An Improved Reductant Calculus using Fuzzy Partial Evaluation Techniques

Pascual Julián [a] Ginés Moreno [b] Jaime Penabad [c]

[a] *Dept. of Information Technologies and Systems*
ESI, Univ. of Castilla–La Mancha
Paseo de la Universidad, 4;
13071 Ciudad Real, Spain

[b] *Dept. of Computing Systems*
[c] *Dept. of Mathematics*
EPSA, Univ. of Castilla–La Mancha
Campus Universitario, s/n;
02071 Albacete, Spain

Abstract

Partial evaluation (PE) is an automatic program transformation technique aiming to obtain, among other advantages, the optimization of a program with respect to parts of its input: hence, it is also known as *program specialization*. This paper introduces the subject of PE into the field of fuzzy logic programming. We define the concept of PE for multi-adjoint logic programs and goals, and apart from discussing the benefits achieved by this technique, we also introduce in the fuzzy setting a completely novel application of PE which allows us the computation of *reductants* guaranteeing *completeness* properties without harming the computational *efficiency*. *Reductants* are a special kind of fuzzy rules which constitute an essential theoretical tool for proving correctness properties. As observed in the specialized literature, a multi-adjoint logic program, when interpreted on a partially ordered lattice, has to include all its reductants in order to preserve the (approximate) completeness property. This introduces severe penalties in the implementation of efficient multi-adjoint logic programming systems: not only the size of programs increases but also their execution time. In this paper we define a refinement to the notion of reductant based on PE techniques, that we call *PE-reductant*. We establish the main properties of PE-reductants (i.e., the classical concept of reductant and the new notion of PE-reductant are both, *semantically* and *operationally*, equivalent) and, what is the best, we demonstrate that our refined notion of PE-reductant is even able to increase the efficiency of multi-adjoint logic programs.

Key words: Fuzzy Logic Programming, Partial Evaluation, Reductants

⋆ 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.

Email addresses: `Pascual.Julian@uclm.es` (Pascual Julián), `Gines.Moreno@uclm.es` (Ginés Moreno), `Jaime.Penabad@uclm.es` (Jaime Penabad).

1 Introduction

Many transformation techniques have been proposed in the literature, in order to improve program code. One of the best known is *Partial evaluation* (PE) [1], which also offers a unified framework for the study of compilers and interpreters. PE is an automatic program transformation technique aiming at the optimization of a program with respect to parts of its input: hence, it is also known as *program specialization*. It is expected that the specialized program (also called *residual* program or *partially evaluated* program) could be executed more efficiently than the original program. This is because the residual program is able to save some computations, at execution time, that were done only once at PE time. To fulfill this goal, PE uses symbolic computation as well as some techniques provided by the field of program transformation [2], specially the so called *unfolding* transformation. Unfolding is essentially the replacement of a call by its definition, with appropriate substitutions. In general, PE techniques include stop criteria to guarantee the termination of the PE process [1]. Therefore, PE is an automatic transformation technique (that is, the PE process can be completed without human intervention). This feature distinguishes PE from other program transformation techniques [2–4].

PE has been widely applied in the field of declarative languages: functional programming [5,1,6]; logic programming [7–9,3], where it is usually called *partial deduction*; and functional logic programming [10–12]. Also it has been applied to the area of imperative languages (e.g., the language C [13]). Although the objectives are similar, the general methods are often different due to the distinct underlying computational models. Techniques in conventional partial deduction of logic programs usually rely on the unification-based parameter propagation [14], which is part of the resolution principle. In this context, the input data are provided as partial data arguments in a goal (mainly an atomic formula).

Also, PE has been applied extensively to a variety of concrete problems, such as [1]: specialization of database queries; mechanical theorem proving; optimization of ray tracing procedures in the area of computer graphics; software maintenance and program understanding; specialization of simulators for training neuronal networks; and circuit simulators specialization.

On the other hand, *Multi-adjoint logic programming* [15–17] is an extremely flexible framework combining fuzzy logic and logic programming. Informally speaking, a multi-adjoint logic program can be seen as a set of rules each of which is annotated by a truth degree (a value of a complete lattice, for instance the real interval $[0, 1]$) and a query to the system is a goal plus a substitu-

[1] Controlling the PE process is an important task. However, it is out of the scope of this work.

tion (initially the identity substitution, denoted by id). Given a multi-adjoint logic program, goals are evaluated in two separate computational phases. During the *operational* phase, *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. More precisely, in an admissible step, for a selected atom A in a goal and a rule $\langle H \leftarrow \mathcal{B}; v \rangle$ of the program, if there is a most general unifier θ of A and H, the atom A is substituted by the expression $(v \& \mathcal{B})\theta$, where "&" is an adjoint conjunction evaluating *modus ponens*. Finally, the operational phase returns 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 [18], hence returning a pair $\langle$*truth degree*; *substitution*$\rangle$ which is the fuzzy counterpart of the classical notion of computed answer traditionally used in pure logic programming.

A multi-adjoint logic program, when interpreted on a partially ordered lattice, needs to incorporate a special kind of rules, called reductants, in order to preserve the (approximate) completeness property [17]. This introduces severe penalties in the implementation of efficient multi-adjoint logic programming systems. Because a multi-adjoint logic program has to include all its reductants, not only the size of programs increases but also their execution time. Therefore, if we want to develop complete, efficient implementation systems for the multi-adjoint logic framework, it is essential to define methods for optimizing the computation of reductants. In this work we apply PE techniques to achieve this goal. The idea is to diminish the negative impact of incorporating reductants to a multi-adjoint logic program, by means of a preprocess where reductants are partially evaluated before they are included in the target program: the PE phase produces a set of refined reductants; hence, the computational effort done (only once) during the generation time is avoided (many times) at execution time.

In this work, after recalling in Section 2 the main concepts of the multi-adjoint logic programming paradigm [15–17], being inspired by our own experience in the development of PE techniques for declarative programs [19,11] and program transformation rules for multi-adjoint logic programs [20,18], we deal with the following objectives:

- We define the concept of PE for the new fuzzy setting in Section 3. The idea is to adapt, for this richer framework, the techniques arisen around the field of partial deduction of pure logic programs [7,21,9], but incorporating the unfolding rule developed in [20,18] for this class of fuzzy logic programs. Following this path, we try to unfold admissible goals, as much as possible, in order to obtain an optimized (specialized) version of the original program.

- Using the partial evaluation techniques just developed, in Section 4 we give a more refined version of the concept of reductant considered in [17], which we call PE-reductant. We present its formal definition and then, we relate it with the classical concept of reductant and also with the notion of partial evaluation. We also informally discuss the benefits of the resulting technique by means of some representative examples.

- In Section 5 we establish the formal correctness properties of PE-reductants focusing in both, semantic and procedural aspects. These properties enjoyed by our PE technique, are formally proved and its benefits are easily evidenced with some elucidating examples. Moreover, we also confirm the reduction of the length of admissible derivations when using the improved notion of PE-reductant instead of the older one of reductant, which allows us to obtain more efficient residual programs in practice.
- Finally, before concluding in Section 7, we discuss in Section 6 some other different approaches appeared in the specialized literature. Our contrast is made at several levels, including language extensions, transformation techniques and implementation issues.

2 Preliminaries

This section summarizes the main features of multi-adjoint logic programming. We refer the interested reader to [15–17] for a complete formulation.

2.1 The multi-adjoint logic language

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 $\&_i$ and $\vee_i$ are binary operators, $@_i$ operators are usually functions with an arbitrary number of arguments. Also note that, we often abstract a complex expression, $\mathcal{E}$, containing $\&_i$, $\vee_i$ and $@_i$ operators, as simply writing $@(x_1, x_2, \ldots, x_n)$, where $\{x_1, x_2, \ldots, x_n\}$ is the set of distinct variables occurring in $\mathcal{E}$.

Aggregation operators are useful to describe/specify user preferences. An aggregation operator, when interpreted as a truth function, may be an arithmetic mean, a weighted sum or in general any monotone application whose arguments are values of a complete bounded lattice L. For example, if an aggregator @ is interpreted as $\dot{@}(x, y, z) = (3x + 2y + z)/6$, 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 $\dot{@} : L^n \to L$ is required to be monotone and fulfill $\dot{@}(\top, \ldots, \top) = \top$, $\dot{@}(\bot, \ldots, \bot) = \bot$.

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 of the form $\langle \leftarrow_i, \&_i \rangle$ where each $\&_i$ is a conjunctor [7] intended to produce the evaluation of *modus ponens*. In general, the set of truth values L may be the carrier of any complete bounded lattice, as occurs for instance with the simpler case of the set of real numbers in the interval $[0, 1]$ (which is a totally ordered lattice or chain).

A *rule* is a formula $H \leftarrow_i \mathcal{B}$, where H is an atomic formula (usually called the *head*) and $\mathcal{B}$ (which is called the *body*) is a formula built from atomic formulas $B_1, \ldots, B_n$, $n \geq 0$, truth values of L and aggregation operators. Rules whose body is $\top$ are called *facts* (usually, we will represent a fact as a rule with an empty body). A *goal* is a body submitted as a query to the system. Variables in a rule are assumed to be universally quantified. 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 that the user of the system has in the truth of the rule $\mathcal{R}$. Observe that, truth degrees are axiomatically assigned (for instance) by an expert. By abuse of language, we sometimes refer a tuple $\langle \mathcal{R}; \alpha \rangle$ as a "rule".

Formulas are interpreted on a multi–adjoint lattice. In this framework, it is sufficient to consider Herbrand interpretations, in order to define a declarative semantics [17]. Therefore, a *fuzzy interpretation*, $\mathcal{I}$, is a mapping from the Herbrand base, $B_\mathcal{L}$, into the multi–adjoint lattice of truth values L. The truth value of a ground atom $A \in B_\mathcal{L}$ is $\mathcal{I}(A)$. Given an assignment ϑ from terms into elements of the Herbrand universe $U_\mathcal{L}$, the valuation of a formula in an interpretation is obtained by structural induction:

$$\begin{aligned}
\mathcal{I}(p(t_1, \ldots, t_n))[\vartheta] &= \mathcal{I}(p(t_1\vartheta, \ldots, t_n\vartheta)), \\
\mathcal{I}(@(A_1, \ldots, A_n))[\vartheta] &= \dot{@}(\mathcal{I}(A_1)[\vartheta], \ldots, \mathcal{I}(A_n)[\vartheta]), \\
\mathcal{I}(A \leftarrow \mathcal{B})[\vartheta] &= \mathcal{I}(A)[\vartheta] \dot{\leftarrow} \mathcal{I}(\mathcal{B})[\vartheta], \\
\mathcal{I}((\forall x)\mathcal{A})[\vartheta] &= inf\{\mathcal{I}(\mathcal{A})[\vartheta'] \mid \vartheta' \ \ x\text{–equivalent to } \ \vartheta\},
\end{aligned}$$

[7] For a formal definition of a multi-adjoint lattice and the semantic properties of the connectives in $\mathcal{L}$, see [17]. It is noteworthy that a symbol $\&_j$ of $\mathcal{L}$ does not always need to be part of an adjoint pair.

where p is a predicate symbol, @ an arbitrary aggregator, A and A_i atomic formulas, $\mathcal{B}$ any body, $\mathcal{A}$ any formula and we denote the truth value function of a connective @ by $\dot{@}$. An assignment ϑ' is x–equivalent to ϑ when $z[\vartheta'] = z[\vartheta]$ for all variable $z \neq x$ of $\mathcal{L}$. When the assignment would not be relevant, we shall omit it during the valuation of a formula. Moreover, an interpretation $\mathcal{I}$ satisfies a rule $\langle A \leftarrow_i \mathcal{B}; v\rangle$ if, and only if, $v \leq \mathcal{I}(A \leftarrow_i \mathcal{B})$, and an interpretation $\mathcal{I}$ is a *model* of $\mathcal{P}$ if, and only if, all rules in $\mathcal{P}$ are satisfied by $\mathcal{I}$.

