Multi-Adjoint Lattices for Manipulating Truth-Degrees into the FLOPER System

Pedro J. Morcillo[a,2] Ginés Moreno,[a,2] Jaime Penabad[b,3]
Carlos Vázquez[a,2]

[a] *Department of Computing Systems, U. Castilla-La Mancha, Albacete (02071), Spain*

[b] *Department of Mathematics, U. Castilla-La Mancha, Albacete (02071), Spain*

Abstract

FLOPER is a "Fuzzy LOgic Programming Environment for Research" developed in our research group which currently offers running/debugging/tracing capabilities for managing programs belonging to the so-called *multi-adjoint logic approach*. In this recent and flexible framework, typical Prolog clauses have been extended with fuzzy features, including a wide repertoire of connectives for manipulating truth degrees beyond the simple case of $\{true, false\}$. Multi-adjoint lattices capture the mathematical foundations of this enrichment. In this paper, we report our last developments performed into the FLOPER tool, which are devoted to put in practice the management of such structures in an easy, quite comprehensible way.

Keywords: Fuzzy Logic Programming, Truth-Degrees, Multi-adjoint Lattices, Software Tools

1 Introduction

Research in the fields of *Declarative Programming* and *Fuzzy Logic* have traditionally provided programming languages and techniques with important applications in the fields of AI, soft-computing, and so on. In particular, *Logic Programming* [1] has been widely used for problem solving and knowledge representation in the past without incorporating techniques or constructs to explicitly treat 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 agglutinate the efforts for introducing fuzzy logic into logic programming. During the last decades, several fuzzy logic programming systems have been developed [2,3,4,5,6], where the classical inference mechanism of SLD–Resolution is replaced with a fuzzy variant which is able to handle partial truth and uncertainty.

[1] Work supported by the EU, under 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.

[2] Email: {pmorcillo,gmoreno,cvazquez}@dsi.uclm.es

[3] Email: Jaime.Penabad@uclm.es

This is the case of *multi-adjoint logic programming* [7] one of the most powerful and promising approaches 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 an state composed by a computed substitution together with an expression where all atoms have been exploited. Next, during the so called interpretive phase, (see [8]), 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 [9,10] (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/precision features.

The structure of the paper is as follows. Section 2 summarizes the main features of multi-adjoint logic programming. 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 [7] for a complete formulation of this framework). In what follows, we will use the abbreviation MALP for referencing programs belonging to this setting. 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* [7]. More exactly, in this setting the following items must be satisfied:

- $\langle L, \preceq\rangle$ is a bounded lattice, i.e. it has bottom/top elements, denoted by $\bot/\top$.
- 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*:

$\&_{\mathtt{P}}(x,y) \triangleq x * y$	$\leftarrow_{\mathtt{P}}(x,y) \triangleq \min(1, x/y)$	*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 MALP 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, we sometimes refer a tuple $\langle \mathcal{R}; \alpha\rangle$ as a "rule".

The procedural semantics of the multi–adjoint logic language $\mathcal{L}$ can be thought 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 2.1 (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* $\overset{AS}{\rightsquigarrow} \subseteq (\mathcal{E} \times \mathcal{E})$ *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):*

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

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

As usual, rules are taken renamed apart. We shall use the symbols $\overset{\text{AS1}}{\rightsquigarrow}$ and $\overset{\text{AS2}}{\rightsquigarrow}$ 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 $\overset{\text{AS}}{\rightsquigarrow}$ symbol.

