Journal of Software Engineering and Applications, 2014, *, **
Published Online**** 2014 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2014.*****

Fuzzy Logic Programming in Action with $\mathcal{FLOPER}$

Special Issue on "Programming Languages"

Ginés Moreno, Carlos Vázquez

Affiliation: Department of Computing Systems, University of Castilla-La Mancha, Albacete, Spain
Email: {Gines.Moreno, Carlos.Vazquez}@uclm.es

Abstract

During the last years we have developed the $\mathcal{FLOPER}$ platform for providing a practical support to the so-called *Multi-Adjoint Logic Programming* approach (MALP in brief), which represents an extremely flexible framework into the *Fuzzy Logic Programming* arena. Nowadays, $\mathcal{FLOPER}$ is useful for compiling (to standard PROLOG code), executing and debugging (by drawing *execution trees*) MALP programs, and it is ready for being extended in the near future with powerful transformation and optimization techniques designed in our research group during the recent past. Our last update consists in the integration of a graphical interface for a comfortable interaction with the system which allows, among other capabilities, the use of *projects* for packing *scripts* and auxiliary definitions of fuzzy sets/connectives, together with fuzzy programs and their associated lattices modeling truth-degrees beyond the simpler crisp case $\{true; false\}$.

Keywords

Fuzzy Logic Programming; Language Design and Implementation; New Programming Concepts and Paradigms; Software tools; Cost Measures

1 Introduction

Logic programming [25] has been widely used for problem solving and knowledge representation in the past. Nevertheless, traditional *pure logic programming* languages (i.e., PROLOG) do not incorporate techniques or constructs to explicitly deal with uncertainty and approximated reasoning. To overcome this situation, during the last decades several *fuzzy logic programming* systems have been developed where the classical inference mechanism of SLD-resolution is replaced with a fuzzy variant able to handle partial truth and to reason with uncertainty, thus promoting the development of real-world applications in the fields of artificial/computational intelligence, soft-computing, semantic web, etc. Most of these systems implement the fuzzy resolution principle introduced by Lee in [22], such as languages PROLOG-Elf [14], F-PROLOG [24], Fril [4], (S-)QLP [44, 7], RFuzzy [39] and MALP [30], being this last approach our target goal.

Since uncertainty and vagueness are constant elements present in most human thinking activities, fuzzy logic programming seems to be a computing paradigm with a level of expressiveness very close to human reasoning. Daily, we frequently deal with fuzzy predicates (not *boolean, crisp* ones). For instance, a person can be quite young or not very young: the frontier between "absolutely young" and "not young at all" is not sharp, but *indefinite, fuzzy.* If "John" is 18 year old, we can say that he is "young" at a 95% truth degree. If "Mary" is 70 years old, she is young at a lower truth degree. Then, the obvious question is how do we model the concept of "*truth degree*". In our system, we work with the so-called *multi-adjoint lattices* to model them by providing a flexible, wide enough definition of such notion.

Informally speaking, in the multi-adjoint logic 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, i.e., a set of atoms linked with connectives called *aggregators*. A state is a pair $\langle \mathcal{Q}, \sigma \rangle$ where $\mathcal{Q}$ is a goal and σ a substitution (initially, the identity substitution). States are evaluated in two separate computational phases. During the *operational* one, *admissible steps* (a generalization of the classical *modus ponens* inference rule) are systematically applied by a backward reasoning procedure in a similar way to classical resolution steps in pure logic programming, thus returning a computed substitution together with an expression where all atoms have been exploited. This last expression is then interpreted under a given lattice during what we call the *interpretive* phase, hence returning a pair $\langle truth\_degree; substitution \rangle$ which is the fuzzy counterpart of the classical notion of computed answer traditionally used in LP.

The main goal of this report is the detailed description of the $\mathcal{FLOPER}$ system which is available from `http://dectau.uclm.es/floper/`. Nowadays, the tool provides facilities for executing as well as for debugging (by generating declarative traces) such kind of fuzzy programs, thus fulfilling the gap we have detected in the area. In order to explain the tool, we have structured this paper as follows: in Section 2 we present the essence of MALP, including syntax, procedural semantics, and some interesting computational cost measures defined for this programming style; the core of this paper is represented by Section 3, which is dedicated to explain the main capabilities of the $\mathcal{FLOPER}$ system such as running/debugging MALP programs, managing lattices and dealing with additional fuzzy concepts; before finalizing in Section 5, we detail in Section 4 how the use of sophisticated multi-adjoint lattices are very useful for easily coding flexible real-world applications and obtaining low-cost traces at execution time.

2 Multi-Adjoint Logic Programming

This section summarizes the main features of multi-adjoint logic programming (for a complete formulation of this framework, see [28, 29, 30, 20]). In what follows, we use 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 ($|_1, |_2, \ldots$), and hybrid operators (usually denoted by $@_1, @_2, \ldots$), all of them are grouped under the name of "aggregators".

Aggregation operators (or aggregators) 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* 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* [30]) formally defined as follows.

Definition 2.1 (Multi-Adjoint Lattice) *Let $(L, \leq)$ be a lattice. A* multi-adjoint lattice *is a tuple $(L, \leq, \leftarrow_1, \&_1, \dots, \leftarrow_n, \&_n)$ such that:*

1. *$\langle L, \preceq \rangle$ is a bounded lattice, i.e. it has bottom and top elements, denoted by $\bot$ and $\top$, respectively.*
2. *$\langle L, \preceq \rangle$ is a complete lattice, i.e. for all subset $X \subset L$, there are $inf(X)$ and $sup(X)$.*
3. *$\top \&_i v = v \&_i \top = v$, $\forall v \in L, i = 1, \dots, n$.*
4. *Each operation $\&_i$ is increasing in both arguments.*
5. *Each operation $\leftarrow_i$ is increasing in the first argument and decreasing in the second one.*
6. *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:*

$$x \preceq (y \leftarrow_i z) \quad \textit{if and only if} \quad (x \,\&_i\, z) \preceq y$$

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 defined in 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 $@(x_1, \dots, x_n) \triangleq E$, where E is a L-expression not containing variable symbols apart from $x_1, \dots, x_n$. For instance, consider the following classical set of adjoint pairs (conjunctions and implications) in $\langle [0, 1], \leq \rangle$, where labels `L`, `G` and `P` mean respectively *Lukasiewicz logic*, *Gödel intuitionistic logic* and *product logic* (which different capabilities for modeling *pessimist*, *optimist* and *realistic scenarios*, respectively):

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

Moreover, the three disjunctions associated to the previous fuzzy logics are defined as follows: $|_{\mathtt{P}}(x,y) \triangleq x + y - x * y$, $|_{\mathtt{G}}(x,y) \triangleq max\{x,y\}$, and $|_{\mathtt{L}}(x,y) \triangleq min\{x+y,1\}$.

At this point, we wish to make a mention to the notion of *qualification domain* used in the QLP (*Qualified Logic Programming*) scheme described in [44], which plays a role in that framework similar to multi-adjoint lattices in MALP. A qualification domain is a structure $\langle D, \sqsubseteq, \bot, \top, \circ \rangle$, such that $\langle D, \sqsubseteq, \bot, \top \rangle$ is a lattice with top ($\top$) and bottom ($\bot$) elements, a partial ordering $\sqsubseteq$, and where the so-called *attenuation operation* "$\circ$" is a conjunction. Now, given two elements $d, e \in D$, $d \sqcap e$ means for the *greatest lower bound* of d and e, whereas $d \sqcup e$ represents its *least upper bound*. We also write $d \sqsubset e$ as abbreviation of $d \sqsubseteq e \& d \neq e$. The attenuation operator $\circ$ satisfies the following constraints.

1. $\circ$ is associative, commutative and monotonic w.r.t. $\sqsubseteq$.
2. $\forall d \in D : d \circ \top = d$.
3. $\forall d \in D : d \circ \bot = \bot$.
4. $\forall d, e \in D / \{\bot, \top\} : d \circ e \sqsubset e$.
5. $\forall d, e_1, e_2 \in D : d \circ (e_1 \sqcap e_2) = d \circ e_1 \sqcap d \circ e_2$.

Note that the required properties in QLP and MALP are rather close, but instead of the last *distributive law* just pointed out in claim 5, in our setting we use the adjoint property (claim 6 in Definition 2.1). Translations between both worlds can be frequently performed, as occurs for instance with the simple boolean qualification domain $\mathcal{B} = ([0, 1], \leq, 0, 1, \&)$, whose shape as a multi-adjoint lattice looks like

$\langle[0,1],\leq,\leftarrow,\&\rangle$, where & is de boolean conjunction and $\leftarrow$ its adjoint implication (i.e., the usual bi-valuated logic implication). Something similar occurs with the qualification domain of *Van Emden's uncertainty values* used in QLP, $\mathcal{U} = ([0,1],\leq,0,1,\times)$, which is equivalent to the multi-adjoint lattice (in which most of our examples are interpreted) described above, as well as with the so-called *weights domain* $\mathcal{W} = (\mathbb{R}\cup\infty,\geq,\infty,0,+)$, whose detailed explanation is delayed to Section 4 (where we will present powerful extensions and applications related with debugging tasks into the MALP framework).

In general, any qualification domain $(D,\preceq,\perp,\top,\circ)$ whose attenuation operation $\circ$ conforms an adjoint-pair with a given implication operation $\leftarrow_\circ$, can be expressed as the multi-adjoint lattice $\langle D,\preceq,\leftarrow_\circ,\circ\rangle$. Anyway, we wish to finish this brief comparison by highlighting that the variety of connectives definable in multi-adjoint lattices is clearly much greater than those appearing in qualification domains (where for a given program, all rules must always use the same attenuation operator), which justifies the higher expressive power of MALP w.r.t. QLP.

Continuing now with the description of the multi-adjoint logic programming approach, a MALP *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. Rules whose body is $\top$ or equivalently, rules without body (or with empty body) are called *facts*. A *goal* is a body submitted as a query to the system.

Roughly speaking, a MALP program is a set of pairs $\langle\mathcal{R};v\rangle$ (we often write "$\mathcal{R}$ *with* v"), where $\mathcal{R}$ is a rule and v 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 $\langle\mathcal{R};v\rangle$ as a "rule". As an example, in **Figure 1** we show a MALP program whose rules define fuzzy predicates modeling degrees of youth ("y"), heritage ("h") and education ("e"), as well as the confidence level ("c") on people for repaying a loan. Note that the seventh rule models this last notion by considering young persons having good heritage or education degrees. In what follows, we are going to explain the procedural principle of MALP programs, whose application to goal "$c(X)$" w.r.t. our program, will assign truth degrees 0.772 (i.e, a credibility of 77.2%) and 0.38 (or 38% of confidence level) to "*mary*" and "*peter*", respectively.

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 2.2 (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 $\overset{AS}{\leadsto} \subseteq (\mathcal{E}\times\mathcal{E})$ 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 [21]):*

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

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

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

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 $\langle A'\leftarrow_i\top;v\rangle$. Also, the third case is introduced to cope with (possible) unsuccessful admissible derivations. As usual, rules are taken renamed apart. We shall use the symbols $\overset{\text{AS1}}{\leadsto}$, $\overset{\text{AS2}}{\leadsto}$ and $\overset{\text{AS3}}{\leadsto}$ 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 $\overset{\text{AS}}{\leadsto}$ symbol.

