String-based Multi-adjoint Lattices for Tracing Fuzzy Logic Computations

Pedro J. Morcillo[1], **Ginés Moreno**[1], **Jaime Penabad**[1] **and Carlos Vázquez**[1]

[1] PedroJ.Morcillo@alu.uclm.es Gines.Moreno@uclm.es
Jaime.Penabad@uclm.es Carlos.Vazquez@alu.uclm.es
Faculty of Computer Science Engineering
University of Castilla-La Mancha
02071 Albacete (Spain)

Abstract: Classically, most programming languages use in a predefined way the notion of "string" as an standard *data structure* for a comfortable management of arbitrary sequences of characters. However, in this paper we assign a different role to this concept: here we are concerned with fuzzy logic programming, a somehow recent paradigm trying to introduce fuzzy logic into logic programming. In this setting, the mathematical concept of *multi-adjoint lattice* has been successfully exploited into the so-called *Multi-adjoint Logic Programming* approach, MALP in brief, for modeling flexible notions of truth-degrees beyond the simpler case of true and false. Our main goal points out not only our formal proof verifying that string-based lattices accomplish with the so-called *multi-adjoint property* (as well as its Cartesian product with similar structures), but also its correspondence with interesting debugging tasks into the FLOPER system (from *"Fuzzy LOgic Programming Environment for Research"*) developed in our research group.

Keywords: Cartesian Product of Multi-adjoint Lattices; Fuzzy (Multi-adjoint) Logic Programming; Declarative Debugging

1 Introduction

In essence, the notion of multi-adjoint lattice considers a *carrier* set L (whose elements verify a concrete ordering $\leq$) equipped with a set of connectives like implications, conjunctions, disjunctions and other *hybrid operators* (not always belonging to an standard taxonomy) with the particularity that for each implication symbol there exists its adjoint conjunction used for modeling the *modus ponens* inference rule in a fuzzy logic setting. For instance, some adjoint pairs -i.e. conjunctors and implications- in the lattice $([0,1],\leq)$ are presented in the following paragraph (from now on, this lattice will be called $\mathcal{V}$ along this paper), where labels `L`, `G` and `P` mean respectively *Łukasiewicz logic*, *Gödel intuitionistic logic* and *product logic* (with different capabilities for modeling *pessimist*, *optimist* and *realistic scenarios*, respectively):

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 177-191, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

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

Moreover, in the MALP framework [MOV04], each program has its own associated multi-adjoint lattice and each program rule (whose syntax, described in detail in Section 3, extends very significantly a Prolog clause[1]) is "weighted" with an element of L, whereas the components in its body are *linked* with connectives of the lattice. For instance, in the following propositional MALP program (where obviously $@_{\texttt{aver}}$ refers to the classical average operator):

$$
\begin{array}{llll}
p & \leftarrow_{\texttt{P}} & @_{\texttt{aver}}(q,r) & \textit{with} \quad 0.9\\
q & \leftarrow & & \textit{with} \quad 0.8\\
r & \leftarrow & & \textit{with} \quad 0.6
\end{array}
$$

the last two rules directly assign truth values 0.8 and 0.6 to propositional symbols q and r, respectively, and the execution of p using the first rule, simply consists in evaluating the expression "$\&_{\texttt{P}}(0.9,@_{\texttt{aver}}(0.8,0.6))$", which returns the final truth degree 0.63.

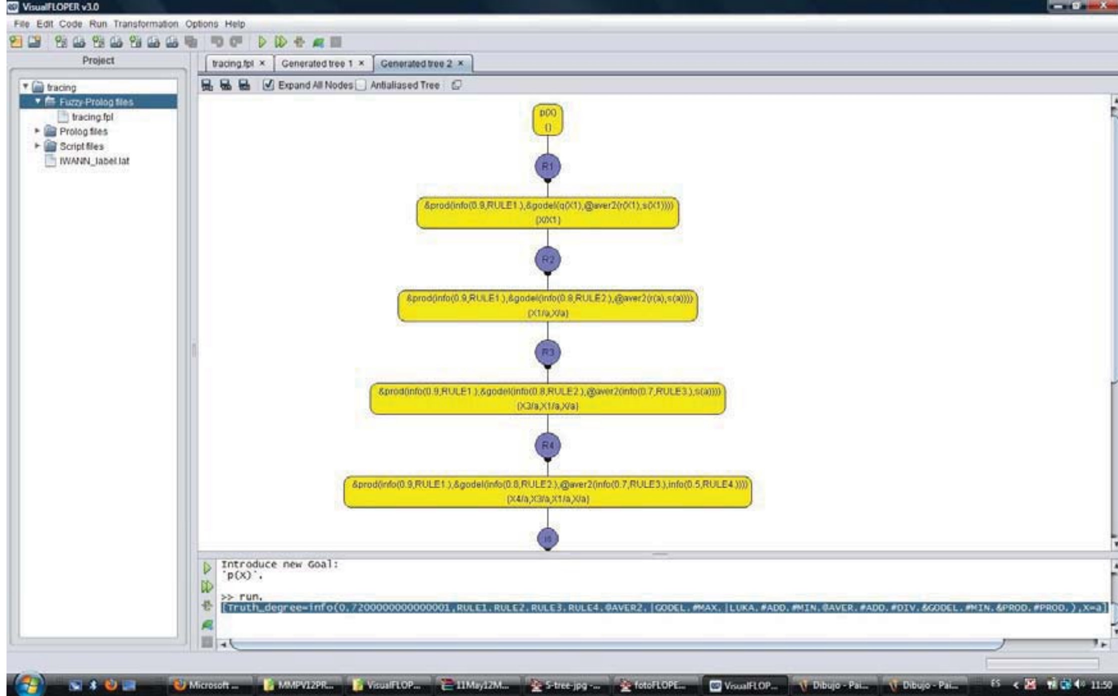
In the following section we describe the shape of the elements, ordering relation and behaviour of the connectives of new multi-adjoint lattices obtained by applying the Cartesian product to previous ones. The main application of such structures into the MALP framework is that it is very easy to attach to program rules and fuzzy connectives "labels" related not only with truth degrees, but also with "augmented information" very useful for designing further *debugging* tasks devoted to "document" proof procedures. Our work is inspired by [RR08, RR09], where authors use the so-called *qualification domain of weights* $\mathscr{W}$ for counting the number of computational steps performed along derivations: however, our proposed technique surpass such approach by providing deeper details on the nature of every fuzzy-logic evaluation step.

