The 28th North American Fuzzy Information Processing Society Annual Conference (NAFIPS2009)
Cincinnati, Ohio, USA - June 14 - 17, 2009

RFuzzy—A Framework for Multi-adjoint Fuzzy Logic Programming

Victor Pablos Ceruelo
Babel Group
Technical University of Madrid, Spain
e-mail: vpablos@fi.upm.es

Hannes Strass
Computational Logic Group
Technical University of Dresden, Germany
e-mail: hannes.strass@inf.tu-dresden.de

Susana Munoz-Hernandez
Babel Group
Technical University of Madrid, Spain
e-mail: susana@fi.upm.es

***Abstract*—Fuzzy Logic Programming aims at combining the advantages of Logic Programming (such as readability, conciseness, and a formally well-defined semantics) with the advantages of Fuzzy Logic (representability of imprecise and uncertain knowledge).**

In this paper, we present the *RFuzzy* framework for Fuzzy Logic Programming. It has three main advantages compared to other Fuzzy Logic Programming frameworks: *RFuzzy* provides constructive answers (it responds with direct results instead of constraints), models multi-adjoint logic, and allows the user[1] to combine fuzzy and crisp reasoning in the same program. It provides some extensions such as default values (to represent missing information) and typed predicates. The truth values of predicates are defined via facts, rules, and functions.

We describe the implementation of our framework and its operational semantics. *RFuzzy* has been implemented and is ready for being used[2].

I. Introduction

Logic Programming is a declarative programming paradigm that has been successfully used in the field of knowledge representation and reasoning for many years. But the classical Logic Programming languages (e.g. Prolog) lack an intuitive (and user-friendly) way to represent vague world information, that is, information that is either imprecise or imperfect, or both.

The result of introducing Fuzzy Logic into Logic Programming has been the development of several fuzzy systems over Prolog. These systems replace the inference mechanism of Prolog, SLD-resolution, with a fuzzy variant that is able to handle partial truth.

Most of these systems implement the fuzzy resolution introduced by Lee in [1], as the Prolog-Elf system [2], the FRIL Prolog system [3], and the F-Prolog language [4]. However, there is no common method for fuzzifying Prolog, as has been noted in [5].

In the present paper, we introduce an expressive, yet simple-to-use framework for fuzzy Logic Programming, RFuzzy. The *R* in the name derives from it representing truth values as *R*eal numbers from the unit interval.

The remainder of the paper is organized as follows. The following subsection compares RFuzzy with other Fuzzy Logic programming frameworks. The next section introduces the syntax of RFuzzy programs. Section III introduces the formal operational semantics underlying the RFuzzy Framework. The last section mentions aspects of current work and concludes.

A. RFuzzy in Comparison

One of the most promising fuzzy tools for Prolog was the "Fuzzy Prolog" system [6], [7], as it combined three important characteristics for modelling fuzziness: 1*) A truth value is a finite union of sub-intervals on $[0,1]$ [3]. 2*) A truth value is propagated through the rules by means of an *aggregation operator*. The definition of this *aggregation operator* is general and it subsumes conjunctive operators (triangular norms [8] like min, prod, etc.), disjunctive operators [9] (triangular co-norms, like max, sum, etc.), average operators (averages as arithmetic average, quasi-linear average, etc.), and hybrid operators (combinations of the above operators [10]). 3*) Crisp and fuzzy reasoning are consistently combined [11].

If we compare RFuzzy with "Fuzzy Prolog", we can see that 1*) It uses real numbers instead of unions of intervals between real numbers to represent truth values. Answers like $v = [0,\ 1] \cap [0,\ 1]$ mean that the program can not conclude anything about the variable truth value. To distinguish between having no information and using default information RFuzzy will fail to answer in this case, so we can infer that no information in available. 2*) It offers the user a concrete syntax to define types, so the user does not need to code them in Prolog code. In "Fuzzy Prolog" we have to code types taking care of the code translation it does, and this introduces a lot of errors. 3*) It does not answer user queries using constraints, so its answers can be used as input in other programs, like web applications. Constructive answers are always returned, via the application of constraints like $v = [X, Y], X \geq 0, X \leq 1, Y \geq 0, Y \leq 1$ to the truth values of the individuals defined by means of types. 4*) Truth value variables do not need to be coded. Taking care of variables

This work is partially supported by the project DESAFIOS - TIN 2006-15660-C02-02 from the Spanish Ministry of Education and Science, by the Spanish Ministry of Science and Innovation Research Staff Training Program - BES-2008-008320 and by the project PROMESAS - S-0505/TIC/0407 from the Madrid Regional Government.