2.2 Procedural Semantics

The procedural semantics of the multi-adjoint logic language $\mathcal{L}$ can be thought of as an operational phase followed by an interpretive one. Similarly to [18], in this section we establish a clear separation between both phases. The operational mechanism uses a generalization of *modus ponens* that, given an atomic goal A and a program rule $\langle H \leftarrow_i \mathcal{B}; v\rangle$, if there is a substitution $\theta = mgu(\{A = H\})$ [1], we substitute the atom A by the expression $(v \&_i \mathcal{B})\theta$. In the following, we write $\mathcal{C}[A]$ to denote a formula where A is a sub-expression (usually an atom) which arbitrarily occurs in the —possibly empty— context $\mathcal{C}[]$. Moreover, an expression $\mathcal{C}[A/H]$ means the replacement of A by H in context $\mathcal{C}[]$. Also we use $\mathcal{V}ar(s)$ for referring to the set of 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 a* state *and we denote by $\mathcal{E}$ the set of states. Given a program $\mathcal{P}$, an* admissible computation *is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathcal{E} \times \mathcal{E})$ is the smallest relation satisfying the following* admissible rules [2] *(where we always consider that A is the selected atom in $\mathcal{Q}$):*

1) $\langle \mathcal{Q}[A]; \sigma\rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v \&_i \mathcal{B}])\theta; \sigma\theta\rangle$ *if* $\theta = mgu(\{H = A\})$, $\langle H \leftarrow_i \mathcal{B}; v\rangle$ *in* $\mathcal{P}$.
2) $\langle \mathcal{Q}[A]; \sigma\rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/\bot]); \sigma\rangle$ *if there is no rule in $\mathcal{P}$ whose head unifies A.*

Formulas involved in admissible computation steps are renamed apart before being used. Note also that the second rule is introduced to cope with (possible) unsuccessful admissible derivations. When needed, we shall use the sym-

[1] Let $mgu(E)$ denote the *most general unifier* of an equation set E (see [22] for a formal definition of this concept).

[2] Note that the first case subsumes the second case in the original definition presented in [17], since a fact $H \leftarrow$ is really the rule $H \leftarrow \top$. However, from a practical point of view, when an admissible step is performed with a fact, we abbreviate the step "$\langle \mathcal{Q}[A]; \sigma\rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v \&_i \top])\theta; \sigma\theta\rangle$" by "$\langle \mathcal{Q}[A]; \sigma\rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v])\theta; \sigma\theta\rangle$", since $\dot{\&}_i(v, \top) = v$.

bols $\rightarrow_{AS1}$ and $\rightarrow_{AS2}$ to distinguish between specific admissible steps. Also, when required, the exact program rule used in the corresponding step will be annotated as a super-index of $\rightarrow_{AS}$. Also the symbols $\rightarrow^{+}_{AS}$ and $\rightarrow^{*}_{AS}$ denote, respectively, the transitive closure and the reflexive, transitive closure of $\rightarrow_{AS}$.

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.*

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.

Definition 3 (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 the smallest one satisfying: $\langle Q[@(r_1, \dots, r_n)]; \sigma\rangle \rightarrow_{IS} \langle Q[@(r_1, \dots, r_n)/\dot{@}(r_1, \dots, r_n)]; \sigma\rangle$, where $\dot{@}$ is the truth function of connective $@$ in the lattice $\langle L, \preceq\rangle$ associated to $\mathcal{P}$.*

Definition 4 *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 $Q' = r \in L$, $\langle L, \preceq\rangle$ being the lattice associated to $\mathcal{P}$, the state $\langle r; \sigma\rangle$ is called a* fuzzy computed answer *(f.c.a.) for that derivation.*

We denote by $\rightarrow^{+}_{IS}$ and $\rightarrow^{*}_{IS}$ the transitive closure and the reflexive, transitive closure of $\rightarrow_{IS}$, respectively. Usually, we refer to a *complete derivation* as the sequence of admissible/interpretive steps of the form $\langle Q; id\rangle \rightarrow^{*}_{AS} \langle \mathcal{Q}'; \sigma\rangle \rightarrow^{*}_{IS} \langle r; \sigma\rangle$ (sometimes we denote it by $\langle Q; id\rangle \rightarrow^{*}_{AS/IS} \langle r; \sigma\rangle$) where $\langle \mathcal{Q}'; \sigma[\mathcal{V}ar(\mathcal{Q})]\rangle$ and $\langle r; \sigma[\mathcal{V}ar(\mathcal{Q})]\rangle$ are, respectively, the a.c.a. and the f.c.a. for the derivation. Also note that, sometimes, when it is not important to pay attention on the substitution component of a f.c.a. $\langle r; \theta\rangle$ (maybe, because $\theta = id$) we shall refer to the value component r as the "f.c.a.".

In the following, we denote by $\mathcal{FCA}(E)$ the set of f.c.a.'s of a given expression (goal) E. Formally, $\mathcal{FCA}(E) = \{\langle r; \theta\rangle \mid \langle E; id\rangle \rightarrow^{*}_{AS/IS} \langle r; \sigma\rangle, r \in L, \theta = \sigma[\mathcal{V}ar(E)]\}$ which can be generalized to a set of expressions $E_1, \dots, E_n$ as $\mathcal{FCA}(E_1, \dots, E_n) = \mathcal{FCA}(E_1) \cup \dots \cup \mathcal{FCA}(E_n)$. Moreover, in order to give a measure of the computational effort needed to compute such set of f.c.a.'s for a given expression E, we denote by $[\![\mathcal{FCA}]\!](E)$ the total sum of admissible/interpretive steps needed to generate the whole set $\mathcal{FCA}(E)$. An evaluation step contributing to reach one or more solutions for a given expression E, is only counted once when computing $\mathcal{FCA}(E)$.

3 Partial Evaluation of Multi-Adjoint Logic Programs

This section formalizes the basic notions involved in the partial evaluation of multi-adjoint logic programs. Some of these concepts were introduced in a preliminary version appeared in [23]. We start with the concept of resultant.

Definition 5 (Resultant) *Let $\mathcal{P}$ be a program and $\mathcal{Q}$ a goal. Given the sequence of admissible and interpretive steps $\langle \mathcal{Q}; id\rangle \rightarrow^{+} \langle \mathcal{Q}'; \sigma\rangle$, whose length is strictly greater than zero, we define the* resultant *of this derivation*[2] *as: $\langle Q\sigma \leftarrow \mathcal{Q}' \ ; \top\rangle$, where "$\leftarrow$" is any implication with an adjoint conjunctor.*

Observe that, in contrast with the operational semantics defined in Section 2.2, admissible and interpretive steps can be interleaved in any order. In practice we will give preference to the interpretive steps over the admissible steps during the PE process. This method resembles the *normalization* technique[3] introduced in the context of functional logic programming to reduce the non-determinism of a computation [24]. In the sequel we call *normalization* the sequence of interpretive steps performed before an operational unfolding step.

Also, note that the "rule" component of a resultant is not in general a rule, since goal $\mathcal{Q}$ stands for an aggregation of atomic formulas. A resultant is particularly significant when the original goal $\mathcal{Q}$ is an atomic formula A, since then, the resultant $\langle A\sigma \leftarrow \mathcal{Q}' \ ; \top\rangle$ is a pair $\langle rule; truth\ degree\rangle$ which is a constituent of the transformed program. The following example illustrates the intuition behind the notion of resultant.

Example 6 *Given the lattice $([0,1], \leq)$, where "$\leq$" is the usual order on real numbers, let $\mathcal{P}$ be the following multi-adjoint logic program, where labels* G *and* L *on program rules refer to* Gödel*'s intuitionistic logic and* Łukasiewicz *logic, respectively:*

$$
\begin{array}{llll}
\mathcal{R}_1 : \langle p(a) \leftarrow_{\mathtt{L}} q(X,a); & 0.7\rangle & \mathcal{R}_5 : \langle s(a) \leftarrow_{\mathtt{G}} t(a); & 0.5\rangle \\
\mathcal{R}_2 : \langle p(a) \leftarrow_{\mathtt{G}} s(Y); & 0.5\rangle & \mathcal{R}_6 : \langle s(b) \leftarrow_{\mathtt{L}} t(b); & 0.8\rangle \\
\mathcal{R}_3 : \langle p(Y) \leftarrow_{\mathtt{G}} q(b,Y) \&_{\mathtt{L}} t(Y); & 0.8\rangle & \mathcal{R}_7 : \langle t(a) \leftarrow_{\mathtt{L}} p(X); & 0.9\rangle \\
\mathcal{R}_4 : \langle q(b,a) \leftarrow \ ; & 0.9\rangle & \mathcal{R}_8 : \langle t(b) \leftarrow_{\mathtt{G}} q(X,a); & 0.9\rangle
\end{array}
$$

Now, we can build the following derivation in $\mathcal{P}$, starting from goal $p(X)$:

[2] Note that, in general, the derivation may possibly be incomplete.

[3] In a *normalizing narrowing* strategy, a term is rewritten to its normal form before a narrowing step is applied. This procedure is also applied in narrowing-based PE and narrowing-based fold/unfold transformation techniques.

$\langle p(X); id\rangle \rightarrow_{AS1}{}^{\mathcal{R}_1} \langle 0.7 \&_{\text{L}} q(X_1, a); \{X/a\}\rangle \rightarrow_{AS1}{}^{\mathcal{R}_4} \langle 0.7 \&_{\text{L}} 0.9; \{X/a, X_1/b\}\rangle$

whose resultant is $\langle p(a) \leftarrow 0.7 \&_{\text{L}} 0.9; 1\rangle$. *Note that, an admissible step performed with this resultant on goal* $p(X)$ *mimics the effects of the two admissible steps of the former derivation.*

Therefore, the resultant encapsulates the whole information of the original derivation for goal A in a single step (hence its name). It also compiles the information of the truth degree inside the body of the resultant. So, in order to reproduce the effect of the original derivation, it suffices that the associated truth degree component of the resultant be the top value of lattice L.

The partial evaluation of an atomic goal is defined by firstly constructing an incomplete search tree for that goal and then, extracting the specialized definition —the *resultants*— from the root-to-leaf branches. Hence, before defining this concept, we precise the introduction of the following notion of unfolding tree.

Definition 7 (Unfolding tree) *Let* $\mathcal{P}$ *be a program and let* $\mathcal{Q}$ *be a goal. An* unfolding tree τ_φ *for* $\mathcal{P}$ *and* $\mathcal{Q}$ *(using the computation rule*[4] φ*) is a set of* $\langle goal; substitution\rangle$ *pair nodes satisfying the following conditions:*

(1) The root node of τ_φ *is* $\langle \mathcal{Q}\ ; id\rangle$, *where id is the identity substitution.*
(2) If $\mathcal{N}_i \equiv \langle \mathcal{Q}[A]; \sigma\rangle$ *is a node of* τ_φ *and assuming that* $\varphi(\mathcal{Q}) = A$ *is the selected atom, then for each rule* $\mathcal{R}_j \equiv \langle H \leftarrow \mathcal{B}; v\rangle$ *in* $\mathcal{P}$, *with* $\theta = mgu(\{H = A\})$, $\mathcal{N}_{ij} \equiv \langle (\mathcal{Q}[A/v\&\mathcal{B}])\theta; \sigma\theta\rangle$ *is a node of* τ_φ.
(3) If $\mathcal{N}_i \equiv \langle \mathcal{Q}[@(r_1, \ldots, r_n)]; \sigma\rangle$ *is a node of the unfolding tree* τ_φ *then,* $\mathcal{N}_{ij} \equiv \langle \mathcal{Q}[@(r_1, \ldots, r_n)/\dot{@}(r_1, \ldots, r_n)]); \sigma\rangle$ *is a node of* τ_φ.

