# Symbolic Execution and Thresholding for Efficiently Tuning Fuzzy Logic Programs*

Ginés Moreno[1], Jaime Penabad[2], José A. Riaza[1], and Germán Vidal[3]

[1] Dept. of Computing Systems, UCLM, 02071 Albacete (Spain)
[2] Dept. of Mathematics, UCLM, 02071 Albacete (Spain)
[3] MiST, DSIC, Universitat Politècnica de València (Spain)
{Gines.Moreno,Jaime.Penabad,JoseAntonio.Riaza}@uclm.es, gvidal@dsic.upv.es

**Abstract.** Fuzzy logic programming is a growing declarative paradigm aiming to integrate fuzzy logic into logic programming. One of the most difficult tasks when specifying a fuzzy logic program is determining the right weights for each rule, as well as the most appropriate fuzzy connectives and operators. In this paper, we introduce a symbolic extension of fuzzy logic programs in which some of these parameters can be left unknown, so that the user can easily see the impact of their possible values. Furthermore, given a number of test cases, the most appropriate values for these parameters can be automatically computed. Finally, we show some benchmarks that illustrate the usefulness of our approach.

**Key words:** Fuzzy logic programming, symbolic execution, tuning

## 1 Introduction

*Logic Programming* [17] has been widely used as a formal method for problem solving and knowledge representation. Nevertheless, traditional logic programming languages do not incorporate techniques or constructs to explicitly deal with uncertainty and approximated reasoning. In order to fill this gap, *fuzzy logic programming* has emerged as an interesting—and still growing—research area which aims to consolidate the efforts for introducing fuzzy logic into logic programming.

During the last decades, several fuzzy logic programming systems have been developed. Here, essentially, the classical SLD resolution principle of logic programming has been replaced by a fuzzy variant with the aim of dealing with partial truth and reasoning with uncertainty in a natural way. Most of these systems implement (extended versions of) the resolution principle introduced by

* This work has been partially supported by the EU (FEDER), the State Research Agency (AEI) and the Spanish *Ministerio de Economía y Competitividad* under grants TIN2013-45732-C4-2-P, TIN2013-44742-C4-1-R, TIN2016-76843-C4-1-R, TIN2016-76843-C4-2-R (AEI/FEDER, UE) and by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic).

Lee [15], such as Elf-Prolog [7], F-Prolog [16], generalized annotated logic programming [13], Fril [4], MALP [18], FASILL [11, 12], the QLP scheme of [22] and the many-valued logic programming language of [23].

In this paper we focus on the so-called *multi-adjoint logic programming* approach MALP [18], a powerful and promising approach in the area of fuzzy logic programming. Intuitively speaking, logic programming is extended with a *multi-adjoint lattice* $L$ of truth values (typically, a real number between 0 and 1), equipped with a collection of *adjoint pairs* $\langle \&_i, \leftarrow_i \rangle$ and connectives: implications, conjunctions, disjunctions, and other operators called aggregators, which are interpreted on this lattice. Consider, for instance, the following MALP rule:

$$good(X) \leftarrow_{\mathtt{P}} @_{\mathtt{aver}}(nice(X), cheap(X)) \ with \ 0.8$$

where the adjoint pair $\langle \&_{\mathtt{P}}, \leftarrow_{\mathtt{P}} \rangle$ is defined as

$$\&_{\mathtt{P}}(x, y) \triangleq x * y \qquad \leftarrow_{\mathtt{P}} (x, y) \triangleq \begin{cases} 1 & \text{if } y \leq x \\ x/y & \text{if } 0 < x < y \end{cases}$$

and the aggregator $@_{\mathtt{aver}}$ is typically defined as $@_{\mathtt{aver}}(x_1, x_2) \triangleq (x_1 + x_2)/2$. Therefore, the rule specifies that $X$ is good—with a truth degree of 0.8—if $X$ is nice and cheap. Assuming that $X$ is nice and cheap with, e.g., truth degrees $n$ and $c$, respectively, then $X$ is good with a truth degree of $0.8 * ((n + c)/2)$.

When specifying a MALP program, it might sometimes be difficult to assign weights—truth degrees—to program rules, as well as to determine the right connectives.[4] This is a common problem with fuzzy control system design, where some trial-and-error is often necessary. In our context, a programmer can develop a prototype and repeatedly execute it until the set of answers is the intended one. Unfortunately, this is a tedious and time consuming operation. Actually, it might be impractical when the program should correctly model a large number of test cases provided by the user.

In order to overcome this drawback, in this paper we introduce a symbolic extension of MALP programs called *symbolic multi-adjoint logic programming* (sMALP). Here, we can write rules containing *symbolic* truth degrees and *symbolic* connectives, i.e., connectives which are not defined on its associated multi-adjoint lattice. In order to evaluate these programs, we introduce a symbolic operational semantics that delays the evaluation of symbolic expressions. Therefore, a *symbolic answer* could now include symbolic (unknown) truth values and connectives. We prove the correctness of the approach, i.e., the fact that using the symbolic semantics and then replacing the unknown values and connectives by concrete ones gives the same result as replacing these values and connectives in the original sMALP program and, then, applying the concrete semantics on the resulting MALP program. Furthermore, we show how sMALP programs can be used to tune a program w.r.t. a given set of test cases, thus easing what is

[4] For instance, we have typically several adjoint pairs: *Łukasiewicz logic* $\langle \&_{\mathtt{L}}, \leftarrow_{\mathtt{L}} \rangle$, *Gödel logic* $\langle \&_{\mathtt{G}}, \leftarrow_{\mathtt{G}} \rangle$ and *product logic* $\langle \&_{\mathtt{P}}, \leftarrow_{\mathtt{P}} \rangle$, which might be used for modeling *pessimist*, *optimist* and *realistic scenarios*, respectively.

considered the most difficult part of the process: the specification of the right weights and connectives for each rule. We plan to integrate this tuning process into the FLOPER system (*Fuzzy LOgic Programming Environment for Research*); see, e.g., [19, 20]. In this paper, we show the results of an experimental evaluation using a prototype implementation of the system, which is available online from `http://dectau.uclm.es/tuning/`.

The structure of this paper is as follows. After some preliminaries in Section 2, we introduce the framework of symbolic multi-adjoint logic programming in Section 3 and prove its correctness. Then, in Section 4, we show the usefulness of symbolic programs for tuning several parameters so that a concrete program is obtained. Moreover, we show some interesting experiments together with an online implementation which also considers a very efficient tuning method improved with thresholding techniques. Finally, Section 5 concludes and points out some directions for further research.

