Efficient Reductants Calculi using Partial Evaluation Techniques with Thresholding [1]

Pascual Julián, Ginés Moreno, Jaime Penabad [2]

University of Castilla-La Mancha, Spain

Abstract

Reductants are a useful theoretical tool introduced for proving correctness properties in the context of generalized annotated logic programming. This concept was adapted to the more recent and flexible framework of multi-adjoint logic programming for solving a problem of incompleteness that arises when working with some lattices. In order to be complete, multi-adjoint logic programs must be extended with their set of reductants. In general, the notion of reductant may introduce an important efficiency drawback. In this work we provide a more refined version of this concept that we call *PE-reductant*, by using (threshold) partial evaluation techniques. Our proposal is intended to be semantically equivalent to the classical notion of reductant, and improves previous approaches at least in the following two efficiency criteria. Firstly, using the new definition of reductant, we can obtain computed answers for a given goal with a lesser computational effort than by using its precedent ones. Secondly, the proper construction of a reductant by means of partial evaluation methods, is drastically improved after introducing thresholding techniques which dynamically reduce the size of the underlying unfolding trees.

Key words: Fuzzy Logic Prog., Partial Evaluation, Reductants.

1 Introduction

Multi-adjoint logic programming [11,12,13] 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, that is, a goal plus a substitution (initially the identity substitution, denoted by id). Given a multi-adjoint logic program, goals are evaluated in two separate computational phases. During the

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

[2] Email: `{Pascual.Julian},{Gines.Moreno},{Jaime.Penabad}@uclm.es`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

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 [6], 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.

Reductancts were introduced in the context of multi-adjoint logic programming to cope with a problem of incompleteness that arises for some lattices. It might be impossible to compute the greatest correct answer, if a lattice $(L, \preceq)$ is partially ordered [13]. 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 [13], both a and b are correct answers and hence, by definition of correct answer [13], the supremum (or lub, least upper bound) $sup\{a, b\}$, is also a correct answer; however, neither $sup\{a, b\}$ nor a more general version of $sup\{a, b\}$ are computed answers and, therefore, completeness is lost. The above problem can be solved by extending the original program with a special rule $\langle A \leftarrow sup\{a, b\}; \top\rangle$, the so called *reductant*, which allows us to obtain the supremum of all the contributions to the goal 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 computational effort done (once) at generation time is avoided (many times) at execution time. Moreover, and what is best, if the proper partial evaluation process is combined with thresholding techniques, we also achieve three extra benefits:

- The proper construction of the underlying unfolding tree consumes less computational resources (both memory and CPU) by efficiently pruning some unnecessary branches of the tree and hence, drastically reducing its size.
- As a direct consequence of the previous fact, the shape of the resulting reductant is largely simplified.
- Finally, those derivation sequences performed at execution time, needs less computation steps when using this refined notion of PE-reductant.

Partial evaluation (PE) [4] 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 [1], specially the so called *unfolding* transformation. Unfolding is essentially the replacement of a call by its definition, with appropriate substitutions.

As we want to support the computation of reductants by means of PE techniques, in [7] we have introduced a preliminary definition of the concept of PE for multi-adjoint logic programs and goals. The idea is to adapt, for this new framework, the techniques arisen around the field of partial deduction of pure logic programs [3,8,10]. Following this path, we try to unfold admissible goals, as much as possible, using the notion of unfolding rule developed in [5,6] for multi-adjoint logic programs, in order to obtain an optimized (specialized) version of the original program.

The structure of the paper is as follows. In Section 2 we give some preliminary notions used along the whole work: subsections 2.1 and 2.2 summarize the main features of multi-adjoint logic programming, both language syntax and procedural semantics, whereas subsection 2.3 introduces some basic concepts that extend, for the multi-adjoint logic programming framework, the notion of partial evaluation of an atom in a program. Section 3 presents a formal definition of *PE-reductant* and relates it with the classical concept of reductant and also with the notion of partial evaluation. Inspired by our experience in the development of partial evaluation techniques, we give a more refined version of the concept of reductant considered in [13], which we call PE-reductant. In Section 4, we provide a concrete algorithm for the construction of PE-reductants which is based on unfolding with a set of dynamic thresholds: subsection 4.1 firstly introduces some preparatory results in order to formally proceed in subsection 4.2 with the improved algorithm, whereas in subsection 4.3 we discuss the benefits of the resulting technique by means of some comparative examples. Finally, in Section 5 we give our conclusions and some lines of future work.

