Journal of Universal Computer Science, vol. 12, no. 11 (2006), 1679-1699
submitted: 1/5/06, accepted: 15/10/06, appeared: 28/11/06 © J.UCS

Operational/Interpretive Unfolding of Multi-adjoint Logic Programs

Pascual Julián, Ginés Moreno and Jaime Penabad
University of Castilla-La Mancha, Spain
{Pascual.Julian,Gines.Moreno,Jaime.Penabad}@uclm.es

Abstract: Multi-adjoint logic programming represents a very recent, extremely flexible attempt for introducing fuzzy logic into logic programming. 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 logic programming, thus returning a computed substitution together with an expression where all atoms have been exploited. This last expression is then interpreted under a given lattice during 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.

Key Words: fuzzy logic programming, program transformation, unfolding
Category: D.1.6, I.2.2, I.2.3

1 Introduction

Multi-adjoint logic programming [Medina et al.2001a, Medina et al.2004] is an extremely flexible framework combining fuzzy logic and logic programming, which largely improves older approaches previously introduced in this field (see, for instance, the Prolog–Elf system of [Ishizuka and Kanai1985], the Fril Prolog system of [Baldwin et al.1995] and the F–Prolog languages of [Li and Liu1990, Vojtáš and Paulík1996, Arcelli and Formato1999], and [Guadarrama et al.2004], where different fuzzy variants of Prolog have been proposed). The fuzzy dialect of Prolog presented in [Guadarrama et al.2004] deserves a special mention since, at least syntactically, it is very close to the language used here. However, we find a slight difference: whereas the multi-adjoint approach is based on "weighted" clauses whose truth degrees are elements of any appropriate lattice, in the Fuzzy Prolog of [Guadarrama et al.2004], truth degrees are based on Borel Algebras (union intervals).

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, given a program, goals are evaluated in two separate computational phases. During the *operational* one, *admissible steps* (a generalization of the classical *modus ponens* inference rule) are systematically applied by a backward reasoning procedure in a similar way to classical resolution steps in pure logic programming, thus returning a computed substitution together with an expression where all atoms have been exploited. This last expression is then interpreted under a given lattice during what we call the *interpretive* phase, hence returning a pair $\langle$*truth degree*; *substitution*$\rangle$ which is the fuzzy counterpart of the classical notion of computed answer traditionally used in logic programming.

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 [Burstall and Darlington1977] to optimize functional programs and then used for logic programs [Tamaki and Sato1984] and (lazy) functional logic programs [Alpuente et al.2004]. 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 [Pettorossi and Proietti1996]. 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 [Julián et al.2005a]) an the interpretive. It is important to remark that in the original (multi-adjoint logic) language proposed in [Medina et al.2004] and then used in [Julián et al.2005a], 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 [Julián et al.2005a], 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 [Julián et al.2005b] 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 12, 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 lost in [Julián et al.2005a] and [Julián et al.2005b], where some auxiliary intermediate languages (with complex constructors and intricate 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. While considering some implementation issues, we analyze in Section 6 some related works. Finally, we end in Section 7, by giving our conclusions and proposing some lines of 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 [Julián et al.2005a] we start from the scratch by manipulating an extended version of the *multi-adjoint logic programing* language presented in [Medina et al.2001a, Medina et al.2004]. We refer 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. In our fuzzy setting, we use implication connectives $(\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m)$ and also other connectives which are grouped under the name of "aggregators" or "aggregation operators". They are used to combine/propagate truth values through the rules. The general definition of aggregation operators subsumes conjunctive operators (denoted by $\&_1, \&_2, \ldots, \&_k$), disjunctive operators $(\vee_1, \vee_2, \ldots, \vee_l)$, and average and hybrid operators (usually denoted by $@_1, @_2, \ldots, @_n$). Although the connectives $\&_i$, $\vee_i$ and $@_i$ are binary operators, we usually generalize them as functions with an arbitrary number of arguments. In the following, we often write $@(x_1, \ldots, x_n)$ instead of $@(x_1, @(x_2, \ldots, @(x_{n-1}, x_n) \ldots))$. Aggregation operators are useful to describe/specify user preferences. An aggregation operator, when interpreted as a truth function, may be an arithmetic mean, a wighted 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[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 to be governed by universal quantifiers.

