DOI: 10.15439/2014F387
ACSIS, Vol. 2

Fuzzy Logic Rules Modeling Similarity-based Strict Equality

Ginés Moreno, Jaime Penabad and Carlos Vázquez
Faculty of Computer Science Engineering
University of Castilla-La Mancha
Albacete (02071), Spain
Email: {Gines.Moreno,Jaime.Penabad,Carlos.Vazquez}@uclm.es

***Abstract*—A classical, but even nowadays challenging research topic in declarative programming, consists in the design of powerful notions of "equality", as occurs with the flexible (fuzzy) and efficient (lazy) model that we have recently proposed for hybrid declarative languages amalgamating functional-fuzzy-logic features. The crucial idea is that, by extending at a very low cost the notion of "strict equality" typically used in lazy functional (HASKELL) and functional-logic (CURRY) languages, and by relaxing it to the more flexible one of similarity-based equality used in modern fuzzy-logic programming languages (such as LIKELOG and BOUSI∼PROLOG), similarity relations can be successfully treated while mathematical functions are lazily evaluated at execution time. Now, we are concerned with the so-called *Multi-Adjoint Logic Programming approach*, MALP in brief, which can be seen as an enrichment of PROLOG based on *weighted rules* with a wide range of fuzzy connectives. In this work, we revisit our initial notion of SSE (*Similarity-based Strict Equality*) in order to re-model it at a very high abstraction level by means of a simple set of MALP rules. The resulting technique (which can be tested on-line in `dectau.uclm.es/sse`) not only simulates, but also surpass in our target framework, the effects obtained in other fuzzy logic languages based on similarity relations (with much more complex/reinforced unification algorithms in the core of their procedural principles), even when the current operational semantics of MALP relies on the simpler, purely syntactic unification method of PROLOG.**

***Index Terms*—Equality, Similarity, Fuzzy Logic Programming**

I. INTRODUCTION

THANKS to the high expressive power and the rule-based nature of declarative languages, their influences are growing in the design of intelligent systems and techniques related with artificial/computational intelligence, expert systems, soft-computing and so on. In particular, *Logic Programming* (LP) [1] has been widely used for problem solving and knowledge representation in the past. Nevertheless, traditional logic programming languages are not able to treat with partial truth. *Fuzzy Logic Programming* is an interesting and still growing research area that agglutinates the efforts for introducing Fuzzy Logic into Logic Programming, in order to provide these traditional languages with techniques or constructs (coming up from the mathematical background of fuzzy logic [2]) to deal with uncertainty in a natural way. In the last two decades, several fuzzy logic programming languages have been developed where, in essence, the classical SLD resolution principle of PROLOG [3] (based on syntactic unification) has been replaced by a fuzzy variant of itself, with the aim of dealing with partial truth and reasoning with uncertainty in a natural way. Most of these languages implement (extended versions of) the resolution principle introduced by Lee [4], such as Elf-Prolog [5], Fril [6], F-Prolog [7] and MALP [8]. There exists also a family of fuzzy languages based on sophisticated unification methods [9] to cope with similarity/proximity relations, as occurs with LIKELOG [10], SQLP [11] and BOUSI∼PROLOG [12], [13] (some related approaches based on *probabilistic logic programming* can be found in [14], [15]).

On the other hand, during the last three decades of investigation in the field of the integration of declarative programming paradigms (functional, fuzzy and logic), the scientific community of the area has produced important and advanced contributions related to both theoretical and practical aspects. However, whereas the functional and logic programming styles have been successfully integrated in the past and, as said before, more recently fuzzy logic has also been introduced into the logic programming paradigm, there is not precedent for a total integration of all these frameworks, apart from our preliminary approach presented in [16].

In [17], we gave a new step in this last sense, by proposing a method combining different equality models traditionally supported by each one of these declarative paradigms. It is important to take into account that an appropriate notion of equality has a crucial importance when designing the repertoire of expressive resources for a particular declarative language. In general, when we use the term "equality" in declarative programming, there are several different meanings depending of the concrete paradigm being considered. A representative (not exhaustive) list of some cases could be:

- **Syntactic equality.** It is the simplest equality model used in the context of classical pure logic programming (as occurs with PROLOG, but also in the fuzzy logic language MALP) which is simply concerned with syntactic identity. In this sense, two element are considered "equal" if they have exactly the same syntax. For instance, $f(a)$ is equal to $f(a)$ but not to $g(b)$.

This work was supported by the EU (FEDER), and the Spanish MINECO Ministry (Ministerio de Economía y Competitividad) under grant TIN2013-45732-C4-2-P.

- **Strict equality.** When considering lazy languages, both pure functional (HASKELL [18]) and integrated functional-logic (CURRY [19]) languages, this new equality notion is the only applicable one in a lazy setting, mainly due to the possible presence of non terminating functions. For instance, if the evaluation of $f(a)$ does not finish then we can not say that $f(a)$ is strictly equal to itself. And, on the contrary, two terms with different syntax, such as $g(b)$ and $h(c)$, could be proved equal if they produce the same final value (for example 0) after being evaluated by rewriting or narrowing.
- **Similarity-based equality.** As we will see in Section II, this model emerges as a direct consequence of several attempts for fuzzifying the original notion of syntactic equality, which are appreciable in the design of fuzzy logic languages such as LIKELOG, SQLP and BOUSI∼PROLOG. In this case, the idea is to allow the presence of a set of the so-called "similarity/proximity equations" between symbols of a given program. So, if we have a program with the equations $eq(a, b) = 0.5$ and $eq(f, g) = 0.3$ then, it could be proved that expressions $f(a)$ and $g(b)$ are similar with a concrete truth degree.

Here, we recall from [17] our original definition of SSE (*Similarity-based Strict Equality*), initially modeled by means of a set of rewriting rules and which fuses the last two equality versions above. The crucial idea of our method is to simply add to a given functional-logic program (written in CURRY, for instance) a set of rewriting rules defining the new symbol $\approx:\approx$ which captures similarities and thus, is implemented at a very low cost by simply performing a syntactic pre-process on programs.

The main goal of this paper is to adapt such definition to the MALP framework. In Section III we will see that SSE admits a much more natural formulation by means of a set of MALP rules instead of using rewriting rules. Moreover, although this fuzzy programming style is based on pure syntactic unification, our method introduces a similarity-based equality model without altering its core, which is useful not only for testing if two ground data terms are comparable (as occurs too with more complex languages -LIKELOG, BOUSI∼PROLOG- with extended unification algorithms), but also for producing complete lists of similar terms (not achievable by LIKELOG and BOUSI∼PROLOG). Although the technique is recasted from [20], the main contribution of the present paper consists in proving some interesting formal properties for it. Moreover, before concluding in Section V, we describe in Section IV some implementation details regarding the two main processes needed for effectively embedding SSE into MALP: after performing the reflexive-symmetric-transitive closure of a set of similarity equations for obtaining a similarity relation, then it is easily translated into a set of MALP rules modeling SSE.

II. SIMILARITY RELATIONS AND FUZZY LOGIC PROGRAMMING

As we have just said, although in principle it is not the case of MALP (whose operational semantics uses syntactic unification on its core), some fuzzy languages such as LIKELOG, SQLP and BOUSI∼PROLOG are able to treat with the mathematical notions of similarity (and proximity), by incorporating a flexible variant of unification -beyond the simpler case of PROLOG- on their procedural principles.

A similarity relation is a mathematical notion able to manipulate alternative instances of a given entity that can be considered equals with concrete truth degrees. Similarity relations are closely related with equivalence relations (and, then, to closure operators) [21]. Let us recall that a T-norm $\wedge$ in $[0, 1]$ is a binary operation $\wedge : [0, 1] \times [0, 1] \to [0, 1]$ associative, commutative, non-decreasing in both the variables, and such that $x \wedge 1 = 1 \wedge x = x$ for any $x \in [0, 1]$. Formally, a *similarity relation* $\Re$ on a domain $\mathcal{U}$ is a fuzzy subset $\Re : \mathcal{U} \times \mathcal{U} \to [0, 1]$ of $\mathcal{U} \times \mathcal{U}$ such that, $\forall x, y, z \in \mathcal{U}$, the following properties hold: reflexivity $\Re(x, x) = 1$, symmetry $\Re(x, y) = \Re(y, x)$ and transitivity $\Re(x, z) \geq \Re(x, y) \wedge \Re(y, z)$. It is important to note that this last property is not required when considering *proximity relations*. In order to simplify our developments, as in [9], we assume that $x \wedge y$ is the minimum between the two elements $x, y \in [0, 1]$.