Multi-adjoint logic program $\mathcal{P}_1$:

$\mathcal{R}_1:$	$y(peter)$			$with\ 0.4$
$\mathcal{R}_2:$	$y(mary)$			$with\ 0.8$
$\mathcal{R}_3:$	$h(peter)$			$with\ 0.9$
$\mathcal{R}_4:$	$h(mary)$			$with\ 0.3$
$\mathcal{R}_5:$	$e(peter)$			$with\ 0.5$
$\mathcal{R}_6:$	$e(mary)$			$with\ 0.95$
$\mathcal{R}_7:$	$c(X)$	$\leftarrow_{\mathtt{P}}$	$(h(X)\ \vert_P\ e(X))\ \&_P\ y(X)$	$with\ 1$

Derivation D_1:

Admissible derivation

$$\begin{array}{ll}
\langle \underline{c(X)};\ id\rangle & \overset{\text{AS1}^{\mathcal{R}_7}}{\leadsto} \\
\langle \&_{\mathtt{P}}(1, \&_{\mathtt{P}}(|_{\mathtt{P}}(\underline{h(X_1)}, e(X_1)), y(X_1))); \{X/X_1\}\rangle & \overset{\text{AS2}^{\mathcal{R}_3}}{\leadsto} \\
\langle \&_{\mathtt{P}}(1, \&_{\mathtt{P}}(|_{\mathtt{P}}(0.9, \underline{e(peter)}), y(peter))); \{X/peter, X_1/peter\}\rangle & \overset{\text{AS2}^{\mathcal{R}_5}}{\leadsto} \\
\langle \&_{\mathtt{P}}(1, \&_{\mathtt{P}}(|_{\mathtt{P}}(0.9, 0.5), \underline{y(peter)})); \{X/peter, X_1/peter\}\rangle & \overset{\text{AS2}^{\mathcal{R}_1}}{\leadsto} \\
\langle \&_{\mathtt{P}}(1, \&_{\mathtt{P}}(|_{\mathtt{P}}(0.9, 0.5), 0.4)); \{X/peter, X_1/peter\}\rangle &
\end{array}$$

Interpretive derivation

$$\begin{array}{ll}
\langle \&_{\mathtt{P}}(1, \&_{\mathtt{P}}(\underline{|_{\mathtt{P}}(0.9, 0.5)}, 0.4)); \{X/peter\}\rangle & \overset{\text{IS}}{\leadsto} \\
\langle \&_{\mathtt{P}}(1, \underline{\&_{\mathtt{P}}(0.95, 0.4)}); \{X/peter\}\rangle & \overset{\text{IS}}{\leadsto} \\
\langle \underline{\&_{\mathtt{P}}(1, 0.38)}; \{X/peter\}\rangle & \overset{\text{IS}}{\leadsto} \\
\langle 0.38; \{X/peter\}\rangle. &
\end{array}$$

Figure 1: MALP program with admissible/interpretive derivations for goal "$c(X)$"

Definition 2.3 *Let $\mathcal{P}$ be a program, $\mathcal{Q}$ a goal and "id" the empty substitution. An* admissible derivation *is a sequence $\langle \mathcal{Q}; id\rangle \overset{AS}{\leadsto} \dots \overset{AS}{\leadsto} \langle \mathcal{Q}'; \theta\rangle$. 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 2.4 *In* **Figure 1** *we illustrate an admissible derivation (note that the selected atom in each step appears underlined), where the admissible computed answer (a.c.a.) is composed by the pair: $\langle \&_{\mathtt{P}}(1, \&_{\mathtt{P}}(|_{\mathtt{P}}(0.9, 0.5), 0.4)); \theta\rangle$ where θ only refers to bindings related with variables in the goal, i.e., $\theta = \{X/peter, X_1/peter\}[\mathcal{V}ar(c(X))] = \{X/peter\}$*

If we exploit all atoms of a given goal, by applying enough admissible steps, then it becomes a formula with no atoms (a L-expression) which can be interpreted w.r.t. lattice L by applying the following definition we initially presented in [18]:

Definition 2.5 (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 $r_1, \dots, r_n, r_{n+1} \in L$, we have that $[\![@]\!](r_1, \dots, r_n) = r_{n+1}$. Then, we formalize the notion of* interpretive computation *as a state transition system, whose transition relation $\overset{IS}{\leadsto} \subseteq (\mathcal{E} \times \mathcal{E})$ is defined as the least one satisfying: $\langle \mathcal{Q}[@(r_1, \dots, r_n)]; \sigma\rangle \quad \overset{IS}{\leadsto} \quad \langle \mathcal{Q}[@(r_1, \dots, r_n)/r_{n+1}];\sigma\rangle$.*

Definition 2.6 *Let $\mathcal{P}$ be a program and $\langle \mathcal{Q}; \sigma\rangle$ an a.c.a., that is, $\mathcal{Q}$ does not contain atoms (i.e., it is a L-expression). An* interpretive derivation *is a sequence $\langle \mathcal{Q}; \sigma\rangle \overset{IS}{\leadsto} \dots \overset{IS}{\leadsto} \langle \mathcal{Q}'; \sigma\rangle$. 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 2.7 *If we complete the previous derivation of Example 2.4 by applying 3 interpretive steps in order to obtain the final f.c.a.* $\langle 0.38; \{X/peter\}\rangle$*, we generate the interpretive derivation shown in* **Figure 1**.

2.3 Interpretive Steps and Cost Measures

A classical, simple way for estimating the computational cost required to built a derivation, consists in counting the number of computational steps performed on it. So, given a derivation D, we define its:

- *operational cost*, $\mathcal{O}_{\mathcal{C}}(D)$, as the number of admissible steps performed in D.
- *interpretive cost*, $\mathcal{I}_{\mathcal{C}}(D)$, as the number of interpretive steps done in D.

Note that the operational and interpretive costs of derivation D_1 performed in **Figure 1** are $\mathcal{O}_{\mathcal{C}}(D_1) = 4$ and $\mathcal{I}_{\mathcal{C}}(D_1) = 3$, respectively. Intuitively, $\mathcal{O}_{\mathcal{C}}$ informs us about the number of atoms exploited along a derivation. Similarly, $\mathcal{I}_{\mathcal{C}}$ seems to estimate the number of connectives evaluated in a derivation. However, this last statement is not completely true: $\mathcal{I}_{\mathcal{C}}$ only takes into account those connectives appearing in the bodies of program rules which are replicated on states of the derivation, but no those connectives recursively *nested* in the definition of other connectives. The following example highlights this fact.

Example 2.8 *A simplified version of rule* $\mathcal{R}_7$*, whose body only contains an aggregator symbol is:*

$$\mathcal{R}_7^* : c(X) \leftarrow_{\mathtt{P}} @_1(h(X), e(X), y(X)) \quad \textit{with} \quad \textit{1}$$

where $@_1$ *is defined as* $@_1(x_1, x_2, x_3) \triangleq \&_{\mathtt{P}}(|_{\mathtt{P}}(x_1, x_2), x_3)$*. Note that* $\mathcal{R}_7^*$ *has exactly the same meaning (interpretation) than* $\mathcal{R}_7$*, although different syntax. In fact, both ones have the same sequence of atoms in their head and bodies. The differences are regarding the set of connectives which explicitly appear in their bodies since in* $\mathcal{R}_7^*$ *we have moved* $\&_{\mathtt{P}}$ *and* $|_{\mathtt{P}}$ *from the body of the rule (see* $\mathcal{R}_7$*) to the connective definition of* $@_1$*. Now, we use rule* $\mathcal{R}_7^*$ *instead of* $\mathcal{R}_7$ *for generating the following derivation* D_1^* *which returns exactly the same f.c.a than* D_1*:*

$$\begin{array}{ll}
\langle \underline{c(X)};\ id\rangle & \overset{AS1^{\mathcal{R}_7}}{\leadsto} \\
\langle \&_{\mathtt{P}}(1, @_1(\underline{h(X_1)}, e(X_1)y(X_1))); \{X/X_1\}\rangle & \overset{AS2^{\mathcal{R}_3}}{\leadsto} \\
\langle \&_{\mathtt{P}}(1, @_1(0.9, \underline{e(peter)}, y(peter))); \{X/peter, X_1/peter\}\rangle & \overset{AS2^{\mathcal{R}_5}}{\leadsto} \\
\langle \&_{\mathtt{P}}(1, @_1(0.9, 0.5, \underline{y(peter)}))); \{X/peter, X_1/peter\}\rangle & \overset{AS2^{\mathcal{R}_1}}{\leadsto} \\
\langle \&_{\mathtt{P}}(1, \underline{@_1(0.9, 0.5, 0.4)}); \{X/peter, X_1/peter\}\rangle & \overset{IS}{\leadsto} \\
\langle \underline{\&_{\mathtt{P}}(1, 0.38)}; \{X/peter\}\rangle & \overset{IS}{\leadsto} \\
\langle 0.38; \{X/peter\}\rangle &
\end{array}$$

Note that, since we have exploited the same atoms with the same rules (except for the first steps performed with rules $\mathcal{R}_7$ *and* $\mathcal{R}_7^*$*, respectively) in both derivations, then* $\mathcal{O}_{\mathcal{C}}(D_1) = \mathcal{O}_{\mathcal{C}}(D_1^*) = 4$*. However, although connectives* $\&_{\mathtt{P}}$ *and* $|_{\mathtt{P}}$ *have been evaluated in both derivations, in* D_1^* *such evaluations have not been explicitly counted as interpretive steps, and consequently they have not been added to increase the interpretive cost measure* $\mathcal{I}_{\mathcal{C}}$*. This unrealistic situation is reflected by the abnormal result* $\mathcal{I}_{\mathcal{C}}(D_1) = 3 > 2 = \mathcal{I}_{\mathcal{C}}(D_1^*)$*. It is important to note that* $\mathcal{R}_7^*$ *must not be considered an optimized version of* $\mathcal{R}_7$*, even when the wrong measure* $\mathcal{I}_{\mathcal{C}}$ *seems to indicate the contrary.*

This problem was initially pointed out in [19], where a preliminary solution was proposed by assigning weights to connectives in concordance with the set of primitive operators involved in the definition of the proper connective @ as well as those ones recursively contained in the definitions of connectives invoked from @. Moreover, in [34] we improved the previous notion of "connective weight" by also taken into account the number of recursive calls to fuzzy connectives (directly or indirectly) performed in the definition of @.

A rather different way for facing the same problem is presented in [33], where instead on connective weights, we opt for the more "visual" method we have just implemented into $\mathcal{FLOPER}$, based on the subsequent re-definition of the behaviour of the interpretive phase.

Definition 2.9 (Small Interpretive Step) *Let $\mathcal{P}$ be a program, $\mathcal{Q}$ a goal and σ a substitution. Assume that the (non interpreted yet) L-expression $\Omega(r_1,\ldots,r_n)$ occurs in $\mathcal{Q}$, where Ω is just a primitive operator or a connective defined in the lattice $\langle L,\preceq\rangle$ associated to $\mathcal{P}$, and $r_1,\ldots,r_n$ are elements of L. We formalize the notion of* small interpretive computation *as a state transition system, whose transition relation $\overset{SIS}{\leadsto}\subseteq(\mathcal{E}\times\mathcal{E})$ is the smallest relation satisfying the following* small interpretive rules *(where we always consider that $\Omega(r_1,\ldots,r_n)$ is the selected L-expression in $\mathcal{Q}$):*

1) $\langle Q[\Omega(r_1,\ldots,r_n)];\sigma\rangle \overset{SIS}{\leadsto} \langle Q[\Omega(r_1,\ldots,r_n)/E'];\sigma\rangle$, *if Ω is a connective defined as $\Omega(x_1,\ldots,x_n)$ $\triangleq E$ and E' is obtained from the L-expression E by replacing each variable (formal parameter) x_i by its corresponding value (actual parameter) r_i, $1\leq i\leq n$, that is, $E'=E[x_1/r_1,\ldots,x_n/r_n]$.*

2) $\langle Q[\Omega(r_1,\ldots,r_n)];\sigma\rangle \overset{SIS}{\leadsto} \langle Q[\Omega(r_1,\ldots,r_n)/r];\sigma\rangle$, *if Ω is a primitive operator such that, once evaluated with parameters $r_1,\ldots,r_n$, produces the result r.*

From now on, we shall use the symbols $\overset{\text{SIS1}}{\leadsto}$ and $\overset{\text{SIS2}}{\leadsto}$ to distinguish between computation steps performed by applying one of the specific "small interpretive" rules. Moreover, when we use the expression *interpretive derivation*, we refer to a sequence of *small interpretive steps* (according to the previous definition) instead of a sequence of *interpretive steps* (regarding Definition 2.5). Note that this fact supposes too a slight revision of Definition 2.6 which does not affect the essence of the notion of fuzzy computed answer: the repeated application of both kinds of small interpretive steps on a given state only affects to the length of the corresponding derivations, but both ones lead to the same final states (containing the corresponding fuzzy computed answers).

Example 2.10 *Recalling again the a.c.a. obtained in Example 2.4, we can reach the final fuzzy computed answer $\langle 0.38;\{X/peter\}\rangle$ (achieved in Example 2.7 by means of interpretive steps) by generating now the following interpretive derivation D_2 based on "small interpretive steps" (Definition 2.9):*

$$\begin{array}{ll}
\langle \&_{\texttt{P}}(1,\&_{\texttt{P}}(\underline{|_{\texttt{P}}(0.9,0.5)},0.4));\{X/peter\}\rangle & \overset{SIS1}{\leadsto}\\
\langle \&_{\texttt{P}}(1,\&_{\texttt{P}}(\underline{(0.9+0.5)}-(0.9*0.5),0.4));\{X/peter\}\rangle & \overset{SIS2}{\leadsto}\\
\langle \&_{\texttt{P}}(1,\&_{\texttt{P}}(1.4-\underline{(0.9*0.5)},0.4));\{X/peter\}\rangle & \overset{SIS2}{\leadsto}\\
\langle \&_{\texttt{P}}(1,\&_{\texttt{P}}(\underline{1.4-0.45},0.4));\{X/peter\}\rangle & \overset{SIS2}{\leadsto}\\
\langle \&_{\texttt{P}}(1,\underline{\&_{\texttt{P}}(0.95,0.4)});\{X/peter\}\rangle & \overset{SIS1}{\leadsto}\\
\langle \&_{\texttt{P}}(1,\underline{0.95*0.4});\{X/peter\}\rangle & \overset{SIS2}{\leadsto}\\
\langle \underline{\&_{\texttt{P}}(1,0.38)};\{X/peter\}\rangle & \overset{SIS1}{\leadsto}\\
\langle \underline{1*0.38};\{X/peter\}\rangle & \overset{SIS2}{\leadsto}\\
\langle 0.38;\{X/peter\}\rangle &
\end{array}$$

Going back now to Example 2.8, we can rebuild the interpretive phase of Derivation D_1^* in terms of small interpretive steps, thus generating the following interpretive derivation D_2^*. Firstly, by applying a $\overset{SIS1}{\leadsto}$ step on the L-expression $\&_{\texttt{P}}(1,\underline{@_1(0.9,0.5,0.4)})$, it becomes $\&_{\texttt{P}}(1,\&_{\texttt{P}}(|_{\texttt{P}}(0.9,0.5),0.4))$, and from here, the interpretive derivation evolves exactly in the same way as derivation D_2 we have just done above.