2 Preliminaries

This section gives a short summary of the main features of Multi-adjoint logic programming (we refer the interested reader to [11,12,13] for a complete formulation) and formalizes the basic notions involved in the partial evaluation of multi-adjoint logic programs as introduced in [7].

2.1 The multi-adjoint 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 $\&_i$, $\vee_i$ and $@_i$ are binary operators, we usually generalize them as functions with an arbitrary number of arguments. In the following, we often write $@(x_1, \ldots, x_n)$ instead of $@(x_1, @(x_2, \ldots, @(x_{n-1}, x_n)\ldots))$. Aggregation operators are useful to describe/specify user preferences. An aggregation operator, when interpreted as a truth function, may be an arithmetic mean, a 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 monotonous and fulfills $\dot{@}(\top, \ldots, \top) = \top$, $\dot{@}(\bot, \ldots, \bot) = \bot$.

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 $\langle \leftarrow_i, \&_i\rangle$, where each $\&_i$ is a conjunctor [7] intended to the evaluation of *modus ponens*. In general, the set of truth values L may be the carrier of any complete bounded lattice but, for readability reasons, in the examples we shall select L as the set of real numbers in the interval $[0, 1]$ (which is a totally ordered lattice or chain).

A *rule* is a formula $H \leftarrow_i \mathcal{B}$, where H is an atomic formula (usually called the *head*) and $\mathcal{B}$ (which is called the *body*) is a formula built from atomic formulas $B_1, \ldots, B_n$ — $n \geq 0$ —, truth values of L, conjunctions, disjunctions and aggregations. 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".

[7] For a formal definition of a multi-adjoint lattice and the semantic properties of the connectives in $\mathcal{L}$, see [13]. It is noteworthy that a symbol $\&_j$ of $\mathcal{L}$ does not always need to be part of an adjoint pair.

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 [6], 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 occur in the —possibly empty— context $\mathcal{C}[]$. Moreover, 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 2.1 (Admissible Steps) *Let $\mathcal{Q}$ be a goal and let σ be a substitution. The pair $\langle \mathcal{Q}; \sigma \rangle$ is an* state *and we denote by $\mathcal{E}$ the set of states. Given a program $\mathcal{P}$, an* admissible computation *is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathcal{E} \times \mathcal{E})$ is the smallest relation satisfying the following* admissible rules [2] *(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 second rule is introduced to cope with (possible) unsuccessful admissible derivations. When needed, we shall use the symbols $\rightarrow_{AS1}$ 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 the $\rightarrow_{AS}$ symbol. Also the symbols $\rightarrow^+_{AS}$ and $\rightarrow^*_{AS}$ denote, respectively, the transitive closure and the reflexive, transitive closure of $\rightarrow_{AS}$.

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

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.

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

[2] Note that case one subsumes the second case in the original definition presented in [13], 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 $\dot{\&}_i(v, \top) = v$.

Definition 2.3 (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 defined as the smallest one satisfying: $\langle Q[@(r_1, r_2)]; \sigma\rangle \rightarrow_{IS} \langle Q[@(r_1,r_2)/\dot{@}(r_1,r_2)];\sigma\rangle$, where $\dot{@}$ is the truth function of connective $@$ in the lattice $\langle L, \preceq\rangle$ associated to $\mathcal{P}$.*

We denote by $\rightarrow^{+}_{IS}$ and $\rightarrow^{*}_{IS}$ the transitive closure and the reflexive, transitive closure of $\rightarrow_{IS}$, respectively.

Definition 2.4 *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 $Q' = r \in L$, being $\langle L, \preceq\rangle$ the lattice associated to $\mathcal{P}$, the state $\langle r; \sigma\rangle$ is called a* fuzzy computed answer *(f.c.a.) for that derivation.*

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.

2.3 Partial Evaluation of Multi-Adjoint Logic Programs

In [7] we formalize the basic notions involved in the partial evaluation of multi-adjoint logic programs. Observe that, in contrast with the operational semantics defined in Section 2.2, the 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 nondeterminism of a computation [2]. In the sequel we call *normalization* the sequence of interpretive steps performed before an operational unfolding step.

The partial evaluation of an atomic goal is defined by constructing incomplete search trees for the goal and extracting the specialized definition —usually called *resultants*, as defined in [7]— from the root-to-leaf branches. Hence, before defining this concept, we precise the notion of unfolding tree.