Definition 2.2 *Let $\mathcal{P}$ be a program, $\mathcal{Q}$ a goal and "id" the empty substitution. An* admissible derivation *is a sequence* $\langle \mathcal{Q}; id \rangle \overset{AS}{\rightsquigarrow} \ldots \overset{AS}{\rightsquigarrow} \langle \mathcal{Q}'; \theta \rangle$. *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 2.3 *Let $\mathcal{P}$ be the following MALP program:*

$\mathcal{R}_1 : \; p(X) \leftarrow_{\mathtt{P}} \; \&_{\mathtt{G}}(q(X), @_{\mathtt{aver}}(r(X), s(X)))$ *with* 0.9

$\mathcal{R}_2 : \; q(a) \; \leftarrow$ *with* 0.8

$\mathcal{R}_3 : \; r(X) \leftarrow$ *with* 0.7

$\mathcal{R}_4 : \; s(X) \leftarrow$ *with* 0.5

where the equation defining the average aggregator must obviously have the form: $@_{\mathtt{aver}}(x_1, x_2) \triangleq (x_1 + x_2)/2$. *Now, we can generate the next admissible derivation (we underline the selected atom in each step):*

$\langle \underline{p(X)};\ id \rangle \overset{AS1}{\rightsquigarrow} \mathcal{R}_1$

$\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\underline{q(X_1)}, @_{\mathtt{aver}}(r(X1), s(X1)))); \{X/X_1\} \rangle \overset{AS2}{\rightsquigarrow} \mathcal{R}_2$

$\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(\underline{r(a)}, s(a)))); \{X/a, X_1/a\} \rangle \overset{AS2}{\rightsquigarrow} \mathcal{R}_3$

$\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(0.7, \underline{s(a)}))); \{X/a, X_1/a, X_2/a\} \rangle \overset{AS2}{\rightsquigarrow} \mathcal{R}_4$

$\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(0.7, 0.5))); \{X/a, X_1/a, X_2/a, X_3/a\} \rangle$

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 referenciates 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 [8]:

Definition 2.4 (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* $\overset{IS}{\rightsquigarrow} \subseteq (\mathcal{E} \times \mathcal{E})$ *is defined as the least one satisfying:* $\langle \mathcal{Q}[@(r_1, \ldots, r_n)]; \sigma \rangle \overset{IS}{\rightsquigarrow} \langle \mathcal{Q}[@(r_1, \ldots, r_n)/r_{n+1}]; \sigma \rangle$.

Definition 2.5 *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 $\langle \mathcal{Q}; \sigma \rangle \overset{IS}{\rightsquigarrow} \ldots \overset{IS}{\rightsquigarrow} \langle \mathcal{Q}'; \sigma \rangle$. 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 2.6 *If we complete the previous derivation of Example 2.3 by applying 3 interpretive steps in order to obtain the final f.c.a. $\langle 0.54; \{X/a\} \rangle$, we generate the following interpretive derivation D_1:*

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

3 Truth-Degrees and Multi-adjoint Lattices in Practice

We have recently conceived a very easy way to model truth degrees 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

```
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. 1. Multi-adjoint lattice modeling truth degrees in the real interval [0,1] (file "num.pl").

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).`

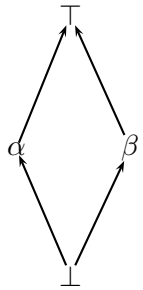
The reader can easily check that the use of lattice "bool.pl" when working with MALP programs whose rules have the form "$A \leftarrow_{bool} \&_{bool}(B_1,\ldots,B_n)$ *with* 1", being A and B_i typical atoms [4], successfully mimics the behaviour of classical Prolog programs where clauses accomplish with the shape "$A :- B_1,\ldots,B_n$". 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 Figure 1 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 [5] 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 described before (i.e., the *Łukasiewicz*, *Gödel* and *product* logics), as well as an useful description for the hybrid aggregator *average*. Note also we have included definitions for auxiliary predicates, whose names always begin with the prefix "`pri_`". All 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).

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

[5] Note that this condition disables the implementation of the consulting predicate "`members/1`".

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




