Modeling Interpretive Steps into the FLOPER Environment

Pedro J. Morcillo[1], **Ginés Moreno**[1], **Jaime Penabad**[2] **and Carlos Vázquez**[1]
[1] Department of Computing Systems and [2] Department of Mathematics
High School of Computer Science Engineering, University of Castilla-La Mancha, Albacete (02071), Spain

Abstract— *FLOPER is a "Fuzzy LOgic Programming Environment for Research" trying to help the development of applications supporting approximated reasoning and uncertain knowledge in the fields of AI, symbolic computation, soft-computing, semantic web, declarative programming and so on. The tool, which is able to directly translate a powerful kind of fuzzy logic programs belonging to the so-called "multi-adjoint logic approach" into standard Prolog code, currently offers running/debugging/tracing capabilities with close connections to other sophisticated manipulation techniques (program optimization, program specialization, etc.) under development in our research group. In this setting, the execution of a program is done in two separate phases: operational and interpretive. During the first stage, "admissible steps" are systematically applied in a similar way to classical resolution steps in pure logic programming (LP), thus returning an expression where all atoms have been exploited. This last expression is then interpreted under a given lattice during the so called interpretive phase. Whereas the operational phase has been successfully formalized in the past, more effort is needed to clarify the notion of "interpretive step". In this paper, we provide a real implementation into FLOPER of a refinement of this concept which fairly models at a very low level the computational behaviour of the interpretive phase aiming too to visualize the computational effort required to solve a goal. The resulting method also puts in practice an accurate and realistic way for future efficiency studies regarding our ongoing research on fuzzy techniques for Fold/Unfold, Partial Evaluation, Thresholded Tabulation, etc.*

Keywords: Fuzzy Logic Programming, Languages and Techniques for AI, Software Tools, Computational Cost Measures

1. Introduction

Among other purposes, research in *Declarative Programming* and *Fuzzy Logic* has traditionally provided languages and programming techniques for AI, soft-computing, and so on. In particular, *Logic Programming* [1] has been widely used for problem solving and knowledge representation in the past, with recognized influences in the field of AI [2], [3]. Nevertheless, traditional logic languages do not incorporate techniques or constructs to explicitly treat with uncertainty and approximate reasoning.

To fulfill this gap, *Fuzzy Logic Programming* has emerged as an interesting and still growing research area trying to agglutinate the efforts for introducing fuzzy logic into logic programming. During the last decades, several fuzzy logic programming systems have been developed [4], [5], [6], [7], [8], [9], where the classical inference mechanism of SLD–resolution is replaced with a fuzzy variant which is able to handle partial truth and to reason with uncertainty.

This is the case of *multi-adjoint logic programming* [10], [11], [12], one of the most powerful and promising approaches in the area. In this 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 plus a substitution (initially the empty substitution, denoted by id). *Admissible steps* (a generalization of the classical *modus ponens* inference rule) are systematically applied on goals in a similar way to classical resolution steps in pure logic programming, thus returning an state composed by a computed substitution together with an expression where all atoms have been exploited. Next, during the so called interpretive phase, (see [13], [14]), this expression is 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 used in pure logic programming.

On the other hand, the most common approach for analyzing the efficiency of a program is measurement of its execution time and memory usage. However, in order to (theoretically) analyze the efficiency of programs, computing strategies or program transformation techniques, it is convenient to define abstract approaches to cost measurement. In particular, we are specially interested in to contrast the behaviour of computations performed on fuzzy programs obtained via some program transformation techniques developed in our research group (see some examples of fuzzy fold/unfold and partial evaluation in [15], [16], [17], [13], [18], [19], as well as the fuzzy tabulation techniques documented in [20], [21]).

Specially motivated by this goal, we have observed that in declarative programming frameworks it is usual to estimate the computational effort needed to execute a goal in a program by simply counting the number of derivation steps required to reach their solutions. In the context of multi-adjoint logic programming, we have unfortunately discovered that, although this method seems to be acceptable during the operational phase, it becomes inappropriate when considering the interpretive one, since it does not take into account the possible changes in the complexities of the different kinds of fuzzy connectives involved in such computations.

The problem was faced in [22], [23] by proposing more refined (interpretive) cost measures based on "connective weights" which were able of counting the number of primitive operators directly/indirectly appearing in the definition of the connectives evaluated in each (interpretive) step of a given derivation. Anyway, in this paper we are specially concerned with the approach presented in [14], which redefines the notion of interpretive step in order to explicitly expand connective definitions and evaluating primitive (arithmetic) operators on derivation states. The method does not alter the final set of solutions, but it has the extra ability of exhibiting the complexity of the interpretive phase in detail.

The main goal of the present paper is to present our last developments performed on the FLOPER system (see [24], [25] and visit `http://www.dsi.uclm.es/investigacion/dect/FLOPERpage.htm`) which implements several kinds of interpretive steps, including the refined notion of "small interpretive step" we have just commented before [14]. Nowadays, the tool provides facilities for executing as well as for debugging (by generating declarative traces) such kind of fuzzy programs, by means of two main representation (high/low-level, Prolog-based) ways which are somehow antagonistic regarding simplicity and precision features.

The structure of the paper is as follows. In Section 2 we summarize the main features of multi-adjoint logic programming, both language syntax and procedural semantics. Section 3 presents a discussion on cost measures proposed in the past for the considered fuzzy setting, focusing in our approach based on "small interpretive steps" which has been successfully implemented into the FLOPER tool, as described in Section 4 (which contains the main important contribution of this paper). Finally, in Section 5 we give our conclusions and some lines of future work.