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

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 [Medina et al.2001a, Medina et al.2004], 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.

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 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*:

[2] Let $mgu(E)$ denote the *most general unifier* of an equation set E (see [Lassez et al.1988] for a formal definition of this concept).

Rule 1.

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

Rule 2.

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

Rule 3.

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

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 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 *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 se-*

lected expression in each admissible step:

$\langle \underline{p(X)} \&_{\mathtt{G}} r(a);\ id\rangle \rightarrow_{AS1}{}^{\mathcal{R}_1}$

$\langle (0.8 \&_{\mathtt{prod}} (\underline{q(X_1, Y_1)} \&_{\mathtt{G}} r(Y_1))) \&_{\mathtt{G}} r(a);\ \{X/X_1\}\rangle \rightarrow_{AS1}{}^{\mathcal{R}_2}$

$\langle (0.8 \&_{\mathtt{prod}} ((0.7 \&_{\mathtt{prod}} \underline{s(Y_2)}) \&_{\mathtt{G}} r(Y_2))) \&_{\mathtt{G}} r(a);\ \{X/a, X_1/a, Y_1/Y_2\}\rangle \rightarrow_{AS2}{}^{\mathcal{R}_5}$

$\langle (0.8 \&_{\mathtt{prod}} ((0.7 \&_{\mathtt{prod}} 0.9) \&_{\mathtt{G}} \underline{r(b)})) \&_{\mathtt{G}} r(a);\ \{X/a, X_1/a, Y_1/b, Y_2/b\}\rangle \rightarrow_{AS2}{}^{\mathcal{R}_4}$

$\langle (0.8 \&_{\mathtt{prod}} ((0.7 \&_{\mathtt{prod}} 0.9) \&_{\mathtt{G}} 0.7)) \&_{\mathtt{G}} \underline{r(a)};\ \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b\}\rangle \rightarrow_{AS2}{}^{\mathcal{R}_4}$

$\langle (0.8 \&_{\mathtt{prod}} ((0.7 \&_{\mathtt{prod}} 0.9) \&_{\mathtt{G}} 0.7)) \&_{\mathtt{G}} 0.7;\ \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b, Y_4/a\}\rangle.$

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

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 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 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 6 *We complete the previous derivation of Example 3 by executing the necessary interpretive steps to obtain the interpretive computed answer (i.c.a.) with respect to lattice* $([0,1],\leq)$.

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

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 logic programming 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 [Pettorossi and Proietti1996].

As detailed in [Julián et al.2005a], we have adapted this transformation to deal with multi-adjoint logic programs by defining it in terms of operational steps (see Definition 1). Also, in [Julián et al.2005a], 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 [Julián et al.2005a], 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 7 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 [Tamaki and Sato1984], 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 shows us that the body of transformed rules also contains 'compiled–in' information coming from both components of a partial computed answer (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 8 *Consider again program $\mathcal{P}$ shown in Example 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 1) generates the new program $(\mathcal{P} - \{\mathcal{R}_2\}) \cup \{\mathcal{R}_6\}$, where $\mathcal{R}_6$ is the new unfolded rule $q(a,b) \leftarrow_{\mathtt{prod}} 0.9$ 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:

$$\langle \underline{q(X,Y)} \&_{\mathtt{G}} r(Y); id \rangle \rightarrow_{AS1}{}^{\mathcal{R}_6} \langle (0.7 \&_{\mathtt{prod}} 0.9) \&_{\mathtt{G}} r(b); \{X/a, Y/b\} \rangle,$$
$$\langle \underline{q(X,Y)} \&_{\mathtt{G}} r(Y); id \rangle \rightarrow_{AS1}{}^{\mathcal{R}_3} \langle (0.8 \&_{\mathtt{luka}} r(Y_1)) \&_{\mathtt{G}} r(a); \{X/Y_1, Y/a\} \rangle.$$

