Declarative Traces Into Fuzzy Computed Answers [⋆]

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

University of Castilla-La Mancha
Faculty of Computer Science Engineering
2071, Albacete, Spain
`{pmorcillo,cvazquez}@dsi.uclm.es`
`{Gines.Moreno,Jaime.Penabad}@uclm.es`

Abstract. Fuzzy logic programming is a growing declarative paradigm aiming to integrate fuzzy logic into logic programming. In this setting, the so-called *Multi-Adjoint Logic Programming* approach, MALP in brief, represents an extremely flexible fuzzy language for which we are developing the FLOPER tool (*Fuzzy LOgic Programming Environment for Research*). Currently, the platform is useful for compiling (to standard Prolog code), executing and debugging fuzzy programs in a safe way and it is ready for being extended in the near future with powerful transformation and optimization techniques designed in our research group in the recent past. In this paper, we focus in a nice property of the system regarding its ability for easily collecting declarative traces at execution time, without modifying the underlying procedural principle. The clever point is the use of lattices modeling truth degrees (beyond $\{true, false\}$) enriched with constructs for directly *visualizing* on fuzzy computed answers not only the sequence of program rules exploited when reaching solutions, but also the set of evaluated fuzzy connectives together with the sequence of primitive (arithmetic) operators they call, thus giving a detailed description of their computational complexities.

Key words: Fuzzy Logic Programming, Declarative Traces, Lattices

1 Introduction

Logic Programming (LP) [15] 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 treat explicitly with uncertainty and approximated reasoning. 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.

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

During the last decades, several fuzzy logic programming systems have been developed, such as [2, 3, 5, 14, 12, 27], the QLP scheme of [24] and the many-valued logic programming language of [25, 26], where the classical inference 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 also the case of *multi-adjoint logic programming* approach MALP [18, 16, 17], 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 [9, 20, 23]), 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 the benefits of introducing different notions of multi-adjoint lattices for managing truth degrees even in a single FLOPER's work-session without changing a given MALP program and goal. In particular, we are especially interested now in showing the collateral effect of these actions regarding debugging capabilities (i.e., the generation of declarative traces inside fuzzy computed answers).

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, we propose too a sophisticated kind of lattices capable for taking into account details on declarative traces, such as the sequence of computations (regarding program rules, fuzzy connectives and primitive operators) needed for evaluating a given goal. 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 [18, 16, 17] 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* [18]. 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 [13]):*

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 $\mathcal{P}$.
3) $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/\bot]); \sigma \rangle$,
   if there is no rule in $\mathcal{P}$ whose head unifies with A.

Note that 3^{th} case is introduced to cope with (possible) unsuccessful admissible derivations (this kind of step is useful when evaluating, for instance, an expression like "$\vee(p, 0.8)$", which returns a value different from 0 even when there is no program rule defining p). As usual, rules are taken renamed apart. We shall use the symbols $\rightarrow_{AS1}$, $\rightarrow_{AS2}$ and $\rightarrow_{AS3}$ 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*

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:

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

Interpretive derivation:

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

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

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 $@_{\mathtt{aver}}$ must obviously has the form: $@_{\mathtt{aver}}(x_1, x_2) \triangleq (x_1 + x_2)/2$. Now, we can generate the admissible 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 [9]:

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

In [22] we describe a very easy way to model truth-degree lattices for being included into the FLOPER tool. All relevant components of each lattice are 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*

```
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).
agr_aver2(X,Y,Z)   :- or_godel(X,Y,Z1),or_luka(X,Y,Z2),agr_aver(Z1,Z2,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").

"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).`

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

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

.... being A and B_i typical atoms[1], successfully mimics the behaviour of classical Prolog programs where clauses accomplish with the shape "$A \quad :- \quad B_1, \dots, B_n$". 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

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

(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).

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* intuistionistic 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 $\&_{\texttt{G}}(x, y) \triangleq inf(x, y)$, where it is important to note that we must replace the use of "*min*" by "*inf*" in the connective definition.

$$\begin{tikzcd}
 & \top & \\
\alpha \arrow[ur] & & \beta \arrow[ul] \\
 & \bot \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

```
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 Declarative Traces into f.c.a.'s using FLOPER

