An Unfolding-based Preprocess for Reinforcing Thresholds in Fuzzy Tabulation

P. Julián[1], J. Medina[2], P. J. Morcillo[3], G. Moreno[3], and M. Ojeda-Aciego[4]

[1] Dept. of Information Technologies and Systems. University of Castilla-La Mancha
Email: pascual.julian@uclm.es
[2] Department of Mathematics. University of Cadiz.
Email: jesus.medina@uca.es
[3] Department of Computing Systems. University of Castilla-La Mancha.
Email: gines.moreno@uclm.es,pmorcillo@dsi.uclm.es
[4] Department of Applied Mathematics. University of Málaga.
Email: aciego@uma.es

Abstract. We have recently proposed a technique for generating thresholds (filters) useful for avoiding useless computations when executing fuzzy logic programs in a tabulated way. The method was conceived as a static preprocess practicable on program rules before being executed with our *fuzzy thresholded tabulation* principle, thus increasing the opportunities of prematurely disregarding those computation steps which are redundant (tabulation) or directly lead to non-significant solutions (thresholding). In this paper we reinforce the power of such static preprocess—which obviously does not require the consumption of extra computational resources at execution time—by re-formulating it in terms of the fuzzy unfolding technique initially designed in our group for transforming and optimizing fuzzy logic programs.

Key words: Fuzzy Logic Programming, Tabulation, Thresholding, Unfolding

1 Introduction

The fields of logic programming and fuzzy logic have shown its complementarity for more than two decades [2, 4, 9, 11]. In this work we continue our efforts to provide a refined and improved fuzzy query answering procedure for multi-adjoint logic programming MALP [10, 11]. Roughly speaking, the general idea is that, when trying to perform a computation step by using a given program rule $\mathcal{R}$, we firstly analyze if such step might contribute to reach further significant solutions (not yet tabulated, saved or stored). After recalling the static approach introduced in [5], the main contribution starts in Section 3, where we focus on a new refinement in order to improve the existing static preprocessing step.

The MALP approach[5] considers a language, $\mathcal{L}$, containing propositional variables, constants, and a set of logical connectives. In our fuzzy setting, we use

[5] Visit http://dectau.uclm.es/floper/ and http://dectau.uclm.es/fuzzyXPath/ for downloading related tools [1, 12].

implication connectives ($\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m$) together with a number of aggregators. They will be used to combine/propagate truth values through the rules. The general definition of aggregation operators subsumes conjunctive operators (denoted by $\&_1, \&_2, \ldots, \&_k$), disjunctive operators ($\vee_1, \vee_2, \ldots, \vee_l$), and average and hybrid operators (usually denoted by $@_1, @_2, \ldots, @_n$). Aggregators are useful to describe/specify user preferences: when interpreted as a truth function they may be considered, for instance, as an arithmetic mean or a weighted sum. The language $\mathcal{L}$ will be interpreted on a *multi-adjoint lattice*, $\langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n\rangle$, which is a complete lattice equipped with a collection of adjoint pairs $\langle \leftarrow_i, \&_i\rangle$, where each $\&_i$ is a conjunctor[6] intended to provide a *modus ponens*-rule w.r.t. $\leftarrow_i$. In general, the set of truth values L may be the carrier of any complete bounded lattice but, for simplicity, in the examples of this work we shall select L as the set of real numbers in the interval $[0, 1]$.

A *rule* is a formula $A \leftarrow_i \mathcal{B}$, where the *head* A is an propositional symbol and the *body* $\mathcal{B}$ is a formula built from propositional symbols $B_1, \ldots, B_n$ ($n \geq 0$), truth values of L and conjunctions, disjunctions and aggregations. Rules with an empty body are called *facts*. A *goal* is a body submitted as a query to the system. Roughly speaking, a MALP program is a set of pairs $\langle R; \alpha\rangle$, where R is a rule and α is a value of L, which might express the confidence which the user of the system has in the truth of the rule R (note that the truth degrees in a given program are expected to be assigned by an expert).