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

Moreover, by performing a new admissible step with the second rule of Definition 1 on the body of rule $\mathcal{R}_7$, we obtain the new unfolded rule $\mathcal{R}_9 : p(a) \leftarrow_{\mathtt{prod}} (0.7 \&_{\mathtt{prod}} 0.9) \&_{\mathtt{G}} 0.7$ 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 simulates the effect 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 degree calculations during the second, interpretive, phase of the procedural semantics. In Definition 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 [Medina et al.2004] and [Julián et al.2005a]). This fact allows us to clarify the formalization of our interpretive unfolding rule as follows.

Definition 9 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:

[IU 1]. 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 $v' = v$.

[IU 2]. 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 the interpretive unfolding rule (IU1), simply consists in applying an interpretive step on the body of a rule. In this sense, an alternative formalization, more similar to Definition 7, 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 compare the IU1 transformation (in any of its alternative formats), with the T-Norm replacement rule of [Julián et al.2005b], 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 alternative 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 10 *Let's perform now some interpretive unfolding steps on the rules obtained by operational unfolding in Example 8. 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* $\mathcal{R}_{11} : p(a) \leftarrow_{\texttt{prod}} 0.63$ *with* $\alpha = 0.8$. *Finally, rule* $\mathcal{R}_{11}$ *becomes the fact* $\mathcal{R}_{12} : p(a) \leftarrow_{\texttt{prod}}$ *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 can reduce its length in six steps thanks to the use of clause* $\mathcal{R}_{12}$, *as follows:*

$$\begin{aligned}
&\langle \underline{p(X)} \&_{\texttt{G}} r(a);\ id\rangle \ \rightarrow_{AS2}{}^{\mathcal{R}_{12}}\\
&\langle (0.504 \&_{\texttt{G}} \underline{r(a)});\ \{X/a\}\rangle \ \rightarrow_{AS2}{}^{\mathcal{R}_4}\\
&\langle \underline{0.504 \&_{\texttt{G}} 0.7}; \{X/a\}\rangle \ \rightarrow_{IS}\\
&\langle 0.504; \{X/a\}\rangle
\end{aligned}$$

Observe that 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 ours, 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 [Julián et al.2005a] the benefits of using operational unfolding in isolation.

Theorem 11 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,* $\langle \mathcal{Q}; id\rangle \rightarrow^n_{AS} \langle Q'; \theta\rangle$ *in* $\mathcal{P}$ *iff* $\langle \mathcal{Q}; id\rangle \rightarrow^m_{AS} \langle Q'; \theta'\rangle$ *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 [Julián et al.2005a]. 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 the interpretive unfolding, where we obtain the correlate 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 12 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. *$m \leq n$.*

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

Strong Soundness ($\Leftarrow$). Let $\mathcal{D}' : [\langle Q; id \rangle \rightarrow^k_{AS} \langle e'; \theta \rangle \rightarrow^l_{IS} \langle r; \theta \rangle]$, where $k + l = m$, be the (generic) complete derivation for $\mathcal{Q}$ in $\mathcal{P}'$ that we plan to simulate by using rules of program $\mathcal{P}$. Consider also that rule $\mathcal{R}' : (A \leftarrow_i \mathcal{B}[r_1 @ r_2 / [\![@]\!](r_1, r_2)]$ with $\alpha = v)$ has been obtained by interpretive unfolding of rule $\mathcal{R} : (A \leftarrow_i \mathcal{B}$ 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. This implies that we can safely construct the following complete 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: $[\langle Q; id\rangle \rightarrow^k_{AS} \langle e; \theta\rangle \rightarrow^j_{IS} \langle e'; \theta\rangle \rightarrow^l_{IS} \langle r; \theta\rangle]$, where $k+j+l = n$, which implies that $m \leq n$ (remember that $m = k+l$) as we wanted to prove.