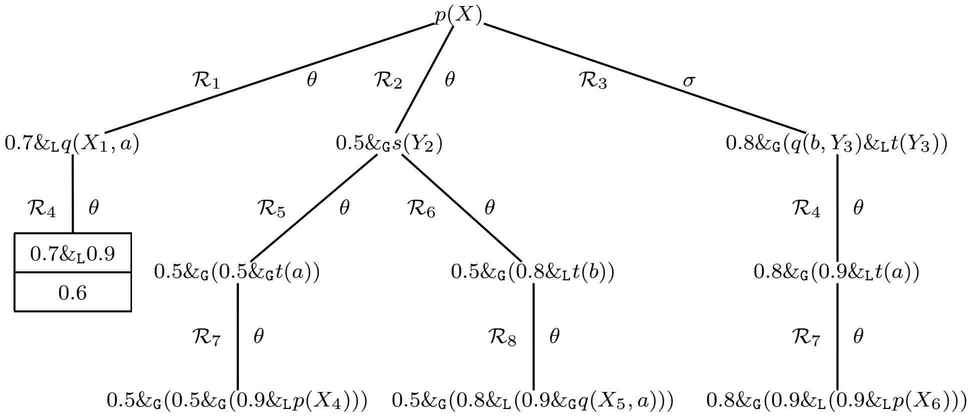
As defined in [20,18], the second and third cases relate to the application of an operational unfolding step and an interpretive unfolding step, respectively. An *incomplete* unfolding tree is an unfolding tree which, in addition to completely evaluated leaves, may also contain leaves where no atom (or interpretable expression) has been selected for a further unfolding step, which allows us to finish derivations at any adequate point.

Definition 8 (Partial evaluation of an atom) *Let* $\mathcal{P}$ *be a program,* A *be an (atomic) goal, and* τ *a (possibly incomplete) finite unfolding tree for* $\mathcal{P}$ *and* A, *containing at least one non-root node. Let* $\{\mathcal{Q}_i \mid i = 1, \ldots, k\}$ *be the leaves of the branches of* τ, *and let* $\mathcal{P}' = \{\langle A\sigma_i \leftarrow \mathcal{Q}_i\ ; \top\rangle \mid i = 1, \ldots, k\}$ *be the set of resultants associated to derivations* $\{\langle A\ ; id\rangle \rightarrow^+ \langle \mathcal{Q}_i\ ; \sigma_i\rangle \mid i = 1, \ldots, k\}$.

[4] A *computation rule* is a function that, when applied to a query $\mathcal{Q}$, outputs one of the atoms of $\mathcal{Q}$ to be evaluated by an admissible step. Note that a computation rule may select an atom for which no admissible step is possible.

Then, the set $\mathcal{P}'$ is called a partial evaluation *of A in $\mathcal{P}$ (using τ).*

Example 9 *Given the program $\mathcal{P}$ of Example 6, we can construct the following unfolding tree with depth 3 (that is, all its branches have been unfolded no more than 3 steps) for program $\mathcal{P}$ and atom $p(X)$:*



Depicting the figure, we have followed these conventions: the substitution component of each node is annotated as a label of the arc connecting that node and its parent node (the only exception is the root node, whose associate substitution is by default the identity substitution, id); by the sake of simplicity, substitutions are restricted to variables of the initial goal, hence: $\theta = \{X/a\}$ and $\sigma = \{X/Y_3\}$. Note also that, as usual in classical resolution procedures, rules in admissible steps are taken standardized apart, whereas those nodes where a normalization sequence (of interpretive steps) has been applied, generating a new additional node, are surrounded by boxes. From the above incomplete unfolding tree we obtain the following set of resultants:

$$\mathcal{R}'_1 : \langle p(a) \leftarrow 0.6; 1\rangle$$
$$\mathcal{R}'_2 : \langle p(a) \leftarrow 0.5\&_{\mathtt{G}}(0.5\&_{\mathtt{G}}(0.9\&_{\mathtt{L}}\ p(X_4))); 1\rangle$$
$$\mathcal{R}'_3 : \langle p(a) \leftarrow 0.5\&_{\mathtt{G}}(0.8\&_{\mathtt{L}}(0.9\&_{\mathtt{G}}q(X_5, a))); 1\rangle$$
$$\mathcal{R}'_4 : \langle p(a) \leftarrow 0.8\&_{\mathtt{G}}(0.9\&_{\mathtt{L}}(0.9\&_{\mathtt{L}}p(X_6))); 1\rangle$$

It is easy to extend Definition 8 to sets of atomic formulas. If S is a finite set of atoms, then a partial evaluation of S in $\mathcal{P}$ (also called a *partial evaluation of $\mathcal{P}$ with respect to S*) is the union of the partial evaluations of the elements of S in $\mathcal{P}$. Moreover, the restriction to specialize atom goals is not a severe limitation in order to prevent the specialization of more complex goals, if we convey to specialize their components by separate [5]. Given a program $\mathcal{P}$ and

[5] However, it is necessary to admit that some opportunities to achieve a good specialization are lost. This can be avoided by introducing some particular techniques

a complex goal $\mathcal{Q}$, assuming that $S = \{B_1, \ldots, B_n\}$ is the set of the atomic constituents of $\mathcal{Q}$, the partial evaluation of $\mathcal{Q}$ in $\mathcal{P}$ is the partial evaluation of $\mathcal{P}$ with regard to S.

4 Reductants versus PE-reductants

In this section we recall from [17] the notion of reductant and, after establishing the relationship between the construction of reductants and techniques coming from the field of partial evaluation, we override the original definition by proposing our improved notion of PE-reductant. But let us firstly state the relevance of reductants and discuss more deeply the problems they introduce in the framework of multi-adjoint logic programming.

Reductancts were introduced in the context of multi-adjoint logic programming to cope with a problem of incompleteness that arises when dealing with some (non totally-ordered) lattices. In general, it might be not possible to compute the greatest correct answer (for a given goal and program) when considering a partially ordered lattice $(L, \preceq)$ [17]. For instance, let a, b be two non comparable elements in L; assume that for a (ground) goal A there are only two (fact) rules ($\langle A \leftarrow; a\rangle$ and $\langle A \leftarrow; b\rangle$) whose heads directly match with it; the first rule contributes with truth degree a, and derives the fuzzy computed answer a (with empty substitution); similarly, the second one contributes with b, and derives the fuzzy computed answer b; therefore, by the soundness theorem of multi-adjoint logic programming [17], both a and b are correct answers and hence, by definition of correct answer [17], the supremum (or *lub*, least upper bound) $sup\{a, b\}$, is also a correct answer; however, since there exists $c = sup\{a, b\}$ in L, our computational principle, as described in Section 2.2, will never return c as a computed answer, thus implying that completeness would be lost. The above problem can be solved by extending the original program with an special rule $\langle A \leftarrow sup\{a, b\}; \top\rangle$, the so called *reductant*, which allows us to obtain the supremum of all the computational contributions to A.

The above discussion shows that a multi-adjoint logic program, interpreted inside a partially ordered lattice, needs to contain all its reductants in order to guarantee the completeness property. This obviously increases both the size and execution time of the final *"completed"* program. However, this negative effects can be highly diminished if the proposed reductants have been partially evaluated before being introduced in the target program: the compu-

which are able to adequately reorder and split up complex goals into smaller pieces before considering them for specialization, as it has been proposed in the field of conjunctive partial deduction [25,26].

tational effort done (once) at generation time would be avoided (many times) at execution time. In what follows, we are interested in showing how a refined notion of a reductant can be constructed using PE techniques in order to solve the problems detailed before. The starting point is the original definition presented in [17], where the classical notion of reductant was initially adapted to the multi-adjoint logic programming framework in the following terms:

Definition 10 (Reductant [17]) *Let $\mathcal{P}$ be a program, A a ground atom, and $\langle C_i \leftarrow_i \mathcal{B}_i; v_i\rangle$ the (non empty) set of rules in $\mathcal{P}$ whose head matches with A (i.e., there are θ_i such that $A = C_i\theta_i$). A reductant for A in $\mathcal{P}$ is a rule $\langle A \leftarrow\ @(\mathcal{B}_1, \ldots, \mathcal{B}_n)\theta; \top\rangle$ where $\theta = \theta_1 \cdots \theta_n$, $\leftarrow$ is any implication with an adjoint conjunctor, and the truth function for the intended aggregator $@$ is defined as $\dot{@}(b_1, \ldots, b_n) = sup\{v_1 \dot{\&}_1 b_1, \ldots, v_n \dot{\&}_n b_n\}$.*

Now we are going to show how Definition 10 can be improved, leading to a more flexible approximation to this concept, by using proper notions of partial evaluation. So, using an arbitrary unfolding tree, τ, for a program $\mathcal{P}$ and a ground atom A, it is possible to construct a more refined version of the notion of a reductanct which we call *PE-reductant* for A in $\mathcal{P}$. The main novelty of the following definition (which generalizes a very close, precedent notion of PE-reductant, that we firstly introduced in [23]), is the fact that it is directly based on the set of leaves of a given unfolding tree. Similarly to the previous definition, in the sequel we assume that $\leftarrow$ is the implication of any adjoint pair $\langle \leftarrow, \& \rangle$.

Definition 11 (PE-reductant) *Let $\mathcal{P}$ be a program, A a ground atom, and τ an unfolding tree for A in P. A PE-reductant for A in $\mathcal{P}$ with respect to τ, is a rule $\langle A \leftarrow\ @_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n); \top\rangle$, where the truth function for the intended aggregator $@_{sup}$ is defined as $\dot{@}_{sup}(d_1, \ldots, d_n) = sup\{d_1, \ldots, d_n\}$, and $\mathcal{D}_1, \ldots, \mathcal{D}_n$ are, respectively, the leaves of τ.*

Observe that our definition of PE-reductant respects the syntax of our extended language (Section 2.1), where truth degrees and adjoint conjunctions are really allowed in the body of program rules. As in the case of resultants, PE-reductants incorporate information about all the relevant aspects of the rules $\langle C_i \leftarrow_i \mathcal{B}_i; v_i\rangle$ used for the evaluation of the atom A: the truth degree v_i, the adjoint implication and conjunction operators, the computed substitutions and the instances of the bodies $\mathcal{B}_i$. On the other hand, in the particular case in which the tree used in Definition 11 is unfolded only one step (assuming that $\{\langle C_i \leftarrow_i \mathcal{B}_i; v_i\rangle \in \mathcal{P} \mid \text{there is a } \theta_i, A = C_i\theta_i\}$ is the -nonempty- set of rules in $\mathcal{P}$ whose heads match with A) then, the resulting PE-reductant is the rule $\langle A \leftarrow @_{sup}((v_1 \&_1 \mathcal{B}_1)\theta_1, \ldots, (v_n \&_n \mathcal{B}_n)\theta_n); \top\rangle$, whose shape greatly mirrors Definition 10.

Example 12 *Given the lattice $([0, 1], \preceq)$, where "$\preceq$" is the usual order on* *real numbers, consider the following multi-adjoint logic program $\mathcal{P}$:*

$$\begin{array}{llll}
\mathcal{R}_1 : \langle p(a) \leftarrow_{\mathtt{L}} q(X, a); & 0.7\rangle & \quad \mathcal{R}_5 : \langle q(b, a) \leftarrow\ ; & 0.9\rangle \\
\mathcal{R}_2 : \langle p(a) \leftarrow_{\mathtt{G}} s(Y); & 0.5\rangle & \quad \mathcal{R}_6 : \langle s(a) \leftarrow_{\mathtt{G}} t(a); & 0.5\rangle \\
\mathcal{R}_3 : \langle p(Y) \leftarrow\ ; & 0.6\rangle & \quad \mathcal{R}_7 : \langle s(b) \leftarrow\ ; & 0.8\rangle \\
\mathcal{R}_4 : \langle p(Y) \leftarrow_{\mathtt{G}} q(b, Y) \&_{\mathtt{L}}\ t(Y); & 0.8\rangle & \quad \mathcal{R}_8 : \langle t(a) \leftarrow_{\mathtt{L}} p(X); & 0.9\rangle
\end{array}$$

The one-step unfolding tree for program $\mathcal{P}$ and atom $p(a)$ is:

$$\begin{array}{cccc}
 & \langle p(a); id\rangle & & \\
\mathcal{R}_1 & \mathcal{R}_2 & \mathcal{R}_3 & \mathcal{R}_4 \\
\langle 0.7\&_{\mathtt{L}} q(X_1, a); id\rangle & \langle 0.5\&_{\mathtt{G}} s(Y_2); id\rangle & \langle 0.6; \{Y_3/a\}\rangle & \langle 0.8\&_{\mathtt{G}}(q(b, a)\&_{\mathtt{L}} t(a)); \{Y_4/a\}\rangle
\end{array}$$

from which we obtain the PE-reductant:

$$\langle p(a) \leftarrow @_{sup}(0.7\&_{\mathtt{L}} q(X_1, a), 0.5\&_{\mathtt{G}} s(Y_2), 0.6, 0.8\&_{\mathtt{G}}(q(b, a)\&_{\mathtt{L}} t(a)));\ 1\rangle.$$

On the other hand, it is important to contrast the similarities/ differences between this rule and the reductant that the application of Definition 10 would have produced: $\langle p(a) \leftarrow \dot{@}(q(X_1, a), s(Y_2), 1, q(b, a)\&_{\mathtt{L}} t(a));\ 1\rangle$, where $\dot{@}$ is defined by $\dot{@}(b_1, b_2, b_3, b_4) = sup\{0.7\dot{\&}_{\mathtt{L}} b_1, 0.5\dot{\&}_{\mathtt{G}} b_2, 0.6, 0.8\dot{\&}_{\mathtt{G}} b_4\}$.

