Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 219 (2008) 19–34
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Optimizing Fuzzy Logic Programs by Unfolding, Aggregation and Folding [1]

Juan Antonio Guerrero and Ginés Moreno[2]

Dep. of Computing Systems, U. of Castilla–La Mancha, 02071 Albacete, Spain

Abstract

Multi-adjoint logic programming represents a very recent, extremely flexible attempt for introducing fuzzy logic into logic programming. Inspired by our previous experiences in the field of (declarative) program transformation, in this paper we propose the development of a fold/unfold based transformation system for optimizing such kind of fuzzy logic programs. The starting point is a set of unfolding-based transformations together with a reversible kind of fuzzy folding, that we have designed in the past. The present work substantially improves this last transformation operation by allowing the possibility of using rules belonging to different programs in a transformation sequence when performing a folding step, which is crucial to obtain better, recursive and elegant definitions of fuzzy predicates. In contrast with other declarative paradigms, in the fuzzy setting it is mandatory to pack sets of fuzzy predicates in tuples, if we really want the folding operation to proceed. This implies the need for re-defining the classical "definition introduction" transformation rule and introducing a completely new operation, that we call "aggregation", which is especially tailored for the new framework. Finally, we illustrate how the effects of appropriately applying our set of transformation rules (definition introduction, aggregation, folding, unfolding and facting) to a given program, are able to improve the execution of goals against transformed programs.

Keywords: Fuzzy Logic Programming, Program Transformation, Folding/Unfolding Rules

1 Introduction

Logic Programming (LP) has been widely used for problem solving and knowledge representation in the past [17]. Nevertheless, traditional LP languages do not incorporate techniques or constructs to deal explicitly with uncertainty and approximated reasoning.

On the other hand, *Fuzzy Logic Programming* is an interesting and still growing research area that agglutinates the efforts to introduce fuzzy logic into LP. 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

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

[2] Emails: {guerrero,gmoreno}@dsi.uclm.es

doi:10.1016/j.entcs.2008.10.032

of these systems implement the fuzzy resolution principle introduced by Lee in [15], such as languages Prolog–ELF [10], Fril [5] and F–Prolog [16].

This is also the case of *Multi-adjoint logic programming* [19,18], an extremely flexible framework combining fuzzy logic and logic programming. Informally speaking, a multi–adjoint logic 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). In this setting, 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. 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 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. So, we often write $@(x_1, \ldots, x_n)$ instead of $@(x_1, \ldots, @(x_{n-1}, x_n), \ldots)$. 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 intended to the evaluation of *modus ponens* [18]. In general, the set of truth values 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 $[0, 1]$ (which is a totally ordered lattice or chain). 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. As we will explain in Section 2, a *goal* is a body submitted as a query to the system which is then executed in two separate phases (*operational* and *interpretive*). Roughly speaking, a multi-adjoint logic program is a set of pairs $\langle \mathcal{R}; \alpha \rangle$ (we often write $\mathcal{R}$ *with* α), 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 $\mathcal{R}$. By abuse, we sometimes refer a tuple $\langle \mathcal{R}; \alpha \rangle$ as a "rule".

On the other hand, the so called *Fold/Unfold transformation* approach is an optimization technique for computer programs that, starting with an initial program $\mathcal{P}_0$, derives a sequence $\mathcal{P}_1, .., \mathcal{P}_n$ of transformed programs by applying *elementary*

transformation rules such as folding and unfolding (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. The basic idea is to divide the program development activity, starting with a (possibly naive) problem specification written in a programming language, into a sequence of small transformation steps. The development of useful fold/unfold based transformation systems was first introduced in [7] to optimize functional programs, then used for logic programs [29], and more recently, we have adapted this methodology to functional–logic programs in [21,4]. Nowadays, we are working in transferring our best experiences in this field to a fuzzy logic setting. As a result, in our preliminary work [22], we have just proposed a naive fuzzy transformation system including different transformation rules for unfolding (see also [11,12], for a detailed explanation) and a weak formulation of reversible folding. The present work largely improves such approach by solving a pending point we left open there. More exactly, we redefine here a non-reversible, much more powerful version of folding, whose effects can be reinforced when combined with a new (tupled) variant of the definition introduction rule and a completely original transformation operation called "aggregation", never previously known in the literature.

The structure of the paper is as follows. In Section 2, we summarize the main features of the procedural semantics of the programming language we use in this work. In Section 3 we recall from [22] our preliminary set of fuzzy transformation rules, which is further refined and extended in Section 4. Finally, in Section 5 we give our conclusions and propose future work.

2 Procedural Semantics of Fuzzy Logic Programs

Let us now formally introduce a summary of the main features of the fuzzy framework we are interested in (we send the interested reader to [19,18] for a complete formulation). The procedural semantics (i.e., the way in which programs are executed) of the multi–adjoint logic language $\mathcal{L}$ can be thought of as an operational phase followed by an interpretive one. Similarly to [12], in this section we establish a clear separation between both phases.

The operational mechanism uses a generalization of *modus ponens* that, given a goal A and a program rule $\langle A' \leftarrow_i \mathcal{B}, v\rangle$, if there is a substitution $\theta = mgu(\{A = A'\})$ [3], we substitute the atom A by the expression $(v \&_i \mathcal{B})\theta$. In the following, we write $\mathcal{C}[A]$ to denote a formula where A is a sub-expression (usually an atom) which occurs arbitrarily in the —possibly empty— context $\mathcal{C}[]$. Moreover, expression $\mathcal{C}[A/A']$ means the replacement of A by A' in context $\mathcal{C}[]$. Also we use $\mathcal{V}ar(s)$ for referring to the set of distinct variables occurring in the syntactic object s, whereas $\theta[\mathcal{V}ar(s)]$ denotes the substitution obtained from θ by restricting its domain, $Dom(\theta)$, to $\mathcal{V}ar(s)$.

Definition 2.1 [Admissible Steps] Let $\mathcal{Q}$ be a goal and let σ be a substitution. The

[3] Let $mgu(E)$ denote the *most general unifier* (see [14]) of an equation set E.

pair $\langle \mathcal{Q}; \sigma \rangle$ is a *state* and we denote by $\mathcal{E}$ the set of states. Given a program $\mathcal{P}$, an *admissible computation* is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathcal{E} \times \mathcal{E})$ is the smallest relation satisfying the following *admissible rules* (where we always consider that A is the selected atom in $\mathcal{Q}$):

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.