Definition 2.5 (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 φ) is a set of $\langle goal; substitution\rangle$ pair nodes satisfying the following conditions:*

(i) *The root node of τ_φ is $\langle \mathcal{Q}\ ; id\rangle$, where id is the identity substitution.*

(ii) *If $\mathcal{N}_i \equiv \langle \mathcal{Q}[A]; \sigma\rangle$ is a node of τ_φ and assuming $\varphi(\mathcal{Q}) = A$ 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.*

(iii) *If $\mathcal{N}_i \equiv \langle \mathcal{Q}[@(r, r')]; \sigma\rangle$ is a node of τ_φ then, $\mathcal{N}_{ij} \equiv \langle \mathcal{Q}[@(r, r')/\dot{@}(r, r')]); \sigma\rangle$ is a node of τ_φ.*

[3] In a *normalizing narrowing* strategy a term is rewritten to its normal form before a narrowing step is applied.

As defined in [5,6], the second and third cases respectively relate to the application of an operational unfolding step and an interpretive unfolding step.

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. That is, we are allowed to terminate a derivation at any adequate point.

Definition 2.6 (Partial evaluation of an atom) *Let $\mathcal{P}$ be a program, A be an atomic goal, and τ be a finite (possibly incomplete) 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 $\mathcal{P}' = \{\langle A\sigma_i \leftarrow \mathcal{Q}_i\ ; \top\rangle \mid i = 1, \ldots, k\}$ the set of rules (the so called resultants) associated with the derivations $\{\langle A\ ; id\rangle \to^{+} \langle \mathcal{Q}_i\ ; \sigma_i\rangle \mid i = 1, \ldots, k\}$. Then, the set $\mathcal{P}'$ is called a* partial evaluation *of A in $\mathcal{P}$ (using τ).*

3 Reductants versus PE-Reductants

In this section we define a new concept of reductant based on techniques coming from the field of partial evaluation. The starting point is the original definition presented in [13], where the classical notion of reductant was initially adapted to the multi-adjoint logic programming framework in the following terms:

Definition 3.1 (Reductant [13]) *Let $\mathcal{P}$ be a program, A a ground atom, and $\langle C_i {\leftarrow_i}\ \mathcal{B}_i; v_i\rangle$ be the (non empty) set of rules in $\mathcal{P}$ whose head matches with A (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 $\theta = \theta_1 \ldots \theta_n$, $\leftarrow$ 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 3.1 can be improved, leading to a more flexible approximation of 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 [7]), 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 3.2 (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, in the particular case that the tree used in Definition 3.2 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 –non empty– set of rules in $\mathcal{P}$ whose heads match with A) then, the resulting PE-reductant is the rule $\langle A \leftarrow sup\{(v_1 \&_1 \mathcal{B}_1)\theta_1, \ldots, (v_n \&_n \mathcal{B}_n)\theta_n\}; \top\rangle$, which is very similar to the Definition 3.1. It is easy to prove that this particular case of PE-reductant which uses a one-step unfolding tree, conforms with the original definition of reductant appeared in [13].

Example 3.3 *Given the lattice* $([0,1], \preceq)$, *where "*$\preceq$*" is the usual order on real numbers, let* $\mathcal{P}$ *be the following multi-adjoint logic program:*

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

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

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

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, Definition 3.1 builds the reductant:

$\langle p(a) \leftarrow @(q(X_1,a),\ s(Y_2), 0.6, q(b,a) \&_{\mathtt{L}} t(a));\ 1\rangle$ *where* $\dot{@}(b_1,b_2,b_3,b_4) = sup\{0.7 \dot{\&}_{\mathtt{L}} b_1, 0.5 \dot{\&}_{\mathtt{G}} b_2, b_3, 0.8 \dot{\&}_{\mathtt{G}} b_4\}$.

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

Definition 3.4 (Construction of PE-reductants) *Given a program* $\mathcal{P}$ *and a ground atomic goal* A. *We can enumerate the following steps in the construction of a PE-reductant of* A *in* $\mathcal{P}$:

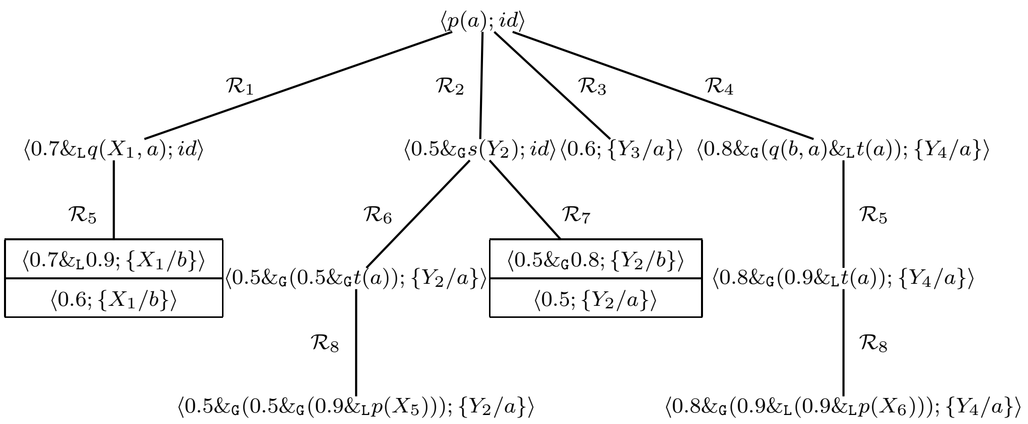
(i) *Construct an unfolding tree,* τ, *for* $\mathcal{P}$ *and* A, *that is, the tree obtained by unfolding the atom* A *in the program.*