This particular case of PE-reductant which uses a one-step unfolding tree, conforms with the original definition of reductant appeared in [17]. In particular, the following result shows that both kinds of reductants are semantically equivalent [6], since they have the same value when they are interpreted.

Theorem 13 *Let $\mathcal{P}$ be a program, A a ground atom and $\mathcal{R} \equiv \langle A \leftarrow @(\mathcal{B}_1, \ldots, \mathcal{B}_n)\theta; \top\rangle$ the reductant for A in $\mathcal{P}$, where $\theta = \theta_1 \cdots \theta_n$ and each substitution θ_i is a matcher of A and the head of a rule $\langle C_i \leftarrow_i \mathcal{B}_i; v_i\rangle$. The PE-reductant $\mathcal{R}' \equiv \langle A \leftarrow @_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n); \top\rangle$ (where $\mathcal{D}_i \equiv v_i \&_i \mathcal{B}_i\theta$, $1 \leq i \leq n$) obtained from an unfolding tree of depth one for $\mathcal{P}$ and A, is semantically equivalent to the reductant $\mathcal{R}$.*

PROOF. In order to state the semantic equivalence between both notions of reductants it suffices to prove that $\mathcal{I}(@(\mathcal{B}_1, \ldots, \mathcal{B}_n)\theta) = \mathcal{I}(@_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n))$. First note that, the rules $C_i \leftarrow_i \mathcal{B}_i$ whose head matches with A, are taken standardized apart. Moreover, the atom A is ground. Therefore, the substitutions θ_i, such that $A = C_i\theta_i$, do not share variables in common either in

[6] The procedural counterpart of this result is proved in Section 5 (see Theorem 18).

their domains or in their ranges. Hence, $\theta = \theta_1\theta_2 \cdots \theta_n = \theta_1 \cup \theta_2 \cup \cdots \cup \theta_n$. Then:

$$
\begin{aligned}
\mathcal{I}(@(\mathcal{B}_1, \ldots, \mathcal{B}_n)\theta) &= \mathcal{I}(@(\mathcal{B}_1\theta, \ldots, \mathcal{B}_n\theta)) \\
&= \mathcal{I}(@(\mathcal{B}_1\theta_1, \ldots, \mathcal{B}_n\theta_n)) \\
&= \dot{@}(\mathcal{I}(\mathcal{B}_1\theta_1), \ldots, \mathcal{I}(\mathcal{B}_n\theta_n)) \\
&= sup\{v_1 \dot{\&}_1 \mathcal{I}(\mathcal{B}_1\theta_1), \ldots, v_n \dot{\&}_n \mathcal{I}(\mathcal{B}_n\theta_n)\} \\
&= sup\{\mathcal{I}(v_1 \&_1 \mathcal{B}_1\theta_1), \ldots, \mathcal{I}(v_n \&_n \mathcal{B}_n\theta_n)\} \\
&= sup\{\mathcal{I}(\mathcal{D}_1), \ldots, \mathcal{I}(\mathcal{D}_n)\} \\
&= \dot{@}_{sup}(\mathcal{I}(\mathcal{D}_1), \ldots, \mathcal{I}(\mathcal{D}_n)) \\
&= \mathcal{I}(@_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n)),
\end{aligned}
$$

which concludes the proof.

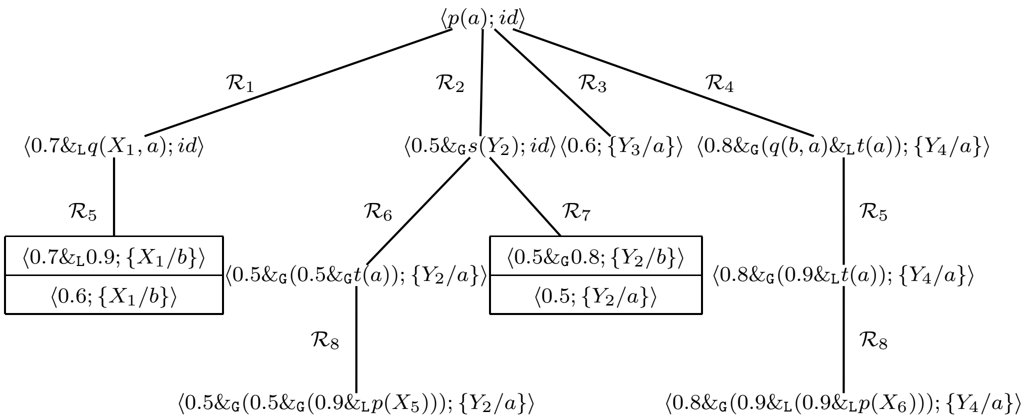
It is noteworthy that a PE-reductant can be constructed by using the notion of unfolding tree in the following way.

Definition 14 (Construction of PE-reductants) *Let $\mathcal{P}$ be a program and let A be a ground atomic goal. We can enumerate the following steps in the construction of a PE-reductant of A in $\mathcal{P}$:*

(1) Construct an unfolding tree, τ, for $\mathcal{P}$ and A, that is, the tree obtained by unfolding the atom A in the program.
(2) Collect the set of leaves $S = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ in τ.
(3) Construct the rule $\langle A \leftarrow @_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n); \top\rangle$, which is the PE-reductant of A in $\mathcal{P}$ with regard to τ.

The following example presents a PE-reductant obtained from an unfolding tree of depth 3 (all its branches have been unfolded no more than 3 steps).

Example 15 *Let $\mathcal{P}$ be the program of Example 12 and consider atom $p(a)$. In the next figure, nodes where normalization steps have been applied, producing additional nodes, are remarked by boxes.*



After collecting the leaves of this unfolding tree, we obtain the following PE-reductant: $\langle p(a) \leftarrow @_{sup}(0.6, 0.5\&_{\texttt{G}}(0.5\&_{\texttt{G}}(0.9\&_{\texttt{L}}p(X_5))), 0.5, 0.6, 0.8\&_{\texttt{G}}(0.9\&_{\texttt{L}}(0.9\&_{\texttt{L}}p(X_6)))); 1\rangle$.

Since this formulation is based on partial evaluation techniques, it can be seen as a method that produces a specialization of a program with respect to an atomic goal which, in particular, means that it is able to compute the greatest correct answer for that goal too. Moreover, although for the same program $\mathcal{P}$ and a ground atom A, it is possible to derive distinct reductants, depending on the precision of the underlying unfolding tree, we claim the remarkable fact that all of them are able to compute the same greatest correct answer for the ground goal A.

To finish this section, we wish to introduce a brief comment about our running examples. We have seen that PE techniques are in general useful for different software engineering purposes, independently of the partial/total ordering among the elements of the underlying lattices associated to fuzzy programs. For the sake of simplicity, all the previous examples, specially those illustrating the generation of unfolding trees in Sections 3 and 4, only consider the simple case corresponding to lattice $([0,1], \leq)$. However, as we also advanced in the introduction section, (PE-)reductants are really interesting only when we consider partially ordered lattices. For this reason, all the examples we provide in the sequel use this kind of more complex lattices.

5 Formal Properties of PE-reductants

In this section we establish the formal correctness properties of PE-reductants, starting with the procedural properties and following with their semantic counterparts.

As we have seen in the previous section, it is important to note that the original notion of reductant as well as our improved definition of PE-reductant, are both referred to ground atoms. On the other hand, the substitution component of any fuzzy computed answer, for a given ground atom A, is irrelevant. So, in this section, we will take advantage of these facts when defining and formally proving our main results. In particular, in what follows, we will use with no risk a simpler version (where the second component —substitution— of a f.c.a. is dropped out) of the notions of $\mathcal{FCA}$ and $[\![\mathcal{FCA}]\!]$ introduced in Section 2. That is, from now, $\mathcal{FCA}(E) = \{r \in L \mid \langle E; id\rangle \rightarrow^*_{AS/IS} \langle r; \sigma\rangle\}$, whereas $[\![\mathcal{FCA}]\!](E)$ is the total sum of admissible/interpretive steps needed to generate the whole set $\mathcal{FCA}(E)$.

Moreover, for readability reasons too, in the sequel we will use the words PE^1-reductant and PE^k-reductant for referring to those PE-reductants which are obtained from unfolding trees of depth 1 and depth k, respectively.

5.1 Procedural Correctness and Efficiency

This section is devoted to establish that the classical notion of reductant, according Definition 10, the PE^1-reductant and the PE^k-reductant contribute with the same fuzzy computed answer for a ground goal A in a program $\mathcal{P}$ and, therefore, they can be considered equivalent under a procedural point of view. In addition, we will also prove that the notion of PE^k-reductant is more efficient than the one defined in [17], in the sense that, by using a PE^k-reductant we can obtain fuzzy computed answers for a given goal with a lesser computational effort than by using the ordinary reductans of [17]. In order to achieve this goal, we firstly formalize the following preparatory result.

Lemma 16 *Given an unfolding tree τ for a program $\mathcal{P}$ and a ground atom A, let E_1 and E_2 be expressions (nodes) of τ, such that there exists a one-step derivation of the form $E_1 \rightarrow_{AS/IS} E_2$. Then,*

(1) $\mathcal{FCA}(E_2) \subseteq \mathcal{FCA}(E_1)$, *and*
(2) $[\![\mathcal{FCA}]\!](E_2) < [\![\mathcal{FCA}]\!](E_1)$.

PROOF. We prove each claim of the Lemma separately:

(1) It suffices to show that, for each value $r \in \mathcal{FCA}(E_2)$, then $r \in \mathcal{FCA}(E_1)$. By hypothesis, there exists a one-step derivation $D : \langle E_1; id\rangle \rightarrow_{AS/IS} \langle E_2; \sigma\rangle$ and moreover, since $r \in \mathcal{FCA}(E_2)$, there also exists a derivation $D' : \langle E_2; id\rangle \rightarrow^*_{AS/IS} \langle r; \sigma'\rangle$. Now, by composing both derivations D and D', we have the new derivation $D'' : \langle E_1; id\rangle \rightarrow_{AS/IS} \langle E_2; \sigma\rangle \rightarrow^*_{AS/IS} \langle r; \sigma\sigma'\rangle$, which justifies that $r \in \mathcal{FCA}(E_1)$, as we wanted to prove.

(2) This claim trivially follows from the fact that the previous derivation D'' has just one step more (the first one, associated to D) than D', that is $length(D') < length(D) + length(D') = 1 + length(D') = length(D'')$, which directly implies that $[\![\mathcal{FCA}]\!](E_2) < [\![\mathcal{FCA}]\!](E_1)$.

The following proposition proves that, given an unfolding tree of a ground atom A and a program $\mathcal{P}$, the set of fuzzy computed answers of a node (state) E is the union of the sets of fuzzy computed answers of its successor states (that is, those nodes obtained from E after executing an admissible or interpretive step).