## 2 Preliminaries

We assume the existence of a multi-adjoint lattice $\langle L, \preceq, \&_1, \leftarrow_1, \ldots, \&_n, \leftarrow_n \rangle$, equipped with a collection of *adjoint pairs* $\langle \&_i, \leftarrow_i \rangle$—where each $\&_i$ is a conjunctor which is intended to be used for the evaluation of *modus ponens* [18]—. In addition, on each program rule, we can have a different adjoint implication ($\leftarrow_i$), conjunctions (denoted by $\wedge_1, \wedge_2, \ldots$), adjoint conjunctions ($\&_1, \&_2, \ldots$), disjunctions ($|_1, |_2, \ldots$), and other operators called aggregators (usually denoted by $@_1, @_2, \ldots$); see [21] for more details. More exactly, a multi-adjoint lattice fulfills the following properties:

- $\langle L, \preceq \rangle$ is a (bounded) complete lattice.[5]
- For each truth function of $\&_i$, an increase in any of the arguments results in an increase of the result (they are *increasing*).
- For each truth function of $\leftarrow_i$, the result increases as the first argument increases, but it decreases as the second argument increases (they are *increasing* in the consequent and *decreasing* in the antecedent).
- $\langle \&_i, \leftarrow_i \rangle$ is an *adjoint pair* in $\langle L, \preceq \rangle$, namely, for any $x, y, z \in L$, we have that: $x \preceq (y \leftarrow_i z)$ if and only if $(x \,\&_i\, z) \preceq y$.

The last condition, called the *adjoint property*, could be considered the most important feature of the framework (in contrast with other approaches) which justifies most of its properties regarding crucial results for soundness, completeness, applicability, etc. [18].

Aggregation operators are useful to describe or specify user preferences. An aggregation operator, when interpreted as a truth function, may be an arith-

[5] A complete lattice is a (partially) ordered set $\langle L, \preceq \rangle$ such that every subset $S$ of $L$ has infimum and supremum elements. It is bounded if it has bottom and top elements, denoted by $\bot$ and $\top$, respectively. $L$ is said to be the *carrier set* of the lattice, and $\preceq$ its ordering relation.

metic mean, a weighted sum or in general any monotone function whose arguments are values of a multi-adjoint lattice $L$. Although, formally, these connectives are binary operators, we often use them as $n$-ary functions so that $@(x_1, \ldots, @(x_{n-1}, x_n), \ldots)$ is denoted by $@(x_1, \ldots, x_n)$. By abuse of notation, in these cases, we consider $@$ an $n$-ary operator. The truth function of an $n$-ary connective $\varsigma$ is denoted by $[\![\varsigma]\!] : L^n \mapsto L$ and is required to be monotonic and fulfill the following conditions: $[\![\varsigma]\!](\top, \ldots, \top) = \top$ and $[\![\varsigma]\!](\bot, \ldots, \bot) = \bot$.

In this work, given a multi-adjoint lattice $L$, we consider a first order language $\mathcal{L}_L$ built upon a signature $\Sigma_L$, that contains the elements of a countably infinite set of variables $\mathcal{V}$, function and predicate symbols (denoted by $\mathcal{F}$ and $\Pi$, respectively) with an associated arity—usually expressed as pairs $f/n$ or $p/n$, respectively, where $n$ represents its arity—, and the truth degree literals $\Sigma_L^T$ and connectives $\Sigma_L^C$ from $L$. Therefore, a well-formed formula in $\mathcal{L}_L$ can be either:

- A *value* $v \in \Sigma_L^T$, which will be interpreted as itself, i.e., as the truth degree $v \in L$.
- $p(t_1, \ldots, t_n)$, if $t_1, \ldots, t_n$ are terms over $\mathcal{V} \cup \mathcal{F}$ and $p/n$ is an n-ary predicate. This formula is called *atomic* (or just an atom).
- $\varsigma(e_1, \ldots, e_n)$, if $e_1, \ldots, e_n$ are well-formed formulas and $\varsigma$ is an $n$-ary connective with truth function $[\![\varsigma]\!] : L^n \mapsto L$.

As usual, a *substitution* $\sigma$ is a mapping from variables from $\mathcal{V}$ to terms over $\mathcal{V} \cup \mathcal{F}$ such that $Dom(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is its domain. Substitutions are usually denoted by sets of pairs like, e.g., $\{x_1/t_1, \ldots, x_n/t_n\}$. Substitutions are extended to morphisms from terms to terms in the natural way. The identity substitution is denoted by $id$. Composition of substitutions is denoted by juxtaposition, i.e., $\sigma\theta$ denotes a substitution $\delta$ such that $\delta(x) = \theta(\sigma(x))$ for all $x \in \mathcal{V}$.

In the following, an *$L$-expression* is a well-formed formula of $\mathcal{L}_L$ which is composed only by values and connectives from $L$, i.e., expressions over $\Sigma_L^T \cup \Sigma_L^C$.

In what follows, we assume that the truth function of any connective $\varsigma$ in $L$ is given by a corresponding definition of the form $[\![\varsigma]\!](x_1, \ldots, x_n) \triangleq E$.[6] For instance, in this work, we will be mainly concerned with the classical set of adjoint pairs (conjunctors and implications) over $\langle [0,1], \leq \rangle$ shown in Figure 1, where labels `L`, `G` and `P` mean respectively *Łukasiewicz logic*, *Gödel logic* and *Product logic* (which might be used for modeling *pessimist*, *optimist* and *realistic scenarios*, respectively).

A MALP *rule* over a multi-adjoint lattice $L$ is a formula $H \leftarrow_i \mathcal{B}$, where $H$ is an *atomic formula* (usually called the *head* of the rule), $\leftarrow_i$ is an implication symbol belonging to some adjoint pair of $L$, and $\mathcal{B}$ (which is called the *body* of the rule) is a well-formed formula over $L$ without implications. A *goal* is a body submitted as a query to the system. A MALP program is a set of expressions $R$ *with* $v$, where $R$ is a rule and $v$ is a *truth degree* (a value of $L$) expressing the confidence of a programmer in the truth of rule $R$. By abuse of the language, we often refer to $R$ *with* $v$ as a rule. See, e.g., [18] for a complete formulation of the MALP framework.

[6] For convenience, in the following sections, we do not distinguish between the connective $\varsigma$ and its truth function $[\![\varsigma]\!]$.

$$\&_{\mathtt{P}}(x,y) \triangleq x * y \qquad \leftarrow_{\mathtt{P}}(x,y) \triangleq \begin{cases} 1 & \text{if } y \leq x \\ x/y & \text{if } 0 < x < y \end{cases} \qquad \textit{Product logic}$$

$$\&_{\mathtt{G}}(x,y) \triangleq \min(x,y) \qquad \leftarrow_{\mathtt{G}}(x,y) \triangleq \begin{cases} 1 & \text{if } y \leq x \\ x & \text{otherwise} \end{cases} \qquad \textit{Gödel logic}$$

$$\&_{\mathtt{L}}(x,y) \triangleq \max(0, x+y-1) \qquad \leftarrow_{\mathtt{L}}(x,y) \triangleq \min(x-y+1, 1) \qquad \textit{Łukasiewicz logic}$$

**Fig. 1.** Adjoint pairs of three different fuzzy logics over $\langle [0,1], \leq \rangle$.

## 3 Symbolic Multi-adjoint Logic Programming

In this section, we introduce a *symbolic* extension of multi-adjoint logic programming. Essentially, we will allow some undefined values (truth degrees) and connectives in the program rules, so that these elements can be systematically computed afterwards. In the following, we will use the abbreviation sMALP to refer to programs belonging to this setting.

