The Fuzzy Logic Programming Environment FLOPER

Pedro J. Morcillo and Gines Moreno
Department of Computing Systems
University of Castilla-La Mancha
02071, Albacete (Spain)
{pmorcillo, gmoreno}@dsi.uclm.es

Abstract

Declarative programming plays an important role when designing formal methods for software engineering. *Fuzzy Logic Programming* is an interesting and still growing research area that agglutinates the efforts for introducing fuzzy logic into logic programming (LP), in order to incorporate more expressive resources on such languages for dealing with uncertainty and approximated reasoning. The *multi-adjoint logic programming* approach represents a recent and extremely flexible fuzzy logic paradigm for which, unfortunately, we have not found practical tools implemented so far.

In this work we describe a prototype system, the FLOPER tool, which is able to directly translate fuzzy logic programs into Prolog code in order to safely execute these residual programs inside any standard Prolog interpreter in a completely transparent way for the final user. Moreover, the system also generates a low-level representation of the fuzzy code offering debugging (tracing) capabilities and opening the door to the design of more sophisticated program manipulation tasks such as program optimization, program specialization, program analysis and so on.

In order to support and apply formal methods in industry, we think that the development of such fuzzy logic languages and programing tools might have high relevance, especially when designing advanced software applications for medicine, industrial control, and so on.

1 Introduction

Logic Programming [10] has been widely used for problem solving and knowledge representation in the past. Nevertheless, traditional LP languages do not incorporate techniques or constructs to treat explicitly with uncertainty and approximated reasoning. To overcome this situation, during the last decades several fuzzy logic programming systems have been developed where the classical inference mechanism of SLD–Resolution is replaced with a fuzzy variant able to handle partial truth and to reason with uncertainty. Most of these systems implement the fuzzy resolution principle introduced by Lee in [8], such as languages Prolog-Elf [4], Fril [2] and F-Prolog [9].

Following this line, in the original version of [13], a fuzzy logic program is conceived as a set of weighted formulas, where the *truth degree* of each clause is explicitly annotated. The task of computing and propagating truth degrees relies on an extension of the resolution principle, whereas the (syntactic) unification mechanism remains untouched. Continuing this trail, one of the most modern, flexible and evolved fuzzy dialects of Prolog which follow this scheme, is the one presented in[12].

Informally speaking, in the multi-adjoint logic framework, a program can be seen as a set of rules each one annotated by a truth degree, and a goal is a query to the system, i.e., a set of atoms linked with connectives called *aggregators*. A *state* is a pair $\langle \mathcal{Q}, \sigma \rangle$ where $\mathcal{Q}$ is a goal and σ a substitution (initially,

the identity substitution). States are evaluated in two separate computational phases. Firstly, *admissible steps* (a generalization of the classical *modus ponens* inference rule) are systematically applied by a backward reasoning procedure in a similar way to classical resolution steps in pure logic programming, thus returning a computed substitution together with an expression where all atoms have been exploited. This last expression is then interpreted under a given lattice, hence returning a pair $\langle$*truth degree*; *substitution*$\rangle$ which is the fuzzy counterpart of the classical notion of computed answer traditionally used in LP.

The main contribution of this paper is the detailed description of the FLOPER system (see a preliminary introduction in [1]), a *"Fuzzy LOgic Programming Environment for Research"* that we have developed in our research group and which is freely available in: `http://www.dsi.uclm.es/investigacion/dect/FLOPERpage.htm`. Nowadays, the tool provides facilities for executing as well as for debugging (by generating declarative traces) such kind of fuzzy programs, thus fulfilling the gap we have detected in the area. Our implementation methods are based on two different, almost antagonistic ways (regarding simplicity and precision features), for generating pure Prolog code, with some correspondences with other previous attempts described in the specialized literature, specially the ones detailed in [3] and [11].