The standard procedural semantics of the multi–adjoint logic language $\mathcal{L}$ is based on the notion of *admissible step* (which can be seen as a fuzzy extension of classical SLD-resolution) whose definition is also crucial for describing the unfolding transformation [7]. Hereafter, $\mathcal{C}[A]$ denotes a formula where A is a subexpression (usually a propositional symbol) which occurs in the (possibly empty) context $\mathcal{C}[]$, whereas $\mathcal{C}[A/A']$ means the replacement of A by A' in context $\mathcal{C}[]$. In the following definition, we always consider that A is the selected propositional symbol in goal $\mathcal{Q}$.

Definition 1 (Admissible Steps). *Let $\mathcal{Q}$ be a goal, which is considered as a state, and let $\mathcal{G}$ be the set of goals. Given a program $\mathbb{P}$, an* admissible computation *is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathcal{G} \times \mathcal{G})$ is the smallest relation satisfying the following* admissible rules*:*

1. $\mathcal{Q}[A] \rightarrow_{AS} \mathcal{Q}[A/v\&_i\mathcal{B}]$ *if there is a rule $\langle A \leftarrow_i \mathcal{B}; v\rangle$ in $\mathbb{P}$ and $\mathcal{B}$ is not empty.*
2. $\mathcal{Q}[A] \rightarrow_{AS} \mathcal{Q}[A/v]$ *if there is a fact $\langle A \leftarrow_i; v\rangle$ in $\mathbb{P}$.*

It is obvious that if we exploit all propositional symbols of a goal, by applying admissible steps as much as needed, then the goal becomes a formula (with no propositional symbols) which can then be directly interpreted in the multi–adjoint lattice L and thus obtaining the desired *fuzzy computed answer* (or f.c.a., in brief) for that goal.

Although the procedural principle we have just seen suffices for executing MALP programs, there exists a much more efficient mechanism for solving

[6] An increasing operator satisfying boundary conditions with the top element.

queries as occurs with the thresholded tabulation procedure proposed in [5, 6] that we are going to resume in the following section.

2 Fuzzy Thresholded Tabulation with Static Preprocess

Tabulation arises as a technique to solve two important problems in deductive databases and logic programming: termination and efficiency. The datatype we will use for the description of the proposed method is that of a *forest*, that is, a finite set of trees. Each one of these trees has a root labeled with a propositional symbol together with a truth-value from the underlying lattice (called the *current value* for the *tabulated* symbol); the rest of the nodes of each of these trees are labeled with an "extended" formula in which some of the propositional symbols have been substituted by its corresponding value.

The following descriptions are considered in order to prune some useless branches or, more exactly, for avoiding the use (at execution time) of those program rules whose weights do not surpass a given "threshold" value.

- Let $\mathcal{R} = \langle A \leftarrow_i \mathcal{B}; \vartheta \rangle$ be a program rule.
- Let $\mathcal{B}'$ be an expression with no atoms, obtained from body $\mathcal{B}$ by replacing each occurrence of a propositional symbol by $\top$.
- Let $v \in L$ be the result of interpreting $\mathcal{B}'$ under a given lattice.
- Then, $Up\_\ body(\mathcal{R}) = v$.

Apart from the truth degree ϑ of a program rule $\mathcal{R} = \langle A \leftarrow_i \mathcal{B}; \vartheta \rangle$ and the maximum truth degree of its body $Up\_\ body(\mathcal{R})$, in the multi-adjoint logic setting, we can consider a third kind of filter for reinforcing thresholding. The idea is to combine the two previous measures by means of the adjoint conjunction $\&_i$ of the implication $\leftarrow_i$ in rule $\mathcal{R}$. Now, we define the *maximum truth degree of a program rule*, symbolized by function $Up\_\ rule$, as: $Up\_\ rule(\mathcal{R}) = \vartheta \&_i (Up\_\ body(\mathcal{R}))$.