Observe in the first case that, when the truth degree v of the considered program rule is just $\top$, then $\mathcal{Q}[A/v\&_i\mathcal{B}] = \mathcal{Q}[A/\top\&_i\mathcal{B}]$, and by the property of conjunctor operators, $\top \& r = r \& \top = r$, we can safely perform the direct replacement $\mathcal{Q}[A/\mathcal{B}]$ (we will use this convention in what follows). Note also that, whereas rules 1 and 2 refers to effective admissible steps using program rules with and without bodies, respectively, rule 3 is introduced to cope with (possible) unsuccessful admissible derivations. Formulas involved in admissible computation steps are renamed before being used. When needed, 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, when required, the exact program rule used in the corresponding step will be annotated as a superscript of the $\rightarrow_{AS}$ symbol.

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

Example 2.3 Let $\mathcal{P}$ be the following program and let $([0,1], \leq)$ be the lattice where $\leq$ is the usual order on real numbers.

$$
\begin{array}{llll}
\mathcal{R}_1 : p(X,Y) & \leftarrow_{\mathtt{L}} q(X,Z) \&_{\mathtt{G}} q(Z,Y) & \text{with} & 0.93 \\
\mathcal{R}_2 : q(a,b) & \leftarrow & \text{with} & 0.9 \\
\mathcal{R}_3 : q(b,c) & \leftarrow & \text{with} & 0.85 \\
\mathcal{R}_4 : q(f(X), g(Y)) & \leftarrow_{\mathtt{P}} q(X,Y) & \text{with} & 0.8 \\
\mathcal{R}_5 : q(g(X), h(Y)) & \leftarrow_{\mathtt{P}} q(X,Y) & \text{with} & 0.95
\end{array}
$$

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

admissible step:

$$\begin{array}{ll}
\langle \underline{p(f(f(a)), W)}; id\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_1} \\
\langle (0.93 \&_{\mathtt{L}} (\underline{q(f(f(a)), Z1)} \&_{\mathtt{G}} q(Z1, Y1))); \sigma_1\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_4} \\
\langle (0.93 \&_{\mathtt{L}} ((0.8 \&_{\mathtt{P}} \underline{q(f(a), Y2)}) \&_{\mathtt{G}} q(g(Y2), Y1))); \sigma_2\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_4} \\
\langle (0.93 \&_{\mathtt{L}} ((0.8 \&_{\mathtt{P}} (0.8 \&_{\mathtt{P}} \underline{q(a, Y3)})) \&_{\mathtt{G}} q(g(g(Y3)), Y1))); \sigma_3\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_2} \\
\langle (0.93 \&_{\mathtt{L}} ((0.8 \&_{\mathtt{P}} (0.8 \&_{\mathtt{P}} 0.9)) \&_{\mathtt{G}} \underline{q(g(g(b)), Y1)})); \sigma_4\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_5} \\
\langle (0.93 \&_{\mathtt{L}} ((0.8 \&_{\mathtt{P}} (0.8 \&_{\mathtt{P}} 0.9)) \&_{\mathtt{G}} (0.95 \&_{\mathtt{P}} \underline{q(g(b), Y4)}))); \sigma_5\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_5} \\
\langle (0.93 \&_{\mathtt{L}} ((0.8 \&_{\mathtt{P}} (0.8 \&_{\mathtt{P}} 0.9)) \&_{\mathtt{G}} (0.95 \&_{\mathtt{P}} (0.95 \&_{\mathtt{P}} \underline{q(b, Y5)})))); \sigma_6\rangle & \rightarrow_{AS2}{}^{\mathcal{R}_3} \\
\langle (0.93 \&_{\mathtt{L}} ((0.8 \&_{\mathtt{P}} (0.8 \&_{\mathtt{P}} 0.9)) \&_{\mathtt{G}} (0.95 \&_{\mathtt{P}} (0.95 \&_{\mathtt{P}} 0.85)))); \sigma_7\rangle &
\end{array}$$

where:

$$\begin{aligned}
\sigma_1 &= \{X1/f(f(a)), W/Y1\}, \\
\sigma_2 &= \{X1/f(f(a)), W/Y1, X2/f(a), Z1/g(Y2)\}, \\
\sigma_3 &= \{X1/f(f(a)), W/Y1, X2/f(a), Z1/g(g(Y3)), X3/a, Y2/g(Y3)\}, \\
\sigma_4 &= \{X1/f(f(a)), W/Y1, X2/f(a), Z1/g(g(b)), X3/a, Y2/g(b), Y3/b\}, \\
\sigma_5 &= \{X1/f(f(a)), W/h(Y4), X2/f(a), Z1/g(g(b)), X3/a, \\
&\quad Y2/g(b), Y3/b, X4/g(b), Y1/h(Y4)\}, \\
\sigma_6 &= \{X1/f(f(a)), W/h(h(Y5)), X2/f(a), Z1/g(g(b)), X3/a, \\
&\quad Y2/g(b), Y3/b, X4/g(b), Y1/h(h(Y5)), X5/b, Y4/h(Y5)\}, \\
\sigma_7 &= \{X1/f(f(a)), W/h(h(c)), X2/f(a), Z1/g(g(b)), X3/a, \\
&\quad Y2/g(b), Y3/b, X4/g(b), Y1/h(h(c)), X5/b, Y4/h(c), Y5/c\}.
\end{aligned}$$

Since σ_7, when restricting its domain to the single variable W of the original goal, coincides with the single binding $\{W/h(h(c))\}$, the a.c.a. associated to this admissible derivation is: $\langle (0.93 \&_{\mathtt{L}} ((0.8 \&_{\mathtt{P}} (0.8 \&_{\mathtt{P}} 0.9)) \&_{\mathtt{G}} (0.95 \&_{\mathtt{P}} (0.95 \&_{\mathtt{P}} 0.85)))); \{W/h(h(c))\}\rangle$.

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 as follows.

Definition 2.4 [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 satisfaying: $\langle Q[@(r_1, r_2)]; \sigma\rangle \rightarrow_{IS} \langle Q[@(r_1, r_2)/[\![@]\!](r_1, r_2)]; \sigma\rangle$, where $[\![@]\!]$ is the truth function of connective $@$ in the lattice $\langle L, \preceq\rangle$ associated to $\mathcal{P}$.

