Building a Fuzzy Logic Programming Tool[1]

José M. Abietar, Pedro J. Morcillo and Ginés Moreno

Dept. of Computing Systems. University of Castilla-La Mancha. 02071, Albacete (Spain).

JoseM.Abietar@alu.uclm.es, PedroJ.Morcillo@alu.uclm.es, Gines.Moreno@uclm.es

Abstract

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 is a recent and extremely flexible fuzzy logic paradigm for which, unfortunately, we have not found practical tools implemented till now. In this work, we describe a prototype system that we are just developing to fulfill this gap. As an starting point, our tool is able to directly translate fuzzy logic programs into Prolog code in a similar style to other implementations described in the literature for analogous fuzzy logic languages. This action suffices for executing fuzzy programs inside a Prolog interpreter in a completely transparent way for the final user. However, since this option has a rather limited power for program transformation purposes, we are nowadays incorporating to the system the capability for generating a low-level representation of the fuzzy code with the final goal of building (partial) execution trees, unfolded rules, reductants, 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 [3], Fril [1] 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, two of the most modern, flexible and evolved fuzzy dialects of Prolog which follow this scheme, are the ones presented in [2] and [12]. Both approaches are very close between themselves, with a similar syntax based on "weighted" rules and levels of flexibility and expressiveness perfectly comparable. However, whereas in the fuzzy language presented in [2] 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 [11, 12] truth degrees are elements of any given lattice.

Perhaps the main differences between both languages emerge at an operational level. 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

[1]This work was partially supported by the EU (FEDER), and the Spanish Science and Education Ministry (MEC) under grant TIN 2004-07943-C04-03.

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. During the *operational* one, *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 during what we call the *interpretive* phase, 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.

On the other hand, the underlying operational principle of the language of [2] introduces several problems when considering most of the transformation techniques we are developing in our group. As we detail in [5], the adequacy of this language for being used as the basis of a fuzzy unfolding rule is rather limited: 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. For this reasons, in this work we are mainly concerned with the approach of [11, 12], even when several important ideas regarding implementation issues have been inspired by [2] where an interpreter conceived using Constraint Logic Programming over real numbers ($CLP(\mathcal{R})$) has been efficiently implemented by directly translating source programs into executable CLP-based Prolog code. In particular, our notions of multi-adjoint lattice and interpretive step have direct correspondences with constraint domain and constraint solving, respectively, in the $CLP(\mathcal{R})$ representation of [2].

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 $\langle atoms, substitution, constraints \rangle$. Initial states have the first component (input) fulfilled with a set of atoms of a given goal and the two last components (outputs) are empty. Vice versa, in final states the goal component is empty whereas the two last ones represents the fuzzy computed answer (substitution and truth degree) for the original goal. Remember that the notion of state used in the multi–adjoint logic approach avoids the last component since atoms, aggregators and truth degrees can safely cohabit inside the first component of an state, and also in the body of (transformed) program rules, which enables the effective definition of unfolding in our setting [4, 5]. Conversely, in the language of [2], the 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.

Anyway, we remark again that we can profit of the ideas presented in [2] when implementing our tool. In fact, we think that when using Borel lattices[1] for modeling the fuzzy component of truth-degrees in a given multi-adjoint logic program, it could be almost immediate to adopt a "constraint-solving" mechanism very similar to the one used there.

The outline of this work is as follows. After summarizing in Section 2 the main features of multi-adjoint logic programming at a syntactic level, in Section 3 we detail its procedural semantics. Next, in Section 4 we propose an elegant method for "compiling" fuzzy programs into standard Prolog code. Section 5 explains the technical details of the prototype system we are designing nowadays, which apart from implementing the previous method, is being equipped with options that manipulates the fuzzy code at a low level, in order to help the implementation of future options for performing advanced manipulations on such programs. Finally, in Section 6 we conclude by proposing some lines of future work.

[1]In our preliminary implementation we only consider the real interval $[0, 1]$ as carrier set.

2 Multi-Adjoint Logic Programs

This section is a short summary of the main features of our language. We refer the reader to [11, 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 $@(x_1, @(x_2, \ldots, @(x_{n-1}, x_n) \ldots))$.

Aggregation operators are useful to describe/specify user preferences. An aggregation operator, when interpreted as a truth function, may be an arithmetic mean, a weighted sum or in general any monotone application whose arguments are values of a complete bounded lattice L. For example, if an aggregator $@$ is interpreted as $[\![@]\!](x, y, z) = (3x+2y+z)/6$, we are giving the highest preference to the first argument, then to the second, being the third argument the least significant. By definition, the truth function for an n-ary aggregation operator $[\![@]\!] : L^n \to L$ is required to be monotonous 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[2] 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 $\langle \mathcal{R}; \alpha \rangle$. Observe that, truth degrees are axiomatically assigned (for instance) by an expert.

3 Procedural Semantics

The procedural semantics of the multi–adjoint logic language $\mathcal{L}$ can be thought as an operational phase followed by an interpretive one [5].