(ii) *Collect the set of leaves* $S = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ *in* τ.

(iii) *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 3.5 *Let* $\mathcal{P}$ *be the program of Example 3.3 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: $\langle p(a) \leftarrow @sup\{0.6, 0.5\&_{\mathtt{G}}(0.5\&_{\mathtt{G}}(0.9\&_{\mathtt{L}}p(X_5))), 0.5, 0.6, 0.8\&_{\mathtt{G}}(0.9\&_{\mathtt{L}} (0.9\&_{\mathtt{L}}p(X_6)))\}; 1\rangle$.

Because 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 is able to compute the greatest correct answer for that goal. Moreover, although for the same program $\mathcal{P}$ and ground atom A, it is possible to derive distinct reductants, depending on the precision of the underlying unfolding tree, we claim that all of them are able to compute the same greatest correct answer for the goal A.

4 Threshold Construction of PE-Reductants

In this section we provide an efficient algorithm for the construction of a PE-reductant based on unfolding with a set of dynamic thresholds.

4.1 Upper bound of a computation and thresholds

In the context of a fuzzy computation it makes sense to disregard a derivation if the truth degree of a (partial) fuzzy computer answer falls down below of a certain threshold value $\mathcal{V}$. In our framework, this situation could be detected in "advance", that is, before the fuzzy computation has been completed. The next result provides the theoretical basis which allows us to support this "look-ahead".

Proposition 4.1 *Let* $\langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n \rangle$ *be a multi-adjoint lattice. Then, for any* $x, y \in L$, *1)* $x\&_i y \ \preceq\ x$ *and 2)* $x\&_i y \ \preceq\ y$.

Proof. Item (1) is an easy consequence of the definition of multi-adjoint lattice [13]. Firstly, $x\&_i y \preceq x\&_i \top$ because the adjoint operator $\&_i$ is, by definition, increasing in both arguments –that is, if $x_1, x_2, x_3 \in L$ and $x_1 \preceq x_2$ then $x_1\&_i x_3 \preceq x_2\&_i x_3$ and $x_3\&_i x_1 \preceq x_3\&_i x_2$– and L has a top element ($\top$) –that is, $y \preceq \top$ for all $y \in L$–. Secondly, the adjoint operator $\&_i$ also fulfill, by definition of multi-adjoint lattice, that $x \&_i \top = x$ for all $x \in L$, which concludes the proof. The proof of item (2) is completely analogous. □

The following result is a corolary of Proposition 4.1 showing that $inf\{x, y\}$ is an upper bound of $x \& y$.

Proposition 4.2 *Let $\langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n \rangle$ be a multi-adjoint lattice. Then, for any $x, y \in L$ and adjoint conjunction $\&_i$: $x \&_i y \preceq inf\{x, y\}$, where inf is the lowest of x and y.*

As a consequence of Proposition 4.1, it is noteworthy that, in an admissible step $\langle \mathcal{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathcal{Q}[A/v \&_i \mathcal{B}])\theta; \sigma\theta \rangle$, the component $v \&_i \mathcal{B}$, introduced by the rule, is lesser than v. This is independent of the truth degree eventually computed for the subgoal $\mathcal{B}$. Therefore, if the goal $\mathcal{Q}$ is compounded by conjunctors fulfilling the conditions of Proposition 4.1 (note that this restriction is vacuously true for an atomic goal), v is an upper bound of the truth degree computable for $\mathcal{Q}$.

The above discussion leads, in a natural way, to the notion of threshold unfolding, where only unfolding steps leading to nodes with a foreseeable truth degree value greater than a threshold value $\mathcal{V}$ are allowed. In other words, when the upper bound truth degree of a node falls down the threshold value $\mathcal{V}$, the unfolding of the node is stopped. Next section precises this concept.

