Operational/Interpretive Unfolding of Multi-adjoint Logic Programs[1]

Pascual Julián
Dept. de Informática
Universidad de CLM
Escuela Superior de Informática
Campus Univ. 13071 Ciudad Real
Pascual.Julian@uclm.es

Ginés Moreno
Dept. de Informática
Universidad de CLM
Escuela Politécnica Superior
Campus Univ. 02071 Albacete
Gines.Moreno@uclm.es

Jaime Penabad
Dept. de Matemáticas
Universidad de CLM
Escuela Politécnica Superior
Campus Univ. 02071 Albacete
Jaime.Penabad@uclm.es

Abstract

Multi-adjoint logic programming represents a very recent, extremely flexible attempt for introducing fuzzy logic into logic programming (LP). In this setting, the execution of a goal w.r.t. a given program is done in two separate phases. During the operational one, *admissible steps* are systematically applied in a similar way to classical resolution steps in pure LP, thus returning a computed substitution together with an expression where all atoms have been exploited. This last expression is then interpreted under a given lattice during the so called interpretive phase, hence returning a value which represents the fuzzy component (*truth degree*) of the computed answer.

On the other hand, unfolding is a well known transformation rule widely used in declarative programming for optimizing and specializing programs, among other applications. In essence, it is usually based on the application of operational steps on the body of program rules. The novelty of this paper consists in showing that this process can also be made in terms of interpretive steps. We present two strongly related kinds of unfolding (operational and interpretive), which, apart from exhibiting strong correctness properties (i.e. they preserve the semantics of computed substitutions and truth degrees) they are able to significantly simplify the two execution phases when solving goals.

1 Introduction

Multi-adjoint logic programming [12, 13] is an extremely flexible framework combining fuzzy logic and logic programming, which largely improves older approaches previously introduced in this field (see, for instance [10, 6, 3, 11, 17, 2], where different fuzzy variants of Prolog have been proposed). An special mention deserves the fuzzy dialect of Prolog presented in [5], since it is very close to the language used here. However, we find two slight differences: whereas the multi-adjoint approach is based on "weighted" clauses (with and without body) whose truth degrees are elements of any appropriate lattice, in the Fuzzy Prolog of [5], truth degrees are based on Borel Algebras (union intervals), and they are only applied to facts (i.e., clauses with no body).

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 identity substitution). In the multi-adjoint logic programming framework, goals are evaluated, in a given program, in two separate computational phases. During the *operational* one, *admissible steps* (a generalization of the classical *modus ponens* inference rule) are systematically applied by a

[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.

backward reasoning procedure in a similar way to classical resolution steps in pure LP, thus returning a computed substitution together with an expression where all atoms have been exploited. This last expression is then interpreted under a given lattice during what we call the *interpretive* phase, hence returning a pair $\langle$*truth degree*; *substitution*$\rangle$ which is the fuzzy counterpart of the classical notion of computed answer traditionally used in LP.

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* (fold/unfold) which improve the original program. The fold/unfold transformation approach was first introduced in [4] to optimize functional programs and then used for logic programs [16] and (lazy) functional logic programs [1]. Program transformation also can be seen as a methodology for software development, hence its importance. 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.

Unfolding is a well-known, widely used, semantics-preserving program transformation rule. In essence, it is usually based on the application of operational steps on the body of program rules [14]. 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, etc.

The main contribution of this paper consists in showing that, in the framework of multi-adjoint logic programming, the unfolding process can be better understood, if resembling the two separate phases of the underlaying procedural semantics of multi-adjoint logic programming languages, we distinguish between operational and interpretive unfolding steps. Therefore, we present two strongly related kinds of unfolding: the operational (firstly introduced in [7]) an the interpretive. It is important to remark that in the original (multi-adjoint logic) language proposed in [13] and then used in [7], the interpretive phase is not modeled in terms of an state transition system, which prevents the definition/ application of the (interpretive) unfolding rule by means of interpretive steps. As a consequence, all results presented there are restricted to the operational phase. In this paper we overcome all these limitations, and we prove that the interpretive unfolding, apart from exhibiting the analogous strong correctness properties (i.e. it preserves the semantics of computed substitutions and truth degrees) of the operational unfolding originally presented in [7], is able to simplify and accelerate the interpretive phase when solving goals w.r.t. a given program.

On the other hand, we have recently introduced in [8] a transformation rule, the so-called T-Norm replacement, which can be seen as a primitive precedent of the present interpretive unfolding. In fact, the four variants of T-Norm replacements, also perform low-level manipulations on fuzzy expression involving T-Norm operations on program rules. However, our new approach improves this poorer technique in the following points:

- It manages a much more powerful and expressive language (the multi-adjoint logic programming language) but with a simpler syntax, a clearer procedural (operational/interpretive) semantics and, in general, a better formalization.