At this moment, it is mandatory to meditate on cost measures regarding derivations D_1, D_1^*, D_2 and D_2^*. First of all, note that the operational cost $\mathcal{O}_{\mathcal{C}}$ of all them coincides, which is quite natural. However, whereas $\mathcal{I}_{\mathcal{C}}(D_1)=3>2=\mathcal{I}_{\mathcal{C}}(D_1^*)$, we have now that $\mathcal{I}_{\mathcal{C}}(D_2)=8<9=\mathcal{I}_{\mathcal{C}}(D_2^*)$. This apparent contradiction might confuse us when trying to decide which program rule ($\mathcal{R}_7$ or $\mathcal{R}_7^*$) is "better". The use of Definition 2.9 in derivations D_2 and D_2^* is the key point to solve our problem, as we are going to see. In Example 2.8 we justified that by simply counting the number of interpretive steps performed in Definition 2.5 might produce abnormal results, since the evaluation of connectives with different complexities were (wrongly) measured with the same computational cost. Fortunately, the notion of small interpretive step makes visible in the proper derivation all the connectives and primitive operators appearing in the (possibly recursively nested) definitions of any connective appearing in any derivation state. As we have seen, in D_2 we have expanded in three $\overset{\text{SIS1}}{\leadsto}$ steps the definitions of three connectives, i.e. $|_{\texttt{P}}$, and $\&_{\texttt{P}}$ twice, and we have applied five $\overset{\text{SIS2}}{\leadsto}$ steps to solve five primitive operators, that is, $+$, $-$, and $*$ (three times). The same computational effort as been performed in D_2^*, but also one more $\overset{\text{SIS1}}{\leadsto}$ step was applied to accomplish with the expansion of the extra connective $@_1$. This justifies why $\mathcal{I}_{\mathcal{C}}(D_2) = 8 < 9 = \mathcal{I}_{\mathcal{C}}(D_2^*)$ and contradicts the wrong measures of Example 2.8: the interpretive effort developed in derivations D_1 and D_2 (both using the program rule $\mathcal{R}_7$), is slightly lower than the one performed in derivations D_1^* and D_2^* (which used rule $\mathcal{R}_7^*$), and not the contrary.

The accuracy of our new way for measuring and performing interpretive computations seems to be crucial when comparing the execution behaviour of programs obtained by transformation techniques such as the fold/unfold framework we describe in [17, 11]. In this sense, instead of measuring the absolute cost of derivations performed in a program, we are more interested in the relative gains/lost of efficiency produced on transformed programs. For instance, by applying the so-called "aggregation operation" described in [11] we can transform rule $\mathcal{R}_7$ into $\mathcal{R}_7^*$ and, in order to proceed with alternative transformations (fold,unfold, etc.) if the resulting program degenerates w.r.t. the original one (as occurs in this case), we need an appropriate cost measure as the one proposed here to help us for taken decisions. This fact has capital importance for discovering drastic situations which can appear in degenerated transformation sequences such as the generation of highly nested definitions of aggregators. For instance, assume the following sequence of connective definitions: $@_{100}(x_1, x_2) \triangleq @_{99}(x_1, x_2)$, $@_{99}(x_1, x_2) \triangleq @_{98}(x_1, x_2), \dots,$ and finally $@_1(x_1, x_2) \triangleq x_1 * x_2$. When trying to solve two expression of the form $@_{99}(0.9, 0.8)$ and $@_1(0.9, 0.8)$, cost measures based on number of interpretive steps ([18]) and weights of interpretive steps ([19]) would assign 1 unit of interpretive cost to both derivations. Fortunately, our new approach is able to clearly distinguish between both cases, since the number of $\overset{\text{SIS1}}{\leadsto}$ steps performed in each one is rather different (100 and 1, respectively).

3 The "Fuzzy LOgic Programming Environment for Research" $\mathcal{FLOPER}$

As shown in the web page `http://dectau.uclm.es/floper/` designed in our research group for freely accessing $\mathcal{FLOPER}$ -see also references [32, 36, 35, 37, 31, 38]- during the last years we have been involved in the development of this tool for aliving MALP programs (which can be easily loaded into the system by means of plain-text files with extension ".fpl"). For example, our previous illustrative program included into file "`P1.fpl`", contains the following rules where, note for instance that the fuzzy connectives for implication, disjunction and conjunction symbols belonging to the *product logic* are respectively referred as "`<prod`", "`|prod`" and "`&prod`":

```
y(peter)                                   with 0.4.
y(mary)                                    with 0.8.
h(peter)                                   with 0.9.
h(mary)                                    with 0.3.
e(peter)                                   with 0.5.
e(mary)                                    with 0.95.
c(X) <prod (h(X) |prod e(X)) &prod y(X)    with 1.
```

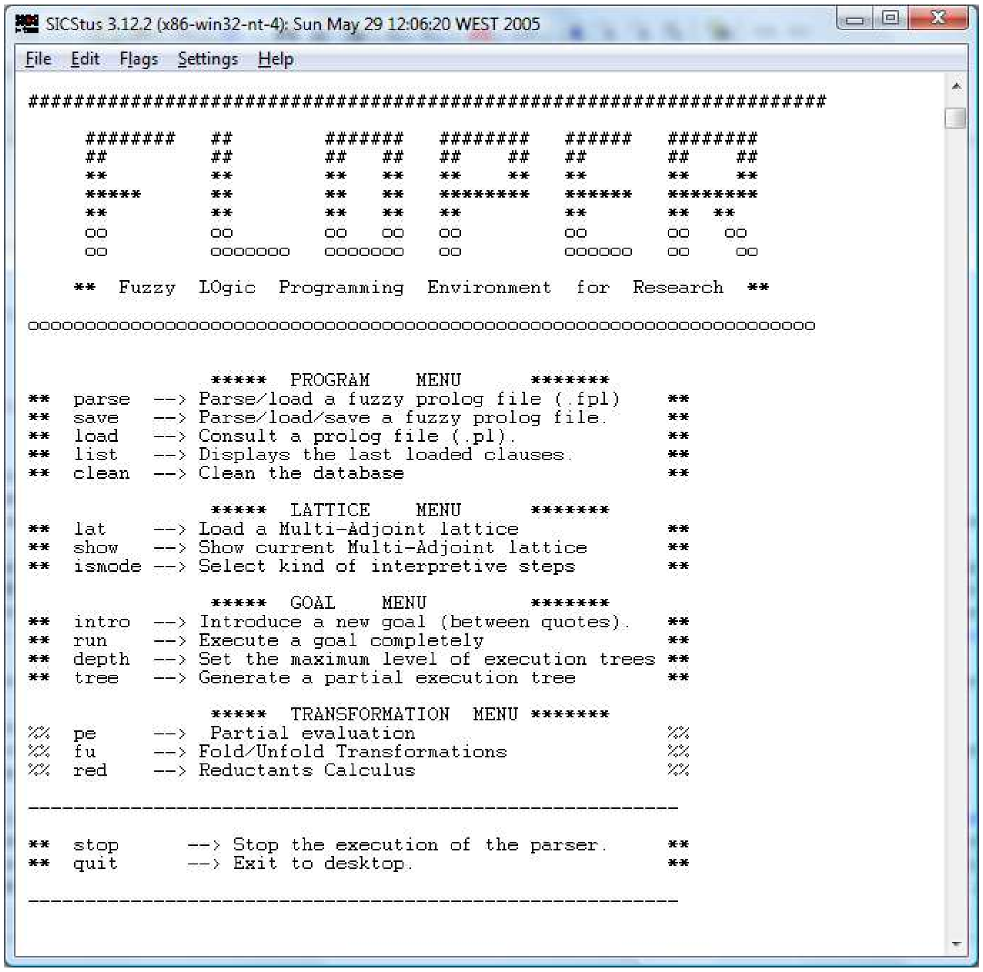
Figure 2: MALP program $\mathcal{P}_1$ loaded into $\mathcal{FLOPER}$

In order to simplify the task of coding fuzzy logic programs, our tool is able to parse MALP rules with 'syntactic sugar" trying to look as conservative extensions of Prolog clauses:

- Since the weight of a rule can be omitted if it coincides with the $\top$ element of the corresponding lattice then, a rule like '`p(X) with 1.`" can be simply expressed as "`p(X).`".
- When the concrete implication symbol connecting the body and head of a given a rule be irrelevant, we can write "`<-`" instead of using a particular label "$<_{logic}$", since $\mathcal{FLOPER}$ will choose an arbitrary implication (i.e., the last one found when textually exploring the lattice stored into the system) for this rule. So, "`p(X) <- q(X).`" is a valid rule.

- Something similar occurs (but even without the need of "`-`") for connectives used in the bodies of program rules, thus implying that "`p(X) <- q(X) & r(X).`" is a valid rule for any conjunction operator. At this point, it is important to remark that the last rule in **Figure 2** (we assume that the connectives belonging to the *product logic* are the last ones defined into the lattice stored into $\mathcal{FLOPER}$) can be highly simplified to the following shape: "`c(X) <- (h(X) | e(X)) & y(X).`".
- We can also include PROLOG clauses between "$$" symbols, for instance "`$p(X):-!,var(X).$`", and moreover, it is possible too to insert pure PROLOG code between "{}" symbols into the body of a fuzzy logic MALP rule, as occurs with "`p([H|T]) <- {Y is H+1} q(Y) & p(T).`".

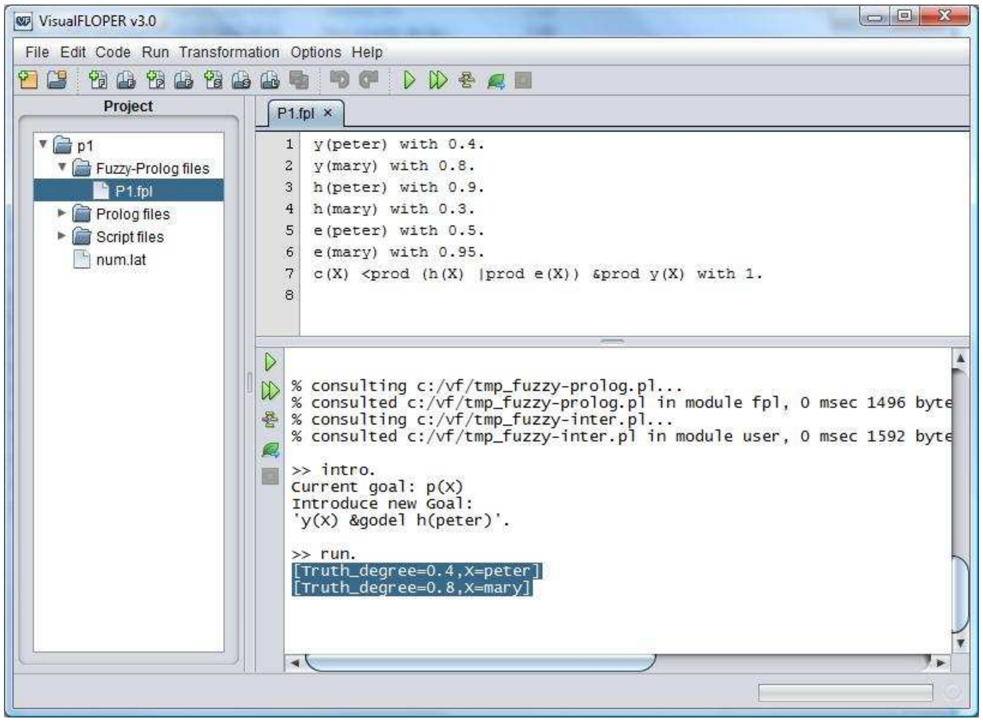
Once the application is loaded inside any standard PROLOG interpreter (like SWI or Sicstus -which is our case, by using v3.12.5-), it shows the menu shown in **Figure 3**. The parser has been implemented