4.2 A concrete algorithm

In this section we assume that Proposition 4.1 holds for all connectives and aggregators. This does not imply a serious loss of generality in practice.

During the construction of a PE-reductant many unfolding steps are useless, since they generate leaf nodes that cannot lead to the final computation of the supremum. For instance, in Example 3.5, node $\langle 0.5; \{Y_2/a\} \rangle$ does not contribute, since $0.5 \preceq 0.6$ —the truth degree of a completely evaluated leaf node— nor the node $\langle 0.5 \&_{\mathtt{G}} (0.5 \&_{\mathtt{G}} (0.9 \&_{\mathtt{L}} p(X_5))); \{Y_2/a\} \rangle$, since, by Proposition 4.1, even when the subsequent complete evaluation of the subgoal $p(X_5)$ would reach the top value, we have $0.5 \&_{\mathtt{G}} (0.5 \&_{\mathtt{G}} (0.9 \&_{\mathtt{L}} \top)) \preceq 0.5 \preceq 0.6$. So, the PE-reductant for $p(a)$ in the program of Example 3.5 can be written in a more accurate/simplifiedform as: $\langle p(a) \leftarrow @_{sup}\{0.6, 0.8 \&_{\mathtt{G}} (0.9 \&_{\mathtt{L}} (0.9 \&_{\mathtt{L}} p(X_6)))\}; 1 \rangle$.

We can optimize the construction of PE-reductants if we use an adaptation of the notion of unfolding tree (Definition 2.5) where: i) nodes contain information about an upper bound of the truth degree associated to the goal component; and ii) a set of threshold values is set dynamically to limit the generation of useless nodes. This last feature provides great opportunities to reduce the unfolding tree shape, by stopping unfolding of those nodes whose truth degree upper bound component falls down a threshold value $\mathcal{V}$.

We propose a construction procedure in two phases. In the first phase we build (traverse) an incomplete threshold unfolding tree, for a program $\mathcal{P}$ and a goal A, trying to limit the generation of useless nodes. During the construction of the tree we store the leaf nodes in a list. In the second phase, in order to construct the PE-reductant, we traverse the former list and remove the leaf nodes that cannot contribute to the computation of the supremum.

As for a classical proof procedure, three points are important: the computation rule (that is, the selection function used to decide which atom must be exploited in the next computation step[4]); the order rule (i.e., the order in which the rules of the program are tried for unfolding) and the search strategy (either a breadth-first or a depth-first). The algorithm we present is parametric with regard all these points, as well as a stop criterion to ensure termination ofunfolding[5].

Algorithm 1 (Unfolding with a set of dynamic thresholds)

»»»» **[INPUT]:** `A program` $\mathcal{P}$ `and a ground atom` A.

(i) `Set` $LEAVES = []$ `(the empty list), and` $THRESHOLDS = [\bot]$;

(ii) `Build the root node` $\langle A; id; \top\rangle$ `and set` $OPEN = [\langle A; id; \top\rangle]$;

(iii) `While` $OPEN \neq []$ `do:`

(a) `Take a node, say` $\mathcal{N}_i$, `of the list` $OPEN$ `(following the search strategy);`

(b) `If` $\mathcal{N}_i$ `holds the stop criterion then add the node` $\mathcal{N}_i$ `to the list` $LEAVES$;

(c) `Else, assume that` $\mathcal{N}_i \equiv \langle \mathcal{Q}[E]; \sigma; u\rangle$, `where` E `is the selected atom in` $\mathcal{Q}$ `(following the computation rule);`
`For each rule` $\mathcal{R}_j \equiv \langle H \leftarrow \mathcal{B}; v\rangle \in \mathcal{P}$ `(following the order rule), with` $\theta = mgu(\{E = H\})$ `and THERE IS NOT any` $\mathcal{V} \in THRESHOLDS$ `such that` $v < \mathcal{V}$ `do:`