Proposition 17 *Given an unfolding tree τ for a program $\mathcal{P}$ and a ground atom A, containing at least one non-root node, let E be an expression (node) of τ and let $\mathcal{U}(E) = \{E' \mid E \rightarrow_{AS/IS} E'\}$ be the set of successors of E. Then, $\mathcal{FCA}(E) = \bigcup_{E_i \in \mathcal{U}(E)} \mathcal{FCA}(E_i)$.*

PROOF. If E is not a value of L and for a certain index i there is a step $E \rightarrow_{AS/IS} E_i$, by the first claim of Lemma 16, we have that $\mathcal{FCA}(E_i) \subseteq \mathcal{FCA}(E)$ and, by definition of union, we can conclude that

$$\bigcup_{E_i \in \mathcal{U}(E)} \mathcal{FCA}(E_i) \subseteq \mathcal{FCA}(E)$$

On the contrary, if $r \in \mathcal{FCA}(E)$, we shall prove that r is a f.c.a. of the set $\bigcup_{E_i \in \mathcal{U}(E)} \mathcal{FCA}(E_i)$. Since $r \in \mathcal{FCA}(E)$, there exists a derivation $E \rightarrow^n_{AS/IS} r$, for which we consider two cases:

- if $n = 0$, then E is a value of L and the result vacuously holds.
- if $n > 0$, let E_j be the expression verifying $\langle E; id\rangle \rightarrow_{AS/IS} \langle E_j; \sigma\rangle \rightarrow^{n-1}_{AS/IS} \langle r; \theta\rangle$. Then, $r \in \mathcal{FCA}(E_j) \subseteq \bigcup_{E_i \in \mathcal{U}(E)} \mathcal{FCA}(E_i)$, as we wanted to prove.

Now, we are ready to prove the first main result of this section, which is perfectly symmetrical to Theorem 13, but focusing now in procedural aspects instead in semantic notions. More exactly, we prove that the reductant considered in Definition 10 and the PE^1-reductant are procedurally equivalent.

Theorem 18 *Let $\mathcal{P}$ be a program, A a ground atom and $\mathcal{R} \equiv \langle A \leftarrow @(\mathcal{B}_1, \ldots, \mathcal{B}_n)\theta; \top\rangle$ the reductant for A in $\mathcal{P}$, where $\theta = \theta_1 \cdots \theta_n$ and each substitution θ_i is a matcher of A and the head of a rule $\langle C_i \leftarrow_i \mathcal{B}_i; v_i\rangle$. The PE^1-reductant $\mathcal{R}' \equiv \langle A \leftarrow @_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n); \top\rangle$ (where $\mathcal{D}_i \equiv v_i \&_i \mathcal{B}_i\theta$, $1 \leq i \leq n$) obtained from an unfolding tree of depth one for $\mathcal{P}$ and A, is procedurally equivalent to the reductant $\mathcal{R}$.*

PROOF. In order to state the procedural equivalence between both notions of reductants it suffices to prove that $\mathcal{FCA}(@(\mathcal{B}_1, \ldots, \mathcal{B}_n)\theta) = \mathcal{FCA}(@_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n))$. Due to Proposition 17, if we first perform all the admissible steps with the PE^1-reductant before performing the interpretive ones, we obtain with both reductants the same admissible computed answers (for atom A). Moreover, by the strong correctness of the transformation of interpretive unfolding (see [18]), the interpretive phase considered in [17] leads to the same fuzzy computed answers than the execution of all the interpretive steps defined in [18].

Till here, we have connected the notion of classical reductant with the one of PE^1-reductant, by proposing equivalence results at a semantic and a procedural level (Therorems 13 and 18, respectively). From here, our interest is also to establish equivalence links between PE^1-reductants and PE^k-reductants (Theorem 19). Moreover, we also show that as much a PE-reductant is partially evaluated, the gains in efficiency are more relevant at execution time (Theorem 20).

The following result shows, in a simple formulation, that the f.c.a.'s of the expressions in the leaves of an unfolding tree associated to the PE^1-reductant, for a ground atom A in a program $\mathcal{P}$, coincide with the f.c.a.'s of the expressions in the leaves of an unfolding tree associated to the PE^k-reductant. Therefore, the PE^1-reductant and the PE^k-reductant are procedurally equivalent.

Theorem 19 (Procedural Correctness) *If $\mathcal{R}^1 = \langle A \leftarrow @_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n); \top\rangle$ and $\mathcal{R}^k = \langle A \leftarrow @_{sup}(\mathcal{D}'_1, \ldots, \mathcal{D}'_m); \top\rangle$ are, respectively, the PE^1-reductant and the PE^k-reductant for a ground atom A in a program $\mathcal{P}$, then, $\mathcal{R}^1$ and $\mathcal{R}^k$ are procedurally equivalent, that is, $\mathcal{FCA}(\mathcal{D}_1, \ldots, \mathcal{D}_n) = \mathcal{FCA}(\mathcal{D}'_1, \ldots, \mathcal{D}'_m)$.*

PROOF. By Proposition 17, we know that $\mathcal{FCA}(A) = \mathcal{FCA}(\mathcal{D}_1, \ldots, \mathcal{D}_n) = \mathcal{FCA}(\mathcal{D}_1) \cup \cdots \cup \mathcal{FCA}(\mathcal{D}_n)$ and also, each $\mathcal{FCA}(\mathcal{D}_i) = \mathcal{FCA}(\mathcal{B}_{i_1}, \ldots, \mathcal{B}_{i_r})$, where $\mathcal{B}_{i_1}, \ldots, \mathcal{B}_{i_r}$ are all the successors of $\mathcal{D}_i$, which are the formulas in body of the PE^2-reductant. Following this way as far as reaching the nodes of depth k, we obtain $\mathcal{FCA}(A) = \mathcal{FCA}(\mathcal{D}'_1, \ldots, \mathcal{D}'_m)$ and the result is verified.

And now, we are able to introduce the following result which complements the previous one by confirming the gains in efficiency achieved by PE^k-reductants in comparison with PE^1-reductants.

Theorem 20 (Procedural Efficiency) *If $\mathcal{R}^1 = \langle A \leftarrow @_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n); \top\rangle$ and $\mathcal{R}^k = \langle A \leftarrow @_{sup}(\mathcal{D}'_1, \ldots, \mathcal{D}'_m); \top\rangle$ are, respectively, the PE^1-reductant and the PE^k-reductant of a ground atom A in a program $\mathcal{P}$, then, $\mathcal{R}^k$ is more efficient than $\mathcal{R}^1$ when $k{>}1$, that is, $[\![\mathcal{FCA}]\!](\mathcal{D}_1, \ldots, \mathcal{D}_n) > [\![\mathcal{FCA}]\!](\mathcal{D}'_1, \ldots, \mathcal{D}'_m)$.*

PROOF. This proof is perfectly analogous to the previous one used in Theorem 19, but also exploiting now the second claim (instead of the first one) of Lemma 16.

To finish this section, in the following corollary we connect at a procedural level (as a direct consequence of Theorems 18 and 19) the notion of classical reductant and our improved definition of PE^k-reductant.

Corollary 21 *Let $\mathcal{R} = \langle A\leftarrow @(\mathcal{B}_1, \ldots, \mathcal{B}_n); \top\rangle$, and $\mathcal{R}^k = \langle A\leftarrow @_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_m); \top\rangle$ be, respectively, a reductant according Definition 10 and a PE^k-reductant of a ground atom A in $\mathcal{P}$. Then $\mathcal{R}$ and $\mathcal{R}^k$ are procedurally equivalent, that is, $\mathcal{FCA}(@(\mathcal{B}_1, \ldots, \mathcal{B}_n)) = \mathcal{FCA}(@_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_m))$.*

In the following sub-section we plan to reach a similar result to the previous corollary, but focusing now in semantic properties.

5.2 Semantic Correctness

In Theorem 13 we proved that the concept of an ordinary reductant (see Definition 10) and the notion of PE^1-reductant, were semantically equivalent. Now we are going to establish that the interpretation of the PE^1-reductant is greater or equal than the interpretation of the PE^k-reductant (and therefore, by transitivity, we will show that the interpretation of an ordinary reductant will be greater or equal than the interpretation of the PE^k-reductant). Surprisingly (at least when compared with pure logic programming), this is the strongest result we can achieve in the fuzzy setting, as we are going to detail in what follows.

In order to obtain this result, we need to introduce the immediate consequence operator, $T_\mathcal{P}$, defined by van Emden and Kowalski, and extended in [17] for the framework of multi-adjoint logic programming.

Definition 22 *Let $\mathcal{P}$ be a multi-adjoint program, $\mathcal{I}$ an interpretation and A a ground formula. We define the operator $T_\mathcal{P}$ as a mapping in the set of interpretations such that for each ground atom A*

$$T_\mathcal{P}(\mathcal{I})(A) = sup\{v\dot{\&}_i\mathcal{I}(\mathcal{B}\theta) \mid \langle C\leftarrow_i \mathcal{B}; v\rangle \in \mathcal{P}, A = C\theta\}$$

This operator allows us to formalize the semantics of themulti-adjoint programs: the semantics of a program $\mathcal{P}$ is defined as the least fix point of $T_\mathcal{P}$. It is possible to prove that the operator $T_\mathcal{P}$ fulfills the following suitable property

[17]: an interpretation $\mathcal{I}$ is a model of a multi-adjoint program $\mathcal{P}$ if, and only if, $T_{\mathcal{P}}(\mathcal{I}) \subseteq \mathcal{I}$. Therefore, for every model $\mathcal{I}$ of $\mathcal{P}$ we have that

$$T_{\mathcal{P}}(\mathcal{I})(A) = sup\{v \dot{\&}_i \mathcal{I}(\mathcal{B}\theta) \mid \langle C \leftarrow_i \mathcal{B}; v\rangle \in \mathcal{P}, A = C\theta\} \leq \mathcal{I}(A)$$

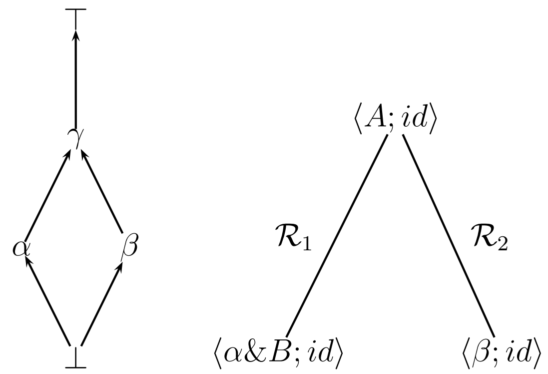
Thus, the interpretation of a ground atom A, by a model $\mathcal{I}$ of $\mathcal{P}$, is greater or equal than the interpretation of the body of the PE^1-reductant corresponding to this atom.

Moreover, in general, the equality $T_{\mathcal{P}}(\mathcal{I})(A) = \mathcal{I}(A)$ does not hold. Roughly speaking, it tells us that "as much as an atom is evaluated/interpreted, its truth degree might even decrease". The following example illustrates this fact.

Example 23 *Let $\mathcal{P}$ be the following program, where A, B are any ground atoms; let $(L, \leq)$ be the multi-adjoint lattice associated to $\mathcal{P}$ described by the diagram of the figure below, in which we have also included the unfolding tree for A in $\mathcal{P}$. Let $(\&, \leftarrow)$ be the adjoint pair in $(L, \leq)$ collected from the* Gödel *intuistionistic logic, such that the truth functions for $\&$ and $\leftarrow$ are defined respectively by $\dot{\&}(x, y) = inf\{x, y\}$ and* $\dot{\leftarrow}(y, x) = \begin{cases} \top, & if\ x \leq y \\ y, & otherwise \end{cases}$

$\mathcal{R}_1 : \langle A \leftarrow B;\ \alpha\rangle$

$\mathcal{R}_2 : \langle A \leftarrow\ ;\quad \beta\rangle$



Then, for every interpretation $\mathcal{I} : B_{\mathcal{P}} \longrightarrow L$ such that $\mathcal{I}(A) = \top$, $\mathcal{I}(B) = \gamma$, $\mathcal{I}$ is a model of $\mathcal{P}$ since $\alpha \dot{\&} \mathcal{I}(B) \leq \mathcal{I}(A), \beta \leq \mathcal{I}(A)$ and, moreover, it fulfills: $T_{\mathcal{P}}(I)(A) = sup\{\alpha \dot{\&} \mathcal{I}(B), \beta\} = sup\{inf\{\alpha, \gamma\}, \beta\} = sup\{\alpha, \beta\} = \gamma < \top = \mathcal{I}(A)$.