In the following, $\mathcal{C}[A]$ denotes 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* (see [7]) of an equation set E. In the following definition, we always consider that A is the selected atom in goal $\mathcal{Q}$.

Definition 3.1 (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 $\to_{AS} \subseteq (\mathcal{E} \times \mathcal{E})$ is the smallest relation satisfying the following* admissible rules*:*

[2]It is noteworthy that a symbol $\&_j$ of $\mathcal{L}$ does not always need to be part of an adjoint pair.

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 3th 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 3.2 *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.*

Example 3.3 *Let* $\mathcal{P}$ *be the following program and let* $([0,1], \leq)$ *be the lattice where* $\leq$ *is the usual order on real numbers.*

$\mathcal{R}_1 : p(X) \leftarrow_{\texttt{prod}} q(X,Y) \&_{\texttt{G}}\, r(Y)$	*with*	0.8
$\mathcal{R}_2 : q(a,Y) \leftarrow_{\texttt{prod}} s(Y)$	*with*	0.7
$\mathcal{R}_3 : q(b,Y) \leftarrow_{\texttt{luka}} r(Y)$	*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,* $[\![\&_{\texttt{P}}]\!](x,y) = x \cdot y$*,* $[\![\&_{\texttt{G}}]\!](x,y) = min(x,y)$*, and* $[\![\&_{\texttt{L}}]\!](x,y) = max(0, x+y-1)$*. In the following admissible derivation for the program* $\mathcal{P}$ *and the goal* $\leftarrow p(X) \&_{\texttt{G}} r(a)$*, we underline the selected expression in each admissible step:*

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

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:* $\langle (0.8 \&_{\texttt{P}} ((0.7 \&_{\texttt{P}} 0.9) \&_{\texttt{G}} 0.7)) \&_{\texttt{G}} 0.7; \{X/a\}\rangle$*.*

If we exploit all atoms of a goal, by applying admissible steps as much as needed during the operational phase, then it becomes a formula with no atoms which can be then directly interpreted in the multi–adjoint lattice L.

Definition 3.4 (Interpretive Step) *Let* $\mathcal{P}$ *be a program,* $\mathcal{Q}$ *a goal and* σ *a substitution. We formalize the notion of* interpretive computation *as a state transition system, whose transition relation* $\rightarrow_{IS} \subseteq (\mathcal{E} \times \mathcal{E})$ *is defined as the least one satisfying:* $\langle Q[@(r_1, r_2)]; \sigma\rangle \rightarrow_{IS} \langle Q[@(r_1,r_2)/[\![@]\!](r_1,r_2)]; \sigma\rangle$*, where* $[\![@]\!]$ *is the truth function of connective* @ *in the lattice* $\langle L, \preceq\rangle$ *associated to* $\mathcal{P}$*.*

Definition 3.5 *Let* $\mathcal{P}$ *be a program and* $\langle Q; \sigma\rangle$ *an a.c.a., that is,* $\mathcal{Q}$ *is a goal not containing atoms. An* interpretive derivation *is a sequence* $\langle Q; \sigma\rangle \rightarrow^{*}_{IS} \langle Q'; \sigma\rangle$*. When* $Q' = r \in L$*, being* $\langle L, \preceq\rangle$ *the lattice associated to* $\mathcal{P}$*, the state* $\langle r; \sigma\rangle$ *is called a* fuzzy computed answer *(f.c.a.) for that derivation.*

Example 3.6 *We complete the previous derivation of Example 3.3 by executing the necessary interpretive steps to obtain the final fuzzy computed answer* $\langle 0.504; \{X/a\}\rangle$ *with respect to lattice* $([0,1], \leq)$*.*

$$\langle (0.8 \&_{\texttt{P}} ((\underline{0.7 \&_{\texttt{P}} 0.9}) \&_{\texttt{G}} 0.7)) \&_{\texttt{G}} 0.7; \{X/a\}\rangle \rightarrow_{IS}$$
$$\langle (0.8 \&_{\texttt{P}} (\underline{0.63 \&_{\texttt{G}} 0.7})) \&_{\texttt{G}} 0.7; \{X/a\}\rangle \rightarrow_{IS}$$
$$\langle (\underline{0.8 \&_{\texttt{P}} 0.63}) \&_{\texttt{G}} 0.7; \{X/a\}\rangle \rightarrow_{IS}$$
$$\langle \underline{0.504 \&_{\texttt{G}} 0.7}; \{X/a\}\rangle \rightarrow_{IS}$$
$$\langle 0.504; \{X/a\}\rangle$$

4 Translating Fuzzy Programs into Prolog Code

This section is devoted to detail a simple, but powerful method for translating fuzzy programs into directly executable standard Prolog code. 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.

