Formal Properties of Needed Narrowing with Similarity Relations

Ginés Moreno and Vicente Pascual

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
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(s_1, s_2) = \alpha, whith the intended meaning that s_1 and s_2 are predicate/function
symbols of the same arity with a similarity degree \alpha. 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 as-
sumed 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 prop-
erty. 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, sim-
ilarity 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 vice versa).

Let us recall that a T-norm \( \land \) in \([0, 1]\) is a binary operation \( \land : [0, 1] \times [0, 1] \rightarrow [0, 1] \)
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 \( x \land y \) is
equivalent to \( \min(x, y) \), that is, the minimum between two elements \( x, y \in [0, 1] \),
whereas \( x \lor y \) and \( \max(x, y) \) refers to the maximum operator. Both notations have
a long presence in the specialized literature and help us to clarify expressions in
recursive calculations. A similarity relation \( \mathcal{R} \) on a domain \( \mathcal{U} \) is a fuzzy subset
\( \mathcal{R} : \mathcal{U} \times \mathcal{U} \rightarrow [0, 1] \) of \( \mathcal{U} \times \mathcal{U} \) such that, \( \forall x, y, z \in \mathcal{U} \), the following properties
hold: reflexivity \( \mathcal{R}(x, x) = 1 \), symmetry \( \mathcal{R}(x, y) = \mathcal{R}(y, x) \) and transitivity
\( \mathcal{R}(x, z) \geq \mathcal{R}(x, y) \land \mathcal{R}(y, z) \). In the following, we assume that the intended similarity relation
\( \mathcal{R} \) 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: \( T^t_\mathcal{R}(R) \bigcup_{f=1}^\infty R^f \) 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 \( \mathcal{R} = T_\mathcal{R}(R) = R^{n-1} \) [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].

\[
\begin{pmatrix}
1 & .7 & .6 & .4 \\
.7 & 1 & .8 & .9 \\
.6 & .8 & 1 & .7 \\
.4 & .9 & .7 & 1
\end{pmatrix}
\rightarrow
\begin{pmatrix}
1 & .7 & .7 \\
.7 & 1 & .8 & .9 \\
.7 & .8 & 1 & .8 \\
.7 & .9 & .8 & 1
\end{pmatrix}
\]

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 C of constructors and a set F of defined functions (also called operations). The set of constructor terms (with variables) is obtained by using symbols from C (and a set of variables X). The set of variables occurring in a term t is denoted by Var(t). We write o_n for the list of objects o_1, ..., o_n. A pattern is a term of the form f(a_n) where f/n ∈ F and d_1, ..., 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 Pos(t). Positions are ordered by the prefix ordering: p ≤ q, if ∃w such that p.w = q. t|_p and t↑_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, ..., x_n \mapsto t_n\} the substitution σ whose application to a term t is denoted by σ(t). A set of rewrite rules l → r such that l /∈ X, and Var(r) ⊆ Var(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 R is left-linear if l is linear for all l → r ∈ 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 →^{p,R}_σ s if there exists a position p in t, a rewrite rule R = (l → r) and a substitution σ with t|_p = σ(l) and s = t[σ(r)]|_p. Narrowing is a combination of variable instantiation and reduction. Formally, t ↔^{p,R}_σ s is a narrowing step if p is a non-variable position in t and σ(t) →^{p,R}_σ 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:

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

LR (LEAF-RULE) CASE: \( \lambda_c(t, \mathcal{P}_f) = \{(\Lambda, R, id)\} \)
BV (BRANCH-VAR) CASE: If \( t \mid_o = x \in \mathcal{X} \), then \( \lambda_c(t, \mathcal{P}_f) = \{(p, R, \sigma \circ id) \mid \pi_i \equiv \pi[c_i(\overline{x_n})] \in \mathcal{P}_f \) and \( \tau = \{x \mapsto c_i(\overline{x_n})\} \) and \( (p, R, \sigma) \in \lambda_c(\tau(t), \mathcal{P}_f^n)\}\}

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

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

When none of the previous cases is satisfied, we assume that function \( \lambda_c \) 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 \( \lambda_c \), 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 \( \sigma \) is a substitution appearing in a tuple \( (p, R, \sigma) \in \lambda_{crisp}(t) \), then \( \sigma \) only contains bindings for variables of \( t \). So, the application of \( \sigma \) 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, \( t \rightarrow_{p, R} s \) 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 \( \lambda_{crisp}(f(X, g(X, X))) = \{(\land, R_1, \{X \mapsto a\}), (2, R_5, \{X \mapsto s(X')\})\} \) which enables the following pair of needed narrowing steps: \( f(X, g(X, X)) \rightarrow_{\{X \mapsto a\}}^\land_{R_1} a \), and \( f(X, g(X, X)) \rightarrow_{\{X \mapsto s(X')\}}^{2, R_5} f(s(X'), s(g(X, s(X')))) \).