2. Multi-Adjoint Logic Programming

This section summarizes the main features of multi-adjoint logic programming (see [10], [11], [12] for a complete formulation of this framework).

We work with a first order language, $\mathcal{L}$, containing variables, constants, function symbols, predicate symbols, and several (arbitrary) connectives to increase language expressiveness: implication connectives ($\leftarrow_1, \leftarrow_2, \ldots$); conjunctive operators (denoted by $\&_1, \&_2, \ldots$), disjunctive operators ($\vee_1, \vee_2, \ldots$), and hybrid operators (usually denoted by $@_1, @_2, \ldots$), all of them are grouped under the name of "aggregators". 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. Although these connectives are binary operators, we usually generalize them as functions with an arbitrary number of arguments. So, we often write $@(x_1, \ldots, x_n)$ instead of $@(x_1, \ldots, @(x_{n-1}, x_n), \ldots)$. By definition, the truth function for an n-ary aggregation operator $[\![@]\!] : L^n \to L$ is required to be 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 which is intended to the evaluation of *modus ponens* [10]. In general, L may be the carrier of any complete bounded lattice but, for readability reasons, in the examples we shall select L as the set of real numbers in the interval $[0, 1]$. A L-expression is a well-formed expression composed by values and connectives of L, as well as variable symbols and *primitive operators* (i.e., arithmetic symbols such as $*, +, min$, etc.). In what follows, we assume that the truth function of any connective $@$ in L is given by its corresponding *connective definition*, that is, an equation of the form $@(x_1, \ldots, x_n) \triangleq E$, where E is a L-expression not containing variable symbols apart from $x_1, \ldots, x_n$.

A *rule* is a formula $H \leftarrow_i \mathcal{B}$, where H 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, conjunctions, disjunctions and aggregations. A *goal* is a body submitted as a query to the system. Roughly speaking, a multi-adjoint logic program is a set of pairs $\langle \mathcal{R}; v\rangle$ (we often write $\mathcal{R}\ with\ v$), where $\mathcal{R}$ is a rule and v is a *truth degree* (a value of L) expressing the confidence of a programmer in the truth of the rule $\mathcal{R}$. By abuse of language, we sometimes refer a tuple $\langle \mathcal{R}; v\rangle$ as a "rule".

The procedural semantics of the multi–adjoint logic language $\mathcal{L}$ can be thought as an operational phase (based on admissible steps) followed by an interpretive one. In the following, $\mathcal{C}[A]$ denotes a formula where A is a subexpression which occurs in the –possibly empty– context $\mathcal{C}[]$. Moreover, $\mathcal{C}[A/A']$ means the replacement of A by A' in context $\mathcal{C}[]$, whereas $\mathcal{V}ar(s)$ refers to the set of distinct variables occurring in the syntactic object s, and $\theta[\mathcal{V}ar(s)]$ denotes the substitution obtained from θ by restricting its domain to $\mathcal{V}ar(s)$.

Definition 2.1 (Admissible Step): 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 $\overset{\text{AS}}{\rightsquigarrow} \subseteq (\mathcal{E} \times \mathcal{E})$ is the smallest relation satisfying the following *admissible rules* (where we always consider that A is the selected atom in $\mathcal{Q}$ and $mgu(E)$ denotes the *most general unifier* of an equation set E):

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

As usual, rules are taken renamed apart. We shall use the symbols $\overset{\text{AS1}}{\leadsto}$, $\overset{\text{AS2}}{\leadsto}$ and $\overset{\text{AS3}}{\leadsto}$ 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 $\overset{\text{AS}}{\leadsto}$ symbol.

Definition 2.2: Let $\mathcal{P}$ be a program, $\mathcal{Q}$ a goal and "id" the empty substitution. An *admissible derivation* is a sequence $\langle \mathcal{Q}; id\rangle \overset{\text{AS}}{\leadsto} \ldots \overset{\text{AS}}{\leadsto} \langle \mathcal{Q}';\theta\rangle$. When $\mathcal{Q}'$ is a formula not containing atoms (i.e., a L-expression), 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 2.3: Let $\mathcal{P}$ be the following fuzzy program:

$$\begin{array}{llll}
\mathcal{R}_1: & p(X) & \leftarrow_{\mathtt{P}} \&_{\mathtt{G}}(\vee_{\mathtt{L}}(q(X), 0.6), r(X)) & \text{with } 0.9\\
\mathcal{R}_2: & q(a) & \leftarrow & \text{with } 0.8\\
\mathcal{R}_3: & r(X) & \leftarrow & \text{with } 0.7
\end{array}$$

where the labels L, G and P mean respectively for *Łukasiewicz logic*, *Gödel intuitionistic logic* and *product logic*, that is, $\vee_{\mathtt{L}}(x_1,x_2) \triangleq min(1, x_1+x_2)$, $\&_{\mathtt{G}}(x_1,x_2) \triangleq min(x_1,x_2)$ and $\&_{\mathtt{P}}(x_1,x_2) \triangleq x_1 * x_2$.