[1]We refer as 'user' to the programmer that wants to represent a fuzzy problem in a programming framework to make queries and obtain results.

[2]The RFuzzy module with installation instructions and examples can be downloaded from http://babel.ls.fi.upm.es/software/rfuzzy/.

[3]An interval is a particular case of union of one element, and a unique truth value is a particular case of having an interval with only one element.

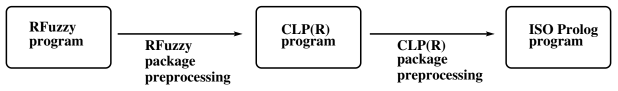

Fig. 1. RFuzzy architecture

to manage the predicates' truth value introduces errors and makes the code illegible, without giving any advantage. 5*) Default truth values' functionality is increased by conditional default truth values, which are suitable for defining a default truth value for subsets of individuals.

RFuzzy shares with Fuzzy Prolog most of its nice expressive characteristics: Prolog-like syntax (based on using facts and clauses), use of any aggregation operator, flexibility of query syntax, definition of credibility in rules, etc. Both of them add fuzziness to a Prolog compiler using CLP($\mathcal{R}$) instead of implementing a new fuzzy resolution, as other fuzzy Prologs do. CLP($\mathcal{R}$) enables them to represent intervals of truth values and truth values as constraints over real numbers and *aggregation operators* as operations with these constraints, so they use Prolog's built-in inference mechanism to handle the concept of partial truth. Besides, its use enables portability between ISO Prolog systems, so code transformations performed by RFuzzy and CLP($\mathcal{R}$) packages convert RFuzzy code into ISO Prolog code. Fig. 1 shows the whole process.

There is another fuzzy Logic Programming system that models multi-adjoint logic, FLOPER [12]. It has a Logic-Programming-inspired syntax and provides free choice of aggregation operators and credibility of rules just as RFuzzy does. There are however some things that FLOPER cannot do: deal with missing information (which RFuzzy does by default truth value declarations), type atoms and predicates to give constructive answers, and provide syntactic sugar to express truth value functions. RFuzzy will be explained in greater detail in the following section.

II. RFuzzy Syntax

In this section we describe the syntax of new predicates added to Prolog by RFuzzy to deal with fuzziness.

A. Type Definitions

Prolog code consists of predicate definitions and it is run via posing a query that is subsequently tried to be proven, so we can say that Prolog does not have types. In RFuzzy we use an extension to restrict the predicates' domains. This extension is named "types" and its syntax is shown in (1).

$$\textbf{:- set\_prop}\ pred/ar \ \textbf{=>}\ t\_pred\_1/1\ [,\ t\_pred\_2/1\ ]^*\ . \quad (1)$$

where *set_prop* is a reserved word, *pred* is the name of the typed predicate, *ar* is its arity and *t_pred_*$\{n\}$ is the predicate used to assure that the value given to the argument in the position *n* of a call to *pred/ar* is correctly typed. The predicate *t_pred_*$\{n\}$ must have arity 1. In the example below we define that the valid values for the argument of the predicate *nice_weather/1* are the ones defined by the predicate *city/1*: Madrid, Moscow, and Berlin.

```
:- set_prop nice_weather/1 => city/1.

city(madrid).
city(moscow).
city(berlin).
```

B. Fact Truth Values

Fuzzy facts are facts to which we assign a truth value. To code them in programs we offer a special syntax, shown in (2), so Prolog can distinguish between crisp and fuzzy facts.

$$pred(args)\ \textbf{value}\ truth\_val. \quad (2)$$

Arguments (*args*) should be ground terms and the truth value (*truth_val*) must be a real number between 0 and 1. The example below defines (among other facts) that the city *madrid* has *nice_weather* with a truth value of 0.8.

```
nice_weather(madrid) value 0.8.
nice_weather(moscow) value 0.2.
many_sights(madrid) value 0.6.
```

C. Functional Truth Values

For some fuzzy concepts, it is very straightforward to define truth values as function values of a parameter. For the concept *tall*, for example, we could define the truth value of a person being tall as a function of their body height. Fig. 2 shows an example in which the truth value function assigns the truth value of being *far_away* from the city's distance (in kilometers) to the user's hometown.