### 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 \( \lambda_{fuzzy} \) 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 \( \lambda_{fuzzy} \), we must guarantee that any term (including \( t \) itself), rooted with a symbol similar to \( f \) be will be treated. So, \( \lambda_{fuzzy}(t)\{(p, R, \alpha, \beta) \mid \Re(f, g) = \alpha \) and \( (p, R, \sigma, \beta) \in \lambda_f(g(\overline{s_n}), \mathcal{P}_g)\} \), where function \( \lambda_f \) is defined as follows:
**LR (LEAF-RULE) CASE:**  
\[ \lambda_f(t, P_f) = \{(\lambda, \pi \rightarrow r, id, 1}\}

**BV (BRANCH-VAR) CASE:** if \( t|_o = x \in \mathcal{X} \), then  
\[ \lambda_f(t, P_f) = \{(p, R, \sigma \circ \tau, \alpha) \mid \pi_i \equiv \pi[c_i(\overrightarrow{x})]|_{o} \in P_f, \ \tau = \{x \mapsto c_i(\overrightarrow{x})\} \} \]

**BC (BRANCH-CONS) CASE:**  
if \( t|_o = d(\overrightarrow{t_n}) \), then \( \lambda_f(t, P_f) = \{(p, R, \sigma, \min(\alpha, \beta)) \mid \mathcal{R}(d/n, c_i/n) = \alpha > 0 \ \text{and} \ \pi_i \equiv \pi[c_i(\overrightarrow{x})]|_{o} \in P_f \)  
and \( (p, R, \sigma, \beta) \in \lambda_f(t[c_i(\overrightarrow{t_n})]|_{o}, P_f^{\overrightarrow{\tau_i}}) \)

**BF (BRANCH-FUNC) CASE:**  
if \( t|_o = g(\overrightarrow{t_n}) \), where \( g \in \mathcal{F} \), then \( \lambda(t, P_f) = \{(o,p, R, \sigma, \min(\alpha, \beta)) \mid \mathcal{R}(g/n, h/n) = \alpha > 0 \ \text{and} \ (p, R, \sigma, \beta) \in \lambda(h(\overrightarrow{t_n}), 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 \( \lambda_f_{\text{fuzzy}} \) seen before, the last case (BF) performs recursive calls to \( \lambda_f \) 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 \( R_6 : h(r(X), Y) \rightarrow r(Y) \) together with the similarity equations  
\[ S_1 : \text{eq}(g, h) = 0.7 \ \text{and} \ S_2 : \text{eq}(s, r) = 0.5 \]. Then,  
\[ \lambda_f_{\text{fuzzy}}(f(X, g(X), X)) = \lambda_f(f(X, g(X), X), P_f^a) = \{(\lambda, R_1, \{X \mapsto a\}, 1), (2, R_5, \{X \mapsto s(X)\}, 1), (2, R_6, \{X \mapsto s(X)\}, \min(0.7, 0.5))\} \]

**BV1.** The first alternative in this BV case, consists in generating the binding \( \tau_1 = \{X \mapsto a\} \) and then computing  
\[ \lambda_f(\tau_1(f(X, g(X))), P_f^{(a,Y)}) = \lambda_f(f(a, g(a), a), P_f^{(a,Y)}) \]. Since this last call represents a LR case, it returns  
\( \{(\lambda, R_1, \{X \mapsto a\}, 1)\} \). Then, after applying the binding \( \tau_1 \) to the third element of this last tuple, the returned set for this case is  
\( \{(\lambda, R_1, \{X \mapsto a\}, 1)\} \).