Here, given a multi-adjoint lattice $L$, we consider an augmented language $\mathcal{L}_L^s \supseteq \mathcal{L}_L$ which may also include a number of symbolic values, symbolic adjoint pairs and symbolic connectives which do not belong to $L$. Symbolic objects are usually denoted as $o^s$ with a superscript $s$.

**Definition 1 (sMALP program).** *Let $L$ be a multi-adjoint lattice. An* sMALP *program over $L$ is a set of symbolic rules, where each symbolic rule is a formula $(H \leftarrow_i \mathcal{B}$ with $v)$ that meets the following conditions:*

- *$H$ is an atomic formula of $\mathcal{L}_L$ (the head of the rule);*
- *$\leftarrow_i$ is a (possibly symbolic) implication from either a symbolic adjoint pair $\langle \&^s, \leftarrow^s \rangle$ or from an adjoint pair of $L$;*
- *$\mathcal{B}$ (the body of the rule) is a symbolic goal, i.e., a well-formed formula of $\mathcal{L}_L^s$;*
- *$v$ is either a truth degree (a value of $L$) or a symbolic value.*

*Example 1.* We consider the multi-adjoint lattice $\langle [0,1], \leq, \&_{\mathtt{P}}, \leftarrow_{\mathtt{P}}, \&_{\mathtt{G}}, \leftarrow_{\mathtt{G}}, \&_{\mathtt{L}}, \leftarrow_{\mathtt{L}} \rangle$, where the adjoint pairs are defined in Section 2, also including $@_{\mathtt{aver}}$ which is defined as follows: $@_{\mathtt{aver}}(x_1, x_2) \triangleq (x_1 + x_2)/2$. Then, the following is an sMALP program $\mathcal{P}$:

$$\begin{array}{lll} p(X) \leftarrow^{s_1} \&^{s_2}(q(X), @_{\mathtt{aver}}(r(X), s(X))) & \textit{with} & 0.9 \\ q(a) & \textit{with} & v^s \\ r(X) & \textit{with} & 0.7 \\ s(X) & \textit{with} & 0.5 \end{array}$$

where $\langle \&^{s_1}, \leftarrow^{s_1} \rangle$ is a symbolic adjoint pair (i.e., a pair not defined in $L$), $\&^{s_2}$ is a symbolic conjunction, and $v^s$ is a symbolic value.

The procedural semantics of sMALP is defined in a stepwise manner as follows. First, an *operational* stage is introduced which proceeds similarly to SLD resolution in pure logic programming. In contrast to standard logic programming,

though, our operational stage returns an expression still containing a number of (possibly symbolic) values and connectives. Then, an *interpretive* stage evaluates these connectives and produces a final answer (possibly containing symbolic values and connectives). The procedural semantics of both MALP and sMALP programs is based on a similar scheme. The main difference is that, for MALP programs, the interpretive stage always returns a value, while for sMALP programs we might get an expression containing symbolic values and connectives that should be first instantiated in order to compute a value.

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 $\theta$ by restricting its domain to $\mathcal{V}ar(s)$. An sMALP *state* has the form $\langle \mathcal{Q}; \sigma \rangle$ where $\mathcal{Q}$ is a symbolic goal and $\sigma$ is a substitution. We let $\mathcal{E}^s$ denote the set of all possible sMALP states.

**Definition 2 (admissible step).** *Let $L$ be a multi-adjoint lattice and $\mathcal{P}$ an* sMALP *program over $L$. An* admissible step *is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathcal{E}^s \times \mathcal{E}^s)$ is the smallest relation satisfying the following transition rules:*[7]

1. $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v \&_i \mathcal{B}])\theta; \sigma\theta \rangle$,
   *if* $\theta = mgu(\{H = A\}) \neq fail$, $(H \leftarrow_i \mathcal{B}$ *with* $v) \ll \mathcal{P}$ *and* $\mathcal{B}$ *is not empty.*[8]
2. $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/\bot]); \sigma \rangle$,
   *if there is no rule* $(H \leftarrow_i \mathcal{B}$ *with* $v) \ll \mathcal{P}$ *such that* $mgu(\{H = A\}) \neq fail$.

*Here, $(H \leftarrow_i \mathcal{B}$ with $v) \ll \mathcal{P}$ denotes that $(H \leftarrow_i \mathcal{B}$ with $v)$ is a renamed apart variant of a rule in $\mathcal{P}$ (i.e., all its variables are fresh). Note that symbolic values and connectives are not renamed.*

Observe that the second rule is needed to cope with expressions like $@_{aver}(p(a), 0.8)$, which can be evaluated successfully even when there is no rule matching $p(a)$ since $@_{aver}(0, 0.8) = 0.4$.

In the following, given a relation $\rightarrow$, we let $\rightarrow^*$ denote its reflexive and transitive closure. Also, an *$L^s$-expression* is now a well-formed formula of $\mathcal{L}^s_L$ which is composed by values and connectives from $L$ as well as by symbolic values and connectives.

**Definition 3 (admissible derivation).** *Let $L$ be a multi-adjoint lattice and $\mathcal{P}$ be an* sMALP *program over $L$. Given a goal $\mathcal{Q}$, an* admissible derivation *is a sequence $\langle \mathcal{Q}; id \rangle \rightarrow^*_{AS} \langle \mathcal{Q}'; \theta \rangle$. When $\mathcal{Q}'$ is an $L^s$-expression, the derivation is called* final *and the pair $\langle \mathcal{Q}'; \sigma \rangle$, where $\sigma = \theta[\mathcal{V}ar(\mathcal{Q})]$, is called a* symbolic admissible computed answer *(saca, for short) for goal $\mathcal{Q}$ in $\mathcal{P}$.*

[7] Here, we assume that $A$ in $\mathcal{Q}[A]$ is the selected atom. Furthermore, as it is common practice, $mgu(E)$ denotes the *most general unifier* of the set of equations $E$ [14].

[8] For simplicity, we consider that facts ($H$ *with* $v$) are seen as rules of the form ($H \leftarrow_i \top$ *with* $v$) for some implication $\leftarrow_i$. Furthermore, in this case, we directly derive the state $\langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle$ since $v \,\&_i \top = v$ for all $\&_i$.