A very simple, but effective way, to introduce similarity relations into pure logic programming, generating one of the most promising ways for the integrated paradigm of fuzzy logic programming, consists of modeling them by a set of the so-called *similarity equations* of the form $eq(s1, s2) = \alpha$, with the intended meaning that $s1$ and $s2$ are predicate/function symbols of the same arity with a similarity degree α. As in [16], we assume here that the intended similarity relation $\Re$ associated to a given program $\mathcal{P}$, is induced from the (safe) set of similarity equations of $\mathcal{P}$, verifying that the similarity degree of two symbols s_1 and s_2 is 1 if $s_1 = s_2$ or, otherwise, it is recursively defined as the transitive closure of the similarity equations.

This approach is followed, for instance, in the fuzzy logic languages LIKELOG [10] and BOUSI∼PROLOG [12], where a set of usual PROLOG clauses are accompanied by a set of similarity equations playing an important role at (fuzzy) unification time. Instead of classical *syntactic unification*, we speak now about *weak unification* [12]. Of course, the set of similarity equations is assumed to be safe in the sense that each equation connects two symbols of the same arity and nature (both predicates or both functions) and the properties of the definition of similarity relation are not violated, as occurs, for instance, with the wrong set $\{eq(a, b) = 0.5,\ eq(b, a) = 0.9\}$ which, in particular, it does not satisfy the symmetric property.

Example 2.1: Following [10], if we consider a database of books containing the fact "`book(horror,drakula)`", then goal "`?-book(adventures,Title)`" should not have classical solution in the case that there were no rule in the database unifying with atom "`book(adventures,Title)`". Nevertheless, it seems reasonable that the user considers the words "`adventures`" and "`horror`" to be *similar* with a certain degree. More precisely, if the user introduces a similarity equation like "`eq(adventures, horror) = 0.9`" into a

LIKELOG or BOUSI∼PROLOG interpreter, the system should successfully respond with a computed answer incorporating the corresponding truth degree "`0.9`" (i.e, something like the 90 % of credibility) to substitution "`Title/ drakula`", as obviously expected.

III. SSE FOR/WITH MULTI-ADJOINT LOGIC PROGRAMMING

In this section we firstly summarize the main features of the MALP language[1], next we introduce the "*Fuzzy LOgic Programming Environment for Research*", $\mathcal{FLOPER}$ in brief, developed in our research group (see [26], [27] and visit `http://dectau.uclm.es/floper/`) and finally, we illustrate and formally prove the properties of our new MALP-based model of SSE according to Figure 2.

A. MALP

We work with a first order language containing variables, function symbols, predicate symbols, constants, quantifiers ($\forall$ and $\exists$), and several arbitrary connectives such as implications ($\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m$), conjunctions ($\&_1, \&_2, \ldots,\ \&_k$), disjunctions ($\vee_1, \vee_2, \ldots, \vee_l$), and general hybrid operators ("aggregators" $@_1, @_2,\ \ldots, @_n$), used for combining/propagating truth values through the rules, and thus increasing the language expressiveness. Additionally, our language contains the values of a multi-adjoint lattice $\mathcal{L} = \langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n \rangle$, equipped with a collection of adjoint pairs $\langle \leftarrow_i, \&_i \rangle$ (where each $\&_i$ is a conjunctor intended to the evaluation of *modus ponens*) verifying the so-called *adjoint property*: $\forall x, y, z \in L,\ \ x \preceq (y \leftarrow_i z)$ if and only if $(x \,\&_i\, z) \preceq y$. The set of truth values L may be the carrier of any complete bounded lattice, as for instance occurs with the set of real numbers in the interval $[0, 1]$ with their corresponding ordering $\leq$. A *rule* is a formula $[A \leftarrow_i \mathcal{B}\ with\ \alpha]$, where A is an atomic formula (usually called the *head*), $\mathcal{B}$ (which is called the *body*) is a formula built from atomic formulas $B_1, \ldots, B_n$ ($n \geq 0$), truth values of L and conjunctions, disjunctions and aggregators, and finally $\alpha \in L$ is the "weight" or *truth degree* of the rule. A rule with empty body, written $[A\ with\ \alpha]$, is called *fact*. Consider, for instance, the following program $\mathcal{P}$ composed by three rules with associated multi-adjoint lattice $\langle [0, 1], \leq, \leftarrow_{\mathtt{P}}, \&_{\mathtt{P}}, \leftarrow_{\mathtt{G}}, \&_{\mathtt{G}} \rangle$ (where labels `P` and `G` mean for *Product logic* and *Gödel intuitionistic logic*, respectively, with the following connective definitions: "$\leftarrow_{\mathtt{P}}\ (x, y) = \min(1, x/y)$", "$\&_{\mathtt{P}}(x, y) = x * y$", "$\leftarrow_{\mathtt{G}}\ (x, y) = 1$ if $y \leq x$ or x otherwise" and "$\&_{\mathtt{G}}(x, y) = min(x, y)$"):

$$\begin{array}{llllll}
\mathcal{R}_1: & p(X) & \leftarrow_{\mathtt{P}} & q(X,Y) \,\&_{\mathtt{G}}\, r(Y) & with & 0.8 \\
\mathcal{R}_2: & q(a,Y) & & & with & 0.9 \\
\mathcal{R}_3: & r(b) & & & with & 0.7
\end{array}$$

[1]As said before, this fuzzy language uses a syntax near to PROLOG and enjoys high level of flexibility, for which we give some theoretical/practical reinforcements in our precedent works [22], [23], [24], [25].

In order to describe the procedural semantics of the multi–adjoint logic language, in the following we denote by $\mathcal{C}[A]$ a formula where A is a sub-expression (usually an atom) which occurs in the –possibly empty– one hole context $\mathcal{C}[]$ whereas $\mathcal{C}[A/A']$ means the replacement of A by A' in context $\mathcal{C}[]$, and $mgu(E)$ is the *most general unifier* of an equation set E. The pair $\langle \mathcal{Q}; \sigma \rangle$ composed by a goal and a substitution is called a *state*. So, given a program $\mathcal{P}$, an *admissible computation* is formalized as a state transition system, whose transition relation $\overset{\text{AS}}{\leadsto}$ is the smallest relation satisfying the following *admissible rules*:

1) $\langle \mathcal{Q}[A]; \sigma \rangle \overset{\text{AS}}{\leadsto} \langle (\mathcal{Q}[A/v \&_i \mathcal{B}])\theta; \sigma\theta \rangle$ if A is the selected atom in goal $\mathcal{Q}$, $[A' \leftarrow_i \mathcal{B}\ with\ v] \in \mathcal{P}$, where $\mathcal{B}$ is not empty, and $\theta = mgu(\{A' = A\})$.

2) $\langle \mathcal{Q}[A]; \sigma \rangle \overset{\text{AS}}{\leadsto} \langle (\mathcal{Q}[A/v])\theta; \sigma\theta \rangle$ if $[A'\ with\ v] \in \mathcal{P}$ and $\theta = mgu(\{A' = A\})$.

The following derivation illustrates our definition (note that the exact program rule used -after being renamed- in the corresponding step is annotated as a super–index of the $\overset{\text{AS}}{\leadsto}$ symbol, whereas exploited atoms appear underlined and id represents the empty substitution):

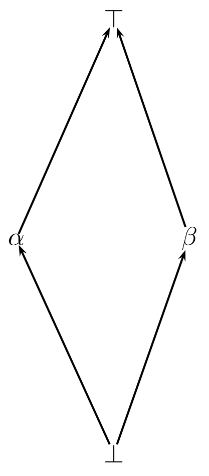
$$\begin{array}{ll}
\langle \underline{p(X)}; id \rangle & \overset{\text{AS}}{\leadsto}^{\mathcal{R}_1} \\
\langle 0.8 \,\&_{\mathtt{P}}\, (\underline{q(X_1, Y_1)} \,\&_{\mathtt{G}}\, r(Y_1)); \{X/X_1\} \rangle & \overset{\text{AS}}{\leadsto}^{\mathcal{R}_2} \\
\langle 0.8 \,\&_{\mathtt{P}}\, (0.9 \,\&_{\mathtt{G}}\, \underline{r(Y_2)}); \{X/a, X_1/a, Y_1/Y_2\} \rangle & \overset{\text{AS}}{\leadsto}^{\mathcal{R}_3} \\
\langle 0.8 \,\&_{\mathtt{P}}\, (0.9 \,\&_{\mathtt{G}}\, \underline{0.7}); \{X/a, X_1/a, Y_1/b, Y_2/b\} \rangle &
\end{array}$$

The final formula without atoms can be directly interpreted in lattice $\mathcal{L}$ to obtain the desired *fuzzy computed answer* (or *f.c.a.*, in brief), where the substitution only contains bindings associated to variables of the initial goal. So, since $0.8 \,\&_{\mathtt{P}}\, (0.9 \,\&_{\mathtt{G}}\, 0.7) = 0.8{*}\min(0.9, 0.7) = 0.56$, in our case the fuzzy computed answer is $\langle 0.56, \{X/a\} \rangle$ indicating that goal $p(X)$ is true at 56 % when X is a.

