Similarity-based Unification Embedded into Narrowing

Ginés Moreno, Vicente Pascual

Similarity-based Unification Embedded into Narrowing *

Ginés Moreno and Vicente Pascual

Department of Computing Systems
U. Castilla-La Mancha, 02071 Albacete, Spain
{gmoreno,vpascual}@dsi.uclm.es

Abstract. Likelog is a fuzzy–logic language with a Prolog-like syntax augmented with similarity equations (modeling the fuzzy component of the language) and with an operational semantics based on resolution where the classical syntactic unification algorithm of pure logic programming has been replaced by a similarity-based unification method. On the other hand, Curry is a functional–logic language with a Haskell-like syntax and an operational principle based on (needed) narrowing, that is, a combination of syntactic unification and rewriting. In this paper we propose a new similarity-based unification method which empowers the original one used in Likelog by also taking into account functional features like laziness. This calculus is specially well suited for being embedded into the kernel of the original needed narrowing strategy of Curry in a very natural way, also verifying nice formal properties such as termination, crispness (i.e., it computes at least the same elements of the crisp case) and fuzzyness (i.e., similarities collected in a given program are exploited as much as possible). Our final goal is to achieve the complete integration of the (also integrated) declarative paradigms of functional–logic and fuzzy–logic programming, in order to obtain a richer and much more expressive programming scheme where mathematical functions cohabit with fuzzy logic features.

Keywords: Similarity-based Unification, Needed Narrowing

1 Introduction

Fuzzy Logic Programming amalgamates fuzzy logic [14] and pure logic programming [10], 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, 16], 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 that can be considered "equal" with a given degree [17].

* 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.

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, similarity-based) 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 it 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 remark again that, similarly to Likelog, no similarity equations are allowed between two symbols with different arity and/or nature.

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 [16], in the sequel we assume that $x \wedge y$ is equivalent to $min(x, y)$, that is, the minimum between two elements $x, y \in [0, 1]$. 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: $T^t(R) = \bigcup_{f=1\ldots\infty} R^f$ 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 [6]. Finally, by simply assuming that the set of similarity equations in $\mathcal{R}$ is trivially extended by reflexivity, then $\Re = T^t(R) = R^{(n-1)}$ [6]. For instance, in the following pair of matrix, we consider similarities among four arbitrary constant symbols. The second matrix has been obtained from the first one after applying the algorithm described in [6].

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

On the other hand, functional logic programming languages combine the operational principles of the most important declarative programming paradigms, namely functional and logic programming (see [7] for a survey). The operational semantics of such integrated languages is usually based on narrowing, a combination of variable instantiation and reduction, where efficient demand-driven functional computations are amalgamated with the flexible use of logical variables providing for function inversion and search for solutions. In order to proceed with a maximal integration (thus fulfilling the gap we have detected in the area), in this paper we propose the combined use of similarity equations (Likelog) together with rewriting rules (instead of Horn clauses) typically used in pure functional (Haskell) and integrated functional–logic (Curry) languages, as we have recently done in our preliminar works [12, 13]. Nevertheless, the present work extends both precedents since, in particular, it is the first time that we formalize and effectively apply the flexible and powerful notion of "needed narrowing with similarity relations". In order to intuitively illustrate these ideas and to motivate this work, let us consider the following extremely simple example.

Example 1. The following rewriting rule R : $\texttt{colour}(\texttt{car}) \rightarrow \texttt{red}$, with an obvious meaning, can be used by a pure functional language like Haskell to evaluate (by rewriting) the term $\texttt{colour}(\texttt{car})$, thus producing the trivial value `red` as final answer. Moreover, Haskell fails when evaluating $\texttt{colour}(\texttt{X})$, being `X` a variable symbol, due to the fact that the operational principle of pure functional languages is based on pure rewriting which uses pattern matching instead of unification before applying a reduction step. In contrast with Haskell, functional logic languages enrich such operational principle by allowing the instantiation of free variables on a given goal before reducing it. This is the case of Curry, which extends Haskell by simply using (needed) narrowing instead of pure rewriting as operational mechanism. Now, goal $\texttt{colour}(\texttt{X})$ can be successfully processed by Curry, thus obtaining as final computed answer the pair $\langle \texttt{red}, \{\texttt{X} \mapsto \texttt{car}\} \rangle$.

On the other hand, one might think that the concepts of `colour` and `tone`, although not completely equivalents, can be considered "similars" at (for instance) an 80 % *similarity degree*. This idea can be modeled in a fuzzy logic language such as Likelog by using a similarity equation of the form $S : \texttt{eq}(\texttt{colour}, \texttt{tone}) = 0.8$. In this (fuzzy) logic setting, we can also model function `colour` as a predicate, and instead of rule R, we simply need to define it by means of the equivalent fact (that is, a clause with empty body) C : $\texttt{colour}(\texttt{car}, \texttt{red})$. Now, the Likelog program composed by clause C and similarity equation S is able to execute goal $\texttt{tone}(\texttt{X}, \texttt{Y})$ providing as final fuzzy computed answer the pair $\langle \{\texttt{X} \mapsto \texttt{car}, \texttt{Y} \mapsto \texttt{red}\}; 0.8 \rangle$, by simply exploiting the similarity of predicates `colour` and `tone`, even when there is no clause defining `tone`.