Our approach is somehow inspired by [2], where an interpreter conceived using Constraint Logic Programming over real numbers ($CLP(\mathcal{R})$) has been efficiently implemented. We remark once again that this approach represents a real and interesting inspiration source for implementation issues, specially taking into account that there is not yet available an interpreter for the language originally described in [12]. In fact, due to the parallelism of both fuzzy logic programming approaches, the multi-adjoint language also admits a "constraint solving"-based implementation when considering Borel lattices, by adapting the guidelines detailed in [2]. Nowadays, due to the preliminary state of the systems we are designing, we have opted by a direct translation to standard Prolog code by focusing 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). Although the expressive power of our preliminary approach is rather limited, and it needs more research/implementation efforts, we wish to remark some of its advantages:

- We think that using standard Prolog instead of $CLP(R)$ will make our ideas more accessible to a wider audience.

- The approach of [2] is based in Borel Algebras, which is clearly a more powerful way for modeling truth degrees instead of real numbers in the interval $[0, 1]$ as we do. However, in both approaches the underlying lattice associated to fuzzy programs are fixed, whereas in the general multi-adjoint logic scheme, these lattices might vary as much as wanted. In this sense 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.

- Moreover, as we will see in the next section, our Prolog code generation will largely help us to produce a low level representation of the final code very useful for transformation purposes.

Now, we are ready to summarize the main features of our technique. 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 examples as:

```
p(X) <P q(X,Y) &G r(Y)  with 0.8.
q(a,Y) <P s(Y) with 0.7.
q(b,Y)  <L  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 Example 3.3. 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 `_TVi`. 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:

```
and_P(X,Y,Z):- Z is X * Y.
and_G(X,Y,Z):- (X=<Y,Z=X;X>Y,Z=Y).
and_L(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 Example 3.3, 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 Example 3.3, can be represented by the Prolog clauses:

```
p(X,_TV0)   :- q(X,Y,_TV1),r(Y,_TV2),
               and_G(_TV1,_TV2,_TV3),
               and_P(0.8,_TV3,_TV0).
q(a,Y,_TV0) :- s(Y,_TV1),
               and_P(0.7,_TV1,_TV0).
q(b,Y,_TV0) :- r(Y,_TV1),
               and_L(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 $\leftarrow p(X) \&_{\mathtt{G}} r(a)$ in Example 3.3, can be represented by the following Prolog goal:

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

Following this method, we have just translated to 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 tool that we are going to explain immediately.

5 The Prototype Tool

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 (among others) options for:

• `Loading` 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.

• `Parsing` 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 preceding them with a `'$'` symbol.

• `Listing` 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.

• `Saving` the resulting Prolog code into a file. Here we want 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.

• `Executing` a fuzzy goal after being introduced from the keyboard. As we have seen in the previous section, if the goal provided by the user is `p(X) &G`

`r(a)`, then the system translates it into the standard Prolog goal `p(X,_TV1),r(a,_TV2), and_G(_TV1,_TV2,_TV3)`. However, this expression needs a final manipulation before being executed, which consists in renaming its last truth variable (in this case `_TV3`) by `With_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 (i.e., `With_Truth_Degree`). Then, after reevaluating the Prolog goal `?- p(X,_TV1),r(a,_TV2), and_G(_TV1,_TV2, With_Truth_Degree)`, the Prolog interpreter returns the following pair of desired fuzzy computed answers:

```
X=a, With_Truth_Degree=0.504;
X=b, With_Truth_Degree=0.4;
no
```

The previous sets of options suffices for executing 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 user of being working with a purely fuzzy 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/interpretive 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 [4, 5, 6]. 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 $\alpha \mid \langle \mathcal{B}; id\rangle \rightarrow_{AS/IS} \langle \mathcal{B}'; \sigma\rangle\}$, 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: 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 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(‘P’),
  body(and(‘G’,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. From here, we are just implementing all the transformation techniques we are proposed in the past: 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 Conclusions and Future Work

In this paper we were concerned with implementation techniques for fuzzy logic programming, being this field an emergent declarative paradigm for which there are not abundant tools available in practice. We have adopted the so called multi-adjoint logic approach, due to its high levels of expressivity and its clear operational mechanism. We have firstly proposed a technique for executing such kinds of programs based on a "transparent compilation process" to standard Prolog code. To the best of our knowledge, it is the first time that this method (which have some correspondences with other approaches rehearsed in other similar fuzzy settings) has been put in practice in the multi-adjoint logic approach. Going on deeper, we have next proposed a second compilation way which produces a low-level representation of the fuzzy code thus enabling the possibility of performing "partial computations" which opens the door to new program manipulation techniques in which we are working nowadays. This actions, together with the study of mechanism for surpassing the simpler case of modeling truth degrees with real numbers, are some prioritary task in our group for the near future.

References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] P. Julián, G. Moreno, and J. Penabad. Efficient Reductants Calculi using Partial Evaluation Techniques with Thresholding. In P. Lucio, editor, *Electronic Notes in Theoretical Computer Science*, page 15. Elsevier (in press), 2007.

[7] J. L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

[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, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. *Proc of Logic Programming and Non-Monotonic Reasoning, LPNMR'01, Springer-Verlag, LNAI*, 2173:351–364, 2001.

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