Strong Completeness ($\Rightarrow$). Although this direction can be proved in a similar way to the previous one, we prefer to detail it since it introduces some subtleties regarding the order in which computation steps are performed in the original derivation. 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. $\langle r, \theta\rangle$.

Similarity to the soundness proof, the length of the derivation on transformed programs is reduced. 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.
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 11 and 12.

Theorem 13 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* $\mathcal{P}_0$ *used during the interpretive unfolding process,*
2. $\theta' = \theta[\mathcal{V}ar(Q)]$*, and*
3. $m \leq n$*.*

6 Related Work and Implementation Issues

Although unfolding is a classical transformation rule which is well-known in other declarative paradigms (pure functional, pure logic and integrated functional-logic languages), to the best of our knowledge there exist no precedents of such technique (apart from our approaches in [Julián et al.2004, Julián et al.2005b] and [Julián et al.2005a]) in the specialized literature related to fuzzy logic programming. The present work, culminates the efforts we initiated three years ago, by providing a simple, clean and powerful scheme for unfolding fuzzy logic programs with a high level of flexibility.

At the very beginning of our developments, when we analyzed different fuzzy variants of Prolog in order to adapt the classical unfolding transformation to the new paradigm,we found two major and rather different approaches based on similarity relations and weighted rules, respectively. The first approach, represented by languages such as Likelog [Arcelli and Formato1999], replaces the syntactic unification mechanism of classical SLD-resolution by a fuzzy unification algorithm, based on similarity relations (over constants and predicates). In this context, the fuzzy unification algorithm provides an extended most general unifier as well as a numerical value, called *unification degree*. Intuitively, the unification degree represents the truth degree associated with the (query) computed instance. Programs written in this kind of languages consist, in essence, in a set of ordinary (Prolog) clauses jointly with a set of "similarity equations" which plays an important role during the unification process. Unfortunately, when we try to unfold a Likelog program, it is mandatory to extend the language if we really want to code the residual program. Both, the syntax and the operational

principle require important manipulations in its design to cope with the elements that unfolding introduces in the body of transformed clauses. This fact prevented us from developing our work in this direction.

For the second approach, fuzzy logic programs are sets of weighted formulas, where the *truth degree* of each clause is explicitly annotated. The task of computing and propagating truth degrees relies on an extension of the resolution principle, whereas the (syntactic) unification mechanism remains untouched. Examples of this kind of languages are the pioneer one described in [Vojtáš and Paulík1996], and the most modern and flexible ones shown in [Medina et al.2004] and [Guadarrama et al.2004]. In contrast with Likelog, fuzzy languages based on weighted rules admit, in general, a much more natural formalization of unfolding, without the need of modifying the procedural mechanism (even when an extended syntax is needed for coding transformed programs), as we have shown in the following previous works:

- In [Julián et al.2004] we proposed our first attempt of operational unfolding for programs written with the language of [Vojtáš and Paulík1996], where all clauses of a given program obey the same fuzzy logic.
- The language used in our second approach, [Julián et al.2005b], enriches the previous one by allowing the use of different fuzzy logics inside the same program. Now, operational unfolding is complemented with a set of low-level, interpretive-based, transformation rules called T-Norm replacement.
- In [Julián et al.2005a] we try once more again to reinforce the power of the underlying language by adopting the extremely flexible multi-adjoint approach of [Medina et al.2004] (where, among other extensions, it allows to cohabit different fuzzy logics even inside a same program rule) when defining an unfolding rule strictly based on operational steps.