The following Lemma will allow us to relate the interpretation of the body of the PE^1-reductant to the one of the body of the PE^2-reductant, and as a consequence, with the interpretation of the body of the PE^k-reductant.

Lemma 24 *Let A be a ground atom that matches with the head of the rules $\langle C_1 \leftarrow_1 \mathcal{B}_1; v_1\rangle, \ldots, \langle C_i \leftarrow_i B_i; v_i\rangle, \ldots, \langle C_r \leftarrow_r \mathcal{B}_r; v_r\rangle$ in $\mathcal{P}$ (substitution θ_i being such that $C_i\theta_i = A$, one of the matchers). Assume that formula $B_i\theta_i$ unifies* *with the head of rules $\langle H_1 \leftarrow^1 \mathcal{D}_1; u_1\rangle, \ldots, \langle H_s \leftarrow^s \mathcal{D}_s; u_s\rangle$ in $\mathcal{P}$. Then, if $\mathcal{I}$ is a model of $\mathcal{P}$, $L_2 \le L_1$, where:*

- $L_1 = sup\{v_1 \dot{\&}_1 \mathcal{I}(\mathcal{B}_1\theta_1), \ldots, v_i \dot{\&}_i \mathcal{I}(B_i\theta_i), \ldots, v_r \dot{\&}_r \mathcal{I}(\mathcal{B}_r\theta_r)\}$
- $L_2 = sup\{v_1 \dot{\&}_1 \mathcal{I}(\mathcal{B}_1\theta_1), \ldots, v_i \dot{\&}_i (u_1 \dot{\&}^1 \mathcal{I}(\mathcal{D}_1\sigma_1)), \ldots, v_i \dot{\&}_i (u_s \dot{\&}^s \mathcal{I}(\mathcal{D}_s\sigma_s)), \ldots, v_r \dot{\&}_r \mathcal{I}(\mathcal{B}_r\theta_r)\}$

Moreover, $\mathcal{I}(A) \ge T_{\mathcal{P}}(\mathcal{I})(A) \ge sup\{v_1 \dot{\&}_1 \mathcal{I}(\mathcal{B}_1\theta_1), \ldots, v_i \dot{\&}_i (u_1 \dot{\&}^1 \mathcal{I}(\mathcal{D}_1\sigma_1)), \ldots, v_i \dot{\&}_i (u_s \dot{\&}^s \mathcal{I}(\mathcal{D}_s\sigma_s)), \ldots, v_r \dot{\&}_r \mathcal{I}(\mathcal{B}_r\theta_r)\}$.

PROOF.

Since the atom $B_i\theta_i$ unifies with the head of all the rules $\langle H_1 \leftarrow^1 \mathcal{D}_1; u_1\rangle, \ldots, \langle H_s \leftarrow^s \mathcal{D}_s; u_s\rangle$, there is θ_j such that $(B_i\theta_i)\theta_j = H_j\theta_j$, for each $j = 1, \ldots, s$. By the adjoint property, a model interpretation $\mathcal{I}$ of $\mathcal{P}$ fulfills that $u_j \dot{\&}_j \mathcal{I}(D_j\theta_j) \le \mathcal{I}(H_j\theta_j) = \mathcal{I}((B_i\theta_i)\theta_j) \le \mathcal{I}(B_i\theta_i)$ for all $j = 1, \ldots, s$ and, since & is increasing in both arguments, we have that $v_i \dot{\&}_i (u_j \dot{\&}_j \mathcal{I}(D_j\theta_j)) \le v_i \dot{\&}_i \mathcal{I}(B_i\theta_i)$. By definition of least upper bound, it turns out that $sup\{v_i \dot{\&}_i (u_1 \dot{\&}^1 \mathcal{I}(\mathcal{D}_1\sigma_1)), \ldots, v_i \dot{\&}_i (u_s \dot{\&}^s \mathcal{I}(\mathcal{D}_s\sigma_s))\} \le v_i \dot{\&}_i \mathcal{I}(B_i\theta_i)$ and therefore $L_2 \le L_1$.

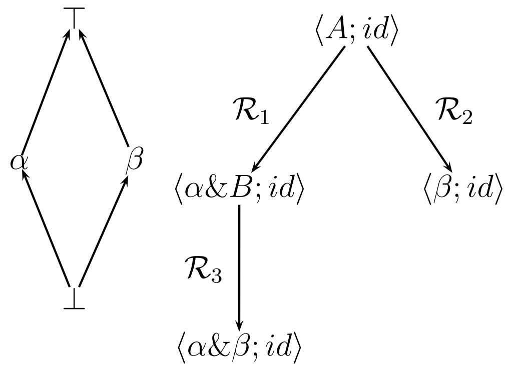
Finally, if $\mathcal{I}$ is a model of $\mathcal{P}$, we have $\mathcal{I}(A) \ge T_{\mathcal{P}}(\mathcal{I})(A) \ge sup\{v_1 \dot{\&}_1 \mathcal{I}(\mathcal{B}_1\theta_1), \ldots, v_i \dot{\&}_i (u_1 \dot{\&}^1 \mathcal{I}(\mathcal{D}_1\sigma_1)), \ldots, v_i \dot{\&}_i (u_s \dot{\&}^s \mathcal{I}(\mathcal{D}_s\sigma_s)), \ldots, v_r \dot{\&}_r \mathcal{I}(\mathcal{B}_r\theta_r)\}$, using the inequality $L_1 \ge L_2$.

Observe that rule $\langle A \leftarrow @_{sup}(v_1 \&_1 \mathcal{B}_1\theta_1, \ldots, v_i \&_i B_i\theta_i, \ldots, v_r \&_r \mathcal{B}_r\theta_r); \top\rangle$, associated with expression L_1, is a PE^1-reductant for the ground atom A in $\mathcal{P}$, and rule $\langle A \leftarrow @_{sup}(v_1 \&_1 \mathcal{B}_1\theta_1, \ldots, v_i \&_i (u_1 \&^1 \mathcal{D}_1\sigma_1), \ldots, v_i \&_i (u_s \&^s \mathcal{D}_s\sigma_s), \ldots, v_r \&_r \mathcal{B}_r\theta_r); \top\rangle$, associated with expression L_2, is a PE^2-reductant.

If $\mathcal{R}^1 = \langle A \leftarrow @_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n); \top\rangle$, $\mathcal{R}^2 = \langle A \leftarrow @_{sup}(\mathcal{D}'_1, \ldots, \mathcal{D}'_m); \top\rangle$ are, respectively, a PE^1-reductant and a PE^2-reductant of a ground atom A in a program $\mathcal{P}$, then, as the following example shows, $\mathcal{I}(@_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n))$ and $\mathcal{I}(@_{sup}(\mathcal{D}'_1, \ldots, \mathcal{D}'_m))$ do not coincide in general (in concordance with the fact we pointed out at the beginning of this section).

Example 25 *Let $\mathcal{P}$ be the following program, where A and B are any ground atoms; let $(L, \le)$ be the multi-adjoint lattice associated to $\mathcal{P}$ described by the diagram of the figure below and consider again the adjoint pair based on the* Gödel *intuistionistic logic used in Example 23.*

$$\mathcal{R}_1 : \langle A \leftarrow B;\ \alpha \rangle$$
$$\mathcal{R}_2 : \langle A \leftarrow\ ;\ \ \beta \rangle$$
$$\mathcal{R}_3 : \langle B \leftarrow\ ;\ \ \beta \rangle$$



Then, given an interpretation $\mathcal{I} : B_{\mathcal{P}} \longrightarrow L$ *such that* $\mathcal{I}(A) = \mathcal{I}(B) = \top$, $\mathcal{I}$ *is a model of* $\mathcal{P}$ *since* $\alpha \dot{\&} \mathcal{I}(B) \leq \mathcal{I}(A), \beta \leq \mathcal{I}(A), \beta \leq \mathcal{I}(B)$. *Moreover, the* PE^1*-reductant is the rule* $\langle A \leftarrow @_{sup}(\mathcal{D}_1, \mathcal{D}_2); \top \rangle = \langle A \leftarrow @_{sup}(\alpha \& B, \beta); \top \rangle$ *and the* PE^2*-reductant the rule* $\langle A \leftarrow @_{sup}(\mathcal{D}'_1, \mathcal{D}'_2); \top \rangle = \langle A \leftarrow @_{sup}(\alpha \& \beta, \beta); \top \rangle$*, which fulfills:*
$\mathcal{I}(@_{sup}(\mathcal{D}'_1, \mathcal{D}'_2)) = sup\{\alpha \dot{\&} \beta, \beta\} = sup\{inf\{\alpha, \beta\}, \beta\} = sup\{\bot, \beta\} = \beta < \top = sup\{\alpha, \beta\} = sup\{inf\{\alpha, \top\}, \beta\} = sup\{\alpha \dot{\&} \mathcal{I}(B), \beta\} = \mathcal{I}(@_{sup}(\mathcal{D}_1, \mathcal{D}_2))$.

The following result complements at a semantic level the (procedural) Theorem 19 by generalizing Lemma 24 and relating the interpretation (within a model of $\mathcal{P}$) of the bodies of PE^1-reductants and PE^k-reductants (for ground atom A).

Theorem 26 (Semantic Correctness) *If* $\mathcal{R}^1 = \langle A \leftarrow @_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n); \top \rangle$ *and* $\mathcal{R}^k = \langle A \leftarrow @_{sup}(\mathcal{D}'_1, \ldots, \mathcal{D}'_m); \top \rangle$ *are, respectively, the* PE^1*-reductant and the* PE^k*-reductant of a ground atom* A *in a program* $\mathcal{P}$*, then,* $\mathcal{R}^1$ *and* $\mathcal{R}^k$ *fulfills* $\mathcal{I}(@_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n)) \geq \mathcal{I}(@_{sup}(\mathcal{D}'_1, \ldots, \mathcal{D}'_m))$ *where* $\mathcal{I}$ *is a model of* $\mathcal{P}$.

PROOF. The proof is made by recurrence. By Theorem 24, if $\mathcal{I}$ is a model of $\mathcal{P}$, $\mathcal{I}(@_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n)) = L_1 \geq L_2 = \mathcal{I}(@_{sup}(\mathcal{B}'_1, \ldots, \mathcal{B}'_r))$, where $\mathcal{R}' = \langle A \leftarrow @_{sup}(\mathcal{B}'_1, \ldots, \mathcal{B}'_r); \top \rangle$ is the PE^2-reductant. Iterating the process $k-1$ times (by unfolding, in every step, the formulas in the body of the reductant that has been obtained in the previous step), we have that $\mathcal{I}(@_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_n)) \geq \mathcal{I}(@_{sup}(\mathcal{D}'_1, \ldots, \mathcal{D}'_m))$ where rule $\mathcal{R}' = \langle A \leftarrow @_{sup}(\mathcal{D}'_1, \ldots, \mathcal{D}'_m); \top \rangle$ is the PE^k-reductant, which concludes the proof.

We finish this section in a similar way to the previous one, by proposing the following corollary which this time connects at a semantic level (as a direct consequence of Theorems 13 and 26) the notion of classical reductant and our improved definition of PE^k-reductant.

Corollary 27 *Let* $\mathcal{R} \equiv \langle A \leftarrow @(\mathcal{B}_1, \ldots, \mathcal{B}_n); \top\rangle$ *and* $\mathcal{R}^k \equiv \langle A \leftarrow @_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_m)\ ; \top\rangle$ *be, respectively, a reductant according Definition 10 and a* PE^k*-reductant,* $k \geq 1$*, of a ground atom* A *in a program* $\mathcal{P}$*. Then,* $\mathcal{R}$ *and* $\mathcal{R}^k$ *fulfills* $\mathcal{I}(@(\mathcal{B}_1, \ldots, \mathcal{B}_n)) \geq \mathcal{I}(@_{sup}(\mathcal{D}_1, \ldots, \mathcal{D}_m))$*.*

6 Final Discussion

This section is devoted to relate both the underlying language and the transformation techniques presented in this work with other approaches appeared in the literature. At the same time, we also plan to provide some implementation issues regarding our proposal.