B. $\mathcal{FLOPER}$

As detailed in [28], [26], our parser has been implemented by using the classical DCG's (*Definite Clause Grammars*) resource of the PROLOG language, since it is a convenient notation for expressing grammar rules. Once the application is loaded inside a PROLOG interpreter (such as Sicstus or SWI), it shows a menu which includes options for loading/compiling, parsing, listing and saving fuzzy programs, as well as for executing/debugging goals and managing multi-adjoint lattices.

All these actions are based in the compilation of the fuzzy code into standard PROLOG code. The key point is to extend each atom with an extra argument, called *truth variable* of the form "`_TVi`", which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. For instance, the first rule in our target program is translated into



```
member(bottom).    member(alpha).
member(beta).      member(top).

bot(bottom).       top(top).

leq(bottom,X).      leq(alpha,alpha).
leq(alpha,top).    leq(beta,beta).
leq(beta,top).     leq(X,top).

and_godel(X,Y,Z) :- pri_inf(X,Y,Z).

pri_inf(bottom,X,bottom) :- !.
pri_inf(alpha,X,alpha) :- leq(alpha,X),!.
pri_inf(beta,X,beta) :- leq(beta,X),!.
pri_inf(top,X,X) :- !.
pri_inf(X,Y,bottom).
```

Figure 1. A finite, partially ordered multi-adjoint lattice modeled in PROLOG

$$\begin{array}{lr}
\texttt{sse}(c, d) & \texttt{with}\ \Re(c, d) \\
\texttt{sse}(c(x_1, .., x_n), d(y_1, .., y_n)) \ \leftarrow_G \texttt{sse}(x_1, y_1)\ \&_G \ldots \&_G\ \texttt{sse}(x_n, y_n) & \texttt{with}\ \Re(c, d)
\end{array}$$

Figure 2. MALP Rules defining "Similarity-based Strict Equality"

the clause:

```
p(X,_TV0) :- q(X,Y,_TV1), r(Y,_TV2),
             and_godel(_TV1,_TV2,_TV3),
             and_prod(0.8,_TV3, _TV0).
```

Moreover, the remaining rules in our fuzzy program, becomes the pure PROLOG facts "`q(a,Y,0.9)`" and "`r(b,0.7)`", whereas the corresponding lattice is expressed by these clauses (the meaning of the mandatory predicates `member`, `top`, `bot` and `leq` is obvious):

```
member(X) :- number(X),0=<X,X=<1.
bot(0).                  top(1).
leq(X,Y) :- X=<Y.
and_godel(X,Y,Z):- pri_min(X,Y,Z).
pri_min(X,Y,Z) :- (X=<Y,Z=X;X>Y,Z=Y).
and_prod(X,Y,Z) :- pri_prod(X,Y,Z).
pri_prod(X,Y,Z) :- Z is X * Y
```

Finally, a fuzzy goal like "`p(X)`", is obviously translated into the pure PROLOG goal: "`p(X, Truth_degree)`" (note that the last truth degree variable is not anonymous now) for which, after choosing option "`run`", the PROLOG interpreter returns the desired fuzzy computed answer $[\texttt{Truth\_degree} = 0.56, \texttt{X} = \texttt{a}]$. Note that 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, thus producing the illusion on the final user of being working with a purely fuzzy logic programming tool.

Moreover, it is also possible to select into the menu of $\mathcal{FLOPER}$, options "`tree`" and "`depth`", which are useful for tracing execution trees and fixing the maximum length allowed for their branches (initially 3), respectively. To finish this block, in Figure 1 we show the PROLOG clauses modeling a lattice which will be used afterwards in Section IV. Here, apart for dealing with a partially ordered lattice, we use the conjunction of the *Gödel* logic described in this non numeric case as: $\&_{\texttt{G}}(x, y) \triangleq inf(x, y)$. From `http://dectau.uclm.es/floper/` it is possible can download our last version of the $\mathcal{FLOPER}$ tool, which incorporates a graphical interface as shown in Figures 3 and 4.

C. SSE, MALP *and* $\mathcal{FLOPER}$

Now, we are ready to illustrate and prove the properties of our MALP-based model of SSE which is defined according to [20] in Figure 2, where we assume that both c and d are constants (i.e., constructor symbols with arity 0) in the first rule, or both are functions with the same arity n in the second rule and then, $\Re(c, d)$ represents the similarity degree between such pair of symbols with the same arity. In order to illustrate our technique, assume that we plan to compare data terms built with constants "mary" and "maria", which have a similarity degree of 80% and function symbols (with arity one) "brother" and "sibling" which are similar at 90%. According to our

MALP-based definition of SSE we generate a set of MALP rules using the "min" operator (based on *Gödel logic*, as usual in LIKELOG and BOUSI∼PROLOG) to propagate similarity degrees. Instead, in the following MALP program loaded into $\mathcal{FLOPER}$ we have used a version inspired on "product logic' (in the following section we describe an application which allows to select the desired conjunction operator or t-norm for composing similarity degrees):

```
sse(maria,maria)                   with 1.
sse(mary,mary)                     with 1.
sse(mary,maria)                    with 0.8.
sse(maria,mary)                    with 0.8.
sse(sibling(X),sibling(Y)) <prod
                         sse(X,Y) with 1.
sse(brother(X),brother(Y)) <prod
                         sse(X,Y) with 1.
sse(sibling(X),brother(Y)) <prod
                         sse(X,Y) with 0.9.
sse(brother(X),sibling(Y)) <prod
                         sse(X,Y) with 0.9.
```

Now, for a goal like "`sse(brother(mary), sibling(maria))`", our technique tests that both parameters are similar terms (with degree $0.9 * 0.8 = 0.72$) in the same way than LIKELOG and BOUSI∼PROLOG. Anyway, these last languages only would report just one solution for goals "`sse(brother(mary),X)`" and "`sse(X,Y)`" (the answers computed by LIKELOG and BOUSI∼PROLOG for those queries would include the bindings "`{X/ brother(mary)}`" and "`{ X/ Y }`", respectively, both ones with similarity degree 1), whereas our system is able to provide the corresponding four answer for the first query shown in Figure 3, as well as infinite solutions for the second goal (some of them displayed in Figure 4), including the following ones:

```
[Truth_degree=1,X=mary,Y=mary]
[Truth_degree=0.8,X=mary,Y=maria]
[Truth_degree=0.9, X=brother(maria),
                   Y=sibling(maria)]
[Truth_degree=0.72,X=brother(mary),
                   Y=sibling(maria)]
```

In order to formally prove the properties we have just illustrated, it is mandatory to introduce the following auxiliary definition:

Definition 3.1 (Similar terms): Let t and t' be two ground terms, $\Re$ a similarity relation and $\mathcal{L} = \langle L, \preceq, \leftarrow, \& \rangle$ a multi-adjoint lattice. We say that t and t' are similar terms according $\Re$ and $\&$ with similarity degree $s \in L$, if the evaluation of function $\Phi(t, t')$ returns $s \neq \bot$, where function Φ is recursively defined as follows:

$$\Phi(t,t') = \begin{cases} \Re(t,t'), & \text{if } t \text{ and } t' \text{ are constants} \\ \Re(c,c')\&\Phi(t_1,t'_1)\& & \text{if } t = c(t_1,\ldots,t_n) \text{ and} \\ \ldots\&\Phi(t_n,t'_n) & \quad t' = c'(t'_1,\ldots,t'_n) \end{cases}$$

The following result reveals the ability of our technique for testing similar terms.

Theorem 3.2: Let t and t' be two ground terms, $\mathcal{L} = \langle L, \preceq, \leftarrow, \& \rangle$ a multi-adjoint lattice, $\Re$ a similarity relation and $\mathcal{P}^{\Re}_{sse}$ the set of MALP rules defining predicate sse w.r.t. $\Re$. Then, t and t' are similar terms according $\Re$ and $\&$ with similarity degree $s \in L$, iff $\langle s, id \rangle$ is a fuzzy computed answer for goal $sse(t, t')$ in $\mathcal{P}^{\Re}_{sse}$.

Proof: We prove this claim by structural induction on the shape of t and t'.

