Unfolding Fuzzy Logic Programs

Pascual Julián
Department of Computer Science
ESICR, Univ. of Castilla-La Mancha
13071 Ciudad Real, Spain
Email: Pascual.Julian@uclm.es

Ginés Moreno
Department of Computer Science
EPSA, Univ. of Castilla-La Mancha
02071 Albacete, Spain
Email: Gines.Moreno@uclm.es

Jaime Penabad
Department of Mathematics
EPSA, Univ. of Castilla-La Mancha
02071 Albacete, Spain
Email: Jaime.Penabad@uclm.es

Abstract—**Unfolding is a semantics-preserving program transformation technique that consists in the expansion of subexpressions of a program using their own definitions. The unfolding transformation is able to improve programs, generating more efficient code. Unfolding is the basis for developing sophisticated and powerful programming tools, such as fold/unfold transformation systems or partial evaluators. In this paper we address the problem of extending the classical definition of the unfolding rule (for pure logic programs) to a fuzzy logic setting. We use a fuzzy variant of Prolog [1] where a *fuzzy computed answer* is a pair ⟨*truth degree*; *substitution*⟩ computed by Fuzzy SLD-Resolution. We adapt the concept of a *computation rule*, a mapping that selects the subexpression of a goal involved in a computation step, and we prove the independence of the computation rule. Moreover, we define a fuzzy unfolding rule and we demonstrate its strong correctness properties, that is, the original and the unfolded program compute the same fuzzy computed answers. Finally, we discuss how to improve the expressive power (of the fuzzy component) of our language by introducing a more general labeled mark language that the one described in [1].**

I. Introduction

Logic Programming [2] has been widely used for problem solving and knowledge representation in the past. Nevertheless, traditional logic programming languages do not incorporate techniques or constructs in order to treat explicitly uncertainty and approximated reasoning.

Fuzzy Logic provides a mathematical background for modeling uncertainty and/or vagueness. Fuzzy logic relays on the concept of fuzzy set, the theory of fuzzy connectives (t-norms, t-conorms, etc.) and the extension of two-values classical predicate logic to a logic where formulas can be evaluated in the range of the $[0, 1]$ real interval (see [3] or [4] for a comprehensive introduction of this subject). Fuzzy sets [5] are objects introduced to deal with the fuzziness or vagueness we find in the real world when we try to describe phenomena that have not sharply defined boundaries. Given a set U, an ordinary subset A of U can be defined in terms of its *characteristic function* $\chi_A(x)$ which neatly specifies whether or not an element x is in A, (i.e., returns 1 if $x \in A$, or 0 otherwise). On the other hand, a *fuzzy subset* A of U is a function $A : U \rightarrow [0, 1]$. The function A is called the *membership function*, and the value $A(x)$ represents the *degree of membership*[1] of x in the fuzzy set A. Different functions A can be considered for a fuzzy concept and, in general, they will present a soft shape instead of the characteristic function's crisp slope of an ordinary set.

Fuzzy Logic Programming is an interesting and still growing research area that agglutinates the efforts to introduce Fuzzy Logic into Logic Programming. 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 which is able to handle partial truth and to reason with uncertainty. Most of these systems implement the fuzzy resolution principle introduced by Lee in [6], such as the Prolog-Elf system [7], Fril Prolog system [8] and the F-Prolog language [9].

On the other hand, there is also no agreement about which fuzzy logic must be used when fuzzifying Prolog. Most systems use min-max logic (for modeling the conjunction and disjunction operations) but other systems just use Lukasiewicz logic [10]. Other approaches are parametric with respect the interpretation of the fuzzy connectives, letting them unspecified to obtain a more general framework [1]. Recently, it has been appeared in [11] a theoretical model for fuzzy logic programming which deals with many values implications. Finally, in [12] we find an extremely flexible scheme where, apart from introducing negation and dealing with interval-valued fuzzy sets [13], each clause on a given program may be interpreted with a different logic. At the end of this paper, we show that this last extension can be partially simulated in our setting in a very natural way.

Program transformation is an optimization technique for computer programs that starting with an initial program $\mathcal{P}_0$ derives a sequence $\mathcal{P}_1, \ldots, \mathcal{P}_n$ of transformed programs by applying *elementary transformation rules*. The aim is that the final program $\mathcal{P}_n$ have the same meaning as $\mathcal{P}_0$, but with a more efficient behaviour with regard some criterion. Program transformation can be seen as a methodology for software development, hence its importance.

Among the elementary transformation rules the so called *unfolding* rule has been widely studied. In essence, an unfolding rule is a program transformation operation which replaces a program rule by the set of rules obtained after applying a *symbolic computation* step (in all its possible forms) on the body of the selected rule [14]. Depending on the concrete paradigm taken into account (functional [15], logic [16] or integrated functional–logic [17]) the considered computation step will be performed using —some variant of—

[1] It is not meant to convey the likelihood that x has some particular attribute such as "young" [4].

its associated operational mechanism (rewriting, resolution or narrowing, respectively). The unfolding rule is able to produce, by itself (i.e., without being combined with any other kind of transformation), important optimizations on the original program code. Beyond this initial benefit, different unfolding formulations have shown their usefulness in the construction of advanced techniques for program synthesis, program analysis, debugging, compiling, learning, and so on. But, perhaps, the fields where unfolding has exhibited its best properties and powerful capabilities were partial evaluation and fold/unfold-based program transformation [18].