Underlying language. As it was commented, our work is concerned with the introduction of partial evaluation techniques for specializing programs and computing efficient reductants. The selection of the underlying language (based in the multi-adjoint logic programming approach of [15–17]) has been mainly motivated by its high level of expressiveness as well as for its clear procedural semantics. The first point is useful for increasing the relevance and generality of our results, also contributing to a broader dissemination, whereas the second point is crucial to effectively define our transformations (in particular, for a formal definition of unfolding rules and partial evaluation techniques, a procedural semantics formalized in terms of a state transition system seems to be mandatory).

Regarding expressiveness, the multi-adjoint logic programming approach represents an extremely flexible fuzzy framework based on weighted rules, which largely improves older approaches previously introduced in this field (see, for instance, the Prolog–Elf system of [27], the Fril system of [28] and the fuzzy variants of Prolog proposed in [29–31]). Moreover, in [32] [7], the authors justify the need for considering such a very abstract framework in order to be able to state and prove general results on termination, fix-point semantics, query answering procedures, etc..., which can apply to several apparently distinct settings such as van Emden's Quantitative Deduction, Possibilistic Logic Pro-

[7] In this last work, a sorted version of the multi-adjoint logic language is proposed. This extension does not seriously affect our techniques and therefore, in order to preserve the readability of this paper and our methods, we postpone their adaptation to the sorted case as future work.

gramming, Non-classical SLD resolution, Ordinary Probabilistic Logic Programs and Probabilistic Deductive Databases.

Due to the increasing interest in models of reasoning under "imperfect" information, we have observed, in recent years, the proliferation of an enormous number of proposals for the integration of approximate reasoning into the context of (classical) Logic Programming. Consequently, there exist some cases not fully covered by the multi-adjoint logic approach. These are, for instance, the cases of similarity-based [33] and annotated [34] logic programming. Nevertheless, for the first case, we can find in [17] some (theoretical) analysis establishing nice correspondences between both languages. In particular, it can be proved that the effects of the similarity-based unification/resolution methods of [33] can be somehow replicated (at a theoretical level) by applying the procedural mechanism seen here on multi-adjoint logic programs augmented with special weighted rules which simulate similarity equations. On the other hand, for the annotated logic programs of [34] and specially, for the more recent (and much easier in its formulation) version of [35], we need to make a more detained analysis.

In contrast with [35], most of the reported approaches (including the multi-adjoint logic one[8]) exhibit an important limitation, as they do not address any mode of non-monotonic negation. We can compare this approach with the one used along this paper at the following levels:

- **Expressiveness**. Apart from the fact that in the multi-adjoint logic approach the discussion is centered on the monotonic case, ignoring default negation, the underlying notion of lattice is different from the concept of bilattice [36]. Bilattices are slightly more general structure than lattices. They are able to cope with non-monotonic negation and provide an elegant, powerful way for combining belief and doubt degrees on program rules. Although this gap of the multi-adjoint framework has been alleviated in [37] by considering multiple sorts via the so called multi-lattices, the proposed treatment seems to be anyway weaker than the one carried out in [35].
- **Syntax**. Program rules in [35] have the form $\mathcal{A} \leftarrow f(\mathcal{B}_1, \ldots, \mathcal{B}_n)$ where f is an operator interpreted as a computable truth combination function, whereas $\mathcal{A}$ and $\mathcal{B}_i$ are atoms. Note that this syntax is very close to the one used in the multi-adjoint logic approach, where as explicitly said in [32] "sometimes, we will represent bodies of formulas as $@[\mathcal{B}_1, \ldots, \mathcal{B}_n]$, where ... @ is the monotone aggregator obtained as a composition"[9]. As seen so far, both approaches are quite similar from a syntactic point of view, apart

[8] In [32], authors claim that they "have already started the research in this direction".

[9] See also [38], where we provide a transformation operation called "aggregation" which automatically adapts program rules to this clearer shape.

from the different representation of lattices commented before. However, no function symbols (apart of constants) are allowed in [35] where to ease the presentation, the attention is limited to the propositional case (authors argue that the "first order case can be handled by grounding").

- **Procedural semantics**. Maybe the major differences between both approaches, emerge at the procedural semantics level. We have seen that the formalization (as a transition system) presented in sub-section 2.2 was crucial for effectively defining the construction of unfolding trees in Section 3. In [35] a general top-down query answering procedure for normal logic programs over lattices and bilattices is provided. The method could be understood as a procedural semantics for such kind of programs, and has the advantage that it can be instantiated for "mirroring" differen semantics (as the Kripke-Kleene and the Well-Founded semantics) when evaluating goals. Unfortunately, the query answering procedure is presented in an algorithmic way which is far away from the desirable transition-system like formulation traditionally used in program transformation tasks.

Now, putting together all the pieces presented before, we can extract the following three conclusions: i) the framework of [35] is a challenging approach for which (once surpassed the propositional case and/or allowed the presence of function symbols in its syntax) we are interested in the adaptation of our partial evaluation techniques in order to capture important expressive resources including non-monotonic negation; ii) in order to do this, it is mandatory to previously investigate in new re-formulation ways of its procedural semantics and query answering mechanisms; and iii) we think that the experience we can accumulate developing the techniques proposed in this paper (for the multi-adjoint logic approach), will largely help us to generalize our results to the alternative fuzzy framework of [35] in the near future.

Tabulation techniques. As it was reported in [35], any logic programming language should be accompanied with query answering procedures. In this paper we are not directly concerned with this subject, having into account that our main goals are twofold: i) to specialize programs; and ii) to generate sophisticated (partially evaluated) reductants which, once added to original programs, contribute to a correct, complete and efficient evaluation of goals using the procedural semantics described in sub-section 2.2. Nevertheless, inspired by the tabulation goal-oriented query procedure presented in [39,40] (for residuated and multi-adjoint logic programs) and extended in [32] (for sorted multi-adjoint logic programs), we think that it is possible to improve the efficiency of both, PE-reductant calculus and query answering w.r.t. those programs generated by our partial evaluation techniques.

The underlying idea of tabulation (tabling, or memoising) is, essentially, that atoms of selected tabled predicates as well as their answers are stored in a table. When an identical atom is recursively called, the selected atom is not resolved against program clauses; instead, all corresponding answers computed so far are looked up in the table and the associated answer substitutions are applied to the atom. The process is repeated for all subsequent computed answer substitutions corresponding to the atom.

Tabulation methods have been largely used in the past to increase the efficiency of proof procedures. The usual SLD based implementations of Fuzzy Logic Programming languages (e.g. [30]) are goal-oriented and inherit the problems of non-termination and re-computation of goals. For grappling with these problems, tabulation implementation techniques have been proposed in the deductive databases and logic programming communities. More recently, an extension of SLD for implementing generalised annotated logic programs has been proposed in [34,41]. Following these ideas, the tabulation goal-oriented query procedure provided in [32] is accompanied with interesting termination results considering a significant class of sorted multi-adjoint logic programs but only applicable to ground goals [10].

It is important to note that in [32] complete trees are generated for executing goals; hence, the termination results provided there are mandatory. However, the use of tabulation techniques is neither directly dependent on the shape of the generated trees (finite or infinite, partial or complete) nor on their final use (query answering, goal solving, program transformation, partial evaluation, etc.). On the other hand, partial evaluation techniques are also based on the generation of (incomplete) search trees. Therefore, the more intelligent the method for generating trees is (by using, for instance, tabulation or thresholding techniques), the more efficient will be the final application of such trees in practice. So, we think that tabulation techniques can be successfully embedded inside a transformation task such as the partial evaluation method we are dealing with.

In particular, the calculus of partial unfolding trees presented in Section 3 admits a tabulation based reformulation near to the one proposed in [32], but with the advantage in our case that since we simply generate partial trees during the PE process, we are not directly limited for any termination constraint. In this sense, we also plan to take advantage of the experience acquired in [42], where we introduced thresholding techniques for dynamically reducing the size of unfolding trees. Our thresholding techniques have some correspondences with the tabulation method used in [32] in the sense that we also store/update/compare the current best truth degree associated to a given (first order) atom when building its associated (partial) unfolding tree. Anyway, more research is needed to relate tabulation and thresholding

[10] In [32], although it is argued that there is no loss of generality since infinite programs are allowed, authors also make the claim that it is important to extend this technique to the first order case as future work.

if we really want to take profit of their simultaneous use when computing reductants, specializing programs and solving goals in the multi-adjoint logic setting. In this sense, some prioritary lines of future work are:

- The techniques presented so far here as well as those described in [42] can be extended to cope with the possibility of tabulating several thresholds instead of a single one [11] .
- Our experience in managing trees where nodes contain atoms with variables, might help to lift the tabulation method of [32] to both, the non-ground case and the first-order case.
- Finally, we also think that the proper tabulated query answering procedure of [32] admits an almost immediate characterization as a procedural semantics muc more efficient than the one we have used along this paper.

Regarding this last point, we think that such (tabulation based) procedural semantics can be conceived as an state transition system, where states might contain tree forest instead of simple goals: the great amount of information provided by forest will be the key point for to efficiently formulate the new improved procedural mechanism.

Implementation issues. As we announced in [18], the implementation of a prototype interpreter/compiler for the multi-adjoint logic language was a prioritary task proposed as future work. In [43] we have recently reported our preliminary results in such direction, also describing a powerful method for translating fuzzy programs into directly executable standard Prolog code. The final goal is that the compiled code be executed in any Prolog interpreter in a completely transparent way for the final user, i.e., our intention is that after introducing fuzzy programs and fuzzy goals to the system, it be able to return fuzzy computed answers (i.e., pairs including truth degrees and substitutions) even when all intermediate computations have been executed in a pure (not fuzzy) logic programming environment.

Our approach is somehow inspired by [31], where an interpreter conceived using Constraint Logic Programming over real numbers ($CLP(\mathcal{R})$) has been efficiently implemented for a fuzzy logic language close to ours. Anyway, we use the Prolog programming language without considering a $CLP(R)$ extension, since it suffices for implementing our ideas. Nowadays, our systems only deals with programs focusing in a simple lattice whose carrier set is the real interval $[0, 1]$ and the connectives are collected from standard fuzzy logic (as the product, Łukasiewicz and Gödel intuitionistic logic). Although the expres-

[11] Note that in our case, since unfolding trees have been mainly used for computing reductants, we only needed to generate a single tree for a given atom. However, in the general case, when partial evaluation techniques are used for specialization purposes, they really generate tree forest, which implies the need for tabulating several values as it is similarly done in [32].

sive power of our preliminary implementation is rather limited, a high-priority task for future developments will be to let our system accept fuzzy programs as well as multi-adjoint lattices in a parametric way, which implies the design of appropriate protocols, interfaces, etc.

We have used Sicstus Prolog v.3.12.5 for executing fuzzy programs once translated to Prolog code, as well as for implementing the proper tool. Our parser has been developed by using the classical DCG's (*Definite Clause Grammars*) resource of the Prolog language, since it is a convenient notation for expressing grammar rules. The application contains about 300 clauses and once it is loaded inside a Prolog interpreter (in our case, Sicstus Prolog), it shows a menu which includes (among others) options for:

• `Loading` a prolog file with extension `'.pl'`. This action is useful for reading a file containing a set of clauses implementing aggregators, user predicates, etc. Nevertheless, the original connectives of the *Product*, *Gödel* and *Łukasiewicz logic*, expressed in the Prolog style seen in the previous section, are defined in file `prelude.pl`, which is automatically loaded by the system at the beginning of each work session.

• `Parsing` a fuzzy program included in a file with extension `'.fpl'`. In order to simultaneously perform the parsing process with the code generation, each *parsing* predicate used in DCG's rules, has been augmented with a variable as extra argument which is intended to contain the Prolog code generated after parsing the corresponding fragment of fuzzy code.

• `Listing` the set of Prolog clauses loaded from a `'.pl'` file as well as those ones obtained after compiling an `'.fpl'` file. Of course, the original fuzzy program contained in this last file is also displayed.

• `Saving` the resulting Prolog code into a file, and finally

• `Executing` a fuzzy goal after being introduced from the keyboard.

Therefore, our implementation system translates a multi-adjoint logic program and goal to standard Prolog code, allowing the execution of fuzzy programs inside a classical logic programming environment. A detailed explanation of the implementation techniques we use can be found in [43].