SICStus 3.12.2 (x86-win32-nt-4): Sun May 29 12:06:20 WEST 2005
File Edit Flags Settings Help

```
######################################################################

     ########   ##        #######   ########   ######   ########
     ##         ##        ##   ##   ##    ##   ##       ##    ##
     **         **        **   **   **    **   **       **    **
     *****      **        **   **   ********   ******   ********
     **         **        **   **   **         **       **  **
     oo         oo        oo   oo   oo         oo       oo   oo
     oo         ooooooo   ooooooo   oo         oooooo   oo    oo

    **  Fuzzy  LOgic  Programming  Environment  for  Research  **

oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo


                *****  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.        **
**  clean  --> Clean the database                       **

                *****  LATTICE    MENU      *******
**  lat    --> Load a Multi-Adjoint lattice             **
**  show   --> Show current Multi-Adjoint lattice       **
**  ismode --> Select kind of interpretive steps        **

                *****  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        **

                *****  TRANSFORMATION  MENU *******
%%  pe     -->  Partial evaluation                      %%
%%  fu     --> Fold/Unfold Transformations              %%
%%  red    --> Reductants Calculus                      %%

--------------------------------------------------------

**  stop      --> Stop the execution of the parser.     **
**  quit      --> Exit to desktop.                      **

--------------------------------------------------------
```

Figure 3: Running $\mathcal{FLOPER}$ into any standard PROLOG platform

by using the classical DCG's (*Definite Clause Grammars*) resource of the PROLOG language, since it is a convenient notation for expressing grammar rules. Via the "parse" option, it is possible to load a ".fpl" file for which $\mathcal{FLOPER}$ generates two different PROLOG representations of the fuzzy code, as we will describe in sub-sections 3.1 and 3.2. Such code will cohabit with the set of clauses introduced by the user (together the PROLOG-based definition of the associated lattice that we will describe in sub-section 3.3) via the "load" option for consulting pure PROLOG files (".pl"). The system is equipped too with choices for saving and listing such rules as well as the "clean" option for removing all clauses from the $\mathcal{FLOPER}$ database.
The remaining menus are useful for executing goals and displaying evaluation trees, as well as for managing multi-adjoint lattices, as we are going to explain in what follows helped by the graphical interface recently



```
y(peter) with 0.4.
y(mary) with 0.8.
h(peter) with 0.9.
h(mary) with 0.3.
e(peter) with 0.5.
e(mary) with 0.95.
c(X) <prod (h(X) |prod e(X)) &prod y(X) with 1.
```

```
% consulting c:/vf/tmp_fuzzy-prolog.pl...
% consulted c:/vf/tmp_fuzzy-prolog.pl in module fpl, 0 msec 1496 byte
% consulting c:/vf/tmp_fuzzy-inter.pl...
% consulted c:/vf/tmp_fuzzy-inter.pl in module user, 0 msec 1592 byte

>> intro.
Current goal: p(X)
Introduce new Goal:
'y(X) &godel h(peter)'.

>> run.
[Truth_degree=0.4,X=peter]
[Truth_degree=0.8,X=mary]
```

Figure 4: The graphical interface of $\mathcal{FLOPER}$

developed for $\mathcal{FLOPER}$ and shown in **Figure 4**, which allows the comfortable use of "projects" in order to manage files with the following different purposes:

- several ".fpl" files can contain the set of MALP rules implementing a single MALP program,
- the set of clauses modeling its (unique) associated multi-adjoint lattice must be included into a file with obvious extension ".pl",
- additional Prolog code (for representing linguistic modifiers/variables, etc) can be also attached in different ".pl" files and
- several "script" files could be useful for executing in one go more than one of the set of commands we are going to explain in this section, as illustrated in the following example:

```
ord: intro.
arg: c(X).
ord: ismode.
arg: s.
ord: tree.
ord: leaves.
ord: run.
```

3.1 Running Programs

In order to fully execute a goal, $\mathcal{FLOPER}$ employs the high-level representation of the MALP program compiled via the "parse" option. The key point of this Prolog code is to extend each atom of the program with an extra argument, called *truth variable*, of the form `TVi`, which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. In the case of a fact, the extra argument obviously contains its weight. For instance, "`p(X) with 0.5`" is simply translated into the Prolog fact "`p(X, 0.5)`".

Fuzzy connectives are represented as predicates defined in the lattice associated to the program. For instance, the role of "`&godel`" is played by the Prolog predicate "`and_godel`". Since the fuzzy connective "`&godel`" is a binary operation, then its associated predicate "`and_godel`" has arity three: two parameters plus the result, returned in the third argument `TV`.

When compiling MALP rules, the last atom called in the body of the translated clause is the adjoint conjunction, which is intended to combine the truth degree of the body and the weight of the rule, in order to propagate the final truth degree to the head. For instance, given a MALP rule like "`p(X) <prod q(X,Y) &godel r(Y) with 0.8`", the resulting translated Prolog clause would look like "`p(X,_TV0) :- q(X,Y,_TV1), r(Y,_TV2), and_godel(_TV1,_TV2,_TV3), and_prod(0.8,_TV3,_TV0)`".

In order to execute a fuzzy program stored in a ".fpl" file , *FLOPER* needs firstly to load and compile it by using the "parse" option. Internally, while *FLOPER* analyzes the content of the file (following the DCG specification of the MALP syntax) it also generates two different Prolog representations of the fuzzy code: the high level representation coincides with a set of executable Prolog clauses as we have just described, while the low level representation allows *FLOPER* to draw execution trees as we will see in the next subsection.

Here we have an example of using the "parse" option, where note that *FLOPER* lists the content of any previously loaded ".pl" file (none in our case), the parsed MALP program and the generated Prolog code.

```
>> parse.
File to parse: ‘P1.fpl’.

No loaded files.

ORIGINAL FUZZY-PROLOG CODE:
y(peter) with 0.4.
y(mary) with 0.8.
h(peter) with 0.9.
h(mary) with 0.3.
e(peter) with 0.5.
e(mary) with 0.95.
c(X) <prod (h(X) |prod e(X)) &prod y(X) with 1.

GENERATED PROLOG CODE:
y(peter,0.4).
y(mary,0.8).
h(peter,0.9).
h(mary,0.3).
e(peter,0.5).
e(mary,0.95).
c(X,TV0):-h(X,_TV1),e(X,_TV2),lat:or_prod(_TV1,_TV2,_TV3),
          y(X,_TV4),lat:and_prod(_TV3,_TV4,_TV5),
          lat:and_prod(1,_TV5,TV0).
```

While parsing, *FLOPER* creates a file *tmp_fuzzy-prolog.pl* to allocate the generated Prolog code, in order to allow the possibility of saving it afterwards into a new file by using the "save" option.

One important feature is that translated connectives (like "`lat:and_prod/3`") are prefixed by "`lat:`", which means that the Prolog interpreter will search their definitions in the "`lat`" module, a different name-space designed to avoid name collisions. The resulting Prolog code can be executed in any Prolog engine, with the only requirement that the associated lattice must be loaded in the corresponding module. This can be easily achieved with the following two goals:

```
?- lat:consult(lattice.pl).
?- consult(program.fpl).
```

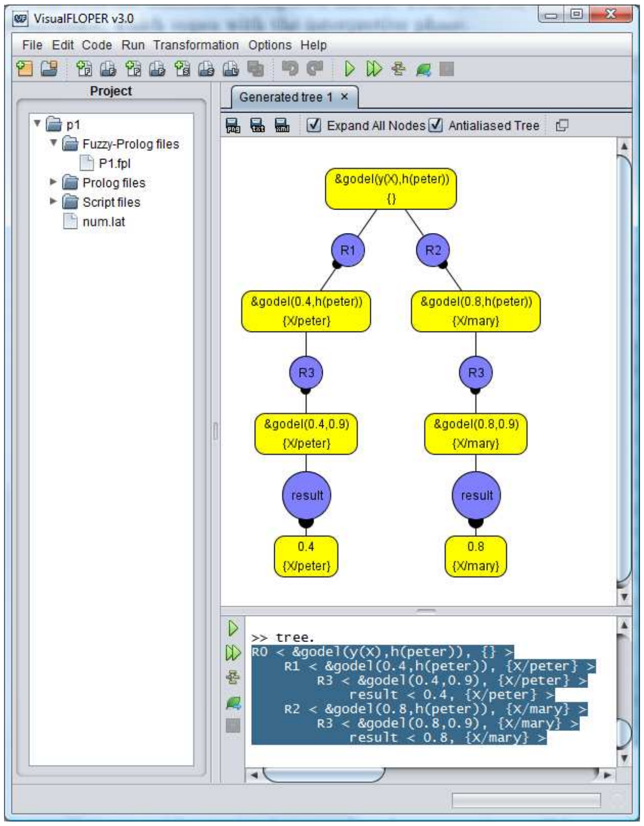
Moreover, goals are introduced in *FLOPER* by choosing the "intro" option and they suffer a very similar translation process to program rules. So, it is easy to see that a fuzzy goal like "`y(X) &godel`

`h(peter)`", is translated into the pure PROLOG goal "`y(X, _TV1),h(peter, _TV2), and_godel(_TV1, _TV2, Truth_degree)`" (note that the last truth degree variable is not anonymous now) for which the PROLOG interpreter returns the two desired fuzzy computed answers after selecting the "run" option (see **Figure 4**) :

```
[Truth_degree=0.4, X=peter]
[Truth_degree=0.8, X=mary]
```

3.2 Execution Trees

Apart from the compilation method to PROLOG code commented before, we have conceived a new low-level representation for the fuzzy code which is useful for building execution trees with any level of depth and offering too debugging (tracing) capabilities. For instance, after parsing the last rule of our program, we obtain the following expression (which is "asserted" into the database of the interpreter as a PROLOG fact, but it is never executed directly, in contrast with the previous PROLOG-based, high-level representation of the fuzzy code) whose components have obvious meanings:



```
>> tree.
R0 < &godel(y(X),h(peter)), {} >
    R1 < &godel(0.4,h(peter)), {X/peter} >
        R3 < &godel(0.4,0.9), {X/peter} >
            result < 0.4, {X/peter} >
    R2 < &godel(0.8,h(peter)), {X/mary} >
        R3 < &godel(0.8,0.9), {X/mary} >
            result < 0.8, {X/mary} >
```

Figure 5: *FLOPER* drawing an execution tree

```
rule(7,
     head(atom(pred(c,1),[var('X')])),
     impl(prod),
     body(and(prod,2,[or(prod,2,[atom(pred(h,1),[var('X')]),
                                 atom(pred(e,1),[var('X')])]),
                      atom(pred(y,1),[var('X')])])),
     td(1)).
```

FLOPER is equipped with three options related with *tracing* tasks. Option "tree" draws the execution tree (which collects a different derivation from the root to each leaf) of a goal w.r.t. a program. Option "depth" fixes the maximum length allowed for their branches (initially 3). And, finally, option "ismode" fixes the detail level of the interpretive phase associated to each derivation in the tree.