The outline of this work is as follows. In Section 2 we detail the main features of multi-adjoint logic programming, both syntax and procedural semantics. Next, in Section 3 we propose an elegant method for "compiling" fuzzy programs into standard Prolog code. Sections 4 and 5 explain, respectively, how to execute and debug such programs inside our FLOPER tool, which nowadays is being equipped with new options for performing other advanced program manipulation tasks (transformation, specialization, optimization). The benefits of our approach are highlighted by contrasting them with some related works in Section 6. Finally, in Section 7 we conclude with some lines of future work.

2 Multi-Adjoint Logic Programs

In what follows, we present a short summary of the main features of our language (we refer the reader to [12] for a complete formulation). We work with a first order language, $\mathcal{L}$, containing variables, function symbols, predicate symbols, constants, quantifiers ($\forall$ and $\exists$), and several (arbitrary) connectives to increase language expressiveness. In our fuzzy setting, we use implication connectives ($\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m$) and also other connectives which are grouped under the name of "aggregators" or "aggregation operators". They are 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 $@_1, @_2, \ldots, @_n$). Although the connectives $\&_i$, $\vee_i$ and $@_i$ are binary operators, we usually generalize them as functions with an arbitrary number of arguments. In the following, we often write $@(x_1, \ldots, x_n)$ instead of $@(x_1, @(x_2, \ldots, @(x_{n-1}, x_n)\ldots))$. By definition, the truth function for an n-ary aggregation operator $[\![@]\!] : L^n \to L$ is required to be monotone and fulfills $[\![@]\!](\top, \ldots, \top) = \top$, $[\![@]\!](\bot, \ldots, \bot) = \bot$. Additionally, our language $\mathcal{L}$ contains the values of a multi-adjoint lattice, $\langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n\rangle$, equipped with a collection of adjoint pairs $\langle \leftarrow_i, \&_i\rangle$, where each $\&_i$ is a conjunctor intended to the evaluation of *modus ponens*. In general, the set of truth values L may be the carrier of any complete bounded lattice but, for simplicity, in this paper 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 A is an atomic formula (usually called the *head*) and $\mathcal{B}$ (which is called the *body*) is a formula built from atomic formulas $B_1, \ldots, B_n$ ($n \geq 0$), truth values of L 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. Variables in a rule are assumed to be governed by universal quantifiers. Roughly speaking, a multi-adjoint logic program is a set of pairs $\langle \mathcal{R}; \alpha\rangle$,

where $\mathcal{R}$ is a rule and α is a *truth degree* (a value of L) expressing the confidence which the user of the system has in the truth of the rule $\mathcal{R}$. Often, we will write "$\mathcal{R}$ with α" instead of $\langle \mathcal{R}; \alpha \rangle$.

In order to describe the procedural semantics of the multi–adjoint logic 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 context $\mathcal{C}[]$. Moreover, $\mathcal{V}ar(s)$ denotes the set of distinct variables occurring in the syntactic object s, $\theta[\mathcal{V}ar(s)]$ refers to the substitution obtained from θ by restricting its domain to $\mathcal{V}ar(s)$ and $mgu(E)$ denotes the *most general unifier* of an equation set E. In the following definition, we always consider that A is the selected atom in goal $\mathcal{Q}$.

Definition 2.1 (Admissible Steps) *Let $\mathcal{Q}$ be a goal and let σ be a substitution. The pair $\langle \mathcal{Q}; \sigma \rangle$ is a* state *and we denote by $\mathcal{E}$ the set of states. Given a program $\mathcal{P}$, an* admissible computation *is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathcal{E} \times \mathcal{E})$ is the smallest relation satisfying the following* admissible rules*:*

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

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

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

Note that the 3^{th} case is introduced to cope with (possible) unsuccessful admissible derivations. As usual, rules are taken renamed apart. We shall use the symbols $\rightarrow_{AS1}$, $\rightarrow_{AS2}$ and $\rightarrow_{AS3}$ to distinguish between computation steps performed by applying one of the specific admissible rules. Also, the application of a rule on a step will be annotated as a superscript of the $\rightarrow_{AS}$ symbol.