*Example 2.* Consider again the multi-adjoint lattice $L$ and the sMALP program $\mathcal{P}$ of Example 1. Here, we have the following final admissible derivation for $p(X)$ in $\mathcal{P}$ (the selected atom is underlined):

$$\begin{aligned}\langle \underline{p(X)};\ id\rangle &\to_{AS} \langle \&^{s_1}(0.9, \&^{s_2}(\underline{q(X_1)}, @_{\mathtt{aver}}(r(X1), s(X1)))); \{X/X_1\}\rangle\\ &\to_{AS} \langle \&^{s_1}(0.9, \&^{s_2}(v^s, @_{\mathtt{aver}}(\underline{r(a)}, s(a)))); \{X/a, X_1/a\}\rangle\\ &\to_{AS} \langle \&^{s_1}(0.9, \&^{s_2}(v^s, @_{\mathtt{aver}}(0.7, \underline{s(a)}))); \{X/a, X_1/a, X_2/a\}\rangle\\ &\to_{AS} \langle \&^{s_1}(0.9, \&^{s_2}(v^s, @_{\mathtt{aver}}(0.7, 0.5))); \{X/a, X_1/a, X_2/a, X_3/a\}\rangle\end{aligned}$$

Therefore, the associated saca is $\langle \&^{s_1}(0.9, \&^{s_2}(v^s, @_{\mathtt{aver}}(0.7, 0.5))); \{X/a\}\rangle$.

Given a goal $\mathcal{Q}$ and a final admissible derivation $\langle \mathcal{Q}; id\rangle \to^*_{AS} \langle \mathcal{Q}'; \sigma\rangle$, we have that $\mathcal{Q}'$ does not contain atomic formulas. Now, $Q'$ can be *solved* by using the following interpretive stage:

**Definition 4 (interpretive step).** *Let $L$ be a multi-adjoint lattice and $\mathcal{P}$ be an* sMALP *program over $L$. Given a* saca *$\langle Q; \sigma\rangle$, the* interpretive *stage is formalized by means of the following transition relation $\to_{IS} \subseteq (\mathcal{E}^s \times \mathcal{E}^s)$, which is defined as the least transition relation satisfying:*

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

*where $\varsigma$ denotes a connective defined on $L$ and $[\![\varsigma]\!](r_1, \ldots, r_n) = r_{n+1}$.*

*An interpretive derivation of the form $\langle \mathcal{Q}; \sigma\rangle \to^*_{IS} \langle \mathcal{Q}'; \theta\rangle$ such that $\langle \mathcal{Q}'; \theta\rangle$ cannot be further reduced, is called a* final *interpretive derivation. In this case, $\langle \mathcal{Q}'; \theta\rangle$ is called a* symbolic fuzzy computed answer *(sfca, for short). Also, if $\mathcal{Q}'$ is a value of $L$, we say that $\langle \mathcal{Q}'; \theta\rangle$ is a fuzzy computed answer (fca, for short).*

*Example 3.* Given the saca of Ex. 2: $\langle \&^{s_1}(0.9, \&^{s_2}(v^s, @_{\mathtt{aver}}(0.7, 0.5))); \{X/a\}\rangle$, we have the following final interpretive derivation (the connective reduced is underlined):

$$\langle \&^{s_1}(0.9, \&^{s_2}(v^s, \underline{@_{\mathtt{aver}}(0.7, 0.5)})); \{X/a\}\rangle \to_{IS} \langle \&^{s_1}(0.9, \&^{s_2}(v^s, 0.6)); \{X/a\}\rangle$$

with $[\![@_{\mathtt{aver}}]\!](0.7, 0.5) = 0.6$. Therefore, $\langle \&^{s_1}(0.9, \&^{s_2}(v^s, 0.6)); \{X/a\}\rangle$ is a sfca of $p(X)$ in $\mathcal{P}$ since it cannot be further reduced.

Given a multi-adjoint lattice $L$ and a symbolic language $\mathcal{L}^s_L$, in the following we consider *symbolic substitutions* that are mappings from symbolic values and connectives to expressions over $\Sigma^T_L \cup \Sigma^C_L$. Symbolic substitutions are denoted by $\Theta, \Gamma, \ldots$ Furthermore, for all symbolic substitution $\Theta$, we require the following condition: $\leftarrow^s/\leftarrow_i \in \Theta$ iff $\&^s/\&_i \in \Theta$, where $\langle \&^s, \leftarrow^s\rangle$ is a symbolic adjoint pair and $\langle \&_i, \leftarrow_i\rangle$ is an adjoint pair in $L$. Intuitively, this is required for the substitution to have the same effect both on the program and on an $L^s$-expression.

Given an sMALP program $\mathcal{P}$ over $L$, we let sym($\mathcal{P}$) denote the symbolic values and connectives in $\mathcal{P}$. Given a symbolic substitution $\Theta$ for sym($\mathcal{P}$), we denote by $\mathcal{P}\Theta$ the program that results from $\mathcal{P}$ by replacing every symbolic symbol $e^s$ by $e^s\Theta$. Trivially, $\mathcal{P}\Theta$ is now a MALP program.

The following theorem is our key result in order to use sMALP programs for tuning the components of a MALP program:

**Theorem 1.** *Let $L$ be a multi-adjoint lattice and $\mathcal{P}$ be an* sMALP *program over $L$. Let $\mathcal{Q}$ be a goal. Then, for any symbolic substitution $\Theta$ for* sym$(\mathcal{P})$*, we have that $\langle v;\theta\rangle$ is a* fca *for $Q$ in $\mathcal{P}\Theta$ iff there exists a* sfca *$\langle Q';\theta'\rangle$ for $\mathcal{Q}$ in $\mathcal{P}$ and $\langle \mathcal{Q}'\Theta;\theta'\rangle \to^*_{IS} \langle v;\theta'\rangle$, where $\theta'$ is a renaming of $\theta$.*

*Proof.* (Sketch) For simplicity, we consider that the same fresh variables are used for renamed apart rules in both derivations.

Consider the following derivations for goal $\mathcal{Q}$ w.r.t. programs $\mathcal{P}$ and $\mathcal{P}\Theta$, respectively:

$$\begin{array}{l}\mathcal{D}_{\mathcal{P}} \;\; : \langle \mathcal{Q}; id\rangle \to^*_{AS} \langle \mathcal{Q}''; \theta\rangle \;\; \to^*_{IS} \langle \mathcal{Q}'; \theta\rangle \\ \mathcal{D}_{\mathcal{P}\Theta} : \langle \mathcal{Q}; id\rangle \to^*_{AS} \langle \mathcal{Q}''\Theta; \theta\rangle \to^*_{IS} \langle \mathcal{Q}'\Theta; \theta\rangle\end{array}$$

Our proof proceeds now in three stages:

1. Firstly, observe that the sequences of symbolic admissible steps in $\mathcal{D}_{\mathcal{P}}$ and $\mathcal{D}_{\mathcal{P}\Theta}$ exploit the whole set of atoms in both cases, such that a program rule $R$ is used in $\mathcal{D}_{\mathcal{P}}$ iff the corresponding rule $R\Theta$ is applied in $\mathcal{D}_{\mathcal{P}\Theta}$ and hence, the saca's of the derivations are $\langle \mathcal{Q}''; \theta\rangle$ and $\langle \mathcal{Q}''\Theta; \theta\rangle$, respectively.
2. Then, we proceed by applying interpretive steps until reaching the sfca $\langle \mathcal{Q}'; \theta\rangle$ in the first derivation $\mathcal{D}_{\mathcal{P}}$ and it is easy to see that the same sequence of interpretive steps are applied in $\mathcal{D}_{\mathcal{P}\Theta}$ thus leading to state $\langle \mathcal{Q}'\Theta; \theta\rangle$, which is not necessarily a sfca.
3. Finally, it suffices to instantiate the sfca $\langle \mathcal{Q}'; \theta\rangle$ in the first derivation $\mathcal{D}_{\mathcal{P}}$ with the symbolic substitution $\Theta$, for completing both derivations with the same sequence of interpretive steps until reaching the desired fca $\langle v; \theta\rangle$.