Definition 2.5 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 2.6 We complete the previous derivation of Example 2.3 by executing the necessary interpretive steps in order to obtain the fuzzy computed answer (f.c.a.):

$$\begin{aligned}
&\langle (0.93 \&_{\mathtt{L}} ((0.8 \&_{\mathtt{P}} \underline{(0.8 \&_{\mathtt{P}} 0.9)}) \&_{\mathtt{G}} (0.95 \&_{\mathtt{P}} (0.95 \&_{\mathtt{P}} 0.85)))); \{W/h(h(c))\}\rangle && \rightarrow_{IS} \\
&\langle (0.93 \&_{\mathtt{L}} (\underline{(0.8 \&_{\mathtt{P}} 0.72)} \&_{\mathtt{G}} (0.95 \&_{\mathtt{P}} (0.95 \&_{\mathtt{P}} 0.85)))); \{W/h(h(c))\}\rangle && \rightarrow_{IS} \\
&\langle (0.93 \&_{\mathtt{L}} (0.576 \&_{\mathtt{G}} (0.95 \&_{\mathtt{P}} \underline{(0.95 \&_{\mathtt{P}} 0.85)}))); \{W/h(h(c))\}\rangle && \rightarrow_{IS} \\
&\langle (0.93 \&_{\mathtt{L}} (0.576 \&_{\mathtt{G}} \underline{(0.95 \&_{\mathtt{P}} 0.8075)})); \{W/h(h(c))\}\rangle && \rightarrow_{IS} \\
&\langle (0.93 \&_{\mathtt{L}} \underline{(0.576 \&_{\mathtt{G}} 0.7671)}); \{W/h(h(c))\}\rangle && \rightarrow_{IS} \\
&\langle \underline{(0.93 \&_{\mathtt{L}} 0.576)}; \{W/h(h(c))\}\rangle && \rightarrow_{IS} \\
&\langle 0.506; \{W/h(h(c))\}\rangle
\end{aligned}$$

Then, the f.c.a for this complete derivation is the pair $\langle 0.506; \{W/h(h(c))\}\rangle$. Usually, we refer to a *complete derivation* as the sequence of admissible/interpretive steps of the form $\langle Q; id\rangle \rightarrow^*_{AS} \langle \mathcal{Q}'; \sigma\rangle \rightarrow^*_{IS} \langle r; \sigma\rangle$ (sometimes we denote it by $\langle Q; id\rangle \rightarrow^*_{AS/IS} \langle r; \sigma\rangle$) where $\langle \mathcal{Q}'; \sigma[\mathcal{V}ar(\mathcal{Q})]\rangle$ and $\langle r; \sigma[\mathcal{V}ar(\mathcal{Q})]\rangle$ are, respectively, the a.c.a. and the f.c.a. for the derivation.

3 The Preliminary Set of Fuzzy Transformation Rules

In this section, we recall from [22] our initial set of transformation rules. From now, we consider a fixed transformation sequence $(\mathcal{P}_0, \ldots, \mathcal{P}_k)$, $k \geq 0$. Let us first give the rule for the introducing new predicates in a similar style to [29].

Definition 3.1 [Definition introduction] We may get program $\mathcal{P}_{k+1}$ by adding to $\mathcal{P}_k$ a new rule called "definition rule" (or "eureka") of the form [4] $p(\overline{x_n}) \leftarrow \mathcal{B}$ with 1, such that:

(i) p is a *new* predicate symbol not occurring in the sequence $\mathcal{P}_0, \ldots, \mathcal{P}_k$ [5].

(ii) $\overline{x_n}$ is the set of variables appearing in $\mathcal{B}$, and

(iii) every non-variable symbol occurring in $\mathcal{B}$ belongs to $\mathcal{P}_0$.

The introduction of a new eureka definition is virtually always the first step of a transformation sequence. Determining which definitions should be introduced is a clever task (which justifies the name "eureka" for the new rules) which falls into the realm of *strategies* (see [26] for a survey). In general, the main idea consists in producing a new rule whose body contains a subset of predicates appearing in the

[4] Observe that the $\leftarrow$ symbols does not need to be labeled with any sub-index due to the fact that the truth degree associated to eurekas are always the maximum one.

[5] Predicate symbols belonging to $\mathcal{P}_0$ are called *old* predicate symbols.

body of a program rule whose definition is intended to be improved by subsequent transformation steps.

Example 3.2 Consider now the following initial program $\mathcal{P}_0$ of a transformation sequence, for which we plan to improve the definition of its first rule:

$$\begin{array}{lll} \mathcal{R}_1 : p(X) \quad \leftarrow_{\mathtt{P}} q(X,Y) \&_{\mathtt{G}}\ r(Y) & \text{with} & 0.8 \\ \mathcal{R}_2 : q(a,Y) \leftarrow_{\mathtt{P}} s(Y) & \text{with} & 0.7 \\ \mathcal{R}_3 : q(Y,a) \leftarrow_{\mathtt{L}} r(Y) & \text{with} & 0.8 \\ \mathcal{R}_4 : r(Y) \quad \leftarrow & \text{with} & 0.6 \\ \mathcal{R}_5 : s(b) \quad \leftarrow & \text{with} & 0.9 \end{array}$$

Inspired by its first rule whose definition we want to improve, we can build the eureka rule $\mathcal{R}_6 : new(X,Y) \leftarrow q(X,Y) \&_{\mathtt{G}} r(Y)$ with 1, and then, the next program in the sequence is $\mathcal{P}_1 = \mathcal{P}_0 \cup \{\mathcal{R}_6\}$.

Let us now introduce the folding rule, which roughly speaking consists of the compression of a piece of code into an equivalent call. Our definition is closely related to the *reversible folding* rule defined for pure logic programs in [8].

Definition 3.3 [Reversible Folding] Let $\mathcal{R} : (A \leftarrow_i \mathcal{B}$ with $v) \in \mathcal{P}_k$ be a non-eureka rule (the "folded rule") and let $\mathcal{R}' : (A' \leftarrow \mathcal{B}'$ with $1) \in \mathcal{P}_k$ be a eureka rule (the "folding rule") such that, there exists a substitution σ verifying that $\mathcal{B}'\sigma$ is contained in $\mathcal{B}$. We may get program $\mathcal{P}_{k+1}$ by folding rule $\mathcal{R}$ w.r.t. eureka $\mathcal{R}'$ as follows: $\mathcal{P}_{k+1} = (\mathcal{P}_k - \{\mathcal{R}\}) \cup \{A \leftarrow_i \mathcal{B}[\mathcal{B}'\sigma / A'\sigma]$ with $v\}$.

