Extended homogenization for multi-adjoint logic programs

Jesús Medina Moreno
Dept. Matemática Aplicada.
Universidad de Málaga
jmedina@ctima.uma.es

Manuel Ojeda Aciego
Dept. Matemática Aplicada.
Universidad de Málaga
aciego@ctima.uma.es

Resumen

The theory of multi-adjoint logic programs has been introduced as a unifying framework to deal with uncertainty, imprecise data or incomplete information. From the applicative part, a neural net based implementation of homogeneous propositional multi-adjoint logic programming on the unit interval has been presented elsewhere, but restricted to the case in which the only connectives involved in the program were the usual product, Gödel and Łukasiewicz together with weighted sums.

A generalization of the homogenization process needed for the neural implementation is presented here in order to deal with a more general family of adjoint pairs, but maintaining the advantage of the existing implementation. Its soundness is proved and the complexity of the transformation is analysed.

Keywords: Uncertainty and Approximate Reasoning.

1 Introduction

The study of reasoning methods under uncertainty, imprecise data or incomplete information has received increasing attention in the recent years. A number of different approaches have been proposed with the aim of better explaining observed facts, specifying statements, reasoning and/or executing programs under some type of uncertainty whatever it might be.

One important and powerful mathematical tool that has been used for this purpose at theoretical level is fuzzy logic. From the applicative side, neural networks have a massively parallel architecture-based dynamics which are inspired by the structure of human brain, adaptation capabilities, and fault tolerance. The recent paradigm of soft computing promotes the use and integration of different approaches for the problem solving.

The main advantages of fuzzy logic systems are the capability to express nonlinear input/output relationships by a set of qualitative if-then rules, and to handle both numerical data and linguistic knowledge, especially the latter, which is extremely difficult to quantify by means of traditional mathematics. The main advantage of neural networks, on the other hand, is the inherent learning capability, which enables the networks to adaptively improve their performance.

Recently, a new approach presented in [2] introduced a hybrid framework to handling uncertainty, expressed in the language of multi-adjoint logic but implemented by using ideas from the world of neural networks. The handling of uncertainty inside their logic model is based on the use of a generalised set of truth-values as a generalization of [6]. On the other hand, multi-adjoint logic programming [4] generalizes residuated logic programming [1] in that several different implications are allowed in the same program, as a means to facilitate the task of the specification.

Considering several implications in the same program is interesting because it provides a more flexible framework for the specification of problems, for instance, in situations in which connectives are built from the users preferences. In these contexts, it is likely that knowledge is described by a many-valued logic program where connectives have many-valued truth functions and, perhaps, aggregation operators (such as arithmetic mean or weighted sum) where different implications could be needed for different purposes, and different aggregators are defined for different users, depending on their preferences.

In [2] a neural-like implementation of multi-adjoint logic programming was presented with the restriction

that the only connectives involved in the program were the usual product, Gödel and Łukasiewicz together with weighted sums. A key point of the implementation was a preprocessing of the initial program to transform it into a *homogeneous* program, detailed in [3]. As the theoretical development of the multi-adjoint framework does not rely on particular properties of the product, Gödel and Łukasiewicz adjoint pairs, it seems convenient to allow for a generalization of the implementation to admit, at least, a family of continuous t-norms (recall that any continuous t-norm can be interpreted as the ordinal sum of product and Łukasiewicz t-norms).

The purpose of this paper is to present a "finer" homogenization process for multi-adjoint logic programs such that conjunctors which are built as ordinal sums of product, Gödel and Łukasiewicz t-norms are decomposed into its components so that, as a result, the original neural approach is still applicable.

The structure of the paper is as follows: In Section 2, the syntax and semantics of multi-adjoint logic programs are introduced, together with the homogenization procedure; in Section 3, the translation from multi-adjoint programs into homogeneous programs is extended in order to cope with conjunctors defined as ordinal sums, the preservation of the semantics is proved under this extended transformation, and the complexity of the translation is studied. The paper finishes with some conclusions and pointers to future work.