□

*Example 4.* Consider again the multi-adjoint lattice $L$ and the sMALP program $\mathcal{P}$ of Example 1. Let $\Theta = \{\leftarrow^{s_1}/\leftarrow_{\mathtt{P}}, \&^{s_1}/\&_{\mathtt{P}}, \&^{s_2}/\&_{\mathtt{G}}, v^s/0.8\}$ be a symbolic substitution. Given the sfca from Example 3, we have:

$$\langle \&^{s_1}(0.9, \&^{s_2}(v^s, 0.6))\Theta; \{X/a\}\rangle = \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, 0.6)); \{X/a\}\rangle$$

So, we have the following interpretive final derivation for the instantiated sfca:

$$\langle \&_{\mathtt{P}}(0.9, \underline{\&_{\mathtt{G}}(0.8, 0.6)}); \{X/a\}\rangle \to_{IS} \langle \underline{\&_{\mathtt{P}}(0.9, 0.6)}; \{X/a\}\rangle \to_{IS} \langle 0.54; \{X/a\}\rangle$$

By Theorem 1, we have that $\langle 0.54; \{X/a\}\rangle$ is also a fca for $p(X)$ in $\mathcal{P}\Theta$.

## 4 Tuning Multi-adjoint Logic Programs

In this section, we introduce an automated technique for tuning multi-adjoint logic programs using sMALP programs.

Consider a typical Prolog clause "$H : -B_1, \ldots, B_n$". It can be fuzzified in order to become a MALP rule "$H \leftarrow_{label} \mathcal{B}$ with $v$" by performing the following actions:

1. weighting it with a truth degree $v$,
2. connecting its head and body with a fuzzy implication symbol $\leftarrow_{label}$ (belonging to a concrete adjoint pair $\langle \leftarrow_{label}, \&_{label} \rangle$) and,
3. linking the set of atoms $B_1, \ldots, B_n$ on its body $\mathcal{B}$ by means of a set of fuzzy connectives (i.e., conjunctions $\&_i$, disjunctions $|_j$ or aggregators $@_k$).

Introducing changes on each one of the three fuzzy components just described above may affect—sometimes in an unexpected way—the set of fuzzy computed answers for a given goal.

Typically, a programmer has a model in mind where some parameters have a clear value. For instance, the truth value of a rule might be statistically determined and, thus, its value is easy to obtain. In other cases, though, the most appropriate values and/or connectives depend on subjective notions and, thus, programmers do not know how to obtain these values. In a typical scenario, we have an extensive set of *expected* computed answers (i.e., *test cases*), so the programmer can follow a "try and test" strategy. Unfortunately, this is a tedious and time consuming operation. Actually, it might even be impractical when the program should correctly model a large number of test cases.

Therefore, we propose an automated technique that proceeds as follows. Here, for simplicity, we only consider the first answer to a goal. Note that this is not a significant restriction since one can encode multiple solutions in a list so that the main goal is always deterministic and all non-deterministic calls are hidden in the computation. Extending the following algorithm for multiple solutions is not difficult, but makes the formalization more cumbersome. Hence, we say that a *test case* is a pair $(Q, f)$ where $Q$ is a goal and $f$ is an fca.

**Definition 5 (naive algorithm for symbolic tuning of MALP programs).**

**Input:** *an* sMALP *program* $\mathcal{P}^s$ *and a number of (expected) test cases* $(Q_i, \langle v_i; \theta_i \rangle)$, *where* $Q_i$ *is a goal and* $\langle v_i; \theta_i \rangle$ *is its expected* fca *for* $i = 1, \ldots, k$.
**Output:** *a symbolic substitution* $\Theta$.

1. *For each test case* $(Q_i, \langle v_i; \theta_i \rangle)$, *compute the* sfca $\langle Q'_i, \theta_i \rangle$ *of* $\langle Q_i, id \rangle$ *in* $\mathcal{P}^s$.
2. *Then, consider a finite number of possible symbolic substitutions for* $\mathsf{sym}(\mathcal{P}^s)$, *say* $\Theta_1, \ldots, \Theta_n$, $n > 0$.
3. *For each* $j \in \{1, \ldots, n\}$, *compute* $\langle Q'_i \Theta_j, \theta_i \rangle \rightarrow^*_{IS} \langle v_{i,j}; \theta_i \rangle$, *for* $i = 1, \ldots, k$. *Let* $d_{i,j} = |v_{i,j} - v_i|$, *where* $|\_|$ *denotes the absolute value.*
4. *Finally, return the symbolic substitution* $\Theta_j$ *that minimizes* $\sum_{i=1}^{k} d_{i,j}$.

Observe that the precision of the algorithm can be parameterized depending on the set of symbolic substitutions considered in step (2). For instance, one can consider only truth values $\{0.3, 0.5, 0.8\}$ or a larger set $\{0.1, 0.2, \ldots, 1.0\}$; one can consider only three possible connectives, or a set including ten of them. Obviously, the larger the domain of values and connectives is, the more precise the results are (but the algorithm is more expensive, of course).

| | | |
|---|---|---|
| $\&_{\mathtt{P}}(x,y) = x * y$ | $\|_{\mathtt{P}}(x,y) = x + y - x * y$ | *Product logic* |
| $\&_{\mathtt{G}}(x,y) = \min(x,y)$ | $\|_{\mathtt{G}}(x,y) = \max(x,y)$ | *Gödel logic* |
| $\&_{\mathtt{L}}(x,y) = \max(x+y-1,0)$ | $\|_{\mathtt{L}}(x,y) = \min(x+y,1)$ | *Łukasiewicz logic* |

**Fig. 2.** Conjunctions and disjunctions of three different fuzzy logics over $\langle [0,1], \leq \rangle$.

This algorithm represents a much more efficient method for tuning the fuzzy parameters of a MALP program than repeatedly executing the program from scratch (see Table 2, column "Basic").

Let us explain the technique by means of a small, but realistic example. Here, we consider a travel agency that offers booking services on a large number of hotels. The travel agency has a web site where the user can rate every hotel with a value between 1% and 100%. The purpose in this case is to specify a fuzzy model that correctly represents the rating of each hotel.