There are two points regarding our last definition which are worth noticing:

- The condition which says that the folded rule $\mathcal{R}$ is a non-eureka rule whereas $\mathcal{R}'$ is a eureka rule is useful to avoid the risk of *self-folding*, that is, the possibility of folding a rule w.r.t. itself, hence producing a wrong rule with the same head and body which may introduce infinite loops on derivations and destroy the correctness properties of the transformation system.
- The substitution σ of Definition 3.3 is not a unifier but just a matcher, similarly to many other folding rules for logic programs, which have been defined in a similar "functional style" (see, e.g., [6,8,26,29]). Moreover, it has the advantage that it is easier to check and can still produce effective optimizations at a lower cost.

Example 3.4 Continuing with Example 3.2, the goal now is to link the eureka definition (which will be afterwards improved by means of unfolding/facting steps) to the original program. This is done by simply folding $\mathcal{R}_1$ w.r.t. eureka $\mathcal{R}_6$, thus obtaining $\mathcal{P}_2 = (\mathcal{P}_1 - \{\mathcal{R}_1\}) \cup \{\ \mathcal{R}_7 : p(X) \leftarrow_{\mathtt{P}} new(X,Y)$ with 0.8 $\}$.

On the other hand, the unfolding transformation can be seen as the inverse of the previous folding rule, and it has been traditionally considered in pure logic programming as the replacement of a program clause C by the set of clauses obtained

after applying a (SLD-resolution) symbolic computation step in all its possible forms on the body of C [26]. As detailed in [11], we have adapted this transformation to deal with multi–adjoint logic programs by defining it in terms of admissible steps. However, in the following definition we increase the power of the transformation by also allowing interpretive steps in its formulation [6].

Definition 3.5 [Unfolding] We may get program $\mathcal{P}_{k+1}$ by unfolding rule $\mathcal{R} : (A \leftarrow_i \mathcal{B} \text{ with } v) \in \mathcal{P}_k$ as follows: $\mathcal{P}_{k+1} = (\mathcal{P}_k - \{\mathcal{R}\}) \cup \{A\sigma \leftarrow_i \mathcal{B}' \text{ with } v \mid \langle \mathcal{B}; id\rangle \rightarrow_{AS/IS} \langle \mathcal{B}'; \sigma\rangle\}$.

With respect to this definition, we would like to mention the following:

- Similarly to the classical SLD–resolution based unfolding rule presented in [29], the substitutions computed by admissible steps during the unfolding process, are incorporated to the transformed rules in a natural way, i.e., by applying them to the head of the rule.
- On the other hand, regarding the propagation of truth degrees, we solve this problem in a very easy way: the unfolded rule directly inherits the truth degree α of the original rule.

We illustrate the use of unfolding by means of the following example.

Example 3.6 The next phase in our transformation sequence is devoted to improve the eureka definition by means of unfolding steps. If we want to unfold now rule $\mathcal{R}_6$, we must firstly build the following one–step admissible derivations:

$\langle \underline{q(X,Y)} \&_{\mathtt{G}} r(Y); id\rangle \rightarrow_{AS1}{}^{\mathcal{R}_2} \langle (0.7 \&_{\mathtt{P}} s(Y_0)) \&_{\mathtt{G}} r(Y_0); \{X/a, Y/Y_0\}\rangle$, and

$\langle \underline{q(X,Y)} \&_{\mathtt{G}} r(Y); id\rangle \rightarrow_{AS1}{}^{\mathcal{R}_3} \langle (0.8 \&_{\mathtt{L}} r(Y_1)) \&_{\mathtt{G}} r(a); \{X/Y_1, Y/a\}\rangle$.

Correspondingly, the resulting rules are:

$\mathcal{R}_8 : new(a, Y_0) \leftarrow ((0.7 \&_{\mathtt{P}} s(Y_0)) \&_{\mathtt{G}} r(Y_0))$ with 1, and

$\mathcal{R}_9 : new(Y_1, a) \leftarrow ((0.8 \&_{\mathtt{L}} r(Y_1)) \&_{\mathtt{G}} r(a))$ with 1.

Moreover, by performing now a $\rightarrow_{AS2}$ admissible step on the body of rule $\mathcal{R}_8$, we obtain the new rule $\mathcal{R}_{10} : new(a, b) \leftarrow ((0.7 \&_{\mathtt{P}} 0.9) \&_{\mathtt{G}} r(Y_0))$ with 1. Finally, a new unfolding step (based again in the second type of admissible step) on rule $\mathcal{R}_{10}$ generates the rule $\mathcal{R}_{11} : new(a, b) \leftarrow ((0.7 \&_{\mathtt{P}} 0.9) \&_{\mathtt{G}} 0.6)$ with 1.

On the other hand, we can now apply an interpretive step to unfold rule $\mathcal{R}_{11}$, obtaining $\mathcal{R}_{12} : new(a, b) \leftarrow (0.63 \&_{\mathtt{G}} 0.6)$ with 1, which, after the last (interpretive) unfolding step finally becomes $\mathcal{R}_{13} : new(a, b) \leftarrow 0.6$ with 1.

So, after these five unfolding steps, the resulting program is the set of rules $\mathcal{P}_7 = \{\mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4, \mathcal{R}_5, \mathcal{R}_7, \mathcal{R}_9, \mathcal{R}_{13}\}$.

Our last transformation rule is not previously known in the literature on program transformation and declarative programming, with the unique exception of [22]. The idea is similar to the application of an interpretive step to the body of program rules, but, in contrast with unfolding, not only the truth degrees of the transformed rules differs from the original ones, but also, and what is better, the transformation is able

[6] This last case case remembers the so-called *interpretive unfolding* formalized in [12].

to simplify program rules by directly eliminating its bodies, and hence, producing facts.

Definition 3.7 [Facting] We may get program $\mathcal{P}_{k+1}$ by facting rule $\mathcal{R} : (A \leftarrow_i r \text{ with } v) \in \mathcal{P}_k$, where $r \in L$, as follows: $\mathcal{P}_{k+1} = (\mathcal{P}_k - \{\mathcal{R}\}) \cup \{A \leftarrow \text{ with } [\![\&_i]\!](v, r)\}$.