**BV2.** After generating the second binding \( \tau_2 = \{X \mapsto s(X)\} \), we must compute  
\[ \lambda_f(\tau_2(f(X, g(X))), P_f^{(s(X),Y)}) = \lambda_f(f(s(X), g(s(X)), s(X)), P_f^{(s(X),Y)}) = \{(2, R_5, id, 1), (2, R_6, id, \min(0.7, 0.5))\} \]

**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):  
\[ \lambda_f(g(s(X), s(X)), P_g) \cup \lambda_f(h(s(X), s(X)), P_h) = [\text{see BC1}] \cup [\text{see BC2}] \{(\lambda, R_5, id, 1), (\lambda, R_6, id, 0.5)\}. \] 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:  
\[ \lambda_f(g(s(X), s(X)), P_g^{(s(X),Y)}) = \{(\lambda, R_5, id, 1)\}. \] Now, since \( \mathcal{R}(s, s) = 1 \), and \( \min(1, 1) = 1 \), the returned tuple in this case is  
\( \{(\lambda, R_5, id, 1)\} \) itself.
As our example reveals, there are three important properties enjoyed by our example. Let both subterms be exactly the same variable. This implies the need for application in both terms are similar with respect to the same position, that is, $P_{a,b}(s(x), s(x')) = \{a, R_5, \min(0.5, 1)\}$ and since $\min(0.5, 1) = 0.5$, then $\lambda_f(h(s(x'), s(x')), P_h) = \{a, R_6, \min(0.5, 0.5)\}$.

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

$$
\lambda_{fuzzy}(f(X, g(X, X))) = \cdots \\
\begin{pmatrix}
\langle\Lambda, R_1, \{X \mapsto a\}, \min(1, 1)\rangle \\
\langle2, R_5, \{X \mapsto s(X')\}, \min(1, 1)\rangle \\
\langle2, R_6, \{X \mapsto s(X')\}, \min(0.5, 0.5)\rangle
\end{pmatrix}
$$

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 $\lambda_{fuzzy}$ is finite and terminating, 2) $\lambda_{fuzzy}$ is conservative w.r.t. $\lambda_{crisp}$ 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, $Pos(t) = Pos(t')$, are similar if each pair of symbols rooting two (non-variable) subterms at the same position in both terms are similar w.r.t. $R$. In symbols, $\forall p \in Pos(t), R(t^p, t'^p) > 0$. 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 $R(f, g) > 0$. 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 \( \alpha \) of two similar terms is defined as the minimum one among all the similarities exploited in the calculus, that is \( \alpha = \min_{p \in P_{os(t)}} R(t \uparrow_p, t' \uparrow_p) \). 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 \( \alpha \) if \( Sim(t, t') = \alpha > 0 \), where function \( Sim \) is recursively defined as:

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

The previous definition extends the similarity relation \( R \) 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 \( R \) 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:

\[
Sim(t, s) = \min_i (R(t \uparrow_i, s \uparrow_i)) = \min_i \{ \max_j [R(t \uparrow_i, t' \uparrow_j) \wedge R(t' \uparrow_j, s \uparrow_i)] \} \geq \min_i (R(t \uparrow_i, t' \uparrow_j) \wedge R(t' \uparrow_j, s \uparrow_i)) = \min_i (R(t \uparrow_i, t' \uparrow_j)) \wedge \min_i (R(t' \uparrow_j, s \uparrow_i)) = Sim(t, t') \wedge Sim(t', s).
\]

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 \( \lambda_c(t, P) \) and \( \lambda_f(t, P) \), since both strategies compute tuples throughout an interleaved descent down both \( t \) and \( P \).
Definition 4.3 We define the Noetherian ordering $\prec$ over the cartesian product of $\text{Terms} \times (\text{sub-})\text{Trees}$ as follows:

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

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 $\lambda_{fuzzy}(f(t_n))$ generates a set of calls of the form $\lambda_f(g_i(t_n), P_{g_i})$ 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 $\lambda_f$ 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 $\lambda_f$ shown in Section 3, is trivial since no recursive calls are performed.
- Induction step. Now, we consider the three subcases of the definition of $\lambda_f$ for branch nodes:

**BV case.** In order to evaluate $\lambda_f(t, P)$ in a BV case, a set of $n$ calls of the form $\lambda_f(\tau(t), P^{\pi_i})$ are performed, being $n$ the number of children of the root of $P$. Since condition b in Definition 4.3 implies that $(\tau(t), P^{\pi_i}) \prec (t, P)$, 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 $P$ are exploited. More exactly, $\lambda_f(t, P)$ performs a finite set of calls of the form $\lambda_f(t_i, P^{\pi_i})$ 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, $\Re(t|_o, t_i|_o) = \alpha > 0$, and then $\text{Sim}(t, t_i) = \alpha > 0$. So, by the second condition in Definition 4.3 we have that $(t_i, P^{\pi_i}) \prec (t, P)$, which let us to confirm our claim by the inductive hypothesis.