- Now, neither interpretive steps, nor interpretive unfolding, are dependent of a selection function (computation rule), as it does occur with any other fuzzy unfolding-based transformation rule described in the literature, since the evaluation order of expressions is fixed by the interpretive semantics.

- As a co-lateral consequence of the previous point and, as we will see when presenting Theorem 5.2, it is the first time that our correctness results admit a clearer proof scheme which is not linked to previous instrumental results about the independence of any kind of computation rule.

• Moreover, it is also the first time that our fuzzy variants of unfolding rules, recover the source-to-source language nature character in [7] and [8], where some auxiliary object languages (with complex constructors and intrincated artifices with no sense for the final user) were mandatory to code residual programs obtained after performing operational unfolding or T-Norm replacement.

The structure of the paper is as follows. In Section 2, we summarize the main features of the programming language we use in this work. Section 3 defines the procedural semantics of the language, establishing a clean separation between the operational and the interpretive phase of a computation. In Section 4 we recall the definition of operational unfolding and we define the new notion of interpretive unfolding. Section 5 focuses in the properties relative to the correctness of the unfolding transformations. Finally, in Section 6 we give our conclusions and propose future work.

2 Multi-Adjoint Logic Programs

This section is a short summary of the main features of our language. Contrary to other previous work [7] we start from the scratch by manipulating an extended version of the *multi–adjoint logic programing* language presented in [12, 13]. We send the reader to these works for a complete formulation.

We work with a first order language, $\mathcal{L}$, containing variables, function symbols, predicate symbols, constants, quantifiers, $\forall$ and $\exists$, and several (arbitrary) connectives to increase language expressiveness:

$\&_1$,	$\&_2$,	...,	$\&_k$	`(conjunctions)`
$\vee_1$,	$\vee_2$,	...,	$\vee_l$	`(disjunctions)`
$\leftarrow_1$,	$\leftarrow_2$,	...,	$\leftarrow_m$	`(implications)`
$@_1$,	$@_2$,	...,	$@_n$	`(aggregations)`

Although the connectives $\&_i$, $\vee_i$ and $@_i$ are binary operators, we usually generalize them as functions with an arbitrary number of arguments. In the following, we often write $@_i(x_1, \ldots, x_n)$ instead of $@_i(x_1, @_i(x_2, \ldots, @_i(x_{n-1}, x_n)\ldots))$. Moreover, the truth function for an n-ary aggregation operator $[\![@]\!] : [0,1]^n \to [0,1]$ is required to be non-monotonous and fulfill $[\![@]\!](1, \ldots, 1) = 1$ and $[\![@]\!](0, \ldots, 0) = 0$.

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 [1] intended to the evaluation of *modus ponens*. In general, the set of truth values L may be the carrier of any complete bounded lattice, but, in the examples, we shall select L as the set of real numbers in the interval $[0, 1]$.

A *rule* is a formula $A \leftarrow_i \mathcal{B}$, where A is an atomic formula (usually called the *head*) and $\mathcal{B}$ (which is called the *body*) is a formula built from atomic formulas $B_1, \ldots, B_n$ — $n \geq 0$ —, truth values of L and conjunctions, disjunctions and aggregations. Rules with an empty body are called *facts*. A *goal* is a body submitted as a query to the system. Variables in a rule are assumed governed by universal quantifiers.

Roughly speaking, a multi–adjoint logic program is a set of pairs $\langle \mathcal{R}; \alpha\rangle$, where $\mathcal{R}$ is a rule and α is a *truth degree* (a value of L) expressing the confidence which the user of the system has in the truth of the rule $\mathcal{R}$. Often, we will write "$\mathcal{R}$ with α" instead of $\langle \mathcal{R}; \alpha\rangle$. Observe that, truth degrees are axiomatically assigned (for instance) by an expert.

3 Procedural Semantics

The procedural semantics of the multi–adjoint logic language $\mathcal{L}$ can be thought as an operational phase followed by an interpretive one. Although this point of view is present in [12, 13], in this section we establish a cleaner separation between both phases. We also give a novel definition of the interpretive phase, with a procedural taste, useful not only for clarify the whole computational mechanism, but also crucial for formalizing the concept of interpretive unfolding in Section 4.2.

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

3.1 Operational phase

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 $\theta = mgu(\{A = A'\})$[2], we substitute the atom A by the expression $(v \&_i \mathcal{B})\theta$.