2 Preliminary definitions

To make this paper as self-contained as possible, the necessary definitions about multi-adjoint structures are included in this section. For motivating comments, the interested reader is referred to [4].

Multi-adjoint logic programming is a general theory of logic programming which allows the simultaneous use of different implications in the rules and rather general connectives in the bodies.

The first interesting feature of multi-adjoint logic programs is that a number of different implications are allowed in the bodies of the rules. The basic definition is the generalization of residuated lattice given below:

Definition 1 *Let $\langle L, \preceq\rangle$ be a lattice. A* multi-adjoint lattice *$\mathcal{L}$ is a tuple $(L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$ satisfying the following items:*

1. *$\langle L, \preceq\rangle$ is bounded, i.e. it has bottom and top elements;*
2. *$\top \,\&_i\, \vartheta = \vartheta \,\&_i\, \top = \vartheta$ for all $\vartheta \in L$ for $i = 1, \ldots, n$;*
3. *$(\&_i, \leftarrow_i)$ is an* adjoint pair *in $\langle L, \preceq\rangle$ for $i = 1, \ldots, n$; i.e.*
   (a) *Operation $\&_i$ is increasing in both arguments,*
   (b) *Operation $\leftarrow_i$ is increasing in the first argument and decreasing in the second argument,*
   (c) *For any $x, y, z \in P$, we have that $x \preceq (y \leftarrow_i z)$ holds if and only if $(x \,\&_i\, z) \preceq y$ holds.*

In the rest of the paper we restrict to the unit interval, although the general framework of multi-adjoint logic programming is applicable to a general lattice.

2.1 Syntax of multi-adjoint logic programs

A *multi-adjoint program* is a set of weighted rules $\langle F, \vartheta\rangle$ satisfying the following conditions:

1. F is a formula of the form $A \leftarrow_i \mathcal{B}$ where A is a propositional symbol called the *head* of the rule, and $\mathcal{B}$ is a well-formed formula, which is called the *body*, built from propositional symbols $B_1, \ldots, B_n$ $(n \geq 0)$ by the use of monotone operators.
2. The *weight* ϑ is an element (a truth-value) of $[0, 1]$.

Facts are rules with body[1] 1 and a *query* (or *goal*) is a propositional symbol intended as a question $?A$ prompting the system.

2.2 Semantics of multi-adjoint logic programs

Once presented the syntax of multi-adjoint programs, the semantics is given below.

Definition 2 *An* interpretation *is a mapping I from the set of propositional symbols Π to the lattice $\langle [0, 1], \leq\rangle$.*

Note that each of these interpretations can be uniquely extended to the whole set of formulas, and this extension is denoted as $\hat{I}$. The set of all the interpretations is denoted $\mathcal{I}_{\mathfrak{L}}$.

The ordering $\leq$ of the truth-values L can be easily extended to $\mathcal{I}_{\mathfrak{L}}$, which also inherits the structure of complete lattice and is denoted $\sqsubseteq$. The minimum element of the lattice $\mathcal{I}_{\mathfrak{L}}$, which assigns 0 to any propositional symbol, will be denoted $\triangle$.

Definition 3

1. *An interpretation $I \in \mathcal{I}_{\mathfrak{L}}$* satisfies *$\langle A \leftarrow_i \mathcal{B}, \vartheta\rangle$ if and only if $\vartheta \leq \hat{I}\,(A \leftarrow_i \mathcal{B})$.*

[1] It is also customary to use write $\top$ instead of 1, and even not to write any body.