Although one of the main goals of a transformation technique is to obtain a better behaviour of the transformed program with respect to some efficiency criterion (for instance, execution time), from the theoretical point of view, the main goal is to achieve the semantics correctness of the transformation. A proper formulation of the unfolding rule must guarantee that it is semantics preserving (i.e., the unfolded program must reproduce as closely as possible the "observable" effects of the original program). For this purpose, several "applicability" conditions must be identified without drastically reducing the class of programs where the transformation could be performed in a safe way. Language syntax and operational semantics play an important role in the identification of such requirements, as we will see in the extension of the unfolding transformation (for pure logic programs) to a fuzzy context that we develop in this paper.

We have considered different fuzzy Prolog dialects and, finally, we have selected the one described in [1], that we call f-Prolog, since we find that its sintax and operational semantics are appropriated for the formalization of the unfolding transformation. In this language a *fuzzy computed answer* is a pair $\langle$*truth degree*; *substitution*$\rangle$ computed by Fuzzy SLD-Resolution. We adapt the concept of a *computation rule*, a mapping that selects the subexpression of a goal involved in a computation step, and we prove a result which is the fuzzy countepart of the independence of the computation rule theorem demonstrated in [2].

We have defined a fuzzy unfolding rule for a labeled mark variant of f-Prolog and, we have studied its correctness properties. The major technical result consists of proving the strong soundness and completeness for the new unfolding rule, namely that the fuzzy answers computed by Fuzzy SLD-Resolution in the initial and the final program coincide. Also, we discuss how to improve the expressive power (of the fuzzy component) of our language by introducing a more general labeled mark language that the one described in [1].

The outline of this paper is as follows. In the next section, we recall the most important features of f-Prolog. In Section III we present the operational semantics of our language. We also introduce an extension of the computation rule an state its independence property. In Section IV we define the fuzzy unfolding rule and we present its strong correctness. Finally, before concluding in Section VI, we discuss in Section V how to extend our fuzzy language while preserving the properties of our transformation. Missing proofs can be found in [19].

II. Fuzzy Prolog Programs

There is no common method for introducing fuzzy concepts into logic programming. We have found two major, and rather different, approaches:

- The first approach, represented by languages as LIKELOG [20], replaces the syntactic unification mechanism of classical SLD-resolution by a fuzzy unification algorithm, based on similarity relations (over constants and predicates). The fuzzy unification algorithm provides an extended most general unifier as well as a numerical value, called *unification degree*. Intuitively, the unification degree represents the truth degree associated with the (query) computed instance. Programs written in this kind of languages consist, in essence, in a set of ordinary (Prolog) clauses jointly with a set of "similarity equations" which plays an important role during the unification process.
- For the second approach programs are fuzzy subsets of clausal formulas, where the *truth degree* of each clause is explicitly annotated. The work of computing and propagating truth degrees relies on an extension of the resolution principle, whereas the (syntactic) unification mechanism remains untouched. Examples of this kind of languages are the one described in [11], [1] or the one presented in [12] (although it uses interval fuzzy logic instead of truth degrees).

We are mainly interested in the second class of fuzzy logic languages. Among the variety of fuzzy logic programming languages in the literature, the one described in [1] is specially appropriated to define the concept of unfolding of fuzzy logic programs. In this section we present this language, that we call f-Prolog (*fuzzy* Prolog).

Let $\mathcal{L}$ be a first order language containing variables, function symbols, predicate symbols, constants, quantifiers, $\forall$ and $\exists$, and connectives $\neg$, seq, et_1, and et_2 (the intended meaning is that seq is an implication —the left-arrow version is written as qes—, et_1 is a conjunction evaluating *modus ponens* with seq, and et_2 is a conjunction typically occurring in the body of clauses). Although et_1 and et_2 are binary connectives, we usually generalize them as functions with an arbitrary number of arguments. That is we write, for instance, $\mathrm{et}_2(x_1, \ldots, x_n)$ instead of $\mathrm{et}_2(x_1, \mathrm{et}_2(x_2, \ldots, \mathrm{et}_2(x_{n-1}, x_n) \ldots))$.

A (definite) *clause* is a formula $\forall(A \text{ qes } \mathrm{et}_2(B_1, \ldots, B_n))$, and a (definite) *goal* is a formula $\forall(\text{qes } \mathrm{et}_2(B_1, \ldots, B_n))$, where A and each B_i are atomic formulas. In general, we call them $(\mathrm{seq}, \mathrm{et}_2)$-*formulas* or simply *formulas* if the kind of connectives used in their writing is not important or can be inferred by the context. Also, we write $A \leftarrow B_1, \ldots, B_n$ as syntactic sugar of $\forall(A \text{ qes } \mathrm{et}_2(B_1, \ldots, B_n))$, etc. As usually, A is said to be the head of the clause and $(B_1, \ldots, B_n)$ the body. Clauses with an empty body are called *facts*, whereas clauses with a head and a body are called *rules*. A sort of degenerate clause is the *empty clause*, denoted '$\square$', representing a contradictory formula.

Definition 2.1: [1] A *fuzzy theory* is a partial mapping T

applying formulas into real numbers in the interval $(0, 1]$.

A *definite* f-Prolog *program*, $\mathcal{P}$, is a fuzzy theory such that:

1) $\mathrm{dom}(\mathcal{P})$ is a set of $(\mathrm{seq}, \mathrm{et}_2)$-definite program clauses or facts,
2) $\mathrm{dom}(\mathcal{P})/_{\approx}$ is finite,
3) for $\mathcal{C}_1 \approx \mathcal{C}_2$ and $\mathcal{C}_1 \in \mathrm{dom}(\mathcal{P})$ we have $\mathcal{C}_2 \in \mathrm{dom}(\mathcal{P})$ and $\mathcal{P}(\mathcal{C}_1) = \mathcal{P}(\mathcal{C}_2)$.

where, given two formulas $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A} \approx \mathcal{B}$ if and only if $\mathcal{A}$ is a variant of $\mathcal{B}$.