Obviously Haskell and Curry are "crisp languages", which in contrast with Likelog, are not designed for recognizing a similarity equation like S, and hence both would fail when trying to evaluate (by rewriting/narrowing steps using the rewriting rule R) the functional–logic expression $\texttt{tone}(\texttt{X})$. On the contrary, Likelog has also an important handicap regarding Haskell and Curry: since it has a logic syntax based on clauses, it is not easy to express "composition of mathematical functions" and consequently, it can not successfully exploit efficient lazy evaluation techniques, higher order features, and other expressive resources with a "functional taste", as Haskell and Curry do.

By proposing an hybrid dialect amalgamating the best properties of Curry and Likelog, in this paper we plan to override all the limitations we have observed

when comparing such languages. The intended "fuzzified" version of Curry we are looking for, must be able for instance, to deal with the previous rewriting rule R and similarity equation S in order to, given a goal like `tone(X)`, produce as fuzzy computed answer a triple (including the final "functional" value, the "logic" substitution and the "fuzzy" similarity degree) of the form $\langle \texttt{red}; \{\texttt{X} \mapsto \texttt{car}\}; 0.8\rangle$.

The structure of this paper is as follows. In Section 2 we explain in detail the original *needed narrowing* strategy used by Curry. Section 3 represents the transition of this "crisp" strategy to the "fuzzy" one used for extending needed narrowing with similarity relations in Section 4. Finally, we give our conclusions and propose future work in Section 5.

2 The Original Needed Narrowing Strategy

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.

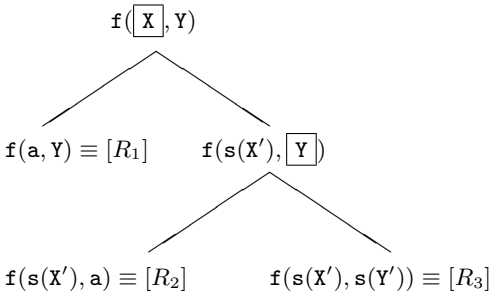
We consider a *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of *defined* functions (also called *operations*). The set of *constructor terms* (with *variables*) is obtained by using symbols from $\mathcal{C}$ (and a set of variables $\mathcal{X}$). The set 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 $\sigma(t)$. We write $t \leq t'$ (*subsumption*) if $t' = \sigma(t)$ for some substitution σ. 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 $s = t[\sigma(r)]_p$.

In order to evaluate terms containing variables, the narrowing mechanism non-deterministically instantiates the variables so that a rewrite step is possible.

The operational semantics of modern integrated languages is usually based on *(needed) narrowing*, which can be seen as a combination of variable instantiation and reduction. Formally, $t \leadsto^{p,R}_{\sigma} s$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \rightarrow^{p,R} s$. We denote by $t_0 \leadsto^{*}_{\sigma} t_n$ a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$. Since we are interested in computing *values* (constructor or constructor-rooted terms) as well as *answers* (substitutions) in functional logic programming, we say that the narrowing derivation $t \leadsto^{*}_{\sigma} c$ *computes the result* c *with answer* σ if c is a constructor term. The evaluation to (ground) constructor terms (and not to arbitrary expressions) is the intended semantics of functional languages and also of most functional logic languages.

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 [8] by using *definitional trees* [1], a concept which refines the standard matching trees of functional programming. 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. 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:



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

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

For defining 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 $R : \pi \rightarrow r$ 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 (sub-)tree in each case. Then, $\lambda_{crisp}(t) = \lambda_c(t, \mathcal{P}_f)$ returns the least set satisfying:

`LR (LEAF-RULE) CASE:` If π is a leaf ($\mathcal{P}_f = \{\pi\}$) then $\lambda_c(t, \mathcal{P}_f) = \{(\Lambda, R, id)\}$
`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.

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, it is important to note 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 3. For instance, if we consider again the rules for `f` and `g` in Example 2 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 pair of needed narrowing steps: $\underline{\mathtt{f(X, g(X, X))}} \leadsto_{\{\mathtt{X} \mapsto \mathtt{a}\}}^{\Lambda, R_1} \mathtt{a}$, and $\mathtt{f(X, } \underline{\mathtt{g(X, X)}}\mathtt{)} \leadsto_{\{\mathtt{X} \mapsto \mathtt{s(X')}\}}^{2, R_5} \mathtt{f(s(X'), s(g(X', s(X'))))}$.

To finish this section, we wish to highlight a remarkable feature of needed narrowing. In contrast to more traditional narrowing strategies [7], 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 directly said in [2], "*We forgo the requirement that the unifier of a narrowing step be most general. The instantiation that we demand in addition to that for the most general unification ensures the need of the position irrespective of future unifiers. It turns out that this extra instantiation would eventually be performed later in the derivation. Thus, we are only "anticipating" it and the completeness of narrowing is preserved.* Although this approach, however, complicates the notion of narrowing strategy, we must always take in mind that it is crucial to achieve the optimal properties of needed narrowing. Moreover, this action will have important consequences when defining in Section 4 out fuzzy extension of this narrowing principle.