- Base case. We assume here that t and t' are similar constants, and then, $\Re(t, t') = s \neq \bot$ whereas rule $[\mathcal{R} : sse(t, t')\ with\ s]$ belongs to $\mathcal{P}^{\Re}_{sse}$. Then, it is easy to see that $\Phi(t, t') = \Re(t, t') = s$ as well as to perform with rule $\mathcal{R}$ the following admissible step $\langle sse(t, t'), id \rangle \overset{\text{AS}}{\leadsto}^{\mathcal{R}} \langle s, id \rangle$.
- Induction step. Now we have that $t = c(t_1, \ldots, t_n)$ and $t' = c'(t'_1, \ldots, t'_n)$. Assuming that $\Re(c, c') = s_0 \neq \bot$ and $\Phi(t_i, t'_i) = s_i \neq \bot$, $1 \leq i \leq n$, then $\Phi(t, t') = s_0\&s_1\&\ldots\&s_n \neq \bot$. Moreover, since our technique generates the rule (which belongs to $\mathcal{P}^{\Re}_{sse}$):

$$\mathcal{R} : sse(c(x_1,\ldots,x_n), c(x'_1,\ldots,x'_n)) \leftarrow \\ sse(x_1,x'_1)\&\ldots\&sse(x_n,x'_n)\ \ with\ s_0$$

and by the inductive hypothesis we can assume that $\langle s_i, id \rangle$ is a fuzzy computed answer for goal $sse(t_i, t'_i)$, $1 \leq i \leq n$, then it is possible to generate the following sequence of admissible steps (for readability reasons, we omit in the substitution component of each state the bindings associated to variables not belonging to the initial goal):

$$\begin{array}{ll} \langle sse(c(t_1,\ldots,t_n), c'(t'_1,\ldots,t'_n)); id \rangle & \overset{\text{AS}}{\leadsto}^{\mathcal{R}} \\ \langle s_0\ \&\ sse(t_1,t'_1)\ \&\ldots\&\ sse(t_n,t'_n); id \rangle & \overset{\text{AS}}{\leadsto} \ldots \overset{\text{AS}}{\leadsto} \\ \langle s_0\ \&\ s_1\ \&\ldots\&\ sse(t_n,t'_n); id \rangle & \overset{\text{AS}}{\leadsto} \ldots \overset{\text{AS}}{\leadsto} \\ \langle s_0\ \&\ s_1\ \&\ldots\&\ s_n; id \rangle & \end{array}$$

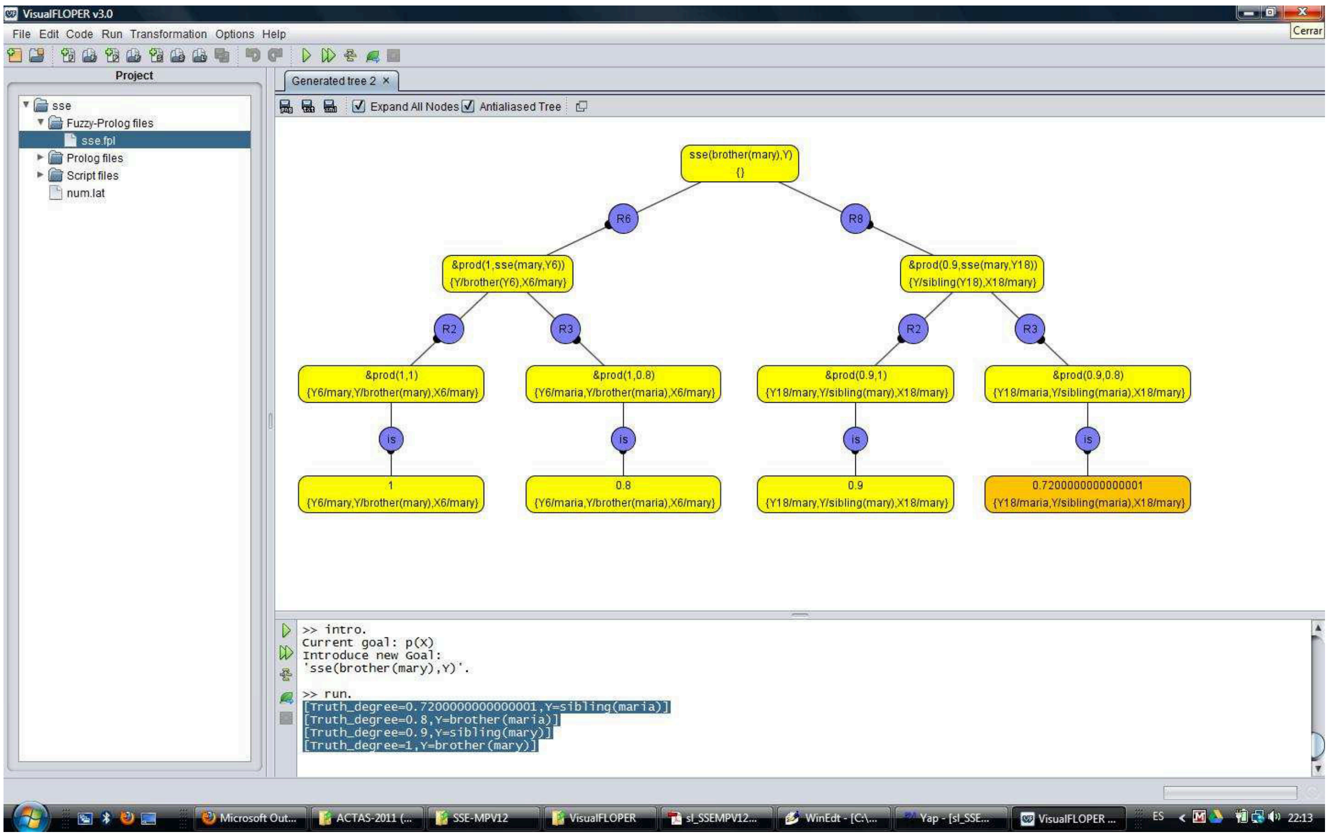
which concludes our proof. ■

Next theorem reinforces the previous one by establishing the capability of our technique for generating (not only for testing) pairs of similar ground terms.

Theorem 3.3: Let t and t' be two ground terms, x a variable, $\mathcal{L} = \langle L, \preceq, \leftarrow, \& \rangle$ a multi-adjoint lattice, $\Re$ a similarity relation and $\mathcal{P}^{\Re}_{sse}$ the set of MALP rules defining predicate sse w.r.t. $\Re$. Then, t and t' are similar terms according $\Re$ and $\&$ with similarity degree $s \in L$, iff $\langle s, \{x/t'\} \rangle$ is a fuzzy computed answer for goal $sse(t, x)$ in $\mathcal{P}^{\Re}_{sse}$.

Proof: Our proof is based again on structural induction on the shape of t, and it clearly resembles the one built for Theorem 3.2 but pointing out now the effects on the variables of the original goal.

- Base case. We assume here that t and t' are similar constants, and then, $\Re(t, t') = s \neq \bot$ whereas rule $[\mathcal{R} : sse(t, t')\ with\ s]$ belongs to $\mathcal{P}^{\Re}_{sse}$. Then, obviously $\phi(t, t') = \Re(t, t') = s$ whereas it is possible too to perform with rule $\mathcal{R}$ the following admissible step $\langle sse(t, x), id \rangle \overset{\text{AS}}{\leadsto}^{\mathcal{R}} \langle s, \{x/t'\} \rangle$.
- Induction step. Now we have that $t = c(t_1, \ldots, t_n)$ and $t' = c'(t'_1, \ldots, t'_n)$. Assuming that $\Re(c, c') = s_0 \neq \bot$


VisualFLOPER v3.0
File Edit Code Run Transformation Options Help
Project
sse
Fuzzy-Prolog files
sse.fpl
Prolog files
Script files
num.lat
Generated tree 2
Expand All Nodes
Antialiased Tree
sse(brother(mary),Y)
{}
R6
R8
&prod(1,sse(mary,Y6))
{Y/brother(Y6),X6/mary}
&prod(0.9,sse(mary,Y18))
{Y/sibling(Y18),X18/mary}
R2
R3
R2
R3
&prod(1,1)
{Y6/mary,Y/brother(mary),X6/mary}
&prod(1,0.8)
{Y6/maria,Y/brother(maria),X6/mary}
&prod(0.9,1)
{Y18/mary,Y/sibling(mary),X18/mary}
&prod(0.9,0.8)
{Y18/maria,Y/sibling(maria),X18/mary}
is
is
is
is
1
{Y6/mary,Y/brother(mary),X6/mary}
0.8
{Y6/maria,Y/brother(maria),X6/mary}
0.9
{Y18/mary,Y/sibling(mary),X18/mary}
0.7200000000000001
{Y18/maria,Y/sibling(maria),X18/mary}

```
>> intro.
Current goal: p(X)
Introduce new Goal:
'sse(brother(mary),Y)'.

>> run.
[Truth_degree=0.7200000000000001,Y=sibling(maria)]
[Truth_degree=0.8,Y=brother(maria)]
[Truth_degree=0.9,Y=sibling(mary)]
[Truth_degree=1,Y=brother(mary)]
```

Figure 3. Screen-shot of a work session with $\mathcal{FLOPER}$

and $\Phi(t_i, t'_i) = s_i \neq \bot$, $1 \leq i \leq n$, then $\Phi(t, t') = s_0 \& s_1 \& \ldots \& s_n \neq \bot$. Moreover, since our technique generates the rule (which belongs to $\mathcal{P}^{\Re}_{sse}$):

$$\mathcal{R} : sse(c(x_1, \ldots, x_n), c(x'_1, \ldots, x'_n)) \leftarrow \\ sse(x_1, x'_1) \& \ldots \& sse(x_n, x'_n) \quad with \ s_0$$

and by the inductive hypothesis we can assume that $\langle s_i, \{x'_i/t'_i\}\rangle$ is a fuzzy computed answer for goal $sse(t_i, x'_i)$, $1 \leq i \leq n$, then it is possible to generate the derivation shown in Figure 5 (for simplifying, we only include in the substitution component of each state those bindings which are relevant for our purposes) which concludes our proof. ■

The repeated application of the previous theorem implies the following result which, in essence, confirms the power of our method for producing all pairs of similar data terms.