Moreover, in Section 3 we present the syntax and procedural semantics of the MALP framework, which exploits multi-adjoint lattices for modeling richer notions of truth degrees to be managed by fuzzy programs. Next, we present the FLOPER tool recently equipped with a graphical interface as shown in Figure 1 [MMPV10, MMPV11c, MMPV11a] (please, visit the web page `http://dectau.uclm.es/floper/`), which currently is successfully used for compiling (to standard Prolog code), executing and debugging MALP programs in a safe way and it is ready for being extended in the near future with powerful transformation and optimization techniques designed in our research group in the recent past [JMP05, GM08].

After describing some guidelines for easily managing multi-adjoint lattices expressed by means of Prolog clauses into the FLOPER system, in Section 4 we propose a sophisticated kind of lattices based on the Cartesian product of previous lattices (based on strings) capable for taking into account details on declarative traces, such as the sequence of computations (regarding program rules, fuzzy connectives and primitive operators) needed for evaluating a given goal. Finally, in Section 5 we give our conclusions and provide some lines for future work.

---
[1] We assume familiarity with pure Logic Programming [Llo87, JA07] and its most popular language Prolog.

Figure 1: Screen-shot of a work session with FLOPER.



2 String-based Multi-adjoint Lattices and Cartesian Product

In this section we focus on the theoretical results which guarantee that a lattice based on strings is also a multi-adjoint lattice, as well as its Cartesian product with any other multi-adjoint lattice (this kind of sophisticated lattices can be associated to MALP programs in order to report at execution time, a detailed description of the computational steps performed for reaching solutions). We start this section with the formal definition of multi-adjoint lattice.

Definition 1 Let $(L, \leq)$ be a lattice. A *multi-adjoint lattice* is a tuple $(L, \leq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$ such that:

i) $(L, \leq)$ is a complete lattice, namely, $\forall S \subset L, S \neq \emptyset, \exists\, inf(S), sup(S)$[2].

ii) $(\&_i, \leftarrow_i)$ is an *adjoint pair* in $(L, \leq)$, i.e.:

1) $\&_i$ is increasing in both arguments, for all i, $i = 1, \ldots, n$.

2) $\leftarrow_i$ is increasing in the first argument and decreasing in the second, for all i.

3) $x \leq (y \leftarrow_i z)$ if and only if $(x \&_i z) \leq y$, for any $x, y, z \in L$ (adjoint property).

iii) $\top \&_i v = v \&_i \top = v$ for all $v \in L, i = 1, \ldots, n$, where $\top = sup(L)$.

[2] Then, it is a bounded lattice, i.e. it has bottom and top elements, denoted by $\bot$ and $\top$, respectively.

This last condition, called *adjoint property*, is the most important feature of the framework. From now, we are going to focus on the classical notion of Cartesian product applied on these structures, which necessarily returns objects inheriting the required properties of such lattices.

Theorem 1 *If $L_1,\dots,L_n$ are multi-adjoint lattices, then its Cartesian product $L = L_1 \times \cdots \times L_n$ is also a multi-adjoint lattice.*

Proof. In order to simplify our sketch but without loss 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. $L = L_1 \times L_2$ has lattice structure with an order 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: $(x_1, y_1)\&(x_2, y_2) \triangleq (x_1 \&_1 x_2, y_1 \&_2 y_2)$ and $(x_1, y_1) \leftarrow (x_2, y_2) \triangleq (x_1 \leftarrow_1 x_2, y_1 \leftarrow_2 y_2)$, 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, etc.) in the Cartesian product $L_1 \times L_2$ from the corresponding pairs of operators defined in both lattices L_1 and L_2. □

Moreover, if we are interested in knowing more detailed data about the set of program rules and connective definitions evaluated for obtaining each solution then it will be mandatory to use a new lattice $\mathscr{S}$ based on *strings* or *labels* (i.e., sequences of characters) for generating the Cartesian product $\mathscr{V} \times \mathscr{S}$. In order to achieve our purposes, we firstly must show not only that $\mathscr{S}$ is a multi-adjoint lattice, but also that the concatenation operation of strings, usually called *append* in many programming languages, plays the role of an adjoint conjunction in such lattice (this last condition is required by practical aspects which are explained in Section 4).

When trying to solve both problems, we have analyzed several alternatives for establishing ordering relations among the elements of $\mathscr{S}$, such as the classical lexicographic ordering typically used for sorting words in dictionaries, or those ones based on prefixes, sub-strings, etc., (for instance 'ab' and 'bc' are respectively a prefix and a sub-string of 'abcd'). Unfortunately we have observed that it is never possible to prove that *append* acts as a t-norm.

However, we have recently conceived an alternative, second way for granting that $\mathscr{S}$ is really a multi-adjoint lattice, which definitely solves our problems. The clever point is to "view" each string as a natural number by associating to each character its corresponding ASCII code. So, it is possible to establish a bijective mapping $[\,]$ with $\mathbb{N}$.

Let be $A = \{a_0, \dots, a_{n-1}\}$ a set, called alphabet, whose elements are symbols. A string s over A is a finite sequence of elements of this set, that is, $s = a_1 \dots a_m$, where $a_i \in A$, $i = 1, \dots, m$. The set of all strings over A, denoted by S, is defined as $S = \cup_{k \in \mathbb{N}} A^k$. The definition of S guarantees its numerable character, that is, S contains a (numerable) infinite number of strings like $a_1 \dots a_m$ formed by elements of A. Although the mentioned numerable character is obtained from well

known results of set theory, in Theorem 2 we will justify it by formalizing a bijection (and its reverse function) from S to $\mathbb{N}$ which will be relevant in the multi-adjoint scope.

