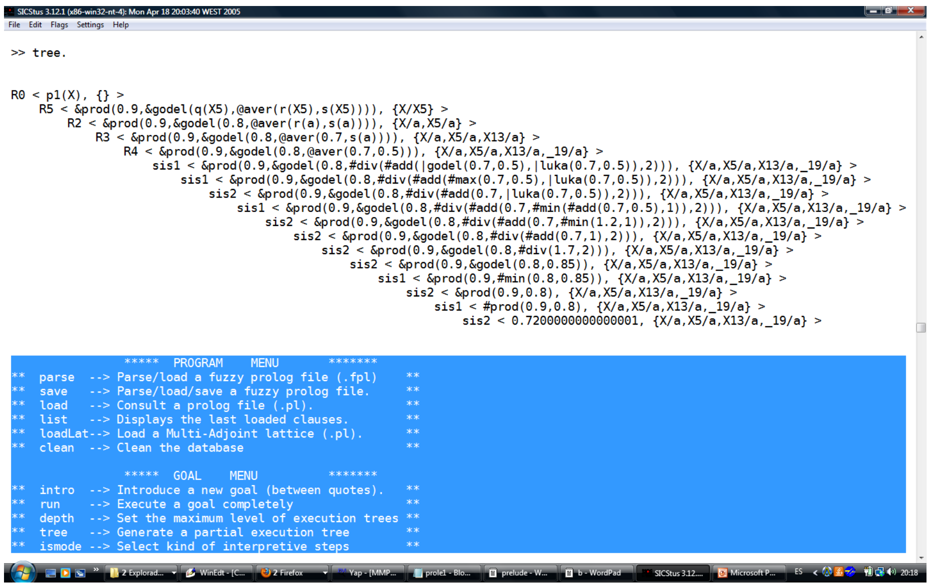
All these actions are based in the translation of the fuzzy code into standard Prolog code. The key point is to extend each atom with an extra argument, called *truth variable* of the form "$\_\mathtt{TV_i}$", which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. For instance, the first clause in our target program is translated into:

```
p(X,_TV0)  :-  q(X,_TV1),
               r(X,_TV2),
               s(X,_TV3),
               agr_aver(_TV2,_TV3,_TV4),
               and_godel(_TV1,_TV4,_TV5),
               and_prod(0.9,_TV5,_TV0).
```

Moreover, the second clause in our target program, becomes the pure Prolog fact "`q(a,0.8)`" 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 desired fuzzy computed answer $[\mathtt{Truth\_degree} = 0.54, \mathtt{X} = \mathtt{a}]$. The previous set of options suffices for running fuzzy programs (the "`run`" choice also uses the clauses contained in "num.pl", which represent the default lattice): 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, thus producing the illusion on the final user of being working with a purely fuzzy logic programming tool.

On the other hand, as showed in the down-middle, dark part of Figure 3, FLOPER has been recently equipped with a new option, called "`loadLat`" for allowing the possibility of changing the multi-adjoint lattice associated to a given program. For instance, assume that "new_num.pl" contains the same Prolog code than "num.pl" with the exception of the definition regarding the average aggregator. Now, instead of computing the average of two truth degrees, let us consider the average between the results achieved after applying to both elements, the disjunctions operators described by Gödel and Łukasiewicz, that is: $@_{\mathtt{aver}}(x_1, x_2) \triangleq (\vee_{\mathtt{G}}(x_1, x_2) + \vee_{\mathtt{L}}(x_1, x_2)) * 0.5$. The corresponding Prolog clause modeling such definition into the "new_num.pl" file could be:

```
agr_aver(X,Y,Z)  :-  or_godel(X,Y,Z1),
                     or_luka(X,Y,Z2),
                     pri_add(Z1,Z2,Z3),
                     pri_prod(Z3,0.5,Z).
```