In the following, we define the concepts of admissible computation step, admissible derivation and admissible computed answer, associated to the operational phase. In the formalization of these concepts, we write $\mathcal{C}[A]$, or more generally $\mathcal{C}[A_1, .., A_n]$, to denote a formula where A, or $A_1, .., A_n$ respectively, are sub-expressions (usually atoms) which arbitrarily occur in the —possibly empty— context $\mathcal{C}[]$. Moreover, expression $\mathcal{C}[A/A']$ (and its obvious generalization) 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 3.1 (Admissible Steps) *Let $\mathcal{Q}$ be a goal and let σ be a substitution. The pair $\langle \mathcal{Q}; \sigma\rangle$ is an* 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*:*

Rule 1.

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

Rule 2.

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

[2]Let $mgu(E)$ denote the *most general unifier* of an equation set E (see [9] for a formal definition of this concept).

Rule 3.

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

Formulas involved in admissible computation steps are renamed before being used. Note also that, Rule 3 is introduced to cope with (possible) unsuccessful admissible derivations.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 super–index of the $\rightarrow_{AS}$ symbol.

Definition 3.2 *Let $\mathcal{P}$ be a program and let $\mathcal{Q}$ be a goal. An* admissible derivation *is a sequence $\langle \mathcal{Q}; id\rangle \rightarrow^*_{AS} \langle \mathcal{Q}'; \theta\rangle$. When $\mathcal{Q}'$ is a formula not containing atoms, the pair $\langle \mathcal{Q}'; \sigma\rangle$, where $\sigma = \theta[\mathcal{V}ar(\mathcal{Q})]$, is called an* admissible computed answer *(a.c.a.) for that derivation.*

We illustrate these concepts by means of the following example.

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

$$\begin{array}{ll}
\mathcal{R}_1 : \langle p(X) \leftarrow_{\texttt{prod}} q(X,Y) \&_{\texttt{G}}\ r(Y); & \alpha = 0.8\rangle \\
\mathcal{R}_2 : \langle q(a,Y) \leftarrow_{\texttt{prod}} s(Y); & \alpha = 0.7\rangle \\
\mathcal{R}_3 : \langle q(Y,a) \leftarrow_{\texttt{luka}} r(Y); & \alpha = 0.8\rangle \\
\mathcal{R}_4 : \langle r(Y) \leftarrow_{\texttt{luka}}; & \alpha = 0.7\rangle \\
\mathcal{R}_5 : \langle s(b) \leftarrow_{\texttt{luka}}; & \alpha = 0.9\rangle
\end{array}$$

The labels `prod`, `G` *and* `luka` *mean for* product logic, Gödel intuitionistic logic *and* Łukasiewicz logic, *respectively. That is,* $[\![\&_{\texttt{prod}}]\!](x,y) = x \cdot y$, $[\![\&_{\texttt{G}}]\!](x,y) = min(x,y)$, *and* $[\![\&_{\texttt{luka}}]\!](x,y) = max(0, x+y-1)$.

In the following admissible derivation for the program $\mathcal{P}$ and the goal $\leftarrow p(X) \&_{\texttt{G}} r(a)$, we underline the selected expression in each admissible step:

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

where

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

So, since $\sigma_5[\mathcal{V}ar(\mathcal{Q})] = \{X/a\}$, *the a.c.a. associated to this admissible derivation is:* $\langle (0.8\&_{\mathrm{prod}}((0.7\&_{\mathrm{prod}}0.9)\&_G 0.7))\&_G 0.7; \{X/a\}\rangle$.

3.2 Interpretive phase

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. This justifies the following notions of interpretive computation step, interpretive derivation and interpretive computed answer.

Definition 3.4 (Interpretive Step) *Let $\mathcal{P}$ be a program, $\mathcal{Q}$ a goal and σ a substitution. We formalize the notion of* interpretive computation *as a state transition system, whose transition relation $\rightarrow_{IS} \subseteq (\mathcal{E} \times \mathcal{E})$ is defined as*

$$\langle Q[@(r_1, r_2)]; \sigma\rangle \rightarrow_{IS} \langle Q[@(r_1, r_2)/[\![@]\!](r_1, r_2)]; \sigma\rangle$$

where $[\![@]\!]$ is the truth function of connective $@$ in the lattice $\langle L, \preceq\rangle$ associated to $\mathcal{P}$.

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

Usually, we refer to a *complete derivation* as the sequence of admissible/interpretive steps of the form $\langle Q; id\rangle \rightarrow^*_{AS} \langle \mathcal{Q}'; \sigma\rangle \rightarrow^*_{IS} \langle r; \sigma\rangle$, 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 i.c.a. for the derivation. Sometimes, we denote it by $\langle Q; id\rangle \rightarrow^*_{AS/IS} \langle r; \sigma\rangle$ and we say that $\langle r; \sigma\rangle$ is the final computed answer of the derivation.