- `Generate the child node` $\mathcal{N}_{ij} \equiv \langle (\mathcal{Q}[E/v\&\mathcal{B}])\theta; \sigma\theta; inf\{u, v\}\rangle$;
- `Normalize the first component of the new node` $\mathcal{N}_{ij}$. `That is, apply a (maximal) sequence of interpretive steps:` $\langle ((\mathcal{Q}[E/v\&\mathcal{B}])\theta; \sigma\theta\rangle \rightarrow_{IS}{}^{*} \langle (\mathcal{Q}'; \sigma\theta\rangle$. `Thus, we obtain a new node` $\mathcal{N}'_{ij} \equiv \langle \mathcal{Q}'; \sigma\theta; inf\{u, v\}\rangle$.
- `If` $\mathcal{Q}' = r \in L$, `then`
  - `If THERE IS NOT any` $\mathcal{V} \in THRESHOLDS$ `s.t.` $r < \mathcal{V}$:
    `Let` $\mathcal{W} \subseteq THRESHOLDS$ `be the (possibly empty) greatest subset of values comparable with` r `such that` $r > \mathcal{V}$ `for each` $\mathcal{V} \in \mathcal{W}$;
    `Replace the set` $\mathcal{W}$ `by` $\{r\}$ `in` $THRESHOLDS$.
- `Else (`$\mathcal{Q}' \neq r \in L$`; i.e., the node is not completely evaluated), add the node` $\mathcal{N}'_{ij}$ `to the list` $OPEN$;

(iv) `Remove nodes` $\langle @(r_1, \dots, r_n, \mathcal{B}_1, \dots, \mathcal{B}_m); \phi; w\rangle$ `in` $LEAVES$ `verifying that, there exists` $\mathcal{V} \in THRESHOLDS$, `such that` $w < \mathcal{V}$ `or` $\dot{@}(r_1, \dots, r_n, \top, \dots, \top) < \mathcal{V}$.

»»»» **[OUTPUT]:** `Lists` $THRESHOLDS$ `and` $LEAVES$.

As we have seen, the algorithm works with four lists:

---

[4] We have recently proved in [5] an independence result for this choice, as it is also usual in other non-fuzzy logic paradigms. Similarly to PROLOG, in our examples we always exploit the left-most atom of a given goal.

[5] The *local termination problem* can be solved in an albeit *ad hoc* way, by imposing an arbitrary depth bound for the unfolding, or using more refined approaches like methods based on well-founded orders or well-quasi orders.

- *OPEN*, which contains the nodes to be unfolded;
- *LEAVES*, which contains the nodes which hold some termination criterion;
- *THRESHOLDS*, which stores a set of nodes completely evaluated (not comparable among them) which are used as thresholds.

Roughly speaking, we only permit to unfold a node (by means of an admissible step) using rules with a truth degree v, such that, v is comparable with none $\mathcal{V} \in THRESHOLDS$, or $v > \mathcal{V}$ for some $\mathcal{V} \in THRESHOLDS$. Otherwise, because a direct consequence of Proposition 4.1, we would reach a node (goal) whose later evaluation never would produce a truth degree greater or equal to $\mathcal{V}$. The inclusion of a normalization step (that is, a sequence of interpretive unfolding steps) after each operational unfolding step increases the possibility of obtaining completely evaluated nodes and therefore the possibility of refining the set of threshold values. Thus, more useless nodes can be disregarded.

Observe that the list LEAVES can be accessed either as a LIFO (stack) or a FIFO (queue) structure, which respectively corresponds with a depth-first or breadth-first generation/ traversal of the underlying tree. The experience shows us that there are not advantages (with regard the elimination of useless nodes) when choosing either a breadth-first or a depth first strategy. We have examples where the breadth-first strategy has a better performance in comparison with the depth first strategy and vice-versa. Also there is not any evidence indicating if a concrete computation rule can improve the elimination of useless nodes. However, the order rule has a mayor impact in the removal of useless nodes. We saw that an order rule which reorders rules on the basis of the number of atoms in their bodies, giving preference to the facts over the other rules, has (possibly) the best behavior.

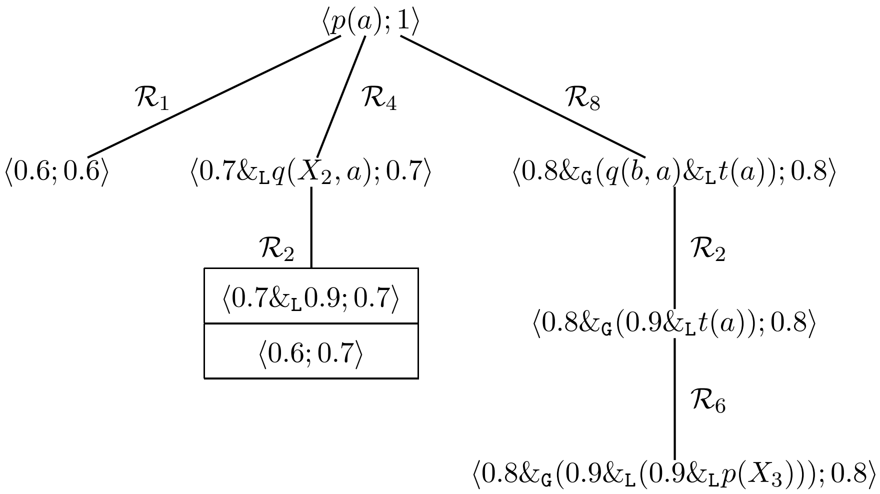
Finally, if $THRESHOLDS = \{r_1, \ldots, r_m\}$ and $LEAVES = \{\langle \mathcal{Q}_1; \phi_1; w_1\rangle, \ldots, \langle \mathcal{Q}_n; \phi_n; w_n\rangle\}$ are the lists of thresholds and leaves returned by Algorithm 1, the PE-reductant of A in $\mathcal{P}$ is: $\langle A \leftarrow @_{sup}\{r_1, \ldots, r_m, \mathcal{Q}_1, \ldots, \mathcal{Q}_n\}; \top\rangle$.