It is important to note that, in our implementation system, all internal computations (including compiling and executing) are pure Prolog derivations whereas inputs (fuzzy programs and goals) and outputs (fuzzy computed answers) have always a fuzzy taste, which generates the illusion on the user of being working with a purely fuzzy tool. However, when trying to go beyond goal solving and program execution, our method becomes insufficient. In particular, observe that we can only simulate complete fuzzy derivations (by performing the corresponding Prolog derivations based on SLD-resolution) but we can not generate partial derivations or even apply a single admissible/interpretive step on a given fuzzy expression. This kind of low-level manipulations are mandatory when trying to incorporate to the tool mechanisms for generating derivations of a fixed number of steps, rearranging the body of a program rule, applying substitutions to its head, etc... in order to implement some program transformation techniques such as the ones presented in this paper as well as those described in [20,18,42].

To achieve this aim, we have conceived a new low-level representation for the fuzzy code: each *parsing* predicate used in DCG's rules (which already contains a parameter allocating the Prolog code obtained after the compilation process) has also been augmented with a second extra argument for storing now the new representation associated to the corresponding fragment of parsed fuzzy code. Details on such low-level representation of the fuzzy code can be found in [43]. Anyway, although the new method allows us nowadays to build unfolding trees with any level of depth (observe that the correct manipulation of the leaves of this kind of partially evaluated trees, opens the door to produce unfolded rules, specialized program, PE-reductants, and so on), we continue investigating on new implementation methods for extending the tool. In particular, we are nowadays studying the recent proposal presented in [32], where an implementation of the tabulation procedure discussed before is underway using the GAP package of XSB Prolog, as well as a distributed implementation for the use in the Semantic Web.

7 Conclusions and Further Research

In this work we have defined, for the first time, a partial evaluation technique for multi-adjoint logic programs. We have shown its capability to specialize programs, as it happens in other (declarative) programming contexts. Moreover, we have introduced a method for computing reductants based on partial evaluation techniques, which constitutes a novel application for the multi-adjoint logic programming framework. The established relationship between reductants and its refined construction by means of partial evaluation techniques, is an important issue if we really want to build complete and efficient systems for the multi-adjoint logic programming framework. Such relationships have been strengthened as much as possible, by providing formal proofs regarding procedural and semantic correctness results, as well as efficiency criteria.

As we advanced in the previous section, there are several ways for extending and improving the partial evaluation framework and the PE-reductant calculus presented in this work:

(1) To overcome the restriction to specialize atom goals separately, thus allowing the specialization of more complex goals to achieve a better specialization. This can be done by introducing some particular techniques able to adequately specialize aggregations of atoms, similarly as it has been proposed in the field of conjunctive partial deduction [25,26] for the specialization of conjunctions of atoms.
(2) To introduce more refined *local control* strategies, like methods based on well-founded orders or well-quasi orders [11], instead of imposing an arbitrary *ad hoc* depth bound for the unfolding. Also it is necessary to introduce an effective procedure for the partial evaluation of a multi-adjoint logic program with regard to a set of (goal) atoms. This problem links with the definition of *global control* strategies. The global level of control concerns with the termination of recursive unfolding, or how to stop recursively constructing unfolding trees while still guaranteeing that the desired amount of specialization is retained and that the semantic correctness of the partial evaluation process is reached. Note that, when a program is partially evaluated, the set of goals appearing in the initial set (with regard the specialization is performed) usually needs to be augmented in order to obtain an effective specialization. These new goals must be recursively unfolded (generating new unfolding trees in a similar way as the tabulation technique of [32] generates tree forests) to complete the specialization of the program. Therefore, to guarantee the termination of the partial evaluation process is important to investigate some appropriate technique able to keep this set finite.
(3) To improve the method for the construction of the so called PE-reductants by means of a refined algorithm based on unfolding with a set of dynamic thresholds (we have some preliminary results in this line [42]) and tabulation techniques (in the style of [39,40,32]), in order to prune tree branches and properly decide which nodes in the unfolding tree must be selected to be exploited.
(4) To allow the presence of variables in the heads of PE-reductants. We think that this action will be especially interesting in practice when considering that all the arguments in these heads be variables, since it might be the key allowing us to attach the notion of reductant to a (finite) set of predicate symbols instead to a (possibly infinite) set of ground atoms.
(5) Last, but no least, it would be interesting to investigate a new, more powerful procedural semantics based on tabulation techniques [32] and capturing non-monotonic negation [35] in order to increase both its efficiency and expressiveness. In a second research stage, we think that those procedural mechanisms, once formulated as state transition systems, would become excellent platforms to support improved partial evaluation and other transformation techniques.

References

[1] N. Jones, C. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, Englewood Cliffs, NJ, 1993.

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

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

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

[5] C. Consel, O. Danvy, Tutorial notes on Partial Evaluation, in: Proc. of 20th Annual ACM Symp. on Principles of Programming Languages, ACM, New York, 1993, pp. 493–501.

[6] V. Turchin, The Concept of a Supercompiler, ACM Transactions on Programming Languages and Systems 8 (3) (1986) 292–325.

[7] J. Gallagher, Tutorial on Specialisation of Logic Programs, in: Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993, ACM, New York, 1993, pp. 88–98.

[8] H. Komorowski, Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog, in: Proc. of 9th ACM Symp. on Principles of Programming Languages, 1982, pp. 255–267.

[9] J. Lloyd, J. Shepherdson, Partial Evaluation in Logic Programming, Journal of Logic Programming 11 (1991) 217–242.

[10] M. Alpuente, M. Falaschi, G. Vidal, Partial Evaluation of Functional Logic Programs, ACM Transactions on Programming Languages and Systems 20 (4) (1998) 768–844.

[11] E. Albert, M. Alpuente, M. Falaschi, P. Julián, G. Vidal, Improving Control in Functional Logic Program Specialization, in: G. Levi (Ed.), Proc. of Static Analysis Symposium, SAS'98, Springer, Lecture Notes in Computer Science 1503, 1998, pp. 262–277.

[12] E. Albert, M. Alpuente, M. Hanus, G. Vidal, A Partial Evaluation Framework for Curry Programs, in: Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning, LPAR'99, Springer, Lecture Notes in Artificial Intelligence 1705, 1999, pp. 376–395.

[13] C. Consel, L. Hornof, F. Noël, J. Noyé, E. Volanschi, A Uniform Approach for Compile-Time and Run-Time Specialisation, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation, Springer, Lecture Notes in Computer Science 1110, 1996, pp. 54–72.

[14] R. Glück, M. Sørensen, Partial Deduction and Driving are Equivalent, in: Proc. Int'l Symp. on Programming Language Implementation and Logic Programming, PLILP'94, Springer, Lecture Notes in Computer Science 844, 1994, pp. 165–181.

[15] J. Medina, M. Ojeda-Aciego, P. Vojtáš, Multi-adjoint logic programming with continuous semantics, Proc of Logic Programming and Non-Monotonic Reasoning, LPNMR'01, Springer-Verlag, Lecture Notes in Artificial Intelligence 2173 (2001) 351–364.

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

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

[18] P. Julián, G. Moreno, J. Penabad, Operational/Interpretive Unfolding of Multi-adjoint Logic Programs, Journal of Universal Computer Science 12 (11) (2006) 1679–1699.

[19] M. Alpuente, M. Falaschi, P. Julián, G. Vidal, Specialization of Lazy Functional Logic Programs, in: Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97, Vol. 32, 12 of Sigplan Notices, ACM Press, New York, 1997, pp. 151–162.

[20] P. Julián, G. Moreno, J. Penabad, On Fuzzy Unfolding. A Multi-adjoint Approach, Fuzzy Sets and Systems, Elsevier 154 (2005) 16–33.

[21] J. Komorowski, An Introduction to Partial Deduction, in: A. Pettorossi (Ed.), Meta-Programming in Logic, Uppsala, Sweden, Springer Lecture Notes in Computer Science 649, 1992, pp. 49–69.

[22] J. L. Lassez, M. J. Maher, K. Marriott, Unification Revisited, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Los Altos, Ca., 1988, pp. 587–625.

[23] P. Julián, G. Moreno, J. Penabad, Evaluación Parcial de Programas Lógicos Multi-adjuntos y Aplicaciones, in: A. Fernández (Ed.), Proc. of Campus Multidisciplinar en Percepción e Inteligencia, CMPI-2006, Albacete, Spain, July 10-14, UCLM, 2006, pp. 712–724.

[24] M. Fay, First Order Unification in an Equational Theory, in: Proc of 4th Int'l Conf. on Automated Deduction, 1979, pp. 161–167.

[25] R. Glück, J. Jørgensen, B. Martens, M. Sørensen, Controlling Conjunctive Partial Deduction of Definite Logic Programs, in: Proc. Int'l Symp. on Programming Languages: Implementations, Logics and Programs, PLILP'96, Springer, Lecture Notes in Computer Science 1140, 1996, pp. 152–166.

[26] M. Leuschel, D. De Schreye, A. de Waal, A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration, in: M. Maher

(Ed.), Proc. of the Joint International Conference and Symposium on Logic Programming, JICSLP'96, The MIT Press, Cambridge, MA, 1996, pp. 319–332.

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

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

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

[30] P. Vojtáš, L. Paulík, Soundness and completeness of non-classical extended SLD-resolution, in: R. Dyckhoff et al (Ed.), Proc. ELP'96 Leipzig, LNCS 1050, Springer Verlag, 1996, pp. 289–301.

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

[32] C. Damásio, J. Medina, M. Ojeda-Aciego, Termination of logic programs with imperfect information: applications and query procedure, Journal of Applied Logic 5 (2007) 435–458.

[33] M. Sessa, Approximate reasoning by similarity-based SLD resolution, Fuzzy Sets and Systems 275 (2002) 389–426.

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

[35] U. Straccia, Query answering in normal logic programs under uncertainty, in: L. Godó (Ed.), Proc. of 8th. European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-05), Barcelona, Spain, Lecture Notes in Computer Science 3571, Springer Verlag, 2005, pp. 687–700.

[36] M. L. Ginsberg, Multi-valued logics: a uniform approach to reasoning in artificial intelligence, Computational Intelligence 4 (1988) 265–316.

[37] J. Medina, M. Ojeda-Aciego, J. Ruiz-Calviño, On the ideal semantics of multilattice-based logic programs, in: Information Processing and Management of Uncertainty, IPMU'06, 2006, pp. 463–470.
URL `./TR/ipmu2006-ss.pdf`

[38] J. Guerrero, G. Moreno, Optimizing Fuzzy Logic Programs by Unfolding, Aggregation and Folding, in: J. Visser, V. Winter (Eds.), Proc. of the 8th. International Workshop on Rule-Based Programming, RULE-07, Paris, France, June 29, Electronic Notes in Theoretical Computer Science (to appear), 2007, p. 15.

[39] C. Damásio, J. Medina, M. Ojeda-Aciego, A tabulation proof procedure for residuated logic programming, In Proc. of the European Conference on Artificial Intelligence, Frontiers in Artificial Intelligence and Applications 110 (2004) 808–812.

[40] C. Damásio, J. Medina, M. Ojeda-Aciego, Sorted multi-adjoint logic programs: termination results and applications, in: Proc. of Logics in Artificial Intelligence, JELIA'04, Springer, Lecture Notes in Artificial Intelligence 3229, 2004, pp. 260–273.

[41] T. Swift, Tabling for non-monotonic programming, Annals of Mathematics and Artificial Intelligence 25 (3–4) (1999) 201–240.

[42] P. Julián, G. Moreno, J. Penabad, Efficient reductants calculi using partial evaluation techniques with thresholding, in: P. Lucio (Ed.), Electronic Notes in Theoretical Computer Science, Vol. 188C, Elsevier, 2007, pp. 77–90.

[43] J. Abietar, P. Morcillo, G. Moreno, Building a Fuzzy Logic Programming Tool, in: E. Pimentel (Ed.), Proc. of VII Jornadas sobre Programación y Lenguajes, PROLE'2007, Zaragoza, Spain, September 12-14, University of Zaragoza, 2007, pp. 215–222.