Example 3.6 *We complete the previous derivation of Example 3.3 by executing the necessary interpretive steps to obtain the interpretive computed answer (i.c.a.) with respect to lattice* $([0, 1], \leq)$.

$\langle (0.8\&_{\mathrm{prod}}(\underline{(0.7\&_{\mathrm{prod}}0.9)}\&_G 0.7))\&_G 0.7; \{X/a\}\rangle$
$\rightarrow_{IS}$ $\langle (\underline{0.8\&_{\mathrm{prod}}(0.63}\&_G 0.7))\&_G 0.7; \{X/a\}\rangle$
$\rightarrow_{IS}$ $\langle \underline{(0.8\&_{\mathrm{prod}}0.63)}\&_G 0.7; \{X/a\}\rangle$
$\rightarrow_{IS}$ $\langle \underline{0.504\&_G 0.7}; \{X/a\}\rangle$
$\rightarrow_{IS}$ $\langle 0.504; \{X/a\}\rangle$

Then the i.c.a for this complete derivation is the pair $\langle 0.504; \{X/a\}\rangle$.

4 Fuzzy Unfolding Transformations

The unfolding transformation traditionally considered in pure LP consists in the replacement of a program clause C by the set of clauses obtained after applying a symbolic computation step in all its possible forms on the body of C [14].

As detailed in [7], we have adapted this transformation to deal with multi–adjoint logic programs by defining it in terms of operational steps (see Definition 3.1). Also, in [7], we proved that the application of unfolding transformation step to multi–adjoint logic programs is able to speed up goal evaluation by reducing the length of admissible derivations during the operational phase.

The main objective of the present section is to recall the definition of operational unfolding and to define and unfolding rule for interpretive steps. Note that our new notion of interpretive unfolding is intended to facilitate the evaluation of truth degrees during the interpretive phase.

4.1 Operational Unfolding

The following definition is recalled from [7], but we have slightly simplified it in the sense that now, the operational unfolding is formulated as a source-to-source language transformation instead of a (more involved) source-to-object language transformation.

Definition 4.1 (Operational Unfolding) *Let $\mathcal{P}$ be a program and let $\mathcal{R} : (A \leftarrow_i \mathcal{B}$ with $\alpha = v) \in \mathcal{P}$ be a (non unit) program rule. Then, the operational unfolding of rule $\mathcal{R}$ in program $\mathcal{P}$ is the new program $\mathcal{P}' = (\mathcal{P} - \{\mathcal{R}\}) \cup \mathcal{U}$ where $\mathcal{U} = \{A\sigma \leftarrow_i \mathcal{B}'$ with $\alpha = v \mid \langle \mathcal{B}; id\rangle \rightarrow_{AS} \langle \mathcal{B}'; \sigma\rangle\}$.*

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 admissible steps during the operational unfolding, 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.

However, a deeper analysis of the operational unfolding transformation reveals us that the body of transformed rules also contains 'compiled–in' information on both components of final computed answers (i.e., truth degree and substitution). Regarding truth degrees, we observe that the body of the transformed rule includes symbol $\perp$ if we performed a $\rightarrow_{AS3}$ admissible step, or the truth degree together with the corresponding adjoint conjunction of the second rule involved in the unfolded step when the applied admissible step was based on $\rightarrow_{AS2}$ or $\rightarrow_{AS1}$, respectively. So, the propagation of truth degrees during unfolding is done at two different levels:

1. by directly assigning the truth degree of the original rule as the truth degree of the transformed one, and

2. by introducing new truth degrees (of other rules or alternatively $\perp$) in its body.

We illustrate the definition of operational unfolding and its advantages by means of the following example.

Example 4.2 *Consider again program $\mathcal{P}$ shown in Example 3.3.It is easy to see that the unfolding of rule $\mathcal{R}_2$ in program $\mathcal{P}$ (exploiting the second admissible rule of Definition 3.1) generates the new program $(\mathcal{P} - \{\mathcal{R}_2\}) \cup \{\mathcal{R}_6\}$, where $\mathcal{R}_6$ is the new unfolded rule $q(a,b) \leftarrow_{\texttt{prod}} 0.9$ with $\alpha = 0.7$.*

On the other hand, if we want to unfold now rule $\mathcal{R}_1$, we must firstly build the following one–step admissible derivations:

$\langle \underline{q(X,Y)} \&_{\texttt{G}} r(Y); id\rangle \rightarrow_{AS1}{}^{\mathcal{R}_6}$
$\langle (0.7 \&_{\texttt{prod}} 0.9) \&_{\texttt{G}} r(b); \{X/a, Y/b\}\rangle$, *and*

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

So, the resulting unfolded rules are $\mathcal{R}_7$: $p(a) \leftarrow_{\texttt{prod}} (0.7 \&_{\texttt{prod}} 0.9) \&_{\texttt{G}} r(b)$ with $\alpha = 0.8$, and $\mathcal{R}_8 : p(Y_1) \leftarrow_{\texttt{prod}} (0.8 \&_{\texttt{luka}} r(Y_1)) \&_{\texttt{G}} r(a)$ with $\alpha = 0.8$.