3 From the Crisp (λ_{crisp}) to the Fuzzy (λ_{fuzzy}) Strategy

This section recalls from [12] the basis of our first 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 narrowing 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 will eventually be treated. So, $\lambda_{fuzzy}(t) = \{(p, R, \sigma, min(\alpha, \beta)) \mid \Re(f/n, g/n) = \alpha > 0 \text{ and } (p, R, \sigma, \beta) \in \lambda_f(g(\overline{s_n}), \mathcal{P}_g)\}$, where function λ_f is defined as follows:

`LR (LEAF-RULE) CASE:` If π is a leaf then $\lambda_f(t, \mathcal{P}_f) = \{(\Lambda, \pi \rightarrow r, id, 1)\}$

`BV (BRANCH-VAR) CASE:` if $t|_o = x \in \mathcal{X}$, then $\lambda_f(t, \mathcal{P}_f) = \{(p, R, \sigma \circ \tau, \alpha) \mid \pi_i \equiv \pi[c_i(\overline{x_n})]_o \in \mathcal{P}_f$ and $\tau = \{x \mapsto c_i(\overline{x_n})\}$ 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 $\lambda_f(t, \mathcal{P}_f) = \{(p, R, \sigma, min(\alpha, \beta)) \mid \Re(d/n, c_i/n) = \alpha > 0 \text{ and } \pi_i \equiv \pi[c_i(\overline{x_n})]_o \in \mathcal{P}_f$ 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 $\lambda_f(t, \mathcal{P}_f) = \{(o.p, R, \sigma, min(\alpha, \beta)) \mid \Re(g/n, h/n) = \alpha > 0 \text{ and } (p, R, \sigma, \beta) \in \lambda_f(h(\overline{t_n}), \mathcal{P}_h)\}$

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 4. Consider again the same program of Example 2 augmented with the new rule $R_6 : \mathtt{h(r(X), Y)} \rightarrow \mathtt{r(Y)}$ 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, $\lambda_{fuzzy}(\mathtt{f(X, g(X, X))}) = \lambda_f(\mathtt{f(X, g(X, X))}, \mathcal{P}_\mathtt{f}) =$ [see `BV1`] $\cup$ [see `BV2`]$= \{(\Lambda, R_1, \{\mathtt{X} \mapsto \mathtt{a}\}, 1), (2, R_5, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, 1), (2, R_6, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, min(0.7, 0.5))\}$.

`BV1.` The first alternative in this `BV` case, consists in generating a binding of the form $\tau_1 = \{\mathtt{X} \mapsto \mathtt{a}\}$ and then computing the recursive call $\lambda_\mathtt{f}(\tau_1(\mathtt{f(X, g(X, X))}), \mathcal{P}_\mathtt{f}^{\mathtt{f(a,Y)}}) = \lambda_\mathtt{f}(\mathtt{f(a, g(a, a))}, \mathcal{P}_\mathtt{f}^{\mathtt{f(a,Y)}})$. Since this last call represents a `LR` case, it directly 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 the call $\lambda_\mathtt{f}(\tau_2(\mathtt{f(X, g(X, X))}),\ \mathcal{P}_\mathtt{f}^{\mathtt{f(s(X'),Y)}}) = \lambda_\mathtt{f}(\mathtt{f(s(X'), g(s(X'), s(X')))},\ \mathcal{P}_\mathtt{f}^{\mathtt{f(s(X'),Y)}}) =$ [see `BF1` below] $= \{(2, R_5, id, 1), (2, R_6, id, min(0.7, 0.5))\}$. 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`] $\cup$ [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 `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 second similarity equation $S_2 : \mathtt{eq(s,r)} = 0.5$, this `BC` case also computes the `LR` case $\lambda_f(\mathtt{h(r(X'),s(X'))},\ \mathcal{P}_\mathtt{h}^{\mathtt{h(r(X),Y)}}) = \{(\Lambda, R_6, id, 1)\}$. Now, since $min(0.5, 1) = 0.5$, then $\lambda_f(\mathtt{h(s(X'),s(X'))},\ \mathcal{P}_\mathtt{h}) = \{(\Lambda, R_6, id, 0.5)\}$.

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

$$\lambda_{\mathtt{fuzzy}}(\mathtt{f(X,g(X,X))}) = \cdots \qquad \begin{pmatrix} \langle \Lambda, \mathtt{R_1}, \{\mathtt{X} \mapsto \mathtt{a}\}, \mathtt{min}(1,1)\rangle \\ \langle 2, \mathtt{R_5}, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, \mathtt{min}(1,1)\rangle \\ \langle 2, \mathtt{R_6}, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, \mathtt{min}(1,0.5)\rangle \end{pmatrix}$$

$$[\mathtt{BV}] \quad \lambda_\mathtt{f}(\mathtt{f(X,g(X,X))},\ \mathcal{P}_\mathtt{f}) = \cdots \qquad \left| \begin{matrix} \langle \Lambda, \mathtt{R_1}, \{\mathtt{X} \mapsto \mathtt{a}\}, 1\rangle \\ \langle 2, \mathtt{R_5}, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, 1\rangle \\ \langle 2, \mathtt{R_6}, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, 0.5\rangle \end{matrix} \right|$$

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

$$[\mathtt{BF}] \quad \lambda_\mathtt{f}(\mathtt{f(s(X'),g(s(X'),s(X')))},\ \mathcal{P}_\mathtt{f}^{\mathtt{f(s(X'),Y)}}) = \cdots \left| \begin{matrix} \langle 2, \mathtt{R_5}, \mathtt{id}, \mathtt{min}(1,1)\rangle \\ \langle 2, \mathtt{R_6}, \mathtt{id}, \mathtt{min}(0.7, 0.5)\rangle \end{matrix} \right|$$

$$[\mathtt{BC}] \quad \lambda_\mathtt{f}(\mathtt{g(s(X'),s(X'))},\ \mathcal{P}_\mathtt{g}) = \cdots \qquad \langle \Lambda, \mathtt{R_5}, \mathtt{id}, \mathtt{min}(1,1)\rangle$$

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

$$[\mathtt{BC}] \quad \lambda_\mathtt{f}(\mathtt{h(s(X'),s(X'))},\ \mathcal{P}_\mathtt{h}) = \cdots \qquad \langle \Lambda, \mathtt{R_6}, \mathtt{id},\ \mathtt{min}(0.5, 1)\rangle$$

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

As our example reveals, there are three important properties enjoyed by our extended strategy : 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 3), 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.