Example 3.8 Let's perform now a facting step on rule $\mathcal{R}_{13}$. Since, as we have said before, the truth function for any conjunction operator verifies $[\![\&]\!](1, v) = [\![\&]\!](v, 1) = v$, then $[\![\&]\!](1, 0.6) = 0.6$, which implies that $\mathcal{R}_{14}$ is $new(a, b) \leftarrow$ with 0.6. So, the final program $\mathcal{P}_8$ of our transformation sequence contains the original rules $\mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4$ and $\mathcal{R}_5$ together with:

$$\begin{array}{lllll} \mathcal{R}_7 : & p(X) & \leftarrow_{\mathtt{P}} new(X, Y) & \text{with} & 0.8 \\ \mathcal{R}_9 : & new(Y_1, a) & \leftarrow (0.8 \;\&_{\mathtt{L}}\; r(Y_1)) \;\&_{\mathtt{G}}\; r(a) & \text{with} & 1 \\ \mathcal{R}_{14} : & new(a, b) & \leftarrow & \text{with} & 0.6 \end{array}$$

Now, it is possible to generate the following derivation in program $\mathcal{P}_8$:

$$\begin{array}{rl} \langle \underline{p(X)} \&_{\mathtt{G}} r(a); id\rangle \rightarrow_{AS1}{}^{\mathcal{R}_7} & \langle (0.8 \&_{\mathtt{P}} \underline{new(X_1, Y_1)}) \&_{\mathtt{G}} r(a); \{X/X_1\}\rangle \\ \rightarrow_{AS2}{}^{\mathcal{R}_{14}} & \langle (0.8 \&_{\mathtt{P}} 0.6) \&_{\mathtt{G}} \underline{r(a)}; \{X/a, X_1/a, Y_1/b\}\rangle \\ \rightarrow_{AS2}{}^{\mathcal{R}_4} & \langle (\underline{0.8 \&_{\mathtt{P}} 0.6}) \&_{\mathtt{G}} 0.6; \{X/a, X_1/a, Y_1/b, Y_2/a\}\rangle \\ \rightarrow_{IS} & \langle \underline{0.48 \&_{\mathtt{G}} 0.6}; \{X/a, X_1/a, Y_1/b, Y_2/a\}\rangle \\ \rightarrow_{IS} & \langle 0.48; \{X/a, X_1/a, Y_1/b, Y_2/a\}\rangle. \end{array}$$

The reader may easily check that the same f.c.a. $\langle 0.48; \{X/a\}\rangle$ can be obtained for this goal if we generate a derivation sequence w.r.t. the original program $\mathcal{P}_0$ in Example 3.2. However, such derivation needs almost twice as many (admissible and interpretive) steps as the one seen before, which illustrates the benefits obtained by fold/unfold on the transformed program $\mathcal{P}_8$.

4 Non-Reversible Folding and Related Rules

In this section, our aim is to refine and extend the set of program transformation rules analyzed in the previous section. More exactly, we are interested in reinforcing the power of the reversible folding described in Definition 3.3, which similarly to the first approach of *reversible* folding presented in [8] for pure logic programs, has effects that can always be undone by an unfolding step. The main problem of such kind of formulations is that they usually require too strong applicability conditions, such as requiring that both the folded and the folding rules belong to the same program. This fact drastically reduces the power of the transformation, as also occurred with the folding rule we proposed in [1] for a functional logic language. In fact, the unique objective of reversible folding consists in "linking" a eureka definition to the body of a program rule, which indirectly implies that it is not able

to generate recursive definitions of eureka predicates.

On the other hand, a large number of proposals also allow the folded and the folding rule to belong to different programs (see, e.g., [6,13,25,26,29]), which is crucial to achieve an effective optimization in many cases, mainly due to its capability for obtaining recursive definitions of eureka predicates. This is not a trivial task, as we are going to explain. A *eureka rule* maintains its status only as long as no transformation step (in particular, unfolding) is applied on it. Once we transform this rule—even if the resulting rule is syntactically equal to the original one—it is not considered a *eureka rule* anymore (as we have seen in Definition 3.3, this is important for the folding operation, since we can only fold *non eureka rules* using *eureka rules*). So, assume that we apply a definition introduction step in a program $\mathcal{P}_i$. Then, the new eureka rule, say $\mathcal{R}_{new}$, is furthermore unfolded several times, generating new unfolded rules which are not considered eureka rules anymore. When we eventually find some "regularities" (that is, sub-expressions which could be compared with the body of the eureka rule) in the body of such unfolded rules, we try to fold them with respect to the original eureka rule but, unfortunately, $\mathcal{R}_{new}$ is not present now in the last program of the sequence (remember that it was lost just when being unfolded in program $\mathcal{P}_{i+1}$). So, any kind of reversible folding (in particular, the one described in Definition 3.3), always would fail in these cases, since it has not the capability of recalling a eureka rule from any program (not necessarily the last one) of the transformation sequence. We recover this power in the following definition, which is almost identical to Definition 3.3, but relaxing the condition for choosing the rule to be used during folding from any program of the transformation sequence.

Definition 4.1 [Folding] Let $\mathcal{R} : (A \leftarrow_i \mathcal{B} \text{ with } v) \in \mathcal{P}_k$ be a non-eureka rule (the "folded rule") and let $\mathcal{R}' : (A' \leftarrow \mathcal{B}' \text{ with } 1) \in \mathcal{P}_i, \quad 0 \leq i \leq k$, be a eureka rule (the "folding rule") such that, there exist a substitution σ verifying that $\mathcal{B}'\sigma$ is contained in $\mathcal{B}$. We may get program $\mathcal{P}_{k+1}$ by folding rule $\mathcal{R}$ w.r.t. eureka $\mathcal{R}'$ as follows: $\mathcal{P}_{k+1} = (\mathcal{P}_k - \{\mathcal{R}\}) \cup \{A \leftarrow_i \mathcal{B}[\mathcal{B}'\sigma/A'\sigma] \text{ with } v\}$.

Let us now have a look again to the transformation sequence we built in the previous section. In particular, the definition introduction step performed in Example 3.2, generates the eureka rule $\mathcal{R}_6 : new(X,Y) \leftarrow q(X,Y) \&_{\mathtt{G}} r(Y)$ with 1, whose body only contains two atoms and a single aggregation operator. Moreover, after unfolding it several times in Example 3.6, we have seen that the resulting unfolded rules never reached "regularities" (w.r.t. the original eureka definition) in their bodies. This prevents the application of the folding steps we are looking for, in order to achieve the intended recursive definition for predicate *new*.

