Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 188 (2007) 21–35
www.elsevier.com/locate/entcs

Formal Properties of Needed Narrowing with Similarity Relations[1]

Ginés Moreno and Vicente Pascual[2]

Dep. of Computing Systems, U. of Castilla–La Mancha, 02071 Albacete, Spain

Abstract

The integration of declarative paradigms such as functional-logic and fuzzy-logic programming seems to be an interesting challenge in the design of highly expressive declarative languages where mathematical functions cohabit with fuzzy logic features. Starting with two representative languages from both settings, namely Curry and Likelog, we have recently proposed an hybrid dialect where a set of (Curry) rewrite rules together with a set of (Likelog) similarity equations can be safely executed by means of a fuzzy variant of needed narrowing. This paper is devoted to show some important properties enjoyed by the new narrowing strategy. Firstly, we prove the termination of the process which determines the set of tuples that enable new narrowing steps for a given term. Termination in the fuzzy context is not trivial since, apart that it is required to compute the transitive closure of the initial set of similarity equations, it is also mandatory to ensure the finiteness of the set of recursive calls performed by the narrowing strategy, as well as that each call never falls into an infinite loop. From here, we prove two important (somehow complementary) properties, that we call crispness and fuzziness, respectively. The first one implies that the new strategy is (at least) conservative with respect to the original one of needed narrowing, in the sense that each tuple obtained in the crisp case, is also replicated (with the maximum truth degree) in the fuzzy case. On the other hand, fuzziness means the maximality of the new strategy when exploiting (as much as possible) the similarity relations collected in a given program.

Keywords: Functional Logic Prog. Needed Narrowing, Similarity

1 Introduction

Fuzzy Logic Programming amalgamates fuzzy logic [8] and pure logic programming [6], in order to provide these pure logic languages with techniques or constructs to deal with uncertainty and approximated reasoning. Although there is no common method for introducing fuzzy concepts into logic programming, in this paper we are specially interested in the promising approach presented in [3,9], which basically consists in combining *similarity relations* with classical Horn clauses. Similarity relation is a mathematical notion strictly related with equivalence relations and closure operators, that provides a way to manage alternative instances of an entity

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

[2] Emails: {gmoreno,vpascual}@dsi.uclm.es

doi:10.1016/j.entcs.2007.05.036

that can be considered "equal" with a given degree [10]. 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 in modeling them by a set of the so-called *similarity equations* of the form $eq(s1, s2) = \alpha$, whith the intended meaning that $s1$ and $s2$ are predicate/function symbols of the same arity with a similarity degree α. This approach is followed, for instance, in the fuzzy logic language Likelog [3], where a set of usual Prolog clauses are accompanied by a set of similarity equations which play an important role at (fuzzy) unification time. 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 required for similarity relations 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,does not verify the symmetric property. It is important to note, that since Likelog is oriented to manipulate inductive databases, where no function symbols of arity greater than 0 are allowed, then, similarity equations only consider similarities between two predicates or two constants of the same arity. In this paper, we drop out this last limitation by also allowing similarity equations between any pair of (both defined or both constructor) function symbols with the same arity, which do not necessarily be constants. Moreover, we wish to remark again that, similarly to Likelog, no similarity equations are allowed between two symbols with different arity and/or nature (i.e, a constructor with a defined function symbol and viceversa).

Let us recall that a T-norm $\wedge$ in $[0, 1]$ is a binary operation $\wedge : [0, 1] \times [0, 1] \rightarrow [0, 1]$ associative, commutative, non-decreasing in both arguments, and with the identity symbol 1. In order to simplify our developments, and similarly to other approaches in fuzzy logic programming [9], in the sequel, we assume that $x \wedge y$ is equivalent to $min(x, y)$, that is, the minimum between two elements $x, y \in [0, 1]$, whereas $x \vee y$ and $max(x, y)$ refers to the maximun operator. Both notations have a long presence in the specialized literature and help us to clarify expressions in recursive calculations. A *similarity relation* $\Re$ on a domain $\mathcal{U}$ is a fuzzy subset $\Re : \mathcal{U} \times \mathcal{U} \rightarrow [0, 1]$ 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 $\Re(x, z) \geq \Re(x, y) \wedge \Re(y, z)$. In the following, we assume 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 $s_1 \equiv s_2$ or, otherwise, it is recursively defined as the transitive closure of the equational set defined as: $T_r^t(R)^{\bigcup}_{f=1\ldots\infty} R^f$ where $R^{f+1} = R^f \circ^t R$, for a given T-norm t. Moreover, it can be demonstrated that, if the domain is a finite set with n elements, then only n-1 powers must be calculated [4]. Finally, by simply assuming that the set of similarity equations in $\mathcal{R}$ is trivially extended by reflexivity, then $\Re = T_r(R) = R^{(n-1)}$ [4]. For instance, in the following pair of matrix, we are considering similarities between four arbitrary constant symbols. The second matrix

has been obtained from the first one after applying the algorithm described in [4].

$$\begin{pmatrix} 1 & .7 & .6 & .4 \\ .7 & 1 & .8 & .9 \\ .6 & .8 & 1 & .7 \\ .4 & .9 & .7 & 1 \end{pmatrix} \xrightarrow[\text{Similarity}]{R} \begin{pmatrix} 1 & .7 & .7 & .7 \\ .7 & 1 & .8 & .9 \\ .7 & .8 & 1 & .8 \\ .7 & .9 & .8 & 1 \end{pmatrix}$$

In what follows, we propose the combined use of similarity equations together with rewrite rules (instead of Horn clauses) typically used in languages (with a functional taste) such as Haskell or Curry. In this sense, it is important to note that, although there exists some precedents for introducing fuzzy logic into logic programming, to the best of our knowledge, our approach represents the first attempt for fuzzifying (integrated) functional-logic languages. We consider a *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of *defined* functions (also called *operations*). The set of *constructor terms* (with *variables*) 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 $\mathcal{V}ar(t)$. We write $\overline{o_n}$ for the *list* of objects $o_1, \ldots, o_n$. A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n$ are constructor terms (with variables). A term is *linear* if it does not contain multiple occurrences of one variable. A *position* p in a term t is represented by a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). The set of positions of a term t is denoted by $\mathcal{P}os(t)$. Positions are ordered by the *prefix* ordering: $p \leq q$, if $\exists w$ such that $p.w = q$. $t|_p$ and $t\uparrow_p$ denote the *subterm* of t at a given position p, and the symbol rooting such subterm, respectively. $t[s]_p$ represents the result of *replacing the subterm* $t|_p$ by the term s. We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ whose application to a term t is denoted by $\sigma(t)$. A set of rewrite rules $l \to r$ such that $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (lhs) and the *right-hand side* (rhs) of the rule, respectively. A TRS $\mathcal{R}$ is left-linear if l is linear for all $l \to r \in \mathcal{R}$. A TRS is constructor–based (CB) if each left-hand side is a pattern. In the remainder of this paper, a functional logic *program* is a left-linear CB-TRS without overlapping rules (i.e. the lhs's of two different program rules do not unify). A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to^{p,R} s$ if there exists a position p in t, a rewrite rule $R = (l \to r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. *Narrowing* is a combination of variable instantiation and reduction. Formally, $t \leadsto_\sigma^{p,R} s$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \to^{p,R} s$.