Now, we can generate the following admissible derivation (we underline the selected atom in each step):

$$\begin{array}{ll}
\langle \underline{p(X)};\ id\rangle & \overset{\text{AS1}}{\leadsto} \mathcal{R}_1\\
\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\vee_{\mathtt{L}}(\underline{q(X1)}, 0.6), r(X1))); \{X/X_1\}\rangle & \overset{\text{AS2}}{\leadsto} \mathcal{R}_2\\
\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\vee_{\mathtt{L}}(0.8, 0.6), \underline{r(a)})); \{X/a, X_1/a\}\rangle & \overset{\text{AS2}}{\leadsto} \mathcal{R}_3\\
\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\vee_{\mathtt{L}}(0.8, 0.6), \overline{0.7})); \{X/a, X_1/a, X_2/a\}\rangle &
\end{array}$$

So, the admissible computed answer (a.c.a.) in this case is the pair: $\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\vee_{\mathtt{L}}(0.8, 0.6), 0.7)); \theta\rangle$, where $\theta = \{X/a, X_1/a, X_2/a\}[\mathcal{V}ar(p(X))] = \{X/a\}$.
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 (a L-expression) which can be then directly interpreted w.r.t. lattice L by applying the following definition we initially presented in [13]:

Definition 2.4 (Interpretive Step): Let $\mathcal{P}$ be a program, $\mathcal{Q}$ a goal and σ a substitution. Assume that $[\![@]\!]$ is the truth function of connective $@$ in the lattice $\langle L, \preceq\rangle$ associated to $\mathcal{P}$, such that, for values $r_1, \ldots, r_n, r_{n+1} \in L$, we have that $[\![@]\!](r_1,\ldots,r_n) = r_{n+1}$. Then, we formalize the notion of *interpretive computation* as a state transition system, whose transition relation $\overset{\text{IS}}{\leadsto} \subseteq (\mathcal{E} \times \mathcal{E})$ is defined as the least one satisfying: $\langle \mathcal{Q}[@(r_1,\ldots,r_n)];\sigma\rangle \overset{\text{IS}}{\leadsto} \langle \mathcal{Q}[@(r_1,\ldots,r_n)/r_{n+1}];\sigma\rangle$.

Definition 2.5: Let $\mathcal{P}$ be a program and $\langle \mathcal{Q};\sigma\rangle$ an a.c.a., that is, $\mathcal{Q}$ is a goal not containing atoms (i.e., a L-expression). An *interpretive derivation* is a sequence $\langle \mathcal{Q};\sigma\rangle \overset{\text{IS}}{\leadsto} \ldots \overset{\text{IS}}{\leadsto} \langle \mathcal{Q}';\sigma\rangle$. When $\mathcal{Q}' = r \in L$, 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 2.6: If we complete the previous derivation of Example 2.3 by applying 3 interpretive steps in order to obtain the final f.c.a. $\langle 0.63; \{X/a\}\rangle$, we generate the following interpretive derivation D_1:

$$\begin{array}{ll}
\langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\underline{\vee_{\mathtt{L}}(0.8, 0.6)}, 0.7)); \theta\rangle & \overset{\text{IS}}{\leadsto}\\
\langle \&_{\mathtt{P}}(0.9, \underline{\&_{\mathtt{G}}(1, 0.7)}); \theta\rangle & \overset{\text{IS}}{\leadsto}\\
\langle \underline{\&_{\mathtt{P}}(0.9, 0, 7)}; \theta\rangle & \overset{\text{IS}}{\leadsto}\\
\langle 0.63; \theta\rangle. &
\end{array}$$

3. Interpretive Steps and Cost Measures

A classical, simple way for estimating the computational cost required to built a derivation, consists in counting the number of computational steps performed on it. So, given a derivation D, we define its:

- *operational cost*, $\mathcal{O}_c(D)$, as the number of admissible steps performed in D.
- *interpretive cost*, $\mathcal{I}_c(D)$, as the number of interpretive steps done in D.

Note that the operational and interpretive costs of derivation D_1 performed in the previous section are $\mathcal{O}_c(D_1) = 3$ and $\mathcal{I}_c(D_1) = 3$, respectively. Intuitively, $\mathcal{O}_c$ informs us about the number of atoms exploited along a derivation. Similarly, $\mathcal{I}_c$ seems to estimate the number of connectives evaluated in a derivation. However, this last statement is not completely true: $\mathcal{I}_c$ only takes into account those connectives appearing in the bodies of program rules which are replicated on states of the derivation, but no those connectives recursively *nested* in the definition of other connectives. The following example highlights this fact.

Example 3.1: A simplified version of rule $\mathcal{R}_1$, whose body only contains an aggregator symbol is: $\mathcal{R}_1^* : p(X) \leftarrow_{\mathtt{P}} @_1(q(X), r(X))$ with 0.9 where $@_1(x_1,x_2) \triangleq \&_{\mathtt{G}}(\vee_{\mathtt{L}}(x_1, 0.6), x_2)$. Note that $\mathcal{R}_1^*$ has exactly the same meaning (interpretation) than $\mathcal{R}_1$, although different syntax. In fact, both of them have the same sequence of atoms in their head and bodies. The differences are regarding the set of connectives which explicitly appear in their bodies since in $\mathcal{R}_1^*$ we have moved $\&_{\mathtt{G}}$ and $\vee_{\mathtt{L}}$ (as well as value 0.6) from the body of the rule (see $\mathcal{R}_1$) to the connective definition of $@_1$. Now, we use rule $\mathcal{R}_1^*$ instead of $\mathcal{R}_1$ for generating the following derivation D_1^* which returns exactly the same f.c.a than D_1:

$$\begin{array}{ll}
\langle \underline{p(X)};\ id\rangle & \overset{\text{AS1}}{\leadsto} \mathcal{R}_1^*\\
\langle \&_{\mathtt{P}}(0.9, @_1(\underline{q(X1)}, r(X1))); \{X/X_1\}\rangle & \overset{\text{AS2}}{\leadsto} \mathcal{R}_2\\
\langle \&_{\mathtt{P}}(0.9, @_1(0.8, \underline{r(a)})); \{X/a, X_1/a\}\rangle & \overset{\text{AS2}}{\leadsto} \mathcal{R}_3\\
\langle \&_{\mathtt{P}}(0.9, \underline{@_1(0.8, 0.7)}); \{X/a, X_1/a, X_2/a\}\rangle & \overset{\text{IS}}{\leadsto}\\
\langle \underline{\&_{\mathtt{P}}(0.9, 0.7)}; \{X/a, X_1/a, X_2/a\}\rangle & \overset{\text{IS}}{\leadsto}\\
\langle 0.63; \{X/a, X_1/a, X_2/a\}\rangle &
\end{array}$$

Note that, since we have exploited the same atoms with the same rules (except for the first steps performed with rules $\mathcal{R}_1$ and $\mathcal{R}_1^*$, respectively) in both derivations, then $\mathcal{O}_c(D_1) = \mathcal{O}_c(D_1^*) = 3$. However, although connectives $\&_G$ and $\vee_L$ have been evaluated in both derivations, in D_1^* such evaluations have not been explicitly counted as interpretive steps, and consequently they have not been added to increase the interpretive cost measure $\mathcal{I}_c$. This unrealistic situation is reflected by the abnormal result $\mathcal{I}_c(D_1) = 3 > 2 = \mathcal{I}_c(D_1^*)$: as we will see later, it is important to note that $\mathcal{R}_1^*$ must not be considered an optimized version of $\mathcal{R}_1$, even when the wrong measure $\mathcal{I}_c$ seems to indicate the contrary.

This problem was initially pointed out in [22], where a preliminary solution was proposed by assigning weights to connectives in concordance with the set of primitive operators involved in the definition of the proper connective @ as well as those ones recursively contained in the definitions of connectives invoked from @. Moreover, in [23] we improved the previous notion of "connective weight" by also taken into account the number of recursive calls to fuzzy connectives (directly or indirectly) performed in the definition of @.

A rather different way for facing the same problem is presented in [14], where instead on connective weights, we opt for the more "visual" method we have just implemented into FLOPER, based on the subsequent re-definition of the behaviour of the interpretive phase.

Definition 3.2 (Small Interpretive Step): Let $\mathcal{P}$ be a program, $\mathcal{Q}$ a goal and σ a substitution. Assume that the (non interpreted yet) L-expression $\Omega(r_1, \ldots, r_n)$ occurs in $\mathcal{Q}$, where Ω is just a primitive operator or a connective defined in the lattice $\langle L, \preceq \rangle$ associated to $\mathcal{P}$, and $r_1, \ldots, r_n$ are elements of L. We formalize the notion of *small interpretive computation* as a state transition system, whose transition relation $\overset{\text{SIS}}{\rightsquigarrow} \subseteq (\mathcal{E} \times \mathcal{E})$ is the smallest relation satisfying the following *small interpretive rules* (where we always consider that $\Omega(r_1, \ldots, r_n)$ is the selected L-expression in $\mathcal{Q}$):

1) $\langle Q[\Omega(r_1, \ldots, r_n)]; \sigma \rangle \overset{\text{SIS}}{\rightsquigarrow} \langle Q[\Omega(r_1, \ldots, r_n)/E']; \sigma \rangle$, if Ω is a connective defined as $\Omega(x_1, \ldots, x_n) \triangleq E$ and E' is obtained from the L-expression E by replacing each variable (formal parameter) x_i by its corresponding value (actual parameter) r_i, $1 \leq i \leq n$, that is, $E' = E[x_1/r_1, \ldots, x_n/r_n]$.
2) $\langle Q[\Omega(r_1, \ldots, r_n)]; \sigma \rangle \overset{\text{SIS}}{\rightsquigarrow} \langle Q[\Omega(r_1, \ldots, r_n)/r]; \sigma \rangle$, if Ω is a primitive operator such that, once evaluated with parameters $r_1, \ldots, r_n$, produces the result r.

From now, we shall use the symbols $\overset{\text{SIS1}}{\rightsquigarrow}$ and $\overset{\text{SIS2}}{\rightsquigarrow}$ to distinguish between computation steps performed by applying one of the specific "small interpretive" rules. Moreover, when we use the expression *interpretive derivation*, we refer to a sequence of *small interpretive steps* (according to the previous definition) instead of a sequence of *interpretive steps* (regarding Definition 2.4). Note that this fact, supposes too a slight revision of Definition 2.5 which does not affect the essence of the notion of fuzzy computed answer: the repeated application of both kinds of interpretive steps on a given state only affects to the length of the corresponding derivations, but both ones lead to the same final states (containing the corresponding fuzzy computed answers).