Definition 2.2 *Let $\mathcal{P}$ be a program and let $\mathcal{Q}$ be a goal. An* admissible derivation *is a sequence $\langle \mathcal{Q}; id \rangle \rightarrow^*_{AS} \langle \mathcal{Q}'; \theta \rangle$. When $\mathcal{Q}'$ is a formula not containing atoms, the pair $\langle \mathcal{Q}'; \sigma \rangle$, where $\sigma = \theta[\mathcal{V}ar(\mathcal{Q})]$, is called an* admissible computed answer *(a.c.a.) for that derivation.*

In order to illustrate our definitions, consider now the following program $\mathcal{P}$ and lattice $([0,1], \leq)$, where $\leq$ is the usual order on real numbers.

$\mathcal{R}_1 : p(X) \leftarrow_{\mathtt{P}} q(X,Y) \&_{\mathtt{G}} r(Y)$ *with* 0.8
$\mathcal{R}_2 : q(a,Y) \leftarrow_{\mathtt{P}} s(Y)$ *with* 0.7
$\mathcal{R}_3 : q(b,Y) \leftarrow_{\mathtt{L}} r(Y)$ *with* 0.8
$\mathcal{R}_4 : r(Y) \leftarrow$ *with* 0.7
$\mathcal{R}_5 : s(b) \leftarrow$ *with* 0.9

The labels `P`, `G` and `L` mean for *Product logic*, *Gödel intuitionistic logic* and *Łukasiewicz logic*, respectively. That is, $[\![\&_{\mathtt{P}}]\!](x,y) = x \cdot y$, $[\![\&_{\mathtt{G}}]\!](x,y) = min(x,y)$, and $[\![\&_{\mathtt{L}}]\!](x,y) = max(0, x+y-1)$. In the following admissible derivation for the program $\mathcal{P}$ and the goal $\leftarrow p(X) \&_{\mathtt{G}} r(a)$, we underline the selected expression in each admissible step:

$\langle \underline{p(X)} \&_{\mathtt{G}} r(a); id \rangle \rightarrow_{AS1}{}^{\mathcal{R}_1}$
$\langle \overline{(0.8} \&_{\mathtt{P}} (\underline{q(X_1, Y_1)} \&_{\mathtt{G}} r(Y_1))) \&_{\mathtt{G}} r(a); \sigma_1 \rangle \rightarrow_{AS1}{}^{\mathcal{R}_2}$
$\langle (0.8 \&_{\mathtt{P}} (\overline{(0.7 \&_{\mathtt{P}} \underline{s(Y_2)})} \&_{\mathtt{G}} r(Y_2))) \&_{\mathtt{G}} r(a); \sigma_2 \rangle \rightarrow_{AS2}{}^{\mathcal{R}_5}$
$\langle (0.8 \&_{\mathtt{P}} ((0.7 \&_{\mathtt{P}} \overline{0.9}) \&_{\mathtt{G}} \underline{r(b)})) \&_{\mathtt{G}} r(a); \sigma_3 \rangle \rightarrow_{AS2}{}^{\mathcal{R}_4}$
$\langle (0.8 \&_{\mathtt{P}} ((0.7 \&_{\mathtt{P}} 0.9) \&_{\mathtt{G}} \overline{0.7})) \&_{\mathtt{G}} \underline{r(a)}; \sigma_4 \rangle \rightarrow_{AS2}{}^{\mathcal{R}_4}$
$\langle (0.8 \&_{\mathtt{P}} ((0.7 \&_{\mathtt{P}} 0.9) \&_{\mathtt{G}} 0.7)) \&_{\mathtt{G}} \overline{0.7}; \sigma_5 \rangle$,

where:

$\sigma_1 = \{X/X_1\}$,
$\sigma_2 = \{X/a, X_1/a, Y_1/Y_2\}$
$\sigma_3 = \{X/a, X_1/a, Y_1/b, Y_2/b\}$
$\sigma_4 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b\}$
$\sigma_5 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b, Y_4/a\}$

So, since $\sigma_5[\mathcal{V}ar(\mathcal{Q})] = \{X/a\}$, the a.c.a. associated to this admissible derivation is: $\langle (0.8 \&_{\mathtt{P}} ((0.7 \&_{\mathtt{P}} 0.9) \&_{\mathtt{G}} 0.7)) \&_{\mathtt{G}} 0.7; \{X/a\} \rangle$. Now, after evaluating the first arithmetic expression (where all atoms have been solved), we obtain the final fuzzy computed answer (f.c.a.) $\langle 0.504; \{X/a\} \rangle$.