2 The Original Needed Narrowing Strategy

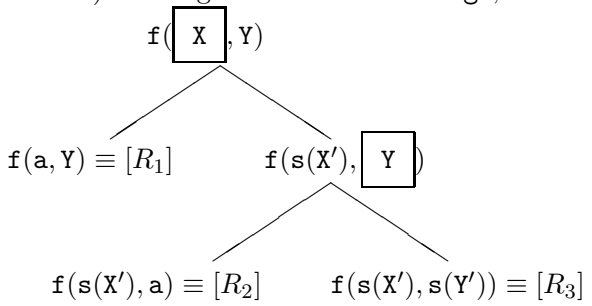
Functional logic programming languages combine the operational principles of the most important declarative programming paradigms, namely functional and logic programming. The operational semantics of such languages is usually based on narrowing, where efficient demand-driven functional computations are amalgamated

with the flexible use of logical variables providing for function inversion and search for solutions. A challenge in the design of functional logic languages is the definition of a "good" narrowing strategy, i.e., a restriction on the narrowing steps issuing from a term without losing completeness.

Needed narrowing [2] is currently one of the best known narrowing strategies due to its optimality properties w.r.t. the length of the derivations and the number of computed solutions. It extends Huet and Lévy's notion of a needed reduction [5]. The definition of needed narrowing uses the notion of a *definitional tree* [1], which refines the standard matching trees of functional programming. However, differently from left-to-right matching trees used in either Hope, Miranda, or Haskell, definitional trees deal with *dependencies* between arguments of functional patterns.

Roughly speaking, a definitional tree for a function symbol f is a tree whose leaves contain all (and only) the rules used to define f and whose inner nodes contain information to guide the (optimal) pattern matching during the evaluation of expressions. Each inner node contains a pattern and a variable position in this pattern (the *inductive position*) which is further refined in the patterns of its immediate children by using different constructor symbols. The pattern of the root node is simply $f(\overline{x_n})$, where $\overline{x_n}$ are different variables.

Example 2.1 It is often convenient and simplifies the understanding to provide a graphic representation of definitional trees, where each node is marked with a pattern, the inductive position in branches is surrounded by a box, and the leaves contain the corresponding rules. For instance, given the following program (right column) defining functions "f" and "g", the definitional tree for f is:



$R_1 : \mathtt{f(a,N)} \to \mathtt{a}$
$R_2 : \mathtt{f(s(M),a)} \to \mathtt{b}$
$R_3 : \mathtt{f(s(M),s(N))} \to \mathtt{f(M,N)}$

$R_4 : \mathtt{g(a,N)} \to \mathtt{N}$
$R_5 : \mathtt{g(s(M),N)} \to \mathtt{s(g(M,N))}$

For the definition of needed narrowing, we assume that $t \equiv f(\overline{s_n})$ is an operation-rooted term and $\mathcal{P}_f$ is a definitional tree for f with root π such that $\pi \leq t$. Hence, when π is a leaf, i.e., $\mathcal{P}_f = \{\pi\}$, we have that $R : \pi \to r$ is a variant of a rewrite rule. On the other hand, if π is a branch, we consider the inductive position o of π and we say that the pattern $\pi_i \equiv \pi[c_i(\overline{x_n})]_o \in \mathcal{P}_f$, is a child of π in $\mathcal{P}_f$. Moreover, the definitional (sub-)tree of $\mathcal{P}_f$ rooted with π_i, (i.e., where all patterns are instances of π_i) is denoted by $\mathcal{P}_f^{\pi_i} = \{\pi' \in \mathcal{P}_f \mid \pi_i \leq \pi'\}$. We define now a function λ_{crisp} from terms to sets of tuples (position, rule, substitution) which uses an auxiliary function λ_c for explicitly referring to the appropriate definitional tree in each case. Then, $\lambda_{crisp}(t) = \lambda_c(t, \mathcal{P}_f)$ returns the least set satisfying:

LR (LEAF-RULE) CASE: $\lambda_c(t, \mathcal{P}_f) = \{(\Lambda, R, id)\}$

`BV (BRANCH-VAR) CASE:` If $t|_o = x \in \mathcal{X}$, then $\lambda_c(t, \mathcal{P}_f) = \{(p, R, \sigma \circ \tau) \mid \pi_i \equiv \pi[c_i(\overline{x_n})]_o \in \mathcal{P}_f$ and $\tau = \{x \mapsto c_i(\overline{x_n})\}$ and $(p, R, \sigma) \in \lambda_c(\tau(t), \mathcal{P}_f^{\pi_i})\}$

`BC (BRANCH-CONS) CASE:` If $t|_o = c_i(\overline{t_n})$, where $c_i \in \mathcal{C}$, then $\lambda_c(t, \mathcal{P}_f) = \{(p, R, \sigma \circ id) \mid \pi_i \equiv \pi[c_i(\overline{x_n})]_o \in \mathcal{P}_f$ and $(p, R, \sigma) \in \lambda_c(t, \mathcal{P}_f^{\pi_i})\}$

`BF (BRANCH-FUNC) CASE:` If $t|_o = g(\overline{t_n})$, where $g \in \mathcal{F}$, then $\lambda_c(t, \mathcal{P}_f) = \{(o.p, R, \sigma \circ id) \mid (p, R, \sigma) \in \lambda_c(t|_o, \mathcal{P}_g)\}$

When none of the previous cases is satisfied, we assume that function λ_c returns $\emptyset$. Informally speaking, needed narrowing directly applies a rule if the term is an instance of some left-hand side (`LR` case), or checks the subterm corresponding to the inductive position of the branch: if it is a variable (`BV` case), it is instantiated to the constructor of each one of the children; if it is already a constructor (`BC` case), we proceed with the corresponding child; if it is a function (`BF` case), we evaluate it by recursively applying needed narrowing. Thus, the strategy differs from lazy functional languages only in the instantiation of free variables. In contrast to more traditional narrowing strategies, needed narrowing does not compute *most general* unifiers. In each recursive step during the computation of λ_c, we compose the current substitution with the local substitution of this step (which can be the identity *id*). As in proof procedures for logic programming, we assume that definitional trees always contain new variables if they are used in a narrowing step. Moreover, as introduced in Section 1, it is important to remember that, if σ is a substitution appearing in a tuple $(p, R, \sigma) \in \lambda_{crisp}(t)$, then σ only contains bindings for variables of t. So, the application of σ to t, enables the subsequent rewriting step at position p with rule R, that is, $\sigma(t) \rightarrow^{p,R} s$, which completes the corresponding needed narrowing step. Then, $t \leadsto_\sigma^{p,R} s$ is a *needed narrowing step* for all $(p, R, \sigma) \in \lambda_{crisp}(t)$.

Example 2.2 For instance, if we consider again the rules for `f` and `g` in Example 2.1 then we have $\lambda_{crisp}(\mathtt{f(X, g(X, X))}) = \{(\Lambda, R_1, \{\mathtt{X} \mapsto \mathtt{a}\}), (2, R_5, \{\mathtt{X} \mapsto \mathtt{s(X')}\})\}$ which enables the following pair of needed narrowing steps: $\underline{\mathtt{f(X, g(X, X))}} \leadsto^{\Lambda, R_1}_{\{\mathtt{X} \mapsto \mathtt{a}\}} \mathtt{a}$, and $\mathtt{f(X, }\underline{\mathtt{g(X, X)}}\mathtt{)} \leadsto^{2, R_5}_{\{\mathtt{X} \mapsto \mathtt{s(X')}\}} \mathtt{f(s(X'), s(g(X', s(X'))))}$.