As shown in [5], such filters can be safely compiled on program rules after applying an easy static preprocess whose benefits will be largely redeemed on further executions of the program. So, for any MALP program $\mathbb{P}$, we can obtain its extended version $\mathbb{P}+$ (for being used during the "query answering" process) by adding to its program rules their *proper threshold* $\mathrm{Up\_rule}(\mathcal{R})$ as follows:

$$\mathbb{P}+ = \{\langle A \leftarrow_i \mathcal{B}; \vartheta; Up\_\ rule(\mathcal{R}) \rangle \ \mid \ \mathcal{R} = \langle A \leftarrow_i \mathcal{B}; \vartheta \rangle \in \mathbb{P}\}.$$

Example 1. Consider the following extended program $\mathbb{P}+$ (with mutual recursion) recasted from [5], where the filter associated with each program rule coincides with its weight.

$$\begin{array}{l}
\mathcal{R}_1 : \langle\ p \ \leftarrow_{\mathtt{P}}\ q\ ;\ 0.6\ ;\ 0.6\ \rangle \\
\mathcal{R}_2 : \langle\ p \ \leftarrow_{\mathtt{P}}\ r\ ;\ 0.5\ ;\ 0.5\ \rangle \\
\mathcal{R}_3 : \langle\ q \ \leftarrow\ \ ;\ 0.9\ ;\ 0.9\ \rangle \\
\mathcal{R}_4 : \langle\ r \ \leftarrow\ \ ;\ 0.8\ ;\ 0.8\ \rangle \\
\mathcal{R}_5 : \langle\ r \ \leftarrow_{\mathtt{L}}\ p\ ;\ 0.9\ ;\ 0.9\ \rangle
\end{array}$$

A more interesting case could be represented by a new extended program $\mathbb{P}'+$ similar to the previous one but simply replacing its second rule by: $\mathcal{R}_2' : \langle p \ \leftarrow_{\mathrm{P}} \ (r \&_{\mathrm{P}} \ 0.9) \ ; \ 0.55; \ 0.495 \ \rangle$ (note the presence of a truth degree in its body, which justifies why in this case the $Up\_\,rule(\mathcal{R}_2')$ value is fortunately lower than the weight of the rule).

Operations for tabulation with thresholding using extended programs

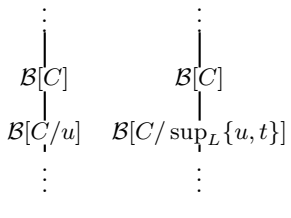
The tabulation procedure requires four basic operations: Root Expansion, New Subgoal/Tree, Value Update, and Answer Return. In the first case we take profit of the *filters* for thresholding compiled on extended programs, whose further use will drastically diminish the number of nodes in trees (note that by avoiding the generation of a single node, the method implicitly avoids the generation of all its possible descendants as well). New Subgoal is applied whenever a propositional variable is found without a corresponding tree in the forest. Value update is used to propagate the truth-values of answers to the root of the corresponding tree. Finally, answer return substitutes a propositional variable by the current truth-value in the corresponding tree. Let us formally describe such operations:

Rule 1: Root Expansion. Given a tree with root $A\colon r$ in the forest, if there is a program rule $\mathcal{R} = \langle A \leftarrow_i \mathcal{B}; \vartheta; \, Up\_\,rule(\mathcal{R}) \rangle \in \mathbb{P}+$ not consumed before and verifying $Up\_\,rule(\mathcal{R}) \not\leq r$, append the new child $\vartheta \&_i \mathcal{B}$ to the root of the tree.

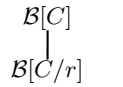
Rule 2: New Subgoal/Tree. Select a non-tabulated propositional symbol C occurring in a leaf of some tree (this means that there is no tree in the forest with the root node labeled with C), then create a new tree with a single node, the root $C\colon \bot$, and append it to the forest.

Rule 3: Value Update. If a tree, rooted at $C\colon r$, has a leaf $\mathcal{B}$ with no propositional symbols, and $\mathcal{B} \rightarrow_{IS}{}^{*} s$, where $s \in L$, then update the current value of the propositional symbol C by the value of $\sup_L\{r, s\}$.

