Symbolic Unfolding versus Tuning of Similarity-based Fuzzy Logic Programs

Ginés Moreno
Department of Computing Systems
University of Castilla-La Mancha
02071 Albacete (Spain)
Email: Gines.Moreno@uclm.es

José Antonio Riaza
Department of Computing Systems
University of Castilla-La Mancha
02071 Albacete (Spain)
Email: JoseAntonio.Riaza@uclm.es

***Abstract*—FASILL introduces "*Fuzzy Aggregators and Similarity Into a Logic Language*". In its symbolic extension, called sFASILL, some truth degrees, similarity annotations and fuzzy connectives can be left unknown, so that the user can easily figure out the impact of their possible values at execution time. In this paper, we firstly adapt to this last setting a similarity-based, symbolic variant of unfolding rule (very well known in most declarative frameworks), which is based on the application of computational steps on the bodies of program rules for improving efficiency. Next, we combine it with previous tuning techniques intended to transform a symbolic sFASILL program into the concrete customized FASILL one that best satisfies the user's preferences. The improved methods have been implemented in a freely available online tool, which has served us to develop several experiments and benchmarks evidencing the good performance of the resulting system. To the best of our knowledge, our analysis is the first one combining unfolding and tuning techniques in a fully integrated fuzzy logic programming setting.**

Index Terms*—*Fuzzy Logic Programming; Similarity; Symbolic Unfolding; Tuning.

I. Introduction

This paper extends our initial approach described in [1] (presented at the *2024 IARIA Annual Congress on Frontiers in Science, Technology, Services, and Applications – IARIA Congress 2024*), by combining and enriching the fuzzy capabilities of a highly flexible programming environment developed in our research group. During the last four decades, the research field of *Fuzzy Logic Programming* has promoted the introduction of *Fuzzy Logic* [2] concepts into *Logic Programming* [3] in order to deal with vagueness in a natural way [4]. It has provided an extensive variety of logic programming dialects promoting the development of flexible real-world applications in the fields of artificial intelligence, soft-computing, semantic web, etc. Some interesting approaches focus on replacing the classic (syntactic) unification algorithm of Prolog by one based on the use of similarity/proximity relations [5][6], such as Likelog [7] and Bousi∼Prolog [8]. Similarity/proximity relations connect the elements of a set with a certain approximation degree and serve for weakening the notion of equality and, hence, to deal with vague information [9]. Other approaches modify the operational principle of pure logic programming to replace it by inference mechanisms based on fuzzy logic, which allow a wide variety of connectives and the use of a gradation of truth degrees (beyond the traditional values of *true* and *false*). Most of these systems implement the fuzzy resolution principle introduced by Lee in [10], such as Prolog-Elf [11], F-Prolog [12], generalized annotated logic programming [13], (S-)QLP [15], Fril [14], Fuzzy-Prolog [16], RFuzzy [17] and MALP [18].

Since the logic language Prolog has been fuzzified by embedding similarity relations or using fuzzy connectives for dealing with truth degrees beyond $\{true, false\}$, respectively, we have recently combined both approaches in the design of FASILL [19], whose symbolic extension (inspired by our initial experiences with MALP [20]) is called sFASILL [21]. This last symbolic language is useful for *flexibly tuning* (according to users preferences) the fuzzy components of fuzzy logic programs. A tuning problem is a pair composed by a symbolic program plus a set of test cases where users express their wishes about the expected behaviour of the final, customized program. As a very simple example, consider just one symbolic program rule like $p \leftarrow @_{aver}(\#sc, 0.8)$, where $@_{aver}$ refers to the average connective and $\#sc$ is a *symbolic constant* representing an unknown truth degree, together with only a test case of the form $0.6 \rightarrow p$, where a user indicates that (s)he wants to obtain 0.6 when evaluating p. Although this tuning problem has the trivial solution of replacing $\#sc$ by 0.4, in the general case, it is not easy to reach good (usually approximate) solutions when the number of program rules, symbolic constants and test cases grow more and more.

In this paper, we will collect from [21][22] two tuning strategies showing that by "partially" executing symbolic sFASILL programs and then replacing the unknown values and connectives (on their program rules and associated similarity relations) by concrete ones, gives the same result than replacing these values and connectives in the original sFASILL program and, then, fully executing the resulting FASILL program. So, sFASILL programs can be used to automatically tune and synthesize a FASILL program w.r.t. a given set of test cases, thus easing what is considered the most difficult part of the process: the specification of the truth/similarity degrees and connectives in the program. Although there exist other approaches, which are able to *tune* fuzzy truth degrees and connectives [23][24][25], none of them manage similarity relations as the tuning technique we describe in [21] does. Let us mention that we have used sFASILL and its tuning engine for developing two real world applications in the fields of the semantic web [26] and neural networks [27].

Besides this, unfolding is a well-known and widely used semantics-preserving program transformation rule, which is able to improve programs, generating more efficient code. The unfolding transformation traditionally considered in pure logic

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

programming consists in the replacement of a program clause C by the set of clauses obtained after applying a computation step in all its possible forms on the body of C [28][29].

In order to briefly illustrate the essence and benefits of the transformation, consider a very simple Prolog program containing a clause, say $p(X)$:-$q(X)$, and a fact, say $q(a)$, for defining two (crisp, not fuzzy) predicates, p and q. It is easy to see that both rules must be used in two computational steps for successfully executing a goal like $p(a)$. Alternatively, we can unfold the first clause by applying a computational step on its body $q(X)$ (using the fact $q(a)$) and next instantiating the head with the achieved substitution $\{X/a\}$. Then, the new unfolded rule is just the simple fact $p(a)$, which must be used in only one computational step (instead of two, as before) to solve goal $p(a)$. This very simple example reveals that all computational steps applied at unfolding time *remain compiled* on unfolded rules forever, and hence, those steps have no longer to be repeated in all subsequent executions of the transformed programs. This justifies why unfolding is able to improve the efficiency of transformed programs by accelerating their computational behaviour.

In [30][31], we successfully adapted such operation to fuzzy logic programs dealing with lattices of truth degrees and similarity relations, but this type of unfolding was not symbolic yet. On the contrary, in [32][33] we defined a symbolic version of the transformation but in absence of similarities. Inspired by both works, in [1] we have recently fused both approaches in the definition of a similarity-based symbolic transformation. Now, we extend such work by using this transformation as a pre-process of our tuning engines in order to improve the performance of the resulting system. In this paper we describe and use a freely available online tool ([34]) for developing some revealing experiments, benchmarks and analisys of our approach.

The structure of this paper is as follows. After summarizing, in Section II, the syntax of FASILL and sFASILL, in Section III we detail how to execute and unfold such programs. Next, Section IV summarizes two tuning engines, which are combined with unfolding in Section V, also analyzing its performance and practicability. Finally, we conclude and propose future work in Section VI.

II. THE FASILL LANGUAGE AND ITS SYMBOLIC EXTENSION

In this work, given a complete lattice L, we consider a first order language $\mathcal{L}_L$ built upon a signature Σ_L, that contains the elements of a countably infinite set of variables $\mathcal{V}$, function and predicate symbols (denoted by $\mathcal{F}$ and Π, 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 Σ_L^T and connectives Σ_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* (atom, for short).
- $\varsigma(e_1, \ldots, e_n)$, if $e_1, \ldots, e_n$ are well-formed formulas and ς is an n-ary connective with truth function $[\![\varsigma]\!] : L^n \mapsto L$.

Definition 1 (Complete Lattice)**.** A *complete lattice* is a partially ordered set $(L, \leq)$ such that every subset S of L has infimum and supremum elements. Then, it is a bounded lattice, i.e., it has bottom and top elements, denoted by $\bot$ and $\top$, respectively.

Example 1. In this paper, we use the lattice $([0,1], \leq)$, where $\leq$ is the usual ordering relation on real numbers, and three sets of conjunctions/disjunctions corresponding to the fuzzy logics of Gödel, Łukasiewicz and Product (with different capabilities for modelling *pessimistic*, *optimistic* and *realistic scenarios*), defined in Figure 1. It is possible to also include other fuzzy connectives (aggregators) like the arithmetical and geometrical averages, say $@_{\texttt{aver}}(x, y) \triangleq (x+y)/2$ and $@_{\texttt{geom}}(x, y) \triangleq \sqrt{xy}$, or the linguistic modifier $@_{\texttt{very}}(x) \triangleq x^2$.

Definition 2 (Similarity Relation)**.** Given a domain $\mathcal{U}$ and a lattice L with a fixed t-norm $\wedge$, a *similarity relation* $\mathcal{R}$ is a fuzzy binary relation on $\mathcal{U}$, that is, a fuzzy subset on $\mathcal{U} \times \mathcal{U}$ (namely, a mapping $\mathcal{R} : \mathcal{U} \times \mathcal{U} \to L$) fulfilling the following properties: reflexive $\forall x \in \mathcal{U}, \mathcal{R}(x, x) = \top$, symmetric $\forall x, y \in \mathcal{U}, \mathcal{R}(x, y) = \mathcal{R}(y, x)$, and transitive $\forall x, y, z \in \mathcal{U}, \mathcal{R}(x, z) \geq \mathcal{R}(x, y) \wedge \mathcal{R}(y, z)$.