Example 4.3 *Let $\mathcal{P}$ be the program and the goal $p(a)$ of Example 3.3. Assume an order rule such that rules in $\mathcal{P}$ are tried in the following order for unfolding:*

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

and a stop criterion that only permits depth-3 unfolding. After we set $\mathcal{V} = 0$ and construct the root node $\langle p(a); 1\rangle$, applying the sequence of steps in Algorithm 1, we obtain the following depth-3 threshold unfolding tree[3] *for the program $\mathcal{P}$ and the ground atom $p(a)$ (which, for this example, is independent of the search strategy used in its construction):*

[3] For the sake of simplicity, we omit the substitution component of the nodes in the representation of the threshold unfolding tree.



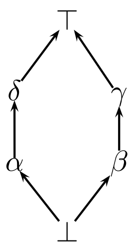
Observe that, at the very beginning, the unfolding step performed with rule $\mathcal{R}_1$ leads to the complete evaluated leaf node $\langle 0.6; 0.6\rangle$. Therefore the threshold $\mathcal{V}$ is set to 0.6 and the unfolding step with the rule $\mathcal{R}_5$ is avoided. At level 2, the normalized leaf node $\langle 0.6; 0.6\rangle$ does not alter the threshold $\mathcal{V}$ and since the computed truth degree 0.6 is not greater than $\mathcal{V}$, this node is not added to LEAVES. Hence, we obtain an unfolding tree smaller than the one obtained in Example 3.5. Finally, the Algorithm 1 returns the set of LEAVES $\{\langle 0.6; 0.6\rangle, \langle 0.8\&_{\mathtt{G}}(0.9\&_{\mathtt{L}}(0.9\&_{\mathtt{L}} p(X_3))); 0.8\rangle\}$, which allows us to generate a simpler PE-reductant: $\langle p(a) \leftarrow @_{sup}\{0.6, 0.8\&_{\mathtt{G}}(0.9\&_{\mathtt{L}}(0.9\&_{\mathtt{L}} p(X_3)))\};\ 1\rangle$.

4.3 A comparative example

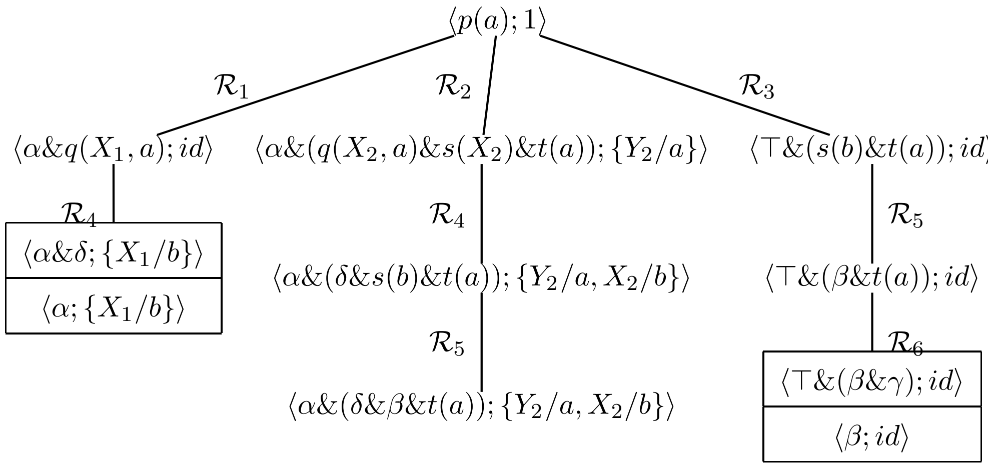
Our last example illustrates the benefits achieved by our threshold-based technique for computing PE-reductants when it is compared with Definitions 3.1 and 3.2. Firstly, we are interested in evidencing that the original program is not able to compute a concrete correct answer. Secondly, we focus our attention in the comparison of the computational effort needed to compute and execute different forms of PE-reductants, as well as their own shapes, which highlights the main advantages of our algorithm.