The present work completes the research line initiated in [Julián et al.2005a] by introducing a complement of the unfolding rule in terms of interpretive steps. Doing this, we have clarified the proper notion of interpretive step and we have removed noisy instrumental elements such as selection functions or independence results. In our context, one remarkable fact is that, for the first time, the requirement of using and auxiliary extended syntax for coding residual programs is dropped out. This allow us to recover, in an elegant way, the classical source-to-source nature of the unfolding transformation.

Regarding the Fuzzy Prolog language of [Guadarrama et al.2004], we have just recognized (see for instance the introduction section) that its level of flexibility and expressiveness are perfectly comparable with the one obtained by the multi-adjoint approach and, what is better, an interpreter conceived using Constraint Logic Programming over real numbers ($CLP(\mathcal{R})$) has been efficiently im-

plemented (where source programs are translated into directly executable CLP-based Prolog code). However and similarly to the case of Likelog, the adequacy of this language for being used as the basis of an unfolding rule is rather limited, as we are going to explain. The real problem does not appear only at the syntactic level (although we have solved similar difficulties for other languages like in [Julián et al.2004, Julián et al.2005b]), but what is worse, the major inconvenience is the need for redefining the core of its procedural mechanism to cope with constraints possibly mixed with atoms.

The notion of multi-adjoint lattice (as well as the use of aggregation operations and truth degrees) has a direct correspondence with the notion of constraint domain in the $CLP(\mathcal{R})$ representation of [Guadarrama et al.2004]. Computation steps are described by means of an state transition system where, instead of two elements, each state contains three components $\langle atoms, substitution, constraints\rangle$. Initial states have the first component (input) fulfilled with a set of atoms of a given goal and the two last components (outputs) are empty. Vice versa, in final states the goal component is empty whereas the two last ones represents the fuzzy computed answer (substitution and truth degree) for the original goal. Remember that the notion of state used in the present work avoids the last component since atoms, aggregators and truth degrees can safely cohabit (as part of an extended language) inside the first component of an state, and also in the body of (transformed) program rules, which enables the effective definition of unfolding in our setting. Conversely, in the language of [Guadarrama et al.2004], the strict separation of atoms and constraints (in both, computation states and clause bodies) represents a severe obstacle for the adaptation of our notion of unfolding rule (it is neither easy to execute nor to code on the body of unfolded clauses the constraints generated by those computation steps performed at transformation time).

On the other hand, the approach of [Guadarrama et al.2004] represents a real and interesting inspiration for implementation issues, specially taking into account that there is not yet available an interpreter for the language originally described in [Medina et al.2004] that we have been using in this work. In fact, due to the parallelism of both proposals, the multi-adjoint language also admits a "constraint solving"-based implementation when considering Borel lattices (i.e., union of intervals of real numbers), by simply following the guidelines detailed in [Guadarrama et al.2004]. Moreover, if we focus in the simpler case of our examples (which uses a multi-adjoint lattice whose carrier set L is the real interval $[0, 1]$ and the connectives are collected from the product, Łukasiewicz and Gödel intuitionistic logic), then it is easy to translate program rules to (pure) Prolog code as follows:

• The role of aggregator operators can be easily played by standard Prolog clauses defining "aggregator predicates" as follows:

```
and_prod(X,Y,Z):- Z is X * Y.
and_godel(X,Y,Z):- (X=<Y,Z=X;X>Y,Z=Y).
and_luka(X,Y,Z):- H is X+Y-1,(H=<0,Z=0;H>0,Z=H).
```

• Each atom appearing in a fuzzy rule is translated into a Prolog atom extended with an extra argument, called *truth variable*, which is intended to contain the truth degree obtained after the subsequent evaluation of such atom.

• Program facts (i.e., fuzzy rules with no body) are expanded at compilation time to Prolog facts, where the additional argument of the (head) atom, instead of being a truth variable, is just the truth degree of the corresponding rule. For instance, rules $\mathcal{R}_4$ and $\mathcal{R}_5$ in Example 3, can be represented by the Prolog facts `r(Y,0.7)` and `s(b,0.9)`, respectively.