Furthermore, once the tabulated truth-value of the tree rooted by C has been modified, for all the occurrences of C in a non-leaf node $\mathcal{B}[C]$ such as the one in the left of the figure below then, update the whole branch substituting the constant u by $\sup_L\{u, t\}$ (where t is the last tabulated truth-value for C, i.e. $\sup_L\{r, s\}$) as in the right of the figure.



Rule 4: Answer Return. Select in any leaf a propositional symbol C which is tabulated, and assume that its current value is r; then add a new successor node as the figure shows:



Example 2. Let us see now how our method proceeds when solving query $?p$ w.r.t. the extended program $\mathbb{P}+$ of Example 1.

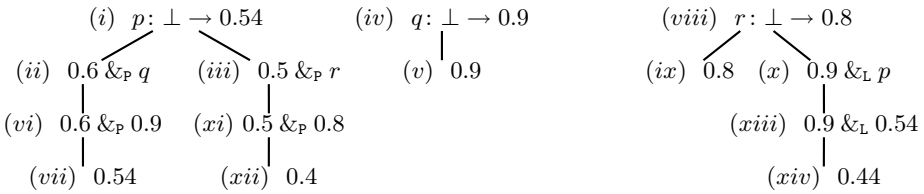


Fig. 1. Example forest for query $?p$ w.r.t. the extended program $\mathbb{P}+$.

The initial tree consisting of nodes $(i), (ii), (iii)$ is generated, see Figure 1. Then *New Subgoal* is applied on q, a new tree is generated with nodes (iv) and (v), and its current value is directly updated to 0.9. By using this value, *Answer Return* extends the initial tree with node (vi). Now *Value Update* generates node (vii) and updates the current value of p to 0.54. Then, *New Subgoal* is applied on r, and a new tree is generated with nodes $(viii), (ix)$ and (x). *Value Update* increases the current value to 0.8. By using this value, *Answer Return* extends the initial tree with node (xi). Now *Value Update* generates node (xii). The current value is not updated since its value is greater than the newly computed one. Finally, *Answer Return* can be applied again on propositional symbol p in node (x), generating node $(xiii)$. A further application of *Value Update* generates node (xiv) and the forest is terminated, as no rule performs any update.

In order to illustrate the advantages of our improved method, consider now our second extended program $\mathbb{P}'+$ of Example 1, where remember that we have replaced the second program rule by: $\mathcal{R}_2' : \langle p \leftarrow_{\mathrm{P}} (r \&_{\mathrm{P}} 0.9) \ ; \ 0.55; \ 0.495 \rangle$. It is important to note now that even when the truth degree of the rule is 0.55, its threshold decreases to $Up\_rule(\mathcal{R}_2') = 0.55 * 0.9 = 0.495 < 0.54$, which avoids extra expansions of the tree as Figure 2 shows.

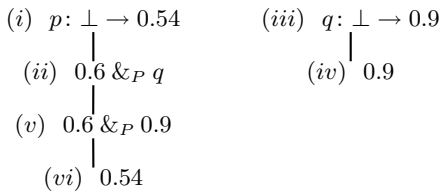


Fig. 2. Example threshold forest for query $?p$ w.r.t. the extended program $\mathbb{P}'+$

3 Improving the Static Preprocess with Fuzzy Unfolding

Program transformation is an optimization technique for computer programs that starting with an initial program $\mathbb{P}_0$ derives a sequence $\mathbb{P}_1, \ldots, \mathbb{P}_n$ of trans-

formed programs by applying *elementary transformation rules* (fold/unfold) which improve the original program. The basic idea is to divide the program development activity, starting with a (possibly naive) problem specification written in a programming language, into a sequence of small transformation steps. *Unfolding* [3, 13] is a well-known, widely used, semantics-preserving program transformation rule. In essence, it is usually based on the application of operational steps on the body of program rules. The unfolding transformation is able to improve programs, generating more efficient code. Unfolding is the basis for developing sophisticated and powerful programming tools, such as fold/unfold transformation systems or partial evaluators, etc.