Moreover, by performing a new admissible step with the second rule of Definition 3.1 on the body of rule $\mathcal{R}_7$, we obtain the new unfolded rule $\mathcal{R}_9 : p(a) \leftarrow_{\texttt{prod}} (0.7 \&_{\texttt{prod}} 0.9) \&_{\texttt{G}} 0.7$ with $\alpha = 0.8$. So, the final program is the set of rules $\{\mathcal{R}_3, \mathcal{R}_4, \mathcal{R}_5, \mathcal{R}_6, \mathcal{R}_8, \mathcal{R}_9\}$. It is important to note that the application of this last rule to the goal proposed in Example 3.3 simulates the effects of the first four admissible steps shown in the derivation of the same example, which evidences the improvements achieved by operational unfolding on transformed programs.

4.2 Interpretive Unfolding

The present section defines the notion of interpretive unfolding. This kind of unfolding is devoted to accelerate truth degrees calculations during the second, interpretive, phase of the procedural semantics. Since in Definition 3.4 we have opted for a procedural characterization of the interpretive phase, (by formalizing it in terms of an state transition system) thus avoiding the use of semantic concepts (which were necessary, for instance, in [13] and [7]),

this fact has strongly helped us to clarify the formalization of our interpretive unfolding rule as follows.

Definition 4.3 (Interpretive Unfolding) *Let $\mathcal{P}$ be a program and let $\mathcal{R} : (A \leftarrow_i \mathcal{B}$ with $\alpha = v) \in \mathcal{P}$ be a (non unit) program rule. Then, the interpretive unfolding of rule $\mathcal{R}$ in program $\mathcal{P}$ with respect to the lattice $\langle L, \preceq \rangle$ associated to $\mathcal{P}$ is the new program $\mathcal{P}' = (\mathcal{P} - \{\mathcal{R}\}) \cup \{A \leftarrow_i \mathcal{B}'$ with $\alpha = v'\}$ such that:*

`IU1` *if expression $r_1 @ r_2$ appear in $\mathcal{B}$ then $\mathcal{B}' = \mathcal{B}[r_1 @ r_2 / [\![@]\!](r_1, r_2)]$, where @ is a connective, and $v' = v$;*

`IU2` *if $\mathcal{B} = r$, where $r \in L$, then $\mathcal{B}'$ is empty and $v' = [\![\&_i]\!](v, r)$, where $(\leftarrow_i, \&_i)$ is an adjoint pair in $\langle L, \preceq \rangle$.*

Observe that the first variant of interpretive unfolding (`IU1`), simply consists in applying an interpretive step on the body of a rule. In this sense, an alternative formalization, more similar to Definition 4.1, but replacing the use of $\rightarrow_{AS}$ by $\rightarrow_{IS}$, might be: $\mathcal{P}' = (\mathcal{P} - \{\mathcal{R}\}) \cup \mathcal{U}$ where $\mathcal{U} = \{A \leftarrow_i \mathcal{B}'$ with $\alpha = v \mid \langle \mathcal{B}; id \rangle \rightarrow_{IS} \langle \mathcal{B}'; id \rangle\}$. In fact, both formulations simply consists in replacing a program rule $\mathcal{R}$ whose body contains a connective @, by an analogous rule, with the same truth degree, but with the calculated truth degree of @ (w.r.t. the lattice associated to the program) in its body. Anyway, it is important to contrast the `IU1` transformation (in any of its alternative formats), with the T-Norm replacement rule of [8], since our new transformation compacts in a single formulation three low-level variants of this primitive transformation.

Focusing now in the `IU2` case, we observe that the second format proposed before for formalizing `IU1`, can not be applied now: not only the truth degree of the transformed rule differs from the original one, but also, and what is better, the `IU2` transformation is able to simplify program rules by directly eliminating its bodies, and hence, producing facts.

The following example illustrates the application of interpretive unfolding and some of their advantages.