It is important to note that this problem is not exclusive of this simple example but, what is worse, it seems to prevail in many other cases. The main reason for this is that, in the fuzzy setting we are working with, the operational semantics of multi-adjoint logic programming introduces truth values and adjoint conjunction operators (of program rules) each time a computation step is performed at transformation time. This novelty with respect to any other (crisp) declarative paradigm,

drastically harms the transformation process. Hence, such elements, which are crucial to cope with fuzzyness in the new framework, are really dangerous when looking for regularities on transformed rules. So, the more unfolding steps we apply on a given program rule, more "noisy" elements are introduced in its body, and consequently, less opportunities it has to be folded.

In the following, we face the problem by proposing the use of "tuples of atoms" during the transformation process in order to mimimize as much as possible the presence of truth-degrees and adjoint conjunctors inside the bodies of program rules. In particular, we propose the following version of the definition introduction rule, which in contrast with Definition 3.1, produces eureka rules which only contain (tuples of) atoms on their bodies.

Definition 4.2 [Tupled Definition Introduction] We may get program $\mathcal{P}_{k+1}$ by adding to $\mathcal{P}_k$ a new rule called "definition rule" (or "eureka") of the form $p(\overline{x_n}) \leftarrow \mathcal{B}$ with 1, such that:

(i) p is a *new* predicate symbol not occurring in the sequence $\mathcal{P}_0, \ldots, \mathcal{P}_k$.

(ii) $\overline{x_n}$ is the set of variables appearing in $\mathcal{B}$.

(iii) every non-variable symbol occurring in $\mathcal{B}$ belongs to $\mathcal{P}_0$.

(iv) $\mathcal{B}$ is a tuple containing at least two atoms, i.e., $\mathcal{B} = \langle B_1, \ldots, B_n \rangle, n > 1$.

Observe that the new "tupled definition introduction" rule, practically coincides with Definition 3.1, but with the single difference of the last condition we have just added in its formulation: the body of eureka rules only contains a set of predicates (neither truth degrees, nor aggregation operators) packed into a tuple. Now, by applying the tupled definition introduction rule to the program seen in Example 3.2, we would obtain the eureka rule: $new(X, Y) \leftarrow \langle q(X, Y), r(Y) \rangle$ with 1.

Although the presence of tuples in the extended syntax of the language could be interpreted as dangerous, we have the following important reasons for maintaining it with no risk:

- Its use is justified for transformation purposes, as we have detailed before. Moreover, we are just going to complete the explanation immediately, when describing our last transformation rule.
- Tuples are not intended to be managed by users when coding their own programs: they only will appear at transformation time, in a completely transparent way for the final user.
- Moreover, and what is the best, it must be taken into account that the underlying procedural semantics of the extended language remains untouched!

All these facts also hold in the formulation of our last rule, called aggregation, which is intended to remove (once again) truth degrees and adjoint conjunction operators from unfolded rules in order to give more chances to the folding transformation to successfully proceed. Although our aggregation operation slightly re-

sembles the classical "abstraction" transformation[7] used in pure functional and integrated functional-logic programming (see [7,26,27,28,4]), it must be clear that its formulation (apart from being mandatory in our fuzzy context, where we deal with elements such as truth degrees and fuzzy connectives with no sense in other "crisp" declarative paradigms) has never been previously proposed in the specialized literature.

Definition 4.3 [Aggregation] Let $\mathcal{R} : (A \leftarrow_i \mathcal{B}$ with $v) \in \mathcal{P}_k$ be a program rule and let E be a sub-expression of $\mathcal{B}$ containing a set of atoms $B_1, \ldots, B_m$, and a set of aggregators $@_1, \ldots, @_n$, such that $m > 1$. We may get program $\mathcal{P}_{k+1}$ by aggregating sub-expression E in rule $\mathcal{R}$ as follows: $\mathcal{P}_{k+1} = (\mathcal{P}_k - \{\mathcal{R}\}) \cup \{A \leftarrow_i \mathcal{B}[E/@(\langle B_1, \ldots, B_m \rangle)]$ with $v\}$, where $@$ is a *new* aggregator symbol not occurring in the sequence $\mathcal{P}_0, \ldots, \mathcal{P}_k$ whose truth function is defined as $[\![@]\!](\langle b_1, \ldots, b_m \rangle) = E[B_1/b_1, \ldots, B_m/b_m, @_1/[\![@_1]\!], \ldots, @_n/[\![@_n]\!]]$.

We wish to remark some relevant aspects of the previous definition:

- The main goal of this transformation rule is to "clean" the body of a program rule, by displacing those non-atom elements (that is, truth degrees and connectives) from its body to the definition of a new aggregator operator.
- The original set of atoms is then packed into a tuple (in concordance with the body of eureka rules, which increases the opportunities for applying subsequent folding steps) and then, the whole tuple is used as a parameter of a new aggregation operator.
- Consequently, the truth function of the new aggregator uses in its definition a tuple of n arguments instead of directly the proper n arguments themselves. Although this convention contrasts with the one used so far, it has simply been adopted for technical reasons, without supposing a major inconvenient in practice.

The application of an aggregation step is not only crucial, but also mandatory, before applying a folding step, in particular when the intended rule has been unfolded several times, thus implying that the number of noisy elements in its body has considerably grown. Moreover, as we are going to see in the following example, even when the folding step is intended to be applied to an original rule, say $\mathcal{R}$, which has never been unfolded, it must be preceded by the appropriate aggregation step (in this case, similarly to Definition 3.3, the objective of the folding operation is not to obtain a recursive definition of a eureka predicate, but simply to link such eureka predicate to the body of rule $\mathcal{R}$).

Example 4.4 Let us now illustrate the set of transformation rules introduced in this section in action, by recalling again the program of Example 2.3. Our goal is to generate a transformation sequence starting with this initial program $\mathcal{P}_0$, in order to derive a final program which enjoys a better computational behaviour. The transformation process proceeds as follows:

[7] This rule (often known as *where–abstraction* rule [26]) is usually required to implement tupling and it essentially consists of replacing the occurrences of some expressions $e_1, \ldots, e_n$ in the right hand side of a rule $\mathcal{R}$ by fresh variables $z_1, \ldots, z_n$, adding the "local declarations" $z_1 = e_1, \ldots, e_n = z_n$ within a *where* expression in $\mathcal{R}$.