2. *An interpretation $I \in \mathcal{I}_{\mathfrak{L}}$ is a* model *of a multi-adjoint logic program $\mathbb{P}$ iff all weighted rules in $\mathbb{P}$ are satisfied by I.*

3. *An element $\lambda \in L$ is a* correct answer *for a program $\mathbb{P}$ and a query $?A$ if for any interpretation $I \in \mathcal{I}_{\mathfrak{L}}$ which is a model of $\mathbb{P}$ we have $\lambda \leq I(A)$.*

The operational approach to multi-adjoint logic programs used in this paper will be based on the fixpoint semantics provided by the immediate consequences operator, given in the classical case by van Emden and Kowalski [5], which can be generalised to the framework of multi-adjoint logic programs by means of the adjoint property, as shown below:

Definition 4 *Let $\mathbb{P}$ be a multi-adjoint program; the* immediate consequences operator*, $T_{\mathbb{P}} \colon \mathcal{I}_{\mathfrak{L}} \to \mathcal{I}_{\mathfrak{L}}$, maps interpretations to interpretations, and for $I \in \mathcal{I}_{\mathfrak{L}}$ and $A \in \Pi$ is given by*

$$T_{\mathbb{P}}(I)(A) = \sup\left\{\vartheta \,\&_i\, \hat{I}(\mathcal{B}) \mid \langle A \leftarrow_i \mathcal{B}, \vartheta\rangle \in \mathbb{P}\right\}$$

As usual, it is possible to characterise the semantics of a multi-adjoint logic program by the post-fixpoints of $T_{\mathbb{P}}$; that is, an interpretation I is a model of a multi-adjoint logic program $\mathbb{P}$ iff $T_{\mathbb{P}}(I) \sqsubseteq I$. The $T_{\mathbb{P}}$ operator is proved to be monotonic and continuous under very general hypotheses.

Once one knows that $T_{\mathbb{P}}$ can be continuous under very general hypotheses [4], then the least model can be reached in at most countably many iterations beginning with the least interpretation, that is, the least model is $T_{\mathbb{P}} \uparrow \omega(\triangle)$.

2.3 Obtaining a homogeneous program

Regarding the implementation as a neural network of [2], the introduction of the so-called *homogeneous rules*, provided a simpler and standard representation for any multi-adjoint program.

Definition 5 *A weighted formula is said to be* homogeneous *if it has one of the following forms:*

- $\langle A \leftarrow_i \&_i(B_1, \ldots, B_n), \vartheta\rangle$
- $\langle A \leftarrow_i @(B_1, \ldots, B_n), 1\rangle$
- $\langle A \leftarrow_i B_1, \vartheta\rangle$

where $A, B_1, \ldots, B_n$ are propositional symbols.

In the rest of this section we briefly recall the procedure for translating a multi-adjoint logic program into one containing only homogeneous rules. The procedure handles separately rules and facts, the latter are not related to the purpose of this paper, therefore we will only recall the procedure presented for homogenizing rules.

Two types of transformations are considered: The first one handles the main connective of the body of the rule, whereas the second one handles the subcomponents of the body.

$T1$. A weighted rule $\langle A \leftarrow_i \&_j(\mathcal{B}_1, \ldots, \mathcal{B}_n), \vartheta\rangle$ is substituted by the following pair of formulas:

$$\langle A \leftarrow_i A_1, \vartheta\rangle$$
$$\langle A_1 \leftarrow_j \&_j(\mathcal{B}_1, \ldots, \mathcal{B}_n), 1\rangle$$

where A_1 is a fresh propositional symbol, and $\langle \leftarrow_j, \&_j\rangle$ is an adjoint pair.

For the case $\langle A \leftarrow_i @(\mathcal{B}_1, \ldots, \mathcal{B}_n), \vartheta\rangle$ in which the main connective of the body of the rule happens to be an aggregator, the transformation is similar:

$$\langle A \leftarrow_i A_1, \vartheta\rangle$$
$$\langle A_1 \leftarrow @(\mathcal{B}_1, \ldots, \mathcal{B}_n), 1\rangle$$

where A_1 is a fresh propositional symbol, and $\leftarrow$ is a designated implication.