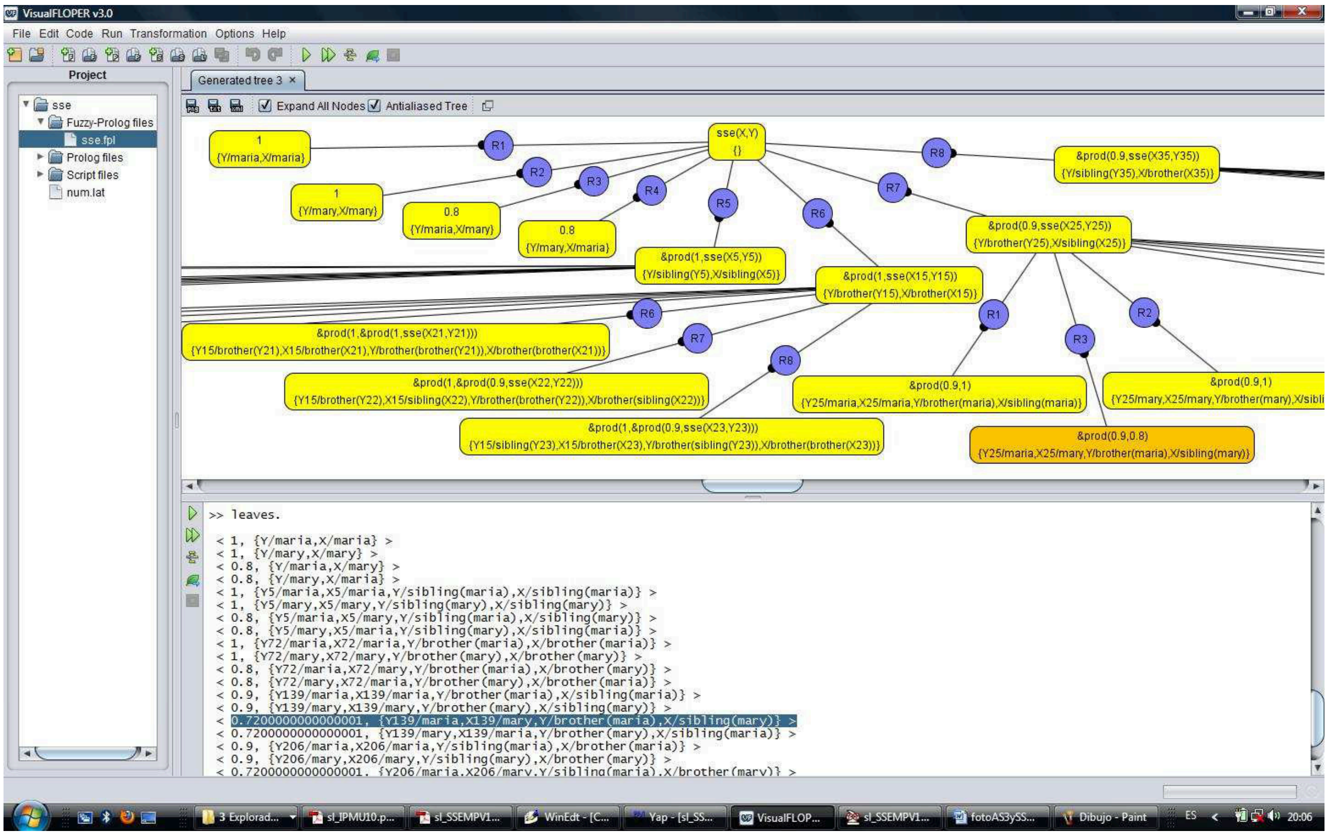
Corollary 3.1: Let t and t' be two ground terms, x and x' two variables, $\mathcal{L} = \langle L, \preceq, \leftarrow, \& \rangle$ a multi-adjoint lattice, $\Re$ a similarity relation and $\mathcal{P}^{\Re}_{sse}$ the set of MALP rules defining predicate sse w.r.t. $\Re$. Then, t and t' are similar terms according $\Re$ and $\&$ with similarity degree $s \in L$, iff $\langle s, \{x/t, x'/t'\}\rangle$ is a fuzzy computed answer for goal $sse(x, x')$ in $\mathcal{P}^{\Re}_{sse}$.

IV. Implementation Issues

We start this section by firstly describing in subsection IV-A how users can introduce into the new SSE tool (written in Prolog and freely accessible from `http://dectau.uclm.es/sse/`) a small set of similarity equations with a natural and very easy syntax. After that, the tool performs the reflexive-symmetric-transitive closure of that specification in order to obtain a similarity relation $\Re$ which is translated into a Prolog program, as explained in sub-section IV-B. Finally, the application uses $\Re$ to generate a MALP program defining SSE, as described in sub-section IV-C.

A. Syntax for Similarity Files

To specify a similarity relation, it is mandatory to load a file with extension '.sim' into the tool. This file is intended to contain a set of similarity equations, where each equation is expressed by separating two literals (the ones to be considered similar) with the '$\sim$' symbol, and adding a *truth value* to the similarity (usually, a number of the real interval [0,1], but our tool also admits an element from any multi-adjoint lattice, in contrast with other fuzzy languages such as Bousi∼Prolog or Likelog) after the '=' symbol. So, for instance, `brother ~ sibling = 0.9`, is a valid similarity equation. Our syntax also allows to specify the arity of each symbol after a suffixed slash (i.e. '`brother/1`'). Thus, it is possible to discriminate between functors with the same name but different arities. When the user does not include arity information, it is simple assumed to be zero. To relate literals without arity specification (i.e., with no arity


VisualFLOPER v3.0
File Edit Code Run Transformation Options Help
Project
sse
Fuzzy-Prolog files
sse.fpl
Prolog files
Script files
num.lat
Generated tree 3
Expand All Nodes
Antialiased Tree
sse(X,Y) {}
R1
R2
R3
R4
R5
R6
R7
R8
1 {Y/maria,X/maria}
1 {Y/mary,X/mary}
0.8 {Y/maria,X/mary}
0.8 {Y/mary,X/maria}
&prod(1,sse(X5,Y5)) {Y/sibling(Y5),X/sibling(X5)}
&prod(1,sse(X15,Y15)) {Y/brother(Y15),X/brother(X15)}
&prod(0.9,sse(X25,Y25)) {Y/brother(Y25),X/sibling(X25)}
&prod(0.9,sse(X35,Y35)) {Y/sibling(Y35),X/brother(X35)}
&prod(1,&prod(1,sse(X21,Y21))) {Y15/brother(Y21),X15/brother(X21),Y/brother(brother(Y21)),X/brother(brother(X21))}
&prod(1,&prod(0.9,sse(X22,Y22))) {Y15/brother(Y22),X15/sibling(X22),Y/brother(brother(Y22)),X/brother(sibling(X22))}
&prod(1,&prod(0.9,sse(X23,Y23))) {Y15/sibling(Y23),X15/brother(X23),Y/brother(sibling(Y23)),X/brother(brother(X23))}
&prod(0.9,1) {Y25/maria,X25/maria,Y/brother(maria),X/sibling(maria)}
&prod(0.9,1) {Y25/mary,X25/mary,Y/brother(mary),X/sibli
&prod(0.9,0.8) {Y25/maria,X25/mary,Y/brother(maria),X/sibling(mary)}
3 Explorad...
sl_IPMU10.p...
sl_SSEMPV1...
WinEdt - [C...
Yap - [sl_SS...
VisualFLOP...
sl_SSEMPV1...
fotoAS3ySS...
Dibujo - Paint
ES
20:06

```
>> leaves.

 < 1, {Y/maria,X/maria} >
 < 1, {Y/mary,X/mary} >
 < 0.8, {Y/maria,X/mary} >
 < 0.8, {Y/mary,X/maria} >
 < 1, {Y5/maria,X5/maria,Y/sibling(maria),X/sibling(maria)} >
 < 1, {Y5/mary,X5/mary,Y/sibling(mary),X/sibling(mary)} >
 < 0.8, {Y5/maria,X5/mary,Y/sibling(maria),X/sibling(mary)} >
 < 0.8, {Y5/mary,X5/maria,Y/sibling(mary),X/sibling(maria)} >
 < 1, {Y72/maria,X72/maria,Y/brother(maria),X/brother(maria)} >
 < 1, {Y72/mary,X72/mary,Y/brother(mary),X/brother(mary)} >
 < 0.8, {Y72/maria,X72/mary,Y/brother(maria),X/brother(mary)} >
 < 0.8, {Y72/mary,X72/maria,Y/brother(mary),X/brother(maria)} >
 < 0.9, {Y139/maria,X139/maria,Y/brother(maria),X/sibling(maria)} >
 < 0.9, {Y139/mary,X139/mary,Y/brother(mary),X/sibling(mary)} >
 < 0.7200000000000001, {Y139/maria,X139/mary,Y/brother(maria),X/sibling(mary)} >
 < 0.7200000000000001, {Y139/mary,X139/maria,Y/brother(mary),X/sibling(maria)} >
 < 0.9, {Y206/maria,X206/maria,Y/sibling(maria),X/brother(maria)} >
 < 0.9, {Y206/mary,X206/mary,Y/sibling(mary),X/brother(mary)} >
 < 0.7200000000000001, {Y206/maria,X206/mary,Y/sibling(maria),X/brother(mary)} >
```