Each element of A^k is a string of k elements (with length k) that can be viewed as words; in this case, S could be interpreted as the set of all words with finite length. This interpretation is attractive since each formal language with alphabet A is the set of formulae constructed with elements or words of S that are, also, constrained to the syntactic rules established by that language. Any language with alphabet A would be, accordingly, a subset of S.

Moreover, in set S we define the concatenation, denoted by $\cdot$, as the internal operation

$$A^p \times A^q \to A^{p+q}$$

$$(s,t) \mapsto s \cdot t$$

such that, given $s = a_1 \dots a_p, t = b_1 \dots b_q$, then $s \cdot t = a_1 \dots a_p b_1 \dots b_q$.

Let be the primitive functions $cod : A \to Im(cod)$ and $asc : Im(cod) \to A$, such that $cod(a_i) = i$ and $asc(i) = a_i$. It is trivial to proof that cod is the inverse of function asc and reciprocally. Symbol $''$ represents the empty (with length 0) string. We define the map $[\,] : S \to \mathbb{N}$ through rules R_1 and R_2:

$$\begin{aligned} R_1 :&\quad [''] = 0 \\ R_2 :&\quad [a_1 \dots a_m] = (cod(a_1)+1)n^{m-1} + \dots + (cod(a_m)+1)n^0 \end{aligned}$$

where $a_i \in A, \forall i$, and $a_1 \dots a_m \in S$. Rule R_2 can be rewritten as:

$$R_2^* : \quad [s.a] = [s]n + cod(a) + 1$$

where $s.a$ indicates a string obtained after adding the symbol a at the end of string s, $s.a = a_1 \dots a_m.a = a_1 \dots a_m a$.

In order to illustrate this mapping, consider the alphabet $\{a,b,c\}$ and the string $s = cab$, where $[cab] = (cod(c)+1)3^2 + (cod(a)+1)3 + (cod(b)+1) = 3*9+1*3+2 = 32$.

Theorem 2 *The map $[\,]$, defined by R_1 and R_2 (or R_2^*), is bijective.*

Proof. Indeed, it is map since each string $s \in S$ is associated with a unique natural number. In order to proof the bijective character of $[\,]$, we will demonstrate that its inverse function is the map $<>: \mathbb{N} \to S$, defined as:

$$\begin{aligned} &< 0 > = '' \\ &< m > = < \lfloor (m-1)/n \rfloor > .asc((m-1)\%n) \end{aligned}$$

where $\lfloor\rfloor : \mathbb{R} \to \mathbb{N}$ is the floor function, so $\lfloor (m-1)/n \rfloor$ is the floor of the quotient, and $(m-1)\%n$ is the remainder modulo n (number of elements of A) of the integer $m-1$.

We will see that compositions $(<> \circ [\,]) : S \to S$ and $([\,] \circ <>) : \mathbb{N} \to \mathbb{N}$ coincide with the identity map (id_S and $id_{\mathbb{N}}$, respectively), which justifies that $[\,]$ is the inverse of $<>$, and reciprocally. Firstly, we prove that $(<> \circ [\,])(s) = s, \forall s \in S$.

1. $(<> \circ [\,])('') = < [''] > = < 0 > = ''$

2. $(<>\circ[\,])(s.a) =$
$<[s.a]>=<[s]n+cod(a)+1>=$
$<\lfloor([s]n+cod(a)+1-1)/n\rfloor>.asc(([s]n+cod(a)+1-1)\%n) =$
$<\lfloor([s]n+cod(a))/n\rfloor>.asc(([s]n+cod(a))\%n) =$
$<\lfloor[s]n/n\rfloor>.asc(cod(a)\%n) =<\lfloor[s]\rfloor>.asc(cod(a)) =$
$<[s]>.a$

Hence, $<[s.a]>=<[s]>.a$, that is, if $s = a_1 \dots a_m$, then $<[a_1 \dots a_m a]>=<[a_1 \dots a_m]>.a = \cdots =<[''] > .a_1.\dots.a_m.a = s.a = id_S(s.a)$, where $id_S : S \to S$ is the identity map of S.

Secondly, we will demonstrate that $([]\circ<>)(j) = j, \forall j \in \mathbb{N}$:

1. $([]\circ<>)(0) = [<0>] = [''] = 0$

Since j can be expressed as $xn+y$, where $x, y \in \mathbb{N}, 0 < y \leq n$, if $j > 0$, we have

2. $([\,]\circ<>)(j) =$
$([\,]\circ<>)(xn+y) = [<xn+y>] =$
$[<\lfloor(xn+y-1)/n\rfloor>.asc((xn+y-1)\%n)] =$
$[<\lfloor(xn+y-1)/n\rfloor>]n+cod(asc((xn+y-1)\%n))+1 =$
$[<\lfloor xn/n\rfloor>]n+cod(asc(y-1))+1 =$
$[<x>]n+(y-1)+1 =$
$[<x>]n+y$

Therefore, $[<xn+y>] = [<x>]n+y, 0 < y \leq n$. That is, j can be expressed as $j = y_1 n^{m-1} + \cdots + y_{m-1} n^1 + y$, with $y_1, \dots, y_{m-1} \in \{1, \dots, n\}$, so $[<j>] = [<y_1 n^{m-1} + \cdots + y_{m-1} n^1 + y>] = [<y_1 n^{m-2} + \cdots + y_{m-1} n^0>]n + y = \cdots = [<0>]n + y_1 n^{m-1} + \cdots + y_{m-1} n^1 + y = j$.

Consequently, if $[\,]$ is the inverse function of $<>$ (and reciprocally) both are bijective functions, which proves our desired result. □

In Example 1 we illustrate these maps with a reduced alphabet, while Example 2 works in a real-world setting, giving an idea of the usefulness of this approach.