3 Needed Narrowing with Similarity Relations

This section recalls from [7] the basis of our attempt for fuzzifying the needed narrowing strategy. For the definition of needed narrowing with similarity relations, we extend the notion of computed answer for also reporting now (apart for the classical components of substitution and value), a real number in the interval $[0, 1]$ indicating the similarity degree computed along the corresponding derivation. Hence, we define function λ_{fuzzy} from terms and definitional trees to sets of tuples (position, rule, substitution, similarity_degree). If $t \equiv f(\overline{s_n})$ is the operation-rooted term we consider in the initial call to λ_{fuzzy}, we must guarantee that any term (including t itself), rooted with a symbol similar to f be will be treated. So, $\lambda_{fuzzy}(t)\{(p, R, \sigma, min(\alpha, \beta)) \mid \Re(f, g) = \alpha$ and $(p, R, \sigma, \beta) \in \lambda_f(g(\overline{s_n}), \mathcal{P}_g\}$, where function λ_f is defined as follows:

LR (LEAF-RULE) CASE: $\lambda_f(t, \mathcal{P}_f) = \{(\Lambda, \pi \to r, id, 1)\}$

BV (BRANCH-VAR) CASE: if $t|_o = x \in \mathcal{X}$, then $\lambda_f(t, \mathcal{P}_f) = \{(p, R, \sigma \circ \tau, \alpha) \mid \pi_i \equiv \pi[c_i(\overline{x_n})]_o \in \mathcal{P}_f \ , \quad \tau = \{x \mapsto c_i(\overline{x_n})\}$ and $(p, R, \sigma, \alpha) \in \lambda_f(\tau(t), \mathcal{P}_f^{\pi_i})\}$

BC (BRANCH-CONS) CASE: if $t|_o = d(\overline{t_n})$, where $d \in \mathcal{C}$, then $\lambda_f(t, \mathcal{P}_f) = \{(p, R, \sigma, min(\alpha, \beta)) \mid \Re(d/n, c_i/n) = \alpha > 0$ and $\pi_i \equiv \pi[c_i(\overline{x_n})]_o \in \mathcal{P}_f$ and $(p, R, \sigma, \beta) \in \lambda_f(t[c_i(\overline{t_n})]_o, \mathcal{P}_f^{\pi_i})\}$

BF (BRANCH-FUNC) CASE: if $t|_o = g(\overline{t_n})$, where $g \in \mathcal{F}$, then $\lambda(t, \mathcal{P}_f) = \{(o.p, R, \sigma, min(\alpha, \beta)) \mid \Re(g/n, h/n) = \alpha > 0$ and $(p, R, \sigma, \beta) \in \lambda(h(\overline{t_n}), \mathcal{P}_h)\}$

As we can see, LR and BV cases are very close to the corresponding ones analyzed in Section 2, but propagating now the corresponding similarity degrees. Moreover, closely related to the initial call to λ_{fuzzy} seen before, the last case (BF) performs recursive calls to λ_f for evaluating the operation–rooted subterm at the considered inductive position, as well as each other (almost identical) subterms rooted with defined function symbols similars to g. Something almost identical occurs with the BC case, but the intention now is to treat all subterms with constructor symbols similars to d at the inductive position.

Example 3.1 Consider again the same program of Example 2.1 augmented with the new rule R_6 : $\mathtt{h(r(X), Y)} \to \mathtt{r(Y)}$ together with the similarity equations S_1 : $\mathtt{eq(g, h)} = \mathtt{0.7}$ and S_2 : $\mathtt{eq(s, r)} = \mathtt{0.5}$. Then, $\lambda_{fuzzy}(\mathtt{f(X, g(X, X))}) = \lambda_f(\mathtt{f(X, g(X, X))}, \mathcal{P}_\mathtt{f}) = \{(\Lambda, R_1, \{\mathtt{X} \mapsto \mathtt{a}\}, 1), (2, R_5, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, 1), (2, R_6, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, min(0.7, 0.5))\}$ = [see BV1] ∪ [see BV2].

BV1. The first alternative in this BV case, consists in generating the binding $\tau_1 = \{\mathtt{X} \mapsto \mathtt{a}\}$ and then computing $\lambda_\mathtt{f}(\tau_1(\mathtt{f(X, g(X, X))}), \mathcal{P}_\mathtt{f}^{\mathtt{f(a,Y)}}) = \lambda_\mathtt{f}(\mathtt{f(a, g(a, a))}, \mathcal{P}_\mathtt{f}^{\mathtt{f(a,Y)}})$. Since this last call represents a LR case, it returns $\{(\Lambda, R_1, id, 1)\}$. Then, after applying the binding τ_1 to the third element of this last tuple, the returned set for this case is $\{(\Lambda, R_1, \{\mathtt{X} \mapsto \mathtt{a}\}, 1)\}$.

BV2. After generating the second binding $\tau_2 = \{\mathtt{X} \mapsto \mathtt{s(X')}\}$, we must compute $\lambda_\mathtt{f}(\tau_2(\mathtt{f(X, g(X, X))}),\ \mathcal{P}_\mathtt{f}^{\mathtt{f(s(X'),Y)}}) = \lambda_\mathtt{f}(\mathtt{f(s(X'), g(s(X'), s(X')))},\ \mathcal{P}_\mathtt{f}^{\mathtt{f(s(X'),Y)}}) = \{(2, R_5, id, 1), (2, R_6, id, min(0.7, 0.5))\}$= [see BF1 below]. Now, we simply need to apply τ_2 to the last component of the tuples obtained in BF1, hence returning $\{(2, R_5, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, 1), (2, R_6, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, min(0.7, 0.5))\}$.

BF1. In this BF case, where the considered inductive position is 2, we perform the following two recursive calls (observe that the second one exploits the similarity equation S_1 and it would not be performed in the crisp case): $\lambda_f(\mathtt{g(s(X'), s(X'))},\ \mathcal{P}_\mathtt{g}) \cup \lambda_f(\mathtt{h(s(X'), s(X'))},\ \mathcal{P}_\mathtt{h})$ = [see BC1] ∪ [see BC2] $\{(\Lambda, R_5, id, 1), (\Lambda, R_6, id, 0.5)\}$. And then, since obviously position $\Lambda.2$ coincides directly with position 2, and the similarity between g and h is 0.7, the set of tuples returned in this case is $\{(2, R_5, id, 1), (2, R_6, id, min(0.7, 0.5))\}$.

BC1. This BC case, immediately evolves to the following LR case: $\lambda_f(\mathtt{g(s(X'), s(X'))},\ \mathcal{P}_\mathtt{g}^{\mathtt{g(s(M),N)}}) = \{(\Lambda, R_5, id, 1)\}$. Now, since $\Re(\mathtt{s}, \mathtt{s}) = 1$, and $min(1, 1) = 1$, the returned tuple in this case is $(\Lambda, R_5, id, 1)$ itself.

BC2. By exploiting the similarity equation $S_2 : \mathtt{eq}(\mathtt{s},\mathtt{r}) = 0.5$, this BC case also computes the LR case $\lambda_f(\mathtt{h}(\mathtt{r}(\mathtt{X}'),\mathtt{s}(\mathtt{X}')),\ \mathcal{P}_\mathtt{h}^{\mathtt{h}(\mathtt{r}(\mathtt{X}),\mathtt{Y})}) = \{(\Lambda, R_6, id, 1)\}$ and since $min(0.5, 1) = 0.5$, then $\lambda_f(\mathtt{h}(\mathtt{s}(\mathtt{X}'),\mathtt{s}(\mathtt{X}')),\ \mathcal{P}_\mathtt{h}) = \{(\Lambda, R_6, id, 0.5)\}$.