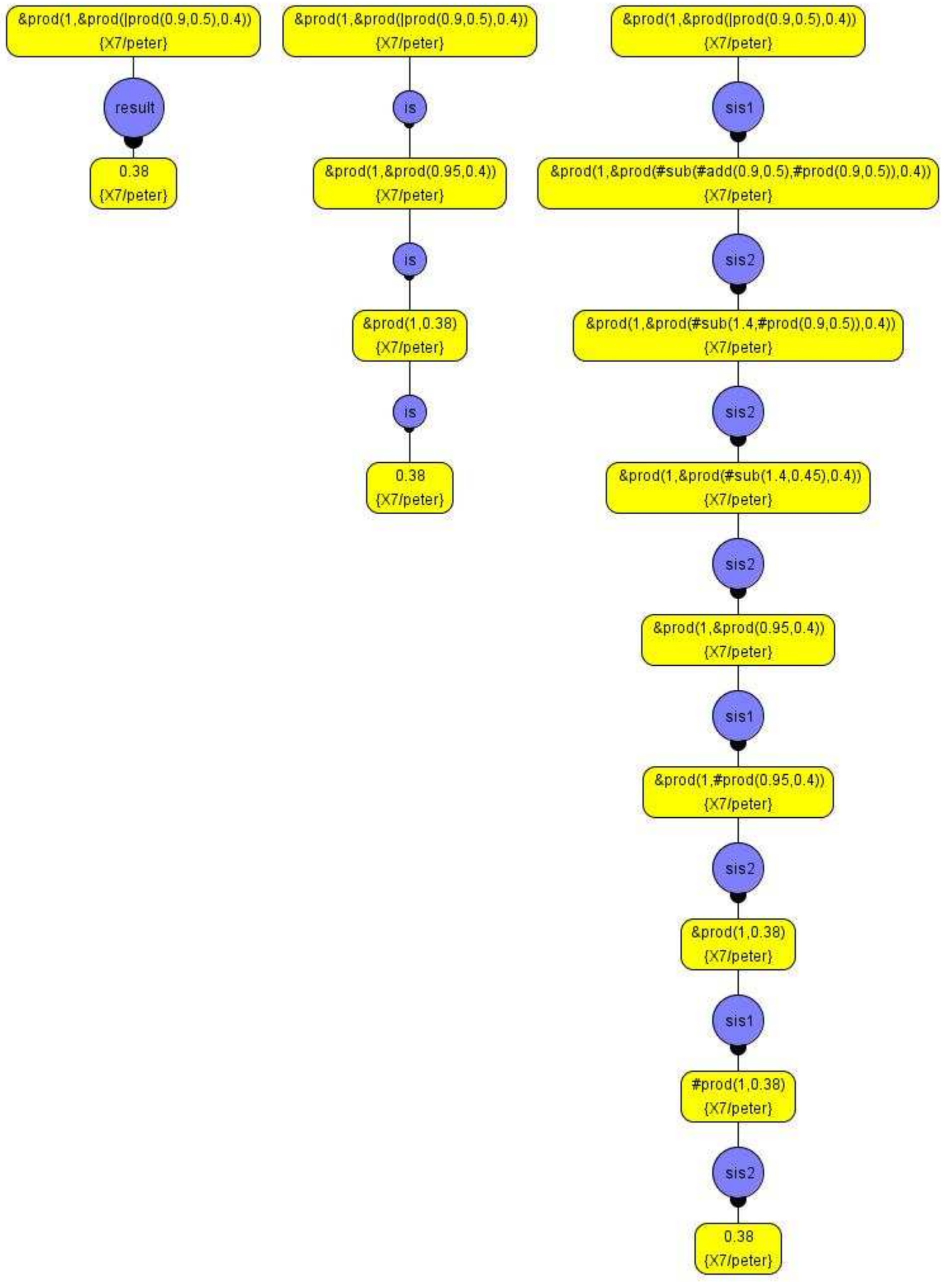


Figure 6: Comparing different modes of performing the interpretive phase

So, let us consider again the MALP program of our running example. For goal "`y(X) &godel h(peter)`", $\mathcal{FLOPER}$ displays the tree showed in **Figure 5**. In this screen-shot, we find two representations of the same tree: the middle-up window shows the proper graphic which is easy to understand and manipulate (by moving nodes, showing only skeletons of trees, etc.), while the marked text in the middle-down window represents the same tree in plain text. In this last case, each line contains a 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 and nodes obtained via $\overset{AS3}{\leadsto}$ are always labeled with the virtual, non existing rule `R0`), and nodes belonging to the same branch appear in different lines appropriately indented to help the readability of the tree (which only contain two different branches in our case). Generated trees can be saved in ".jpg", ".txt" and "xml" formats.

Moreover, $\mathcal{FLOPER}$ allows the user to choose the level of information given by the interpretive phase in the execution tree through option "ismode". We can choose among the following three modes:

- `large`: This mode omits the entire interpretive phase, offering only the final leaf (if exists) of each branch.
- `medium`: Performs classical interpretive steps according Definition 2.5, in order to evaluate each expression of the goal till finding the final solution.
- `small`: This mode applies our improved notion of *small interpretive step* provided in Definition 2.9, which is very useful for visualizing the more or less complexity (distinguishing between connective calls and primitive operators evaluation) of connectives exploited during the interpretive phase. **Figure 6** shows that the shorter the step is, the larger tree is generated.

To finish this block, we are going to illustrate now with a very simple example that the new options are 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 when finding non defined atoms, in contrast with the traces (based on finite, non- failed, admissible/interpretive derivations) that the "tree" option displays. So, consider the following MALP program:

```
p(a)                          with 0.8.
p(X) <prod p(s(s(s(X))))      with 0.9.
p(b)                          with 0.6.
```

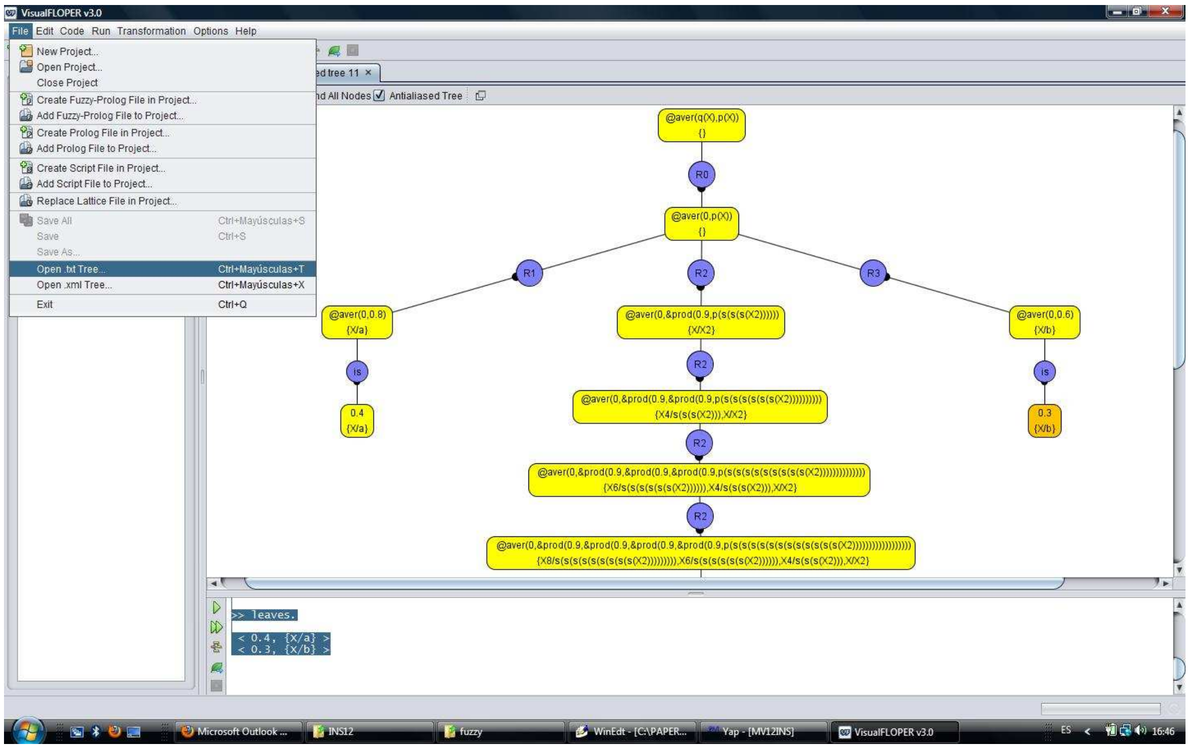
where the first and last rules indicate that a goal like "`p(X)`" admits two solutions, but the second rule would be responsible of introducing an infinite branch between the leaves associated to both fuzzy computed answers in the corresponding execution tree. Moreover, if we plan to run a more complex goal like "`q(X) @aver p(X)`" (where, obviously, the used connective refers to the *average* aggregator), we find a second problem related now with an undefined atom. In contrast with PROLOG, in our fuzzy setting the evaluation of "q(X)" doesn't fail, since $\mathcal{FLOPER}$ proceeds with an $\overset{AS3}{\leadsto}$ step according Definition 2.2, and hence, it is possible to find the two desired fuzzy computed answers for that goal, as shown in the execution tree of **Figure 7**. By choosing option "leaves", the system displays the content of the two fully evaluated leaves "`<0.4,{X/a}>`" and "`<0.3,{X/b}>`", as desired.

As we have seen, the generation of traces based on execution trees, contribute to increase the power of $\mathcal{FLOPER}$ by providing debugging capabilities which allow us to discover solutions for queries even when the pure PROLOG compilation-execution process becomes insufficient.

3.3 Managing Lattices

We have conceived a very easy way to model lattices of truth degrees for being included into the $\mathcal{FLOPER}$ 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 recommended 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).`


VisualFLOPER v3.0
File Edit Code Run Transformation Options Help
New Project...
Open Project...
Close Project
Create Fuzzy-Prolog File in Project...
Add Fuzzy-Prolog File to Project...
Create Prolog File in Project...
Add Prolog File to Project...
Create Script File in Project...
Add Script File to Project...
Replace Lattice File in Project...
Save All Ctrl+Mayúsculas+S
Save Ctrl+S
Save As...
Open .txt Tree... Ctrl+Mayúsculas+T
Open .xml Tree... Ctrl+Mayúsculas+X
Exit Ctrl+Q
ed tree 11
nd All Nodes
Antialiased Tree
@aver(q(X),p(X)) {}
R0
@aver(0,p(X)) {}
R1
R2
R3
@aver(0,0.8) {X/a}
@aver(0,&prod(0.9,p(s(s(s(X2)))))) {X/X2}
@aver(0,0.6) {X/b}
is
0.4 {X/a}
is
0.3 {X/b}
@aver(0,&prod(0.9,&prod(0.9,p(s(s(s(s(s(s(X2)))))))))) {X4/s(s(s(X2))),X/X2}
@aver(0,&prod(0.9,&prod(0.9,&prod(0.9,p(s(s(s(s(s(s(s(s(s(X2))))))))))))))) {X6/s(s(s(s(s(s(X2)))))),X4/s(s(s(X2))),X/X2}
@aver(0,&prod(0.9,&prod(0.9,&prod(0.9,&prod(0.9,p(s(s(s(s(s(s(s(s(s(s(s(s(X2))))))))))))))))))) {X8/s(s(s(s(s(s(s(s(s(X2))))))))),X6/s(s(s(s(s(s(X2)))))),X4/s(s(s(X2))),X/X2}
Microsoft Outlook ...
INS12
fuzzy
WinEdt - [C:\PAPER...
Yap - [MV12INS]
VisualFLOPER v3.0
ES
16:46

```
>> leaves.

 < 0.4, {X/a} >
 < 0.3, {X/b} >
```

Figure 7: Tree with an infinite branch and a $\overset{AS3}{\leadsto}$ step

- `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), $|_{label_2}$ (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).`

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 2.1, each output will contain the corresponding substitution (i.e., the *crisp* notion of computed answer obtained by means of classical SLD-resolution in PROLOG) together with the maximum truth degree 1.

On the other hand and following the PROLOG style regulated by the previous guidelines, in file "num.pl" we have included the clauses shown in **Figure 8**. 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 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 a useful description for the hybrid aggregator *average*.

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

```
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.
```

Figure 8: Multi-adjoint lattice modeling truth degrees in the real interval [0,1] ("num.pl")

Note also that we have included definitions for auxiliary predicates, whose names always begin with 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 expressiveness (this is the case for instance, of the three kinds of fuzzy connectives we are considering: conjunctions, 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 partially ordered multi-adjoint lattice in **Figure 9** for which the conjunction and implication connectives based on the *Gödel* logic described in Section 2.1 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. To this end, observe in the Prolog code accompanying **Figure 9** 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 not comparable, any call to "`leq(alpha,beta)`" or "`leq(beta,alpha)`" will always fail.
- However, goals "`pri_inf(alpha,beta,X)`" and "`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`".

As a final example, we can also define the so called *Borel algebra* based on the union of intervals (where for instance $\mathcal{B}([0,1])$ is the union of intervals between 0 and 1 [46, 39]) as a Prolog program for being used into $\mathcal{FLOPER}$ as follows:

- A member of this algebra is a list of pairs representing disjoined intervals.
- The top element is the point 1 (interval from 1 to 1), and the bottom one is the point 0 (interval from 0 to 0).
- A union of intervals, U, is less or equal than other union of intervals U' if for each $I \in U$, there exists another interval $I' \in U'$, such that $I \subseteq I'$.

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

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

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

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

Figure 9: Partially ordered lattice with four elements

```
member([i(X,Y)]) :-                        number(X),number(Y),X=<Y.
member([i(X,Y),i(Z,T)|U]) :-               number(X),number(Y),number(Z),
                                           X=<Y, Y < Z,member([i(Z,T)|U]).

bot([i(0,0)]).                             top([i(1,1)]).

leq([X1,X2|X],Y) :-                        existsGreater(X1,Y),leq([X2|X],Y).
leq([X1],Y) :-                             existsGreater(X1,Y).

existsGreater(i(Xb,Xt),[i(Yb,Yt)|Y]) :-  Yb=<Xb, Xt=<Yt,!.
existsGreater(X,[_|Y]) :-                  existsGreater(X,Y).
```

3.4 Linguistic modifiers and linguistic variables

Another interesting feature of $\mathcal{FLOPER}$ is its ability for managing linguistic modifiers and linguistic variables in a very easy way. A linguistic modifier can be seen as an aggregator such that, when applied to a fuzzy expression, it alters its final truth degree. Some examples of well known modifiers are *"very"* and *"roughly"*, where the first one tends to return a lower truth degree, and the second one, a higher truth degree. Since the concept of modifier is dependent of a concrete lattice, its definition should appear inside the PROLOG file containing the proper multi-adjoint lattice loaded into $\mathcal{FLOPER}$. For instance, in the typical lattice $\langle [0,1], \leq \rangle$ used in previous examples, we could define some modifiers by means of the following usual formulae:

modifier	formula	implementation
extremely	$extremely(x) = x^4$	$\mathrm{agr\_extremely(X, TV0)\ :-\ TV0\ is\ X * X * X * X.}$
very	$very(x) = x^2$	$\mathrm{agr\_very(X, TV0)\ :-\ TV0\ is\ X * X.}$
moreorless	$moreless(x) = x^{1/2}$	$\mathrm{agr\_moreless(X, TV0)\ :-\ TV0\ is\ sqrt(X).}$
roughly	$roughly(x) = x^{1/4}$	$\mathrm{agr\_roughly(X, TV0)\ :-\ TV0\ is\ sqrt(sqrt(X)).}$

With these modifiers, now we could update the program of **Figure 1** to increase the difficulty for obtaining a credit in crisis times. The only rule to be modified is the seventh one, whose new shape could look like: $R_7^*:\ c(X) \leftarrow_{\mathtt{P}} @very((h(X) \mid_P e(X)) \mathbin{\&_P} y(X))$ *with* 1. Linguistic modifiers increase the expresiveness of the fuzzy language and as said before, they have to be defined as part of the lattice associated to the MALP program in the current project.