(i) By applying a " Tupled Definition Introduction" step, we introduce a eureka rule which packs into a tuple the two atoms in the body of rule $\mathcal{R}_1$, whose definition is intended to be improved. Then,

$$\mathcal{R}_6 : new(X, Z, Y) \leftarrow \langle q(X, Z), q(Z, Y)\rangle \quad \text{with} \quad 1$$

(ii) Now, by means of an unfolding step, we exploit in all its possible forms (by using rules $\mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4$ and $\mathcal{R}_5$, respectively) the first atom in the body of the eureka rule, obtaining:

$$\begin{array}{lll}
\mathcal{R}_7 : & new(a, b, Y) & \leftarrow \langle 0.9, q(b, Y)\rangle \quad \text{with} \quad 1 \\
\mathcal{R}_8 : & new(b, c, Y) & \leftarrow \langle 0.85, q(c, Y)\rangle \quad \text{with} \quad 1 \\
\mathcal{R}_9 : & new(f(X), g(Z), Y) & \leftarrow \langle (0.8 \&_{\mathtt{P}} q(X, Z)), q(g(Z), Y)\rangle \quad \text{with} \quad 1 \\
\mathcal{R}_{10} : & new(g(X), h(Z), Y) & \leftarrow \langle (0.95 \&_{\mathtt{P}} q(X, Z)), q(h(Z), Y)\rangle \quad \text{with} \quad 1
\end{array}$$

(iii) Unfolding (the last atom) of $\mathcal{R}_7, \mathcal{R}_8, \mathcal{R}_9$ and $\mathcal{R}_{10}$.

$$\begin{array}{lll}
\mathcal{R}_{11} : new(a, b, c) & \leftarrow \langle 0.9, 0.85\rangle & \text{with} \quad 1 \\
\mathcal{R}_{12} : new(b, c, Y) & \leftarrow \langle 0.85, \bot\rangle & \text{with} \quad 1 \\
\mathcal{R}_{13} : new(f(X), g(Z), h(Y)) & \leftarrow \langle (0.8 \&_{\mathtt{P}} q(X, Z)), & \\
 & \quad (0.95 \&_{\mathtt{P}} q(Z, Y))\rangle & \text{with} \quad 1 \\
\mathcal{R}_{14} : new(g(X), h(Z), Y) & \leftarrow \langle (0.95 \&_{\mathtt{P}} q(X, Z)), & \\
 & \quad (0.8 \&_{\mathtt{P}} \bot)\rangle & \text{with} \quad 1
\end{array}$$

At this point we observe that rules $\mathcal{R}_{11}$ and $\mathcal{R}_{12}$ do not admit more unfolding steps, whereas rule $\mathcal{R}_{14}$ contains a $\bot$ element in its body, which prevents further unfoldings since its hypothetical use at execution time could lead to unsuccessful derivations. On the other hand, the presence in the body of rule $\mathcal{R}_{13}$ of the same set of atoms appearing in the eureka rule $\mathcal{R}_6$, alerts us indicating that some regularities have emerged. Thus, a folding step seems to be applicable once the body of such rule be appropriately rearranged by the following aggregation step.

(iv) Aggregation of the whole body of rule $\mathcal{R}_{13}$:

$$\mathcal{R}_{15} : \quad new(f(X), g(Z), h(Y)) \leftarrow @_1(\langle q(X, Z), q(Z, Y)\rangle) \quad \text{with} \quad 1$$

such that $[\![@_1]\!](\langle b_1, b_2\rangle) = \langle [\![\&_{\mathtt{P}}]\!](0.8, b_1), [\![\&_{\mathtt{P}}]\!](0.95, b_2)\rangle$

(v) So, by folding rule $\mathcal{R}_{15}$ using eureka $\mathcal{R}_6$, we obtain the desired recursive definition for the eureka predicate *new*.

$$\mathcal{R}_{16} : new(f(X), g(Z), h(Y)) \leftarrow @_1(new(X, Z, Y)) \quad \text{with} \quad 1$$

Now, in order to link the previous definition to the original program, we proceed with the last aggregation and folding steps.

(vi) Aggregation on $\mathcal{R}_1$:

$$\mathcal{R}_{17}: \quad p(X,Y) \leftarrow_{\mathtt{L}} @_2(\langle q(X,Z)), q(Z,Y)\rangle) \quad \text{with} \quad 0.93$$

$$\text{such that} \qquad [\![@_2]\!](\langle b_1, b_2\rangle) = \quad [\![\&_{\mathtt{G}}]\!](b_1, b_2)$$

(vii) Folding $\mathcal{R}_{17}$ using $\mathcal{R}_6$:

$$\mathcal{R}_{18}: p(X,Y) \leftarrow_{\mathtt{L}} @_2(new(X,Z,Y)) \quad \text{with} \quad 0.93$$

So, the final program contains rules $\mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4, \mathcal{R}_5, \mathcal{R}_{12}$ and $\mathcal{R}_{14}$, together with:

$$\begin{array}{lll} \mathcal{R}_{11}: new(a,b,c) & \leftarrow \langle 0.9, 0.85\rangle & \text{with} \quad 1 \\ \mathcal{R}_{16}: new(f(X), g(Z), h(Y)) & \leftarrow @_1(new(X,Z,Y)) & \text{with} \quad 1 \\ \mathcal{R}_{18}: p(X,Y) & \leftarrow_{\mathtt{L}} @_2(new(X,Z,Y)) & \text{with} \quad 0.93 \end{array}$$

By using these rules, we can now highly simplify the evaluation of goal $p(f(f(a)), W)$ (for which we constructed two much more complex admissible and interpretive derivations in Examples 2.3 and 2.6, respectively) as follows[8]:

$$\begin{array}{ll} \langle \underline{p(f(f(a)), W)}; id\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_{18}} \\ \langle (0.93 \&_{\mathtt{L}} @_2(\underline{new(f(f(a)), Z1, W)})); id\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_{16}} \\ \langle (0.93 \&_{\mathtt{L}} @_2(@_1(\underline{new(f(a), Z1, W)}))); id\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_{16}} \\ \langle (0.93 \&_{\mathtt{L}} @_2(@_1(@_1(\underline{new(a, Z1, W)})))); id\rangle & \rightarrow_{AS1}{}^{\mathcal{R}_{11}} \\ \langle (0.93 \&_{\mathtt{L}} @_2(@_1(\underline{@_1(\langle 0.9, 0.85\rangle)}))); \{W/h(h(c))\}\rangle & \rightarrow_{IS} \\ \langle (0.93 \&_{\mathtt{L}} @_2(\underline{@_1(\langle 0.72, 0.8075\rangle)})); \{W/h(h(c))\}\rangle & \rightarrow_{IS} \\ \langle (0.93 \&_{\mathtt{L}} \underline{@_2(\langle 0.576, 0.7671\rangle)}); \{W/h(h(c))\}\rangle & \rightarrow_{IS} \\ \langle \underline{(0.93 \&_{\mathtt{L}} 0.576)}; \{W/h(h(c))\}\rangle & \rightarrow_{IS} \\ \langle (0.506; \{W/h(h(c))\}\rangle & \end{array}$$