All these properties have been formally proved in [13]. In that paper, we also define the notion of similar terms. 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 similars 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, $\forall p \in \mathcal{P}os(t)$, $\Re(t{\uparrow_p}, t'{\uparrow_p}) > 0$. Using this concept, we can summarize now in the following theorem the main formal results detailed in [13].

Theorem 1 (Termination, Crispness and Fuzziness Properties [13]). *Given an operation-rooted term t:*
•TERMINATION*: The evaluation of $\lambda_{fuzzy}(t)$ terminates in a finite time.*
• 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 $\langle p, R, \sigma\rangle \in \lambda_{crisp}(t)$.*
• FUZZINESS*: For any other term t' similar to t, $\langle p, R, \sigma, \alpha\rangle \in \lambda_{fuzzy}(t)$ if and only if $\langle p, R, \sigma, \alpha'\rangle \in \lambda_{fuzzy}(t')$.*

4 Needed Narrowing with Similarity-based Unification

Now we are ready to formalize the fuzzy version of needed narrowing induced by λ_{fuzzy}. In Section 2 we have described the general narrowing calculus as the combination of instantiation of the variables of the term to be evaluated, followed by the reduction (rewriting) of some redex (i.e., a subterm which is an instance of the lhs of a program rule) of the resulting instantiated term. In general, given a TRS and a term to be evaluated, there exists many different ways for selecting tuples of the form $\langle position, rule, substitution\rangle$ in order to enable narrowing steps. In [7] we find an excellent survey describing various narrowing strategies (including the "needed" one), that have been proposed in the past. In order to compute unifiers (instead of simple matchers) most of these strategies use a (syntactic) unification algorithm like the one proposed in [11], which computes *most general unifiers (mgus)*. For this reasons, Padawitz [15] also distinguishes between narrowing and most general narrowing (but, in most papers, narrowing is intended as most general narrowing, e.g., [9]). Most general narrowing has the advantage that most general unifiers are uniquely computable, whereas there exist many distinct unifiers. As we have discussed in Section 2, dropping the requirement that unifiers be most general is crucial to the definition of a needed narrowing step since these steps may be impossible with most general unifiers.

On the other hand, we have seen that if the replacement of pattern matching by (most general) syntactic unification on a given rewriting mechanism, directly can provide a new (most general) narrowing strategy, one might think that the replacement of syntactic unification by a fuzzy, similarity-based unification method as the one proposed in [5, 16], would directly lead to a fuzzy narrowing strategy. This argument suggests a nice alternative for fuzzifying functional–logic languages, but... if we go a bit deeper, the following risks emerge:

- The efficiency of the underlying operational mechanism would be drastically diminished since a similarity-based unification method as the one proposed in [5, 16] is more flexible but also much more inefficient than the syntactic (mgu) unification method of [11].
- Moreover, since as we has just mentioned, needed narrowing is not properly based in more general unifiers (better that this, it can be implemented by simply generating matchers while exploring definitional trees) and hence, the similarity-based unification algorithm of [5, 16] can not be directly embedded in its core.

Fortunately, as we have seen in the previous section, our algorithm for computing λ_{fuzzy} successfully overrides both handicaps since:

- Our strategy simply computes matchers in the BV case, which are always easier to obtain than other kind of unifiers such as those more general or similarity-based ones.
- Our method is able to exploit similarities between constructor symbols (in BC cases), and between defined function symbols (in both, BF cases and the initial calls to λ_f performed by λ_{fuzzy}) at a very low level, without requiring the use of the much more involved algorithm proposed in [5, 16].

In some sense, these two reasons justify the novelty of our approach, and they also intuitively suggest that our fuzzified version of needed narrowing (described by λ_{fuzzy}) seems to be able to safely inherit, in the richer fuzzy setting, the optimal properties enjoyed by the original needed narrowing strategy (represented by λ_{crisp}). However, in order to really put in practice these benefits, we firstly need to apply the final refinement on λ_{fuzzy} that we are ready now to explain.

In contrast with the fuzzy case, since all the strategies described in [7] (including needed narrowing) have been designed in a crisp setting, they can be conceived as the combination of variable instantiation and reduction, that is, if a given strategy, represented by a concrete function λ, computes a tuple $\langle p, R : (l \to r), \sigma \rangle \in \lambda(t)$ for a given term t, we have that:

Claim (1). Substitution σ only needs to bind variables of t.
Claim (2). Subterm $t' \equiv \sigma(t)|_p$ is a redex of the instantiated term $\sigma(t)$, and hence, there exists a matcher σ' which only binds variables of l (the lhs of the program rule R) such that $t' = \sigma'(l)$.
Claim (3). After applying the corresponding rewriting subterm on such redex, we obtain the narrowed term $s \equiv \sigma(t[\sigma'(r)]_p)$.
Claim (4). Moreover, since $Dom(\sigma) \cap Dom(\sigma') = \emptyset$ and $Dom(\sigma') \cap Var(t) = \emptyset$, if we denote by θ the substitution obtained after composing σ with σ', i.e. $\theta = \sigma \circ \sigma'$, then $s \equiv \theta(t[r]_p)$.