Inspired by the schema used in [2], the following diagram reproduces the previous explanation in a concise, but precise manner:

$$\lambda_{\mathtt{fuzzy}}(\mathtt{f}(\mathtt{X},\mathtt{g}(\mathtt{X},\mathtt{X}))) = \cdots \qquad \begin{pmatrix} \langle \Lambda, \mathtt{R}_1, \{\mathtt{X} \mapsto \mathtt{a}\}, \mathtt{min}(1,1)\rangle \\ \langle 2, \mathtt{R}_5, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}')\}, \mathtt{min}(1,1)\rangle \\ \langle 2, \mathtt{R}_6, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}')\}, \mathtt{min}(1,0.5)\rangle \end{pmatrix}$$

[BV] $\lambda_\mathtt{f}(\mathtt{f}(\mathtt{X},\mathtt{g}(\mathtt{X},\mathtt{X})),\ \mathcal{P}_\mathtt{f}) = \cdots$ $\left|\begin{matrix} \langle \Lambda, \mathtt{R}_1, \{\mathtt{X} \mapsto \mathtt{a}\}, 1\rangle \\ \langle 2, \mathtt{R}_5, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}')\}, 1\rangle \\ \langle 2, \mathtt{R}_6, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}')\}, 0.5\rangle \end{matrix}\right|$

[LR] $\lambda_\mathtt{f}(\mathtt{f}(\mathtt{a},\mathtt{g}(\mathtt{a},\mathtt{a})),\ \mathcal{P}_\mathtt{f}^{\mathtt{f}(\mathtt{a},\mathtt{Y})}) =$ $\langle \Lambda, \mathbf{R_1}, \mathbf{id}, \mathbf{1}\rangle$

[BF] $\lambda_\mathtt{f}(\mathtt{f}(\mathtt{s}(\mathtt{X}'),\mathtt{g}(\mathtt{s}(\mathtt{X}'),\mathtt{s}(\mathtt{X}'))),\ \mathcal{P}_\mathtt{f}^{\mathtt{f}(\mathtt{s}(\mathtt{X}'),\mathtt{Y})}) = \cdots$ $\left|\begin{matrix} \langle 2, \mathtt{R}_5, \mathtt{id}, \mathtt{min}(1,1)\rangle \\ \langle 2, \mathtt{R}_6, \mathtt{id}, \mathtt{min}(0.7,0.5)\rangle \end{matrix}\right|$

[BC] $\lambda_\mathtt{f}(\mathtt{g}(\mathtt{s}(\mathtt{X}'),\mathtt{s}(\mathtt{X}')),\ \mathcal{P}_\mathtt{g}) = \cdots$ $\langle \Lambda, \mathtt{R}_5, \mathtt{id}, \mathtt{min}(1,1)\rangle$

[LR] $\lambda_\mathtt{f}(\mathtt{g}(\mathtt{s}(\mathtt{X}'),\mathtt{s}(\mathtt{X}')),\ \mathcal{P}_\mathtt{g}^{\mathtt{g}(\mathtt{s}(\mathtt{M}),\mathtt{N})}) =$ $\langle \Lambda, \mathbf{R_5}, \mathbf{id}, \mathbf{1}\rangle$

[BC] $\lambda_\mathtt{f}(\mathtt{h}(\mathtt{s}(\mathtt{X}'),\mathtt{s}(\mathtt{X}')),\ \mathcal{P}_\mathtt{h}) = \cdots$ $\langle \Lambda, \mathtt{R}_6, \mathtt{id},\ \mathtt{min}(0.5,1)\rangle$

[LR] $\lambda_\mathtt{f}(\mathtt{h}(\mathtt{r}(\mathtt{X}'),\mathtt{s}(\mathtt{X}')),\ \mathcal{P}_\mathtt{h}^{\mathtt{h}(\mathtt{r}(\mathtt{X}),\mathtt{Y})}) =$ $\langle \Lambda, \mathbf{R_6}, \mathbf{id}, \mathbf{1}\rangle$

As our example reveals, there are three important properties enjoyed by our extended definition of needed narrowing: 1) the set of recursive calls performed during the computation of λ_{fuzzy} is finite and terminating, 2) λ_{fuzzy} is conservative w.r.t. λ_{crisp} since, the first two tuples computed before are the same to those ones obtained in the crisp case (see example 2.2), but accompanied now with the maximum truth degree 1, and 3) similarity equations between defined/constructor function symbols are exploited as much as possible (in the initial call and BF/BC cases), which is the key point to obtain the third tuple in our example. In the following, we formally prove all these properties.

4 Termination

We start this section by introducing some auxiliary concepts which will be very helpful when formulating/proving our main results. Intuitively, we say that two terms t and t' which contain exactly the same set of positions, that is, $\mathcal{P}os(t) = \mathcal{P}os(t')$, are similar if each pair of symbols rooting two (non-variable) subterms at the same position in both terms are similars w.r.t. $\Re$. In symbols, $\forall p \in \mathcal{P}os(t),\ \Re(t{\uparrow_p}, t'{\uparrow_p}) > 0$. An special case appears when comparing two variable subterms, where we require that both subterms be exactly the same variable. This implies the need for applying and appropriate variable renaming on both terms before comparing them. For instance, terms $f(X, X)$ and $g(D, D)$, after being renamed become $f(A, A)$ and $g(A, A)$, respectively, and then they can be considered similar if $\Re(f, g) > 0$. On

the other hand, $f(X,X)$ and $f(Z,Y)$, become after renaming $f(A,A)$ and $f(A,B)$, and they can never be considered similar terms. Moreover, the similarity degree α of two similar terms is defined as the minimum one among all the similarities exploited in the calculus, that is $\alpha = min_{p \in \mathcal{P}os(t)}\ \Re(t{\uparrow_p}\ , t'{\uparrow_p})$. We formalize this notion by means of the following recursive definition:

Definition 4.1 [Similar Terms] Two terms t and t' (whose variables have been appropriately renamed) are similar with similar degree α if $Sim(t,t') = \alpha > 0$, where function Sim is recursively defined as:

$$Sim(t,t') = \begin{cases} min(\beta, Sim(u_1,v_1), .., & \text{if } t \equiv f(u_1,..,u_n), t' \equiv g(v_1,..,v_n), \\ \quad .., Sim(u_n,v_n)) & f,g \in (\mathcal{C} \cup \mathcal{F}) \text{ and } \Re(f,g) = \beta > 0 \\ 1 & \text{if } t \text{ and } t' \text{ are the same variable} \\ 0 & \text{otherwise} \end{cases}$$

The previous definition extends the similarity relation $\Re$ between symbols, to similarity between terms, and it represents a nice measure that can be calculated at a purely syntactic level (before performing any kind of evaluation process based on rewriting/narrowing). Moreover, it enjoys the following property.

Lemma 4.2 *Function Sim represents a well defined similarity relation between terms, verifying the properties of reflexivity, symmetry and transitivity.*

Proof.

(i) *Reflexivity:* $Sim(t,t) = 1$. Obvious by Definition 4.1 of Sim.

(ii) *Symmetry:* $Sim(t,s) = Sim(s,t)$. This result is again trivially derived from the Symmetry of $\Re$ and Definition 4.1.