The fuzzy logic language FASILL relies on complete lattices and similarity relations [19]. We are now ready for summarizing its *symbolic* extension where, in essence, we allow some undefined values (truth degrees) and connectives in program rules as well as in the associated similarity relation, so that these elements can be systematically computed afterwards. The symbolic extension of FASILL we initially presented in [21] is called sFASILL.

Given a complete lattice L, we consider an augmented signature $\Sigma_L^\#$ producing an augmented language $\mathcal{L}_L^\# \supseteq \mathcal{L}_L$, which may also include a number of symbolic values and symbolic connectives, which do not belong to L. Symbolic objects are usually denoted as $o^\#$ with a superscript $\#$ and, in our tool, their identifiers always start with $\#$. An $L^\#$-$expression$ is now a well-formed formula of $\mathcal{L}_L^\#$, which is composed by values and connectives from L as well as by symbolic values and connectives. We let $\mathsf{exp}_L^\#$ denote the set of all $L^\#$-expressions in $\mathcal{L}_L^\#$. Given a $L^\#$-expression E, $[\![E]\!]$ refers to the new $L^\#$-expression obtained after evaluating as much as possible the connectives in E. Particularly, if E does not contain any symbolic value or connective, then $[\![E]\!] = v \in L$.

In the following, we consider *symbolic substitutions* that are mappings from symbolic values and connectives to expressions over $\Sigma_L^T \cup \Sigma_L^C$. We let $\mathsf{sym}(o^\#)$ denote the symbolic values and connectives in $o^\#$. Given a symbolic substitution Θ for $\mathsf{sym}(o^\#)$, we denote by $o^\#\Theta$ the object that results from $o^\#$ by replacing every symbolic symbol $e^\#$ by $e^\#\Theta$.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

$\&_{\texttt{prod}}(x,y) \triangleq x * y$	$\|_{\texttt{prod}}(x,y) \triangleq x + y - xy$	*Product logic*
$\&_{\texttt{godel}}(x,y) \triangleq \min(x,y)$	$\|_{\texttt{godel}}(x,y) \triangleq \max(x,y)$	*Gödel logic*
$\&_{\texttt{luka}}(x,y) \triangleq \max(0, x+y-1)$	$\|_{\texttt{luka}}(x,y) \triangleq \min(x+y,1)$	*Łukasiewicz logic*

Fig. 1. Conjunctions and disjunctions of three different fuzzy logics over $([0,1],\leq)$.

Definition 3 (Symbolic Similarity Relation)**.** Given a domain $\mathcal{U}$ and a lattice L with a fixed —possibly symbolic— t-norm $\wedge$, a *symbolic similarity relation* is a mapping $\mathcal{R}^{\#} : \mathcal{U} \times \mathcal{U} \rightarrow \exp_L^{\#}$ such that, for any symbolic substitution Θ for $\mathsf{sym}(\mathcal{R}^{\#})$, the result of fully evaluating all L-expressions in $\mathcal{R}^{\#}\Theta$, say $[\![\mathcal{R}^{\#}\Theta]\!]$, is a similarity relation.

Definition 4 (Symbolic Rule and Symbolic Program)**.** Let L be a complete lattice. A *symbolic rule* over L is a formula $A \leftarrow \mathcal{B}$, where the following conditions hold:

- A is an atomic formula of $\mathcal{L}_L$ (the head of the rule);
- $\leftarrow$ is an implication from L or a symbolic implication;
- $\mathcal{B}$ (the body of the rule) is a symbolic goal, i.e., a well-formed formula of $\mathcal{L}_L^{\#}$;

A sFASILL *program* is a tuple $\mathcal{P}^{\#} = \langle \Pi^{\#}, \mathcal{R}^{\#}, L\rangle$ where $\Pi^{\#}$ is a set of symbolic rules, $\mathcal{R}^{\#}$ is a symbolic similarity relation between the elements of the signature Σ of $\Pi^{\#}$, and L is a complete lattice.

Example 2. Consider a symbolic sFASILL program $\mathcal{P}^{\#} = \langle \Pi^{\#}, \mathcal{R}^{\#}, L\rangle$ based on lattice $L = ([0,1],\leq)$, where $\Pi^{\#}$ is the following set of symbolic rules:

$$\Pi^{\#} = \begin{cases} R_1 : & vanguardist(rizt) \leftarrow 0.9 \\ R_2 : & elegant(hydropolis) \leftarrow s_3^{\#} \\ R_3 : & close(hydropolis, taxi) \leftarrow 0.7 \\ R_4 : & good_hotel(x) \leftarrow \\ & \quad @_{s4}^{\#}(elegant(x), @_{very}(close(x, metro))) \end{cases}$$

Note here that we leave unknown the level in which the hotel $hydropolis$ is more or less elegant (see the symbolic constant $s_3^{\#}$ in the second fact) as well as which should be the most appropriate connective for combining two features required on good hotels (see the symbolic constant $@_{s4}^{\#}$ in the body of the fourth rule).

The symbolic similarity relation $\mathcal{R}^{\#}$ on $\mathcal{U} = \{vanguardist, elegant, modern, metro, taxi, bus\}$, is represented by the graph shown in Figure 2 (a matrix can be also used to represent this concept).

This symbolic similarity relation $\mathcal{R}^{\#}$ has been obtained after applying the closure algorithm we initially introduced in [21], which is inspired by [35][36][37] and, in essence, is an adaptation of the classical Warshall's algorithm for computing transitive closures. In this particular example, we have selected the symbolic t-norm $\&_{s2}^{\#}$ and the following set of similarity equations: $elegant \sim modern = s_0^{\#}$, $modern \sim vanguardist = 0.9$, $metro \sim bus = 0.5$ and $bus \sim taxi = s_1^{\#}$.

In what follows, we plan to introduce and combine the unfolding and tuning techniques we have developed in the last years for reinforcing their power in this novel, symbolic plus similarity-based fuzzy logic setting.

III. Running and Unfolding sFASILL Programs

As a logic language, sFASILL inherits the concepts of substitution, unifier and most general unifier (mgu) from pure logic programming, but extending some of them in order to cope with similarities, as Bousi∼Prolog [8] does, where the concept of most general unifier is replaced by the one of *weak most general unifier* (w.m.g.u.). One step beyond, in [21] we extended again this notion by referring to *symbolic weak most general unifiers* (s.w.m.g.u.) and a symbolic weak unification algorithm was introduced to compute them. Roughly speaking, the *symbolic weak unification algorithm* states that two *expressions* (i.e., terms or atomic formulas) $f(t_1,\ldots,t_n)$ and $g(s_1,\ldots,s_n)$ weakly unify if the root symbols f and g are close with a certain —possibly symbolic— degree (i.e., $\mathcal{R}^{\#}(f,g) = r \neq \bot$) and each of their arguments t_i and s_i weakly unify. Therefore, there is a symbolic weak unifier for two expressions even if the symbols at their roots are not syntactically equal ($f \not\equiv g$).

More technically, the symbolic weak unification algorithm can be seen as an reformulation/extension of the ones appearing in [6] (since now we manage arbitrary complete lattices) and [19][8] (because now we deal with symbolic similarity relations). In essence, the *symbolic weak most general unifier* of two expressions $\mathcal{E}_1$ and $\mathcal{E}_2$, say $wmgu^{\#}(\mathcal{E}_1,\mathcal{E}_2) = \langle\sigma, E\rangle$, is the simplest *symbolic substitution* σ of $\mathcal{E}_1$ and $\mathcal{E}_2$ together with its *symbolic unification degree* E verifying that $E = \hat{\mathcal{R}}(E_1\sigma, E_2\sigma)$.