Example 1 *Consider the alphabet* $A = \{a, b, c\}$. *Then, the string associated to the number* 32 *is* $<32>=<\lfloor 31/3\rfloor>.asc(32\%3) =<10>.asc(1) =<\lfloor 9/3\rfloor>.asc(9\%3).b =<3>.asc(0).b =<\lfloor 2/3\rfloor>.asc(2\%3).a.b =<0>.asc(2).a.b ='' .c.a.b = cab$

Example 2 *Consider the following strings (using the ASCII code as alphabet):* $s = sea$ *and* $t = son$. *We will apply the* append *operation on them and obtain its associated number.*

- $[s] =$
$[sea] = (cod(s)-1)128^2 + (cod(e)-1)128 + (cod(a)-1) =$
$(115-1)128^2 + (101-1)128 + (97-1) =$
1880672

- $[t] =$
$[son] = (cod(s)-1)128^2 + (cod(o)-1)128 + (cod(n)-1) =$
$(115-1)128^2 + (111-1)128 + (110-1) =$
1881965

- $[s \cdot t] =$
$[season] = (cod(s)-1)128^5 + (cod(e)-1)128^4 + (cod(a)-1)128^3 + (cod(s)-1)128^2 + (cod(o)-1)128 + (cod(n)-1) =$
$(115-1)128^5 + (101-1)128^4 + (97-1)128^3 + (115-1)128^2 + (111-1)128 + 109 =$
3944056928109

Starting from the reverse usual order $(\mathbb{N}, \leq)$, we define an ordering relation in S that compares strings $s, t \in S$ by

$$s \leq t \iff [s] \leq [t]$$

so the supreme element of S is the empty string, $''$. The application of append over strings *sea* and *son* from the Example 2, which is $s \cdot t = season$, is less than both s and t, that is (by the definition of S), $[s \cdot t] \leq [s]$ and $[s \cdot t] \leq [t]$. As a result of Theorem 2, we have that S is bijective with $\mathbb{N}$. Therefore, the completion (see [MMPV11b]) of S, which is $\mathscr{S} = S \cup \{inf(S)\}$, is bijective with the completion of $\mathbb{N}$, $(\mathbb{N} \cup \{inf(\mathbb{N})\})$, expressed also as $\mathscr{W}$ [RR08, RR09]. So, since $(\mathscr{W}, \leq) = (\mathbb{N} \cup \{inf(\mathbb{N})\}, \leq)$ is a *multi-adjoint lattice*, then $\mathscr{S}$ inherits the same property[3]. After showing that $\mathscr{S}$ is a multi-adjoint lattice via the mapping bijection established with the multi-adjoint lattice $\mathscr{W}$, and before illustrating the benefits that the Cartesian product $\mathscr{V} \times \mathscr{S}$, among others, might play in several software engineering tasks, it is mandatory to explain in the following two sections some details of the MALP language and its associated FLOPER tool.

3 Multi-adjoint Logic Programming and FLOPER

This section summarizes the main features of multi-adjoint logic programming as illustrated in Figure 2 (see [MOV04, JMP09] for a complete formulation of this framework). We work with a first order language, $\mathscr{L}$, containing variables, constants, function symbols, predicate symbols, and several (arbitrary) connectives to increase language expressiveness: implication connectives (denoted by $\leftarrow_1, \leftarrow_2, \ldots$); conjunctive connectives $(\wedge_1, \wedge_2, \ldots)$ adjoint conjunctions $(\&_1, \&_2, \ldots)$, disjunctive connectives $(\vee_1, \vee_2, \ldots)$, and hybrid operators called "aggregators" (usually denoted by $@_1, @_2, \ldots$). 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 connective $[\![@]\!] : L^n \to L$ is required to be monotonous and fulfills $[\![@]\!](\top, \ldots, \top) = \top$, $[\![@]\!](\bot, \ldots, \bot) = \bot$.

Additionally, our language $\mathscr{L}$ contains the values of a multi-adjoint lattice $(L, \leq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$, equipped with a collection of *adjoint pairs* $(\leftarrow_i, \&_i)$ as detailed in previous sections. In general, L may be the carrier of any complete bounded lattice where a L-expression is a well-formed expression composed by values and connectives of L, as well as variable symbols and *primitive operators* (i.e., arithmetic symbols such as $*, +, min$, etc.). In what follows, we assume that the truth function of any connective @ in L is given by its corresponding *connective definition*, that is, an equation of the form $@(x_1, \ldots, x_n) \triangleq E$, where E is a L-expression not containing variable symbols apart from $x_1, \ldots, x_n$. See for instance, the classical set of adjoint pairs (conjunctors and implications of *Łukasiewicz logic*, *Gödel intuitionistic logic* and *product logic*) in $([0,1], \leq)$ defined at the beginning of this paper in Section 1.

[3] It is easy to prove that $\&_{\text{append}}$ is really an adjoint conjunction in lattice $\mathscr{S}$.

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

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

Definition 2 (Admissible Step) Let $\mathscr{Q}$ be a goal and let σ be a substitution. The pair $\langle \mathscr{Q}; \sigma \rangle$ is a *state* and we denote by $\mathscr{E}$ the set of states. Given a program $\mathscr{P}$, an *admissible computation* is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathscr{E} \times \mathscr{E})$ is the smallest relation satisfying the following *admissible rules* (where we always consider that A is the selected atom in $\mathscr{Q}$ and $mgu(E)$ denotes the *most general unifier* of an equation set E):

1) $\langle \mathscr{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathscr{Q}[A/v \&_i \mathscr{B}])\theta; \sigma\theta \rangle$, if $\theta = mgu(\{H = A\})$ and $\langle H \leftarrow_i \mathscr{B}; v \rangle$ in $\mathscr{P}$.
2) $\langle \mathscr{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathscr{Q}[A/v])\theta; \sigma\theta \rangle$, if $\theta = mgu(\{H = A\})$ and $\langle H \leftarrow; v \rangle$ in $\mathscr{P}$.

As usual, rules are taken renamed apart. We shall use the symbols $\rightarrow_{AS1}$ and $\rightarrow_{AS2}$ to distinguish between computation steps performed by applying one of the specific admissible rules. Also, the application of a rule on a step will be annotated as a superscript of the $\rightarrow_{AS}$ symbol.