**BF case.** The evaluation of $\lambda_f(t, P)$ in a BF case, generates a finite set of recursive calls in a similar way to the initial call to $\lambda_{fuzzy}$, 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. 

$\square$
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 $\lambda_{fuzzy}$. Instead of this, we prefer to compute the transitive closure of such set of similarity equations, thus obtaining the finite similarity relation $\mathcal{R}$ which can be safely accessed from $\lambda_{fuzzy}$ 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 $\lambda_{fuzzy}$ when compared with $\lambda_{crisp}$. 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 $\langle \Lambda, R_1, \{x \mapsto a\} \rangle$ and $\langle 2, R_5, \{x \mapsto s(x')\} \rangle$, obtained by $\lambda_{crisp}$, are also replicated by $\lambda_{fuzzy}$ with the maximum truth degree 1, which implies a sort of crisp-completeness. On the other hand, whereas $\lambda_{fuzzy}$ produces three tuples, only two of them (except $\langle 2, R_6, \{x \mapsto s(x')\}, 0.5 \rangle$) 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 $t = f(t^n)$:

- **Crisp - Completeness.** Our goal is to show that 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 $\lambda_{crisp}$ shown in Section 2, $\lambda_{crisp}(t) = \lambda_c(t, \mathcal{P}_f)$ and also by the definition of $\lambda_{fuzzy}$ shown in Section 3, $\lambda_{fuzzy}(t)$ makes at least the call $\lambda_f(t, \mathcal{P}_f)$ since $\mathcal{R}(f, f) = 1$ (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 $\lambda_c$.

- **Base Case.** This case corresponds to the so called LR case of the definition of $\lambda_c$ 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 $\lambda_c(t, \mathcal{P}_f) = \{(p, R, \sigma)\}$ and by the definition of $\lambda_f$ shown in Section 3, we directly conclude with $\lambda_f(t, \mathcal{P}_f) = \{(p, R, \sigma, 1)\}$, as we wanted to prove.

- **Induction step.** We consider now the three subcases of the definition of $\lambda_c$ shown in Section 2, for branch nodes:

**BV case.** Here $t|_o = x \in \mathcal{X}$, and there exists a pattern $\pi_i \equiv \pi(c_i(x_n)|_o) \in \mathcal{P}_f$
and a substitution \( \tau = \{ x \mapsto c_i(\overline{x_n}) \} \), such that, after performing the recursive call \( \lambda_c(\tau(t), \mathcal{P}_f^\tau) \), we obtain \((p, R, \theta)\) and hence, \( \lambda_c(t, \mathcal{P}_f) \) returns the intended tuple \((p, R, \sigma)\) 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 \( \mathcal{P}_f \) \( \lambda_f(\tau(t), \mathcal{P}_f^\tau) \) also generates tuple \((p, R, \theta, 1)\), and finally, by the definition of \( \lambda_f \) 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 \( \pi_i = \pi[c_i(\overline{x_n})]_o \in \mathcal{P}_f \) such that, after performing the recursive call \( \lambda_c(\pi_i, \mathcal{P}_f) \), we obtain \((p, R, \sigma)\) and hence, \( \lambda_c(t, \mathcal{P}_f) \) returns the intended tuple \((p, R, \sigma)\) (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 \( \lambda_f(t, \mathcal{P}_f^\pi) \) also generates the tuple \((p, R, \sigma, 1)\). This implies that by the definition of \( \lambda_f \) shown in Section 3, \( \lambda_f(t, \mathcal{P}_f) \) returns the tuple \((p, R, \sigma, min(\alpha, 1))\), and since \( \alpha = \mathcal{R}(c_i, c_i) = 1 \), 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 \((q, R, \sigma)\) and hence, \( \lambda_c(t, \mathcal{P}_f) \) returns the intended tuple \((p, R, \sigma)\), 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 \( \lambda_f \) 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 \( \mathcal{R}(g, g) = 1 \), 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 \( \lambda_{fuzzy} \) are the reflexive ones, that is, those of the form \( \mathcal{R}(f, f) = 1, \forall f \in (\mathcal{C} \cup \mathcal{F}) \).