Informally, a (definite) f-Prolog program can be seen as a set of pairs $(\mathcal{C}; \alpha)$, where $\mathcal{C}$ is a (definite) clause and $\alpha = \mathcal{P}(\mathcal{C})$ is a *truth degree* expressing the confidence which the user of the system has in the truth of the clause $\mathcal{C}$. Often, we'll write '$\mathcal{C}$ with $\alpha = \mathcal{P}(\mathcal{C})$" instead of $(\mathcal{C}; \mathcal{P}(\mathcal{C}))$. A truth degree $\alpha = 1$ means that the user believes the clause $\mathcal{C}$ is true; on the other hand, a truth degree less than 1 represents the degree of uncertainty or lost of confidence on the truth of a belief; a truth degree near 0 expresses the lack of confidence on the truth of a belief.

In [11], [1], the declarative semantics of a f-Prolog program is given in terms of a *least fuzzy Herbrand model*. Also, its characterization by a fix point semantics is presented.

Sometimes, the meaning functions for connectives et_1 and et_2 are defined as follows: $[\![\mathrm{et}_1]\!](r_1, \ldots, r_n) = \prod_{i=1}^{n} r_i$ and $[\![\mathrm{et}_2]\!](r_1, \ldots, r_n) = min(r_1, \ldots, r_n)$, but we prefer let them unspecified as arbitrary t-norms $[\![\mathrm{et}_i]\!] : [0,1]^2 \to [0,1]$ — i.e. they are commutative, associative, and monotone in both arguments and $[\![\mathrm{et}_i]\!](x, 1) = x$ (hence, they subsume classical conjunction $\{0,1\}^2 \to \{0,1\}$)— properly extended as many valued functions. Note that, in general, meaning functions for et_i connectives are not distributive.

III. Operational Semantics and Labeled Fuzzy Prolog

Given a goal $\mathcal{G}$ its truth degree, α, is obtained by evaluating a sequence of Fuzzy SLD-Resolution steps leading to an empty clause. In the sequel we formalize the concepts of Fuzzy SLD-Resolution, Fuzzy SLD-Derivation and Fuzzy computer answer, with slight variations with regard to the definitions that appear in [1].

Let $\mathcal{P}$ be a program and $\mathcal{G}$ a goal. Since $\mathrm{dom}(\mathcal{P})/_{\approx}$ is a classical definite program, the classical SLD-resolution should still work. Therefore, the main operational problem is to define the machinery for evaluating truth degrees. The truth degree of an expression is a semantic notion that must be evaluated using meaning functions. Considering a program rule $\mathcal{C} \equiv A \leftarrow B_1, \ldots, B_m$, with $\alpha = q$, and a goal $\mathcal{G} \equiv \leftarrow A'$, where A' unifies with the head A of $\mathcal{C}$, it is possible a SLD-resolution step leading to the resolvent $\mathcal{G}' \equiv \leftarrow (B_1, \ldots, B_m)$. If we want to evaluate the truth degree of $\mathcal{G}$, we have to compute the truth degrees $r_1, \ldots, r_n$ of all subgoals $B_1, \ldots, B_m$ before the truth degree q of the rule can be applied to obtain $[\![\mathrm{et}_1]\!](q, [\![\mathrm{et}_2]\!](r_1, \ldots, r_n))$, the truth degree of the goal $\mathcal{G}$. We need a mechanism in order to remenber that a program rule was applied in former steps, since it is necessary to distinguish when to apply $[\![\mathrm{et}_1]\!]$ or $[\![\mathrm{et}_2]\!]$. In [1] a context grammar was introduced to solve this problem. This grammar contains left and right marks ($\boxed{\mathrm{L}_q}$ and $\boxed{\mathrm{R}_q}$) labeled by a real value, to remember the exact point where a rule with $\alpha = q$ was applied. Hence the previous resolution step can be annotated as: $\boxed{\mathrm{L}_q}\, B_1, \ldots, B_m\, \boxed{\mathrm{R}_q}$.

We call lf-Prolog the extended language obtained by adding labeled marks and real numbers to the f-Prolog alphabet. An lf-expression is an atom, a sequence of real numbers, or a real number enclosed between labeled marks. The following definition makes use of lf-Prolog in the formalization of Fuzzy SLD-Resolution (we write $\overline{o}$ for the -possibly empty- sequence of syntactic objects $o_1, \ldots, o_n$).

Definition 3.1: Let $\mathcal{G} \equiv \leftarrow \mathcal{Q}$ be a lf-Prolog goal and let ϑ be a substitution, a lf-Prolog *state* is a pair $\langle \mathcal{Q}; \vartheta \rangle$. Let $\mathcal{E}$ be the set of lf-Prolog states. Given a f-Prolog program $\mathcal{P}$, we define *Fuzzy SLD-Resolution* as a state transition system, whose transition relation $\to_{FR} \subseteq (\mathcal{E} \times \mathcal{E})$ is the smallest relation satisfying the following rules:

Rule 1. **(Clause Resolution Rule)**

$\langle (\overline{X}, A_m, \overline{Y}); \vartheta \rangle \to_{FR} \langle (\overline{X}, \boxed{\mathrm{L}_q}\, \overline{B}\, \boxed{\mathrm{R}_q}, \overline{Y})\theta; \vartheta\theta \rangle$ if

1) A_m is the selected atom,
2) θ is an mgu of A_m and A,
3) $\mathcal{P}(A \leftarrow \overline{B}) = q$ and $\overline{B}$ is not empty.