The following example shows that, in the fuzzy setting, the second claim might fail in some cases, which also implies that, in general, neither claim (2), nor claims (3) and (4) are accomplished by λ_{fuzzy}.

Example 5. Remember that in Example 4 we computed $\lambda_{fuzzy}(\mathtt{f(X,g(X,X))}) = \{(\Lambda, R_1, \{\mathtt{X} \mapsto \mathtt{a}\}, 1), (2, R_5, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, 1), (2, R_6, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, 0.5)\}$. If we consider now the first tuple $(\Lambda, R_1, \{\mathtt{X} \mapsto \mathtt{a}\}, 1)$, we observe that after applying the binding $\sigma = \{\mathtt{X} \mapsto \mathtt{a}\}$ to the original term $t \equiv \mathtt{f(X,g(X,X))}$, we obtain $\mathtt{f(a,g(a,a))}$, whose subterm at the top position Λ directly coincides with the whole term and, in concordance with claim (2), it is also a redex regarding (the renamed version of) rule $R_1 : \mathtt{f(a,N1)} \to \mathtt{a}$ with matcher $\sigma' = \{\mathtt{N1} \mapsto \mathtt{g(a,a)}\}$. So, by applying claim (3), we can perform the following rewriting step: $\mathtt{f(a,g(a,a))} \to^{\Lambda, R_1} \mathtt{a}$. Observe also that the same result could be achieved by considering claim (4) with the proper substitution $\theta = \sigma \circ \sigma' = \{\mathtt{X} \mapsto \mathtt{a},\ \mathtt{N1} \mapsto \mathtt{g(a,a)}\}$. We wish to remark again that clearly unifier θ is not a simple matcher, since it binds two variables: whereas the first one (`X`) belongs

to the original term, the second one (N1) is included in the lhs of the program rule. However, it has been surprisingly obtained by composing two matchers (σ, which only binds a variable of the term and σ', which only binds a variable of the rule) instead of applying more expensive unification mechanisms (as many narrowing strategies do, in contrast with needed narrowing) like those designed in the past for syntactic [11], semantic [4] or similarity-based [5, 16] unification.

The reader may easily check that a similar reasoning to the previous one also applies to the second tuple obtained by $\lambda_{fuzzy}(\mathtt{f(X,g(X,X))})$. However, the third tuple $(2, R_6, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, 0.5)$, does not fulfills our second claim. In fact, we observe that after applying binding $\sigma = \{\mathtt{X} \mapsto \mathtt{s(X')}\}$ to the original term $t \equiv \mathtt{f(X,g(X,X))}$, we obtain $\mathtt{f(s(X'),g(s(X'),s(X')))}$, whose subterm at position 2, that is, $\mathtt{g(s(X'),s(X'))}$ is not a redex regarding (the renamed version of) rule $R_6 : \mathtt{h(r(X3),Y3)} \rightarrow \mathtt{r(Y3)}$, since we can not find a matcher σ' verifying that $\mathtt{g(s(X'),s(X'))} = \sigma'(\mathtt{h(r(X3),Y3)})$.

Although it is the first time that we point out this serious problem, in what follows we are going to solve it in a very simple but effective way. Remember that in our definition of λ_f shown in Section 3, variable bindings are only computed in the BV case, and they are only referred to variables of the term to be evaluated (i.e., they are simple matchers). By composing all the bindings obtained in (all the BV cases performed by) the recursive calls to λ_f, we obtain the desired substitution σ that we have just mentioned in claim (1). Moreover, in the same definition, the LR case simply returns a tuple of the form $\{(\Lambda, \pi \rightarrow r, id, 1)\}$ where id is the empty substitution and $\pi \rightarrow r$ is a renamed version of a program rule whose lhs π is just an instance of the term to be evaluated t, that is, there exists a substitution, say σ', such that, $t = \sigma'(\pi)$. Observe that σ' is no more than a simple matcher that only binds variables of the lhs of a (renamed version of a) program rule, in correspondence with claim (2). Note also that this matcher (which could be computed only in LR cases) is just the one we need to be composed with substitution σ (obtained when computing BV cases) in order to obtain a valid unifier (in correspondence with substitution θ in the fourth claim, which binds variables from both terms an rules) thus accomplishing with claim (4). In what follows, we modify the LR case of our definition of λ_{fuzzy} in section 3 to cope with this fact as follows:

LR (LEAF-RULE) CASE: $\lambda_f(t, \mathcal{P}_f) = \{(\Lambda, \pi \rightarrow r, \sigma', 1)\}$, where the definitional tree $\mathcal{P}_f$ is just a leaf containing a single pattern π, t is an instance of π, that is, there exists a (matcher) substitution σ' such that, $t = \sigma'(\pi)$, and $\pi \rightarrow r$ is a renamed version of a program rule.

Note that matcher σ' can be obtained by simply using syntactic methods (for instance, we can express $\sigma' = mgu(t, \pi)$ by making use of the unification algorithm proposed in [11]). It is important to note that, once again (as we explained at the beginning of this section, regarding matchers computed in BV cases), we have avoided the use of the much more involved similarity-based unification algorithm of [5, 16] that would greatly harm the efficiency of our final formulation of λ_{fuzzy}.