SICStus 3.12.1 (x86-win32-nt-4): Mon Apr 18 20:03:40 WEST 2005
File Edit Flags Settings Help

```
>> tree.


R0 < p1(X), {} >
    R5 < &prod(0.9,&godel(q(X5),@aver(r(X5),s(X5)))), {X/X5} >
        R2 < &prod(0.9,&godel(0.8,@aver(r(a),s(a)))), {X/a,X5/a} >
            R3 < &prod(0.9,&godel(0.8,@aver(0.7,s(a)))), {X/a,X5/a,X13/a} >
                R4 < &prod(0.9,&godel(0.8,@aver(0.7,0.5))), {X/a,X5/a,X13/a,_19/a} >
                    sis1 < &prod(0.9,&godel(0.8,#div(#add(|godel(0.7,0.5),|luka(0.7,0.5)),2))), {X/a,X5/a,X13/a,_19/a} >
                        sis1 < &prod(0.9,&godel(0.8,#div(#add(#max(0.7,0.5),|luka(0.7,0.5)),2))), {X/a,X5/a,X13/a,_19/a} >
                            sis2 < &prod(0.9,&godel(0.8,#div(#add(0.7,|luka(0.7,0.5)),2))), {X/a,X5/a,X13/a,_19/a} >
                                sis1 < &prod(0.9,&godel(0.8,#div(#add(0.7,#min(#add(0.7,0.5),1)),2))), {X/a,X5/a,X13/a,_19/a} >
                                    sis2 < &prod(0.9,&godel(0.8,#div(#add(0.7,#min(1.2,1)),2))), {X/a,X5/a,X13/a,_19/a} >
                                        sis2 < &prod(0.9,&godel(0.8,#div(#add(0.7,1),2))), {X/a,X5/a,X13/a,_19/a} >
                                            sis2 < &prod(0.9,&godel(0.8,#div(1.7,2))), {X/a,X5/a,X13/a,_19/a} >
                                                sis2 < &prod(0.9,&godel(0.8,0.85)), {X/a,X5/a,X13/a,_19/a} >
                                                    sis1 < &prod(0.9,#min(0.8,0.85)), {X/a,X5/a,X13/a,_19/a} >
                                                        sis2 < &prod(0.9,0.8), {X/a,X5/a,X13/a,_19/a} >
                                                            sis1 < #prod(0.9,0.8), {X/a,X5/a,X13/a,_19/a} >
                                                                sis2 < 0.7200000000000001, {X/a,X5/a,X13/a,_19/a} >


                *****  PROGRAM    MENU       *******
**  parse  --> Parse/load a fuzzy prolog file (.fpl)    **
**  save   --> Parse/load/save a fuzzy prolog file.     **
**  load   --> Consult a prolog file (.pl).             **
**  list   --> Displays the last loaded clauses.        **
**  loadLat--> Load a Multi-Adjoint lattice (.pl).      **
**  clean  --> Clean the database                       **

                *****  GOAL    MENU          *******
**  intro  --> Introduce a new goal (between quotes).   **
**  run    --> Execute a goal completely                **
**  depth  --> Set the maximum level of execution trees **
**  tree   --> Generate a partial execution tree        **
**  ismode --> Select kind of interpretive steps        **
```

Fig. 3. Example of a work session with FLOPER showing "Small Interpretive Steps" and program/goal menus.

and now, by selecting again the "`run`" option (without changing the program and goal), the system would display the new solution: [`Truth_degree` = 0.72, `X` = `a`].

However, when trying to go beyond program execution, the previous method becomes insufficient. In particular, observe that we can only simulate complete fuzzy derivations (by performing the corresponding Prolog derivations based on SLD-resolution) but we can not generate partial derivations or even apply a single admissible step on a given fuzzy expression. This kind of low-level manipulations are mandatory when trying to incorporate to the tool some program transformation techniques such as those based on fold/unfold (i.e., contraction and expansion of sub-expressions of a program using the definitions of this program or of a preceding one, thus generating more efficient code) or partial evaluation we have described in [5, 9, 12]. For instance, our fuzzy unfolding transformation is defined as the replacement of a program rule $\mathcal{R} : (A \leftarrow_i \mathcal{B}$ with $v)$ by the set of rules $\{A\sigma \leftarrow_i \mathcal{B}'$ with $v \mid \langle \mathcal{B}; id \rangle \rightarrow_{AS} \langle \mathcal{B}'; \sigma \rangle\}$, which obviously requires the implementation of mechanisms for generating derivations of a single step, rearranging the body of a program rule, applying substitutions to its head, etc.

To this end, in [21] we have presented a new low-level representation for the fuzzy code which currently offers the possibility of performing debugging actions such as tracing a FLOPER work session. The idea is collect in detail all relevant components associated to each fuzzy rule, such as its number inside the program, composition of the atom conforming its head, kind of implication connecting the head and its body, details about connectives and atoms composing this body and attached weight. For instance, after parsing the first rule of our program, we obtain the following expression which is *asserted* into the interpreter's database as a Prolog fact (which it is never executed directly, in contrast with the high-level, Prolog-based representation, showed at the beginning of this section):

```
rule(1,
     head(atom(pred(p,1),[var('X')])),
     impl(prod),
     body(and(godel,2,
                    [ atom(pred(q,1),[var('X')]),
                      agr(aver,2,[ atom(pred(r,1),[var('X')]),
                                   atom(pred(s,1),[var('X')])
                                 ]
                         )
                    ]
              )
         ),
     td(0.9)
    ).
```

Two more examples: substitutions are modeled by lists of terms of the form link(V, T) where V and T contains the code associated to an original variable and its corresponding (linked) fuzzy term, respectively, whereas a state is represented by a term with functor state/2. We have implemented predicates for manipulating such kind of code at a very low level in order to unify expressions, compose substitutions, apply admisible/interpretive steps, etc...