Definition 3 Let $\mathscr{P}$ be a program, $\mathscr{Q}$ a goal and id the empty substitution. An *admissible derivation* is a sequence $\langle \mathscr{Q}; id \rangle \rightarrow_{AS} \cdots \rightarrow_{AS} \langle \mathscr{Q}'; \theta \rangle$. When $\mathscr{Q}'$ is a formula not containing atoms (i.e., a L-expression), the pair $\langle \mathscr{Q}'; \sigma \rangle$, where $\sigma = \theta[\mathscr{V}ar(\mathscr{Q})]$, is called an *admissible computed answer* (a.c.a.) for that derivation.

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

Definition 4 (Interpretive Step) Let $\mathscr{P}$ be a program, $\mathscr{Q}$ a goal and σ a substitution. Assume that $[\![@]\!]$ is the truth function of connective @ in the lattice $(L, \leq)$ associated to $\mathscr{P}$, such that, for values $r_1, \ldots, r_n, r_{n+1} \in L$, we have that $[\![@]\!](r_1, \ldots, r_n) = r_{n+1}$. Then, we formalize the notion of *interpretive computation* as a state transition system, whose transition relation $\rightarrow_{IS} \subseteq (\mathscr{E} \times \mathscr{E})$ is defined as the least one satisfying: $\langle \mathscr{Q}[@(r_1, \ldots, r_n)]; \sigma \rangle \rightarrow_{IS} \langle \mathscr{Q}[@(r_1, \ldots, r_n)/r_{n+1}]; \sigma \rangle$. An *interpretive derivation* is a sequence $\langle \mathscr{Q}; \sigma \rangle \rightarrow_{IS} \cdots \rightarrow_{IS} \langle \mathscr{Q}'; \sigma \rangle$. If $\mathscr{Q}' = r \in L$, being $(L, \leq)$ the lattice associated to $\mathscr{P}$, then $\langle r; \sigma \rangle$ is called a *fuzzy computed answer* (f.c.a.) for that derivation.

From now on, we proceed with more practical aspects regarding multi-adjoint lattices and implementation issues. The parser of our FLOPER tool [MMPV10, MMPV11c] has been imple-

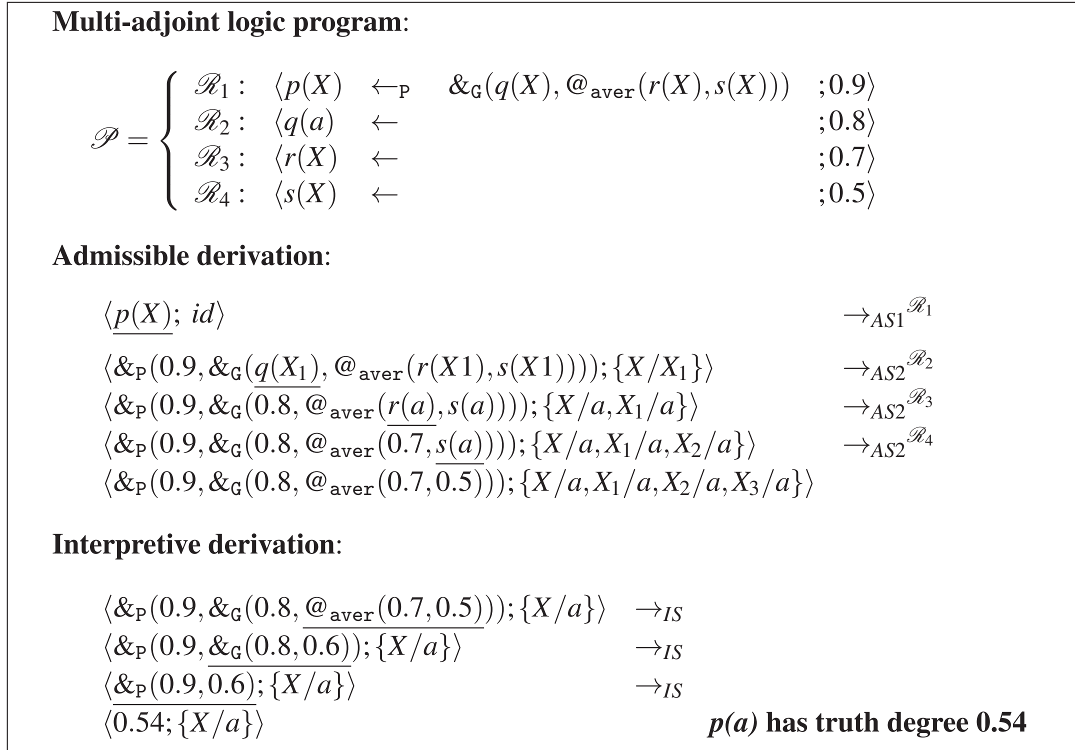


Figure 2: MALP program $\mathcal{P}$ with admissible/interpretive derivations for goal $p(X)$.

mented by using the Prolog language. Once the application is loaded inside a Prolog interpreter, it shows a menu which includes options for loading/compiling, parsing, listing and saving MALP programs, as well as for executing/debugging fuzzy goals. Moreover, in [MMPV10] we explain that FLOPER has been recently equipped with new options, called "`lat`" and "`show`", for allowing the possibility of respectively changing and displaying the multi-adjoint lattice associated to a given program.