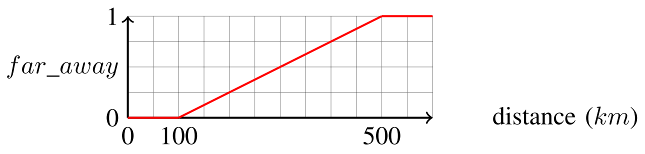

Fig. 2. Truth value of being *far_away* for a given distance (in kms) from the hometown.

Functional truth values are syntactic sugar to be able to code such functions. As those functions are usually continuous and linear over intervals, the simplest way to define them is by means of their inflexion points, using interpolation to determine values between them.

Inflexion points are coded as a list of ordered pairs whose first element is the crisp value and the second element is the truth value assigned to it. The syntax is shown in (3), where external brackets represent the Prolog list symbols and internal brackets represent cardinality in the formula notation. *arg1, ..., argN* should be ground terms (numbers) and *truth_val1, ..., truth_valN* should be border truth values (real numbers between 0 and 1, inclusive them).

$$pred\ \textbf{:\#}\ ([\ \ (arg1, truth\_val1),\ (arg2, truth\_val2)\ \ [,\ (arg3, truth\_val3)\ ]^*\ \ ])\ . \quad (3)$$

The example below shows the code for the function shown in Figure 2. We only need to code up to 500 kms, for greater distances the default truth value of 1 is used. See Subsection II-E for more information on default truth values.

```
:- set_prop far_away/1 => city/1.
:- default(far_away/1, 1).
far_away :# ([ (0,0), (100,1), (500,1) ]).
```

D. Rule Truth Values

A rule is the tool we offer the user to combine the truth values of facts, functions, and other rules by means of aggregation operators like *maximum* or *product*. Besides this combination truth value for the body of the rule, the rule can be given an overall credibility truth value.

This credibility is used to express how much we trust the rule, and it is combined with the resultant truth value from the body of the rule by means of an aggregation operator.

Syntax for defining rule truth values is defined in (4), where the user can choose two aggregation operators [4]: *op2* for combining the truth values of the subgoals of the rule's body and *op1* for combining the previous result with the credibility of the rule. Brackets represent optional fields.

$$pred(arg1\ [,\ arg2]^*\ )\ [\ \textbf{cred}\ (op1, value1)\ ]\ \textbf{:}\sim \\ op2\ pred1(\ arg1\ [,\ arg2]^*\ ) \\ [,\ pred2(\ arg1\ [,\ arg2]^*\ )\ ]\ . \quad (4)$$

The following example shows its usage. As it can be seen in it, it is not mandatory to code the rule's credibility when we don't need it.

```
good_destination(X) :~ prod
        nice_weather(X),
        many_sights(X).
```

E. General and Conditioned Default Truth Values

Unfortunately, information that is provided by the user is not complete in general. So there are many cases in which we have no information about the truth value of an individual or a set of them. Nevertheless, it is interesting not to stop a complex query evaluation just because we have no information about one or more subgoals if we can use a reasonable approximation. The solution to this problem is using default truth values for these cases.

General truth value (Syntax defined in (5)) is an special case of Conditioned default truth values where the condition to fulfil by the individuals is always true. *Conditioned default truth value* (Syntax defined in (6)) assigns a default truth value to a subset of individuals of the predicate's domain (this predicate's domain is defined by means of *type definitions* (see subsection II-A) that fulfil a condition, defined by the membership predicate (*mship_pred/ar*).