• Program rules are translated into Prolog clauses by performing the appropriate calls to the atoms presented in its body. Regarding the calls to aggregator predicates, they must be postponed at the end of the body, in order to guarantee that the truth variables used as arguments be correctly instantiated when needed. In this sense, it is also important to respect an appropriate ordering when performing the calls. In particular, the last call must necessarily be to the "aggregator predicate" modeling the adjoint conjunction of the implication operator of the rule, by also using its truth degree. For instance, rules $\mathcal{R}_1, \mathcal{R}_2$ and $\mathcal{R}_3$ in Example 3, can be represented by the Prolog clauses:

```
p(X,TV0)   :- q(X,Y,TV1),r(Y,TV2),
              and_godel(TV1,TV2,TV3), and_prod(0.8,TV3,TV0).
q(a,Y,TV0) :- s(Y,TV1),and_prod(0.7,TV1,TV0).
q(Y,a,TV0) :- r(Y,TV1),and_luka(0.8,TV1,TV0).
```

• A goal is translated into a Prolog goal where the corresponding calls appear in their textual order before the "aggregator predicates". Since aggregators are not associative in general, they must appear in an appropriate sequence, as also occurred with the translation of clause bodies explained before. For instance, the goal $\leftarrow p(X)\&_{\mathtt{G}} r(a)$ in Example 3, can be represented by the Prolog goal `?- p(X,TV1), r(a,TV2), and_godel(TV1,TV2,TV3).`

Following this method, we have just translated to standard Prolog code each one of the multi-adjoint logic programs shown in this paper. In particular, the Prolog version of the transformed program obtained in Example 10, contains the clauses previously seen defining aggregators and rules $\mathcal{R}_3$, $\mathcal{R}_4$ and $\mathcal{R}_5$, together with the following clauses representing respectively the unfolded rules $\mathcal{R}_{12}$, $\mathcal{R}_8$ and $\mathcal{R}_6$:

```
p(a,0.504).
p(Y,TV0)   :- r(Y,TV1),r(a,TV2),and_luka(0.8,TV1,TV3),
              and_godel(TV3,TV2,TV4),and_prod(0.8,TV4,TV0).
q(a,b,TV0) :- and_prod(0.7,0.9,TV0).
```

The experimental results we have obtained after the execution of such code in SICStus Prolog, confirm the benefits we have largely reported in this work. Nowadays, we are just implementing a prototype of the tool.

7 Conclusions and Future Work

The present paper can be seen as a final step in the development of the research line we started in [Julián et al.2004, Julián et al.2005b] and continued in [Julián et al.2005a], 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 of 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 condenses and improves all our contributions on this research line, by considering one of the most recent and flexible languages in the field [Medina et al.2004]. 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. The main contributions of the present paper can be summarized as follows:

- We have introduced the notion of interpretive unfolding, which greatly enhances the T-Norm replacement rules of [Julián et al.2005b] and complements the operational unfolding for multi-adjoint logic programs described in [Julián et al.2005a].
- In this setting, we have clarified the procedural semantics of the underlaying language, by formalizing its interpretive phase in terms of an state transition system.
- As a consequence, this is the first time that all our formalizations do not depend of auxiliary languages and other intermediate elements, which largely clarifies and empowers our approach.
- We have accompanied our fuzzy unfolding definitions with representative examples, correctness results (including formal proofs verifying the gains in efficiency that they produce on transformed programs) and links to related works.