Let $\mathcal{P}$ be the following program, where connective $\&$ used in all rules has a truth function defined by $\dot{\&}(x, y) = inf\{x, y\}$, and the underlying lattice $(L, \preceq)$ is represented by the corresponding diagram.

$\mathcal{R}_1 : \langle p(a) \leftarrow q(X, a);$	$\alpha\rangle$	$\mathcal{R}_4 : \langle q(b, a) \leftarrow\ ;$	$\delta\rangle$
$\mathcal{R}_2 : \langle p(Y) \leftarrow q(X; Y)\&s(X)\&t(Y);$	$\alpha\rangle$	$\mathcal{R}_5 : \langle s(b) \leftarrow\ ;$	$\beta\rangle$
$\mathcal{R}_3 : \langle p(a) \leftarrow s(b)\&\ t(a);$	$\top\rangle$	$\mathcal{R}_6 : \langle t(a) \leftarrow\ ;$	$\gamma\rangle$



An unfolding tree of depth 3 for the program $\mathcal{P}$ and the ground atom $p(a)$ is:



From this figure we can construct the following PE-reductants exploiting different unfolding trees of depth-1, depth-3, or depth-3 with thresholding (which avoids the generation of the central branch shown in the figure), respectively:

$\mathcal{R}:\ \langle p(a) \leftarrow @_{sup}(\alpha\&(q(X_1,a), \alpha\&q(X_2,a)\&s(X_2)\&t(a)), \top\&(s(b)\&t(a))); \top\rangle$

$\mathcal{R}':\ \langle p(a) \leftarrow @_{sup}(\alpha, \alpha\&(\delta\&(\beta\&t(a))), \beta); \top\rangle$

$\mathcal{R}'':\ \langle p(a) \leftarrow @_{sup}(\alpha, \beta); \top\rangle$

Then, for the considered goal $p(a)$, the following facts hold:

(i) We know that, by the soundness property of multi-adjoint logic programs, since both $\langle \alpha; id\rangle$ and $\langle \beta; id\rangle$ are fuzzy computed answers for $\mathcal{P}$ and $p(a)$, they are correct answers too. Moreover, $\langle sup\{\alpha,\beta\}; id\rangle = \langle \top; id\rangle$ is also a correct answer. However, $\langle \top; id\rangle$ can not be computed in $\mathcal{P}$.

(ii) Fortunately, the PE-reductant $\mathcal{R}$ allows us to obtain the fuzzy computed answer $\langle \top; id\rangle$ after applying 10 computation steps as follows: $\langle p(a); id\rangle \rightarrow^{R}_{AS} \langle @_{sup}(\alpha\&q(X_1,a), \alpha\&(q(X_2,a)\&s(X_2)\&t(a)), \top\&(s(b)\&t(a))); id\rangle \rightarrow^{*(9)}_{AS/IS} \langle \top; id\rangle$. On the other hand, almost half the computational effort is needed when using the simpler PE-reductant $\mathcal{R}'$.

(iii) However, not only $\mathcal{R}''$ has the best shape, but also it proceeds with the best computational behaviour, by simply requiring the following pair of computation steps: $\langle p(a); id\rangle \rightarrow^{R''}_{AS} \langle @_{sup}(\alpha,\beta); id\rangle \rightarrow_{IS} \langle \top; id\rangle$.

5 Conclusions and Further Research

Reductants are crucial to cope with completeness in multi-adjoint logic programming. In this paper we have defined a method for computing the so called *PE-reductants* by using partial evaluation techniques based on unfolding with a set of dynamic thresholds. Moreover, we have discussed the benefits of our technique by means of several comparative examples, referring to the gains in efficiency achieved not only when constructing the proper PE-reductant, but also when using it at execution time. Nowadays we are working in the formulation of the set of properties fulfilled by our improved definition of reductant.

References

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

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

[3] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. ACM, New York, 1993.

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

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

[6] P. Julián, G. Moreno, and J. Penabad. Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. In F. López-Fraguas, editor, *Proc. of V Jornadas sobre Programación y Lenguajes, PROLE'2005, Granada, Spain, September 14-16*, pages 239–248. University of Granada, 2005.

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

[8] J. Komorowski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta-Programming in Logic, Uppsala, Sweden*, pages 49–69. Springer LNCS 649, 1992.

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

[10] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

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

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

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