As detailed in [1,19,22,23], 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 "$\_TV_i$", 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 in Figure 1, 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, in [22] we explain that FLOPER has been recently equipped with new options, called "`lat`" and "`show`", for allowing the possibility of respectively changing and displaying the multi-adjoint lattice associated to a given program. 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 a new version which computes the average between the results achieved after applying to both elements the disjunctions operators described by Gödel and Łukasiewicz, that is: $@_{\texttt{aver}}(x_1, x_2) \triangleq (\vee_{\texttt{G}}(x_1, x_2) + \vee_{\texttt{L}}(x_1, x_2)) * 0.5$. 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).
```

and now, by selecting again the "`run`" option (without changing the program and goal), the system would display the new solution: $[\texttt{Truth\_degree} = 0.72, \texttt{X} = \texttt{a}]$.

Let us consider now the so called *domain of weight values* $\mathcal{W}$ used in the QLP (*Qualified Logic Programming*) framework of [24], whose elements are intended to represent proof costs, measured as the weighted depth of proof trees (although close to MALP, the QLP scheme allows a lesser repertoire of connectives in the body of program rules). In essence, $\mathcal{W}$ can be seen as lattice $\langle \mathbb{R} \cup \infty, \geq \rangle$, where $\geq$ is the reverse of the usual numerical ordering (with $\infty \geq d$ for any $d \in \mathbb{R}$) and thus, the bottom elements is ∞ and the top element is 0 (and not vice versa).

By using again the "`lat`" option of FLOPER, we can associate this lattice $\mathcal{W}$ to the program seen before after changing the "weights" of each program rule to 1 (the underlying idea is that "the use of each program rule in a derivation implies the application of one admissible step"). Moreover, since in this lattice the arithmetic operation "+" plays the role of a conjunction (*t-norm*) connective, we assume the definitions of the set of connectives appearing in the program *mapped* to "+" (i.e. $\&_{\texttt{P}}(x, y) \triangleq x + y$, $\&_{\texttt{G}}(x, y) \triangleq x + y$ and $@_{\texttt{aver}}(x, y) \triangleq x + y$). Now, for goal "$p(X)$" we could generate an admissible derivation similar to the one seen in Figure 1, but ending now with $\langle \&_{\texttt{P}}(1, \&_{\texttt{G}}(1, @_{\texttt{aver}}(1, 1))); \{X/a\} \rangle$ And since: $\&_{\texttt{P}}(1, \&_{\texttt{G}}(1, @_{\texttt{aver}}(1, 1))) = +(1, +(1, +(1, 1))) = 4$, the final fuzzy computed answer or f.c.a. $\langle 4; \{X/a\} \rangle$ indicates that goal "$p(X)$" holds when X is a, as proved after applying 4 admissible steps, as wanted.

Moreover, we can also conceive a more powerful lattice expressed as the *cartesian product* of the one seen in Figure 2 (real numbers in the interval $[0, 1]$) and $\mathcal{W}$. Now, each element has two components, coping with truth degrees and cost measures. 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,Cost_Number_Steps)`". Moreover, the clauses defining some predicates required for managing them are:

```
member(info(X,Y)) :- number(X), 0=<X, X=<1, number(Y), Y=<0.
leq(info(X1,Y1),info(X2,Y2)) :- X1=<X2, Y1>=Y2.       top(info(1,0)).
and_godel(info(X1,Y1),info(X2,Y2),info(X3,Y3)) :-  pri_min(X1,X2,X3),
                                                   pri_add(Y1,Y2,Y3).
```

Finally, if the weights assigned to the rules of our running example be "`info(0.9,1)`" for $\mathcal{R}_1$, "`info(0.8,1)`" for $\mathcal{R}_2$, "`info(0.7,1)`" for $\mathcal{R}_3$ and "`info(0.5,1)`" for $\mathcal{R}_4$, then, for goal "$p(X)$" we would obtain the desired f.c.a. $\langle \texttt{info}(0.54, 4); \{X/a\} \rangle$ with the obvious meaning that we need 4 admissible steps to prove that the query is true at a 56 % degree when X is a".