Example 4.4 *Let's perform now some interpretive unfolding steps on the rules obtained by operational unfolding in Example 4.2. By interpretive unfolding -of kind* `IU1`*- of rule $\mathcal{R}_9$ (note that $[\![\&_{\mathtt{prod}}]\!](0.9, 0.7) = 0.63$) we obtain the new unfolded rule $\mathcal{R}_{10} : p(a) \leftarrow_{\mathtt{prod}} 0.63 \&_{\mathtt{G}} 0.7$ with $\alpha = 0.8$. Moreover, by applying a new* `IU1` *interpretive unfolding step on this last rule, we obtain $\mathcal{R}_{11} : p(a) \leftarrow_{\mathtt{prod}} 0.63$ with $\alpha = 0.8$. Finally, rule $\mathcal{R}_{11}$ becomes the fact $\mathcal{R}_{12} : p(a) \leftarrow_{\mathtt{prod}}$ with $\alpha = 0.504$ after a final* `IU2` *interpretive unfolding step. So, the final program is the set of rules $\{\mathcal{R}_3, \mathcal{R}_4, \mathcal{R}_5, \mathcal{R}_6, \mathcal{R}_8, \mathcal{R}_{12}\}$ and now the derivation shown in example 3.3 can reduce its length in six steps thanks to the use of clause $\mathcal{R}_{12}$. More exactly, we have avoided three admissible and three interpretive steps, thanks to the fact that rule $\mathcal{R}_{12}$ comes from $\mathcal{R}_1$ after having been modified by three operational plus three interpretive unfolding operations. Again, this shows the improvements achieved by the combined use of operational/interpretive unfolding, on transformed programs.*

5 Properties of the Transformations

In this section, we formalize and prove the best properties one can expect of a transformation system like the our, which is based on the two kinds of unfolding described before. Namely,

- on the theoretical side, the total correspondence between i.c.a.'s for goals executed against original/transformed programs, and
- on the practical side, the gains in efficiency on unfolded programs by reducing the number of (both, admissible and interpretive) steps needed to solve a goal.

Before presenting our combined, global result, we proceed separately with the particular properties of each kind of unfolding. We start by recalling from [7] the benefits of using operational unfolding in isolation.

Theorem 5.1 (Strong Correctness of Operational Unfolding) *Let $\mathcal{P}$ be a program, and let $\mathcal{Q}$ be a goal. If $\mathcal{P}'$ is a program obtained by operational unfolding of $\mathcal{P}$, then, $\langle \mathcal{Q}; id \rangle \rightarrow_{AS}^{n} \langle Q'; \theta \rangle$ in $\mathcal{P}$ iff $\langle \mathcal{Q}; id \rangle \rightarrow_{AS}^{m} \langle Q'; \theta' \rangle$ in $\mathcal{P}'$, where*

1. $\mathcal{Q}'$ *does not contain atoms,*

2. $\theta = \theta'[\mathcal{V}ar(\mathcal{Q})]$, *and*

3. $m \leq n$.

The proof of this theorem is detailed in [7]. It is important to note that the main advantages of operational unfolding can be already appreciated during the first (operational or admissible) phase of goal executions. The first two claims of the theorem imply the exact correspondence between a.c.a.'s in both programs, which also implies that i.c.a.'s are preserved with independence of the lattice $\langle L, \preceq \rangle$ used to interpret the a.c.a.'s. Besides this, profit is also achieved in efficiency, by diminishing the length of admissible derivations (claim 3).

Now we proceed with interpretive unfolding, where we obtain the counterpart of the previous theorem. Advantages in this case are only appreciated during the second (interpretive) phase of goal executions. In this sense, although we can not properly speak about a.c.a.'s preservation, we prove that it is possible to maintain the set of i.c.a's associated to a given goal (when a.c.a.'s are interpreted with respect to the same lattice used during the interpretive unfolding process). Regarding the reduction of the length of derivation in transformed programs, interpretive unfolding is able to reduce the number of interpretive steps needed to solve a goal, similarly as operational unfolding did w.r.t. admissible steps.

Theorem 5.2 *(***Strong Correctness of Interpretive Unfolding)** *Let $\mathcal{P}$ be a program and let $\mathcal{Q}$ be a goal. If $\mathcal{P}'$ is a program obtained by interpretive unfolding of $\mathcal{P}$, then, $\langle \mathcal{Q}; id \rangle \rightarrow^{n}_{AS/IS} \langle r; \theta \rangle$ in $\mathcal{P}$, iff $\langle \mathcal{Q}; id \rangle \rightarrow^{m}_{AS/IS} \langle r; \theta \rangle$ in $\mathcal{P}'$, where*

1. $r \in L$, *being $\langle L, \preceq \rangle$ the lattice associated to $\mathcal{P}$ used during the interpretive unfolding process, and*

2. $m \leq n$.

In order to prove this theorem, we treat separately both claims of the double implication.