$T2$. A weighted rule $\langle A \leftarrow_i \Theta(\mathcal{B}_1, \ldots, \mathcal{B}_n), \vartheta\rangle$, where Θ is either $\&_i$ or an aggregator, and a component $\mathcal{B}_k$ is assumed to be either of the form $\&_j(\mathcal{C}_1, \ldots, \mathcal{C}_l)$ or $@(\mathcal{C}_1, \ldots, \mathcal{C}_l)$, is substituted by the following pair of formulas in either case:

$$\langle A \leftarrow_i \Theta(\mathcal{B}_1, \ldots, \mathcal{B}_{k-1}, A_1, \mathcal{B}_{k+1}, \ldots, \mathcal{B}_n), \vartheta\rangle$$
$$\langle A_1 \leftarrow_j \&_j(\mathcal{C}_1, \ldots, \mathcal{C}_l), \top\rangle$$

or

$$\langle A \leftarrow_i \Theta(\mathcal{B}_1, \ldots, \mathcal{B}_{k-1}, A_1, \mathcal{B}_{k+1}, \ldots, \mathcal{B}_n), \vartheta\rangle$$
$$\langle A_1 \leftarrow @(\mathcal{C}_1, \ldots, \mathcal{C}_l), \top\rangle$$

where A_1 is a fresh propositional symbol.

The procedure to transform the rules of a program so that all the resulting rules are homogeneous, is presented in Fig. 2. It is based in the two previous transformations, and in its description by abuse of notation the terms T1-rule (resp. T2-rule) are used to mean an adequate input rule for transformation T1 (resp. T2).

3 Considering compound conjunctors

As stated in the introduction, a neural net implementation of the immediate consequences operator of

Program Homogenization
  begin
    repeat
      for each T1-rule **do**
        Apply transformation T1
      end-for
      for each T2-rule **do**
        Apply transformation T2
      end-for
    until neither T1-rules nor T2-rules exist
  end

Figure 1: Pseudo-code for translating into a homogeneous program.

an homogeneous program was introduced in [2] for the case of the multi-adjoint lattice $([0,1], \leq, \&_P, \leftarrow_P, \&_G, \leftarrow_G, \&_L, \leftarrow_L)$. As the theoretical development of the multi-adjoint framework does not rely on particular properties of these three adjoint pairs, it seems convenient to allow for a generalization of the implementation to, at least, a family of continuous t-norms, for any continuous t-norm can be interpreted as the ordinal sum of product and Łukasiewicz t-norms.

In order to maintain the most of the proposed implementation, it makes sense to consider an extra process in the homogenization process in order to further translate a program with new types of t-norms into one on which the original neural approach is still applicable.

Recall the definition of ordinal sum of a family of t-norms:

Definition 6 *Let $(\&_i)_{i\in A}$ be a family of t-norms and a family of non-empty pairwise disjoint subintervals $[a_i, b_i]$ of $[0,1]$. The* ordinal sum *of the summands $(a_i, b_i, \&_i)$, $i \in A$ is the t-norm $\&$ defined as*

$$\&(x,y) = \begin{cases} a_i + (b_i - a_i)\,\&_i\left(\frac{x-a_i}{b_i-a_i}, \frac{y-a_i}{b_i-a_i}\right) & \text{if } x,y \in [a_i,b_i] \\ \min(x,y) & \text{otherwise} \end{cases}$$

In order to simplify the notation of ordinal sum, let us introduce suitable functions for change of scale, to be able to switch between the intervals $[0,1]$ and $[a_i, b_i]$. Given the unit interval $[0,1]$ and a subset $[a_i, b_i] \subseteq [0,1]$, the function

$$f_i \colon [a_i, b_i] \to [0,1], \text{ with } f_i(x) = \frac{x - a_i}{b_i - a_i}$$

is bijective, and its inverse is

$$f_i^{-1} \colon [0,1] \to [a_i, b_i], \text{ with } f_i^{-1}(x) = a_i + (b_i - a_i)x$$

Now, given a t-norm $\&_i$ consider the following (adapted) t-norm

$$x \,\&_i^* \, y = \begin{cases} f_i^{-1}(f_i(x)\,\&_i\, f_i(y)) & \text{if } x,y \in [a_i,b_i] \\ \min\{x,y\} & \text{otherwise} \end{cases}$$

for all $x, y \in [0,1]$.