```
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).
or_godel(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-
                  pri_max(X1,Y1,Z1,DatMAX),pri_app(X2,Y2,Dat1),
                  pri_app(Dat1,'|GODEL.',Dat2),pri_app(Dat2,DatMAX,Z2).
or_luka(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-
                     pri_add(X1,Y1,U1,DatADD),pri_min(U1,1,Z1,DatMIN),
                     pri_app(X2,Y2,Dat1),pri_app(Dat1,'|LUKA.',Dat2),
                     pri_app(Dat2,DatADD,Dat3),pri_app(Dat3,DatMIN,Z2).
agr_aver(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-
                     pri_add(X1,Y1,Aux,DatADD),pri_div(Aux,2,Z1,DatDIV),
                     pri_app(X2,Y2,Dat1),pri_app(Dat1,'@AVER.',Dat2),
                     pri_app(Dat2,DatADD,Dat3),pri_app(Dat3,DatDIV,Z2).
agr_aver2(info(X1,X2),info(Y1,Y2),info(Z1,Z2)):-
                     or_godel(info(X1,''),info(Y1,''),Za),
                     or_luka(info(X1,''),info(Y1,''),Zb),
                     agr_aver(Za,Zb,info(Z1,Dat3)),pri_app(X2,Y2,Dat1),
                     pri_app(Dat1,'@AVER2.',Dat2),pri_app(Dat2,Dat3,Z2).

pri_add(X,Y,Z,'#ADD.') :- Z is X+Y.   pri_sub(X,Y,Z,'#SUB.') :- Z is X-Y.
pri_prod(X,Y,Z,'#PROD.'):-Z is X * Y. pri_div(X,Y,Z,'#DIV.') :- Z is X/Y.
pri_min(X,Y,Z,'#MIN.') :- (X=<Y,Z=X;X>Y,Z=Y).
pri_max(X,Y,Z,'#MAX.') :- (X=<Y,Z=Y;X>Y,Z=X).
pri_app(X,Y,Z) :-     name(X,L1),name(Y,L2),append(L1,L2,L3),name(Z,L3).
append([],X,X).                append([X|Xs],Y,[X|Zs]):-append(Xs,Y,Zs).
```

Fig. 3. Multi-adjoint lattice modeling truth degrees with labels.

One step beyond, in what follows we are going to design a much more complex lattice to cope with declarative traces. Its elements must have two components, taking into account truth degrees and "labels" collecting information about the program rules, fuzzy connectives and primitive operators used when executing programs. In order to be loaded into FLOPER, we need to define again the new lattice as a Prolog program, whose elements will be expressed now as data terms of the form "`info(Fuzzy_Truth_Degree, Label)`" as shown in Figure 3 (note that the complex version of the *average connective* is called here `agr_aver2` and invokes the simple version `agr_aver`).

Here, we see 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 set of rules of our example (where we use the complex version of average, i.e., `@aver2` in the first rule) must have the form:

```
p(X) <prod &godel(q(X),@aver2(r(X),s(X))) with info(0.9,'RULE1.').
q(a) with   info(0.8,'RULE2.').
r(X) with   info(0.7,'RULE3.').
s(X) with   info(0.5,'RULE4.').
```

Now, the reader can easily tests that, after executing goal `p(X)`, we obtain the desired fuzzy computed answers which includes the desired declarative trace regarding program-rules/connective-calls/primitive-operators evaluated till finding the final solution:

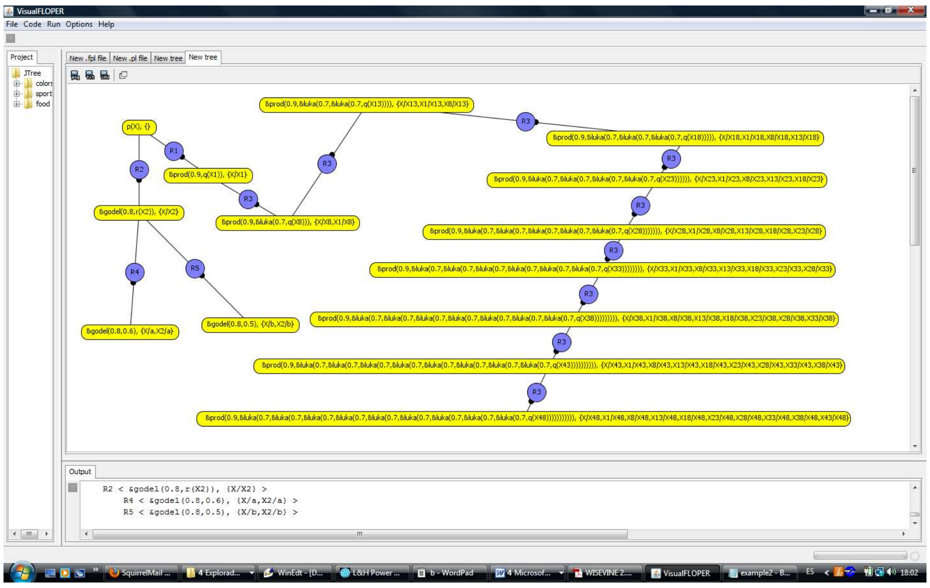
```
>> run.