Example 3. Given the complete lattice $L = ([0,1],\leq)$ of Example 1 and the symbolic similarity relation $\mathcal{R}^{\#}$ of Example 2, we can use the symbolic t-norm $\&_{s2}^{\#}$ for computing the following two symbolic symbolic weak most general unifiers: $wmgu^{\#}(modern(taxi), vanguardist(bus)) = \langle\{\}, 0.9 \;\; \&_{s2}^{\#} \;\; s_1^{\#}\rangle$ and $wmgu^{\#}(close_to(X, taxi), close_to(ritz, bus)) = \langle\{X/ritz\}, s_1^{\#}\rangle$

In order to describe the procedural semantics of the sFASILL language, in the following, we denote by $\mathcal{C}[A]$ a formula where A is a sub-expression (usually an atom), which occurs in the –possibly empty– context $\mathcal{C}[]$ whereas $\mathcal{C}[A/A']$ means the replacement of A by A' in the context $\mathcal{C}[]$. Moreover, $\mathcal{V}ar(s)$ denotes the set of distinct variables occurring in the syntactic object s and $\theta[\mathcal{V}ar(s)]$ refers to the substitution obtained from θ by restricting its domain to $\mathcal{V}ar(s)$. In the next definition, we always consider that A is the

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

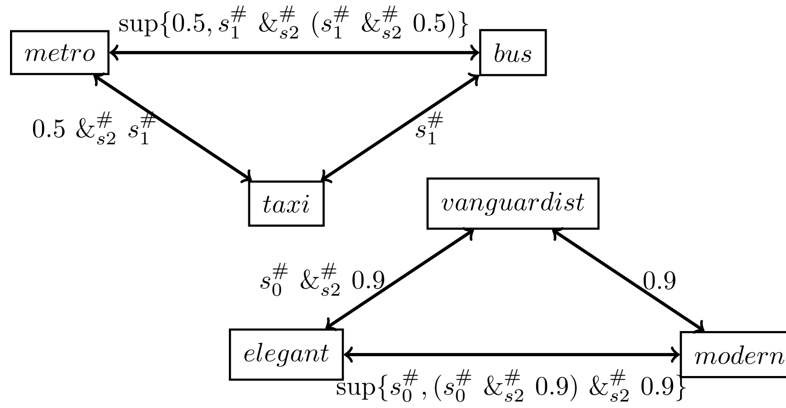

Fig. 2. Example of symbolic similarity relation $\mathcal{R}^\#$.

selected atom in a goal $\mathcal{Q}$, L is the complete lattice associated to $\Pi^\#$ and, as usual, rules are renamed apart:

Definition 5 (Computational Step)**.** Let $\mathcal{Q}$ be a goal and σ a substitution. The pair $\langle \mathcal{Q}; \sigma \rangle$ is a *state*. Given a symbolic program $\langle \Pi^\#, \mathcal{R}^\#, L \rangle$ and a (possibly symbolic) t-norm $\wedge$ in L, a *computation* is formalized as a state transition system, whose transition relation $\leadsto$ is the smallest relation satisfying these rules:

1) *Successful step* (denoted as $\overset{\text{SS}}{\leadsto}$):

$$\frac{\langle \mathcal{Q}[A], \sigma \rangle \quad A' \leftarrow \mathcal{B} \in \Pi^\# \quad wmgu^\#(A, A') = \langle \theta, E \rangle \quad E \neq \bot}{\langle \mathcal{Q}[A/E \wedge \mathcal{B}]\theta, \sigma\theta \rangle} \text{ SS}$$

2) *Failure step* (denoted as $\overset{\text{FS}}{\leadsto}$):

$$\frac{\langle \mathcal{Q}[A], \sigma \rangle \quad \nexists A' \leftarrow \mathcal{B} \in \Pi^\# : wmgu^\#(A, A') = \langle \theta, E \rangle}{\langle \mathcal{Q}[A/\bot], \sigma \rangle} \text{ FS}$$

3) *Interpretive step* (denoted as $\overset{\text{IS}}{\leadsto}$):

$$\frac{\langle \mathcal{Q}; \sigma \rangle \text{ where } \mathcal{Q} \text{ is a } L^\#\text{-expression}}{\langle [\![\mathcal{Q}]\!]; \sigma \rangle} \text{ IS}$$

Definition 6 (Derivation and Symbolic Fuzzy Computed Answer)**.** A *derivation* is a sequence of arbitrary length $\langle \mathcal{Q}; id \rangle \leadsto^* \langle \mathcal{Q}'; \sigma \rangle$. When $\mathcal{Q}'$ is an $L^\#$-expression that cannot be further reduced, $\langle \mathcal{Q}'; \sigma' \rangle$, where $\sigma' = \sigma[\mathcal{V}ar(Q)]$, is called a *symbolic fuzzy computed answer* (sfca). Also, if $\mathcal{Q}'$ is a concrete value of L, we say that $\langle \mathcal{Q}'; \sigma' \rangle$ is a *fuzzy computed answer* (fca).

The following example illustrates the operational semantics of sFASILL.

Example 4. Let $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$ be the program from Example 2. It is possible to perform the following derivation for $\mathcal{P}^\#$ and goal $\mathcal{Q} = good_hotel(x)$ obtaining the sfca $\langle \mathcal{Q}_1; \sigma_1 \rangle = \langle @_{s4}^\#(\&_{s2}^\#(\&_{s2}^\#(s_0^\#, 0.9), 0.9), 0.0); \{x/ritz\} \rangle$:

$$\begin{array}{ll}
\langle \underline{good_hotel(x)}, id \rangle & \overset{\text{SS}}{\leadsto}{}^{R4} \\
\langle @_{s4}^\#(\underline{elegant(x_1)}, @_{very}(close(x_1, metro))), \{x/x_1\} \rangle & \overset{\text{SS}}{\leadsto}{}^{R1} \\
\langle @_{s4}^\#(\&_{s2}^\#(\&_{s2}^\#(s_0^\#, 0.9), 0.9), & \\
\quad @_{very}(\underline{close(ritz, metro)})), \{x/ritz\} \rangle & \overset{\text{FS}}{\leadsto} \\
\langle @_{s4}^\#(\&_{s2}^\#(\&_{s2}^\#(s_0^\#, 0.9), 0.9), \underline{@_{very}(0.0)}), \{x/ritz\} \rangle & \overset{\text{IS}}{\leadsto} \\
\langle @_{s4}^\#(\&_{s2}^\#(\&_{s2}^\#(s_0^\#, 0.9), 0.9), 0.0), \{x/ritz\} \rangle &
\end{array}$$

Apart from this derivation, there exists a second one ending with the alternative sfca $\langle \mathcal{Q}_2; \sigma_2 \rangle = \langle @_{s4}^\#(s_3^\#, @_{very}(\&_{s2}^\#(\&_{s2}^\#(0.5, s_1^\#), 0.7))); \{x/hydropolis\} \rangle$ associated to the same goal. Both derivations are included in the tree shown in Figure 3. Observe the presence of symbolic constants coming from the symbolic similarity relation, which contrast with our precedent work [20].

Now, let $\Theta = \{s_0^\#/0.8, s_1^\#/0.8, \&_{s2}^\#/\&_{\texttt{luka}}, s_3^\#/1.0, @_{s4}^\#/@_{\texttt{aver}}\}$ be a symbolic substitution that can be used for instantiating the previous sFASILL program in order to obtain a non-symbolic, fully executable FASILL program. This substitution can be automatically obtained by the tuning engines we will describe in Section IV ([21]) after introducing a couple of test cases (i.e., `0.4-> good_hotel(hydropolis)` and `0.6-> good_hotel(ritz)`), which represent the desired degrees for two goals accordingly to the user preferences.

Now we are ready to introduce the similarity-based symbolic unfolding transformations relying on the operational semantics described so far.

Definition 7 (Symbolic Unfolding)**.** Let $\mathcal{P}^\# = \langle \Pi^\#, \mathcal{R}^\#, L \rangle$ be a sFASILL program and $R : (H \leftarrow \mathcal{B}) \in \Pi^\#$ be a rule (with non-empty body $\mathcal{B}$). Then, the *symbolic unfolding* of rule R in program $\mathcal{P}^\#$ is the new sFASILL program $\mathcal{P}'^\# = \langle \Pi'^\#, \mathcal{R}^\#, L \rangle$, where $\Pi'^\# = (\Pi^\# - \{R\}) \cup \{H\sigma \leftarrow \mathcal{B}' \mid \langle \mathcal{B}; id \rangle \leadsto \langle \mathcal{B}'; \sigma \rangle\}$.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

