Proceedings of the 12th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2012 La Manga, Spain, July, 2–5, 2012.

SSE: Similarity-based Strict Equality for Multi-Adjoint Logic Programs

Ginés Moreno, Jaime Penabad and Carlos Vázquez[1]

[1] *Faculty of Computer Science Engineering, UCLM, 02071, Albacete (Spain),*

emails: {`Gines.Moreno,Jaime.Penabad`}`@uclm.es`, `Carlos.Vazquez@alu.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 clever 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 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 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.

Key words: Equality, Similarity, Multi-adjoint Logic Programming.

1 Introduction

Logic Programming [13] 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 [23]) 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 [4] (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 [11], such as Elf-Prolog [8], Fril [3], F-Prolog [12] and MALP [14]. There exists also a family of fuzzy languages based on sophisticated unification methods [24] to cope with similarity/proximity relations, as occurs with Likelog [2], SQLP [5] and Bousi∼Prolog [9].

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 [21]. In [20], 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 capital 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 $g(b)$.

- **Strict equality.** When considering lazy languages, both pure functional (Haskell [6]) and integrated functional-logic (Curry [7]) 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 2, 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.

In Section 3, we recast from [20] 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 clever 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 ≈:≈ 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 4 we will see that SSE admits a much more natural definitions 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).

2 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) [25]. 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 [24], 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 [21], we assume here that the intended similarity relation $\Re$ associated to a given program $\mathcal{R}$, is induced from the (safe) set of similarity equations of $\mathcal{R}$, 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 [2] and Bousi∼Prolog [9], 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* [9]. 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, apart for introducing risks of infinite loops when treated computationally, in particular, it does not satisfy the symmetric property.

Example 2.1. *Following [2], if we consider a database of books of different kinds containing the fact "*`horror(drakula)`*", then the goal "*`?-adventurous(Book)`*" would not have classical solution in the case that there were no rule in the database unifying with atom "*`adventurous(X)`*". Nevertheless, it seems reasonable that the user considers the constants "*`adventurous`*" and "*`horror`*"* similar *to a certain degree. More precisely, if the user introduces a similarity equation like "*`eq(adventurous, horror) = 0.9`*" into a* Likelog *or* Bousi∼Prolog *interpreter, the system would successfully respond with a computed answer incorporating the corresponding truth degree "*`0.9`*" (i.e, something like the 90 % of credibility) to substitution "*`Book` $\mapsto$ `drakula`*", as obviously expected.*

3 Rewriting Rules, Strict Equality and Similarity

The theory of Term Rewriting Systems (TRS) has been largely used in declarative programming to develop pure functional and integrated (functional-logic) languages, such as Haskell and Curry, respectively. A Haskell or a Curry program is no more than a TRS, that is, a set of rewrite rules (instead of a set of clauses, as occurs with logic languages) that can not be distinguished under a syntactic point of view: the differences appear only at the operational level, depending whether rewriting or narrowing is used to execute programs. In what follows we give a short summary explaining such concepts [10].

We consider a *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of *defined* functions. The set of *constructor terms* with *variables*[1] is obtained by using symbols from $\mathcal{C}$ and a set of variables $\mathcal{X}$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A rewrite rule is an expression of the form $l \to r$ such that $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The terms l and r are called the *left-hand side* (lhs) and the *right-hand side* (rhs) of the rule, respectively. A set of rewrite rules is called a *term rewriting system* (TRS). In this setting, the evaluation of a complex expression E (possibly containing defined function symbols from $\mathcal{F}$) devoted to obtain a final data term, is performed by applying rewriting/narrowing computations where, in essence, each step replaces a subterm of E which matches/unifies with the lhs of a rewriting rule, by the corresponding instance of its rhs. In lazy computations, only steps exploiting outermost subterms (when there exists several chances on E) are considered.

On the other hand, it is usual in functional logic programming to simulate typical (crisp) predicates of pure logic programming by means of boolean functions. However, a second much more interesting way to face this problem is by using *constraints*. An elementary constraint is an *equational constraint* $\mathtt{e}_1 =:= \mathtt{e}_2$ between two expressions (of base type). Then, $\mathtt{e}_1 =:= \mathtt{e}_2$ is satisfied if both sides are reducible to a same ground data term. This notion of equality, which is the only sensible notion of equality in the presence of non-terminating functions [22] and also used in (lazy) functional languages, it is also called *strict equality*. As a consequence, if one side is undefined (non-terminating), then the strict equality does not hold (so, it is not reflexive). Operationally, an equational constraint $\mathtt{e}_1 =:= \mathtt{e}_2$ is solved by evaluating $\mathtt{e}_1$ and $\mathtt{e}_2$ to unifiable data terms. Constraints can be also combined into a *conjunction* (which can be interpreted concurrently), written as $\mathtt{c}_1\&\mathtt{c}_2$. This evaluation mechanism can be implemented at a very high abstraction level by assuming that each program implicitly incorporates the standard set of rewrite rules shown in Figure 1, defining the semantics of the primitive "strict equality" relation symbols "=:=" and "&" [7, 22].