Figure 4. $\mathcal{FLOPER}$ showing three levels of an infinite evaluation tree

$$
\begin{array}{lr}
\langle sse(c(t_1,\ldots,t_n),x)); id\rangle & \overset{\text{AS}}{\leadsto}{}^{\mathcal{R}} \\
\langle s_0 \ \&\ sse(t_1,x'_1)\ \&\ldots\&\ sse(t_n,x'_n); \{x/c'(x'_1,\ldots,x'_n), x_1/t_1,\ldots,x_n/t_n\}\rangle & \overset{\text{AS}}{\leadsto} \\
\qquad\qquad\qquad\qquad \ldots \ \overset{\text{AS}}{\leadsto}\ \ldots & \overset{\text{AS}}{\leadsto} \\
\langle s_0 \ \&\ s_1 \ \&\ldots\&\ sse(t_n,x'_n); \{x/c'(t'_1,\ldots,x'_n), x_1/t_1,\ldots,x_n/t_n, x'_1/t'_1\}\rangle & \overset{\text{AS}}{\leadsto} \\
\qquad\qquad\qquad\qquad \ldots \ \overset{\text{AS}}{\leadsto}\ \ldots & \overset{\text{AS}}{\leadsto} \\
\langle s_0 \ \&\ s_1 \ \&\ldots\&\ s_n); \{x/c'(t'_1,\ldots,t'_n), x_1/t_1,\ldots,x_n/t_n, x'_1/t'_1,\ldots,x'_n/t'_n\}\rangle &
\end{array}
$$

Figure 5. Proof of Theorem 3.3

discrimination), we need to write an underscore after the slash symbol (i.e., ‘`brother/_`’).

Example 4.1: Consider now the following specification of a similarity relation:

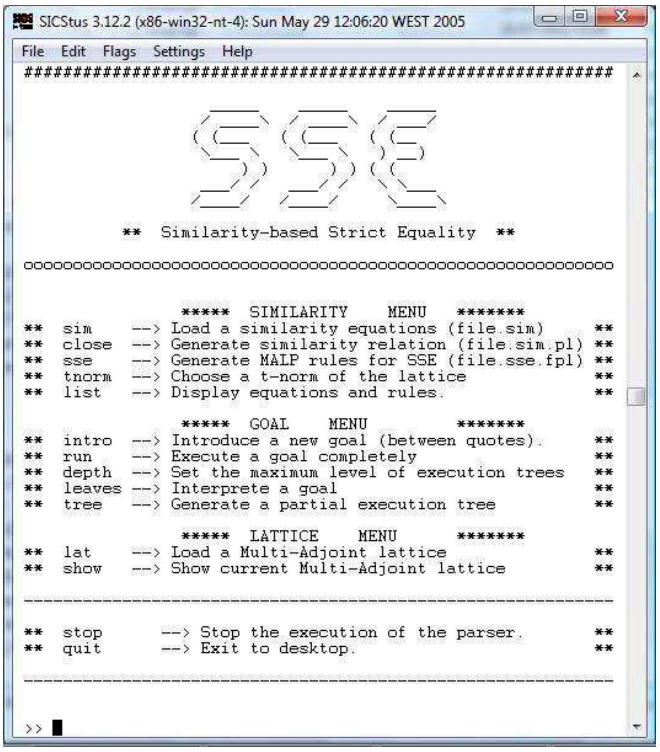
```
mary ∼ maria = 0.8.
sibling/1 ∼ brother/1 = 0.9.
```

It is not necessary to add all similarity equations (for instance, the reflexive equation relating `mary` with `mary`), since the tool is able to “complete” the relation by performing the reflexive, symmetric and transitive closure of the given set of equations, as we will see in sub-section IV-B.

Note again that since our tool can work with different multi-adjoint lattices, similarity equations can be also described beyond the real interval $[0, 1]$: the only required condition is that the similarity degrees of equations have to be members of the multi-adjoint lattice associated to the program or, in other words, with the lattice currently loaded into the system (see Figure IV-A).

B. Closure and Translation to PROLOG

Each similarity equation from the “.sim” file is translated into a PROLOG clause holding all its information. So, a similarity equation `A/`n_A $\sim$ `B/`n_B `= V` is coded as fact `r((A,`n_A`),(B,`n_B`),V)`, thus including the arity of each literal. The previous Example 4.1 (based on real numbers in the unit interval) should then be translated into:



SICStus 3.12.2 (x86-win32-nt-4): Sun May 29 12:06:20 WEST 2005

File Edit Flags Settings Help

```
################################################################

               **  Similarity-based Strict Equality  **

oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo

                    *****  SIMILARITY   MENU   *******
**  sim     --> Load a similarity equations (file.sim)      **
**  close   --> Generate similarity relation (file.sim.pl)  **
**  sse     --> Generate MALP rules for SSE (file.sse.fpl)  **
**  tnorm   --> Choose a t-norm of the lattice              **
**  list    --> Display equations and rules.                **

                    *****  GOAL   MENU        *******
**  intro   --> Introduce a new goal (between quotes).      **
**  run     --> Execute a goal completely                   **
**  depth   --> Set the maximum level of execution trees    **
**  leaves  --> Interprete a goal                           **
**  tree    --> Generate a partial execution tree           **

                    *****  LATTICE   MENU     *******
**  lat     --> Load a Multi-Adjoint lattice                **
**  show    --> Show current Multi-Adjoint lattice          **

----------------------------------------------------------------

**  stop      --> Stop the execution of the parser.         **
**  quit      --> Exit to desktop.                          **

----------------------------------------------------------------

>> █
```

Figure 6. Main menu of the SSE application implemented with PROLOG

Algorithm 1

```
Require: An adjacency matrix $M = [m_{ij}]$, representing a fuzzy binary relation $R$ on a set $A$, whose elements preserve transitivity and with all the elements of the superior triangular matrix set to $\bot$.
Ensure: The adjacency matrix $M^{\equiv}$ corresponding to the reflexive, symmetric, transitive closure of $R$.
    for all $\langle i,i\rangle$ in $M$ do {Build the reflexive closure}
2:      $m_{ii} := \top$;
    end for
4:  for all $\langle i,j\rangle$ in $M$, such that $m_{ij} \neq \bot$ do {Build the symmetric closure}
        $m_{ji} := m_{ij}$;
6:  end for
    for all column $k$ and entry $\langle i,j\rangle$ in $M$ do {Build the transitive closure}
8:      $m_{ij} := m_{ij} \vee (m_{ik} \wedge m_{kj})$; where "$\vee$" and "$\wedge$" are, respectively, the supremum and infimum operators;
    end for
10: $M^{\equiv} := M$
```

```
r((mary, 0), (maria, 0), 0.8).
r((sibling, 1), (brother, 1), 0.9).
```

All these facts are saved in their own PROLOG module "sim", in order to avoid collision of names. Also, the system saves the translated code into a file with the same name but extension "sim.pl".