Looking again to the darked part of Figure 3, observe in the FLOPER's goal menu the "tree" and "depth" options, 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 [21] show, in contrast with the traces (based on finite, non-failed, admissible derivations) that the "tree" option displays. As we are going to illustrate in what follows, the system displays states on different lines, appropriately indented to distinguish the proper relationship -parent/child/grandchild...- among nodes on unfolding trees. Each node contains an state (composed by the corresponding goal and substitution) preceded by the number of the program rule used by the admissible step leading to it (root nodes always labeled with the virtual, non existing rule R0).

Strongly related with these last options, the "ismode" choice showed at the bottom of Figure 3, decides among three levels of detail when visualizing the interpretive phase performed during the generation of "unfolding trees". It is important to remark that together with the possibility of introducing multi-adjoint lattices, it represents our last record achieved in the development of the FLOPER tool. When the user selects such choice, three options are offered:

• "`Large`" means to obtain the final result in one go. For instance, for our running example (with the second notion of "average") FLOPER draws:

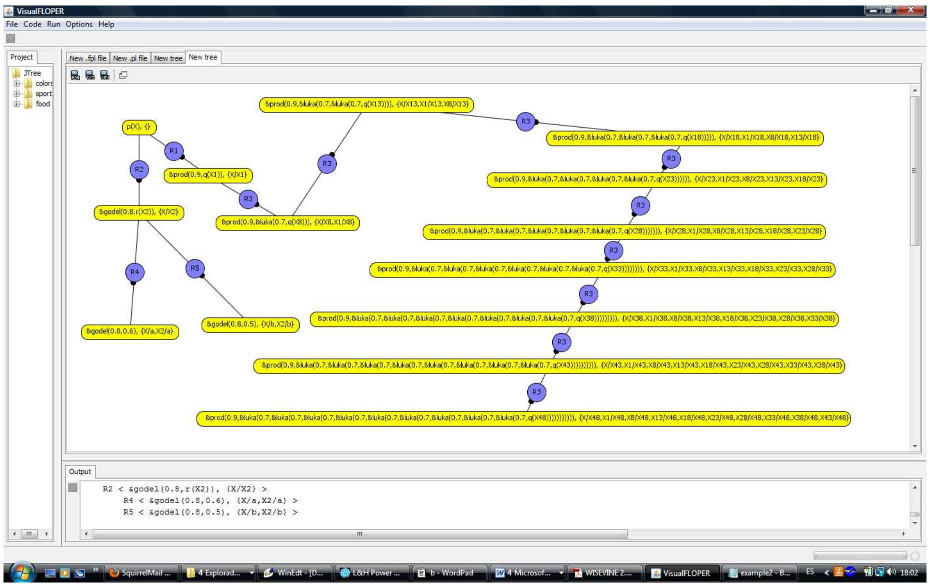
```
R0 <p(X),{}>
 R1 <&prod(0.9,&godel(q(X1),@aver(r(X1),s(X1)))),{X/X1} >
  R2 <&prod(0.9,&godel(0.8,@aver(r(a),s(a)))),{X/a,X1/a}>
   R3 <&prod(0.9,&godel(0.8,@aver(0.7,s(a)))),{X/a,X1/a,X11/a}>
    R4 <&prod(0.9,&godel(0.8,@aver(0.7,0.5))),{X/a,X1/a,X11/a}>
      result < 0.7200000000000001,{X/a,X1/a,X11/a}>
```