In order to simplify the presentation, we consider that there are only three hotels, named *sun*, *sweet* and *lux*. In the web site, these hotels have been rated 0.60, 0.77 and 0.85 (expressed as real numbers between 0 and 1), respectively. Our simple model just depends on three factors: the hotel facilities, the convenience of its location, and the rates, denoted by predicates *facilities*, *location* and *rates*, respectively. An sMALP program modelling this scenario is the following:

| | | |
|---|---|---|
| $popularity(X) \leftarrow^s \ \|^s(facilities(X), @_{\mathtt{aver}}(location(X), rates(X)))$ | | *with* 0.9 |
| $facilities(sun)$ | | *with* $v^s$ |
| $location(sun)$ | | *with* 0.4 |
| $rates(sun)$ | | *with* 0.7 |
| $facilities(sweet)$ | | *with* 0.5 |
| $location(sweet)$ | | *with* 0.3 |
| $rates(sweet)$ | | *with* 0.1 |
| $facilities(lux)$ | | *with* 0.9 |
| $location(lux)$ | | *with* 0.8 |
| $rates(lux)$ | | *with* 0.2 |

Here, we assume that all weights can be easily obtained except for the weight of the fact $facilities(sun)$, which is unknown, so we introduce a symbolic weight $v^s$. Also, the programmer has some doubts on the connectives used in the first rule, so she introduced a number of symbolic connectives: the implication and disjunction symbols, i.e. $\leftarrow^s$ and $|^s$.

We consider, for each symbolic connective, the three possibilities shown in Figure 2 over the lattice $\langle [0,1], \leq \rangle$, which are based on the so-called *Product*, *Gödel* and *Łukasiewicz* logics. Adjectives like *pessimist*, *realistic* and *optimist* are sometimes applied to the *Łukasiewicz*, *Product* and *Gödel* logics, respectively, since conjunctive operators satisfy that, for any pair of real numbers $x$ and $y$ in

$[0, 1]$, we have:

$$0 \leq \&_{\mathtt{L}}(x, y) \leq \&_{\mathtt{P}}(x, y) \leq \&_{\mathtt{G}}(x, y) \leq 1$$

In contrast, the contrary holds for the disjunction operations, that is:

$$0 \leq |_{\mathtt{G}}(x, y) \leq |_{\mathtt{P}}(x, y) \leq |_{\mathtt{L}}(x, y) \leq 1$$

Note that it is more difficult to satisfy a condition based on a pessimist conjunction/disjunction (i.e, inspired by the *Łukasiewicz* and *Gödel* fuzzy logics, respectively) than with *Product* logic based operators. The optimistic versions of these connectives are less restrictive, obtaining greater truth degrees on fca's. This is a consequence of the following chain of inequalities:

$$0 \leq \&_{\mathtt{L}}(x, y) \leq \&_{\mathtt{P}}(x, y) \leq \&_{\mathtt{G}}(x, y) \leq |_{\mathtt{G}}(x, y) \leq |_{\mathtt{P}}(x, y) \leq |_{\mathtt{L}}(x, y) \leq 1$$

Therefore, it is desirable to tune the symbolic constants $\leftarrow^s$ and $|^s$ in the first rule of our symbolic sMALP program by selecting operators in the previous sequence until finding solutions satisfying in a stronger (or weaker) way the user's requirements.

Focusing on our particular sMALP program, we consider the following three test cases:

$(popularity(sun), \langle 0.60; id\rangle)$,
$(popularity(sweet), \langle 0.77; id\rangle)$,
$(popularity(lux), \langle 0.85; id\rangle)$

for which the respective three sfca's achieved after applying the first step of our tuning algorithm are:

$\langle \&^s(0.9, |^s(v^s, 0.55)); id\rangle$
$\langle \&^s(0.9, |^s(0.5, 0.65)); id\rangle$
$\langle \&^s(0.9, |^s(0.9, 0.5)); id\rangle$

In the second step of the algorithm, we must provide symbolic substitutions for being applied to this set of sfca's in order to transform them into fca's which are as close as possible to those in the test cases. Table 1 shows the results of the tuning process, where each column has the following meaning:

- The first pair of columns serve for choosing the implication[9] and disjunction connectives of the first program rule (i.e., $\leftarrow^s$ and $|^s$) from each one of the three fuzzy logics considered so far.
- In the third column, we consider three possible truth degrees (0.3, 0.5 and 0.7) as the potential assignment to the symbolic weight $v^s$. In this example, this set suffices to obtain an accurate solution.

[9] It is important to note that, at execution time, each implication symbol belonging to a concrete adjoint pair is replaced by its adjoint conjunction (see again our repertoire of adjoint pairs in Figure 1 in the preliminaries section).

– Each row represents a different symbolic substitution, which are shown in column four.
– Next, headed by the name of each hotel in the test cases, we have pairs of columns which represent, respectively, the potential truth degree associated to the fca obtained with the corresponding symbolic substitution, and the deviation of such value w.r.t. the expected truth degree, thus summarizing the computations performed on the third step of our algorithm.
– The sum of the three deviations is expressed in the last column of the table, which constitutes the value to be minimized as indicated in the final, fourth step of the algorithm.

According to these criteria, we observe that the cell with the lower value (0.04) in the last column of Table 1 refers to the symbolic substitution

$$\Theta_{13} = \{\leftarrow^s/\leftarrow_{\mathtt{P}},\ |^s/\,|_{\mathtt{P}},\ v^s/0.3\}$$

which solve our tuning problem by suggesting that the first pair of rules in our final, tuned MALP program should be the following ones:

$$\begin{array}{lll} popularity(X) & \leftarrow_{\mathtt{P}}\ |_{\mathtt{P}}(services(X), @_{\mathtt{aver}}(location(X), rates(X))) & with\ 0.9 \\ facilities(sun) & & with\ 0.3 \end{array}$$

Unfortunately, the naive algorithm introduced so far might be very inefficient when dealing with many symbolic values and connectives, or when the considered set of their possible substitutions is large. Here, in order to improve its efficiency, we consider *thresholding* techniques—well-known in the fuzzy logic arena—for prematurely disregarding useless computations leading to non-significant answers (see our previous experiences in [10, 8, 2]).

The improved algorithm is perfectly analogous to the algorithm in Definition 5, but makes use of a threshold $\tau$ for determining when a *partial* solution is acceptable. The value of $\tau$ is initialized to $\infty$ (in practice, a very large number). Then, this threshold dinamically decreases whenever we find a symbolic substitution with an associated deviation which is lower that the actual value of $\tau$. Moreover, a partial solution is discarded as soon as the cumulative deviation computed so far is greater than $\tau$. In our running example, $\tau$ takes the following values: 0.42, 0.27, 0.05, and 0.04, associated to $\Theta_1$, $\Theta_3$, $\Theta_4$, and $\Theta_{13}$, respectively. In general, the number of discarded solutions grows as the value of $\tau$ decreases, improving the pruning power of thesholding. In Table 1, the discarded solutions are shown in bold. They represent a significant percentage of the total computations.