(iii) *min-Transitivity:* $Sim(t,s) \geq (Sim(t,t') \wedge Sim(t',s))$. From definition 4.1 we have that:
$Sim(t,s) = min_i(\Re(t{\uparrow_i}, s{\uparrow_i})) = min_i\{Max_j[\Re(t{\uparrow_i}, t'{\uparrow_j}) \wedge \Re(t'{\uparrow_j}, s{\uparrow_i})]\} \geq$
$min_i(\Re(t{\uparrow_i}, t'{\uparrow_i}) \wedge \Re(t'{\uparrow_i}, s{\uparrow_i})) = min_i(\Re(t{\uparrow_i}, t'{\uparrow_i})) \wedge min_i(\Re(t'{\uparrow_i}, s{\uparrow_i})) =$
$Sim(t,t') \wedge Sim(t',s)$.

□

The following definition will largely help us to prove by induction all the properties of our fuzzified version of needed narrowing. Inspired by [2], we extend their notion of Noetherian ordering $\prec$ between tuples of terms and definitional (sub-)trees[3] by taken also into account the presence of similarities among terms in the new fuzzy setting.

[3] This ordering is very useful in induction-based proofs for $\lambda_c(t,\mathcal{P})$ and $\lambda_f(t,\mathcal{P})$, since both strategies compute tuples throughout an interleaved descent down both t and $\mathcal{P}$.

Definition 4.3 We define the Noetherian ordering $\prec$ over the cartesian product of $Terms \times (sub{-})Trees$ as follows:

$$(t,\mathcal{P}) \prec (t',\mathcal{P}') \Leftrightarrow \begin{cases} a)\ t \text{ has fewer occurrences of defined symbols than t'} \\ b)\ \text{There exists a (possibly empty) substitution } \sigma, \text{ such} \\ \text{that } Sim(t,\sigma(t')) > 0 \text{ and } \mathcal{P} \text{ is a proper sub-tree of } \mathcal{P}' \end{cases}$$

The first case of our definition corresponds exactly with the homologous one presented in [2], and it is very useful in our proofs when dealing with BF. Moreover, the second case generalizes the associated one in the same paper, but taking now into account the specificities introduced in the new fuzzy setting by similar terms. This last case will be applied when analyzing BV and BC cases, as we are going to see in the following proof of our first theorem.

Theorem 4.4 (Termination) *Given an operation-rooted term t, the evaluation of $\lambda_{fuzzy}(t)$ terminates in a finite time.*

Proof. As defined in Section 3, the initial call to $\lambda_{fuzzy}(f(\overline{t_n}))$ generates a set of calls of the form $\lambda_f(g_i(\overline{t_n}),\mathcal{P}_{g_i})$ where each g_i is a symbol verifying $\Re(g_i,f) > 0$. This set of calls is finite due to the proper finiteness of $\Re$. So, our goal simply consists in proving that each call to λ_f eventually terminates. The proof is made by induction on the ordering introduced in Definition 4.3.

- Base Case. This case, which corresponds to the so called **LR case** of the definition of λ_f shown in Section 3, is trivial since no recursive calls are performed.
- Induction step. Now, we consider the three subcases of the definition of λ_f for branch nodes:

BV case. In order to evaluate $\lambda_f(t,\mathcal{P})$ in a BV case, a set of n calls of the form $\lambda_f(\tau(t),\mathcal{P}^{\pi_i})$ are performed, being n the number of children of the root of $\mathcal{P}$. Since condition b in Definition 4.3 implies that $(\tau(t),\mathcal{P}^{\pi_i}) \prec (t,\mathcal{P})$, then the claim follows by the inductive hypothesis.

BC case. This case is very close to the previous one, but now only a few (not all) children of the root of $\mathcal{P}$ are exploited. More exactly, $\lambda_f(t,\mathcal{P})$ performs a finite set of calls of the form $\lambda_f(t_i,\mathcal{P}^{\pi_i})$ such that t and t_i are exactly the same terms except for the constructor symbols rooting subterms at the inductive position o in both terms. Moreover, $\Re(t{\uparrow_o},t_i{\uparrow_o}) = \alpha > 0$, and then $Sim(t,t_i) = \alpha > 0$. So, by the second condition in Definition 4.3 we have that $(t_i,\mathcal{P}^{\pi_i}) \prec (t,\mathcal{P})$, which let us to confirm our claim by the inductive hypothesis.

BF case. The evaluation of $\lambda_f(t,\mathcal{P})$ in a BF case, generates a finite set of recursive calls in a similar way to the initial call to λ_{fuzzy}, but the main difference now is that each new call uses as first parameter a proper subterm of t. Since t is an operation rooted term, obviously $t|_o$ has less defined operation symbols than t (at least the first occurrence of the operation symbol rooting t has been removed in $t|_o$). This satisfies the first condition of the $\prec$ ordering, and concludes our proof by the inductive hypothesis.

□

To finish this section, it is important to remember the following remark we introduced in the preliminaries section. In order to avoid the risk of infinite loops associated to the intrinsic (reflexive, symmetric and transitive) properties of similarity relations, we avoid the direct use of similarity equations in our definition of λ_{fuzzy}. Instead of this, we prefer to compute the transitive closure of such set of similarity equations, thus obtaining the finite similarity relation $\Re$ which can be safely accessed from λ_{fuzzy} without harming the termination property we have just proved.

5 Crispness

This section is devoted to prove a kind of *correctness* result (both soundness and completeness) enjoyed by λ_{fuzzy} when compared with λ_{crisp}. The idea is to show that our fuzzy strategy is conservative w.r.t. the crisp case. In this sense, observe in Example 3.1 that tuples $\langle \wedge, R_1, \{\mathtt{X} \mapsto \mathtt{a}\}\rangle$ and $\langle 2, R_5, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}')\}\rangle$, obtained by λ_{crisp}, are also replicated by λ_{fuzzy} with the maximum truth degree 1, which implies a sort of *crisp-completeness*. On the other hand, whereas λ_{fuzzy} produces three tuples, only two of them (except $\langle 2, R_6, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}')\}, 0.5\rangle$) have the maximum truth degree 1, but these tuples are also generated in the crisp case, which represents a kind of *crisp-soundness* related to tuples with maximum truth degree. The following theorem formalizes this property.

Theorem 5.1 (Crispness) *For any operation-rooted term t, we have that:*

Crisp-Completeness. If $\langle p, R, \sigma\rangle \in \lambda_{crisp}(t)$ then $\langle p, R, \sigma, 1\rangle \in \lambda_{fuzzy}(t)$.

Crisp-Soundness. If $\langle p, R, \sigma, 1\rangle \in \lambda_{fuzzy}(t)$ and there is no similarity relations with truth degree 1 apart from the reflexive one, then $\langle p, R, \sigma\rangle \in \lambda_{crisp}(t)$.

Proof. We consider each claim separately, assuming that $t = f(\overline{t_n})$:

• *Crisp − Completeness*. Our goal is to show how tuple $\langle p, R, \sigma\rangle \in \lambda_{crisp}(t)$ can be also obtained in the fuzzy case by applying $\lambda_{fuzzy}(t)$. We know that by the definition of λ_{crisp} shown in Section 2, $\lambda_{crisp}(t) = \lambda_c(t, \mathcal{P}_f)$ and also by the definition of λ_{fuzzy} shown in Section 3, $\lambda_{fuzzy}(t)$ makes at least the call $\lambda_f(t, \mathcal{P}_f)$ since $\Re(f, f) = 1$ (being f the defined function symbol rooting term t). So, we must prove that if $\langle p, R, \sigma\rangle \in \lambda_c(t, \mathcal{P}_f)$ then $\langle p, R, \sigma, 1\rangle \in \lambda_f(t, \mathcal{P}_f)$. The proof is made by induction on the Noetherian ordering $\prec$ while considering the different cases of the definition of λ_c.