```
members([bottom,alpha,beta,top]).
leq(bottom,X). leq(alpha,alpha).  leq(alpha,top).
leq(beta,beta).  leq(beta,top).  leq(top,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,bot).
```

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 to 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 "`member(s)/1`") are incomparable then, any call to both "`?- 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 [9], 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

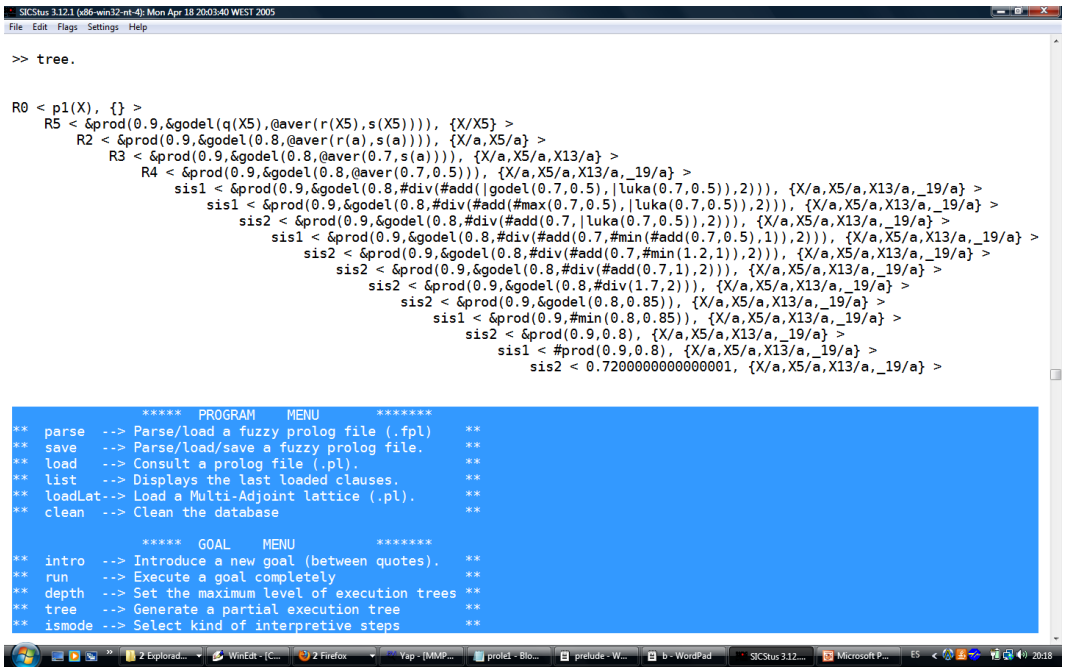


Fig. 2. Example of a work session with FLOPER showing "Small Interpretive Steps" and program/goal menus

includes options for loading, parsing, listing and saving fuzzy programs, as well as for executing fuzzy goals (see Figure 2). 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, becomes in the pure Prolog fact "`q(a,0.8)`" while a fuzzy goal like "`p(X)`", is translated into the 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 [`Truth_degree` $= 0.54$, `X` $=$ `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 of being working with a purely fuzzy logic programming tool.

On the other hand, as showed in the down-middle, dark part of Figure 2, 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 aggrega-

tor. 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 of Gödel and Łukasiewicz, that is, $@_{\texttt{aver}}(x_1, x_2) \triangleq (\vee_{\texttt{G}}(x_1, x_2) + \vee_{\texttt{L}}(x_1, x_2))/2$. A 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_div(Z3,2,Z).
```

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 or partial evaluation we have described in [11,12,13]. To this end, in [9] we 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. For instance, after parsing the second rule of our program, we obtain the following expression which is *asserted* into the interpreter's database as a Prolog fact and collects in detail all relevant components of rules (composition of atoms in heads/bodies, attached weights, etc):

```
rule(2,head(atom(pred(q,1),[con(a)])),impl(empty),body(empty),td(0.8)).
```

Looking again to the darked part of Figure 2, 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 [9] show, in contrast with the traces (based on finite, non-failed, admissible derivations) that the "`tree`" option displays. As the top-middle of Figure 2 illustrates, the system displays states on different lines, appropriately indented to distinguish the proper relationship -parent/child/grandchild...- among nodes.