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 \( \lambda_{fuzzy} \). Given a program composed by the single rewrite rule \( R : f(a) \rightarrow a \) and the single similarity equation \( E : eq(f, g) = 1 \), then \( \lambda_{fuzzy}(g(a)) = \{ (\lambda, R_1, id, 1) \} \) whereas \( \lambda_{crisp}(g(a)) = \emptyset \). 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 \( R : g(a) \rightarrow a \) to recover the previously lost crisp-soundness).
6 Fuzziness

In this section we prove our last property related to $\lambda_{\text{fuzzy}}$, 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 $\lambda_{\text{fuzzy}}$. 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 $\lambda_{\text{fuzzy}}$.

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

**Proof.** Due to the reflexivity, symmetry and transitivity of the similarity relation $\Re$, we have that, if $\mathcal{R}(s, s') > 0$, then for any other symbol $s''$, it is verified that $\mathcal{R}(s, s'') = \beta > 0$ if and only if $\mathcal{R}(s', s'') = \beta' > 0$. In other words, if we denote by $\mathcal{S}(s)$ the set of symbols $\{s_i \in (C \cup \mathcal{F}) \mid \mathcal{R}(s, s_i) > 0\}$, 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 $\lambda_{\text{fuzzy}}(t)$ and $\lambda_{\text{fuzzy}}(t')$ do coincide in the case that position $p$ is never considered as an inductive position in any recursive call to $\lambda_f$. Otherwise, we have the following cases:

- The intended position $p$ is the top position $\Lambda$. Then, By the definition of $\lambda_{\text{fuzzy}}$ shown in Section 3, $\lambda_{\text{fuzzy}}(s(\bar{p}))$ performs exactly the same set of calls to $\lambda_f$ than $\lambda_{\text{fuzzy}}(s'(\bar{p}))$, say $\lambda_f(s_i(\bar{p}), \mathcal{P}_{s_i})$, for all $s_i \in \mathcal{S}(s)$. Now, if $\langle q, R, \sigma, \gamma \rangle \in \lambda_f(s_i(\bar{p}))$, then $\langle q, R, \sigma, \min(\mathcal{R}(s, s_i), \gamma) \rangle \in \lambda_{\text{fuzzy}}(s(\bar{p}))$ and $\langle q, R, \sigma, \min(\mathcal{R}(s', s_i), \gamma) \rangle \in \lambda_{\text{fuzzy}}(s'(\bar{p}))$, 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 $\lambda_f$ shown in Section 3, $\lambda_f(t[s(\bar{p})]|_p, \mathcal{P})$ performs exactly the same set of recursive calls than $\lambda_f(t[s'(\bar{p})]|_p, \mathcal{P})$, by using as first arguments terms of the form $t[s_i(\bar{p})]|_p$, for all $s_i \in \mathcal{S}(s)$. So, similarly to the previous case, if $\langle q, R, \sigma, \gamma \rangle \in \lambda_f(t[s_i(\bar{p})]|_p, \mathcal{P})$, then $\langle q, R, \sigma, \min(\mathcal{R}(s, s_i), \gamma) \rangle \in \lambda_f(t[s(\bar{p})]|_p, \mathcal{P})$ and $\langle q, R, \sigma, \min(\mathcal{R}(s', s_i), \gamma) \rangle \in \lambda_f(t[s'(\bar{p})]|_p, \mathcal{P})$ 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 $\lambda_f$ shown in Section 3, $\lambda_f(t[s(\bar{p})]|_p, \mathcal{P})$ performs exactly the same set of recursive calls than $\lambda_f(t[s'(\bar{p})]|_p, \mathcal{P})$, that is $\lambda_f(s_i(\bar{p}), \mathcal{P}_{s_i})$, for all $s_i \in \mathcal{S}(s)$. So, if $\langle q, R, \sigma, \gamma \rangle \in \lambda_f(s_i(\bar{p}), \mathcal{P}_{s_i})$, then $\langle p.q, R, \sigma, \min(\mathcal{R}(s, s_i), \gamma) \rangle \in \lambda_f(t[s(\bar{p})]|_p, \mathcal{P})$ and it is also verified that $\langle p.q, R, \sigma, \min(\mathcal{R}(s', s_i), \gamma) \rangle \in \lambda_f(t[s'(\bar{p})]|_p, \mathcal{P})$, 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 $\lambda_{fuzzy}$ with two similar terms.