- Base Case. This case corresponds to the so called **LR case** of the definition of λ_c shown in Section 2, where the associated definitional tree $\mathcal{P}_f = \{\pi\}$ only contains a pattern (leaf-rule node) and no recursive calls are performed. Then $\lambda_c(t, \mathcal{P}_f) = \{\langle p, R, \sigma\rangle\}$ and by the definition of λ_f shown in Section 3, we directly conclude with $\lambda_f(t, \mathcal{P}_f) = \{\langle p, R, \sigma, 1\rangle\}$, as we wanted to prove.
- Induction step. We consider now the three subcases of the definition of λ_c shown in Section 2, for branch nodes:

BV case. Here $t|_o = x \in \mathcal{X}$, and there exists a pattern $\pi_i \equiv \pi[c_i(\overline{x_n})]_o \in \mathcal{P}_f$

and a substitution $\tau = \{x \mapsto c_i(\overline{x_n})\}$, such that, after performing the recursive call $\lambda_c(\tau(t), \mathcal{P}_f^{\pi_i})$, we obtain (p, R, θ) and hence, $\lambda_c(t, \mathcal{P}_f)$ returns the intended tuple (p, R, σ) where $\sigma = \theta \circ \tau$. By the inductive hypothesis (condition b in Definition 4.3 is fulfilled since the recursive call is done with an instance of t, and a proper sub-tree of $\mathcal{P}_f$) $\lambda_f(\tau(t), \mathcal{P}_f^{\pi_i})$ also generates tuple $(p, R, \theta, 1)$, and finally, by the definition of λ_f shown in Section 3, $\lambda_f(t, \mathcal{P}_f)$ returns the intended tuple $(p, R, \sigma, 1)$, as we wanted to prove.

BC case. This case is very similar to the previous one. Now, we have that $t|_o = c_i(\overline{t_n})$, where $c_i \in \mathcal{C}$, and there exists a pattern $\pi_i \equiv \pi[c_i(\overline{x_n})]_o \in \mathcal{P}_f$ such that, after performing the recursive call $\lambda_c(t, \mathcal{P}_f^{\pi_i})$, we obtain (p, R, σ) and hence, $\lambda_c(t, \mathcal{P}_f)$ returns the intended tuple (p, R, σ) (since $\sigma = \sigma \circ id$). The inductive hypothesis is now applicable in the same way than the previous case (observe that the recursive call is done with the same term, which always can be considered an instance of itself, and an smaller sub-tree) so $\lambda_f(t, \mathcal{P}_f^{\pi_i})$ also generates the tuple $(p, R, \sigma, 1)$. This implies that by the definition of λ_f shown in Section 3, $\lambda_f(t, \mathcal{P}_f)$ returns the tuple $(p, R, \sigma, min(\alpha, 1))$, and since $\alpha = \Re(c_i, c_i) = 1$, the resulting tuple $(p, R, \sigma, 1)$ belongs to $\lambda_f(t, \mathcal{P}_f)$.

BF case. Now, $t|_o = g(\overline{s_m})$, where $g \in \mathcal{F}$, and there exists a definitional tree $\mathcal{P}_g$ for g such that, after performing the recursive call $\lambda_c(t|_o, \mathcal{P}_g)$, we obtain (q, R, σ) and hence, $\lambda_c(t, \mathcal{P}_f)$ returns the intended tuple (p, R, σ), where $p = o.q$ and $\sigma = \sigma \circ id$. The inductive hypothesis is applicable here since the recursive call is made with the proper subterm of t, $t|_o$ (which obviously contains less defined functions symbols than t and hence satisfies condition a of Definition 4.3). So, $\lambda_f(t|_o, \mathcal{P}_g)$ also generates the tuple $(q, R, \sigma, 1)$. By the definition of λ_f shown in Section 3, $\lambda_f(t, \mathcal{P}_f)$ makes at least a recursive call to $\lambda_f(t|_o, \mathcal{P}_g)$, due to the fact that $\Re(g, g) = 1$, and hence, it returns the tuple $(p, R, \sigma, min(1, 1))$, which obviously is $(p, R, \sigma, 1)$, as wanted.

• *Crisp – Soundness*. This proof is very similar to the previous one, by simply reverting the reasoning in each case and assuming that the unique similarity relations exploited by λ_{fuzzy} are the reflexive ones, that is, those of the form $\Re(f, f) = 1, \forall f \in (\mathcal{C} \cup \mathcal{F})$. □

As a final remark, we illustrate by a simple example the need for requiring that no similarity equations of the form $eq(f, g) = 1$ belongs to a given program, if we really want to preserve the crisp-soundness of λ_{fuzzy}. Given a program composed by the single rewrite rule $R : \mathtt{f(a)} \rightarrow \mathtt{a}$ and the single similarity equation $E : \mathtt{eq(f, g)} = \mathtt{1}$, then $\lambda_{fuzzy}(\mathtt{g(a)}) = \{\langle \Lambda, \mathtt{R_1}, \mathtt{id}, \mathtt{1} \rangle\}$ whereas $\lambda_{crips}(\mathtt{g(a)}) = \emptyset$. Fortunately, these cases are not usual in practice, and moreover, in the worst case, the original program often admits a transformation process (we are nowadays working in its automation) that removes those risky similarity equations and replaces them by a set of rewrite rules without altering the program meaning (for instance, in the previous case it suffices by replacing equation E by rule $R : \mathtt{g(a)} \rightarrow \mathtt{a}$ to recover the previously lost crisp-soundness).

6 Fuzziness

In this section we prove our last property related to λ_{fuzzy}, which is conceived as a complementary result of the previous property. In fact, the fuzzy strategy not only is conservative w.r.t. the crisp one, but also it is able to exploit the similarity relation $\Re$ as much as possible when evaluating λ_{fuzzy}. In doing this, we need the following instrumental result showing that, when we consider two identical terms with the exception that they have two different, but similar, symbols at a given position p, they produce the same set of tuples (except for their truth degrees) when being evaluated with λ_{fuzzy}.

Lemma 6.1 (Similar Symbols Replacement) *Let t and t' be two operation rooted terms with $\mathcal{P}os(t) = \mathcal{P}os(t')$ such that, for a given position $p \in \mathcal{P}os(t)$, we have: $t|_p = s(\overline{o})$, $t'|_p = s'(\overline{o})$, $t' = t[s'(\overline{o})]_p$, and $\Re(s, s') > 0$. Then, $\langle p, R, \sigma, \alpha\rangle \in \lambda_{fuzzy}(t)$ if and only if $\langle p, R, \sigma, \alpha'\rangle \in \lambda_{fuzzy}(t')$.*

Proof. Due to the reflexivity, symmetry and transitivity of the similarity relation $\Re$, we have that, if $\Re(s, s') > 0$, then for any other symbol s'', it is verified that $\Re(s, s'') = \beta > 0$ if and only if $\Re(s', s'') = \beta' > 0$. In other words, if we denote by $\mathcal{S}(s)$ the set of symbols $\{s_i \in (\mathcal{C} \cup \mathcal{F}) \mid \Re(s, s_i) > 0\}$, then $\mathcal{S}(s) = \mathcal{S}(s')$. On the other hand, if t and t' are the same terms with the exception that they have different, but similar symbols (say s and s'), at a given position p, we have that the sets of tuples returned by $\lambda_{fuzzy}(t)$ and $\lambda_{fuzzy}(t')$ do coincide in the case that position p is never considered as an inductive position in any recursive call to λ_f. Otherwise, we have the following cases:

- The intended position p is the top position $\wedge$. Then, By the definition of λ_{fuzzy} shown in Section 3, $\lambda_{fuzzy}(s(\overline{o}))$ performs exactly the same set of calls to λ_f than $\lambda_{fuzzy}(s'(\overline{o}))$, say $\lambda_f(s_i(\overline{o}), \mathcal{P}_{s_i})$, for all $s_i \in \mathcal{S}(s)$. Now, if $\langle q, R, \sigma, \gamma\rangle \in \lambda_f(s_i(\overline{o}))$, then $\langle q, R, \sigma, min(\Re(s, s_i), \gamma)\rangle \in \lambda_{fuzzy}(s(\overline{o}))$ and $\langle q, R, \sigma, min(\Re(s', s_i), \gamma)\rangle \in \lambda_{fuzzy}(s'(\overline{o}))$, which confirms our claim in this case.
- The intended position p is eventually considered as an inductive position in a BC case. Then, by the definition of λ_f shown in Section 3, $\lambda_f(t[s(\overline{o})]_p, \mathcal{P})$ performs exactly the same set of recursive calls than $\lambda_f(t[s'(\overline{o})]_p, \mathcal{P})$, by using as first arguments terms of the form $t[s_i(\overline{o})]_p$, for all $s_i \in \mathcal{S}(s)$. So, similarly to the previous case, if $\langle q, R, \sigma, \gamma\rangle \in \lambda_f(t[s_i(\overline{o})]_p, \mathcal{P}^{\pi_i})$, then $\langle q, R, \sigma, min(\Re(s, s_i), \gamma)\rangle \in \lambda_f(t[s(\overline{o})]_p, \mathcal{P})$ and $\langle q, R, \sigma, min(\Re(s', s_i), \gamma)\rangle \in \lambda_f(t[s'(\overline{o})]_p, \mathcal{P})$ too, which confirms our claim once again.
- The intended position p is eventually considered as an inductive position in a BF case. Our last case has many similarities with the two previous ones. Now, by the definition of λ_f shown in Section 3, $\lambda_f(t[s(\overline{o})]_p, \mathcal{P}_s)$ performs exactly the same set of recursive calls than $\lambda_f(t[s'(\overline{o})]_p, \mathcal{P}_{s'})$, that is $\lambda_f(s_i(\overline{o}), \mathcal{P}_{s_i})$, for all $s_i \in \mathcal{S}(s)$. So, if $\langle q, R, \sigma, \gamma\rangle \in \lambda_f(s_i(\overline{o}), \mathcal{P}_{s_i})$, then $\langle p.q, R, \sigma, min(\Re(s, s_i), \gamma)\rangle \in \lambda_f(t[s(\overline{o})]_p, \mathcal{P}_s)$ and it is also verified that $\langle p.q, R, \sigma, min(\Re(s', s_i), \gamma)\rangle \in \lambda_f(t[s'(\overline{o})]_p, \mathcal{P}_{s'})$, which finishes our proof.

□

Next lemma extends the previous one by considering multiple similar symbols replacements (instead of a single symbol) when comparing the evaluation of λ_{fuzzy} with two similar terms.

Lemma 6.2 (Similar Subterms Replacement) *Let t and t' be two operation rooted terms with $\mathcal{P}os(t) = \mathcal{P}os(t')$ such that, for a given position $p \in \mathcal{P}os(t)$, we have: $t|_p = s$, $t'|_p = s'$, $t' = t[s']_p$, and $Sim(s, s') > 0$. Then, $\langle p, R, \sigma, \alpha\rangle \in \lambda_{fuzzy}(t)$ if and only if $\langle p, R, \sigma, \alpha'\rangle \in \lambda_{fuzzy}(t')$.*

Proof. The result is easily obtained by repeatedly applying Lemma 6.1. □

Finally, we are now ready to prove our main result in this section.

Theorem 6.3 (Fuzziness) *Let t and t' be two operation rooted terms, such that $Sim(t, t') > 0$. Then, $\langle p, R, \sigma, \alpha\rangle \in \lambda_{fuzzy}(t)$ iff $\langle p, R, \sigma, \alpha'\rangle \in \lambda_{fuzzy}(t')$.*

Proof. This theorem directly follows from the previous result, by simply considering that the intended position p used in the formulation of Lemma 6.2, is the top position Λ. □

Example 6.4 Going back again to Example 3.1 and denoting now with $\alpha_{\mathtt{rs}}$ and $\alpha_{\mathtt{h,g}}$ to $\Re(\mathtt{r},\mathtt{s})$ and $\Re(\mathtt{h},\mathtt{g})$, respectively, it is easy to see that any pair of terms t and t' chosen among $\mathtt{g(s(X),X)}$, $\mathtt{g(r(X),X)}$, $\mathtt{h(r(x),x)}$ and $\mathtt{h(s(x),x)}$ verify $Sim(t, t') > 0$. The following diagrams reproduce the evaluation of λ_{fuzzy} for each one of the previous terms for illustrating our fuzziness result:

$$\lambda_{\mathtt{fuzzy}}(\mathtt{g(s(X),X)}) = \cdots \qquad \begin{pmatrix}\langle 1, \mathtt{R_5}, \mathtt{id}, 1\rangle \\ \langle 1, \mathtt{R_6}, \mathtt{id}, \alpha_{\mathtt{hg}}\rangle\end{pmatrix}$$

$$\begin{array}{ll}
[\mathtt{BC}] & \lambda_{\mathtt{f}}(\mathtt{g(s(X),X)}, \mathcal{P}_{\mathtt{g}}^{\mathtt{g(s)}}) = \cdots \qquad \langle 1, \mathtt{R_5}, \mathtt{id}, 1\rangle \\
[\mathtt{LR}] & \lambda_{\mathtt{f}}(\mathtt{g(s(X),X)}, \mathcal{P}_{\mathtt{g}}^{\mathtt{g(s(X'),Y)}}) = \qquad \langle \mathbf{1}, \mathbf{R_5}, \mathbf{id}, 1\rangle \\
[\mathtt{BC}] & \lambda_{\mathtt{f}}(\mathtt{h(s(X),X)}, \mathcal{P}_{\mathtt{h}}) = \cdots \qquad \langle 1, \mathtt{R_6}, \mathtt{id}, \alpha_{\mathtt{hg}}\rangle \\
[\mathtt{LR}] & \lambda_{\mathtt{f}}(\mathtt{h(s(X),X)}, \mathcal{P}_{\mathtt{h}}^{\mathtt{h(s(X'),Y)}}) = \qquad \langle \mathbf{1}, \mathbf{R_6}, \mathbf{id}, 1\rangle
\end{array}$$

$$\lambda_{\mathtt{fuzzy}}(\mathtt{g(r(X),X)}) = \cdots \qquad \begin{pmatrix}\langle 1, \mathtt{R_5}, \mathtt{id}, \alpha_{\mathtt{rs}}\rangle \\ \langle 1, \mathtt{R_6}, \mathtt{id}, \alpha_{\mathtt{hg}}\rangle\end{pmatrix}$$