3 Translating Multi-adjoint Logic Programs into Pure Prolog Code

This section is devoted to detail a simple, but powerful method for translating fuzzy programs into directly executable standard Prolog code [1]. The final goal is that the compiled code be executed in any Prolog interpreter in a completely transparent way for the

final user, i.e., our intention is that after introducing fuzzy programs and fuzzy goals to the system, it be able to return fuzzy computed answers (i.e., pairs including truth degrees and substitutions) even when all intermediate computations have been executed in a pure (not fuzzy) logic environment.

The syntactic conventions that our system accepts when parsing multi-adjoint logic programs are very close to those seen in Section 2. For instance, we can code the program of our running example as:

```
p(X) <prod q(X,Y) &godel r(Y)  with 0.8.
q(a,Y) <prod s(Y) with 0.7.
q(b,Y)  <luka  r(Y) with 0.8.
r(Y)  with 0.7.
s(b) with 0.9.
```

The reader may easily check the strong similarities between the previous code and the program shown in the example of the previous section. During the parsing process, our system produces Prolog code following these guidelines:

• Each atom appearing in a fuzzy rule is translated into a Prolog atom extended with an extra argument, called *truth variable* of the form $\texttt{\_TV}_\texttt{i}$. The intended objective of this anonymous variable, is to contain the truth degree obtained after the subsequent evaluation of such atom.

• The role of aggregator operators can be easily played by standard Prolog clauses defining "aggregator predicates" as follows:

```
agr_aver(X,Y,Z) :-  Z is (X+Y)/2.
and_prod(X,Y,Z):-   Z is X * Y.
and_godel(X,Y,Z):-  (X=<Y,Z=X;X>Y,Z=Y).
and_luka(X,Y,Z):-   H is X+Y-1,
                    (H=<0,Z=0;H>0,Z=H).
```

• Program facts (i.e., rules with no body) are expanded at compilation time to Prolog facts, where the additional argument of the (head) atom, instead of being a truth variable, is just the truth degree of the corresponding rule. For instance, rules $\mathcal{R}_4$ and $\mathcal{R}_5$ in our running example, can be represented by the Prolog facts `r(Y,0.7)` and `s(b,0.9)`, respectively.

• Program rules are translated into Prolog clauses by performing the appropriate calls to the atoms presented in its body. Regarding the calls to aggregator predicates, they must be postponed at the end of the body, in order to guarantee that the truth variables used as arguments be correctly instantiated when needed. In this sense, it is also important to respect an appropriate ordering when performing the calls. In particular, the last call must necessarily be to the "aggregator predicate" modeling the adjoint conjunction of the implication operator of the rule, by also using its truth degree. For instance, rules $\mathcal{R}_1, \mathcal{R}_2$ and $\mathcal{R}_3$ in the example of the previous section, can be represented by the Prolog clauses:

```
p(X,_TV0)   :- q(X,Y,_TV1),r(Y,_TV2),
               and_godel(_TV1,_TV2,_TV3),
               and_prod(0.8,_TV3,_TV0).
q(a,Y,_TV0) :- s(Y,_TV1),
               and_prod(0.7,_TV1,_TV0).
q(b,Y,_TV0) :- r(Y,_TV1),
               and_luka(0.8,_TV1,_TV0).
```

• A fuzzy goal is translated into a Prolog goal where the corresponding calls to atoms appear in their textual order before the ones for "aggregator predicates". Since aggregators are not associative in general, they must appear in an appropriate sequence, as also occurred with the translation of clause bodies explained before. For instance, the goal $\leftarrow p(X) \&_G r(a)$ in our running example, can be represented by the following Prolog goal:

```
?-    p(X,_TV1),r(a,_TV2),
      and_godel(_TV1,_TV2,_TV3).
```

Following this method, we have just translated into standard Prolog code the multi-adjoint logic program and goal shown in previous sections. In particular, we have used Sicstus Prolog v.3.12.5 for executing them as well as for implementing the FLOPER tool, whose capabilities for running/debugging fuzzy programs will be explained immediately.