Rule 2. **(Fact Resolution Rule)**

$\langle (\overline{X}, A_m, \overline{Y}); \vartheta \rangle \to_{FR} \langle (\overline{X}, r, \overline{Y})\theta; \vartheta\theta \rangle$ if

1) A_m is the selected atom,
2) θ is an mgu of A_m and A, and
3) $\mathcal{P}(A \leftarrow) = r$.

Rule 3. **($[\![\mathrm{et}_1]\!]$ Resolution Rule)**[2]

$\langle (\overline{X}\, \boxed{\mathrm{L}_q}\, r\, \boxed{\mathrm{R}_q}\, \overline{Y}); \vartheta \rangle \to_{FR} \langle \overline{X}, [\![\mathrm{et}_1]\!](q, r), \overline{Y}; \vartheta \rangle$ if

1) r is a real number.

Rule 4. **($[\![\mathrm{et}_2]\!]$ Resolution Rule)**

$\langle \overline{X}, r_1, \ldots, r_n, \overline{Y}; \vartheta \rangle \to_{FR} \langle \overline{X}, [\![\mathrm{et}_2]\!](r_1, \ldots, r_n), \overline{Y}; \vartheta \rangle$ if

1) $r_1, \ldots, r_n$ are real numbers.

All familiar logic programming concepts (SLD-resolution step, SLD-derivation, etc.) can be extended for the fuzzy case, assuming also that clauses involved in fuzzy SLD-computation steps are renamed before being used. In the following, symbols $\to_{FR1}, \to_{FR2}, \to_{FR3}$ and $\to_{FR4}$ may be used for explicitly referring to the application of each one of the four fuzzy resolution rules. When needed, the exact lf-expression and/or clause used in the corresponding step, will be also annotated as a super-index of the $\to_{FR}$ symbol. In order to extend the notion of computer answer in our fuzzy setting, in the following definition we use id to refer to the empty substitution, $\mathcal{V}ar(s)$ denotes the set of distinct variables occurring in the syntactic object s, and $\theta[\mathcal{V}ar(s)]$ corresponds to the substitution obtained from θ by restricting its domain, $Dom(\theta)$, to $\mathcal{V}ar(s)$.

Definition 3.2: Let $\mathcal{P}$ be a f-Prolog program and $\mathcal{G} \equiv \leftarrow \mathcal{Q}$ be a lf-Prolog goal. A pair $\langle r; \theta \rangle$ consisting of a real number

[2] In [1], the $[\![\mathrm{et}_1]\!]$ resolution rule is expressed as a combination of our third and fourth rules but our definition is fully equivalent to the original one.

r and a substitution θ is a *fuzzy computed answer* (f.c.a.) if there is a sequence $\mathcal{E}_0, \ldots, \mathcal{E}_n$ (called f-derivation) such that:

1) $\mathcal{E}_0 = \langle \mathcal{Q}; id \rangle$;
2) for each $0 \leq i < n$, $\mathcal{G}_i \rightarrow_{FR} \mathcal{G}_{i+1}$ is a fuzzy SLD-resolution step;
3) $\mathcal{E}_n = \langle r; \theta' \rangle$ and $\theta = \theta'[Var(\mathcal{Q})]$.

We illustrate the last definition by means of an example.

Example 3.3: Let $\mathcal{P}$ be a f-Prolog program,

$\mathcal{C}_1$: $p(X) \leftarrow q(X,Y), r(Y)$ with $\alpha = 0.8$
$\mathcal{C}_2$: $q(a,Y) \leftarrow s(Y)$ with $\alpha = 0.7$
$\mathcal{C}_3$: $q(Y,a) \leftarrow r(Y)$ with $\alpha = 0.8$
$\mathcal{C}_4$: $r(Y) \leftarrow$ with $\alpha = 0.7$
$\mathcal{C}_5$: $s(b) \leftarrow$ with $\alpha = 0.9$

For the sake of simplycity, assume $[\![\mathrm{et}_1]\!](x,y) = x \cdot y$ and $[\![\mathrm{et}_2]\!](x,y) = \min(x,y)$. Moreover, in the following successful f-derivation for the program $\mathcal{P}$ and the goal $\leftarrow p(X)$, we underline the selected lf-expression in each resolution step and we also assume that: $\theta_1 = \{X/X_1\}$, $\theta_2 = \{X/a, X_1/a, Y_1/Y_2\}$, $\theta_3 = \{X/a, X_1/a, Y_1/b, Y_2/b\}$ and $\theta_4 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b\}$. So, the f-derivation is:

$\langle \underline{p(X)}; id \rangle \rightarrow_{FR1}{}^{\mathcal{C}_1}$
$\langle \boxed{\mathrm{L}_{0.8}}\ \underline{q(X_1,Y_1)}, r(Y_1)\ \boxed{\mathrm{R}_{0.8}}\ ; \theta_1 \rangle \rightarrow_{FR1}{}^{\mathcal{C}_2}$
$\langle \boxed{\mathrm{L}_{0.8}}\ \boxed{\mathrm{L}_{0.7}}\ \underline{s(Y_2)}\ \boxed{\mathrm{R}_{0.7}}, r(Y_2)\ \boxed{\mathrm{R}_{0.8}}\ ; \theta_2 \rangle \rightarrow_{FR2}{}^{\mathcal{C}_5}$
$\langle \boxed{\mathrm{L}_{0.8}}\ \boxed{\mathrm{L}_{0.7}}\ 0.9\ \boxed{\mathrm{R}_{0.7}}, \underline{r(b)}\ \boxed{\mathrm{R}_{0.8}}\ ; \theta_3 \rangle \rightarrow_{FR2}{}^{\mathcal{C}_4}$
$\langle \boxed{\mathrm{L}_{0.8}}\ \underline{\boxed{\mathrm{L}_{0.7}}\ 0.9\ \boxed{\mathrm{R}_{0.7}}}, 0.7\ \boxed{\mathrm{R}_{0.8}}\ ; \theta_4 \rangle \rightarrow_{FR3}$
$\langle \boxed{\mathrm{L}_{0.8}}\ \underline{0.63, 0.7}\ \boxed{\mathrm{R}_{0.8}}\ ; \theta_4 \rangle \rightarrow_{FR4}$
$\langle \underline{\boxed{\mathrm{L}_{0.8}}\ 0.63\ \boxed{\mathrm{R}_{0.8}}}\ ; \theta_4 \rangle \rightarrow_{FR3}$
$\langle 0.504\ ; \theta_4 \rangle$, with f.c.a. $\langle 0.504; \{X/a\} \rangle$.

In [1], the authors established the correctness results for the Fuzzy SLD-Resolution operational mechanism (following a technique similar as the one proposed by Lloyd, in [2], for classical logic programming), but extending all results with the treatment of truth degrees. As for the classical SLD-Resolution calculus, we assume the existence of a fixed selection function, also called *fuzzy computation rule*, deciding, for a given goal, which is the selected lf-expression to be exploited in the next fuzzy SLD-Resolution step. For instance, when building the f-derivation shown in Example 3.3, we have used a computation rule similar to the left to right selection rule of Prolog but delaying the application of the $[\![\mathrm{et}_1]\!]$ and $[\![\mathrm{et}_2]\!]$ resolution rules until all atoms have been resolved. Given a fuzzy computation rule $\mathcal{R}$, we say that a fuzzy SLD-derivation is *via* $\mathcal{R}$ if the selected lf-expression in every step is obtained by the application of the mapping $\mathcal{R}$ to the corresponding goal in that step. The following result extends to our fuzzy setting the independence of the computation rule proved in [2] for the pure logic programming case.

Theorem 3.4 (Independence of the Fuzzy Computation Rule): Let $\mathcal{P}$ be a lf-Prolog program, $\mathcal{G} \equiv \leftarrow \mathcal{Q}$ a lf-Prolog goal and $\mathcal{R}$ a fuzzy computation rule. If $\langle \mathcal{G}; id \rangle \rightarrow_{FR}{}^{n}_{\mathcal{R}} \langle r; \theta \rangle$ is a fuzzy SLD-derivation via $\mathcal{R}$, with length n, then there exists a fuzzy SLD-derivation $\langle \mathcal{G}; id \rangle \rightarrow_{FR}{}^{n}_{\mathcal{R}'} \langle r; \theta \rangle$ via any other fuzzy computation rule $\mathcal{R}'$, with the same length n.

IV. Fuzzy Unfolding of lf-Prolog Programs

As we have seen in the previous sections, the differences between f-Prolog and lf-Prolog programs appear only at the syntactic level: whereas the body B of a (non unit) f-Prolog program clause (which in essence, is no more than a simple goal, that is, an atom or a conjunction of atoms) respects the grammar $B \rightarrow B, \ldots, B \mid atom$, we need to enrich this set of grammar rules with $B \rightarrow \boxed{\mathrm{L}_p}\ B\ \boxed{\mathrm{R}_p} \mid number$, if we really want to cope with the possibility of including marks and real numbers in the body of lf-Prolog clauses (which intuitively have the same structure of any initial, intermediate or final goal appearing in fuzzy SLD-derivations). This implies that any f-Prolog program is also an lf-Prolog program, although the contrary is not always true (i.e., the set of f-Prolog programs is a proper subclass of the set of lf-Prolog programs). Apart from this simple fact (which, on the other hand, is mandatory to define the fuzzy SLD-resolution principle) both languages share all kind of semantics, like is obvious the operational one, but also the declarative and the least fix-point ones and their correctness/completeness properties, as described in [1].

As we know, the unfolding rule consists in essence in the application of a symbolic computation step on (the selected lf-expressions of) the body of a clause which, in our fuzzy setting, corresponds to the application of any of the four rules described in Definition 3.1. Observe that this process always generates clauses whose bodies include marks or numbers. Hence, an unfolding transformation based on fuzzy SLD-resolution is able to preserve the syntactic structure of lf-Prolog programs but, even in the case that original programs be also f-Prolog programs, the transformed ones will never belong to this subclass: the marks or real numbers incorporated in the transformed clauses by unfolding steps (based on the first or the second resolution rule) force the lost of the original f-Prolog syntax. In order to avoid this inconvenience, our following definition focuses in the general framework of lf-Prolog programs instead of the more restricted subclass of f-Prolog programs.