With the notations above it is obvious that the ordinal sum of the summands $(a_i, b_i, \&_i)$, $i \in A$ can be written as

$$x \,\&\, y = \min\{x \,\&_i^*\, y \mid i \in A\} \tag{1}$$

With this expression for the ordinal sum, we can introduce a third type of transformation, to be applied to those rules of a homogeneous program with an ordinal sum in the body.

First of all, it is convenient to consider these sum-like conjunctors as general aggregator operators, so that after the (standard) homogenization process we can assume that the weight of the obtained rule is 1.

*T*3. The homogeneous rule[2] $\langle A \leftarrow_i B_1 \,\&\, B_2, 1\rangle$, where $\&$ is expressed as in (1) is substituted by the following $n+1$ formulas:

$$\begin{aligned} \langle A_1 &\leftarrow B_1 \,\&_1^*\, B_2, 1\rangle \\ &\vdots \\ \langle A_n &\leftarrow B_1 \,\&_n^*\, B_2, 1\rangle \\ \langle A &\leftarrow_G \&_G(A_1, \ldots, A_n), 1\rangle \end{aligned}$$

where $A_1, \ldots, A_n$ are fresh propositional symbols.

After applying this transformation to a homogeneous program, we obtain another homogeneous program in whose bodies no t-norm appears as an ordinal sum. This kind of homogenous program is called *sum-free homogeneous program* (in short *sf-homogeneous program*).

Example 1 *Consider the elements $a_1 = 0.1$, $b_1 = 0.5$, $a_2 = 0.7$, $b_2 = 0.9$ and the ordinal sum $\&_s$ defined as*

$$x \,\&_s\, y \begin{cases} f_1^{-1}(f_1(x)\,\&_P\, f_1(y)) & \text{if } x,y \in [a_1,b_1] \\ f_2^{-1}(f_2(x)\,\&_L\, f_2(y)) & \text{if } x,y \in [a_2,b_2] \\ \min\{x,y\} & \text{otherwise} \end{cases}$$

Then if we have the homogeneous rule $\langle A \leftarrow_s B_1 \,\&_s\, B_2, 1\rangle$, applying T3, this is transformed in

$$\begin{aligned} \langle A_1 &\leftarrow B_1 \,\&_P^*\, B_2, 1\rangle && \textit{sf-homogeneous} \\ \langle A_n &\leftarrow B_1 \,\&_L^*\, B_2, 1\rangle && \textit{sf-homogeneous} \\ \langle A &\leftarrow_G \&_G(A_1, \ldots, A_n), 1\rangle && \textit{sf-homogeneous} \end{aligned}$$

[2] To simplify the presentation, let us assume that the body has just two arguments.

The procedure to transform the rules of a program so that all the resulting rules are sf-homogeneous, is presented in Fig. 2. In its description by abuse of notation we use the terms T1-rule (resp. T2-rule, T3-rule) to mean an adequate input rule for transformation T1 (resp. T2, T3).

Program sf-Homogenization
  begin
    repeat
      for each T1-rule **do**
          Apply transformation T1
      end-for
      for each T2-rule **do**
          Apply transformation T2
      end-for
      for each T3-rule **do**
          Apply transformation T3
      end-for
    until neither T1- nor T2- nor T3-rules exist
    end

Figure 2: Pseudo-code for translating into a sf-homogeneous program.

3.1 Preservation of the semantics

It is necessary to check that the semantics of the initial program has not been changed by the transformation. The following results will show that every model of the sf-homogenized program $\mathbb{P}^*$ is also a model of the original program $\mathbb{P}$ and, in addition, the minimal model of $\mathbb{P}^*$ is also the minimal model of $\mathbb{P}$.