4 Running Fuzzy Programs

As detailed in [1], our parser has been implemented by using the classical DCG's (*Definite Clause Grammars*) resource of the Prolog language, since it is a convenient notation for expressing grammar rules. The application contains about 300 clauses and once it is loaded inside a Prolog interpreter (in our case, Sicstus Prolog), it shows a menu which includes the following options (among other classical ones, like `"clean"`, `"stop"` or `"quit"`):

• "load", in order to charge a prolog file with extension '.pl'. This action is useful for reading a file containing a set of clauses implementing aggregators, user predicates, etc. Nevertheless, the original connectives of the *Product*, *Gödel* and *Łukasiewicz logic*, expressed in the Prolog style seen in the previous section, are defined in file prelude.pl, which is automatically loaded by the system at the beginning of each work session.

• "parse", for loading a fuzzy program included in a file with extension '.fpl'. In order to simultaneously perform the parsing process with the code generation, each *parsing* predicate used in DCG's rules, has been augmented with a variable as extra argument which is intended to contain the Prolog code generated after parsing the corresponding fragment of fuzzy code. We also admit the presence of pure Prolog clauses inside a '.fpl' file, by including them between '$'.

• "list", which displays the set of Prolog clauses loaded from a '.pl' file as well as those ones obtained after compiling an '.fpl' file. Of course, the original fuzzy program contained in this last file is also displayed.

• "save", which stores the resulting Prolog code into a file. We wish to point out that the set of clauses obtained during the compilation process is also automatically *asserted* in the data base of the Prolog interpreter, which obviously also contains the clauses implementing the proper tool. This action helps the development of the following option.

• "run", for executing a fuzzy goal after being introduced from the keyboard by using option "intro". As we have seen in the previous section, if the goal provided by the user is 'p(X) &godel r(a)', then the system translates it into the standard Prolog goal 'p(X,_TV1),r(a,_TV2), and_godel(_TV1,_TV2,_TV3)'. However, this query needs a final manipulation before being executed, which consists in renaming its last truth variable (_TV3) by Truth_degree. Now, note that the set of non anonymous variables in the resulting Prolog goal, are simply those ones belonging to the original fuzzy goal (i.e., X) and the one containing its associated 'Truth_degree'. Then, after reevaluating the Prolog goal 'p(X,_TV1),r(a,_TV2), and_godel(_TV1,_TV2, Truth_degree)', the Prolog interpreter returns the following pair of desired fuzzy computed answers: [Truth_degree=0.504,X=a] and [Truth_degree=0.4,X=b].
The previous set of options suffices for running fuzzy programs: all internal computations (including compiling and executing) are pure Prolog derivations whereas inputs (fuzzy programs and goals) and outputs (fuzzy computed answers) have always a fuzzy taste, which produces the illusion on the final user of being working with a purely fuzzy logic programming tool.

However, when trying to go beyond program execution, our method becomes insufficient. In particular, observe that we can only simulate complete fuzzy derivations (by performing the corresponding Prolog derivations based on SLD-resolution) but we can not generate partial derivations or even apply a single admissible step on a given fuzzy expression. This kind of low-level manipulations are mandatory when trying to incorporate to the tool some program transformation techniques such as those based on fold/unfold or partial evaluation we have described in [5, 6, 7]. For instance, our fuzzy unfolding transformation is defined as the replacement of a program rule $\mathcal{R} : (A \leftarrow_i \mathcal{B}$ with $\alpha)$ by the set of rules $\{A\sigma \leftarrow_i \mathcal{B}'$ with $\alpha \mid \langle \mathcal{B}; id\rangle \rightarrow_{AS} \langle \mathcal{B}'; \sigma\rangle\}$, which obviously requires the implementation of mechanisms for generating derivations of a single step, rearranging the body of a program rule, applying substitutions to its head, etc. To achieve this aim, we have conceived a new low-level representation for the fuzzy code which nowadays already offers the possibility of performing debugging actions such as tracing a FLOPER work session.