Definition 4.1: Let $\mathcal{P}$ be an lf-Prolog program and let $\mathcal{C} \equiv (A \leftarrow \overline{B}$ with $\alpha = p) \in \mathcal{P}$ be a (non unit) lf-Prolog program clause. Then, the fuzzy unfolding of program $\mathcal{P}$ w.r.t. clause $\mathcal{C}$ is the new lf-Prolog program $\mathcal{P}' = (\mathcal{P} - \{\mathcal{C}\}) \cup \mathcal{U}$ such that: $\mathcal{U} = \{A\sigma \leftarrow \overline{B'}$ with $\alpha = p \mid \langle \overline{B}; id \rangle \rightarrow_{FR} \langle \overline{B'}; \sigma \rangle\}$.

There are some remarks to do regarding our definition. Similarly to the classical SLD-resolution based unfolding rule presented in [16], the substitutions computed by resolution steps during unfolding are incorporated to the transformed rules in a natural way, i.e., by applying them to the head of the clause. On the other hand, regarding the propagation of truth degrees, we solve this problem in a very easy way: the unfolded clause directly inherits the truth degree α of the original clause.

However, a deeper analysis of the unfolding transformation reveals us that the body of the transformed clause, also

contains 'compiled-in' information on both components of a fuzzy computed answer (i.e., truth degree and substitution). Regarding truth degrees, we observe that if the unfolding steps are based on the third or fourth rules of Definition 3.1, the marks and numbers in the body of the transformed clause are simplified. Otherwise, the truth degree of the second clause involved in an unfolded step based on clause or fact resolution, is collected in the body of the transformed clause adopting the form of marks or real numbers. Summarizing, the propagation of truth degrees during unfolding is done at two different levels: i) by directly assigning the truth degree of the original clause as the truth degree of the transformed one, and ii) by simplifying/introducing marks and real numbers in its body. Those manipulations in the body of the clause will affect drastically the computation/propagation of truth degrees when solving goals against transformed programs. Let us now illustrate all these facts with an example.

Example 4.2: Consider again program $\mathcal{P}$ shown in Example 3.3. It is easy to see that the unfolding of program $\mathcal{P}$ w.r.t. clause $\mathcal{C}_2$ (exploiting the fact resolution rule of Definition 3.1) generates the new program $\mathcal{P}' = (\mathcal{P} - \{\mathcal{C}_2\}) \cup \{\mathcal{C}_{25}\}$, where $\mathcal{C}_{25}$ is the new unfolded rule $q(a,b) \leftarrow 0.9$ with $\alpha = 0.7$. On the other hand, if we want to unfold now clause $\mathcal{C}_1$ in program $\mathcal{P}'$, we must firstly generate the following one-step Fuzzy SLD-derivations (which only uses Rule 1 of Definition 3.1) where $\theta_0 = \{X/a, Y/b\}$ and $\theta_1 = \{X/Y_1, Y/a\}$:

$\langle \underline{q(X,Y)}, r(Y); id\rangle \rightarrow_{FR1}{}^{\mathcal{C}_{25}} \langle \boxed{\mathrm{L}_{0.7}}\ 0.9\ \boxed{\mathrm{R}_{0.7}}, r(b); \theta_0\rangle$

$\langle \underline{q(X,Y)}, r(Y); id\rangle \rightarrow_{FR1}{}^{\mathcal{C}_3} \langle \boxed{\mathrm{L}_{0.8}}\ r(Y_1)\ \boxed{\mathrm{R}_{0.8}}, r(a); \theta_1\rangle$

So, the unfolded program $\mathcal{P}'' = (\mathcal{P}' - \mathcal{C}_1) \cup \{\mathcal{C}_{125}, \mathcal{C}_{13}\}$ where $\mathcal{C}_{125} \equiv p(a) \leftarrow \boxed{\mathrm{L}_{0.7}}\ 0.9\ \boxed{\mathrm{R}_{0.7}}, r(b)$ with $\alpha = 0.8$ and $\mathcal{C}_{13} \equiv p(Y_1) \leftarrow \boxed{\mathrm{L}_{0.8}}\ r(Y_1)\ \boxed{\mathrm{R}_{0.8}}, r(a)$ with $\alpha = 0.8$. Moreover, by performing a new resolution step with the second rule of Definition 3.1 on the body of clause $\mathcal{C}_{125}$, we obtain the new unfolded rule $p(a) \leftarrow \boxed{\mathrm{L}_{0.7}}\ 0.9\ \boxed{\mathrm{R}_{0.7}}, 0.7$ with $\alpha = 0.8$. Observe that a subsequent unfolding step done now with the third rule of Definition 3.1 leads to $p(a) \leftarrow 0.63, 0.7$ with $\alpha = 0.8$ which, finally, becomes $p(a) \leftarrow 0.63$ with $\alpha = 0.8$ after the last unfolding step done with the fourth rule of Definition 3.1. It is important to note that the application of this last rule to the goal $\leftarrow p(X)$ simulates the effects of the first six resolution steps shown in the derivation of Example 3.3, which evidences the improvements achieved by unfolding on transformed programs.

The most important and practical purpose of the unfolding transformation, apart from preserving the program semantics, is to optimize code, independently of the object language. Classical fold/unfold based transformation systems optimize programs by returning code which uses the same source language, but unfolding has also played important roles in the design of compilers (see [21]) which generate an object code written in a target language. In this sense, our unfolding transformation can be seen as a mixed technique that optimizes f-Prolog programs and compiles it into lf-Prolog programs, with the advantage in our case that both programs are executable with exactly the same operational principle, apart from sharing any other kind of semantics and related strong correctness results. Moreover, the following result formalizes the best property one can expect in a transformation like our fuzzy unfolding, i.e., the exact and total correspondence between fuzzy computed answers for goals executed in the original and the transformed programs.