On the other hand, as revealed in the examples of the previous sections, the presence of truth degrees on the body of fuzzy program rules is always desirable for optimizing the power of thresholding at tabulation time. In [7], we show that it is possible to transform a program rule into a semantically equivalent set of rules with the intended shape. The following definition is recalled from [7], but we have slightly simplified it in the sense that here we deal with propositional (instead of first order) MALP programs:

Definition 2 (Fuzzy Unfolding). *Let $\mathbb{P}$ be a program and let $\mathcal{R} : \langle A \leftarrow_i \mathcal{B}; \alpha\rangle \in \mathbb{P}$ be a program rule which is not a fact. Then, the fuzzy unfolding of program $\mathbb{P}$ with respect to rule $\mathcal{R}$ is the new program:* $\mathbb{P}' = (\mathbb{P} - \{\mathcal{R}\}) \cup \mathcal{U}$ *where* $\mathcal{U} = \{\langle A \leftarrow_i \mathcal{B}'; \alpha\rangle \mid \mathcal{B} \rightarrow_{AS} \mathcal{B}'\}$.

It is important to note in the previous definition that the set of *unfolded rules* $\mathcal{U}$ is not a singleton in general. For instance, assume a program $\mathbb{P}$ with the following two rules $\mathcal{R}_1 : \langle p \leftarrow; 0.5\rangle$ and $\mathcal{R}_2 : \langle p \leftarrow_{\mathrm{P}} p; 1\rangle$. Note that the first rule does not admit unfolding since there are no propositional symbols on its body (it is a "fact"), but $\mathcal{R}_2$ can be unfolded by using both rules for obtaining $\mathcal{U} = \{\mathcal{R}_{2-1} : \langle p \leftarrow_{\mathrm{P}} 0.5 ; 1\rangle,\ \mathcal{R}_{2-2} : \langle p \leftarrow_{\mathrm{P}} 1 \,\&_{\mathrm{P}}\, p ; 1\rangle\}$. [7] Now the unfolded program $\mathbb{P}'$ is composed by the following set of rules defining p (note that we have replaced the original rule $\mathcal{R}_2$ by the transformed rules $\mathcal{R}_{2-1}$ and $\mathcal{R}_{2-2}$):

$$
\begin{array}{lllll}
\mathcal{R}_1 : & \langle\, p & \leftarrow & & ; 0.5 \,\rangle \\
\mathcal{R}_{2-1} : & \langle\, p & \leftarrow_{\mathrm{P}} & 0.5 & ; 1 \,\rangle \\
\mathcal{R}_{2-2} : & \langle\, p & \leftarrow_{\mathrm{P}} & 1 \,\&_{\mathrm{P}}\, p & ; 1 \,\rangle
\end{array}
$$

Assume now that we add more rules to the previous program $\mathbb{P}$ (note that the tabulation procedure would never end if the program is infinite) as follows:

$$
\begin{array}{llllll}
\mathcal{R}_1 : & \langle & p & \leftarrow & & ; 0.5 \,\rangle \\
\mathcal{R}_2 : & \langle & p & \leftarrow_{\mathrm{P}} & p & ; 1 \,\rangle \\
\mathcal{R}_3 : & \langle & p & \leftarrow_{\mathrm{P}} & q_1 & ; 0.7 \,\rangle \\
\mathcal{R}_4 : & \langle & q_1 & \leftarrow_{\mathrm{P}} & q_2 & ; 0.7 \,\rangle \\
\mathcal{R}_5 : & \langle & q_2 & \leftarrow_{\mathrm{P}} & q_3 & ; 0.7 \,\rangle \\
\mathcal{R}_6 : & \langle & q_3 & \leftarrow_{\mathrm{P}} & q_4 & ; 0.7 \,\rangle \\
 & & \vdots & & &
\end{array}
$$

[7] Note that in this step the body of $\mathcal{R}_{2-2}$ is not computed, although the interpretation of $1 \,\&_{\mathrm{P}}\, p$ is clearly p, since this part is made in the tabulation procedure.