Example 6. Let us continue with example 5 but using now the new definition of λ_{fuzzy}. Assume that the renamed version of rules R_1, R_5 and R_6 used in the LR cases of the diagram showed at the end of Section 3, are respectively: $\mathtt{f(a,N1)} \rightarrow \mathtt{a}$,

$\mathtt{g(s(M2),N2)} \rightarrow \mathtt{s(g(M2,N2))}$ and $\mathtt{h(r(X3),Y3)} \rightarrow \mathtt{r(Y3)}$. Now, these three LR cases instead of returning tuples with empty substitutions, they provide matchers for the variables of the (renamed) program rules as follows:

$$[\mathtt{LR}] \quad \lambda_{\mathtt{f}}(\mathtt{f(a,g(a,a))},\ \mathcal{P}_{\mathtt{f}}^{\mathtt{f(a,N1)}}) = \langle \Lambda, \mathtt{R_1}, \{\mathbf{N1} \mapsto \mathbf{g(a,a)}\}, 1\rangle$$
$$[\mathtt{LR}] \quad \lambda_{\mathtt{f}}(\mathtt{g(s(X'),s(X'))},\ \mathcal{P}_{\mathtt{g}}^{\mathtt{g(s(M2),N2)}}) = \langle \Lambda, \mathtt{R_5}, \{\mathbf{M2} \mapsto \mathbf{X'}, \mathbf{N2} \mapsto \mathbf{s(X')}\}, 1\rangle$$
$$[\mathtt{LR}] \quad \lambda_{\mathtt{f}}(\mathtt{h(r(X'),s(X'))},\ \mathcal{P}_{\mathtt{h}}^{\mathtt{h(r(X3),Y3)}}) = \langle \Lambda, \mathtt{R_6}, \{\mathbf{X3} \mapsto \mathbf{X'}, \mathbf{Y3} \mapsto \mathbf{s(X')}\}, 1\rangle$$

Thus, by completing the whole diagram we finally would reach:

$$\lambda_{\mathtt{fuzzy}}(\mathtt{f(X,g(X,X))}) = \begin{pmatrix} \langle \Lambda, \mathtt{R_1}, \{\mathtt{X} \mapsto \mathtt{a}, \mathtt{N1} \mapsto \mathtt{g(a,a)}\}, 1\rangle \\ \langle 2, \mathtt{R_5}, \{\mathtt{X} \mapsto \mathtt{s(X')}, \mathtt{M2} \mapsto \mathtt{X'}, \mathtt{N2} \mapsto \mathtt{s(X')}\}, 1\rangle \\ \langle 2, \mathtt{R_6}, \{\mathtt{X} \mapsto \mathtt{s(X')}, \mathtt{X3} \mapsto \mathtt{X'}, \mathtt{Y3} \mapsto \mathtt{s(X')}\}, 0.5\rangle \end{pmatrix}$$

In Example 5 we have observed that the first two tuples can be successfully used for performing effective narrowing steps and they fulfill the four claims commented before. However, this was not the case of a tuple of the form $\tau = \langle 2, \mathtt{R_6}, \{\mathtt{X} \mapsto \mathtt{s(X')}\}, 0.5\rangle$. We have just seen that instead of τ, our new strategy produces tuple $\tau' = \langle 2, \mathtt{R_6}, \{\mathtt{X} \mapsto \mathtt{s(X')}, \mathtt{X3} \mapsto \mathtt{X'}, \mathtt{Y3} \mapsto \mathtt{s(X')}\}, 0.5\rangle$. Unfortunately, it is easy to check that τ' does not verify our second and third claims, by exactly the same reasons we commented for τ in Example 5: after applying binding $\sigma = \{\mathtt{X} \mapsto \mathtt{s(X')}, \mathtt{X3} \mapsto \mathtt{X'}, \mathtt{Y3} \mapsto \mathtt{s(X')}\}$ to the original term $t \equiv \mathtt{f(X,g(X,X))}$, we obtain $\mathtt{f(s(X'),g(s(X'),s(X')))}$, whose subterm at position 2, that is, $\mathtt{g(s(X'),s(X'))}$ is not a redex regarding (the renamed version of) rule $\mathtt{R_6}$: $\mathtt{h(r(X3),Y3)} \rightarrow \mathtt{r(Y3)}$, since we can not find a matcher σ' verifying that $\mathtt{g(s(X'),s(X'))} = \sigma'(\mathtt{h(r(X3),Y3)})$. In some sense, this fact evidences that, in contrast with the crisp case, a fuzzy narrowing step can not be seen as the combination of variable instantiation followed by a "fuzzy" rewriting step, because, among other reasons, to the best of our knowledge there exists no notion of fuzzy rewriting in the specialized literature. However, what it is really surprising is that tuple τ', in contrast with τ, just satisfies claim 4!!!. Observe that this extra ability enjoyed by τ' is related to the bindings of variables $X3$ and $Y3$ that τ was not able to generate. So, following claim 4, the effective narrowing step (which also needs to propagate the new similarity degree 0.5) can be performed by firstly replacing the subterm at position 2 in term $\mathtt{f(X,g(X,X))}$ by the rhs of rule $R_6 : \mathtt{h(r(X3),Y3)} \rightarrow \mathtt{r(Y3)}$, thus obtaining $\mathtt{f(X,r(Y3))}$, whose variables are finally instantiated by applying substitution $\{\mathtt{X} \mapsto \mathtt{s(X')}, \mathtt{X3} \mapsto \mathtt{X'}, \mathtt{Y3} \mapsto \mathtt{s(X')}\}$, thus producing the final term: $\mathtt{f(s(X'),r(s(X')))}$.