The symbolic execution and tuning methods explained so far can be tested online via the following URL:

`http://dectau.uclm.es/tuning/`

When introducing an sMALP program into the system, symbolic constants must be preceded by the symbol "#". For instance, the first couple of rules in our running example have the following form:

**Table 1.** Table summarizing the results achieved when tuning connectives and weights.

| $\leftarrow^s$ | $\mid^s$ | $v^s$ | $\Theta$ | sun | | sweet | | lux | | z |
|---|---|---|---|---|---|---|---|---|---|---|
| $\leftarrow_L$ | $\mid_G$ | 0.3 | $\Theta_1$ | 0.45 | 0.15 | 0.55 | 0.22 | 0.80 | 0.05 | 0.42 |
| | | 0.5 | $\Theta_2$ | 0.45 | 0.15 | 0.55 | 0.22 | 0.80 | 0.05 | 0.42 |
| | | 0.7 | $\Theta_3$ | 0.60 | 0.00 | 0.55 | 0.22 | 0.80 | 0.05 | 0.27 |
| | $\mid_P$ | 0.3 | $\Theta_4$ | 0.59 | 0.01 | 0.73 | 0.04 | 0.85 | 0.00 | 0.05 |
| | | 0.5 | $\Theta_5$ | 0.68 | 0.08 | **0.73** | **0.04** | **0.85** | **0.00** | **0.12** |
| | | 0.7 | $\Theta_6$ | 0.77 | 0.17 | **0.73** | **0.04** | **0.85** | **0.00** | **0.21** |
| | $\mid_L$ | 0.3 | $\Theta_7$ | 0.75 | 0.15 | **0.90** | **0.13** | **0.90** | **0.05** | **0.33** |
| | | 0.5 | $\Theta_8$ | 0.90 | 0.30 | **0.90** | **0.13** | **0.90** | **0.05** | **0.48** |
| | | 0.7 | $\Theta_9$ | 0.90 | 0.30 | **0.90** | **0.13** | **0.90** | **0.05** | **0.48** |
| $\leftarrow_P$ | $\mid_G$ | 0.3 | $\Theta_{10}$ | 0.50 | 0.10 | **0.59** | **0.18** | **0.81** | **0.04** | **0.32** |
| | | 0.5 | $\Theta_{11}$ | 0.50 | 0.10 | **0.59** | **0.18** | **0.81** | **0.04** | **0.32** |
| | | 0.7 | $\Theta_{12}$ | 0.63 | 0.03 | 0.59 | 0.18 | **0.81** | **0.04** | **0.25** |
| | $\mid_P$ | 0.3 | $\Theta_{13}$ | 0.61 | 0.01 | 0.74 | 0.03 | 0.85 | 0.00 | 0.04 |
| | | 0.5 | $\Theta_{14}$ | 0.70 | 0.10 | **0.74** | **0.03** | **0.86** | **0.01** | **0.14** |
| | | 0.7 | $\Theta_{15}$ | 0.78 | 0.18 | **0.74** | **0.03** | **0.86** | **0.01** | **0.22** |
| | $\mid_L$ | 0.3 | $\Theta_{16}$ | 0.77 | 0.17 | **0.90** | **0.13** | **0.90** | **0.05** | **0.35** |
| | | 0.5 | $\Theta_{17}$ | 0.90 | 0.30 | **0.90** | **0.13** | **0.90** | **0.05** | **0.48** |
| | | 0.7 | $\Theta_{18}$ | 0.90 | 0.30 | **0.90** | **0.13** | **0.90** | **0.05** | **0.48** |
| $\leftarrow_G$ | $\mid_G$ | 0.3 | $\Theta_{19}$ | 0.55 | 0.05 | **0.65** | **0.12** | **0.90** | **0.05** | **0.22** |
| | | 0.5 | $\Theta_{20}$ | 0.55 | 0.05 | **0.65** | **0.12** | **0.90** | **0.05** | **0.22** |
| | | 0.7 | $\Theta_{21}$ | 0.70 | 0.10 | **0.65** | **0.12** | **0.90** | **0.05** | **0.27** |
| | $\mid_P$ | 0.3 | $\Theta_{22}$ | 0.69 | 0.09 | **0.83** | **0.06** | **0.90** | **0.05** | **0.20** |
| | | 0.5 | $\Theta_{23}$ | 0.78 | 0.18 | **0.83** | **0.06** | **0.90** | **0.05** | **0.29** |
| | | 0.7 | $\Theta_{24}$ | 0.87 | 0.27 | **0.83** | **0.06** | **0.90** | **0.05** | **0.38** |
| | $\mid_L$ | 0.3 | $\Theta_{25}$ | 0.86 | 0.26 | **0.90** | **0.13** | **0.90** | **0.05** | **0.44** |
| | | 0.5 | $\Theta_{26}$ | 0.90 | 0.30 | **0.90** | **0.13** | **0.90** | **0.05** | **0.48** |
| | | 0.7 | $\Theta_{27}$ | 0.90 | 0.30 | **0.90** | **0.13** | **0.90** | **0.05** | **0.48** |

```
popularity(X) #<s1 facilities(X) #|s2 @aver(locatin(X),rates(X)) with 0.9
facilities(sun) with #s3
```

The lattice of truth degrees is encoded as a set of Prolog clauses (see [19, 20]) where predicate `members/1` contains the list of truth degrees used during the tuning process. Each test case appears in a different line as follows: $r$ `-> Q`, where $r$ is the desired truth degree for the first fca associated to query $\mathcal{Q}$. For tuning an sMALP program, we have implemented the three methods mentioned so far:

Basic: The basic method is based on applying each symbolic substitution to the original sMALP program and then fully executing the resulting instantiated MALP programs (both the operational and the interpretive stages).

Symbolic: This row refers to the naive algorithm introduced in Definition 5, where the considered substitutions are directly applied to sfca's (thus only the interpretive stage is repeatedly executed).

**Table 2.** Tuning runtime (in milliseconds).

| | truth degrees | | | symbolic constants | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 10 | 100 | 1000 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Basic | 120 | 1130 | 11360 | 320 | 990 | 3030 | 9180 | 28170 | 86760 | 264850 |
| Symbolic | 30 | 290 | 2860 | 100 | 290 | 980 | 2970 | 9930 | 30570 | 93360 |
| Thresholded | 15 | 130 | 1300 | 50 | 140 | 420 | 1580 | 4390 | 13460 | 38310 |

Thresholded: In this row, we consider the symbolic method improved with thresholding techniques, as explained above.

The system also reports the processing time required by each method and offers an option for applying the best symbolic substitution to the original sMALP program in order to show the final, tuned MALP program.

Table 2 summarizes the results of an experimental evaluation[10] of the three tuning methods described above, varying the number of truth degrees (10, 100 and 1000) used when manipulating our running example. Note that, in the most complex case, 9000 different symbolic substitutions are considered at tuning time, and the thresholded method is about 2 to 3 times more efficient than the symbolic method, and even 6 to 8 times more efficient than the basic method, which witnesses the advantages of our improved tuning mechanism. In the last column, we consider variations of the number of symbolic constants (among connectives and truth degrees) from 5 to 11, thus showing that the thresholded method scales up well and solves the problem in just a few seconds.