On the other hand, linguistic variables allow us to represent linguistic symbols defined by means of fuzzy sets. A linguistic variable is characterized by a tuple $\langle x, T, U, G, M \rangle$, where x is the name of the

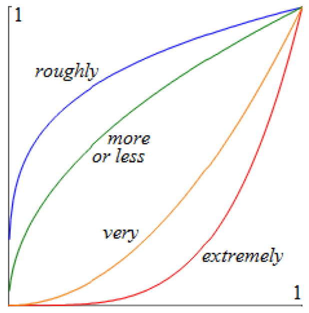


Figure 10: Linguistic modifiers

variable, T the set of linguistic symbols or terms of x, U the universe where x is defined, G represents a syntactic rule to generate linguistic terms, and M is a semantic rule for assigning to each linguistic symbol t its fuzzy set $M(t)$. For instance, we can represent the linguistic variable *distance*, with values $\{near,\ far\_away\}$ defined over the universe $[0, +\infty)$ in kilometers, as shown in **Figure 11**. We can implement this variable into a Prolog file defining a fuzzy predicate for each symbol. The arguments are an input variable (D) giving the crisp distance, and an output variable (TV) returning the degree of membership of the input variable to the fuzzy set. Then, it is possible to load this Prolog program into the current project of $\mathcal{FLOPER}$, in order to use such definitions in the corresponding MALP program.

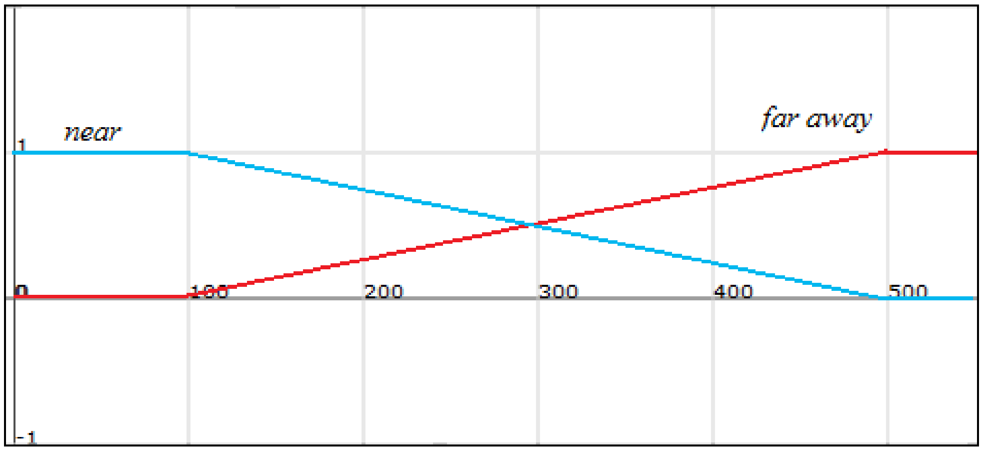


Figure 11: Linguistic variable "distance" with terms "near" and "far away"

```
near(D,1) :- D=<100.
near(D,TV):- 100<X, X=<500, TV is (500-X)/400.
near(D,0) :- 500<D.

far_away(D,0) :- D=<100.
far_away(D,TV):- 100<X, X=<500, TV is (X-100)/400.
far_away(D,1) :- 500<D.
```

In the following example, we design a touristic application to compute the best destination for vacations. The database includes some cities and relevant information relating them (kind of weather, good sights, distance from our hometown). It is clear that some of this information has a very fuzzy taste, so the choice of a fuzzy language is desirable. We can *fuzzify* the crisp distance using the linguistic variable defined above as follows.

```
nice_weather(madrid) with 0.8.
nice_weather(istanbul) with 0.7.
nice_weather(moscow) with 0.2.
nice_weather(sydney) with 0.5.

many_sights(madrid) with 0.6.
many_sights(istanbul) with 0.7.
many_sights(moscow) with 0.2.
many_sights(sydney) with 0.6.

crisp_distance(madrid, 250).
crisp_distance(istanbul, 3700).
crisp_distance(moscow, 4200).
crisp_distance(sydney, 18000).

good_destination(X) <- @roughly(crisp_distance(X,D) & near(D))
                       @aver
                       @very(nice_weather(X) & many_sights(X)).
```

Note that the use too of linguistic modifiers in the last MALP rule means that we give little importance to the distance since we are more interested on weather and sights. Now, for goal `good_destination(X)`, the execution of the previous program into $\mathcal{FLOPER}$ returns the following results (meaning that the best destination, according to our specification, is Madrid, followed by far by Istanbul and Sydney, while Moscow scores 0):

```
[Truth_degree=0.0,X=moscow]
[Truth_degree=0.005000000000000009,X=sydney]
[Truth_degree=0.07999999999999996,X=istanbul]
[Truth_degree=0.5245698525097306,X=madrid]
```

Note that, with the current definition of fuzzy predicates 'near' and 'far_away', no matter how we reduce the importance of distance since, if it is over 500 kilometers, it will always be near with 0 truth degree, and if it is under 100 kilometers, it will be far with 0 truth degree. If our perception of distance changes, these notions will become useless. To fix that, we can define them with non-linear rules like the following ones, which show that $\mathcal{FLOPER}$ is flexible enough to deal with fuzzy predicates defined (in PROLOG) by means of non-linear arithmetic expressions:

```
near(D,TV) :- TV is 250/(D+250).
far(D,TV)  :- TV is 1 - 250/(D+250).
```

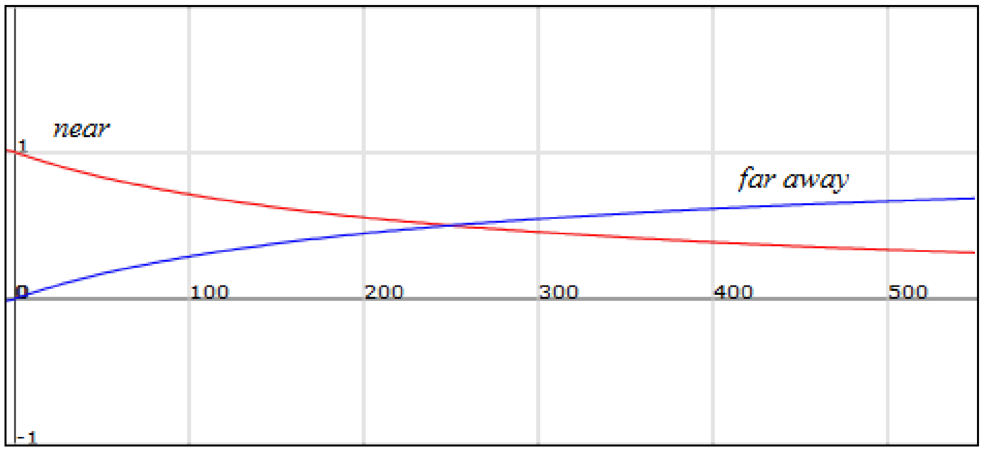


Figure 12: Linguistic variable "distance" with flexible versions of "near" and "far away"

4 Extending Lattices and Declarative Traces

This section presents different methods to gain expressiveness when designing a MALP program by manipulating its associated lattice. The main technique consists on agglutinating several lattices in order to obtain the Cartesian product of them, which is also a multi-adjoint lattice [31]. Once this is done, a new functionality emerges for obtaining declarative traces (when evaluating goals) at a very low cost.

Theorem 4.1 *If $L_1, \ldots, L_n$ are a finite number of multi-adjoint lattices, then its Cartesian product $L = L_1 \times \cdots \times L_n$ is also a multi-adjoint lattice.*

In order to simplify our explanation, but without lost of generality, we only consider two multi-adjoint lattices $(L_1, \leq_1, \&_1, \leftarrow_1)$ and $(L_2, \leq_2, \&_2, \leftarrow_2)$, each one equipped with just a single adjoint pair. Then, $L = L_1 \times L_2$ has lattice structure with an ordering relation induced in the product from $(L_1, \leq_1)$ and $(L_2, \leq_2)$ as follows:

$$(x_1, y_1) \leq (x_2, y_2) \Leftrightarrow x_1 \leq_1 x_2, y_1 \leq_2 y_2$$

Moreover, being $\top_1 = sup(L_1)$, $\bot_1 = inf(L_1)$, $\top_2 = sup(L_2)$ and $\bot_2 = inf(L_2)$, we have that $(\top_1, \top_2) = sup(L)$ and $(\bot_1, \bot_2) = inf(L)$ which implies that the Cartesian product L is a bounded lattice if both L_1 and L_2 are also bounded lattices. Analogously, $L_1 \times L_2$ is a complete lattice if L_1 and L_2 verify too the same property. Finally, from the adjoint pairs $(\&_1, \leftarrow_1)$ and $(\&_2, \leftarrow_2)$ in L_1 and L_2, respectively, it is possible to define the following connectives in L:

$$\begin{aligned} (x_1, y_1)\&(x_2, y_2) &\triangleq (x_1 \&_1 x_2, y_1 \&_2 y_2) \\ (x_1, y_1) \leftarrow (x_2, y_2) &\triangleq (x_1 \leftarrow_1 x_2, y_1 \leftarrow_2 y_2) \end{aligned}$$

for which it is easy to justify that they conform an adjoint pair in $L_1 \times L_2$ (thus satisfying, in particular, the adjoint property). In a similar way, it is also possible to define new connectives (conjunctions, disjunctions and aggregators) in the Cartesian product $L_1 \times L_2$ from the corresponding pairs of operators defined in L_1 and L_2.

$\mathcal{FLOPER}$ is able to deal with Cartesian products of multi-adjoint lattices by acting on the "`member/1`" predicate, which accept as argument any object which belong to a lattice. Some examples of simple "`member/1`" definitions are:

- `member(X) :- number(X), X=<1, X>=0`: [0,1] interval
- `member([X|L])`: lists
- `member(info(X,Y))`: pairs

The very intuitive way to obtain Cartesian product of lattices is to use Prolog functions with arity 2 (indeed, the cartesian product of n lattices can be represented using functions with arity n). In order to implement a cartesian product of lattices in a Prolog file, we define "`member/1`" predicate whose parameter is a term headed with a function symbol, for instance: `member(f(X,Y)) :- check1(X), check2(Y)`. Of course, predicates "`leq/2`", "`top/1`", "`bot/1`" and connectives have to be defined following the same criterium in order to implement the concrete Cartesian product of lattices.

Before showing an example of Cartesian product modeled in $\mathcal{FLOPER}$, let us consider again the so called *domain of weight values* $\mathcal{W}$ used in the QLP (*Qualified Logic Programming* framework of [44, 7, 43], whose elements are intended to represent *proof costs*, measured as the weighted depth of proof trees. As explained in Section 2, $\mathcal{W}$ can be seen as lattice $(\mathbb{N} \cup \{\infty\}, \geq)$, where $\geq$ is the reverse of the usual numerical ordering (with $\infty \geq d$ for any $d \in \mathbb{N}$) and thus, the bottom elements is ∞ and the top element is 0 (and not vice versa). Note that in this lattice the arithmetic operation "+" plays the role of a *conjunction* (it is easy to prove that in this setting such definition of & verifies the properties required by t-norms [40]). Moreover, we can obtain the residual implication of the "+" t-norm, defined as $y \leftarrow z \triangleq sup\{t \in \mathcal{W} : t + z \leq y\}$, which in this particular case acquires the following shape:

$$y \leftarrow z \triangleq \begin{cases} y - z, & if \quad z \geq y \\ 0, & if \quad y > z \end{cases}$$

So, the reader can easily check that $(+, \leftarrow)$ conforms an adjoint pair in $(\mathbb{N} \cup \{\infty\}, \geq)$, thus accomplishing with Definition 2.1 in Section 2, which implies that $\mathcal{W}$ is in fact a multi-adjoint lattice. A valid implementation of $\mathcal{W}$ lattice presents the following definitions:

```
member(X) :- number(X).      member(infty).
leq(infty,X).                leq(X,Y):-X>=Y.
top(0).                      bot(infty).
and_plus(X,infty,infty).     and_plus(infty,X,infty).     and_plus(X,Y,Z):-Z is X+Y.
```

In order to associate the lattice $\mathcal{W}$ to our program $\mathcal{P}_1$, we have to change the weights of each rule by valid truth degrees belonging to the new lattice. For instance, rule "$y(peter)$ *with* 0.4" would be rewritten as "$y(peter)$ *with* 1" (the underlying idea is that "the use of each program rule in a derivation implies the application of one admissible step"), resulting in a new program, $\mathcal{P}_{\mathcal{W}}$. By using the "`lat`" option of $\mathcal{FLOPER}$, we can built a project which associates lattice $\mathcal{W}$ to program $\mathcal{P}_{\mathcal{W}}$ and now, for goal "$c(X)$", we can generate an admissible derivation similar to the one seen in **Figure 1**, but ending now with $\langle \&_{\mathtt{P}}(info(1, \&_{\mathtt{P}}(|_{\mathtt{P}}(1,1),1)), \{X/peter\} \rangle$. Since: $\&_{\mathtt{P}}(1, \&_{\mathtt{P}}(|_{\mathtt{P}}(1,1),1)) = +(1, +(+(1,1),1))) = 4$, the final fuzzy computed answer or f.c.a. $\langle 4; \{X/peter\}\rangle$ indicates that goal "$c(X)$" holds when X is *peter*, as proved after applying 4 admissible steps.