5 Debugging Fuzzy Programs

Each *parsing* predicate used in DCG's rules (which already contains a parameter allocating the Prolog code obtained after the compilation process) has also been augmented with

```
% TRACE 1: Execution tree with depth 4 for goal p(a) w.r.t. the multi-adjoint logic program P1.
R0 < p(a), {} >
   R1 < &prod(0.9,q(a)), {X1/a} >
       R3 < &prod(0.9,&luka(0.7,q(a))), {X1/a,X7/a} >
           R3 < &prod(0.9,&luka(0.7,&luka(0.7,q(a)))), {X1/a,X7/a,X11/a} >
               R3 < &prod(0.9,&luka(0.7,&luka(0.7,&luka(0.7,q(a))))), {X1/a,...} >
   R2 < &godel(0.8,r(a)), {X2/a} >
       R4 < &godel(0.8,0.6), {X2/a} >

% TRACE 2: Execution tree with depth 2 for goal p(X) w.r.t. the multi-adjoint logic program P2.
     R0 < p(X), {} >
         R1 < &prod(0.9,@aver(1,p(b))), {X/a} >
             R0 < &prod(0.9,@aver(1,0)), {X/a} >
```

Figure 1: Traces and execution trees generated by FLOPER.

a second extra argument for storing now the new representation associated to the corresponding fragment of parsed fuzzy code. For instance, after parsing the first rule of our program, we obtain the following expression (whose components have obvious meanings):

```
rule(number(1),
 head(atom(pred(p,1),var('X')])),
 impl('prod'),
 body(and('godel',2,
  [atom(pred(q,2),[var('X'),var('Y')]),
   atom(pred(r,1),[var('Y')])])),
 td(0.8)).
```

Once obtained at compilation time, this term is then *asserted* into the data base of the Prolog interpreter as a Prolog fact, thus making accessible this low-level representation of the fuzzy rule to the whole application. Two more examples: substitutions are modeled by lists of terms of the form `link(V,T)` where `V` and `T` contains the code associated to an original variable and its corresponding (linked) fuzzy term, respectively, whereas an state is represented by a term with functor `state/2`. We have implemented predicates for manipulating such kind of code at a very low level in order to unify expressions, compose substitutions, apply admisible/interpretive steps, etc.

With this nice representation, we can also build execution trees with any level of depth, thus producing terms of the the form `tree(S,L)`, where `S` represents the state rooting the tree, and `L` is the list containing its set of children trees. Recently, FLOPER has been equipped with two new options, called `"tree"` and `"depth"`, for visualizing such trees and fixing the maximum length allowed for their branches (initially 3), respectively. Apart for the important role they could play in future developments, these options are nowadays very useful for debugging purposes: in particular, they allow the possibility of generating declarative traces of the execution of a given goal and program, as showed in Figure 1.

By displaying execution trees, FLOPER provides a much more precise information than the one obtained by using the simple `"run"` option based on the method described in Section 3. The complete trace of the execution of a given goal w.r.t. a program seems to be crucial when the `"run"` option fails. Let us explain its power by means of two examples which, thanks to their simplicity, reinforce this claim. Firstly, consider the following fuzzy program $\mathcal{P}_1$:

```
p(X) <prod q(X) with 0.9.
p(X) <godel r(X) with 0.8.
q(X) <luka q(X) with 0.7.
r(a) with 0.6.
```

For goal `p(a)`, FLOPER displays the first tree (trace 1) showed in Figure 1. Observe that each node contains an state (composed by the corresponding goal and substitution) preceded by the number of the program rule used by the admissible step leading to it (root nodes are always labeled with the virtual, non existing rule `R0`). Nodes belonging to the same branch appear in different lines appropriately indented to help the readability of the figure. In our case, the tree contains only two different branches. It is easy to see that the first one, corresponding to the first five lines of the figure, represents an infinite branch, whereas the second one, identified by lines 1, 6 and 7, indicates that the goal has just one solution with truth degree `0.6` (which is the result of evaluating the arithmetic expression `&godel(0.8,0.6)`).