Now, by unfolding rule $\mathcal{R}_2$ we obtain three transformed rules, the two ones seen before (i.e., $\mathcal{R}_{2-1}$ and $\mathcal{R}_{2-2}$) as well as $\mathcal{R}_{2-3} : \langle p \quad \leftarrow_{\mathtt{P}} \quad 0.7 \ \&_{\mathtt{P}} \ q_1 \ ; \ 1 \rangle$. Two more examples: the unfolding of this last rule (using $\mathcal{R}_4$) replaces $\mathcal{R}_{2-3}$ by $\mathcal{R}_{2-3-4} : \langle p \quad \leftarrow_{\mathtt{P}} \quad 0.7 \ \&_{\mathtt{P}} \ 0.7 \ \&_{\mathtt{P}} \ q_2 \ ; \ 1 \rangle$, whereas the unfolding of $\mathcal{R}_3$ (using again $\mathcal{R}_4$) replaces such rule by $\mathcal{R}_{3-4} : \langle p \quad \leftarrow_{\mathtt{P}} \quad 0.7 \ \&_{\mathtt{P}} \ q_2 \ ; \ 0.7 \rangle$.

On the other hand, let us remember that the static preprocess described in [5] and resumed at the beginning of the previous section, simply consists in generating extended programs where each program rule $\mathcal{R}$ is annotated with its proper *Up_ rule*($\mathcal{R}$) value in order to increase the opportunities of bounding branches when applying thresholded tabulation. Now, our new proposal reinforces the static preprocess (since the new annotated *Up_ rule*($\mathcal{R}$) values tends to be lower and thus, more powerful) by applying several unfolding steps (at least one) on program rules before generating the extended program $\mathbb{P}+$. The question now is when to stop the application of unfolding steps.

To answer this question, let us first introduce a new concept. Assume that, for a given propositional symbol p, we define the truth degree $\tau_p \in L$ as the infimum one among the weights of the rules with head p in the original program $\mathbb{P}$. For instance, in our previous example we have that $\tau_p = \inf\{0.5, 1, 0.7\} = 0.5$, $\tau_{q_1} = 0.7$, $\tau_{q_2} = 0.7$ and so on. Then, after applying the first unfolding step on each program rule $\mathcal{R} \in \mathbb{P}$ we obtain a new set of transformed rules, say $\mathcal{R}'_i$, $1 \leq i \leq n$, such that no more unfolding steps are applied on each rule $\mathcal{R}'_i$ if obviously it has no propositional symbols on its body (as occurs with $\mathcal{R}_{2-1}$) or one of the following conditions hold:

- *Up_ rule*($\mathcal{R}'_i$) $\leq$ *Up_ rule*($\mathcal{R}$). This situation is represented by rule $\mathcal{R}_{2-2}$ in our example (since *Up_ rule*($\mathcal{R}_{2-2}$) = *Up_ rule*($\mathcal{R}_2$) = 1), and its is useful for preventing an infinite unfolding loop on rules whose *Up_ rule* values cannot be improved by unfolding.
- *Up_ rule*($\mathcal{R}'_i$) $\leq \tau_p$. This case is really the more interesting one in our technique, as revealed in rules $\mathcal{R}_{2-3-4}$ and $\mathcal{R}_{3-4}$ since:
  - *Up_ rule*($\mathcal{R}_{2-3-4}$) $= 1 * 0.7 * 0.7 * 1 = 0.49 < 0.5 = \tau_p$.
  - *Up_ rule*($\mathcal{R}_{3-4}$) $= 0.7 * 0.7 * 1 = 0.49 < 0.5 = \tau_p$.