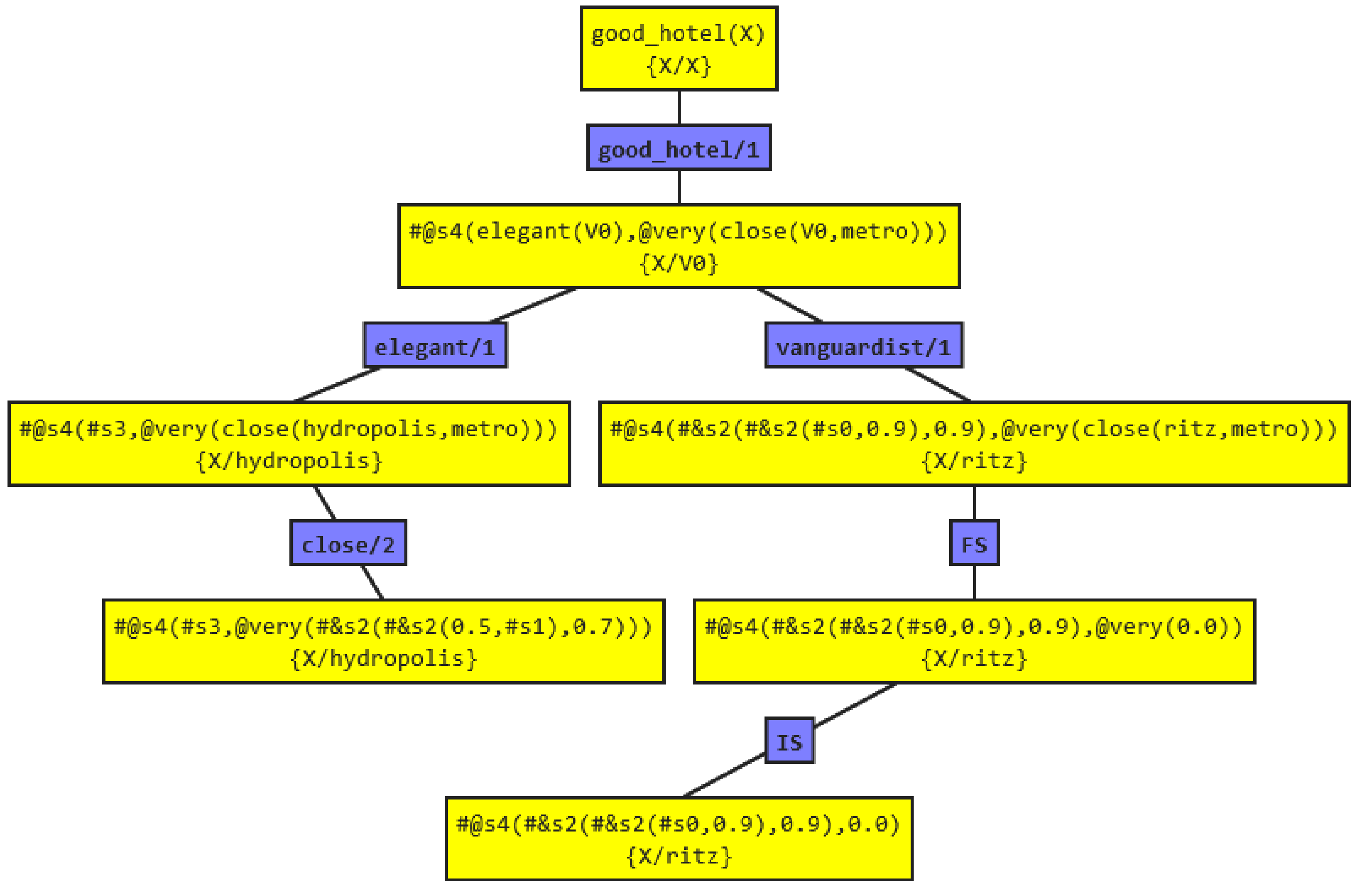

Fig. 3. Screenshot of the FASILL online tool depicting a symbolic derivation tree.

Example 5. Let us built a transformation sequence where each sFASILL program in the sequence is obtained from the immediately preceding one by applying symbolic unfolding, except the initial one $\mathcal{P}_0^\# = \langle \Pi_0^\#, \mathcal{R}^\#, L\rangle$, which, in our case, is the one illustrated in Example 2, that is:

$$\Pi_0^\# = \begin{cases} R_1: & vanguardist(rizt) \leftarrow 0.9 \\ R_2: & elegant(hydropolis) \leftarrow s_3^\# \\ R_3: & close(hydropolis, taxi) \leftarrow 0.7 \\ R_4: & good_hotel(x) \leftarrow \\ & \quad @_{s4}^\#(elegant(x), @_{very}(close(x, metro))) \end{cases}$$

Program $\mathcal{P}_1^\# = \langle \Pi_1^\#, \mathcal{R}^\#, L\rangle$ is obtained after unfolding rule R_4 (with selected atom $elegant(x)$) by applying a $\overset{SS}{\leadsto}$ step with rules R_1 and R_2:

$$\Pi_1^\# = \begin{cases} R_1: & vanguardist(rizt) \leftarrow 0.9 \\ R_2: & elegant(hydropolis) \leftarrow s_3^\# \\ R_3: & close(hydropolis, taxi) \leftarrow 0.7 \\ R_{41}: & good_hotel(ritz) \leftarrow @_{s4}^\#(\\ & \quad \&_{s2}^\#(\&_{s2}^\#(s_0^\#, 0.9), 0.9), \\ & \quad @_{very}(close(ritz, metro))) \\ R_{42}: & good_hotel(hydropolis) \leftarrow \\ & \quad @_{s4}^\#(s_3^\#, @_{very}(close(hydropolis, metro))) \end{cases}$$

After unfolding rule R_{41} (with selected atom $close(ritz, metro)$) by applying a $\overset{FS}{\leadsto}$ step, we obtain program $\mathcal{P}_2^\# = \langle \Pi_2^\#, \mathcal{R}^\#, L\rangle$:

$$\Pi_2^\# = \begin{cases} R_1: & vanguardist(rizt) \leftarrow 0.9 \\ R_2: & elegant(hydropolis) \leftarrow s_3^\# \\ R_3: & close(hydropolis, taxi) \leftarrow 0.7 \\ R_{41F}: & good_hotel(ritz) \leftarrow \\ & \quad @_{s4}^\#(\&_{s2}^\#(\&_{s2}^\#(s_0^\#, 0.9), 0.9), @_{very}(0.0)) \\ R_{42}: & good_hotel(hydropolis) \leftarrow \\ & \quad @_{s4}^\#(s_3^\#, @_{very}(close(hydropolis, metro))) \end{cases}$$

When unfolding rule R_{42} (with selected atom $close(hydropolis, metro)$) by applying a $\overset{SS}{\leadsto}$ step with rule R_3, we reach the program $\mathcal{P}_3^\# = \langle \Pi_3^\#, \mathcal{R}^\#, L\rangle$:

$$\Pi_3^\# = \begin{cases} R_1: & vanguardist(rizt) \leftarrow 0.9 \\ R_2: & elegant(hydropolis) \leftarrow s_3^\# \\ R_3: & close(hydropolis, taxi) \leftarrow 0.7 \\ R_{41F}: & good_hotel(ritz) \leftarrow \\ & \quad @_{s4}^\#(\&_{s2}^\#(\&_{s2}^\#(s_0^\#, 0.9), 0.9), @_{very}(0.0)) \\ R_{423}: & good_hotel(hydropolis) \leftarrow \\ & \quad @_{s4}^\#(s_3^\#, @_{very}(\&_{s2}^\#(\&_{s2}^\#(0.5, s_1^\#), 0.7))) \end{cases}$$

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
1 vanguardist(rizt) <- 0.9.
2 elegant(hydropolis) <- #s3.
3 close(hydropolis,taxi) <- 0.7.
4 good_hotel(X) <- #@s4(elegant(X), @very(close(X, metro))).
```

Linearize program | Extend program | Unfold program

```
1 vanguardist(rizt) <- 0.9.
```

```
2 elegant(hydropolis) <- #s3.
```

```
3 close(hydropolis,taxi) <- 0.7.
```

```
4 good_hotel(X) <- #@s4(elegant(X),@very(close(X,metro))).
```

(a) Original sFASILL program before being transformed.

```
1 vanguardist(rizt) <- 0.9.
2 elegant(hydropolis) <- #s3.
3 close(hydropolis,taxi) <- 0.7.
4 good_hotel(hydropolis) <- #@s4(#s3,@very(close(hydropolis,metro))).
5 good_hotel(rizt) <- #@s4(#&s2(#&s2(#s0,0.9),0.9),@very(close(rizt,metro))).
```

(b) sFASILL program obtained after unfolding the last program rule.

Fig. 4. The FASILL online tool unfolding a symbolic program.

Finally, by unfolding rule R_{41F} (with selected expression $@_{very}(0.0)$) after applying a $\overset{\text{IS}}{\leadsto}$ step, we obtain the final program $\mathcal{P}_4^\# = \langle \Pi_4^\#, \mathcal{R}^\#, L\rangle$:

$$\Pi_4^\# = \begin{cases} R_1: & vanguardist(rizt) \leftarrow 0.9 \\ R_2: & elegant(hydropolis) \leftarrow s_3^\# \\ R_3: & close(hydropolis, taxi) \leftarrow 0.7 \\ R_{41FI}: & good_hotel(ritz) \leftarrow \\ & \quad @_{s4}^\#(\&_{s2}^\#(\&_{s2}^\#(s_0^\#, 0.9), 0.9), 0.0) \\ R_{423}: & good_hotel(hydropolis) \leftarrow \\ & \quad @_{s4}^\#(s_3^\#, @_{very}(\&_{s2}^\#(\&_{s2}^\#(0.5, s_1^\#), 0.7))) \end{cases}$$

In the previous example, it is easy to see that each program in the sequence produces the same set of sfca's for a given goal but reducing the length of derivations. For instance, the derivation performed w.r.t. the original program $\mathcal{P}_0^\#$ illustrated in Example 4, can be emulated in the final program $\mathcal{P}_4^\#$ with just one computational step (instead of four) as:

$$\langle good_hotel(x); id\rangle \overset{\text{SS}}{\leadsto}{}^{R41FI} \langle @_{s4}^\#(\&_{s2}^\#(\&_{s2}^\#(s_0^\#, 0.9), 0.9), 0.0); \{x/ritz\}\rangle.$$

IV. Tuning Techniques for sFASILL Programs

We start this section by summarizing the automated technique for tuning sFASILL programs that we initially presented in [21][22].

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 can be easily obtained. In other cases, though, the most appropriate values and/or connectives depend on subjective notions and,

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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.

The first action for initializing the tuning process in the FASILL online tool obviously consists in introducing a set of test cases. The tune area of our online tool is shown in Figure 5a. Each test case appears in a different line with syntax: $r \rightarrow \mathcal{Q}$, where r is the desired truth degree for the fca associated to query $\mathcal{Q}$ (which obviously does not contain symbolic constants). For instance, in our running example we can introduce the following three test cases:

```
0.60 -> modern(hydropolis).
0.45 -> good_hotel(ritz).
0.38 -> good_hotel(hydropolis).
```

Then, users need to select a tuning method and click on the Tune program button to proceed with the tuning process. The precision of the technique depends on the set of symbolic substitutions considered at tuning time. So, for assigning values to the symbolic constants, our tool takes into account all the truth values defined on a members/1 predicate (which in our case is declared as members([0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1])) as well as the set of connectives defined in the lattice associated to the program, which in our running example coincides with the three conjunction and disjunction connectives based on the so-called *Product*, *Gödel* and *Łukasiewicz* logics, as shown in Figure 1, and the arithmetic/geometric average aggregators defined in Example 1. Obviously, the larger the domain of values and connectives is, the more precise the results are (but the algorithm is more expensive, of course).

As we have seen in this work, symbolic programs can be seen as *partial specifications* of fuzzy programs, where some elements on program rules have not been clearly identified yet. Here, we assume that test cases are provided a priori by users in order to establish well known information about a particular domain and, for this reason, they will usually be expressed as ground atoms with associated (desired) truth degrees. It is easy to think that the three test cases used in our running example (see Figure 5a) refers to user's preferences that could be collected, for instance, by means of satisfaction questionnaires or any other alternative kind of statistically-based analysis. However, our system admits test cases non necessarily based on ground atoms,

For tuning an sFASILL program, we have implemented several methods in previous works, including some very efficient versions based on SAT/SMT solvers [38], which are out of the scope of this paper since they only work with connectives whose truth functions are linear. In this work, we focus on the two more general methods described in [21], which exhibit different run-times depending on when and where symbolic substitutions are applied (to symbolic programs or to sfca's):

- Basic: The basic method is based on applying each symbolic substitution to the original sFASILL program and then fully executing the resulting instantiated FASILL programs. This method is represented by Algorithm 1.
- Symbolic: In this version, symbolic substitutions are directly applied to sfca's and thus, only interpretive (but neither successful nor failure) steps are repeatedly executed on the instantiated fca's. This method is represented by Algorithm 2.

Both algorithms use thresholding techniques for prematurely disregarding computations leading to non significant solutions. In [19], [39], [40], [41] we document some interesting results achieved in our research group when designing sophisticated tools for manipulating fuzzy logic programs and, now, we apply the same techniques for improving the efficiency of the basic and symbolic tuning methods under comparison in this section.

Algorithm 1: Basic Tuning for FASILL programs.

Data: A symbolic program $\mathcal{P}^{\#}$ and a set of test cases $\{\langle \mathcal{G}_1, v_1\rangle, \ldots, \langle \mathcal{G}_k, v_k\rangle\}$.
Result: A symbolic substitution Θ.
Consider a finite number of symbolic substitutions $\Theta_1, \ldots, \Theta_n$ for $\mathsf{sym}(\mathcal{P}^{\#}) \cup \bigcup_{i=1}^{k} \mathsf{sym}(\mathcal{G}_i)$;
$\tau \leftarrow +\infty$;
foreach $i \in \{1, \ldots, n\}$ **do**
  $\mathcal{P}_i \leftarrow \mathcal{P}^{\#}\Theta_i$;
  $\epsilon \leftarrow 0$;
  foreach $j \in \{1, \ldots, k\}$ **do**
    Compute the fca $\langle \mathcal{G}_j\Theta_i; id\rangle \rightsquigarrow^* \langle v_{i,j}; \theta_{i,j}\rangle$ in $\mathcal{P}_i$;
    $\epsilon \leftarrow \epsilon + d(v_{i,j}, v_j)$;
    if $\epsilon \geq \tau$ **then**
      break;
    end
  end
  if $\epsilon < \tau$ **then**
    $\tau \leftarrow \epsilon$;
    $\Theta \leftarrow \Theta_i$;
  end
end
return Θ;

In a the symbolic algorithm and, as we detected in [21], we must take care when attaching concrete values to the symbolic constants appearing in the symbolic similarity relations. Beyond the simpler case of assigning the $\perp$ truth degree to at least a symbolic constant, it is also possible to conceive other non safe symbolic substitutions linking symbolic constant to values bigger than $\perp$. For instance, this is the case of $\Theta = \{s_1^{\#}/0.4, \&_{s2}^{\#}/\&_{\texttt{luka}}, \ldots\}$ in our running example, because if, in particular, we apply this symbolic substitution to $\mathcal{R}^{\#}(taxi, metro) = 0.5 \ \&_{s2}^{\#} \ s_1^{\#}$ we have that $[\![\mathcal{R}^{\#}(taxi, metro)\Theta]\!] = 0.5 \ \&_{\texttt{luka}} \ 0.4 = max(0, 0.5 + 0.4 - 1) = max(0, -0.1) = 0$, which implies that Θ is not a

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Algorithm 2: Symbolic Tuning for FASILL programs.

Data: A symbolic program $\mathcal{P}^{\#}$ and a set of test cases $\{\langle \mathcal{G}_1, v_1 \rangle, \ldots, \langle \mathcal{G}_k, v_k \rangle\}$.
Result: A symbolic substitution Θ.
foreach $i \in \{1, \ldots, k\}$ **do**
  Compute the sfca $\langle \mathcal{G}_i, id \rangle \rightsquigarrow^* \langle \mathcal{G}'_i, \theta_i \rangle$ in $\mathcal{P}^{\#}$;
end
Consider a finite number of symbolic substitutions $\Theta_1, \ldots, \Theta_n$ for $\bigcup_{i=1}^{k} \mathsf{sym}(\mathcal{G}'_i)$;
$\tau \leftarrow +\infty$;
foreach $i \in \{1, \ldots, n\}$ **do**
  if Θ_i *is* $\mathcal{R}$*-safe* **then**
    $\epsilon \leftarrow 0$;
    foreach $j \in \{1, \ldots, k\}$ **do**
      Compute the fca $\langle \mathcal{G}'_j \Theta_i ; \theta_j \rangle \overset{\text{IS}}{\rightsquigarrow}^* \langle v_{i,j} ; \theta_j \rangle$ in $\mathcal{P}^{\#}$;
      $\epsilon \leftarrow \epsilon + d(v_{i,j}, v_j)$;
      if $\epsilon \geq \tau$ **then**
        break;
      end
    end
    if $\epsilon < \tau$ **then**
      $\tau \leftarrow \epsilon$;
      $\Theta \leftarrow \Theta_i$;
    end
  end
end
return Θ;

safe symbolic substitution w.r.t. $\mathcal{R}^{\#}$. This example justifies why, in Algorithm 2, we the use the concept of *safe symbolic substitution* introduced in [21] (Definition 7). In essence, given a symbolic similarity relation $\mathcal{R}^{\#}$ on a domain $\mathcal{U}$ and a symbolic substitution Θ, we say that Θ is a *safe symbolic substitution* w.r.t. $\mathcal{R}^{\#}$ if, for all $x, y \in \mathcal{U}$ such that $\mathcal{R}^{\#}(x, y)$ is an $L^{\#}$-expression containing at least a symbolic constant, then $[\![\mathcal{R}^{\#}(x, y)\Theta]\!] \neq \bot$.

As seen in Figure 5b, the system reports the best symbolic substitution obtained after performing the tuning process, as well as its associated deviation. In our case, the best symbolic substitution is $\Theta = \{s_0^{\#}/0.9, s_1^{\#}/0.4, \&_{s2}^{\#}/\&_{\texttt{godel}}, s_3^{\#}/0.6, @_{s4}^{\#}/@_{\texttt{aver}}\}$, which has no deviation w.r.t. the three test cases (although this is not always the general case) and hence, once Θ is applied to the original sFASILL program and, after executing a goal like `good_hotel(X)` w.r.t. the final, tuned FASILL program, we obtain the fca's `<0.45, {X/ritz}>` and `<0.38, {X/hydropolis}>`, while the execution of goal `modern(hydropolis)` returns `<0.60, {}>`, which coincide with the preferences expressed in the three test cases, as wanted.

Let us finish this section by mentioning that the solutions achieved by both the basic and symbolic algorithms is always the same for all programs described in the previous section (it doesn't matter if they have been more or less unfolded), but the time required to reach the best symbolic substitutions can drastically change, as we are going to compare and explain in the following section.

V. Combining Unfolding and Tuning techniques. Performance, Evaluation and Discussion

Observe in Figure 5b that the FASILL online tool also reports the time required to find the solution of a tuning program, which in our running example is close to 3.7 seconds when applying the symbolic tuning algorithm, but this time almost doubles when selecting the basic method. Let us start this section by explaining the tuning session sketched in Table I, where each row refers to a different symbolic substitution fully checked at tuning time and the first five columns indicate the concrete values and connectives linked to the five symbolic constants in our running example. Columns labelled with t_i and ϵ_i, $1 \leq i \leq 3$, indicate the achieved truth degree and partial deviation, respectively, associated to the evaluation of each one of the three test cases, being the last column ϵ the global deviation produced by the corresponding symbolic substitution.

Since there are two possible aggregators to select for $@_{s4}^{\#}$, three conjunctions for mapping $\&_{s2}^{\#}$ and eleven numeric values as choices for $s_0^{\#}$, $s_1^{\#}$ and $s_3^{\#}$, the system considers 7986 symbolic substitution as candidates. Obviously, we can not display so many rows in the table but, fortunately, there is no need to do so thanks to the benefits introduced by thresholding when evaluating the test cases. In fact, the small subset of rows in the table is sufficient to illustrate our technique, since only these symbolic substitutions are the only ones that both tuning algorithms (basic and symbolic) apply to all test cases for fully evaluating them (any other candidate is abruptly ruled out as soon as possible). Technically, observe in both tuning algorithms that threshold τ (whose initial value is $+\infty$), is dynamically updated in sentence "**if** $(\epsilon < \tau) \ldots$" whenever the systems reaches a symbolic substitution Θ_i whose global deviation is better than the one of the previously proposed as solution Θ, and also τ is used in sentence "**if** $(\epsilon \geq \tau) \ldots$" for prematurely disregarding a concrete symbolic substitution without evaluating all test cases whenever the deviation accumulated by some of them is excessive. So, since the table only displays symbolic substitutions improving the deviation achieved by previous candidates, observe, for instance, that all intermediate candidates between Θ_{923} and Θ_{1043} are omitted because their accumulated deviations (without the mandatory need of evaluating all test cases) are bigger than the one of Θ_{923}, and this last one is also bigger than the deviation associated to Θ_{1043}.

Until now, we have considered unfolding and tuning operations as independent techniques with clearly different objectives: the first one to improve the efficiency of programs, and the second one to calibrate their rules. However, the unfolding transformation can also be used to improve the tuning process,

Test cases