With this last refinement we have just proposed on the definition of λ_{fuzzy}, we are able now to model the richer relation of needed narrowing with similarity relations. Basically, in the following definition we take into account not only the way in which terms evolve after applying a computation step, but also, how substitutions and truth degrees are propagated along narrowing derivations. It is important to note that is the first time we formalize this new variant of fuzzy narrowing which, as also occurs with the simpler crisp case, it successfully copes with the fourth claim commented at the beginning of this section.

Definition 1 (Needed Narrowing with Similarity Relations). *Let t be a term, σ a substitution and α a similarity degree. The triple $[t, \sigma, \alpha]$ is an* state *and we denote by $\mathcal{S}$ the set of states. Given a program $\mathcal{R}$, the notion of "needed narrowing with similarity relations" step is formalized by means of an state transition system, whose transition relation $\leadsto \subseteq (\mathcal{S} \times \mathcal{S})$ is defined as:*

$$\frac{\langle p, R : (l \rightarrow r), \theta, \beta\rangle \in \lambda_{fuzzy}(t)}{[t,\sigma,\alpha] \quad \leadsto_{\theta,\beta}^{p,R} \quad [\theta(t[r]_p), \sigma \circ \theta, min(\alpha,\beta)]}$$

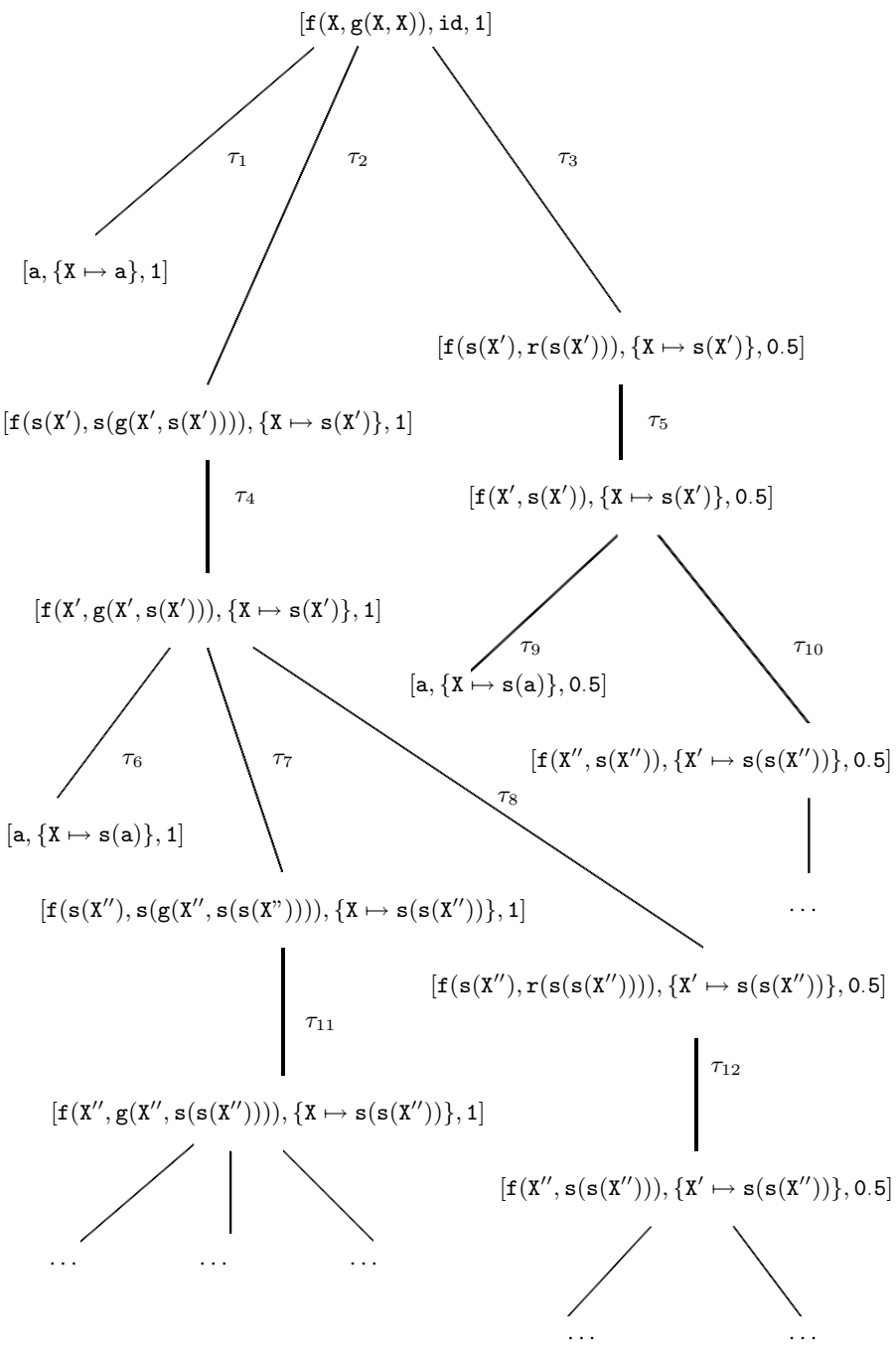


Fig. 1. A fuzzy narrowing tree for term $\mathtt{f(X, g(X, X))}$.