Theorem 1 *Let $\mathbb{P}$ be a homogeneous program, then every model of the program $\mathbb{P}^*$, obtained after to apply the sf-homogenization process to $\mathbb{P}$, is also a model of $\mathbb{P}$ when restricted to the variables occurring in $\mathbb{P}$.*

Proof: It will be sufficient to show that the transformation T3 satisfies that every model of its output is also a model of its input, as the proof for T1 and T2 was given in [3].

Assume that I is a model of the rules

$$\langle A_1 \leftarrow B_1 \,\&_1^*\, B_2, 1\rangle, \quad \ldots, \quad \langle A_n \leftarrow B_1 \,\&_n^*\, B_2, 1\rangle$$
$$\langle A \leftarrow_G \&_G(A_1, \ldots, A_n), 1\rangle$$

therefore we have

$$\hat{I}(B_1 \,\&_i^*\, B_2) \leq I(A_i) \quad \text{and} \quad \hat{I}(\&_G(A_1, \ldots, A_n)) \leq I(A)$$

for all $i \in \{1, \ldots, n\}$. Now, by monotonicity, we have

$$\&_G(\hat{I}(B_1 \,\&_1^*\, B_2), \ldots, \hat{I}(B_1 \,\&_n^*\, B_2)) \leq I(A)$$

Recall that we want to prove that I satisfies the rule $\langle A \leftarrow_i B_1 \,\&\, B_2, 1\rangle$, that is, $\hat{I}(B_1 \,\&\, B_2) \leq I(A)$, where $\&$ is an ordinal sum of $\&_i^*$, $i \in \{1, \ldots, n\}$.

But this is true by Equation (1) since we have

$$\begin{aligned} I(B_1) \,\&\, I(B_2) &= \min_{i\in\{1,\ldots,n\}}\{I(B_1) \,\&_i^*\, I(B_2)\} \\ &= \&_G(\hat{I}(B_1 \,\&_1^*\, B_2), \ldots, \hat{I}(B_1 \,\&_n^*\, B_2)) \\ &\leq I(A) \end{aligned}$$ □

Theorem 2 *Given a program $\mathbb{P}$, the minimal model of the program $\mathbb{P}^*$ obtained after applying sf-homogenization, is also a model of $\mathbb{P}$ when restricted to variables in $\mathbb{P}$.*

The idea underlying the proof is to consider any model I of $\mathbb{P}$, then extend it to $\mathbb{P}^*$ in such a way that it is also a model of $\mathbb{P}^*$, finally use minimality on $\mathbb{P}^*$. The key point is to notice that, for every "fresh" propositional variable A_i introduced by the process, there is only one rule headed with A_i in the resulting program. This feature allows the extension of any model I to these new symbols in purely recursive terms.

Proof:

Let M^* be the minimal model of $\mathbb{P}^*$, and let M denote its restriction to $\mathbb{P}$. By the Theorem 1 we have that M is also a model of $\mathbb{P}$, so we have only to prove that it is minimal.

Once again, only the behaviour of transformation T3 has to be taken into account.

Given a model I of $\mathbb{P}$, consider a rule $\langle A_i \leftarrow B_1 \,\&_i^*\, B_2, 1\rangle$, where A_i is a propositional variable in $\mathbb{P}^*$ but not in $\mathbb{P}$. We argued above that there can be only one such rule headed with A_i, therefore the following extension of I makes sense:

$$I^*(A_i) = \hat{I}(B_1 \,\&_i^*\, B_2)$$

Obviously, by definition this extension I^* is also a model of $\mathbb{P}^*$, therefore the minimal model M^* of $\mathbb{P}^*$ satisfies $M^* \sqsubseteq I^*$. Now, by restricting the domain back to the variables in $\mathbb{P}$ we obtain $M \sqsubseteq I$. Therefore, M is the minimal model of $\mathbb{P}$. □

3.2 Complexity issues

In [3] it was shown that the complexity of the algorithm for transforming a multi-adjoint program into a homogeneous one is linear on the size of the program. Specifically, the following theorem was stated and proved