```
1 0.60 -> modern(hydropolis).
2 0.45 -> good_hotel(ritz).
3 0.38 -> good_hotel(hydropolis).
```

FASILL – Symbolic method | Tune program

FASILL – Symbolic method
FASILL – Basic method
SMT – Boolean lattice
SMT – Real lattice
SMT – Unit lattice

(a) Screenshot of the online tool for starting a tuning process.

Symbolic substitution

```
1 best symbolic substitution: {#@s4/@aver,#&s2/&godel,#s0/0.9,#s1/0.4,#s3/0.6}
2 deviation: 0.0
3 execution time: 3699 milliseconds
```

(b) Screenshot of the online tool after ending a tuning process.

Fig. 5. The FASILL online tool tuning a symbolic program.

in order to approximate the run-time of the basic method to the symbolic one.

Coming back again to our tuning example, when tuning both the original and unfolded programs of examples 2 and 5, we obviously obtain the same symbolic substitution, apart from the fact that the unfolded program will always run faster that the original one. Moreover, the tuning-time of the basic method is considerably reduced when applied to the transformed program. This is due to the fact that, instead of using several successful/failure step w.r.t. the original program, after unfolding it, the basic method only needs to apply just one $\overset{SS/FS}{\leadsto}$ step to get the fca of each test case after applying each symbolic substitution to the entire program.

Anyway, the basic method is still slightly slower than the symbolic one, even with the unfolded version of the program. That is because the basic method have to perform a $\overset{SS/FS}{\leadsto}$ step for each symbolic substitution, while the symbolic method calculates the sfca's (also in one $\overset{SS/FS}{\leadsto}$ step) only once and stores them. So, if we consider n symbolic substitutions and k test cases for the tuning process, the basic method (with the unfolded program) should compute $(n-1)*k$ more admissible steps than the symbolic method. All in all, both tuning algorithms preceded by the unfolding pre-process require approximately the same time than the symbolic algorithm in isolation (the basic alone doubles such time, as commented before) in our running example. But in order to confirm this property in a more complex situation, we have prepared the following benchmarks.

Tables II and III summarize the results of an experimental evaluation (each cell refers to the average of 10 executions using a desktop computer equipped with 4,00 GB RAM and i3-2310M CPU @ 2.10 GHz.) of the tuning techniques preceded by several unfolding iterations. Here, the same sFASILL program is tuned with both basic and symbolic algorithms, varying the numbers of unfolding operations and explored symbolic substitutions. To do this, we consider a general rule of the form $p \leftarrow q \wedge \cdots \wedge q \wedge s^{\#}$ (containing 100 instances of atom q and only a symbolic constant $s^{\#}$), along with a fact $q \leftarrow \top$; and we introduce a single test case: $\top \rightarrow p$. Furthermore, in all executions, the only symbolic substitution that produces a deviation of 0 is considered the last one in the search space to ensure that both methods explore exactly the maximum number of symbolic substitutions.

Focusing on the first row in both tables, referred to the tuning time associated to the manipulation of the original program when considering from 10 to 1000 different symbolic

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE I. Search for the best symbolic substitution by the FASILL system, where t_i and ϵ_i denote the truth degree and deviation for the i-th test case. Only symbolic substitutions that improve the error made by previous substitutions in the search process are shown.

Θ	$@^{\#}_{s4}$	$\&^{\#}_{s2}$	$s^{\#}_0$	$s^{\#}_1$	$s^{\#}_3$	t_1	ϵ_1	t_2	ϵ_2	t_3	ϵ_3	ϵ_{total}
Θ_{309}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	0.0	0.0	0.6	0.0	0.45	0.0	0.38	1.4300
Θ_{310}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	0.1	0.0	0.6	0.0	0.45	0.05	0.33	1.3800
Θ_{311}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	0.2	0.0	0.6	0.0	0.45	0.1	0.28	1.3300
Θ_{312}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	0.3	0.0	0.6	0.0	0.45	0.15	0.23	1.2800
Θ_{313}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	0.4	0.0	0.6	0.0	0.45	0.2	0.18	1.2300
Θ_{314}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	0.5	0.0	0.6	0.0	0.45	0.25	0.13	1.1800
Θ_{315}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	0.6	0.0	0.6	0.0	0.45	0.3	0.08	1.1300
Θ_{316}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	0.7	0.0	0.6	0.0	0.45	0.35	0.03	1.0800
Θ_{317}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	0.8	0.0	0.6	0.0	0.45	0.4	0.02	1.0700
Θ_{318}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	0.9	0.1	0.5	0.0	0.45	0.45	0.07	1.0200
Θ_{319}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.2	0.6	1.0	0.2	0.4	0.0	0.45	0.5	0.12	0.9700
Θ_{438}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.3	0.6	0.8	0.1	0.5	0.05	0.4	0.4	0.02	0.9200
Θ_{439}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.3	0.6	0.9	0.2	0.4	0.05	0.4	0.45	0.07	0.8700
Θ_{440}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.3	0.6	1.0	0.3	0.3	0.05	0.4	0.5	0.12	0.8200
Θ_{559}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.4	0.6	0.8	0.2	0.4	0.1	0.35	0.4	0.02	0.7700
Θ_{560}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.4	0.6	0.9	0.3	0.3	0.1	0.35	0.45	0.07	0.7200
Θ_{561}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.4	0.6	1.0	0.4	0.2	0.1	0.35	0.5	0.12	0.6700
Θ_{680}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.5	0.6	0.8	0.3	0.3	0.15	0.3	0.4	0.02	0.6200
Θ_{681}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.5	0.6	0.9	0.4	0.2	0.15	0.3	0.45	0.07	0.5700
Θ_{682}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.5	0.6	1.0	0.5	0.1	0.15	0.3	0.5	0.12	0.5200
Θ_{801}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.6	0.6	0.8	0.4	0.2	0.2	0.25	0.4	0.02	0.4700
Θ_{802}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.6	0.6	0.9	0.5	0.1	0.2	0.25	0.45	0.07	0.4200
Θ_{803}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.6	0.6	1.0	0.6	0.0	0.2	0.25	0.5	0.12	0.3700
Θ_{922}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.7	0.6	0.8	0.5	0.1	0.25	0.2	0.4	0.02	0.3200
Θ_{923}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.7	0.6	0.9	0.6	0.0	0.25	0.2	0.45	0.07	0.2700
Θ_{1043}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.8	0.6	0.8	0.6	0.0	0.3	0.15	0.4	0.02	0.1700
Θ_{1163}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.9	0.6	0.7	0.6	0.0	0.35	0.1	0.35	0.03	0.1300
Θ_{1196}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.9	0.9	0.7	0.6	0.0	0.35	0.1	0.355	0.025	0.1250
Θ_{1207}	$@_{\texttt{aver}}$	$\&_{\texttt{luka}}$	0.9	1.0	0.7	0.6	0.0	0.35	0.1	0.37	0.01	0.1100
Θ_{2603}	$@_{\texttt{aver}}$	$\&_{\texttt{prod}}$	1.0	0.5	0.6	0.6	0.0	0.405	0.045	0.3153	0.0647	0.1097