For future work, there are many topics to undertake, closely connected with this research line. Some of them, in which we are working nowadays, are the definition of: fuzzy unfolding semantics and their equivalences with fix-point semantics, and fuzzy variants of other transformation rules like folding (see [Moreno2006] for some preliminary results). Also, in [Julián et al.2006] we have started a new research line introducing partial evaluation techniques applied to reductant calculi. *Reductants* are an useful theoretical tool introduced for proving correctness properties in the context of generalized annotated logic programming [Kefer and Subrahmanian1992]. This concept was adapted to the framework of multi-adjoint logic programming in [Medina et al.2001b, Medina et al.2004], aiming to solve a problem of incompleteness that arises when working with some lattices. In order to be complete, multi-adjoint logic programs must be extended with their set of reductants. In [Julián et al.2006] we provide an improved definition of reductant which uses (threshold) partial evaluation techniques and is able to obtain computed answers for a given goal with a lesser computational effort than by using other simpler formulations of reductants. In the near future we want to investigate the formal properties of partial evaluation (in the multi-adjoint framework) and reductants.

Finally, following the ideas sketched at the end of Section 6, we want to put in practice the transformation techniques developed in this paper.

Acknowledgments

We are grateful to Susana Muñoz for providing us free access to worthy material. We also thank to anonymous referees their suggestive judgments on fuzzy dialects of Prolog which helped us to largely improve this paper.

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

References

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

[Arcelli and Formato1999] Arcelli, F. and Formato, F. (1999). Likelog: A logic programming language for flexible data retrieval. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99), February 28 - March 2, 1999, San Antonio, Texas, USA*, pages 260–267. ACM, Artificial Intelligence and Computational Logic. Electronic Edition (DOI: 10.1145/298151.298348).

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

[Burstall and Darlington1977] Burstall, R. and Darlington, J. (1977). A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67.

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

[Ishizuka and Kanai1985] Ishizuka, M. and Kanai, N. (1985). Prolog-ELF Incorporating Fuzzy Logic. In Joshi, A. K., editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI'85). Los Angeles, CA, August 1985.*, pages 701–703. Morgan Kaufmann.

[Julián et al.2004] Julián, P., Moreno, G., and Penabad, J. (2004). Unfolding Fuzzy Logic Programs. In *Proc. of the Fourth International Conference on Intelligent Systems Design and Applications, ISDA'04 (Sponsored by IEEE). Budapest (Hungary), August 26-28*, pages 595–600.

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

[Julián et al.2005b] Julián, P., Moreno, G., and Penabad, J. (2005b). Unfolding-based Improvements on Fuzzy Logic Programs. In Lucas, S., editor, *Electronic Notes in Theoretical Computer Science*, volume 137, pages 69–103. Elsevier.

[Julián et al.2006] Julián, P., Moreno, G., and Penabad, J. (2006). Evaluación Parcial de Programas Lógicos Multi-adjuntos y Aplicaciones. In Fernández, A., editor, *Proc. of Campus Multidisciplinar en Percepción e Inteligencia, CMPI-2006, Albacete, Spain, July 10-14*, pages 712–724. University of Castilla-La Mancha.

[Kefer and Subrahmanian1992] Kefer, M. and Subrahmanian, V. (1992). Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12:335–367.

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

[Li and Liu1990] Li, D. and Liu, D. (1990). *A fuzzy Prolog database system.* John Wiley & Sons, Inc.

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

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

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

[Moreno2006] Moreno, G. (2006). Building a Fuzzy Transformation System. In Wiedermann, J., Tel, G., Pokorn, J., Bielikov, M., and Stuller, J., editors, *Proc. of the 32nd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM'2006. Merin, Czech Republic, January 21-27*, pages 409–418. Springer LNCS 3831.

[Pettorossi and Proietti1996] Pettorossi, A. and Proietti, M. (1996). Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414.

[Tamaki and Sato1984] Tamaki, H. and Sato, T. (1984). Unfold/Fold Transformations of Logic Programs. In Tärnlund, S., editor, *Proc. of Second Int'l Conf. on Logic Programming*, pages 127–139.

[Vojtáš and Paulík1996] Vojtáš, P. and Paulík, L. (1996). Soundness and completeness of non-classical extended SLD-resolution. In et al, R. D., editor, *Proc. ELP'96 Leipzig*, pages 289–301. LNCS 1050, Springer Verlag.