A very easy way to model truth-degree lattices for being included into the FLOPER tool is also described in [MMPV10], according the following guidelines. All relevant components of each lattice are encapsulated inside a Prolog file which must necessarily contain the definitions of a minimal set of predicates defining the set of valid elements (including special mentions to the "`top`" and "`bottom`" ones), the full or partial ordering established among them, as well as the repertoire of fuzzy connectives which can be used for their subsequent manipulation. In order to simplify our explanation, assume that file "bool.pl" refers to the simplest notion of (a binary) adjoint lattice, thus implementing the following set of predicates:

• `member/1` which is satisfied when being called with a parameter representing a valid truth degree. For instance, in the Boolean case, this predicate can be simply modeled by the Prolog facts `member(0).` and `member(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).`

```
member(X) :- number(X),0=<X,X=<1.                      bot(0).
leq(X,Y) :- X=<Y.                                      top(1).

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

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

agr_aver(X,Y,Z) :- pri_add(X,Y,U),pri_div(U,2,Z).
agr_aver2(X,Y,Z):- or_godel(X,Y,Z1),or_luka(X,Y,Z2),
                   agr_aver(Z1,Z2,Z).

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

Figure 3: Prolog code for representing the multi-adjoint lattice $\mathcal{V}$.

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

The reader can easily check that the use of lattice "bool.pl" when working with MALP programs whose rules have the form: "$H \leftarrow_{bool} \&_{bool}(B_1,\ldots,B_n)$ *with* 1", being H and B_i typical atoms, successfully mimics the behaviour of classical Prolog programs where clauses accomplish with the shape "$H\ :-\ B_1,\ldots,B_n$". As a novelty in the fuzzy setting, when evaluating goals according to the procedural semantics described in Section 3, each output will contain the corresponding Prolog's substitution (i.e., the *crisp* notion of computed answer obtained by means of classical SLD-resolution) together with the maximum truth degree 1.

As shown in Figure 3, it is also possible to describe by means of Prolog clauses the more flexible lattice $\mathcal{V}$ for working with truth degrees in the infinite space of the real numbers between 0 and 1, allowing too the possibility of using conjunction and disjunction operators recasted from the three typical fuzzy logics proposals described before, as well as two useful descriptions for the hybrid aggregator *average*.

4 Declarative Traces via Cartesian Product of Lattices

As detailed in [MMPV10, MMPV11c], our parser has been implemented by using the classical DCG's (*Definite Clause Grammars*) resource of the Prolog language, since it is a convenient notation for expressing grammar rules. As already commented in the previous section, once the application is loaded inside any Prolog interpreter, it shows a menu which includes options for loading, parsing, listing and saving fuzzy programs, as well as for executing fuzzy goals. All these actions are based in the translation of the fuzzy code into standard Prolog code. The key point is to extend each atom with an extra argument, called *truth variable* of the form "$_\mathtt{TV_i}$", which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. For instance, the first clause in our target program is translated into:

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

Moreover, the second clause in our target program in Figure 2, becomes into the pure Prolog fact "`q(a,0.8)`" while a fuzzy goal like "`p(X)`", is translated into the pure Prolog goal: "`p(X, Truth_degree)`" (note that the last truth degree variable is not anonymous now) for which the Prolog interpreter returns the desired fuzzy computed answer $[\mathtt{Truth_degree} = 0.54, \mathtt{X} = \mathtt{a}]$. The previous set of options suffices for running fuzzy programs (the "`run`" choice also uses the clauses contained in "num.pl", which represent the default lattice) where all internal computations (including compiling and executing) are pure Prolog derivations whereas inputs (fuzzy programs and goals) and outputs (fuzzy computed answers) have always a fuzzy taste, thus producing the illusion on the final user of being working with a purely fuzzy logic programming tool.

```
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 4: Lattice modeled in Prolog for representing the Cartesian product $\mathscr{V} \times \mathscr{S}$.

On the other hand, in the previous section we have explained that FLOPER is equipped with two new options, called "`lat`" and "`show`", for allowing the possibility of respectively changing and displaying the multi-adjoint lattice associated to a given program. Assume now that, instead of computing the average of two truth degrees, we prefer to compute the average between the results achieved after applying to both elements the disjunctions operators described by Gödel and Łukasiewicz, that is: $@_{\mathtt{aver_2}}(x_1,x_2) \triangleq @_{\mathtt{aver}}(\vee_{\mathtt{G}}(x_1,x_2), \vee_{\mathtt{L}}(x_1,x_2))$. See the corresponding

Prolog clause modeling such definition listed in Figure 3. Now, by selecting again the "run" option, the system would display the new solution: [Truth_degree = 0.72, X = a].

In what follows, we plan to exploit a much more powerful lattice to cope with more involved *debugging* details based on declarative traces beyond the simpler task described in [RR08, RR09] of counting the number of program rules used during the query-answering procedure (some guidelines can be found in our precedent works [MMPV11c, MMPV11a]). The elements of our intended lattice must contain two components, thus belonging to the Cartesian product $\mathscr{V} \times \mathscr{S}$ seen in Section 2, in order capture not only truth degrees, but also "labels" collecting information about the program rules, fuzzy connectives and primitive operators used for solving goals when executing programs. In order to be loaded into FLOPER, we need to define again the new lattice as a Prolog program, whose elements will be expressed now as data terms of the form "info(Fuzzy_Truth_Degree, Label)" as shown in Figure 4 (we simply show an incomplete but representative set of predicates).

Here, we see that when implementing for instance the conjunction operator of the Product Logic, in the second component of our extended notion of "truth degree", we have *appended* the labels of its arguments with the label '&PROD.' (see clauses defining and_prod, pri_app and append). Of course, in the fuzzy program to be run, we must also take into account the use of labels associated to the program rules. For instance, the set of rules in our example (where we use the complex version of average, i.e., @aver2 in the first rule) must have the form:

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

Now the reader can easily check that, after executing goal p(X), we obtain the desired fuzzy computed answer which includes a nice declarative trace collecting the exact sequence of "program-rules, fuzzy-connectives and primitive-operators" evaluated till finding the final solution, which has the following shape by using FLOPER:

```
>> 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 (i.e., 0.72) and substitution (that is, $X = a$) associated to our goal, but also the sequence of program rules exploited when applying admissible steps (RULE1, RULE2, RULE3 and RULE4, in this order) as well as the list of fuzzy connectives evaluated during the interpretive phase, also detailing the set of primitive operators (of the form #*label*) that they call: in our case, note that we have firstly evaluated aggregator @AVER2 (which calls to connectives |GODEL - defined in terms of the primitive operator #MAX -, |LUKA - which invokes the arithmetic operations #ADD and #MIN - and @AVER - expressed with the use of the primitive operators #ADD and #DIV -), then we need to evaluate the conjunction &GODEL (solved again via the arithmetic symbol #MIN) and the final exploited connective is &PROD (described in terms of the primitive operator #PROD).

$$\langle \underline{p(X)};\ id\rangle \quad \to_{AS1}{}^{\mathscr{R}_1}$$

$$\langle \&_{\mathtt{P}}(\mathrm{info}(0.9,\ \text{`}\mathtt{RULE1.}\text{'}), \&_{\mathtt{G}}(\underline{q(X_1)}, @_{\mathtt{aver2}}(r(X1), s(X1)))); \{X/X_1\}\rangle \quad \to_{AS2}{}^{\mathscr{R}_2}$$

$$\langle \&_{\mathtt{P}}(\mathrm{info}(0.9,\ \text{`}\mathtt{RULE1.}\text{'}), \&_{\mathtt{G}}(\mathrm{info}(0.8,\ \text{`}\mathtt{RULE2.}\text{'}), @_{\mathtt{aver2}}(\underline{r(a)}, s(a))))); \{X/a\}\rangle \quad \to_{AS2}{}^{\mathscr{R}_3}$$

$$\langle \&_{\mathtt{P}}(\mathrm{info}(0.9,\ \text{`}\mathtt{RULE1.}\text{'}), \&_{\mathtt{G}}(\mathrm{info}(0.8,\ \text{`}\mathtt{RULE2.}\text{'}), @_{\mathtt{aver2}}(\mathrm{info}(0.7,\ \text{`}\mathtt{RULE3.}\text{'}), \underline{s(a)})))); \{X/a\}\rangle \quad \to_{AS2}{}^{\mathscr{R}_4}$$

$$\langle \&_{\mathtt{P}}(\mathrm{info}(0.9,\ \text{`}\mathtt{RULE1.}\text{'}), \&_{\mathtt{G}}(\mathrm{info}(0.8,\ \text{`}\mathtt{RULE2.}\text{'}), \underline{@_{\mathtt{aver2}}(\mathrm{info}(0.7,\ \text{`}\mathtt{RULE3.}\text{'}), \mathrm{info}(0.5,\ \text{`}\mathtt{RULE4.}\text{'}))})); \{X/a\}\rangle \quad \to_{IS}$$

$$\langle \&_{\mathtt{P}}(\mathrm{info}(0.9,\ \text{`}\mathtt{RULE1.}\text{'}), \underline{\&_{\mathtt{G}}(\mathrm{info}(0.8,\ \text{`}\mathtt{RULE2.}\text{'}), \mathrm{info}(0.85, *))}); \{X/a\}\rangle \quad \to_{IS}$$

$$\langle \underline{\&_{\mathtt{P}}(\mathrm{info}(0.9,\ \text{`}\mathtt{RULE1.}\text{'}), \mathrm{info}(0.8, **))}; \{X/a\}\rangle \quad \to_{IS}$$

$$\langle \mathrm{info}(0.72,\ \text{`}\mathtt{RULE1.RULE2.RULE3.RULE4.@AVER2.|GODEL.\#MAX.|LUKA.\#ADD.\#MIN.@AVER.\#ADD.\#DIV.\&GODEL.\#MIN.\&PROD.\#PROD.}\text{'}); \{X/a\}\rangle.$$

where we have used the following symbols with their corresponding meanings:

* `'RULE3.RULE4.@AVER2.|GODEL.#MAX.|LUKA.#ADD.#MIN.@AVER.#ADD.#DIV.'`

** `'RULE2.RULE3.RULE4.@AVER2.|GODEL.#MAX.|LUKA.#ADD.#MIN.@AVER.#ADD.#DIV.&GODEL.#MIN.'`

Figure 5: Derivation for solving $p(X)$ by using lattice $\mathscr{V} \times \mathscr{S}$.

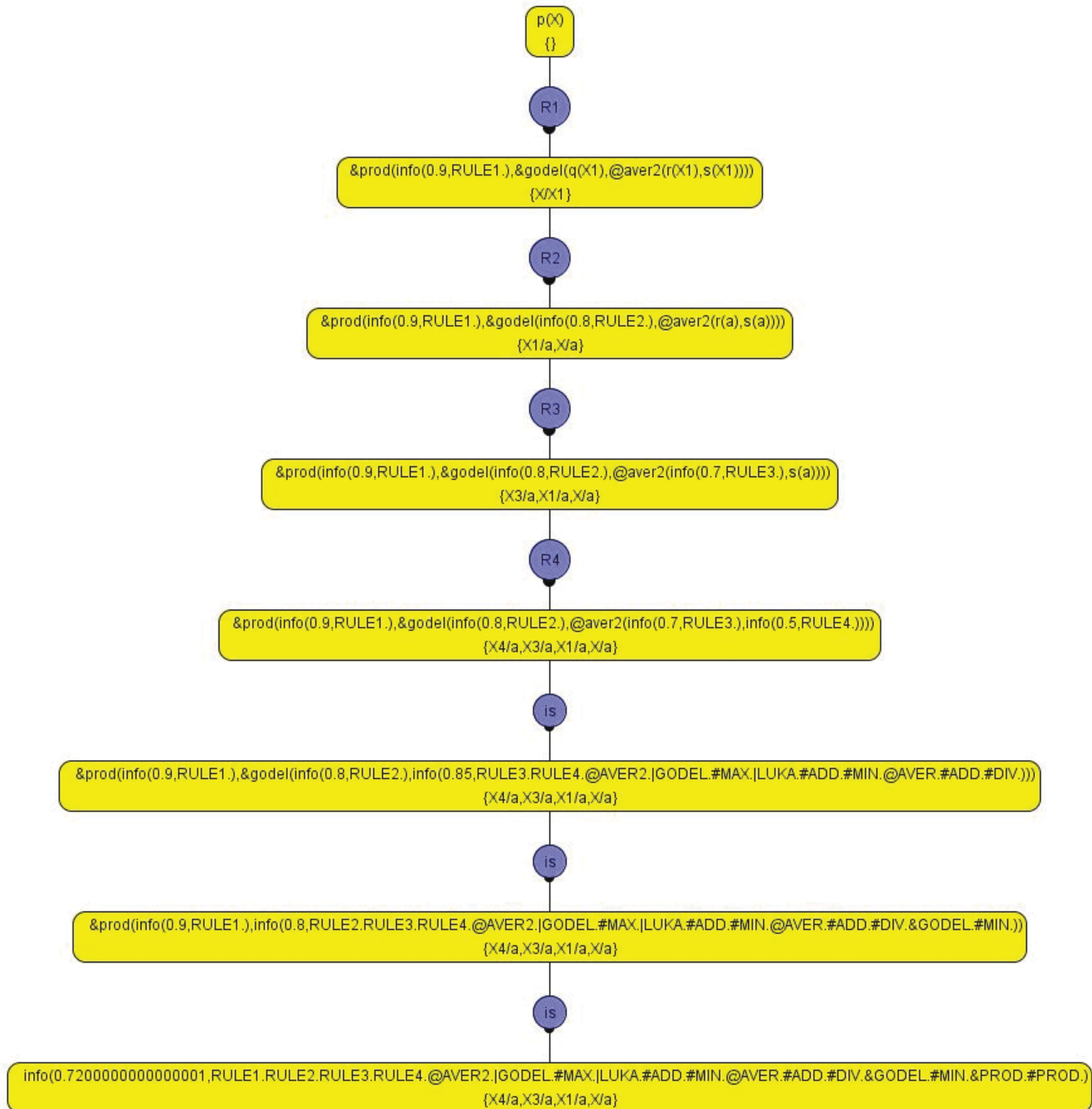
Figures 5 and 6 show the complete derivation built to reach the desired fuzzy computed answer according the procedural semantics described in Section 3.

5 Conclusions and Future Work