Strongly related with these last options, the "`ismode`" choice decides among three levels of detail when visualizing the interpretive phase performed during the generation of "unfolding trees": whereas "`Large`" means to obtain the final result in one go, "`Medium`" implements the notion of "interpretive step" according Definition 2.4 and "`Small`" allows to visualize in detail both the direct/indirect calls to connective definitions and primitive operators performed along the whole interpretive phase [10]. The reader can observe at the beginning of Figure 2, the aspect offered by FLOPER when visualizing the behaviour of our running example (using our last definition of the average aggregator) once we have choosen the last option just commented before. In particular, observe that program rules applied on admissible steps always precede the corresponding state (FLOPER labels the root goal with the "virtual" program rule `R0`)), whereas the interpretive phase applies "small interpretive steps" of kind $\overset{\text{SIS1}}{\leadsto}$ or $\overset{\text{SIS2}}{\leadsto}$ (according [10]) when expanding connective

definitions or evaluating primitive operators, respectively, on states (advise in the figure that each primitive operators is always labeled by prefix "#"). These facts justify why in our Prolog-based implementation of lattices, clauses defining connective predicates only perform calls to predicates of the form "`and_*`", "`or_*`", "`agr_*`" (useful for identifying further $\overset{SIS1}{\leadsto}$ steps) or "`pri_*`" (associated to $\overset{SIS2}{\leadsto}$).

5 Conclusions and Future Work

The experience acquired in our research group regarding the design of techniques/methods based on fuzzy logic in close relationship with the so-called multi-adjoint logic programming approach ([8,11,12,13,14]), has motivated our interest for putting in practice all our developments around the design of the FLOPER environment [9,10]. 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 truth degrees also admit nice characterizations into Prolog. Nowadays we are extending FLOPER with testing techniques for automatically checking that lattices modeled in this way verify the requirements of our fuzzy setting (specially, the *adjoint property*).

References

[1] J. Lloyd, *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987, second edition.

[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*, vol. 144, no. 1, pp. 127–150, 2004.

[4] M. Ishizuka and N. Kanai, "Prolog-ELF Incorporating Fuzzy Logic," in *Proceedings of the 9th Int. Joint Conference on Artificial Intelligence, IJCAI'85*, A. K. Joshi, Ed. Morgan Kaufmann, 1985, pp. 701–703.

[5] M. Kifer and V. Subrahmanian, "Theory of generalized annotated logic programming and its applications." *Journal of Logic Programming*, vol. 12, pp. 335–367, 1992.

[6] P. Vojtáš, "Fuzzy Logic Programming," *Fuzzy Sets and Systems*, vol. 124, no. 1, pp. 361–370, 2001.

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

[8] P. Julián, G. Moreno, and J. Penabad, "Operational/Interpretive Unfolding of Multi-adjoint Logic Programs," *Journal of Universal Computer Science*, vol. 12, no. 11, pp. 1679–1699, 2006.

[9] P. Morcillo and G. Moreno, "Programming with Fuzzy Logic Rules by using the FLOPER Tool," in *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML 2008, Orlando, FL, USA, October 30-31*, N. Bassiliades, G. Governatori, and A. Paschke, Eds. Springer Verlag, LNCS 3521, 2008, pp. 119–126.

[10] 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.

[11] J. Guerrero and G. Moreno, "Optimizing fuzzy logic programs by unfolding, aggregation and folding," *Electronic Notes in Theoretical Computer Science*, vol. 219, pp. 19–34, 2008.

[12] P. Julián, G. Moreno, and J. Penabad, "On Fuzzy Unfolding. A Multi-adjoint Approach," *Fuzzy Sets and Systems*, vol. 154, pp. 16–33, 2005.

[13] ——, "An Improved Reductant Calculus using Fuzzy Partial Evaluation Techniques," *Fuzzy Sets and Systems*, vol. 160, pp. 162–181, 2009, doi: 10.1016/j.fss.2008.05.006.

[14] 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)*, vol. 249, pp. 125–141, 2010.