It is important to remark that, when analyzing the tree with care, we can conclude that the original goal is solvable, even when by using the `"run"` op-

tion of FLOPER, the system answers ‘‘`There is no solution`’’, after aborting the infinite loop in which the Prolog interpreter falls down when generating (the SLD-resolution derivation associated to) the first branch of the tree.

Our second example is not involved with infinite branches, but it copes with other kind of (pure Prolog) unsuccessful behaviour. Consider now a fuzzy program, say $\mathcal{P}_2$, containing the single rule ‘`p(a)` $<$ `prod @aver(1,p(b)) with 0.9`’ (where the average aggregator `@aver` has the obvious meaning: see its Prolog-based definition in Section 3) which, once parsed by FLOPER, is translated into the following pure Prolog clause: `p(a,TV0)` $:-$ `p(b,_TV1),agr_aver(1,_TV1,_TV2), and_prod(0.9,_TV2,TV0)`. It is easy to see that, in order to execute goal `p(X)` by means of the `"run"` option, the Prolog interpreter will fail when trying to solve the first atom, `"p(b,_TV1)"`, appearing in the body of this Prolog clause. However, in the fuzzy setting we know that the proposed goal has a solution, as revealed by the (single) successful branch appearing in the second trace of Figure 1. By applying and admissible step of kind 3 (see $\rightarrow_{AS3}$ in Definition 2.1), on the second node of the tree, we generate the final state showed in the third line of the figure (the system simply replaces the non solvable selected atom `p(b)` by the lowest truth degree 0). Note that this last state (labeled with the virtual rule `R0` -as also occurs with the root node- because no program rule has been applied to perform the computation), once evaluated the associated arithmetic expression, returns a fuzzy computed answer confirming that `p(X)` is true with truth degree 0.45 when `X` $=$ `a`.

As we have seen, the generation of traces based on execution trees, contribute to increase the power of FLOPER by providing debugging capabilities which allows us to discover solutions for queries even when a pure Prolog compilation-execution process fails. For the future and also supported on the generation of execution trees, we plan to introduce new options into the FLOPER menu implementing all the transformation techniques we are proposed in the past [5, 6, 7]: the key point is the correct manipulation of the leaves of this kind of partially evaluated trees, in order to produce unfolded rules, reductants, and so on.

6 Related Work

The multi-adjoint logic approach and the fuzzy logic language described in [3] are very close between themselves, with a similar syntax based on “weighted” rules and levels of flexibility and expressiveness somehow comparable. However, whereas in the fuzzy language presented in [3] truth degrees are based on Borel Algebras (i.e., union of intervals of real numbers), in the so called *multi-adjoint logic programming* approach of [12, 11] truth degrees are elements of any given lattice. Other important difference between both languages emerge at an operational level, since the underlying procedural principle of the language of [3] introduces several problems when considering most of the transformation techniques we are developing in our group: the real problem does not appear only at the syntactic level, but what is worse, the major inconvenience is the need for redefining the core of its procedural mechanism to cope with constraints possibly mixed with atoms. In this last setting, computation steps are described by means of an state transition system where, instead of two elements, each state contains three components $\langle atoms, substitution, constraints\rangle$. This strict separation of atoms and constraints (in both, computation states and clause bodies) represents a severe obstacle for the adaptation of our notion of unfolding rule since it is neither easy to execute nor to code on the body of unfolded clauses the constraints generated by those computation steps performed at transformation time. This is one of the most important reasons for which, in our research group, we are mainly concerned with the approach of [12]. Despite the needs for more research efforts, our approach reported in this paper enjoys the following advantages w.r.t. [3]: 1) we think that using standard Prolog instead of $CLP(R)$ will make our ideas more accessible to a wider audience and 2) as we have seen in previous sections, our Prolog code generation (and implementation) largely helped us to produce a low level representation of the final code very useful for debugging and transformation purposes.