Theorem 3 ([3]) *Let $\langle A \leftarrow_i \Theta(\mathcal{B}_1, \ldots, \mathcal{B}_l), \vartheta\rangle$ be a rule with n connectives in the body ($n \geq 1$), then:*

- *The number of homogeneous rules obtained after applying the procedure is n, if either $\Theta = \&_i$ or $\Theta = @$ with $\vartheta = 1$; and $n + 1$ otherwise.*
- *The number of transformations obtained after applying the procedure is $n - 1$ if either $\Theta = \&_i$ or $\Theta = @$ with $\vartheta = 1$; and n otherwise.*

It has to be pointed out that the number of rules obtained (or transformations applied) is *bounded* by the integer number stated in the theorem. The equality is obtained when all the associative operators are effectively grouped into one 'flexible arity' operator.

Back to the sf-homogenization process, it can also be shown to be linear on the size of the program. The following theorem shows a precise calculation of the complexity of the homogenization procedure.

Theorem 4 *Let $\langle A \leftarrow_i \Theta(\mathcal{B}_1, \ldots, \mathcal{B}_l), \vartheta \rangle$ be a rule with n connectives in the body ($n \geq 1$), out of which exactly m are constructed as ordinal sums of k_i basic connectives for each $i \in \{1, \ldots, m\}$ then:*

- *The number of homogeneous rules obtained after applying the procedure is bounded by $n + m + \sum_{i=1}^{m} k_i$, if either $\Theta = \&_i$ or $\Theta = @$ with $\vartheta = 1$; and $n + m + \sum_{i=1}^{m} k_i + 1$ otherwise.*
- *The number of transformations T_i applied by the procedure is $n+2m-1$ if either $\Theta = \&_i$ or $\Theta = @$ with $\vartheta = 1$; and $n + 2m$ otherwise.*

Proof: Concatenate the thesis of Theorem 3 with the fact that an application of T3 generates exactly $k + 1$ rules, provided that the ordinal sum was built out of k intervals, because in each homogeneous rule there is only one operator in the body. □

4 Conclusions and future work

It has been shown that the homogenization process for multi-adjoint logic programs can be adapted so that it can "decompose" conjunctors defined as an ordinal sum.

This is a very convenient extension of the previous procedure, in that the existing neural-like implementation of the fixpoint multi-adjoint semantics only considers product, Gödel, and Łukasiewicz connectives and weighted sums. Assuming that one knows the definition of a compound connective as an ordinal sum of product and Łukasiewicz connectives, then it would be possible to use the existing neural implementation to execute the program.

A different approach to this problem would be to modify directly the neural implementation so that the ordinal sum constructor can be conveniently represented by the net. Then, it would be interesting to make a comparison of the efficiency of the treatment of compound conjunctors via this extended sf-homogenization an the modified neural approach.

References

[1] C.V. Damásio and L. Moniz Pereira. Monotonic and residuated logic programs. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty, ECSQARU'01*, pages 748–759. Lect. Notes in Artificial Intelligence, 2143, 2001.

[2] J. Medina, E. Mérida-Casermeiro, and M. Ojeda-Aciego. A neural approach to extended logic programs. In *7th Intl Work Conference on Artificial and Natural Neural Networks, IWANN'03*, pages 654–661. Lect. Notes in Computer Science 2686, 2003.

[3] J. Medina and M. Ojeda-Aciego. Homogenizing multi-adjoint logic programs. In *Proc. of Intl Conference on Fuzzy Logic and Technology, EUSFLAT'03*, pages 640–644, 2003.

[4] J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. In *Logic Programming and Non-Monotonic Reasoning, LPNMR'01*, pages 351–364. Lect. Notes in Artificial Intelligence 2173, 2001.

[5] M. H. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[6] P. Vojtáš. Fuzzy logic programming. *Fuzzy Sets and Systems*, 124(3):361–370, 2001.