• "`Medium`" implements the notion of "interpretive step" according Definition 3 [10] which in our case produces the picture (note here that those states produces during the interpretive phase are preceded by the word "`is`" instead of the number of a program rule, since no rules are exploited in this case in contrast with admissible steps):

```
R0 <p(X),{}>
 R1 <&prod(0.9,&godel(q(X1),@aver(r(X1),s(X1)))),{X/X1}>
  R2 <&prod(0.9,&godel(0.8,@aver(r(a),s(a)))),{X/a,X1/a}>
   R3 <&prod(0.9,&godel(0.8,@aver(0.7,s(a)))),{X/a,X1/a,X11/a}>
    R4 <&prod(0.9,&godel(0.8,@aver(0.7,0.5))),{X/a,X1/a,X11/a}>
     is <&prod(0.9,&godel(0.8,0.85)),{X/a,X1/a,X11/a,_16/a}>
      is <&prod(0.9,0.8),{X/a,X1/a,X11/a,_16/a}>
       is <0.7200000000000001,{X/a,X1/a,X11/a,_16/a}>
```

• "`Small`" allows to visualize in detail both the direct/indirect calls to connective definitions and primitive operators performed along the whole interpretive phase (see [22, 24]). The reader can observe at the beginning of Figure 3, the aspect offered by FLOPER when visualizing in detail the behaviour of our running example, where the set of "small interpretive steps" are (we omit here the initial fourth states- associated to admissible steps- since they coincide with our two last illustrations above):

```
...
 R4 <&prod(0.9,&godel(0.8,@aver(0.7,0.5))),{X/a,X1/a,X11/a,_16/a}>
  sis1 <&prod(0.9,&godel(0.8,#prod(#add(|godel(0.7,0.5),|luka( ..
   sis1 <&prod(0.9,&godel(0.8,#prod(#add(#max(0.7,0.5),|luka(0.7.
    sis2 <&prod(0.9,&godel(0.8,#prod(#add(0.7,|luka(0.7,0.5)), ..
     sis1 <&prod(0.9,&godel(0.8,#prod(#add(0.7,#min(#add(0.7, ...
      sis2 <&prod(0.9,&godel(0.8,#prod(#add(0.7,#min(1.2,1)), ...
       sis2 <&prod(0.9,&godel(0.8,#prod(#add(0.7,1),0.5))), .....
        sis2 <&prod(0.9,&godel(0.8,#prod(1.7,0.5))), {X/a,X1/a ..
         sis2 <&prod(0.9,&godel(0.8,0.85)), {X/a,X1/a,X11/a, ....
          sis1 <&prod(0.9,#min(0.8,0.85)), {X/a,X1/a,X11/a, .....
           sis2 <&prod(0.9,0.8), {X/a,X1/a,X11/a,_16/a}>
            sis1 <#prod(0.9,0.8), {X/a,X1/a,X11/a,_16/a}>
             sis2 <0.7200000000000001, {X/a,X1/a,X11/a,_16/a}>
```