Focusing now in the multi-adjoint logic approach, it is unavoidable to mention the implementation issues documented in [11]. Like our proposal, we all deal with the same target ([0;1]-valued) multi-adjoint logic language[1], and also our developments are based in pure Prolog code (even when they are supported on a neural net architecture). However, whereas they are restricted to the propositional case, we have lifted our results to the more gen-

[1] We all focus in a simple lattice whose carrier set is the real interval $[0, 1]$ and the connectives are collected from classical fuzzy logics (as the product, Łukasiewicz and Gödel intuitionistic logic). An high-priority task for future developments will be to let our system accept fuzzy programs as well as multi-adjoint lattices in a parametric way, which implies the design of appropriate protocols, interfaces, etc.

eral first-order case. Moreover, the procedural semantics implemented in [11] has being conceived as a bottom-up procedure where the repeated iteration of an appropriately defined consequence operator reproduces the model of a program, thus obtaining the computed truth-values of all propositional symbols involved in that program (in a parallel way). In a complementary sense (which invokes the integration of both methods in a single framework), the executing and debugging "query answering" procedures implemented in FLOPER, are goal-oriented and have a top-down behaviour.

7 Conclusions and Future Work

In this paper we were concerned with implementation techniques for fuzzy logic programming and more exactly, for the multi-adjoint logic approach, which enjoys high levels of expressivity and a clear operational mechanism. Apart from [11] and our prototype tool FLOPER (see a preliminary description in [1] and visit `http://www.dsi.uclm.es/investigacion/dect/FLOPERpage.htm`), there are not abundant tools available in practice. We have firstly proposed a technique for running such kinds of programs based on a "transparent compilation process" to standard Prolog code. Secondly, we have next proposed a low-level representation of the fuzzy code allowing the possibility of debugging (by generating declarative traces) the execution of a given program and goal. This last development also opens the door to implement new and powerful program manipulation techniques in which we are working nowadays. These actions, together with the study of mechanism for surpassing the simpler case of modeling truth degrees with real numbers, are some prioritary tasks in our research group for the near future.

Acknowledgements

This work has been partially supported by the EU (FEDER), and the Spanish Science and Education Ministry (MEC) under grants TIN 2004-07943-C04-03 and TIN 2007-65749.

References

[1] J.M. Abietar, P.J. Morcillo, and G. Moreno. Designing a software tool for fuzzy logic programming. In T.E. Simos and G. Maroulis, editors, *Proc. of the International Conference of Computational Methods in Sciences and Engineering ICCMSE'07, Volume 2 (Computation in Modern Science and Engineering)*, pages 1117–1120. American Institute of Physics, Springer, 2007.

[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] S. Guadarrama, S. Muñoz, and C. Vaucheret. Fuzzy Prolog: A new approach using soft constraints propagation. *Fuzzy Sets and Systems, Elsevier*, 144(1):127–150, 2004.

[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). Los Angeles, CA, August 1985.*, pages 701–703. Morgan Kaufmann, 1985.

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

[6] 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.

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

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

[9] Deyi Li and Dongbo Liu. *A fuzzy Prolog database system*. John Wiley & Sons, Inc., 1990.

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

[11] J. Medina, E. Mérida-Casermeiro, and M. Ojeda-Aciego. A neural implementation of multi-adjoint logic programs via sf-homogeneous programs. *Mathware & Soft Computing*, XII:199–216, 2005.

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

[13] P. Vojtáš and L. Paulík. Soundness and completeness of non-classical extended SLD-resolution. In R. Dyckhoff et al, editor, *Proc. ELP'96 Leipzig*, pages 289–301. LNCS 1050, Springer Verlag, 1996.