Since =:= represents a natural way to deal with strict equality and constraints simulating "crisp predicates", our next task consists of introducing a new operator, say $\approx:\approx$, for modeling "fuzzy predicates" by means of the new notions of *similar equality* and *f-constraints*. Given an f-constraint $\mathtt{e}_1{\approx:\approx}\mathtt{e}_2$, the goal now is to reduce both expressions $\mathtt{e}_1$ and $\mathtt{e}_2$ to ground values, and then comparing the resulting data terms $\mathtt{v}_1$ and $\mathtt{v}_2$, having into account the similarity relation $\Re$ induced by the set of the similarity equations of the corresponding program as shown in (see Figure 1). Now, instead of `success`, we are looking for a real number in the interval $[0, 1]$ representing the similarity degree between outputs $\mathtt{v}_1$ and $\mathtt{v}_2$. Basically, the set of rewrite rules defining "$\approx:\approx$" in Figure 1 proceeds as follows. The similarity degree between two constructor symbols of arity 0 is the one returned by

[1]Also called data terms, or directly terms, in the terminology of both pure and fuzzy logic programming languages (Prolog, MALP, Likelog, Bousi∼Prolog, etc).

the induced similarity relation $\Re$. On the other hand, when comparing two data terms (obtained after reducing the original parameters of a f-constraint) with arguments, it is necessary to recursively compute the similarity degree between the corresponding pairs of arguments of the data terms, together with the similarity relation between the constructor symbols heading each data term.

4 SSE **for/with Multi-Adjoint Logic Programming**

In this section we firstly summarize the main features of the MALP language[2], next we introduce the "*Fuzzy LOgic Programming Environment for Research*", $\mathcal{FLOPER}$ in brief, developed in our research group (see [16, 19] and visit `http://dectau.uclm.es/floper/`) and finally, we detail our new MALP-based model of SSE according Figure 1.

% *Rewriting rules modeling classical "Strict Equality"*

$c{=:=}c \rightarrow \texttt{success}$ $\qquad \forall c/0 \in \mathcal{C}$

$c(x_1,..,x_n){=:=}c(y_1,..,y_n) \rightarrow x_1{=:=}y_1 \,\&\ldots\&\, x_n{=:=}y_n$ $\qquad \forall c/n \in \mathcal{C}$

$\texttt{success} \ \& \ \texttt{success} \rightarrow \texttt{success}$

% *SSE modeled with rewriting rules*

$c{\approx:\approx}d \rightarrow \Re(c,d)$ $\qquad \forall c/0, d/0 \in \mathcal{C}$

$c(x_1,..,x_n){\approx:\approx}d(y_1,..,y_n) \rightarrow \texttt{min}(\Re(c,d), x_1{\approx:\approx}y_1, \ldots$
$\ldots x_n{\approx:\approx}y_n)$ $\qquad \forall c/n, d/n \in \mathcal{C}$

% *SSE modeled with MALP rules*

$\texttt{sse}(c,d)$ $\qquad$ *with* $\Re(c,d)$

$\texttt{sse}(c(x_1,..,x_n),d(y_1,..,y_n)) \ \leftarrow_G \texttt{sse}(x_1,y_1) \ \&_G \ldots$
$\ldots \&_G \ \texttt{sse}(x_n,y_n)$ $\qquad$ *with* $\Re(c,d)$

Figure 1: Rules defining "Strict Equality" and "Similarity-based Strict Equality"

4.1 MALP

We work with a first order language, $\mathcal{L}$, 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

[2]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 [18, 17, 1].

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 $\mathcal{L}$ 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 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 α", 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 general aggregations, and finally $\alpha \in L$ is the "weight" or *truth degree* of the rule. 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 "$\&_{\mathtt{G}}(x, y) = min(x, y)$"):

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

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 $\mathcal{Q}$, $\langle A' \leftarrow_i \mathcal{B}; v \rangle$ 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 $\langle A' \leftarrow_i; v \rangle$ 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):

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

The final formula can be directly interpreted in the lattice L to obtain the *fuzzy computed answer* or *f.c.a.*, in brief. So, since $0.8 \mathbin{\&_{\mathtt{P}}} (0.9 \mathbin{\&_{\mathtt{G}}} 0.7) = 0.8{*}\min(0.9, 0.7) = 0.56$, we say that goal $p(X)$ is true at a 56 % when X is a.