Example 3.3: Recalling again the a.c.a. obtained in Example 2.3, we can reach the final fuzzy computed answer $\langle 0.63; \{X/a\} \rangle$ (achieved in Example 2.6 by means of interpretive steps) by generating now the following interpretive derivation D_2 based on "small interpretive steps" (Figure 1):

$$\begin{array}{ll}
\langle \&_{\texttt{P}}(0.9, \&_{\texttt{G}}(\underline{\vee_{\texttt{L}}(0.8, 0.6)}, 0.7)); \{X/a\} \rangle & \overset{\text{SIS1}}{\rightsquigarrow} \\
\langle \&_{\texttt{P}}(0.9, \&_{\texttt{G}}(min(1, \underline{0.8 + 0.6}), 0.7)); \{X/a\} \rangle & \overset{\text{SIS2}}{\rightsquigarrow} \\
\langle \&_{\texttt{P}}(0.9, \&_{\texttt{G}}(\underline{min(1, 1.4)}, 0.7)); \{X/a\} \rangle & \overset{\text{SIS2}}{\rightsquigarrow} \\
\langle \&_{\texttt{P}}(0.9, \underline{\&_{\texttt{G}}(1, 0.7)}); \{X/a\} \rangle & \overset{\text{SIS1}}{\rightsquigarrow} \\
\langle \&_{\texttt{P}}(0.9, \underline{min(1, 0.7)}); \{X/a\} \rangle & \overset{\text{SIS2}}{\rightsquigarrow} \\
\langle \underline{\&_{\texttt{P}}(0.9, 0.7)}; \{X/a\} \rangle & \overset{\text{SIS1}}{\rightsquigarrow} \\
\langle \underline{0.9 * 0.7}; \{X/a\} \rangle & \overset{\text{SIS2}}{\rightsquigarrow} \\
\langle 0.63; \{X/a\} \rangle &
\end{array}$$

Going back now to Example 3.1, we can rebuild the interpretive phase of Derivation D_1^* in terms of small interpretive steps, thus generating the following interpretive derivation D_2^*. Firstly, by applying a $\overset{\text{SIS1}}{\rightsquigarrow}$ step on the L-expression $\&_{\texttt{P}}(0.9, \underline{@_1(0.8, 0.7)})$, it becomes $\&_{\texttt{P}}(0.9, \&_{\texttt{G}}(\vee_{\texttt{L}}(0.8, 0.6), 0.7))$, and from here, the interpretive derivation evolves exactly in the same way as derivation D_2 we have just done above.

At this moment, it is mandatory to meditate on cost measures regarding derivations D_1, D_1^*, D_2 and D_2^*. First of all, note that the operational cost $\mathcal{O}_c$ of all them coincides, which is quite natural. However, whereas $\mathcal{I}_c(D_1) = 3 > 2 = \mathcal{I}_c(D_1^*)$, we have now that $\mathcal{I}_c(D_2) = 7 < 8 = \mathcal{I}_c(D_2^*)$. This apparent contradiction might confuse us when trying to decide which program rule ($\mathcal{R}_1$ or $\mathcal{R}_1^*$) is "better". The use of Definition 3.2 in derivations D_2 and D_2^* is the key point to solve our problem, as we are going to see. In Example 3.1 we justified that by simply counting the number of interpretive steps performed in Definition 2.4 might produce abnormal results, since the evaluation of connectives with different complexities were (wrongly) measured with the same computational cost. Fortunately, the notion of small interpretive step makes visible in the proper derivation all the connectives and primitive operators appearing in the (possibly recursively nested) definitions of any connective appearing in any derivation state. As we have seen, in D_2 we have expanded in three $\overset{\text{SIS1}}{\rightsquigarrow}$ steps the definitions of three connectives, i.e. $\vee_{\texttt{L}}, \&_{\texttt{G}}$ and $\&_{\texttt{P}}$, and we have applied four $\overset{\text{SIS2}}{\rightsquigarrow}$ steps to solve four primitive operators, that is, $+$, min (twice) and $*$. The same computational effort as been performed in D_2^*, but also one more $\overset{\text{SIS1}}{\rightsquigarrow}$ step was applied to accomplish with the expansion of the extra connective $@_1$. This justifies why $\mathcal{I}_c(D_2) = 7 < 8 = \mathcal{I}_c(D_2^*)$

and contradicts the wrong measures of Example 3.1: the interpretive effort developed in derivations D_1 and D_2 (both using the program rule $\mathcal{R}_1$), is slightly lower than the one performed in derivations D_1^* and D_2^* (which used rule $\mathcal{R}_1^*$), and not the contrary.

The accuracy of our new way for measuring and performing interpretive computations seems to be crucial when comparing the execution behaviour of programs obtained by transformation techniques such as the fold/unfold framework we describe in [15], [17]. In this sense, instead of measuring the absolute cost of derivations performed in a program, we are more interested in the relative gains/lost of efficiency produced on transformed programs. For instance, by applying the so-called "aggregation operation" described in [17] we can transform rule $\mathcal{R}_1$ into $\mathcal{R}_1^*$ and, in order to proceed with alternative transformations (fold,unfold, et...) if the resulting program degenerates w.r.t. the original one (as occurs in this case), we need an appropriate cost measure as the one proposed here to help us for taken decisions. This fact has capital importance for discovering drastic situations which can appear in degenerated transformation sequences such as the generation of highly nested definitions of aggregators. For instance, assume the following sequence of connective definitions: $@_{100}(x_1,x_2) \triangleq @_{99}(x_1,x_2)$, $@_{99}(x_1,x_2) \triangleq @_{98}(x_1,x_2),\ldots,$ and finally $@_1(x_1,x_2) \triangleq x_1 * x_2$. When trying to solve two expression of the form $@_{99}(0.9,0.8)$ and $@_1(0.9,0.8)$, cost measures based on number of interpretive steps ([13]) and weights of interpretive steps ([22]) would assign 1 unit of interpretive cost to both derivations. Fortunately, our new approach is able to clearly distinguish between both cases, since the number of $\overset{\text{SIS1}}{\leadsto}$ steps performed in each one is rather different (100 and 1, respectively).