On the other hand, since we have said that $\mathcal{V}$ is the multi-adjoint lattice $([0,1], \leq)$ based on real numbers in the unit interval used along this paper (which is equipped with three adjoint pairs modeling implication and conjunction symbols collected from the *Lukasiewicz logic*, *Gödel intuitionistic logic* and *product logic*) then, in the Cartesian product $\mathcal{V} \times \mathcal{W}$ we will find the top and bottom elements $(1,0)$ and $(0,\infty)$, respectively, as well as the following definitions of conjunction operations (among other connectives) whose names are mirroring to *extensions* of the *product logic* and *Lukasiewicz logic*:

$$(v_1, w_1) \ \&_{\mathtt{P+}} \ (v_2, w_2) \ \triangleq \ (v_1 * v_2, w_1 + w_2)$$
$$(v_1, w_1) \ \&_{\mathtt{L+}} \ (u_2, w_2) \ \triangleq \ (\max(0, v_1 + v_2 - 1), w_1 + w_2)$$

Moreover, we can also conceive a more powerful lattice expressed as the *Cartesian product* of $\mathcal{V}$ (see **Figure 8**) and $\mathcal{W}$. Now, each element includes two components, coping with truth degrees and cost measures. In order to be loaded into $\mathcal{FLOPER}$, we must define in PROLOG the new lattice, whose elements could be expressed as data terms of the form "`info(Fuzzy_Truth_Degree, Cost_Number_Steps)`". Some of the required predicates are:

```
member(info(X,W)):-number(X), 0=<X,X=<1,(W=infty,!; number(W),1=<W).

leq(info(X1,W1),info(X2,W2)):-X1 =< X2, (W1=infty,!; number(W2), W2 =< W1).

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

and_prod(info(X,W1),info(Y,W2),info(Z,W3)) :- pri_prod(X,Y,Z),pri_add(W1,W2,W3).

pri_add(infty,_,infty).                              pri_add(_,infty,infty).
pri_add(X,Y,Z) :- number(X), number(Y), Z is X+Y.
```

Finally, if the weights assigned to the rules of our example are "`info(0.4,1)`" for $\mathcal{R}_1$, "`info(0.8,1)`" for $\mathcal{R}_2$, "`info(0.9,1)`" for $\mathcal{R}_3$ and so on, then we would obtain the desired f.c.a. $\langle \mathtt{info}(0.38, 4); \{X/peter\}\rangle$ for goal "$c(X)$", with the obvious meaning that we need 4 admissible steps to prove that the query is true at a 38% degree when X is "peter".

One step beyond, we will also see that if instead of the number of computational steps, we are interested in knowing more detailed data about the set of program rules and connective definitions evaluated for obtaining each solution then, instead of $\mathcal{W}$ it will be mandatory to use a new lattice $\mathcal{S}$ based on strings or labels (i.e., sequences of characters) for generating the Cartesian product $\mathcal{V} \times \mathcal{S}$. In [38] we show not only that $\mathcal{S}$ is a complete multi-adjoint lattice, but also that the concatenation of strings, usually called "*append*" in many programming languages, plays the role of a conjunction connective in such lattice.

```
member(info(X,Y)):-number(X),0=<X,X=<1,atom(Y).   top(info(1,'')).

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

pri_prod(X,Y,Z,'#PROD.'):-Z is X * Y.

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).
.....
```

Figure 13: Multi-adjoint lattice modeling truth degrees with labels

In order to be loaded into $\mathcal{FLOPER}$, we need to define again the new lattice as a Prolog program, whose elements will be terms of the form "`info(Fuzzy_Truth_Degree, Label)`" as shown in **Figure 13** (we only list some representative clauses).
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 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, as occurs with the following example:

```
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.').
```

where we have used in the first rule aggregator "`@aver2`" (intended to compute the average between the results achieved by applying two different disjunction operations on the parameters) defined in Prolog as:

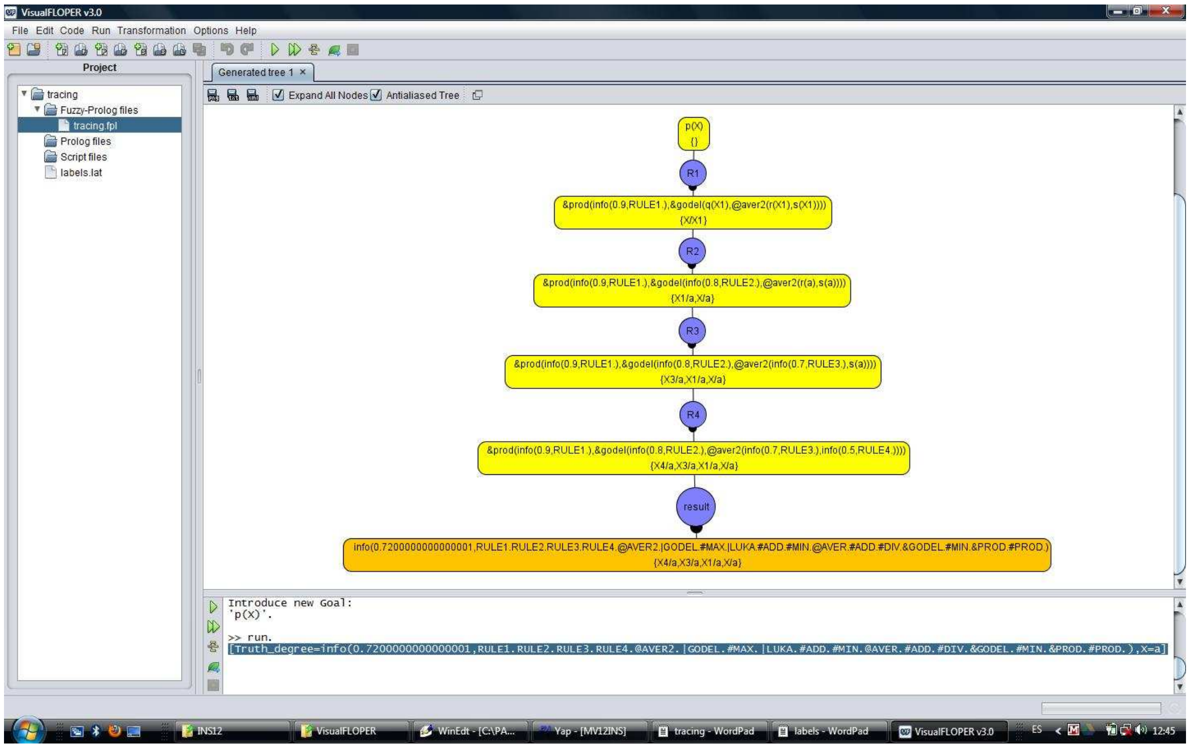
```
agr_aver2(X,Y,info(Za,Zb)) :- or_godel(X,Y,Z1),or_luka(X,Y,Z2),
                              agr_aver(Z1,Z2,info(Za,Zc)),
                              pri_app(Zc,'@AVER2.',Zb).
```

Now, the reader can easily test that, after executing goal "`p(X)`", we obtain the following fuzzy computed answer which includes the desired declarative trace containing the sequence of program-rules and connective-calls (mirroring $\overset{AS}{\leadsto}$ and $\overset{SIS1}{\leadsto}$ steps, according definitions 2.2 and 2.9, respectively) evaluated till finding the final solution:

```
 >> run.

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

With a very little extra effort, we can extend the previous lattice to have into account also the exploited primitive operators during the interpretive phase, thus simulating $\overset{SIS2}{\leadsto}$ steps (see again Definition 2.9). We simply need to include a label in the Prolog definition of each primitive operator in order to identify it (for instance, "#PROD" refers to the product primitive operator).

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

pri_prod(X,Y,Z,'#PROD.') :- Z is X * Y.
```



```
p(X)
{}
&prod(info(0.9,RULE1.),&godel(q(X1),@aver2(r(X1),s(X1))))
{X/X1}
&prod(info(0.9,RULE1.),&godel(info(0.8,RULE2.),@aver2(r(a),s(a))))
{X1/a,X/a}
&prod(info(0.9,RULE1.),&godel(info(0.8,RULE2.),@aver2(info(0.7,RULE3.),s(a))))
{X3/a,X1/a,X/a}
&prod(info(0.9,RULE1.),&godel(info(0.8,RULE2.),@aver2(info(0.7,RULE3.),info(0.5,RULE4.))))
{X4/a,X3/a,X1/a,X/a}
info(0.7200000000000001,RULE1.RULE2.RULE3.RULE4.@AVER2.|GODEL.#MAX.|LUKA.#ADD.#MIN.@AVER.#ADD.#DIV.&GODEL.#MIN.&PROD.#PROD.)
{X4/a,X3/a,X1/a,X/a}
```

```
Introduce new Goal:
'p(X)'.

>> run.
[Truth_degree=info(0.7200000000000001,RULE1.RULE2.RULE3.RULE4.@AVER2.|GODEL.#MAX.|LUKA.#ADD.#MIN.@AVER.#ADD.#DIV.&GODEL.#MIN.&PROD.#PROD.),X=a]
```

Figure 14: Obtaining declarative traces on fuzzy computed answers

As shown in **Figure 14**, the result of executing again goal "p(X)" is:

```
>> 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]
```

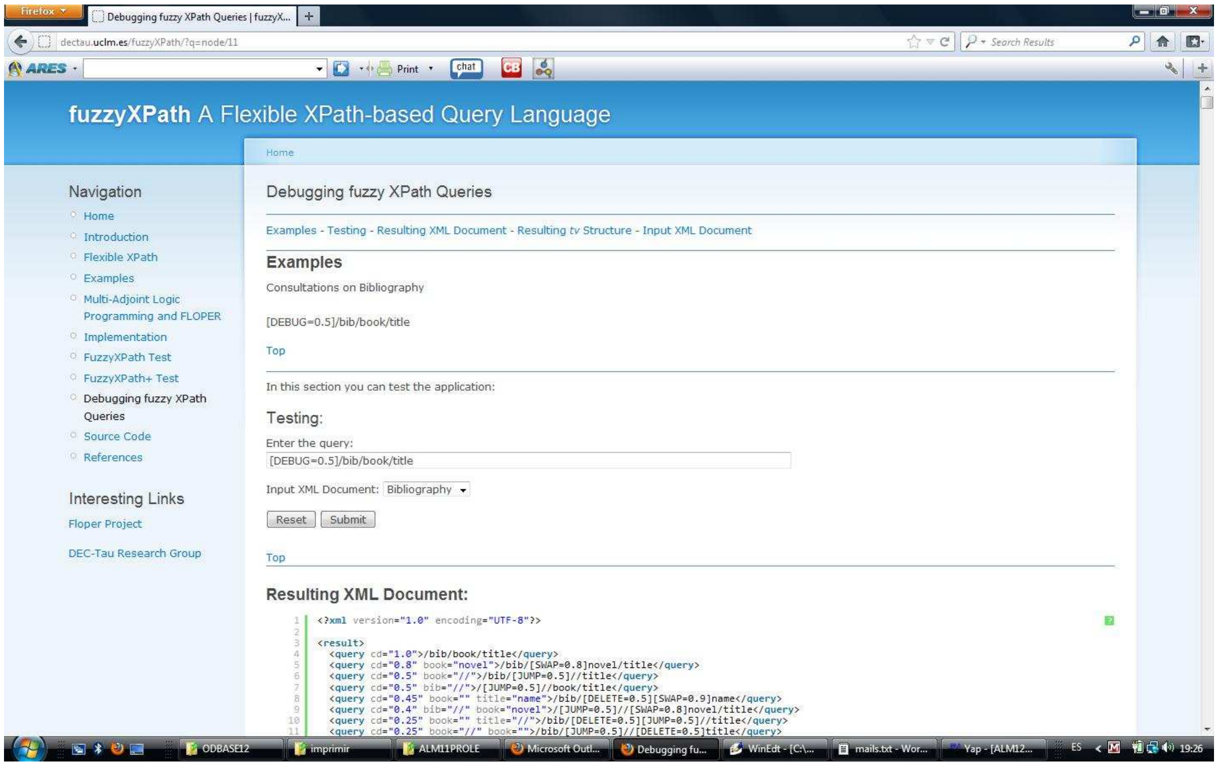
In this fuzzy computed answer we obtain both the truth value (0.72) and substitution ($X = a$) associated to our goal, but also the sequence of program rules exploited when applying admissible steps as well as the proper fuzzy connectives evaluated during the interpretive phase, also detailing the set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

To finish this section, we wish to mention the real-world application we have recently developed by using the $\mathcal{FLOPER}$ tool, where the key point is the use again of a lattice based on the Cartesian product of two previous lattices, but whose second component, instead of *strings*, considers *lists*. In [1, 2, 3], we present both an interpreter and a debugger for a fuzzy variant of the XPath language, which admits new commands with a fuzzy taste for flexibly accessing XML documents, automatically generating alternative queries to retrieve more information, etc. The application can be freely downloaded and even tested on-line (see **Figure 15**) through `http://dectau.uclm.es/fuzzyXPath/`. Moreover, in [5] and [47] we present our last applications recently developed with $\mathcal{FLOPER}$ which connect with the challenging fields of SAT/SMT and cloud computing, respectively.