[4]Aggregation operators available are: *min* for minimum, *max* for maximum, *prod* for the product, *luka* for the Lukasiewicz operator, *dprod* for the inverse product and *dluka* for the inverse Lukasiewicz operator.

pred/ar is in both cases the predicate to which we are defining default values and *tv* their respective truth value.

When defining more than one truth value (explicit, conditioned, and/or default truth value) only one will be given back when doing a query. The precedence when looking for the truth value goes from most to least concrete.

$$\textbf{:- default(}pred/ar,\ tv\textbf{)}\ \textbf{.} \quad (5)$$

$$\textbf{:- default(}pred/ar,\ tv\textbf{)}\ \textbf{=>}\ mship\_pred/ar\textbf{.} \quad (6)$$

In the example below we define that all countries not having a explicit truth value for *mediterranean_diet* that are in the *mediterranean_country* set will have a truth value of 0.8 and those that are not in the set will have a truth value of 0.1. Last two lines are part of the big example we use in the section III.

```
:- default(mediterranean_diet/1, 0.1).
:- default(mediterranean_diet/1, 0.8)
        => mediterranean_country/1.

mediterranean_country(spain).
mediterranean_country(italy).
mediterranean_country(greece).
% And so on (Up to 16 countries).

:- default(nice_weather/1, 0.5).
:- default(many_sights/1, 0.2).
```

F. Queries and Constructive Answers

During compilation, *RFuzzy* adds a new argument to the arguments list of each fuzzy predicate. This argument serves for querying the predicate's truth value, and is always added in the same way, at the end of the arguments list of the predicate. It can be seen as syntactic sugar, as the predicate truth value is not part of its arguments, but metadata information. In the previous example we coded *good_destination/1*, so to query the system we have to give the predicate two arguments instead of only one. The second one will represent the query's truth value. This can be seen in the example below.

```
?- good_destination(moscow, V).
V = 0.04 ?
yes

?- good_destination(D, V), V > 0.4.
D = madrid,
V = 0.48 ? ;
no
```

Of course all the fuzzy tools mentioned before are able to provide constructive answers for the first query, but none of them are able to provide answers to the second query. The regular (easy) questions are asking for the truth value of an element, but the really interesting queries are the ones that ask for values that satisfy constraints over the truth value, as can be seen in the second query.

III. Operational semantics

In this section, we explain the formal syntax and semantics of the language that underlies RFuzzy.

We will use a signature Σ of function symbols and a set of variables V to "build" the *term universe* $\mathrm{TU}_{\Sigma,V}$ (whose elements are the *terms*). It is the minimal set such that each variable is a term and terms are closed under Σ-operations. In particular, constant symbols are terms.

Similarly, we will use a signature Π of predicate symbols to define the *term base* $\mathrm{TB}_{\Pi,\Sigma,V}$ (whose elements are called *atoms*). Atoms are predicates whose arguments are elements of $\mathrm{TU}_{\Sigma,V}$. Atoms and terms are called *ground* if they do not contain variables. As usual, the *Herbrand universe H* is the set of all ground terms, and the *Herbrand base B* is the set of all atoms with arguments from the Herbrand universe.

To combine truth values in the set of real truth values $[0,1]$, we will make use of *aggregation operators*. A function $\hat{F} : [0,1]^n \to [0,1]$ is called an aggregation operator if it verifies $\hat{F}(0,\ldots,0) = 0$ and $\hat{F}(1,\ldots,1) = 1$. We will use the signature Ω to denote the set of used operator symbols F and $\hat{\Omega}$ to denote the set of their associated aggregation operators $\hat{F}$. An n-ary aggregation operator is called *monotonic in the i-th argument*, if additionally $x \le x'$ implies $\hat{F}(x_1,\ldots,x_{i-1},x,x_{i+1},\ldots,x_n) \le \hat{F}(x_1,\ldots,x_{i-1},x',x_{i+1},\ldots,x_n)$. An aggregation operator is called *monotonic* if it is monotonic in all arguments.

Immediate examples for aggregation operators that come to mind are typical examples of t-norms and t-conorms: minimum $\min(a,b)$, maximum $\max(a,b)$, product $a \cdot b$, and probabilistic sum $a + b - a \cdot b$. The above general definition of aggregation operators subsumes however all kinds of minimum, maximum or mean operators.