4. The FLOPER System in Action

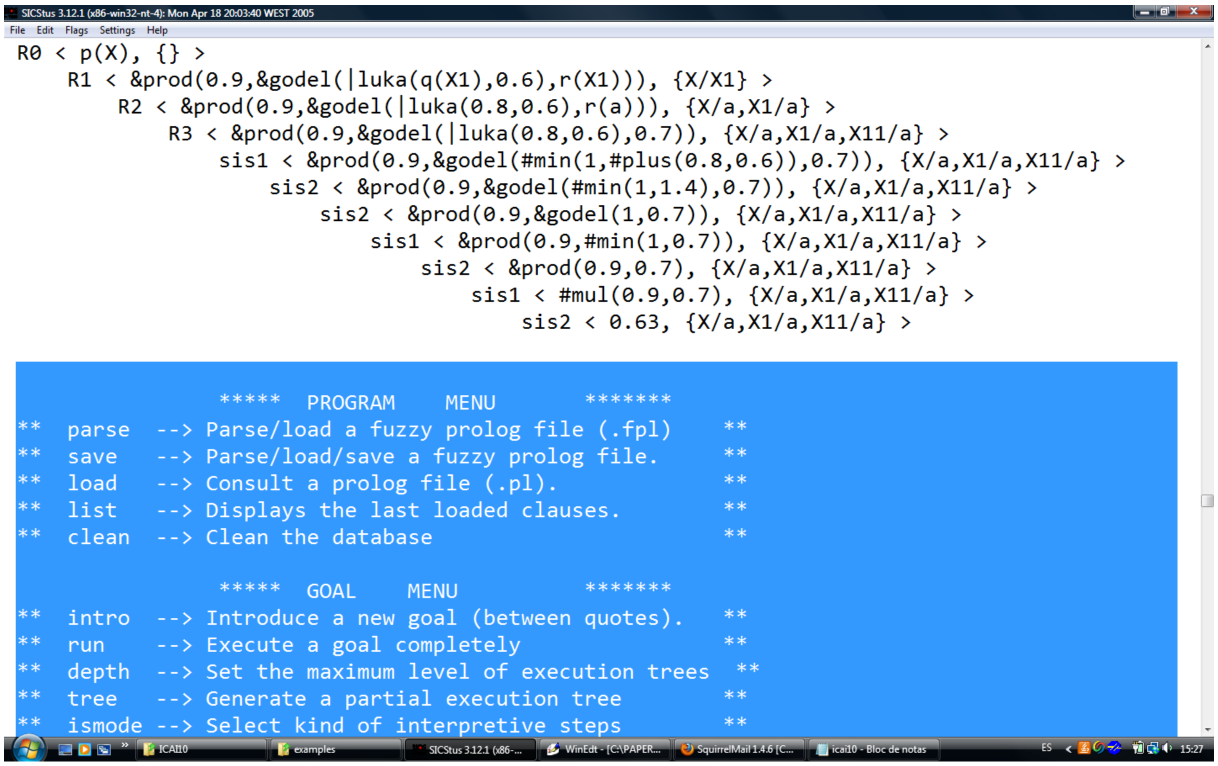
As detailed in [24], [25], our parser has been implemented by using the classical DCG's (*Definite Clause Grammars*) resource of the Prolog language, since it is a convenient notation for expressing grammar rules. Once the application is loaded inside a Prolog interpreter (in our case, Sicstus Prolog v.3.12.5), it shows a menu which includes options for loading, parsing, listing and saving fuzzy programs, as well as for executing fuzzy goals (see Figure 1). All these actions are based in the translation of the fuzzy code into standard Prolog code. The key point is to extend each atom with an extra argument, called *truth variable* of the form `_TVi`, which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. For instance, the first clause in our target program is translated into: "`p(X, TV0) :- q(X,_TV1), or_luka(_TV1,0.6,_TV2), r(X,_TV3), and_godel(_TV2,_TV3,_TV4), and_prod(0.9,_TV4,TV0)`", where some definitions of "aggregator predicates" are: "`and_godel(X,Y,Z) :- (X =< Y,Z = X;X > Y,Z = Y)`" and "`and_prod(X,Y,Z) :- Z is X * Y`". Moreover, the second clause in our target program, becomes in the pure Prolog fact "`q(a,0.8)`" while a fuzzy goal like "`p(X) &godel r(a)`", is translated into the Prolog goal: "`p(X,_TV1), r(a,_TV2), and_godel(_TV1,_TV2,Truth_degree)`" (note that the last truth degree variable is not anonymous now) for which the Prolog interpreter returns the two desired fuzzy computed answer `[Truth_degree = 0.63, X = a]`.

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, the previous 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 (i.e., contraction and expansion of sub-expressions of a program using the definitions of this program or of a preceding one, thus generating more efficient code) or partial evaluation we have described in [17], [15], [19]. For instance, our fuzzy unfolding transformation is defined as the replacement of a program rule $\mathcal{R} : \langle A \leftarrow_i \mathcal{B} \text{ with } v\rangle$ by the set of rules $\{A\sigma \leftarrow_i \mathcal{B}' \text{ with } v \mid \langle \mathcal{B}; id\rangle \rightarrow_{AS} \langle \mathcal{B}'; \sigma\rangle\}$, 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.