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
R3
R4
R5
&prod(0.9,q(X1)), {X/X1}
&godel(0.8,r(X2)), {X/X2}
&prod(0.9,&luka(0.7,q(X8))), {X/X8,X1/X8}
&prod(0.9,&luka(0.7,&luka(0.7,q(X13)))), {X/X13,X1/X13,X8/X13}
&godel(0.8,0.6), {X/a,X2/a}
&godel(0.8,0.5), {X/b,X2/b}
Output

```
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. 4. Building a graphical interface for FLOPER.

Observe in this last case that during the interpretive phase we apply "small interpretive steps" of kind $\rightarrow_{SIS1}$ or $\rightarrow_{SIS2}$ (according to [24]). The intuitive idea is that, whereas a $\rightarrow_{SIS1}$ step "expands" a connective definition on the next state, the role of evaluating primitive operators is played by $\rightarrow_{SIS2}$ steps. Notice in the figure that each primitive operators is always labeled by prefix "#"). These facts justify why in our Prolog-based implementation of multi-adjoint lattices, clauses defining connective predicates only perform calls to predicates of the form "`and_*`", "`or_*`", "`agr_*`" (useful for identifying further $\rightarrow_{SIS1}$ steps) or "`pri_*`" (associated to $\rightarrow_{SIS2}$ steps).

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 ([10, 5, 9, 11, 12, 7, 8, 22, 23]), has motivated our interest for putting in practice all our developments around the design of the FLOPER environment [21, 24]. 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 (in two different representations) 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 also admit a nice rule-based characterizations into Prolog.

Apart for our ongoing efforts devoted to providing FLOPER with a graphical interface as illustrated in Figure 4[2], nowadays we are especially interested in extending the tool with testing techniques for automatically checking that lattices modeled according the Prolog-based method established in this paper, verify the requirements of our fuzzy setting (with special mention to the *adjoint property*). For the future, we have in mind to provide an interface with rules written in Fuzzy-RuleML and other fuzzy languages like the ones presented in [26, 13] (the XSB system supports GAP).

References

1. J.M. Abietar, P.J. Morcillo, and G. Moreno. Designing a software tool for fuzzy logic programming. In T.E. Simos and G. Maroulis, editors, *Proc. of the International Conference of Computational Methods in Sciences and Engineering ICCMSE'07, Volume 2 (Computation in Modern Science and Engineering)*, pages 1117–1120. American Institute of Physics (distributed by Springer), 2007.
2. J. F. Baldwin, T. P. Martin, and B. W. Pilsworth. *Fril- Fuzzy and Evidential Reasoning in Artificial Intelligence.* John Wiley & Sons, Inc., 1995.
3. Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison Wesley, September 2000.
4. S. Guadarrama, S. Muñoz, and C. Vaucheret. Fuzzy Prolog: A new approach using soft constraints propagation. *Fuzzy Sets and Systems*, 144(1):127–150, 2004.
5. 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.
6. 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.
7. P. Julián, J. Medina, G. Moreno, and M. Ojeda. Thresholded tabulation in a fuzzy logic setting. *Electronic Notes in Theoretical Computer Science*, 248:115–130, 2009.
8. 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.
9. P. Julián, G. Moreno, and J. Penabad. On Fuzzy Unfolding. A Multi-adjoint Approach. *Fuzzy Sets and Systems*, 154:16–33, 2005.
10. 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.
11. P. Julián, G. Moreno, and J. Penabad. Measuring the interpretive cost in fuzzy logic computations. In Francesco Masulli, Sushmita Mitra, and Gabriella Pasi, editors, *Proc. of Applications of Fuzzy Sets Theory, 7th International Workshop on Fuzzy Logic and Applications, WILF 2007, Camogli, Italy, July 7-10*, pages 28–36. Springer Verlag, LNAI 4578, 2007.

[2] Here we show an unfolding tree evidencing an infinite branch where states are colored in yellow and program rules exploited in admissible steps are enclosed in circles.

12. P. Julián, G. Moreno, and J. Penabad. An Improved Reductant Calculus using Fuzzy Partial Evaluation Techniques. *Fuzzy Sets and Systems*, 160:162–181, 2009. doi: 10.1016/j.fss.2008.05.006.
13. M. Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12:335–367, 1992.
14. J. L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.
15. D. Li and D. Liu. *A fuzzy Prolog database system*. John Wiley & Sons, Inc., 1990.
16. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.
17. J.W. Lloyd. Declarative programming for artificial intelligence applications. *SIGPLAN Not.*, 42(9):123–124, 2007.
18. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. *Proc. of Logic Programming and Non-Monotonic Reasoning, LPNMR'01, Springer-Verlag, LNAI*, 2173:351–364, 2001.
19. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. A procedural semantics for multi-adjoint logic programing. *Progress in Artificial Intelligence, EPIA'01, Springer-Verlag, LNAI*, 2258(1):290–297, 2001.
20. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146:43–62, 2004.
21. P.J. Morcillo and G. Moreno. Programming with Fuzzy Logic Rules by using the FLOPER Tool. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML 2008, Orlando, FL, USA, October 30-31*, pages 119–126. Springer Verlag, LNCS 3521, 2008.
22. P.J. Morcillo and G. Moreno. Modeling interpretive steps in fuzzy logic computations. In Vito Di Gesù, Sankar K. Pal, and Alfredo Petrosino, editors, *Proc. of the 8th International Workshop on Fuzzy Logic and Applications, WILF 2009. Palermo, Italy, June 9-12*, pages 44–51. Springer Verlag, LNAI 5571, 2009.
23. P.J. Morcillo and G. Moreno. On cost estimations for executing fuzzy logic programs. In Hamid R. Arabnia, David de la Fuente, and José Angel Olivas, editors, *Proceedings of the 11th International Conference on Artificial Intelligence, ICAI 2009, July 13-16, 2009, Las Vegas (Nevada), USA*, pages 217–223. CSREA Press, 2009.
24. P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. Modeling interpretive steps into the FLOPER environment. In *Proceedings of the 12th International Conference on Artificial Intelligence, ICAI 2010, July 12-15, 2010, Las Vegas (Nevada), USA*. CSREA Press (accepted for publication), 2010.
25. U. Straccia. Query answering in normal logic programs under uncertainty. In *8th European Conferences on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-05)*, number 3571 in Lecture Notes in Computer Science, pages 687–700, Barcelona, Spain, 2005. Springer Verlag.
26. 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.
27. P. Vojtáš. Fuzzy Logic Programming. *Fuzzy Sets and Systems*, 124(1):361–370, 2001.