Θ_{2614}	$@_{\texttt{aver}}$	$\&_{\texttt{prod}}$	1.0	0.6	0.6	0.6	0.0	0.405	0.045	0.322	0.058	0.1029
Θ_{2625}	$@_{\texttt{aver}}$	$\&_{\texttt{prod}}$	1.0	0.7	0.6	0.6	0.0	0.405	0.045	0.33	0.05	0.0950
Θ_{2636}	$@_{\texttt{aver}}$	$\&_{\texttt{prod}}$	1.0	0.8	0.6	0.6	0.0	0.405	0.045	0.3392	0.0408	0.0858
Θ_{2647}	$@_{\texttt{aver}}$	$\&_{\texttt{prod}}$	1.0	0.9	0.6	0.6	0.0	0.405	0.045	0.3496	0.0304	0.0754
Θ_{2658}	$@_{\texttt{aver}}$	$\&_{\texttt{prod}}$	1.0	1.0	0.6	0.6	0.0	0.405	0.045	0.3612	0.0188	0.0638
Θ_{3681}	$@_{\texttt{aver}}$	$\&_{\texttt{godel}}$	0.8	0.4	0.6	0.6	0.0	0.4	0.05	0.38	0.0	0.0500
Θ_{3791}	$@_{\texttt{aver}}$	$\&_{\texttt{godel}}$	0.9	0.3	0.6	0.6	0.0	0.45	0.0	0.345	0.035	0.0350
Θ_{3802}	$@_{\texttt{aver}}$	$\&_{\texttt{godel}}$	0.9	0.4	0.6	0.6	0.0	0.45	0.0	0.38	0.0	0.0000

TABLE II. Average runtime (in milliseconds) of the basic tuning method after 10 executions, based on the number of unfolding steps (k) and the number of considered symbolic substitutions.

k	Time (ms)					
	10_Θ	50_Θ	100_Θ	250_Θ	500_Θ	1000_Θ
0	168	781	1359	3515	6946	13772
1	159	693	1372	3453	6768	13578
5	128	672	1356	3681	6593	13059
10	140	634	1265	3334	6368	13006
25	125	562	1140	2825	5640	11425
50	90	428	853	2115	4515	8456
75	75	340	518	1406	2531	5028
100	15	53	106	278	540	1065

TABLE III. Average runtime (in milliseconds) of symbolic tuning method after 10 executions, based on the number of unfolding steps (k) and the number of considered symbolic substitutions.

k	Time (ms)					
	10_Θ	50_Θ	100_Θ	250_Θ	500_Θ	1000_Θ
0	56	65	115	268	578	1240
1	50	65	115	268	540	1043
5	22	59	115	268	518	1037
10	25	62	97	256	534	1043
25	21	62	115	309	525	1075
50	18	59	112	265	522	1159
75	28	56	109	253	512	1034
100	15	53	106	243	515	1025

substitutions, the basic algorithm ranges from 168 to 13772 milliseconds in Table II, which largely contrasts with the rank between 56 and 1240 exhibited by the symbolic method in Table III. Also, the last column in the basic case presents drastic time reductions when comparing the tuning time of the original program with respect to its different unfolded versions, being this measure dramatically reduced (approximately) thirteen times when tuning the last transformed program obtained after unfolding one hundred times the initial one. In contrast to this, note that the tuning time of the symbolic algorithm is not especially affected by unfolding transformations, varying from 1240 to 1025 milliseconds in the last column of Table III and being this last value almost the same (1065) than the one in the cell at the right-bottom corner of Table II. This effect is reinforced in Figure 6, which evidences in a more graphical way that the application of unfolding before tuning a program is not relevant for the symbolic strategy, but the efficiency of the basic method notably increases, which confirms our expectations.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

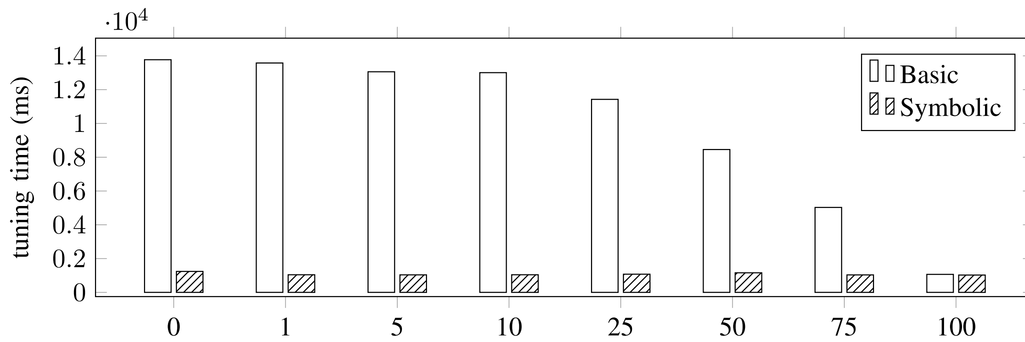

Fig. 6. Comparison of tuning algorithms based on the number of unfolding steps performed over the program.

VI. Conclusion and Future Work

Coping with symbolic similarity relations, the symbolic extension of the FASILL language we introduced in [22], is the basis of the effective unfolding technique for sFASILL programs we have very recently presented in [1]. The new transformation surpasses both the similarity-based (but non-symbolic) unfolding of [30][31], as well as the symbolic (but not dealing yet with similarities) operation of [32][33], thus permitting the optimization of sFASILL programs in an unified, similarity-based symbolic framework.

The present work also considers tuning techniques for extending [1], having in mind that sFASILL programs can be seen as templates to be fulfilled with concrete fuzzy values/operators for producing tuned FASILL programs which satisfy users wishes. In this paper, we have firstly collected from [38][21] two semi-automatic tuning engines (basic and symbolic) for customizing symbolic programs and then, we have combined them with the symbolic unfolding technique of [1]. We have implemented these techniques in a freely available tool [34], which has been used for developing interesting benchmarks and analyzing the good performance of the mixed techniques. Our experiments reveal that, when manipulating sFASILL programs, the sequence "unfolding plus tuning" is preferable than the one of "tuning plus unfolding". To summarize, the benefits of applying symbolic unfolding on a sFASILL program before tuning it, are: 1) the basic method applied on unfolded programs exhibits an efficiency close to the symbolic algorithm (with or without an unfolding pre-process) and 2) both the execution and tuning times are significantly improved for the resulting, unfolded plus tuned, FASILL program.

Some pending related tasks for the near future consist in exploring the synergies between our approach and machine learning strategies (including neural networks, as we managed in [27]), fuzzy SMT/SAT solvers (see our previous experiences in [38]), etc. As ongoing work, we are nowadays developing the formal proofs that reinforce the correctness of symbolic unfolding under certain safe applicability conditions.

Acknowledgment

This work has been partially supported by the EU (FEDER), the State Research Agency (AEI) of the Spanish Ministry of Science and Innovation under grant PID2019-104735RB-C42 (SAFER).

References

[1] G. Moreno and J. A. Riaza, "Symbolic unfolding of similarity-based fuzzy logic programs," in *The 2024 IARIA Annual Congress on Frontiers in Science, Technology, Services, and Applications*. IARIA Congress 2024, ThinkMind Digital Library, 2019, pp. 121–125. [Online]. Available: https://personales.upv.es/thinkmind/dl/conferences/iariacongress/iaria_congress_2024/iaria_congress_2024_2_160_50079.pdf

[2] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, pp. 338–353, 1965. [Online]. Available: http://www-bisc.cs.berkeley.edu/Zadeh-1965.pdf

[3] J. W. Lloyd, *Foundations of Logic Programming*. Springer, 1987.

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

[5] F. Formato, G. Gerla, and M. I. Sessa, "Similarity-based unification," *Fundamenta Informaticae*, vol. 41, no. 4, pp. 393–414, 2000.

[6] M. I. Sessa, "Approximate reasoning by similarity-based SLD resolution," *Theoretical Computer Science*, vol. 275, no. 1-2, pp. 389–426, 2002.

[7] F. Arcelli, "Likelog for flexible query answering," *Soft Computing*, vol. 7, no. 2, pp. 107–114, 2002.

[8] P. Julián-Iranzo and C. Rubio, "A declarative semantics for bousi∼prolog," in *Proc. of 11th Int. ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'09, Coimbra, Portugal*. ACM, 2009, pp. 149–160.