In [25], we conceived a new low-level representation for the fuzzy code which currently offers the possibility of performing debugging actions such as tracing a FLOPER work session. For instance, after parsing the first rule of our program, we obtain the following expression (which is "asserted" into the interpeter's database as a Prolog fact):

```
rule(1, head(atom(pred(p,1),[var('X')])),
        implication(prod),
        body(and(godel,2,
               [or(luka,2,
                  [ atom(pred(q,1),[var('X')]),
                    truth_degree(0.6)]),
                atom(pred(r,1),[var('X')])])),
        truth_degree(0.9)).
```

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


```
R0 < p(X), {} >
    R1 < &prod(0.9,&godel(|luka(q(X1),0.6),r(X1))), {X/X1} >
        R2 < &prod(0.9,&godel(|luka(0.8,0.6),r(a))), {X/a,X1/a} >
            R3 < &prod(0.9,&godel(|luka(0.8,0.6),0.7)), {X/a,X1/a,X11/a} >
                sis1 < &prod(0.9,&godel(#min(1,#plus(0.8,0.6)),0.7)), {X/a,X1/a,X11/a} >
                    sis2 < &prod(0.9,&godel(#min(1,1.4),0.7)), {X/a,X1/a,X11/a} >
                        sis2 < &prod(0.9,&godel(1,0.7)), {X/a,X1/a,X11/a} >
                            sis1 < &prod(0.9,#min(1,0.7)), {X/a,X1/a,X11/a} >
                                sis2 < &prod(0.9,0.7), {X/a,X1/a,X11/a} >
                                    sis1 < #mul(0.9,0.7), {X/a,X1/a,X11/a} >
                                        sis2 < 0.63, {X/a,X1/a,X11/a} >


                ***** PROGRAM   MENU       *******
**   parse  --> Parse/load a fuzzy prolog file (.fpl)    **
**   save   --> Parse/load/save a fuzzy prolog file.     **
**   load   --> Consult a prolog file (.pl).             **
**   list   --> Displays the last loaded clauses.        **
**   clean  --> Clean the database                       **

                ***** GOAL   MENU          *******
**   intro  --> Introduce a new goal (between quotes).   **
**   run    --> Execute a goal completely                **
**   depth  --> Set the maximum level of execution trees  **
**   tree   --> Generate a partial execution tree        **
**   ismode --> Select kind of interpretive steps        **
```

Fig. 1
EXAMPLE OF A WORK SESSION WITH FLOPER SHOWING "SMALL INTERPRETIVE STEPS" AND THE PROGRAM/GOAL MENUS

substitutions, apply admisible/interpretive steps, etc.

As showed in the down-middle, dark part of Figure 1, FLOPER is equipped with two options, called `"tree"` and `"depth"`, for tracing execution trees and fixing the maximum length allowed for their branches (initially 3), respectively. Working with these options is crucial when the `"run"` choice fails: remember that this last option is based on the generation of pure logic SLD-derivations which might fall in loop or directly fail in some cases as the experiments of [25] show, in contrast with the traces (based on finite, non-failed, admissible derivations) that the `"tree"` option displays.

Regarding the last option "`ismode`" showed at the bottom of Figure 1, it is important to remark that it represents our last record achieved in the development of the FLOPER tool. When the user selects such choice, he/she can decide among three levels of detail when visualizing the interpretive phase performed during the generation of "unfolding trees" (the system displays states on different lines, appropriately indented to distinguish the proper relationship -parent/child/grandchild...- among nodes):

- `Large:` means to obtain the final result in one go.
- `Medium:` implements the notion of "interpretive step" according Definition 2.4 [13].
- `Small:` performs "small interpretive steps" on derivations following Definition 3.2 [14].

The reader can observe at the beginning of Figure 1, the aspect offered by FLOPER when visualizing in detail the behaviour of our running example when choosing the last option we have just commented before. It is easy to see the correspondences with Examples 2.3 (note that, although program rules applied on admissible steps always precede the corresponding state, FLOPER labels the root goal with the "virtual" program rule `R0`) and 3.3 (advise that each primitive operators is always labeled with the mark #).

5. Conclusions and Future Work

Declarative programming languages are extensively used in solving AI problems. In recent years, we have observed a growing interest for including expressive resources based on fuzzy logic in order to increase the impact of both research communities. Encouraged by the experience acquired in our research group regarding the design of techniques, tools and applications related with the so-called multi-adjoint logic approach ([17], [15], [13], [22], [19], [20], [21], [25], [14], [23]), in this paper we were concerned with the design and real implementation of good cost measures for computations performed in such flexible fuzzy logic programming framework.

We have highlighted the fact that the usual method of counting the number of computational steps performed in a derivation, might produce wrong results when estimating the computational effort developed in the interpretive phase of a multi-adjoint derivation. The problem emerges when considering connectives on the body of program rules whose definitions also invoke other connectives. Although in [22], [23] two solutions were proposed under the point of view of "connective weights", in this paper we have been concerned with the more visual answer given in [14] (based on a low level re-definition of the notion of interpretive step) which has been finally implemented in our experimental FLOPER prototype.

In the near future, we plan to exploit in practice its accuracy when proving the efficiency of the fuzzy fold/unfold, partial evaluation, reductant calculus and tresholded tabulation techniques developed in our research group ([17], [13], [18], [19], [21], [26], [27]).