4.2 $\mathcal{FLOPER}$

As detailed in [15, 16], 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 clause in our target program is translated into: `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 [`Truth_degree = 0.56, X = 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.

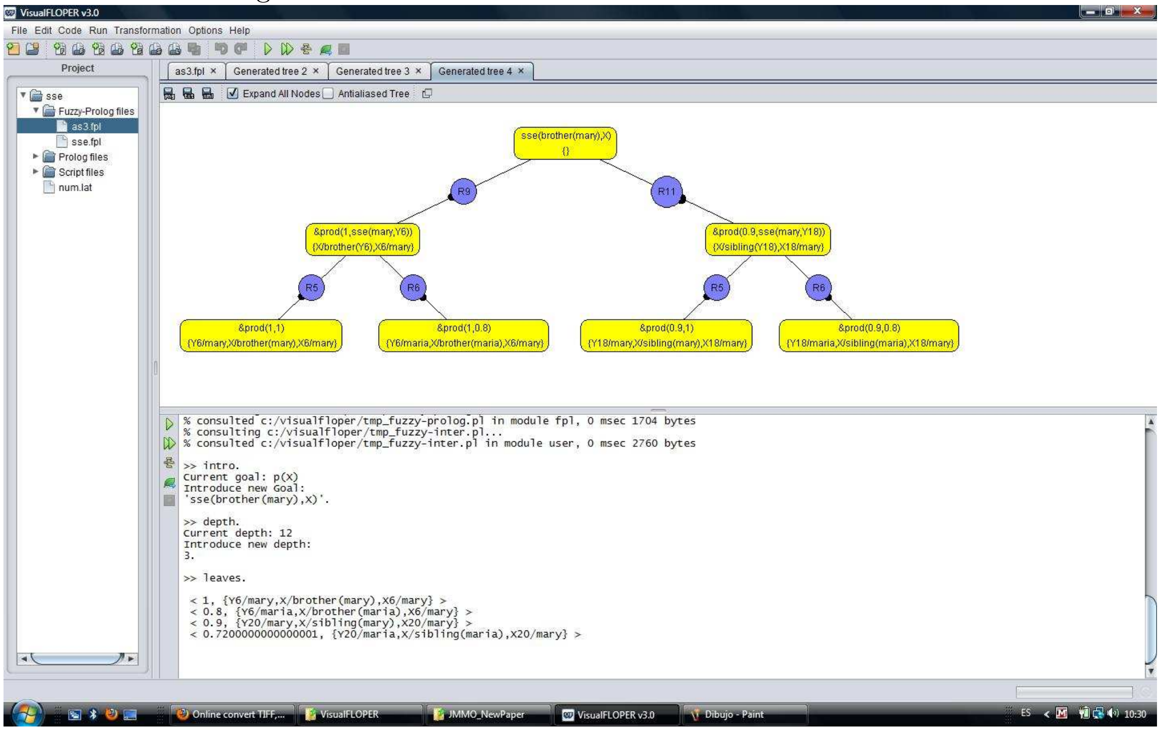
Moreover, it is also possible to select into the FLOPER's goal menu, options "`tree`" and "`depth`", which are useful for tracing execution trees and fixing the maximum length allowed for their branches (initially 3), respectively. From `http://dectau.uclm.es/floper/`

you can download our last version of the $\mathcal{FLOPER}$ tool, which incorporates a graphical interface as shown in Figure 2.

4.3 SSE, MALP and $\mathcal{FLOPER}$

Assume that we plan to compare data terms by using constants "mary" and"maria", which have a similarity degree of 80% and function symbols (with arity one) "brother" and "sibling" with are similars at 90%. According to our MALP-based definition of SSE seen in Figure 1, we could generate a set of MALP rules using the "min' operator (based on Goguel's logic) to propagate similarity degrees. Instead of it, in the following MALP program loaded into $\mathcal{FLOPER}$ we have used a version inspired on "product logic':