[9] C. Rubio-Manzano and P. Julián-Iranzo, "A fuzzy linguistic prolog and its applications," *Journal of Intelligent and Fuzzy Systems*, vol. 26, no. 3, pp. 1503–1516, 2014.

[10] R. Lee, "Fuzzy Logic and the Resolution Principle," *Journal of the ACM*, vol. 19, no. 1, pp. 119–129, 1972.

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

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

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

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

[15] M. Rodríguez-Artalejo and C. Romero-Díaz, "Quantitative logic programming revisited," in *Proc. of 9th Functional and Logic Programming Symposium, FLOPS'08*, J. Garrigue and M. Hermenegildo, Eds. LNCS, 4989, Springer Verlag, 2008, pp. 272–288.

[16] S. Guadarrama, S. Muñoz, and C. Vaucheret, "Fuzzy Prolog: A new approach using soft constraints propagation," *Fuzzy Sets and Systems*, vol. 144, no. 1, pp. 127–150, 2004.

[17] S. Muñoz, V. P. Ceruelo, and H. Strass, "Rfuzzy: Syntax, semantics and implementation details of a simple and expressive fuzzy tool over prolog," *Information Sciences*, vol. 181, no. 10, pp. 1951–1970, 2011.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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

[19] P. Julián, G. Moreno, and J. Penabad, "Thresholded semantic framework for a fully integrated fuzzy logic language," *J. Log. Algebr. Meth. Program.*, vol. 93, pp. 42–67, 2017. [Online]. Available: https://doi.org/10.1016/j.jlamp.2017.08.002

[20] G. Moreno, J. Penabad, J. A. Riaza, and G. Vidal, "Symbolic execution and thresholding for efficiently tuning fuzzy logic programs," in *Logic-Based Program Synthesis and Transformation, Proc. of the 26th International Symposium LOPSTR 2016*. LNCS, 10184, Springer, 2016, pp. 131–147. [Online]. Available: https://doi.org/10.1007/978-3-319-63139-4_8

[21] G. Moreno and J. A. Riaza, "A safe and effective tuning technique for similarity-based fuzzy logic programs," in *Advances in Computational Intelligence - 16th Int. Work-Conference on Artificial Neural Networks, IWANN 2021*, vol. LNCS 12861. Springer, 2021, pp. 190–201. [Online]. Available: https://doi.org/10.1007/978-3-030-85030-2_16

[22] G. Moreno and J. A. Riaza, "Symbolic similarity relations for tuning fully integrated fuzzy logic programs," in *Rules and Reasoning - Proc. of 4th Int. Joint Conference, RuleML+RR 2020*, vol. LNCS 12173. Springer, 2020, pp. 150–158. [Online]. Available: https://doi.org/10.1007/978-3-030-57977-7_11

[23] L. D. Raedt and A. Kimmig, "Probabilistic (logic) programming concepts," *Mach. Learn.*, vol. 100, no. 1, pp. 5–47, 2015. [Online]. Available: https://doi.org/10.1007/s10994-015-5494-z

[24] F. Riguzzi and T. Swift, "The PITA system: Tabling and answer subsumption for reasoning under uncertainty," *Theory Pract. Log. Program.*, vol. 11, no. 4-5, pp. 433–449, 2011. [Online]. Available: https://doi.org/10.1017/S147106841100010X

[25] K. F. Sagonas, T. Swift, and D. S. Warren, "XSB as an Efficient Deductive Database Engine," in *Proc. of ACM SIGMOD International Conference on Management of Data*. ACM Press, 1994, pp. 442–453.

[26] J. Almendros-Jiménez, A. Becerra-Terón, G. Moreno, and J. A. Riaza, "Tuning fuzzy sparql queries," *International Journal of Approximate Reasoning*, vol. 170, p. 109209, 2024. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85191942148&doi=10.1016%2fj.ijar.2024.109209&partnerID=40&md5=42d6833e308f5a0eba0e6240657ae987

[27] G. Moreno, J. Pérez, and J. A. Riaza, "Fuzzy logic programming for tuning neural networks," in *Rules and Reasoning - Proc. of Third International Joint Conference, RuleML+RR 2019*, vol. LNCS 11784. Springer, 2019, pp. 190–197.

[28] A. Pettorossi and M. Proietti, "Rules and Strategies for Transforming Functional and Logic Programs," *ACM Computing Surveys*, vol. 28, no. 2, pp. 360–414, 1996.

[29] H. Tamaki and T. Sato, "Unfold/Fold Transformations of Logic Programs," in *Proc. of Second Int'l Conf. on Logic Programming*, S. Tärnlund, Ed., 1984, pp. 127–139.

[30] P. Julián-Iranzo, G. Moreno, and J. A. Riaza, "Seeking a safe and efficient similarity-based unfolding rule," *Int. J. Approx. Reason.*, vol. 163, p. 109038, 2023. [Online]. Available: https://doi.org/10.1016/j.ijar.2023.109038

[31] P. Julián-Iranzo, G. Moreno, and J. A. Riaza, "Some properties of substitutions in the framework of similarity relations," *Fuzzy Sets Syst.*, vol. 465, p. 108510, 2023. [Online]. Available: https://doi.org/10.1016/j.fss.2023.03.013

[32] G. Moreno, J. Penabad, and J. A. Riaza, "Symbolic unfolding of multi-adjoint logic programs," in *Trends in Mathematics and Computational Intelligence*. Studies in Computational Intelligence, Springer International Publishing, (extended version of a previous paper presented at ESCIM'17), 2019, pp. 43–51. [Online]. Available: https://doi.org/10.1007/978-3-030-00485-9_5

[33] G. Moreno and J. A. Riaza, "An online tool for unfolding symbolic fuzzy logic programs," in *Advances in Computational Intelligence - Proc. of 15th International Work-Conference on Artificial Neural Networks (Part II), IWANN 2019*, vol. LNCS 11507. Springer, 2019, pp. 475–487. [Online]. Available: https://doi.org/10.1007/978-3-030-20518-8_40

[34] G. Moreno and J. A. Riaza, "Fasill: Sandbox," in *https://dectau.uclm.es/fasill/sandbox*, Accessed: 2024-11-30.

[35] P. Julián-Iranzo, "A procedure for the construction of a similarity relation," in *Proc. of the 12th International Conference on Information Processing and Management of Uncertainty in Knoledge-based Systems (IPMU 2008), June 22-27, Torremolinos (Málaga), Spain*. U. Málaga (ISBN 978-84-612-3061-7), 2008, pp. 489–496.

[36] A. Kandel and L. Yelowitz, "Fuzzy chains," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. SMC-4, no. 5, pp. 472–475, 1974.

[37] H. Naessens, H. D. Meyer, and B. D. Baets, "Algorithms for the computation of t-transitive closures," *IEEE Trans. Fuzzy Systems*, vol. 10, no. 4, pp. 541–551, 2002.

[38] G. Moreno and J. A. Riaza, "Using SAT/SMT Solvers for Efficiently Tuning Fuzzy Logic Programs," in *2020 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2020, Glasgow, UK*. IEEE, 2020, pp. 1–8.

[39] P. Julián-Iranzo, J. Medina, G. Moreno, and M. Ojeda, "Thresholded tabulation in a fuzzy logic setting," *ENTCS*, vol. 248, pp. 115–130, 2009.

[40] P. Julián-Iranzo, J. Medina, G. Moreno, and M. Ojeda, "Efficient thresholded tabulation for fuzzy query answering," *Studies in Fuzziness and Soft Computing (Foundations of Reasoning under Uncertainty)*, vol. 249, pp. 125–141, 2010.

[41] P. Julián-Iranzo, G. Moreno, and J. Penabad, "Efficient reductants calculi using partial evaluation techniques with thresholding," *Electronic Notes in Theoretical Computer Science, Elsevier Science*, vol. 188, pp. 77–90, 2007.

2024, © Copyright by authors, Published under agreement with IARIA - www.iaria.org