Theorem 4.3 (Strong Correctness of Fuzzy Unfolding): Let $\mathcal{P}$ be a lf-Prolog program and $\mathcal{G} \equiv \leftarrow \mathcal{Q}$ be a lf-Prolog goal. If $\mathcal{P}'$ is an lf-Prolog program obtained by fuzzy unfolding of $\mathcal{P}$, then, $\langle \mathcal{Q}; id\rangle \rightarrow_{FR}{}^* \langle r; \theta\rangle$ in $\mathcal{P}$ iff $\langle \mathcal{Q}; id\rangle \rightarrow_{FR}{}^* \langle r; \theta'\rangle$ in $\mathcal{P}'$, where $\theta = \theta'[Var(\mathcal{Q})]$.

V. Further Extensions

The developments presented before admit several extensions that may enrich not only the set of transformation rules but also the considered language. At the first level, we plan for the future to combine unfolding with other transformation rules in order to increase its optimization power. Some simplification rules are desirable, for instance the one that simplifies a clause of the form $A \leftarrow n$ with $\alpha = q$, where n is a real number, to a new fact of the form $A \leftarrow$ with $\alpha = [\![\mathrm{et}_1]\!](q, n)$.

On the other hand, it is important to note that some semantics aspect of our language were let unspecified. We have interpreted the operators et_1 and et_2 as arbitrary t-norms $[\![\mathrm{et}_i]\!] : [0,1]^2 \rightarrow [0,1]$. Therefore, all our results are applicable to a general class of fuzzy logic languages.

The concrete instances of lf-Prolog can be established by means of a directive: `:- semantics(`*Tnorm*`,` *Label*`)`, where *Label* is an atom indicating the meaning assigned to the et_1 or et_2 operator. For instance, the directive "`:- semantics(et1, lukasiewicz).`" interprets et_1 as a Lukasiewicz t-norm, that is, $[\![\mathrm{et}_1]\!](x,y) = max(0, x + y - 1)$. Other possible labels would be: `min`, if $[\![\mathrm{et}_1]\!](x,y) = min(x,y)$; `product`, if $[\![\mathrm{et}_1]\!](x,y) = x.y$; etc.

As it has been told in [12], it may be useful from a practical point of view to associate a concrete interpretation for each operator et_1 or et_2 in the context of a program clause instead of a fixed interpretation for the whole program environment (as we did in the previous case). A slightly modification of our language is enough to deal with the new requirement. We can redefine the concept of fuzzy theory to cope with this problem.

Definition 5.1: A *fuzzy theory* is a partial mapping T applying a triple $\langle r, le_1, le_2\rangle$, in $(0,1] \times Sem \times Sem$, to each formula, where Sem is a set of semantics labels indicating the associated meaning for et_1 and et_2 respectively. A `void` value in Sem is employed to express that no meaning for et_1 or et_2 is selected.

Roughly speaking, a program can be seen as a set of tuples $(\mathcal{C}; r, le_1, le_2)$, where $\mathcal{C}$ is a clause, but we prefer to write: $\mathcal{C}$ `with` α`=`r `and el=`le_1 `and e2=`le_2. If clause $\mathcal{C}$ is a fact, le_1 and le_2 are `void` and we simply write: $\mathcal{C}$ `with` α`=`r; omitting the values for le_1 and le_2. Now, a goal has also associated a semantic label. Given a goal $\mathcal{G}$, we write: $\mathcal{G}$ `with e2=`le_2.

In order to manage the unfolding transformation process properly, while extending the expressive power of our language, it is necessary to parameterize the labeled mark language of lf-Prolog. The mechanism to remember that a program rule was applied in former steps is expanded to distinguish what is the meaning operator $[\![et_1]\!]$ or $[\![et_2]\!]$ that must be applied. We introduce the marks $\boxed{L_{q,[\![et_1]\!],[\![et_2]\!]}}$ and $\boxed{R_{q,[\![et_1]\!],[\![et_2]\!]}}$ jointly with the following modifications in the fuzzy SLD-Resolution rules:

Rule 1. **(Clause Resolution Rule)**

$\langle \overline{X}, A_m, \overline{Y}; \vartheta \rangle \rightarrow_{FR}$
$\langle (\overline{X}, \boxed{L_{q,[\![et_1]\!],[\![et_2]\!]}} \overline{B} \boxed{R_{q,[\![et_1]\!],[\![et_2]\!]}}, \overline{Y})\theta; \vartheta\theta \rangle$ if

1) A_m is the selected atom,
2) θ is an mgu of A_m and A,
3) $\mathcal{P}(A \text{ qes } \overline{B}) = \langle q, [\![et_1]\!], [\![et_2]\!] \rangle$ and $\overline{B}$ is not empty.

Rule 2. **(Fact Resolution Rule)**

$\langle \overline{X}, A_m, \overline{Y}; \vartheta \rangle \rightarrow_{FR} \langle (\overline{X}, r, \overline{Y})\theta; \vartheta\theta \rangle$ if

1) A_m is the selected atom,
2) θ is an mgu of A_m and A, and
3) $\mathcal{P}(A \text{ qes}) = \langle r, void, void \rangle$.

Rule 3. **($[\![et_1]\!]$ Resolution Rule)**

$\langle \overline{X}, \boxed{L_{q,[\![et_1]\!],[\![et_2]\!]}}\, r \boxed{R_{q,[\![et_1]\!],[\![et_2]\!]}}, \overline{Y}; \vartheta \rangle \rightarrow_{FR}$
$\langle \overline{X}, [\![et_1]\!](q, r), \overline{Y}; \vartheta \rangle$ if r is a real number.