Strong Completeness ($\Rightarrow$).
Let $\mathcal{D} : [\langle Q; id \rangle \rightarrow^{k}_{AS} \langle e; \theta \rangle \rightarrow^{l}_{IS} \langle r; \theta \rangle]$, where $k+l = n$, be the (generic) complete derivation for $\mathcal{Q}$ in $\mathcal{P}$ that we plan to simulate by constructing a new derivation $\mathcal{D}'$ in $\mathcal{P}'$. Consider also the rule $\mathcal{R} : (A \leftarrow_i \mathcal{B}[r_1 @ r_2]$ with $\alpha = v) \in \mathcal{P}$ such that, by interpretive unfolding of $\mathcal{R}$ in program $\mathcal{P}$, we obtain $\mathcal{R}' : (A \leftarrow_i \mathcal{B}[r_1 @ r_2 / [\![@]\!](r_1, r_2)]$ with $\alpha = v)$. Remember that $\mathcal{R} \in \mathcal{P}$ and $\mathcal{R}' \in \mathcal{P}'$, but $\mathcal{R} \notin \mathcal{P}'$ and $\mathcal{R}' \notin \mathcal{P}$. Since interpretive unfolding only affects expressions with connectives and elements belonging to L, the set of atoms in the heads and bodies of both $\mathcal{R}$ and $\mathcal{R}'$ are exactly the same. Moreover, since interpretive steps are not dependent of any kind of selection function (or computation rule), we can assume w.l.o.g. that the first steps in the interpretive phase in $\mathcal{D}$ are applied to each expression of the form $r_1 @ r_2$ introduced in e by previous admissible steps done with rule $\mathcal{R}$. That is, we can safely suppose that derivation $\mathcal{D}$ has the form $\mathcal{D} : [\langle Q; id \rangle \rightarrow^{k}_{AS} \langle e; \theta \rangle \rightarrow^{l_1}_{IS} \langle e'; \theta \rangle \rightarrow^{l_2}_{IS} \langle r; \theta \rangle]$, with $l_1 + l_2 = l$. This implies that we can easily construct the following admissible derivation $\mathcal{D}' : [\langle Q; id \rangle \rightarrow^{k}_{AS} \langle e'; \theta \rangle]$ in $\mathcal{P}'$, where:

- the length of $\mathcal{D}'$ coincides with the number of admissible steps, k, applied in $\mathcal{D}$,
- the atom reduced in the i-th step of $\mathcal{D}'$ coincides with the atom reduced in the i-th step of $\mathcal{D}$, for $1 \leq i \leq k$, and
- the rule used in the i-th step of $\mathcal{D}'$ is the same that the one used in the i-th step of $\mathcal{D}$, for $1 \leq i \leq k$, except when this last one is $\mathcal{R}$: in this case, we use $\mathcal{R}'$ in $\mathcal{D}'$.

Observe that the a.c.a.'s associated to both derivations are not exactly the same (which reveals that interpretive unfolding is not able to preserve a.c.a.'s, as operational unfolding does) but they are strongly related: both share the same substitution θ, whereas expressions e and e' are very similar. In fact, any admissible step done with rule $\mathcal{R}'$ in $\mathcal{D}'$, introduces a (just interpreted) value of the form $[\![@]\!](r_1, r_2)$ in e', whereas the corresponding steps done with rule $\mathcal{R}$ in $\mathcal{D}$, leaves descendants of the (non yet interpreted) expression $r_1 @ r_2$ in e. Formally,

if P_j is the set of positions of the j occurrences of $r_1@r_2$ introduced in e by the application of j admissible steps using $\mathcal{R}$ in $\mathcal{D}$ (or, equivalently, P_j is the set of positions of the j occurrences of $[\![@]\!](r_1, r_2)$ introduced in e' by the application of j admissible steps using $\mathcal{R}'$ in $\mathcal{D}'$), then $e' = e[r_1@r_2/[\![@]\!](r_1, r_2)]_{P_j}$. Hence, e' can be seen as a *partially interpreted* version of e, and then it is easy to see that, by simply applying several interpretive steps on the corresponding j occurrences of $r_1@r_2$ in e, we can replace them by $[\![@]\!](r_1, r_2)$, until reaching the intended expression e'. From here, we can finish the complete derivations in both programs by applying the same interpretive steps, until obtaining the same i.c.a. $\langle r, \theta \rangle$.

Regarding the reduction of the length of the derivation on transformed programs, we have seen that any step done with the unfolded rule $\mathcal{R}'$ in derivation $\mathcal{D}'$, avoids a later interpretive step which, on the other hand, is unavoidable when building a derivation using rules of the original program $\mathcal{P}$. So, the complete derivation simulating $\mathcal{D}$ in $\mathcal{P}'$ has the form: $[\langle Q; id \rangle \rightarrow^{k}_{AS} \langle e'; \theta \rangle \rightarrow^{l_2}_{IS} \langle r; \theta \rangle]$, where as we said $l_2 \leq l$, which implies that $m = k + l_2 \leq k + l = n$ what completes the proof of the strong completeness.