Once this process has finished, the tool completes the intended similarity relation by performing the reflexive-symmetric-transitive closure according to Algorithm 1 (that we have just implemented in PROLOG) which is inspired by the one described in [29], [12], but generalizing it in order to deal with (multi-adjoint) lattices beyond the $[0, 1]$ case[2]:

As a result of performing this algorithm, the intended similarity relation is completed, and then, all similarity equations are successfully stored into module "sim" as PROLOG facts. The next step is to write these similarity equations into a file with the same name of that of the specification but extension "sim.pl", thus pointing out that the new file includes the same information, but using PROLOG syntax.

For instance, the closure of the similarity specification from Example 4.1 should return the relation of the following table, where a cell $\langle i, j\rangle$ gives the corresponding similarity degree between two symbols.

	maria	mary	brother	sibling
maria	1	0.8	0	0
mary	0.8	1	0	0
brother	0	0	1	0.9
sibling	0	0	0.9	1

This table is modeled by means of the following set of PROLOG facts resulting from the translation process previously described:

```
sim((maria,0),(maria,0),1).
sim((maria,0),(mary,0),0.8).
sim((mary,0),(maria,0),0.8).
sim((mary,0),(mary,0),1).
sim((brother,1),(brother,1),1).
sim((brother,1),(sibling,1),0.9).
sim((sibling,1),(brother,1),0.9).
sim((sibling,1),(sibling,1),1).
```

Example 4.2: For the following two similarity equations using degrees of the partially ordered lattice in Figure 1 (see again sub-section III-B), we show its corresponding table and associated PROLOG facts below (note that the 'top' element is the truth degree for all reflexive equations):

$$\mathrm{c} \sim \mathrm{d} = \mathrm{alpha}.$$
$$\mathrm{f}/2 \sim \mathrm{g}/2 = \mathrm{beta}.$$

	c	d	f	g
c	top	alpha	bot	bot
d	alpha	top	bot	bot
f	bot	bot	top	beta
g	bot	bot	beta	top

```
sim((c,0),(c,0),top).
sim((c,0),(d,0),alpha).
sim((d,0),(c,0),alpha).
sim((d,0),(d,0),top).
```

[2]Note that the algorithm can work with any particular multi-adjoint lattice: since any complete lattice (with supremum, infimum and a concrete ordering relation) is valid, then any multi-adjoint lattice is valid too.

Algorithm 2

```
Require: A set of similarity equations $S = \{S_i, i \in \{0,\ldots,N\}\}$ of the form $S_i = \{A/n_A \sim B/n_B = V\}$, where $A$ and $B$ are function symbols (possibly constants), $n_A$ and $n_B$ are their respective arities and $V$ is the corresponding similarity degree.
Ensure: A set of MALP rules $R = \{R_i, i \in \{0,\ldots,N\}\}$.
  for all $S_i = \{A/n_A \sim B/n_B = V\}$ in $S$ do
    $body :=$ "$with'' + V$;
    for all $j \in \{n_A,\ldots,1\}$ do
      $body :=$ "$, sse(X_j,Y_j)'' + body$;
    end for
    if $n_A > 0$ then
      $body :=$ "$<- sse(X_1,Y_1)'' + body$;
    end if
    $R_i :=$ "$sse(A(X_1,\ldots,X_{n_A}), B(Y_1,\ldots,Y_{n_B}))''$;
    $R_i := R_i + body$;
  end for
```

```
sim((f,2),(f,2),top).
sim((f,2),(g,2),beta).
sim((g,2),(f,2),beta).
sim((g,2),(g,2),top).
```

C. *From similarities to* MALP *rules modeling* SSE

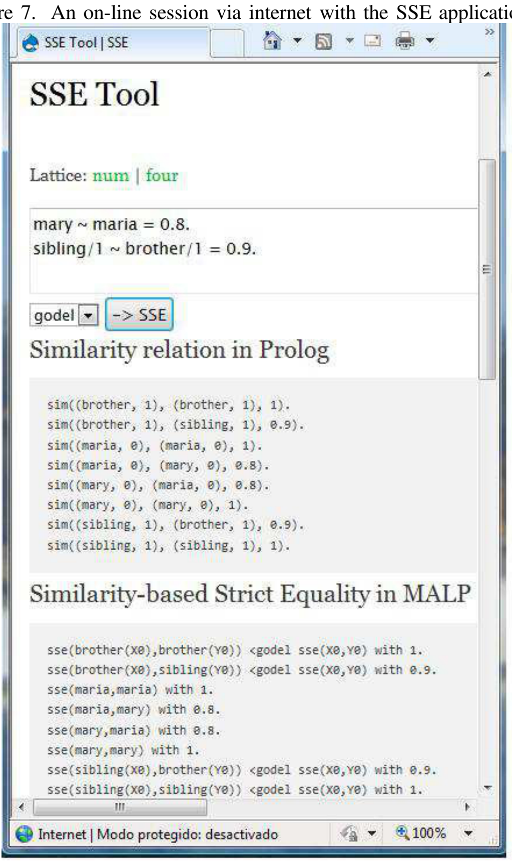
The last step consists on translating the similarity relation from its PROLOG syntax to the MALP syntax. Algorithm 2 performs such process, where the input is the set of PROLOG facts obtained after performing the closure, and the output is the intended MALP program:

Since $R = \{R_i, i \in \{0,\ldots,N\}\}$ is a set of MALP rules, it is also a valid fuzzy program, so it is located in a file with the same name of the original specification, and extension "sse.fpl", thus implementing the notion of "Similarity-based Strict Equality SSE" as a MALP program. The resulting file can be naturally loaded into the $\mathcal{FLOPER}$ tool in order to run and debug goals, depicting evaluation trees, etc.

In order to illustrate this proccess, consider again the specification given in Example 4.1. Once we have the closure of the specification given in Section IV-A (expressed by means of PROLOG facts), the final MALP program has the following form:

```
sse(maria,maria)                    with 1.
sse(maria,mary)                     with 0.8.
sse(mary,maria)                     with 0.8.
sse(mary,mary)                      with 1.
sse(brother(X0),brother(Y0))    <-  sse(X0,Y0)
                                    with 1.
sse(brother(X0),sibling(Y0))    <-  sse(X0,Y0)
                                    with 0.9.
sse(sibling(X0),brother(Y0))    <-  sse(X0,Y0)
                                    with 0.9.
sse(sibling(X0),sibling(Y0))    <-  sse(X0,Y0)
                                    with 1.
```

Figure 7. An on-line session via internet with the SSE application



SSE Tool

Lattice: num | four

```
mary ~ maria = 0.8.
sibling/1 ~ brother/1 = 0.9.
```

godel -> SSE

Similarity relation in Prolog

```
sim((brother, 1), (brother, 1), 1).
sim((brother, 1), (sibling, 1), 0.9).
sim((maria, 0), (maria, 0), 1).
sim((maria, 0), (mary, 0), 0.8).
sim((mary, 0), (maria, 0), 0.8).
sim((mary, 0), (mary, 0), 1).
sim((sibling, 1), (brother, 1), 0.9).
sim((sibling, 1), (sibling, 1), 1).
```

Similarity-based Strict Equality in MALP

```
sse(brother(X0),brother(Y0)) <godel sse(X0,Y0) with 1.
sse(brother(X0),sibling(Y0)) <godel sse(X0,Y0) with 0.9.
sse(maria,maria) with 1.
sse(maria,mary) with 0.8.
sse(mary,maria) with 0.8.
sse(mary,mary) with 1.
sse(sibling(X0),brother(Y0)) <godel sse(X0,Y0) with 0.9.
sse(sibling(X0),sibling(Y0)) <godel sse(X0,Y0) with 1.
```

Moreover, regarding the similarity relation recasted from Example 4.2, we obtain the following set of MALP rules:

```
sse(c,c)                          with top.
sse(c,d)                          with alpha.
sse(d,c)                          with alpha.
sse(d,d)                          with top.
sse(f(X0,X1),f(Y0,Y1)) <- sse(X0,Y0) &
                   sse(X1,Y1) with top.
sse(f(X0,X1),g(Y0,Y1)) <- sse(X0,Y0) &
                   sse(X1,Y1) with beta.
sse(g(X0,X1),f(Y0,Y1)) <- sse(X0,Y0) &
                   sse(X1,Y1) with beta.
sse(g(X0,X1),g(Y0,Y1)) <- sse(X0,Y0) &
                   sse(X1,Y1) with top.
```

In addition to our desktop tool, we have developed too a comfortable on-line version of the application (so it is not necessary to download any file, but only work through the internet) which is located at the web page `dectau.uclm.es/sse`. We provide a link to download the PROLOG-based implementation of the tool but also, and more importantly, this URL enables the possibility of performing on-line work sessions, as illustrated in the screen-shot displayed in Figure 7.