5 Conclusions

In the recent past, several *fuzzy extensions* of the popular pure logic language Prolog have been designed in order to incorporate on its core new expressive resources for dealing with uncertainty in a natural way. However, real tools for putting in practice the power of such languages are not always available to a wide audience. To reduce this gap, the experience acquired in our research group regarding the design

Figure 15: Screen-shot of an on-line work session with the XPath debugger



fuzzyXPath A Flexible XPath-based Query Language

Navigation
- Home
- Introduction
- Flexible XPath
- Examples
- Multi-Adjoint Logic Programming and FLOPER
- Implementation
- FuzzyXPath Test
- FuzzyXPath+ Test
- Debugging fuzzy XPath Queries
- Source Code
- References

Interesting Links

Floper Project

DEC-Tau Research Group

Home

Debugging fuzzy XPath Queries

Examples - Testing - Resulting XML Document - Resulting *tv* Structure - Input XML Document

Examples

Consultations on Bibliography

[DEBUG=0.5]/bib/book/title

Top

In this section you can test the application:

Testing:

Enter the query: [DEBUG=0.5]/bib/book/title

Input XML Document: Bibliography

Top

Resulting XML Document:

```
<?xml version="1.0" encoding="UTF-8"?>

<result>
  <query cd="1.0">/bib/book/title</query>
  <query cd="0.8" book="novel">/bib/[SWAP=0.8]novel/title</query>
  <query cd="0.5" book="//">/bib/[JUMP=0.5]//title</query>
  <query cd="0.5" bib="//">/[JUMP=0.5]//book/title</query>
  <query cd="0.45" book="" title="name">/bib/[DELETE=0.5][SWAP=0.9]name</query>
  <query cd="0.4" bib="//" book="novel">/[JUMP=0.5]//[SWAP=0.8]novel/title</query>
  <query cd="0.25" book="" title="//">/bib/[DELETE=0.5][JUMP=0.5]//title</query>
  <query cd="0.25" book="//" book="">/bib/[JUMP=0.5]//[DELETE=0.5]title</query>
```

of techniques and methods based on fuzzy logic in close relationship with the so-called multi-adjoint logic programming approach, has motivated our interest for putting in practice all our developments around the design of the $\mathcal{FLOPER}$ environment, which offers comfortable resources for programmers trained in declarative languages. In a first stage, we have showed that our tool is able to execute MALP programs via a "transparent compilation process" to standard Prolog code which, to the best of our knowledge, constitutes the first initiative to put in practice the integral development of this kind of fuzzy programs. Going on deeper, we have next proposed a second compilation way which produces a low-level representation of the fuzzy code, thus enabling the possibility of drawing "derivation trees" (which offer too debugging/tracing tasks) and opening the door to new program manipulation techniques such as fold/unfold-based program optimization, program specialization by partial evaluation, etc... in which we are working nowadays after being formally matured in our research group. Moreover, our tool offers the possibility of managing in a single pack, i.e. "project", fuzzy programs with its associated multi-adjoint lattices modeled in Prolog, together with additional clauses and scripts helping to manage and increase the power of this programming tool. Our philosophy is to friendly connect this fuzzy framework with Prolog programmers since our system (which can be easily installed on top of many Prolog platforms) translates the fuzzy code to classical clauses, admits the definition of sophisticated lattices for modeling advanced notions of truth degrees collecting proof traces on fuzzy computed answers and simplifies the implementation of modern, real-world applications where fuzzy logic plays an important role. As an example, our system has recently served us for developing a real-world application devoted to the flexible management of XML documents by means of a fuzzy variant of the popular XPath language. We are nowadays introducing new *thresholding* techniques inspired by [16, 15] for addressing in our framework the well-known "top-k ranking problem" (i.e. determining the top k answers to a query without computing the -usually wider, possibly infinite- whole set of solutions [6, 8, 12, 9, 27, 45, 23, 26, 42, 10, 13]).

Acknowledgements

This work was supported by the EU (FEDER), and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grant TIN2013-45732-C4-2-P.

References

[1] J.M. Almendros-Jiménez, A. Luna, and G. Moreno. Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language. *Electronic Notes on Theoretical Computer Science, ENTCS*, 282:3–18, 2012.

[2] J.M. Almendros-Jiménez, A. Luna, and G. Moreno. Annotating Fuzzy Chance Degrees when Debugging Xpath Queries. In *Advances in Computational Intelligence - Proc of the 12th International Work-Conference on Artificial Neural Networks, IWANN 2013 (Special Session on Fuzzy Logic and Soft Computing Application), Tenerife, Spain, June 12-14*, pages 300–311. Springer Verlag, LNCS 7903, Part II, 2013.

[3] J.M. Almendros-Jiménez, A. Luna, G. Moreno, and C. Vázquez. Analyzing Fuzzy Logic Computations with Fuzzy XPath. In Lars Åke Fredlund and Laura M. Castro, editors, *Actas de las XIII Jornadas sobre Programación y Lenguajes, PROLE'13, Jornadas SISTEDES, Madrid, Spain, September 18-20*, pages 136–150 ("work in progress" track, extended version submitted to ECEASST). Universidad Complutense de Madrid (ISBN: 978-84-695-8331-9), 2013.

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

[5] M. Bofill, G. Moreno, C. Vázquez, and M. Villaret. Automatic Proving of Fuzzy Formulae with Fuzzy Logic Programming and SMT. In Lars Åke Fredlund and Laura M. Castro, editors, *Actas de las XIII Jornadas sobre Programación y Lenguajes, PROLE'13, Jornadas SISTEDES, Madrid, Spain, September 18-20*, pages 151–165 ("work in progress" track, extended version submitted to ECEASST). Universidad Complutense de Madrid (ISBN: 978-84-695-8331-9), 2013.

[6] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.

[7] R. Caballero, M. Rodríguez, and C. A. Romero. A transformation-based implementation for clp with qualification and proximity. *CoRR*, abs/1009.1976, 2010.

[8] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD Conference*, pages 346–357. ACM, 2002.

[9] Surajit Chaudhuri, Luis Gravano, and Amélie Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Trans. Knowl. Data Eng.*, 16(8):992–1009, 2004.

[10] Alan Eckhardt, Tomás Horváth, and Peter Vojtás. Learning different user profile annotated rules for fuzzy preference top-k querying. In Prade and Subrahmanian [41], pages 116–130.

[11] 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.

[12] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3):207–221, 2004.

[13] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[14] 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.

[15] P. Julián, J. Medina, P.J. Morcillo, G. Moreno, and M. Ojeda-Aciego. An unfolding-based preprocess for reinforcing thresholds in fuzzy tabulation. In Ignacio Rojas, Gonzalo Joya Caparrós, and Joan Cabestany, editors, *IWANN (1)*, volume 7902 of *Lecture Notes in Computer Science*, pages 647–655. Springer, 2013.

[16] 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.

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

[18] 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.

[19] P. Julián, G. Moreno, and J. Penabad. Measuring the interpretive cost in fuzzy logic computations. *Applications of Fuzzy Sets Theory, Proc. of 7th International Workshop on Fuzzy Logic and Applications, WILF'07, Springer-Verlag, LNAI*, 4578:28–36, 2007.

[20] P. Julián, G. Moreno, and J. Penabad. On the declarative semantics of multi-adjoint logic programs. *Proc. of 10th International Work-Conference on Artificial Neural Networks, IWANN'09, Springer-Verlag, LNCS*, 5517:253–260, 2009.

[21] 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.

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

[23] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. Ranksql: Query algebra and optimization for relational top-k queries. In Fatma Özcan, editor, *SIGMOD Conference*, pages 131–142. ACM, 2005.

[24] D. Li and D. Liu. *A fuzzy Prolog database system.* John Wiley & Sons, Inc., 1990.

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

[26] Thomas Lukasiewicz and Umberto Straccia. Top-k retrieval in description logic programs under vagueness for the semantic web. In Prade and Subrahmanian [41], pages 16–30.

[27] Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[28] 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.

[29] J. Medina, M. Ojeda-Aciego, and P. Vojtáš. A procedural semantics for multi-adjoint logic programming. *Progress in Artificial Intelligence, EPIA'01, Springer-Verlag, LNAI*, 2258(1):290–297, 2001.

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

[31] Pedro J. Morcillo, Ginés Moreno, Jaime Penabad, and Carlos Vázquez. Dedekind-macneille completion and cartesian product of multi-adjoint lattices. *International Journal of Computer Mathematics*, 89(13-14):1742–1752, 2012.

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

[33] P.J. Morcillo and G. Moreno. Modeling interpretive steps in fuzzy logic computations. *Proc. of the 8th International Workshop on Fuzzy Logic and Applications, WILF'09, Springer Verlag, LNAI*, 5571:44–51, 2009.

[34] P.J. Morcillo and G. Moreno. On cost estimations for executing fuzzy logic programs. *Proc. of the 2009 Int. Conference on Artificial Intelligence, ICAI'09, CSREA Press*, pages 217–223, 2009.

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

[36] P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. Modeling interpretive steps into the FLOPER environment. *Proc. of the 2010 Int. Conference on Artificial Intelligence, ICAI'10, CSREA Press*, pages 16–22, 2010.

[37] P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. Declarative Traces into Fuzzy Computed Answers. In N. Bassiliades et al., editor, *Proc. of 5th Int. Symposium on Rules: Research Based, Industry Focused, RuleML'11*, pages 170–185. Springer Verlag, LNCS 6826, 2011.

[38] P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. String-based Multi-adjoint Lattices for Tracing Fuzzy Logic Computations. *ECEASST*, 55:17, 2012.

[39] S. Muñoz-Hernandez, V. Pablos-Ceruelo, and H. Strass. Rfuzzy: Syntax, semantics and implementation details of a simple and expressive fuzzy tool over prolog. *Information Sciences*, 181(10):1951 – 1970, 2011.

[40] H.T. Nguyen and E.A. Walker. *A First Course in Fuzzy Logic*. Chapman & Hall/CRC, Boca Ratón, Florida, 2000.

[41] Henri Prade and V. S. Subrahmanian, editors. *Scalable Uncertainty Management, First International Conference, SUM 2007, Washington, DC, USA, October 10-12, 2007, Proceedings*, volume 4772 of *Lecture Notes in Computer Science*. Springer, 2007.

[42] Christopher Re, Nilesh N. Dalvi, and Dan Suciu. Efficient top-k query evaluation on probabilistic data. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *ICDE*, pages 886–895. IEEE, 2007.

[43] M. Rodríguez and C. A. Romero. A declarative semantics for clp with qualification and proximity. *Theory and Practice of Logic Programming*, 10(4-6):627–642, 2010.

[44] 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.

[45] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. In Ricardo A. Baeza-Yates, Nivio Ziviani, Gary Marchionini, Alistair Moffat, and John Tait, editors, *SIGIR*, pages 242–249. ACM, 2005.

[46] C. Vaucheret, S. Guadarrama, and S. Muñoz. Fuzzy prolog: A simple general implementation using *clp*(r). In M. Baaz and A. Voronkov, editors, *Proc. of Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2002*, pages 450–463. Springer Verlag, LNAI 2514, 2002.

[47] C. Vázquez, L. Tomás, G. Moreno, and J. Tordsson. A fuzzy approach to cloud admission control for safe overbooking. In Francesco Masulli, Gabriella Pasi, and Ronald Yager, editors, *Proc. of 10th International Workshop on Fuzzy Logic and Applications, WILF 2013, Genoa, Italy, November 7-10*, pages 212–225. Springer Verlag, LNAI 8256, 2013.