It t is a term to be evaluated, then we trivially generate an initial step for it with the form $[t, id, 1]$, and then we apply computation steps as much as needed following the previous definition. When we reach an state whose first component is a term s rooted by a constructor symbol, then we say that we have reached a final step of the form $[s, \theta, \alpha]$. Those final states directly correspond with the new notion of *fuzzy computed answer*, with the intended meaning that "the evaluation of a term t returns a value s when their variables are linked with the binds contained in θ, and with similarity degree α". Remember that in Section 2 we declared that in functional-logic programming, one was "interested in computing *values* (constructor or constructor-rooted terms) as well as *answers* (substitutions)", but in the new functional-fuzzy-logic programming paradigm we also need a third component of "fuzzy" information which is included on final states: the similarity degree collected as the last element on these fuzzy computed answers informs us about the minimum similarity relation exploited along a successful derivation.

Example 7. Figure 1 shows a fuzzy narrowing tree for our running example. For readablity reasons, substitutions appearing in the second component of each tree node have been restricted to variables of the term included in the first component of its parent node. Note also that each branch represents a different narrowing derivation for the original goal, whereas each leaf describes a new fuzzy computed answer for the original term which roots the tree. The children of each node (state) $[t, \sigma, \alpha]$ are obtained after applying a fuzzy narrowing step (following Definition 1) according the set of tuples computed by $\lambda_{fuzzy}(t)$. Moreover, we have labeled the arcs of the tree with the corresponding tuple used when performing the associated computational step. The reader may easily check that: $\tau_1 = \tau_6 = \tau_9 = \{\mathit{\Lambda}, \mathtt{R}_1, \{\mathtt{X} \mapsto \mathtt{a}\}, 1\}$, $\tau_2 = \tau_7 = \{2, \mathtt{R}_5, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}')\}, 1)\}$, $\tau_3 = \tau_8 = \{2, \mathtt{R}_6, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}')\}, 0.5\}$, $\tau_4 = \tau_{11} = \{1, \mathtt{R}_3, \mathtt{id}, 1\}$, $\tau_5 = \tau_{12} = \{1, \mathtt{R}_3, \mathtt{id}, 1\}$ and $\tau_{10} = \{\mathit{\Lambda}, \mathtt{R}_3, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}')\}, 1\}$.

5 Conclusions and Future Work

Starting with our first attempts in [12, 13] for fuzzifying functional-logic programming, in this paper we have 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. In contrast with the crisp case, our extended narrowing principle somehow embeds similarity-based unification, thus achieving extra capabilities for computing and propagating similarity degrees along derivations in an elegant and powerful way.

Moreover, it can be efficiently implemented by using a weak form of syntactic unification: instead of computing most general unifiers [11], we produce more effective unifiers by simple composing those matchers easily obtained by λ_{fuzzy} in LR and BV cases when exploring definitional trees. And what is the best, instead of capturing similarity degrees with the much more involved similarity-based unification method of [5, 16], we simply obtain them with a lower cost by exploiting similarities between constructor symbols and defined function symbols, when solving BC and BF cases, respectively. These actions are modeled in a

coherent and natural way when extending the original needed narrowing principle of Curry, also inheriting their originality and optimal properties in the fuzzy setting. The next step in our ongoing research is centered in the development of a theoretical framework defining a declarative/denotational semantics (based on fuzzy sets) for the new paradigm and the corresponding correctness/completenes proofs. Moreover, we also plan to connect our techniques with other ones related with term indexing, SLDE-resolution, etc.

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 Symposium on Applied Computing (SAC'99)*, pages 260–267. ACM, Artificial Intelligence and Computational Logic, 1999.
4. P. Bosco, E. Giovannetti, and C. Moiso. Refined Strategies for semantic unification. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. of TAPSOFT'87*, pages 276–290. Springer LNCS 250, 1987.
5. Ferrante Formato, Giangiacomo Gerla, and Maria I. Sessa. Similarity-based Unification. *Fundamenta Informaticae*, 41(4):393–414, 2000.
6. L. Garmendia and A. Salvador. Comparing transitive closure with other new T-transivization methods. In *Proc. Modeling Decisions for Artificial Intelligence*, pages 306–315. Springer LNAI 3131, 2004.
7. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
8. G. Huet and J.J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443. The MIT Press, Cambridge, MA, 1992.
9. J.M. Hullot. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.
10. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987.
11. A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
12. 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 Conference on Fuzzy Logic and Applications, WILF'2005. Crema, Italy, September 15-17*, pages 89–98. Springer LNAI 3849, 2005.
13. G. Moreno and V. Pascual. Formal Properties of Needed Narrowing with Similarity Relations. In P. Lucio, editor, *Electronic Notes in Theoretical Computer Science*, page 15. Elsevier (In press), 2007.
14. H.T. Nguyen and E.A. Walker. *A First Course in Fuzzy Logic*. Chapman & Hall/CRC, Boca Ratón, Florida, 2000.
15. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.
16. M.I. Sessa. Approximate Reasoning by Similarity-based SLD Resolution. *Fuzzy Sets and Systems*, 275:389–426, 2002.
17. L. A. Zadeh. Similarity relations and fuzzy orderings. *Informa. Sci.*, 3:177–200, 1971.