V. CONCLUSIONS AND FUTURE WORK

In this paper we have recasted from [20] a static preprocess for improving the expressive power of a fuzzy declarative language in order to easily cope with similarity relations.

More exactly, we have adapted to the MALP framework our preliminary notion of SSE presented in [17], thus dealing with similarity relations by means of a simple but powerful method (somehow inspired by the -non fuzzy- functional paradigm) which surpass in some cases the effects obtained in other fuzzy languages which are not based on the simpler syntactic unification method of PROLOG. The main goal of this paper focused on proving some important formal properties of our technique for which we have shown some experimental results obtained by using our $\mathcal{FLOPER}$ platform as well as a preliminary PROLOG-based implementation of the technique (please, visit `http://dectau.uclm.es/sse/` for testing it on-line), which is nowadays being introduced inside the core of our system.

REFERENCES

[1] J. Lloyd, *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987, second edition.

[2] H. Nguyen and E. Walker, *A First Course in Fuzzy Logic*. Chapman & Hall/CRC, Boca Ratón, Florida, 2000.

[3] I. Bratko, *Prolog Programming for Artificial Intelligence*. Addison Wesley, 2000.

[4] R. Lee, "Fuzzy Logic and the Resolution Principle," *Journal of the ACM*, vol. 19, no. 1, pp. 119–129, 1972. [Online]. Available: http://doi.acm.org/10.1145/321679.321688

[5] M. Ishizuka and N. Kanai, "Prolog-ELF Incorporating Fuzzy Logic," in *Proceedings of the 9th Int. Joint Conference on Artificial Intelligence, IJCAI'85*, A. K. Joshi, Ed. Morgan Kaufmann, 1985, pp. 701–703. [Online]. Available: http://dl.acm.org/citation.cfm?id=1623611.1623612

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

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

[8] J. Medina, M. Ojeda-Aciego, and P. Vojtáš, "Similarity-based Unification: a multi-adjoint approach," *Fuzzy Sets and Systems*, vol. 146, pp. 43–62, 2004. [Online]. Available: http://dblp.uni-trier.de/db/journals/fss/fss146.html#MedinaOV04

[9] M. Sessa, "Approximate reasoning by similarity-based SLD resolution," *Fuzzy Sets and Systems*, vol. 275, pp. 389–426, 2002. [Online]. Available: http://dx.doi.org/10.1016/S0304-3975(01)00188-8

[10] F. Arcelli and F. Formato, "Likelog: A logic programming language for flexible data retrieval," in *Proc. of the 1999 ACM Symposium on Applied Computing (SAC'99), February 28 - March 2, 1999, San Antonio, USA*. ACM, Artificial Intelligence and Computational Logic, 1999, pp. 260–267. [Online]. Available: http://doi.acm.org/10.1145/298151.298348

[11] R. Caballero, M. Rodríguez-Artalejo, and C. A. Romero-Díaz, "Similarity-based reasoning in qualified logic programming," in *Proceedings of the 10th Int. ACM SIGPLAN conference on Principles and practice of declarative programming*, ser. PPDP'08. New York, USA: ACM, 2008, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1389449.1389472

[12] P. Julián, C. Rubio, and J. Gallardo, "Bousi∼prolog: a prolog extension language for flexible query answering," *Electronic Notes in Theoretical Computer Science*, vol. 248, pp. 131–147, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2009.07.064

[13] C. Rubio-Manzano and P. Julián-Iranzo, "A fuzzy linguistic prolog and its applications," *Journal of Intelligent and Fuzzy Systems*, vol. 26, no. 3, pp. 1503–1516, 2014. [Online]. Available: http://dx.doi.org/10.3233/IFS-130834

[14] M. Bröcheler, L. Mihalkova, and L. Getoor, "Probabilistic similarity logic," *Computing Research Repository*, vol. abs/1203.3469, 2012. [Online]. Available: http://arxiv.org/abs/1203.3469

[15] A. Kimmig, B. Demoen, L. D. Raedt, V. S. Costa, and R. Rocha, "On the implementation of the probabilistic logic programming language problog," *TPLP*, vol. 11, no. 2-3, pp. 235–262, 2011. [Online]. Available: http://dx.doi.org/10.1017/S1471068410000566

[16] G. Moreno and V. Pascual, "A hybrid programming scheme combining fuzzy-logic and functional-logic resources," *Fuzzy Sets and Systems*, vol. 160, pp. 1402–1419, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.fss.2008.11.028

[17] G. Moreno, "Similarity-based equality with lazy evaluation," in *Proc. of the 13th Int. Conference on Information Processing and Management of Uncertainty in Knowledge-based Systems, IPMU'10, June 28-July 2, Dortmund, Germany*, E. Hullermeier, R. Kruse, and F. Hoffmann, Eds. Springer CCIS 80 (Part I), 2010, pp. 108–117.

[18] C. V. Hall, K. Hammond, W. Partain, S. L. P. Jones, and P. Wadler, "The glasgow haskell compiler: A retrospective," in *Functional Programming*, ser. Workshops in Computing, J. Launchbury and P. M. Sansom, Eds. Springer, 1992, pp. 62–71. [Online]. Available: http://dl.acm.org/citation.cfm?id=647557.729914

[19] M. Hanus (ed.), "Curry: An Integrated Functional Logic Language," Available at `http://www.informatik.uni-kiel.de/~mh/curry/`, 2003.

[20] G. Moreno, J. Penabad, and C. Vázquez, "SSE: Similarity-based strict equality for multi-adjoint logic programs," in *Proceedings 12th Int. Conference on Mathematical Methods in Science and Engineering, CMMSE'12. La Manga (Murcia), Spain, July 2-5*, J. Vigo-Aguiar, Ed., vol. III. ISBN: 978-84-615-5392-1, 2012, pp. 876–887.

[21] L. A. Zadeh, "Similarity relations and fuzzy orderings," *Information Sciences*, vol. 3, pp. 177–200, 1971. [Online]. Available: http://dx.doi.org/10.1016/S0020-0255(71)80005-1

[22] P. Morcillo, G. Moreno, J. Penabad, and C. Vázquez, "Dedekind-MacNeille completion and cartesian product of multi-adjoint lattices," *Int. Journal of Computer Mathematics*, vol. 89, no. 13-14, pp. 1742–1752, 2012. [Online]. Available: http://dx.doi.org/10.1080/00207160.2012.689826

[23] P. Morcillo, G. Moreno, J. Penabad, and C. Vázquez, "Declarative Traces into Fuzzy Computed Answers," in *Proc. of 5th Int. Symposium on Rules: Research Based, Industry Focused, RuleML'11. Barcelona, Spain, July 19–21*, N. Bassiliades, G. Governatori, and A. Pasckhe, Eds. Springer Verlag, LNCS 6826, 2011, pp. 170–185. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032787.2032806

[24] J. Almendros-Jiménez, A. Luna, and G. Moreno, "A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming," in *Proc. of 5th Int. Symposium on Rules: Research Based, Industry Focused, RuleML'11. Barcelona, Spain, July 19–21*, N. Bassiliades, G. Governatori, and A. Pasckhe, Eds. Springer Verlag, LNCS 6826, 2011, pp. 186–193. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032787.2032807

[25] G. Moreno and C. Vůzquez, "Fuzzy logic programming in action with floper," *Journal of Software Engineering and Applications*, vol. 7, pp. 237–298, 2014. [Online]. Available: http://dx.doi.org/10.4236/jsea.2014.74028

[26] P. Morcillo, G. Moreno, J. Penabad, and C. Vázquez, "A Practical Management of Fuzzy Truth Degrees using FLOPER," in *Proc. of 4nd Int. Symposium on Rule Interchange and Applications, RuleML'10*, M. D. et al., Ed. Springer Verlag, LNCS 6403, 2010, pp. 20–34. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929574.1929580

[27] ——, "Fuzzy Computed Answers Collecting Proof Information," in *Advances in Computational Intelligence - Proc of the 11th Int. Work-Conference on Artificial Neural Networks, IWANN'11*, J. C. et al., Ed. Springer Verlag, LNCS 6692, 2011, pp. 445–452.

[28] P. Morcillo and G. Moreno, "Programming with Fuzzy Logic Rules by using the FLOPER Tool," in *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, Int. Symposium, RuleML'08*, N. B. et al., Ed. Springer Verlag, LNCS 3521, 2008, pp. 119–126. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88808-6$_1$4

[29] P. Julián, "A procedure for the construction of a similarity relation," in *Proc. of 12th Information Processing and Management of Uncertainty, IPMU'08,June 22-27, Málaga, Spain*, M. Ojeda, Ed. Springer CCIS 80 (Part I), 2008, pp. 489–ń496.