This paper has been mainly concerned with theoretical and practical issues focusing into the MALP framework (which could be seen as a very enriched fuzzy extension of Prolog), regarding the use of a string-based multi-adjoint lattice called $\mathscr{S}$. In a more precise way, we have proved and illustrated that the Cartesian product of $\mathscr{S}$ with any other multi-adjoint lattice, is useful to obtain a new, more powerful lattice which can be used for obtaining, at a very low costs, declarative traces on fuzzy computed answers when executing MALP programs inside the FLOPER tool developed in our research group.

We nowadays continue by exploring new uses of such kind of sophisticated lattices in other domains. For instance, we advance in [ALM11, ALM12] that the Cartesian product of $\mathscr{V}$ with a lattice modeling *lists* with a similar shape to $\mathscr{S}$, is very useful for coding with MALP rules a fuzzy variant of the well-known XPath language for the flexible management of XML documents retrieved from the web (our first real-world application using the FLOPER tool can be freely downloaded and tested on-line from `http://dectau.uclm.es/fuzzyXPath/`).

Figure 6: Evaluation tree depicted by FLOPER associated to derivation shown in Figure 5.



Bibliography

[ALM11] J. Almendros-Jiménez, A. Luna, G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In *Proc. 5th Intl. Symp. on Rules, RuleML'11*. Pp. 186–193. Springer Verlag, Lecture Notes in Computer Science 6826, 2011.

[ALM12] J. M. Almendros-Jiménez, A. Luna, G. Moreno. Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language. *Electr. Notes Theor. Com-*

put. Sci., Elsevier 282:3–18, 2012.

[GM08] J. Guerrero, G. Moreno. Optimizing Fuzzy Logic Programs by Unfolding, Aggregation and Folding. *Electr. Notes Theor. Comput. Sci., Elsevier* 219:19–34, 2008.

[JA07] P. Julián, M. Alpuente. *Programación Lógica. Teoría y Práctica*. Pearson Educación, S.A., Madrid, 2007.

[JMP05] P. Julián, G. Moreno, J. Penabad. On Fuzzy Unfolding. A Multi-adjoint Approach. *Fuzzy Sets and Systems, Elsevier* 154:16–33, 2005.

[JMP09] P. Julián, G. Moreno, J. Penabad. On the Declarative Semantics of Multi-Adjoint Logic Programs. In *Proc. 10th Intl. Conf. on Artif. Neural Networks, IWANN'09*. Pp. 253–260. Springer, Lecture Notes in Computer Science 5517, 2009.

[Llo87] J. Lloyd. *Foundations of Logic Programming*. Second edition. Springer-Verlag, Berlin, 1987.

[MMPV10] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In *Proc. 4th Intl. Symp. on Rule Interchange and Applications, RuleML'10*. Pp. 119–126. Springer Verlag, Lecture Notes in Computer Science 6403, 2010.

[MMPV11a] P. J. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Declarative Traces into Fuzzy Computed Answers. In *Proc. 5th Intl. Symp. on Rules, RuleML'11*. Pp. 170–185. Springer Verlag, Lecture Notes in Computer Science 6826, 2011.

[MMPV11b] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Dedekind-Macneille Completion and Multi-Adjoint Lattices. In *Proc. 11th Intl. Conf. on Mathematical Methods in Science and Engineering, CMMSE'11* . Pp. 846–857. Vol. II, 2011.

[MMPV11c] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Fuzzy Computed Answers Collecting Proof Information. In *Proc. 11th Intl. Conf. on Artif. Neural Networks, IWANN'11*. Pp. 445–452. Springer Verlag, Lecture Notes in Computer Science 6692, 2011.

[MOV04] J. Medina, M. Ojeda-Aciego, P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems* 146:43–62, 2004.

[RR08] M. Rodríguez-Artalejo, C. A. Romero-Díaz. Quantitative logic programming revisited. In *Proc. Functional and Logic Programming, FLOPS'08*. Pp. 272–288. Springer, Lecture Notes in Computer Science 4989, 2008.

[RR09] M. Rodríguez-Artalejo, C. A. Romero-Díaz. Qualified Logic Programming with Bivalued Predicates. *Elec. Notes in Theor. Comp. Sci., Elsevier* 248:67–82, 2009.