**Lemma 6.2 (Similar Subterms Replacement)** Let $t$ and $t'$ be two operation rooted terms with $\mathcal{Pos}(t) = \mathcal{Pos}(t')$ such that, for a given position $p \in \mathcal{Pos}(t)$, 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 $\lambda$.

□

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

\[
\begin{align*}
\lambda_{fuzzy}(g(s(x), x)) &= \cdots \\
&= \begin{cases} 
1, R_5, \text{id}, 1 \end{cases} \quad \begin{cases} 
1, R_5, \text{id}, \alpha_{hg} \end{cases} \\
[BC] &\lambda_f(g(s(x), x), P_g^{s(x)}) = \cdots \begin{cases} 
1, R_5, \text{id}, 1 \end{cases} \\
[LR] &\lambda_f(g(s(x), x), P_g^{s(x)}, y) = \begin{cases} 
1, R_5, \text{id}, 1 \end{cases} \\
[BC] &\lambda_f(h(s(x), x), P_h) = \cdots \begin{cases} 
1, R_6, \text{id}, \alpha_{hg} \end{cases} \\
[LR] &\lambda_f(h(s(x), x), P_h^{s(x)}, y) = \begin{cases} 
1, R_6, \text{id}, 1 \end{cases}
\end{align*}
\]

\[
\begin{align*}
\lambda_{fuzzy}(g(r(x), x)) &= \cdots \\
&= \begin{cases} 
1, R_5, \text{id}, \alpha_{rs} \end{cases} \quad \begin{cases} 
1, R_5, \text{id}, \alpha_{hg} \end{cases} \\
[BC] &\lambda_f(g(r(x), x), P_g) = \cdots \begin{cases} 
1, R_5, \text{id}, \alpha_{rs} \end{cases} \\
[LR] &\lambda_f(g(r(x), x), P_g^{s(x)}, y) = \begin{cases} 
1, R_5, \text{id}, 1 \end{cases} \\
[BC] &\lambda(h(r(x), x), P_h) = \cdots \begin{cases} 
1, R_6, \text{id}, \alpha_{hg} \end{cases} \\
[LR] &\lambda_f(h(r(x), x), P_h^{r(x)}, y) = \begin{cases} 
1, R_6, \text{id}, 1 \end{cases}
\end{align*}
\]

\[
\begin{align*}
\lambda_{fuzzy}(h(r(x), x)) &= \cdots \\
&= \begin{cases} 
1, R_6, \text{id}, 1 \end{cases} \quad \begin{cases} 
1, R_5, \text{id}, \alpha_{rs} \end{cases} \quad \begin{cases} 
1, R_5, \text{id}, \alpha_{hg} \end{cases} \\
[BC] &\lambda_f(h(r(x), x), P_h) = \cdots \begin{cases} 
1, R_6, \text{id}, 1 \end{cases} \\
[LR] &\lambda_f(h(r(x), x), P_h^{r(x)}, y) = \begin{cases} 
1, R_6, \text{id}, 1 \end{cases} \\
[BC] &\lambda_f(g(r(x)), x), P_g) = \cdots \begin{cases} 
1, R_5, \text{id}, \alpha_{rs} \end{cases} \\
[LR] &\lambda_f(g(s(x), x), P_g^{s(x)}, y) = \begin{cases} 
1, R_5, \text{id}, 1 \end{cases}
\end{align*}
\]
\[ \lambda_{\text{fuzzy}}(h(s(X), X)) = \cdots \]

\[ \begin{array}{l}
\text{[BC]} \quad \lambda_f(h(s(X), X), P^h_h) = \cdots \\
\text{[LR]} \quad \lambda_f(h(r(X), X), P^h_h(r'(X), Y)) = \\
\text{[BC]} \quad \lambda_f(g(s(X), X), P^g_g) = \cdots \\
\text{[LR]} \quad \lambda_f(g(s(X), X), P^g_g(s'(X), Y)) =
\end{array} \]

\[ \begin{pmatrix}
1, R_6, \text{id}, \alpha_{rs} \\
1, R_5, \text{id}, 1
\end{pmatrix} \]

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 \( \lambda_{fuzzy} \), we have demonstrated its capability not only for simulating the crisp version described by \( \lambda_{crisp} \), 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