$$\begin{array}{ll}
[\mathtt{BC}] & \lambda_{\mathtt{f}}(\mathtt{g(r(X),X)}, \mathcal{P}_{\mathtt{g}}) = \cdots \qquad \langle 1, \mathtt{R_5}, \mathtt{id}, \alpha_{\mathtt{rs}}\rangle \\
[\mathtt{LR}] & \lambda_{\mathtt{f}}(\mathtt{g(s(X),X)}, \mathcal{P}_{\mathtt{g}}^{\mathtt{g(s(X'),Y)}}) = \qquad \langle \mathbf{1}, \mathbf{R_5}, \mathbf{id}, 1\rangle \\
[\mathtt{BC}] & \lambda(\mathtt{h(r(X),X)}, \mathcal{P}_{\mathtt{h}}) = \cdots \qquad \langle 1, \mathtt{R_6}, \mathtt{id}, \alpha_{\mathtt{hg}}\rangle \\
[\mathtt{LR}] & \lambda_{\mathtt{f}}(\mathtt{h(r(X),X)}, \mathcal{P}_{\mathtt{h}}^{\mathtt{h(r(X'),Y)}}) = \qquad \langle \mathbf{1}, \mathbf{R_6}, \mathbf{id}, 1\rangle
\end{array}$$

$$\lambda_{\mathtt{fuzzy}}(\mathtt{h(r(X)),X)}) = \cdots \qquad \begin{pmatrix}\langle 1, \mathtt{R_6}, \mathtt{id}, 1\rangle \\ \langle 1, \mathtt{R_5}, \mathtt{id}, \alpha_{\mathtt{hg}} \wedge \alpha_{\mathtt{rs}}\rangle\end{pmatrix}$$

$$\begin{array}{ll}
[\mathtt{BC}] & \lambda_{\mathtt{f}}(\mathtt{h(r(X)),X)}, \mathcal{P}_{\mathtt{h}}) = \cdots \qquad \langle 1, \mathtt{R_6}, \mathtt{id}, 1\rangle \\
[\mathtt{LR}] & \lambda_{\mathtt{f}}(\mathtt{h(r(X),X)}, \mathcal{P}_{\mathtt{h}}^{\mathtt{h(r(X'),Y)}}) = \qquad \langle \mathbf{1}, \mathbf{R_6}, \mathbf{id}, 1\rangle \\
[\mathtt{BC}] & \lambda_{\mathtt{f}}(\mathtt{g(r(X)),X)}, \mathcal{P}_{\mathtt{g}}) = \qquad \langle 1, \mathtt{R_5}, \mathtt{id}, \alpha_{\mathtt{rs}}\rangle \\
[\mathtt{LR}] & \lambda_{\mathtt{f}}(\mathtt{g(s(X),X)}, \mathcal{P}_{\mathtt{g}}^{\mathtt{g(s(X'),Y)}}) = \qquad \langle \mathbf{1}, \mathbf{R_5}, \mathbf{id}, 1\rangle
\end{array}$$

$$\lambda_{\mathtt{fuzzy}}(\mathtt{h}(\mathtt{s}(\mathtt{X}),\mathtt{X})) = \cdots \qquad \begin{pmatrix} \langle \mathtt{1}, \mathtt{R_6}, \mathtt{id}, \alpha_{\mathtt{rs}} \rangle \\ \langle \mathtt{1}, \mathtt{R_5}, \mathtt{id}, \mathtt{1} \rangle \end{pmatrix}$$

[BC]	$\lambda_{\mathtt{f}}(\mathtt{h}(\mathtt{s}(\mathtt{X}),\mathtt{X}), \mathcal{P}^{\mathtt{h}}_{\mathtt{h}}) = \cdots$	$\langle \mathtt{1}, \mathtt{R_6}, \mathtt{id}, \alpha_{\mathtt{rs}} \rangle$
[LR]	$\lambda_{\mathtt{f}}(\mathtt{h}(\mathtt{r}(\mathtt{X}),\mathtt{X}), \mathcal{P}^{\mathtt{h}(\mathtt{r}(\mathtt{X}'),\mathtt{Y})}_{\mathtt{h}}) =$	$\langle \mathbf{1}, \mathbf{R_6}, \mathbf{id}, \mathbf{1} \rangle$
[BC]	$\lambda_{\mathtt{f}}(\mathtt{g}(\mathtt{s}(\mathtt{X}),\mathtt{X}), \mathcal{P}^{\mathtt{g}}_{\mathtt{g}}) = \cdots$	$\langle \mathtt{1}, \mathtt{R_5}, \mathtt{id}, \mathtt{1} \rangle$
[LR]	$\lambda_{\mathtt{f}}(\mathtt{g}(\mathtt{s}(\mathtt{X}),\mathtt{X}), \mathcal{P}^{\mathtt{g}(\mathtt{s}(\mathtt{X}'),\mathtt{Y})}) =$	$\langle \mathbf{1}, \mathbf{R_5}, \mathbf{id}, \mathbf{1} \rangle$

7 Conclusions and Future Work

In this paper we have been concerned with the first attempt for introducing fuzziness into functional-logic programming that we proposed in [7], where we provided (both the syntax and the operational semantics of) an hybrid functional-fuzzy-logic language combining the functional-logic properties of Curry with the fuzzy-logic features of Likelog. From here, we have reinforced the original idea that a set of rewrite rules together with a set of similarity equations can be successfully processed by using an extended version of needed narrowing which has the extra ability of dealing with similarity relations. Focusing on the new fuzzy narrowing strategy represented by function λ_{fuzzy}, we have demonstrated its capability not only for simulating the crisp version described by λ_{crisp}, but also for capturing the similarities collected in a given program as much as possible without harming its termination properties. We think that the set of results we have just formalized and proved in this paper are not only relevant by themselves, but also, and what is better, they seem to be crucial for proving new soundness/completeness results of the fuzzified version of needed narrowing with respect to a new declarative semantics in which we are working nowadays. In this sense, it must be clear that the denotation we are looking for, is not an application from (ground) expressions to data terms. Rather than this, in the fuzzy setting, we think that it seems more natural to associate fuzzy sets (modeled as pairs composed by a data term together with its corresponding truth degree) to a given initial (ground) term, which, among other things, implies a subsequent re-formulation of the notion of confluence typically used in pure functional and functional-logic programming.

References

[1] S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, pages 143–157. Springer LNCS 632, 1992.

[2] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Journal of the ACM*, volume 47(4), pages 776–822, 2000.

[3] F. Arcelli and F. Formato. Likelog: A logic programming language for flexible data retrieval. In *Proc. of the ACM Symp. on Applied Computing (SAC'99)*, pages 260–267. ACM, Artificial Intelligence and Computational Logic, 1999.

[4] L. Garmendia and A. Salvador. Comparing transitive closure with other new T-transitivization methods. In *Proc. Modeling Decisions for Artificial Intelligence*, pages 306–315. Springer LNAI 3131, 2004.

[5] G. Huet and J.J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, ed., *Computational Logic. Essays in Honor of Alan Robinson*, pages 395–443. The MIT Press, Cambridge, MA, 1992.

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

[7] G. Moreno and V. Pascual. Programming with Fuzzy–logic and Mathematical Functions. In I. Bloch, A. Petrosino, and A. Tettamanzi, editors, *Proc. of the 6th. International Whorshop on Fuzzy Logic and Applications, WILF'05, University of Milan, Crema (Italy)*, pages 89–98. Springer LNCS 3849, 2006.

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

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

[10] L. A. Zadeh. Similarity relations and fuzzy orderings. *Inf. Sci.*, 3:177-200, 1971.