5 Conclusions and Future Work

This paper continues the efforts we initiated in [22] for developing a powerful transformation system for fuzzy logic programs, by considering one of the most recent and flexible languages in the field which is based in the multi-adjoint logic approach presented in [18]. Helped by our previous experiences in the design of similar transformation tools for functional logic languages ([1,2,4,21]) and fuzzy variants of unfolding rules ([11,12]), we have been mainly concerned with the design of a powerful, non-reversible version of folding with largely improves the weaker one we initially proposed in [22]. In doing this, we have been forced to revisit the classical "definition introduction rule", in order to cope with tuples of atoms when generating eureka definitions. Moreover, a completely original transformation, called aggregation, has been introduced in order to appropriately manage such tuples. We have

[8] For readability reasons, we only explicitly annotate bindings related to variables of the original goal in derivation states.

shown that its use is mandatory for cleaning and re-arranging the body of programs rules before being folded.

We have also included a very simple, but effective strategy to guide the generation of transformation sequences in order to produce more efficient residual programs. Basically, the proposed heuristic proceeds in four stages as follows:

(i) We first generate a (tupled) eureka rule based on the body of a program rule whose definition is intended to be optimized.

(ii) Then, the eureka definition is improved as much as wanted by means of unfolding and facting steps.

(iii) When some regularities emerge on unfolded rules, we perform an appropriate aggregation step, followed by its subsequent folding step, thus obtaining and improved recursive definition of the eureka predicate.

(iv) Finally, a new combination of aggregation and folding steps, makes accessible such enhanced definition to the program rules defining old predicates.

Nowadays, we are implementing our technique in order to show its effectiveness in practice. For the future we are also interested in defining more sophisticated and powerful transformation strategies, such as composition and tupling, as we have previously done in functional logic programming ([3,20,9]), but focusing now in the fuzzy field. In the limit, we also think that all these proposals allow a future adaptation to the fully integrated field of *functional–fuzzy–logic* programming, by considering languages such as the one we are nowadays designing in [23,24].

References

[1] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In H. Heering M. Hanus and K. Meinke, editors, *Proc. of the International Conference on Algebraic and Logic Programming, ALP'97, Southampton (England)*, pages 1–15. Springer LNCS 1298, 1997.

[2] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In A. Middeldorp and T. Sato, editors, *Proc. of the 4th Fuji International Symposyum on Functional and Logic Programming, FLOPS'99, Tsukuba (Japan)*, pages 147–162. Springer LNCS 1722, 1999.

[3] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. An Automatic Composition Algorithm for Functional Logic Programs. In V. Hlaváč, K. G. Jeffery, and J. Wiedermann, editors, *Sofsem 2000—Theory and Practice of Informatics*, pages 289–297. Springer LNCS 1963, 2000.

[4] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Theoretical Computer Science, Elsevier*, 311(1-3):479–525, Jan. 2004.

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

[6] A. Bossi and N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16:47–87, 1993.

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

[8] P. A. Gardner and J. C. Shepherdson. Unfold/fold Transformation of Logic Programs. In J.L Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 565–583. The MIT Press, Cambridge, MA, 1991.

[9] S. González and G. Moreno. Improved Tupling for Optimizing Multi-Paradigm Declarative Programs. In C. Lemaitre, C.A. Reyes, and J.A. Gonzlez, editors, *Proc. of the 9th Ibero–American Conference on Artificial Intelligence, IBERAMIA'2004. Puebla, México, November 22-26*, pages 419–429. Springer LNAI 3315, 2004.

[10] 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.

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

[12] 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.

[13] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation. *Theoretical Computer Science*, 75:139–156, 1990.

[14] 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.

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

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

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

[18] J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Similarity-based unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146(1):43–62, 2004.

[19] J. Medina, M. Ojeda-Aciego, and P. Vojtáš. A procedural semantics for multi-adjoint logic programing. *Progress in Artificial Intelligence, EPIA'01, Springer-Verlag, LNAI*, 2258(1):290–297, 2001.

[20] G. Moreno. Automatic Optimization of Multi-Paradigm Declarative Programs. In F. J. Garijo, J. C. Riquelme, and M. Toro, editors, *Proc. of the 8th Ibero–American Conference on Artificial Intelligence, IBERAMIA'2002*, pages 131–140. Springer LNAI 2527, 2002.

[21] G. Moreno. Transformation Rules and Strategies for Functional-Logic Programs. *AI Communications, IO Press (Amsterdam)*, 15(2):3, 2002.

[22] G. Moreno. Building a Fuzzy Transformation System. In J. Wiedermann, G. Tel, J. Pokorn, M. Bielikov, and J. Stuller, editors, *Proc. of the 32nd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM'2006*, pages 409–418. Springer LNCS 3831, 2006.

[23] G. Moreno and V. Pascual. Programming with Fuzzy Logic and Mathematical Functions. In I. Bloch, A. Petrosino, and A. Tettamanzi, editors, *Proc. of the 6th International Conference on Fuzzy Logic and Applications, WILF'2005. Crema, Italy, September 15-17, 2005*, pages 89–98. Springer LNAI 3849, 2006. An extended version has been submitted to the journal: *Fuzzy Sets and Systems.*

[24] G. Moreno and V. Pascual. Soft Computing with Strict Similar Equality. In K. Sirlantzis, editor, *6th International Conference on Recent Advances in Soft Computing, RASC'2006. Canterbury, UK, July 10-12*, pages 24–29. University of Kent, 2006.

[25] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[26] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[27] D. Sands. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[28] W.L. Scherlis. Program Improvement by Internal Specialization. In *Proc. of 8th Annual ACM Symp. on Principles of Programming Languages*, pages 41–49. ACM Press, New York, 1981.

[29] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of Second Int'l Conf. on Logic Programming, Uppsala, Sweden*, pages 127–139, 1984.