Acknowledgements

This work has been partially supported by the EU, under FEDER, and the Spanish Science and Innovation Ministry (MICIN) under grant TIN 2007-65749 and by the Castilla-La Mancha Administration under grant PII1I09-0117-4481.

References

[1] J. Lloyd, *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987, second edition.

[2] I. Bratko, *Prolog Programming for Artificial Intelligence*. Addison Wesley, September 2000.

[3] J. Lloyd, "Declarative programming for artificial intelligence applications," *SIGPLAN Notices*, vol. 42, no. 9, pp. 123–124, 2007.

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

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

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

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

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

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

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

[11] ——, "Multi-adjoint logic programming with continuous semantics," *Proc. of Logic Programming and Non-Monotonic Reasoning, LPNMR'01, Springer-Verlag, LNAI*, vol. 2173, pp. 351–364, 2001.

[12] ——, "A procedural semantics for multi-adjoint logic programing," *Progress in Artificial Intelligence, EPIA'01, Springer-Verlag, LNAI*, vol. 2258, no. 1, pp. 290–297, 2001.

[13] P. Julián, G. Moreno, and J. Penabad, "Operational/Interpretive Unfolding of Multi-adjoint Logic Programs," *Journal of Universal Computer Science*, vol. 12, no. 11, pp. 1679–1699, 2006.

[14] P. Morcillo and G. Moreno, "Modeling interpretive steps in fuzzy logic computations," in *Proc. of the 8th International Workshop on Fuzzy Logic and Applications, WILF'2009. Palermo, Italy, June 9-12*, V. D. Ges, S. Pal, and A. Petrosino, Eds. Springer Verlag, LNAI 5571, 2009, pp. 44–51.

[15] P. Julián, G. Moreno, and J. Penabad, "On Fuzzy Unfolding. A Multi-adjoint Approach," *Fuzzy Sets and Systems*, vol. 154, pp. 16–33, 2005.

[16] G. Moreno, "Building a Fuzzy Transformation System," in *Proc. of the 32nd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM'2006. Merin, Czech Republic, January 21-27*, J. Wiedermann, G. Tel, J. Pokorn, M. Bielikov, and J. Stuller, Eds. Springer Verlag, LNCS 3831, 2006, pp. 409–418.

[17] J. Guerrero and G. Moreno, "Optimizing fuzzy logic programs by unfolding, aggregation and folding," *Electronic Notes in Theoretical Computer Science*, vol. 219, pp. 19–34, 2008.

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

[19] ——, "An Improved Reductant Calculus using Fuzzy Partial Evaluation Techniques," *Fuzzy Sets and Systems*, vol. 160, pp. 162–181, 2009, doi: 10.1016/j.fss.2008.05.006.

[20] P. Julián, J. Medina, G. Moreno, and M. Ojeda, "Thresholded tabulation in a fuzzy logic setting," *Electronic Notes in Theoretical Computer Science*, vol. 248, pp. 115–130, 2009.

[21] ——, "Efficient thresholded tabulation for fuzzy query answering," *Studies in Fuzziness and Soft Computing (Foundations of Reasoning under Uncertainty)*, vol. 249, pp. 125–141, 2010.

[22] P. Julián, G. Moreno, and J. Penabad, "Measuring the interpretive cost in fuzzy logic computations," in *Proc. of Applications of Fuzzy Sets Theory, 7th International Workshop on Fuzzy Logic and Applications, WILF'2007, Camogli, Italy, July 7-10*, F. Masulli, S. Mitra, and G. Pasi, Eds. Springer Verlag, LNAI 4578, 2007, pp. 28–36.

[23] P. J. Morcillo and G. Moreno, "On cost estimations for executing fuzzy logic programs," in *Proceedings of the 2009 International Conference on Artificial Intelligence, ICAI'2009, July 13-16, 2009, Las Vegas, Nevada, USA*, H. R. Arabnia, D. de la Fuente, and J. A. Olivas, Eds. CSREA Press, 2009, pp. 217–223.

[24] J. Abietar, P. Morcillo, and G. Moreno, "Designing a software tool for fuzzy logic programming," in *Proc. of the International Conference of Computational Methods in Sciences and Engineering ICCMSE'2007, Volume 2 (Computation in Modern Science and Engineering)*, T. Simos and G. Maroulis, Eds. American Institute of Physics (distributed by Springer), 2007, pp. 1117–1120.

[25] P. Morcillo and G. Moreno, "Programming with Fuzzy Logic Rules by using the FLOPER Tool," in *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML'2008, Orlando, FL, USA, October 30-31*, N. Bassiliades, G. Governatori, and A. Paschke, Eds. Springer Verlag, LNCS 3521, 2008, pp. 119–126.

[26] ——, "Unfolding connective definitions in multi-adjoint logic programs," in *Proc. of the First Workshop on Computational Logics and Artificial Intelligence, CLAI'2009 (integrated in CAEPIA'2009), Sevilla, Spain, November 9*, F. J. M. Mateos, Ed. Universidad de Sevilla, 2009, pp. 59–70.

[27] ——, "A practical approach for ensuring completeness of multi-adjoint logic computations via general reductants," in *Proc. of IX Jornadas sobre Programación y Lenguajes, PROLE'2009, San Sebastián, Spain, September 8-11*, G. M. P. Lucio and R. Peña, Eds. Universidad del País Vasco, 2009, pp. 355–363, (ISBN 978-84-692-4600-9).