We have just seen how the unfolding process finishes in our example, which let us now to generate the following extended program $\mathbb{P}+$:

$$
\begin{array}{lllll}
\mathcal{R}_1 : & \langle \ p & \leftarrow & & ; 0.5 \ \ ; 0.5 \ \rangle \\
\mathcal{R}_{2-1} : & \langle \ p & \leftarrow & 0.5 & ; 1 \ \ \ ; 0.5 \ \rangle \\
\mathcal{R}_{2-2} : & \langle \ p & \leftarrow_{\mathtt{P}} & 1 \ \&_{\mathtt{p}} \ p & ; 1 \ \ \ ; 1 \ \ \ \rangle \\
\mathcal{R}_{2-3-4} : & \langle \ p & \leftarrow_{\mathtt{P}} & 0.7 \ \&_{\mathtt{p}} 0.7 \ \&_{\mathtt{p}} \ q_2 & ; 1 \ \ \ ; 0.49 \rangle \\
\mathcal{R}_{3-4} : & \langle \ p & \leftarrow_{\mathtt{P}} & 0.7 \ \&_{\mathtt{p}} \ q_2 & ; 0.7 \ \ ; 0.49 \rangle \\
\mathcal{R}_{4-5} : & \langle \ q_1 & \leftarrow_{\mathtt{P}} & 0.7 \ \&_{\mathtt{p}} \ q_3 & ; 0.7 \ \ ; 0.49 \rangle \\
\mathcal{R}_{5-6} : & \langle \ q_2 & \leftarrow_{\mathtt{P}} & 0.7 \ \&_{\mathtt{p}} \ q_4 & ; 0.7 \ \ ; 0.49 \rangle \\
& \vdots & & &
\end{array}
$$

It is not difficult to check that, by following our improved thresholded tabulation technique, the rule $\mathcal{R}_{3-4}$ is not considered in the computation of p and no loops

arise in this program. Therefore, the unique solution 0.5 for our initial query $?p$, which is obtained in a small number of computation steps.

4 Conclusions and Future Work

In this paper we have taken profit of our previous advances on fuzzy unfolding in order to reinforce the power of the static preprocess we described in [5], with the new aim of increasing the performance of the fuzzy thresholded tabulation procedure conceived in [6] for the fast execution of MALP programs. We are nowadays lifting our results to the first order case and implementing the method drawn in this paper into the FLOPER platform [12].

References

1. J.M. Almendros-Jiménez, A. Luna, and G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. *Lect. Notes in Computer Science* 6826:186–193, 2011.
2. J. F. Baldwin, T. P. Martin, and B. W. Pilsworth. *Fril—Fuzzy and Evidential Reasoning in Artificial Intelligence*. John Wiley & Sons, Inc., 1995.
3. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
4. M. Ishizuka and N. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In Aravind K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI'85)*, pages 701–703. Morgan Kaufmann, 1985.
5. P. Julián, J. Medina, P.J. Morcillo, G. Moreno, and M. Ojeda-Aciego. A static preprocess for improving fuzzy thresholded tabulation. *Lect. Notes in Computer Science* 6692:429–436, 2011.
6. P. Julián, J. Medina, G. Moreno, and M. Ojeda. Efficient thresholded tabulation for fuzzy query answering. In *Foundations of Reasoning under Uncertainty*, Studies in Fuzziness and Soft Computing 249:125–141, 2010.
7. P. Julián, G. Moreno, and J. Penabad. On Fuzzy Unfolding. A Multi-adjoint Approach. *Fuzzy Sets and Systems*, 154:16–33, 2005.
8. P. Julián, G. Moreno, and J. Penabad. Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. *Journal of Universal Computer Science*, 12(11):1679–1699, 2006.
9. M. Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12:335–367, 1992.
10. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. *Lect. Notes in Artificial Intelligence* 2173:351–364, 2001.
11. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146:43–62, 2004.
12. P. J. Morcillo and G. Moreno. Programming with fuzzy logic rules by using the FLOPER tool. *Lect. Notes in Computer Science* 5321:119–126, 2008.
13. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Proc. of Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.