[Truth_degree=info(0.72,  RULE1.RULE2.RULE3.RULE4.
                          @AVER2.|GODEL.#MAX.|LUKA.
                          #ADD.#MIN.@AVER.#ADD.#DIV.
                          &GODEL.#MIN.&PROD.#PROD.),    X=a]
```

In this fuzzy computed answer we obtain both the truth value (0.72) and substitution ($X = a$) associated to our goal, but also the sequence of program rules exploited when applying admissible steps as well as the proper fuzzy connectives evaluated during the interpretive phase, also detailing the set of primitive operators (of the form #*label*) they call.

Strongly related with this, in [20] we proposed a variant of the original notion of interpretive step (see Definition 3) which was able to distinguish calls to fuzzy connectives (conjunctions, disjunctions and aggregations) and computations devoted to the evaluation of primitive operators, thus providing cost measures about the complexity of connectives. Such new notion, called *small interpretive step*, has been recently implemented into FLOPER, as described in [23], in order to generate "evaluation trees" like the one shown in Figure 4. However, compared with our present approach where we don't need any additional modification of the underlying execution machinery, the implementation of [23] required strong changes in the core of the systems, including a new representation of the fuzzy code much more involved than the one based in the compilation to Prolog code described at the beginning of this section.

The research line on cost measures mentioned above was motivated after evidencing in our fuzzy fold/unfold framework described in [4, 8] that it is possible to improve the "shape" of a set of program rules but with the "risk" of automatically generating a set of artificial connectives (see the definition of the *aggregation transformation* described in [4]) which necessarily invoke other connectives, thus producing nested definitions of aggregators. For this reason, it is very important to "calibrate" the complexities of these new connectives (i.e., to visualize the number of direct/indirect calls they perform to other connectives and/or primitive operators) in order to detect if the whole transformation process really returns improved sets of program rules and connective definitions. In this sense, the present work can be seen as a first stage to achieve this goal.


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}
R3
&godel(0.8,r(X2)), {X/X2}
&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

```
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.

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 ([9, 4, 8, 10, 11, 6, 7, 20, 21]), has motivated our interest for putting in practice all our developments around the design of the FLOPER environment [19, 23, 22]. 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. The main purpose of this work has been the illustration of an interesting kind of lattices where truth-degrees are accompanied with labels, having the ability of augmenting fuzzy computed answers with declarative traces (i.e., by listing the sequence of program rules, connective calls and primitive operators used for finding solutions) without requiring additional cost.

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 ex-

[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.

tending 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 plan to implementing all the manipulation tasks developed in our group on fold/unfold transformations [4, 8], partial evaluation [11] and thresholded tabulation [7]. Moreover, we continue working in the development of our first real-world application (written in MALP and compiled with FLOPER) which is devoted to manipulate XML documents via fuzzy extension of the popular XPath language (please, visit url `http://www.dsi.uclm.es/investigacion/dect/FuzzyXPath.htm`).

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. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. P. Julián, G. Moreno, and J. Penabad. On Fuzzy Unfolding. A Multi-adjoint Approach. *Fuzzy Sets and Systems*, 154:16–33, 2005.
9. 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.
10. 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.
11. 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.
12. M. Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12:335–367, 1992.

13. 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.
14. D. Li and D. Liu. *A fuzzy Prolog database system*. John Wiley & Sons, Inc., 1990.
15. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.
16. 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.
17. 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.
18. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146:43–62, 2004.
19. 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. International Symposium on Rule Representation, Interchange and Reasoning on the Web, RuleML 2008, Orlando, FL, USA, October 30-31*, pages 119–126. Springer Verlag, LNCS 3521, 2008.
20. 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.
21. 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, Las Vegas (Nevada), USA*, pages 217–223. CSREA Press, 2009.
22. 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 the 4th. International Symposium on Rule Interchange and Applications, RuleML 2010, Washington, USA, October 21–23*, pages 20–34. Springer Verlag, LNCS 6403, 2010.
23. P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. Modeling interpretive steps into the FLOPER environment. In H.R. Arabnia et al., editor, *Proceedings of the 12th International Conference on Artificial Intelligence, ICAI 2010, July 12-15, Las Vegas (Nevada), USA*, pages 16–22. CSREA Press, 2010.
24. 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.
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.