Rule 4. **($[\![et_2]\!]$ Resolution Rule)**

$\langle \overline{X}, \boxed{L_{q,[\![et_1]\!],[\![et_2]\!]}}, \overline{r}, \boxed{R_{q,[\![et_1]\!],[\![et_2]\!]}}, \overline{Y}; \vartheta \rangle \rightarrow_{FR}$
$\langle \overline{X}, \boxed{L_{q,[\![et_1]\!],[\![et_2]\!]}} [\![et_2]\!](\overline{r}), \boxed{R_{q,[\![et_1]\!],[\![et_2]\!]}}, \overline{Y}; \vartheta \rangle$
if $\overline{r} \equiv r_1, \ldots, r_n$ (where $n > 1$), are real numbers.

Given goal $\mathcal{G} \equiv \leftarrow \mathcal{Q}$ with $e2 = [\![et_2]\!]$, the initial state in a computation is $\langle \boxed{L_{0,void,[\![et_2]\!]}}, \mathcal{Q}, \boxed{R_{0,void,[\![et_2]\!]}}; id \rangle$. This is enough to overcome all the problems. Note also that results shown in previous sections trivially hold for the new language.

VI. Conclusions

This work introduces a safe transformation rule for the unfolding of fuzzy logic programs. To the best of our knowledge, this is the first time this issue, of integrating transformation techniques in the context of fuzzy logic languages, is treated in the literature.

After an inspection of the main proposals for the inclusion of fuzzy logic into a logic programming setting, we have selected the language described in [1], that we call lf-Prolog, since we think it is the best suited to deal with the problems that may arise in the transformation process of logic programs. It is remarkable that lf-Prolog is provided with a labeled mark language. We have extended this language in order to be able to code different fuzzy logics inside the same program, which greatly enhances the expressive power of the former language.

We have defined the unfolding of lf-Prolog programs (Definition 4.1) and we have demonstrated the (strong) correctness (Theorem 4.3) of the transformation rule. The results in this paper can be thought as a basis to optimize fuzzy prolog programs and they are the first step in the construction of a fold/unfold framework for optimizing this class of programs.

Acknowledgment

We are grateful to Ferrante Formato and Susana Muñoz for providing us worthy material and suggestive discussions on fuzzy dialects of Prolog which helped us to improve this paper. This work has been partially supported by CICYT under grant TIC 2001-2705-C03-03.

References

[1] P. Vojtas and L. P. k, "Soundness and completeness of non-classical extended SLD-resolution," in *Proc. ELP'96 Leipzig*, R. D. et al, Ed. LNCS 1050, Springer Verlag, 1996, pp. 289–301.

[2] J. Lloyd, *Foundations of Logic Programming*. Springer-Verlag, 1987.

[3] L. Zadeh, "Calculus of fuzzy restrictions," *Fuzzy Sets and Their Applications to Cognitive and Decision Processes*, pp. 1–39, 1975.

[4] H. Nguyen and E. Walker, *A First Course in Fuzzy Logic*. Chapman & Hall/CRC, Boca Ratón, Florida, 2000.

[5] L. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, pp. 338–353, 1965.

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

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

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

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

[10] F. Klawonn and R. Kruse, "A Łukasiewicz logic based Prolog," *Mathware & Soft Computing*, vol. 1, no. 1, pp. 5–29, 1994. [Online]. Available: citeseer.ist.psu.edu/klawonn94lukasiewicz.html

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

[12] C. Vaucheret, S. Guadarrama, and S. Muñoz, "Fuzzy prolog: A simple general implementation using *clp*(r)," in *Proc. of Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2002), Tbilisi, Georgia*, M. Baaz and A. Voronkov, Eds. LNAI 2514, Springer Verlag, 2002, pp. 450–463.

[13] G. J. Klir and B. Yuan, *Fuzzy Sets and Fuzzy Logic*.Prentice-Hall, 1995.

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

[15] R. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," *Journal of the ACM*, vol. 24(1), pp. 44–67, 1977.

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

[17] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal, "Rules + Strategies for Transforming Lazy Functional Logic Programs," *Theoretical Computer Science*, vol. 311, pp. 479–525, 2004.

[18] A. Pettorossi and M. Proietti, "A Comparative Revisitation of Some Program Transformation Techniques," in *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, O. Danvy, R. Glück, and P. Thiemann, Eds. Springer LNCS 1110, 1996, pp. 355–385.

[19] P. Julián, G. Moreno, and J. Penabad, "Unfolding Fuzzy Logic Programs," UCLM, Tech. Rep. DIAB-04-03-2, 2004. [Online]. Available: `http://www.info-ab.uclm.es/trep.php`.

[20] F. Arcelli and F. Formato, "Likelog: A logic programming language for flexible data retrieval," in *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99), February 28 - March 2, 1999, San Antonio, Texas, USA*. ACM, Artificial Intelligence and Computational Logic, 1999, pp. 260–267, electronic Edition (DOI: 10.1145/298151.298348).

[21] P. Julian and C. Villamizar, "Analizing Definitional Trees: Looking for Determinism," in *Proc. of the 7th Fuji International Symposyum on Functional and Logic Programming, FLOPS'04, Nara (Japan)*, Y. Kameyama and M. P. J. Stuckey, Eds. Springer-Verlag LNCS (To appear), 2004, p. 15.