## 5 Discussion

In this paper, we have been concerned with fuzzy programs belonging to the so-called *multi-adjoint logic programming* approach. Our improvements are twofold:

- On one side, we have extended their syntax for allowing the presence of symbolic weights and connectives on program rules, which very often prevents the full evaluation of goals. As a consequence, we have also relaxed the operational principle for producing what we call *symbolic fuzzy computed answers*, where all atoms have been exploited and the maximum number of expressions involving connectives of the underlaying lattice of truth degrees have been solved too.
- On the other hand, we have introduced a tuning process for MALP programs that takes as inputs a set of expected test cases and an sMALP program where some connectives and/or truth degrees are unknown. The efficiency of the method has been largely improved by combining it with thresholding techniques, as can be checked online in our prototype implementation.

[10] Each cell refers to the average of 100 executions using a desktop computer equipped with an i3-2310M CPU @ 2.10 GHz and 4,00 GB RAM.

As future work, we consider the embedding of these techniques in the FLOPER platform, which is freely available from `http://dectau.uclm.es/floper/`. Currently, the system can be used for compiling MALP programs to standard Prolog code, drawing *derivation trees*, generating declarative traces and executing MALP programs [9, 10]. Our last update, described in [11, 12], allows the system to cope with *similarity relations* cohabiting with lattices of truth degrees. Extending our tuning method in order to cope with such similarity relations is also an interesting topic for future work.

Another interesting direction for further research consists in combining our approach with recent fuzzy variants of SAT/SMT techniques. Research on SAT (Boolean Satisfiability) and SMT (Satisfiability Modulo Theories) [5] has provided highly efficient solvers based on classical logic. Some recent approaches deal with propositional fuzzy formulae which might contain connectives defined on lattices of truth degrees quite similar to the ones used on MALP programs [3, 24].[11] In this context, we think that our tuning method could be significantly improved if the set of sfca's instantiated with symbolic substitutions could be expressed as fuzzy formulae, which are solvable by this kind of fuzzy SAT/SMT solvers.

## References


1. Almendros-Jiménez, J.M., Bofill, M., Luna, A., Moreno, G., Vázquez, C., Villaret, M.: Fuzzy XPath for the automatic search of fuzzy formulae models. In: Proc. of SUM'15. LNCS, 9310, pp. 385–398. Springer Verlag (2015)
2. J. M. Almendros-Jiménez, A. Luna, and G. Moreno. Fuzzy XPath through fuzzy logic programming. *New Generation Computing*, 33(2):173–209, (2015).
3. Ansótegui, C., Bofill, M., Manyà, F., Villaret, M.: Building automated theorem provers for infinitely-valued logics with satisfiability modulo theory solvers. In: Proc. of ISMVL'12, pp. 25–30 (2012)
4. Baldwin, J.F., Martin, T.P., Pilsworth, B.W.: Fril- Fuzzy and Evidential Reasoning in Artificial Intelligence. John Wiley & Sons, Inc. (1995)
5. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, 185, pp. 825–885. IOS Press (2009)
6. Bofill, M., Moreno, G., Vázquez, C., Villaret, M.: Automatic proving of fuzzy formulae with fuzzy logic programming and SMT. In: Fredlund, L.A. (ed.) Volume 64: Programming and Computer Languages 2013. pp. 19. ECEASST (2013)
7. Ishizuka, M., Kanai, N.: Prolog-ELF Incorporating Fuzzy Logic. In: Proc. of the IJCAI'85. pp. 701–703. Morgan Kaufmann (1985)
8. P. Julián, J. Medina, G. Moreno, and M. Ojeda. Efficient thresholded tabulation for fuzzy query answering. *Studies in Fuzziness and Soft Computing (Foundations of Reasoning under Uncertainty)*, 249:125–141, (2010).


[11] Instead of focusing on satisfiability, (i.e., proving the existence of at least one model) as usually done in a SAT/SMT setting, in [6, 1] we have faced the problem of finding the whole set of models for a given fuzzy formula by re-using a previous method based on fuzzy logic programming where the formula is conceived as a goal whose derivation tree, provided by the FLOPER tool, contains in its leaves all the models of the original formula, together with other interpretations.

9. Julián, P., Moreno, G., Penabad, J.: Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. Journal of Universal Computer Science 12(11), 1679–1699 (2006)
10. Julián, P., Moreno, G., Penabad, J.: An Improved Reductant Calculus using Fuzzy Partial Evaluation Techniques. Fuzzy Sets and Systems 160, 162–181 (2009), http://dx.doi: 10.1016/j.fss.2008.05.006
11. Julián-Iranzo, P., Moreno, G., Penabad, J., Vázquez, C.: A fuzzy logic programming environment for managing similarity and truth degrees. In EPTCS, 173, pp. 71–86 (2015), http://dx.doi.org/10.4204/EPTCS.173.6
12. Julián-Iranzo, P., Moreno, G., Penabad, J., Vázquez, C.: A Declarative Semantics for a Fuzzy Logic Language Managing Similarities and Truth Degrees. In: Proc. of RuleML'16, LNCS, 9718. pp. 68–82. Springer Verlag (2016)
13. Kifer, M., Subrahmanian, V.S.: Theory of generalized annotated logic programming and its applications. Journal of Logic Programming 12, 335–367 (1992)
14. Lassez, J.L., Maher, M.J., Marriott, K.: Unification Revisited. In: Foundations of Deductive Databases and Logic Programming, pp. 587–625. Morgan Kaufmann, Los Altos, Ca. (1988)
15. Lee, R.: Fuzzy Logic and the Resolution Principle. Journal of the ACM 19(1), 119–129 (1972)
16. Li, D., Liu, D.: A fuzzy Prolog database system. John Wiley & Sons, Inc. (1990)
17. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag, Berlin (1987)
18. Medina, J., Ojeda-Aciego, M., Vojtáš, P.: Similarity-based Unification: a multi-adjoint approach. Fuzzy Sets and Systems 146, 43–62 (2004)
19. Morcillo, P.J., Moreno, G., Penabad, J., Vázquez, C.: A Practical Management of Fuzzy Truth Degrees using FLOPER. In: Proc. of RuleML'10, LNCS, 6403. pp. 20–34. Springer Verlag (2010)
20. Moreno, G., Vázquez, C.: Fuzzy logic programming in action with FLOPER. Journal of Software Engineering and Applications 7, 237–298 (2014)
21. Nguyen, H.T., Walker, E.A.: A First Course in Fuzzy Logic. Chapman & Hall, Boca Ratón, Florida (2006)
22. Rodríguez-Artalejo, M., Romero-Díaz, C.: Quantitative logic programming revisited. In: Proc. of FLOPS'08, pp. 272–288. LNCS 4989, Springer Verlag (2008)
23. Straccia, U.: Managing uncertainty and vagueness in description logics, logic programs and description logic programs. In: Proc. of 4th Int. Summer School Reasoning Web, LNCS 5224. pp. 54–103. Springer Verlag (2008)
24. Vidal, A., Bou, F., Godo, L.: An smt-based solver for continuous t-norm based logics. In: Proc. of the 6th International Conference on Scalable Uncertainty Management. LNCS, 7520, pp. 633–640 (2012)