Figure 2: Screen-shot of a work session with FLOPER.



```
% consulted c:/visualfloper/tmp_fuzzy-prolog.pl in module fpl, 0 msec 1704 bytes
% consulting c:/visualfloper/tmp_fuzzy-inter.pl...
% consulted c:/visualfloper/tmp_fuzzy-inter.pl in module user, 0 msec 2760 bytes

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

>> depth.
Current depth: 12
Introduce new depth:
3.

>> leaves.

 < 1, {Y6/mary,X/brother(mary),X6/mary} >
 < 0.8, {Y6/maria,X/brother(maria),X6/mary} >
 < 0.9, {Y20/mary,X/sibling(mary),X20/mary} >
 < 0.7200000000000001, {Y20/maria,X/sibling(maria),X20/mary} >
```

```
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 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)`", whereas our system is able to provide the corresponding four answer for the first query shown in Figure 2, as well as infinite solutions for the second goal which neccesarily must to include: $[\texttt{Truth\_degree} = 0.72, \texttt{X} = \texttt{brother}(\texttt{mary}), \texttt{Y} = \texttt{sibling}(\texttt{maria})]$.

5 Conclusions and Future Work

In this paper we have adapted to the MALP framework our preliminary notion of SSE presented in [20], in order to cope with similarity relations which surpass in some cases the effects obtained in other fuzzy languages which are not based on the simpler syntactic unification method of Prolog. We have shown an experimental result obtained by using our $\mathcal{FLOPER}$ platform. Due to its interesting behaviour, we are nowadays automating this technique inside the core of such system.

References

[1] J.M. Almendros-Jiménez, A. Luna, and G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In N. Bassiliades, G. Governatori, and A. Pasckhe, editors, *Proc. of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11. Barcelona, Spain, July 19–21*, pages 186–193. Springer Verlag, LNCS 6826, 2011.

[2] 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, Texas, USA*, pages 260–267. ACM, Artificial Intelligence and Computational Logic, 1999.

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

[4] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison Wesley, September 2000.

[5] R. Caballero, M. Rodríguez-Artalejo, and C. A. Romero-Díaz. Similarity-based reasoning in qualified logic programming. In *Proceedings of the 10th international ACM*

SIGPLAN conference on Principles and practice of declarative programming, PPDP '08, pages 185–194, New York, USA, 2008. ACM.

[6] Cordelia V. Hall, Kevin Hammond, Will Partain, Simon L. Peyton Jones, and Philip Wadler. The glasgow haskell compiler: A retrospective. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 62–71. Springer, 1992.

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

[8] M. Ishizuka and N. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In Aravind K. Joshi, editor, *Proceedings of the 9th Int. Joint Conference on Artificial Intelligence, IJCAI'85*, pages 701–703. Morgan Kaufmann, 1985.

[9] P. Julián, C. Rubio, and J. Gallardo. Bousi∼prolog: a prolog extension language for flexible query answering. *Electronic Notes in Theoretical Computer Science*, 248:131–147, 2009.

[10] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[11] R.C.T. Lee. Fuzzy Logic and the Resolution Principle. *Journal of the ACM*, 19(1):119–129, 1972.

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

[13] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.

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

[15] P.J. Morcillo and G. Moreno. Programming with Fuzzy Logic Rules by using the FLOPER Tool. In Nick Bassiliades et al., editor, *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML'08*, pages 119–126. Springer Verlag, LNCS 3521, 2008.

[16] P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In M. Dean et al., editor, *Proc. of 4nd Intl Symposium on Rule Interchange and Applications, RuleML'10*, pages 20–34. Springer Verlag, LNCS 6403, 2010.

[17] P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. Declarative Traces into Fuzzy Computed Answers. In N. Bassiliades, G. Governatori, and A. Pasckhe, editors, *Proc. of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11. Barcelona, Spain, July 19–21*, pages 170–185. Springer Verlag, LNCS 6826, 2011.

[18] P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. Dedekind-Macneille Completion and Multi-Adjoint Lattices. In J. Vigo-Aguiar, editor, *Proc. 11th International Conference on Mathematical Methods in Science and Engineering (special session on 'Mathematical Methods for Computer Science'), CMMSE'11. Benidorm, Spain, June 26–30*, volume 2, pages 846–857 (en proceso de revisión una versión extendida sometida a 'International Journal of Computer Mathematics', de Taylor&Francis). Actas del congreso con ISBN 978-84-614-6167-7, 2011.

[19] P.J. Morcillo, G. Moreno, J. Penabad, and C. Vázquez. Fuzzy Computed Answers Collecting Proof Information. In J. Cabestany et al., editor, *Advances in Computational Intelligence - Proc of the 11th International Work-Conference on Artificial Neural Networks, IWANN 2011*, pages 445–452. Springer Verlag, LNCS 6692, 2011.

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

[21] G. Moreno and V. Pascual. A hybrid programming scheme combining fuzzy-logic and functional-logic resources. *Fuzzy Sets and Systems*, 160:1402–1419, 2009. doi:10.1016/j.fss.2008.11.028.

[22] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.

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

[24] M.I. Sessa. Approximate reasoning by similarity-based SLD resolution. *Fuzzy Sets and Systems*, 275:389–426, 2002.

[25] L. A. Zadeh. Similarity relations and fuzzy orderings. *Informa. Sci.*, 3:177–200, 1971.