Strong Soundness ($\Leftarrow$).
It is perfectly analogous (even easier to prove) to the previous case.

Finally, the strong correctness of interpretive unfolding follows from both, the strong soundness ($\Leftarrow$) and the strong completeness ($\Rightarrow$), as we wanted to prove.

To finish this section, we present the following result which combines the use of operational/interpretive unfolding by considering a transformation sequence of programs $(\mathcal{P}_0, \ldots, \mathcal{P}_k)$, $k \geq 0$. The following theorem formalizes the best properties of the resulting transformation system, namely, its strong correctness and the guarantee for producing improvements on residual programs. The whole result directly follows as a simple corollary from Theorems 5.1 and 5.2.

Theorem 5.3 *(**Strong Correctness of the Transformation System**) Let $(\mathcal{P}_0, \ldots, \mathcal{P}_k)$ be a transformation sequence where each program in the sequence, except the initial one $\mathcal{P}_0$, is obtained from the immediately preceding one by applying operational/interpretive unfolding. Then, $\langle Q; id \rangle \rightarrow^{n}_{AS/IS} \langle r; \theta \rangle$ in $\mathcal{P}_0$ iff $\langle Q; id \rangle \rightarrow^{m}_{AS/IS} \langle r; \theta' \rangle$ in $\mathcal{P}_k$, where*

1. *$r \in L$, being $\langle L, \preceq \rangle$ the lattice associated to $\mathcal{P}$ used during the interpretive unfolding process,*
2. *$\theta' = \theta[\mathcal{V}ar(Q)]$, and*
3. *$m \leq n$.*

6 Conclusions and Future Work

The present paper must be seen as a final step in the development of the research line we started in [8] and continued in [7], where we have tried to adapt and to study the role played by a classical transformation rule like unfolding, in the setting of fuzzy logic programming whit labeled rules. In our investigations, we have dealt with different fuzzy logic programming languages sharing all them the common feature that they are based on program clauses/rules with "weights" expressing the truth degree or confidence factor one may have in their application. The present paper resumes and improves all our contributions in this research line, by considering one of the most recent and flexible languages in the field [13]. We have highlighted that, our unfolding-based transformation rules for multi-adjoint logic programs, inherit both the simplicity and computational power of the original language. When formalizing extended versions of previous unfolding-based transformation rules, we have been able now of avoiding instrumental and noisy elements like intermediate languages, computation rules, independence results, and so on. It is important to note that, all the results presented in this paper are also applicable to the Fuzzy Prolog of [5][3], with the

[3] In this approach, the interpretive phase is modeled in terms of "constraint solving", which similarly to ours avoids the use of computation rules and other related perturbations.

advantage that an implementation for the language is already available in this case. (we are planning to implement our unfolding transformations on such platform).

For the future, there exist many topics to undertake, closely connected with this research line: fuzzy unfolding semantics, fuzzy variants of other transformation rules like folding, and partial evaluation techniques applied to reductants calculi. We are also planning to extend our unfolding rules to fuzzy logic languages that, instead on weighted rules, be based on similarity relations [2, 15]. In this sense, we hope to take advantage of the (partial) correspondences between both kind of languages analyzed in [13].

Acknowledgements We are grateful to Susana Muñoz, for providing us free access to worthy material, and the anonymous referees by their suggestive judgments.

References

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

[2] F. Arcelli and F. Formato. Likelog: A logic programming language for flexible data retrieval. In *Proc. of SAC'99*, pp. 260–267. ACM, Artificial Intelligence and Computational Logic, 1999.

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

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

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

[6] M. Ishizuka and N. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In *Proc. of IJCAI'85*, pp. 701–703, 1985.

[7] P. Julián, G. Moreno, and J. Penabad. On Fuzzy Unfolding. A Multi-Adjoint Approach. *Fuzzy Sets and Systems*, pp. 22, 2005. Accepted for publication.

[8] P. Julián, G. Moreno, and J. Penabad. Unfolding-based Improvements on Fuzzy Logic Programs. In *ENTCS*. pp. 32, 2005. Accepted for publication.

[9] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In *Foundations of Deductive Databases and Logic Programming*, pp. 587–625, 1988.

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

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

[12] J. Medina, M. Ojeda, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. *Proc. of LPNMR'01*, LNAI 2173:351–364, 2001.

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

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

[15] M.I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Fuzzy Sets and Systems*, 275:389–426, 2002.

[16] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Proc. of ICLP'84*, pp. 127–139, 1984.

[17] P. Vojtáš and L. Paulík. Soundness and completeness of non-classical extended SLD-resolution. In *Proc. ELP'96*, pp. 289–301. LNCS 1050, 1996.