Definition. Let Ω be an aggregation operator signature, Π a predicate signature, Σ a term signature, and V a set of variables.

A *fuzzy clause* is written as

$$A \overset{c,F_c}{\longleftarrow}_F B_1,\ldots,B_n$$

where $A \in \mathrm{TB}_{\Pi,\Sigma,V}$ is called the head, $B_1,\ldots,B_n \in \mathrm{TB}_{\Pi,\Sigma,V}$ is called the body, $c \in [0,1]$ is the credibility value, and $F_c \in \Omega^{(2)}$ and $F \in \Omega^{(n)}$ are aggregation operator symbols (for the credibility value and the body resp.)

A *fuzzy fact* is a special case of a clause where $n = 0$, $c = 1$, F_c is the usual multiplication of real numbers "$\cdot$" and $F = v \in [0,1]$. It is written as $A \leftarrow v$.

A *fuzzy query* is a pair $\langle A, v\rangle$, where $A \in \mathrm{TB}_{\Pi,\Sigma,V}$ and v is either a "new" variable that represents the initially unknown truth value of A or it is a concrete value $v \in [0,1]$ that is asked to be the truth value of A. ⌟

Intuitively, a clause can be read as a special case of an implication: we combine the truth values of the body atoms with the aggregation operator associated to the clause to yield the truth value for the head atom. For this truth value calculation we are completely free in the choice of an operator.

In standard logic programming, the closed-world assumption is employed, i.e. the knowledge base is not only assumed to be sound but moreover to be complete. Everything that can not be derived from the knowledge is assumed to be false. This could be easily modelled in this framework by assuming the truth value 0 as "default" truth value, so to speak. Yet we want to pursue a slightly more general approach: arbitrary default truth values will be explicitly stated for each predicate. We even allow the definition of different default truth values for different arguments of a predicate. This is formalized as follows.

Definition. A *default value declaration* for a predicate $p \in \Pi^{(n)}$ is written as $\texttt{default}(p(X_1,\ldots,X_n)) = [\delta_1 \text{ if } \varphi_1,\ldots,\delta_m \text{ if } \varphi_m]$ where $\delta_i \in [0,1]$ for all i. The φ_i are first-order formulas restricted to terms from $\mathrm{TU}_{\Sigma,\{X_1,\ldots,X_n\}}$, the predicates $=$ and $\neq$, the symbol true, and the junctors $\wedge$ and $\vee$ in their usual meaning. ⌟

Types can be viewed as inherent properties of terms – each term can have zero or more types. We use them to restrict the domains of predicates.

Definition. A *term type declaration* assigns a type $\tau \in \mathcal{T}$ to a term $t \in H$ and is written as $t : \tau$. A *predicate type declaration* assigns a type $(\tau_1,\ldots,\tau_n) \in \mathcal{T}^n$ to a predicate $p \in \Pi^n$ and is written as $p : (\tau_1,\ldots,\tau_n)$, where τ_i is the type of p's i-th argument. ⌟

For a ground atom $A = p(t_1,\ldots,t_n) \in B$ we say that it is *well-typed with respect to T* iff $p : (\tau_1,\ldots,\tau_n) \in T$ implies $\tau_i \in \mathfrak{t}_T(t_i)$ for all i.

For a ground clause $A \overset{c,F_c}{\longleftarrow}_F B_1,\ldots,B_n$ we say that it is well-typed w.r.t. T iff all B_i are well-typed for $1 \le i \le n$ implies that A is well-typed (i.e. if the clause preserves well-typing). We say that a non-ground clause is well-typed iff all its ground instances are well-typed.

A *fuzzy logic program P* is a triple $P = (R, D, T)$ where R is a set of fuzzy clauses, D is a set of default value declarations, and T is a set of type declarations.

From now on, when speaking about programs, we will implicitly assume the signature Σ to consist of all function symbols occurring in P, the signature Π to consist of all the predicate symbols occurring in the program, the set $\mathcal{T}$ to consist of all types occurring in type declarations in T, and the signature Ω of all the aggregation operator symbols. For Ω we will furthermore require that all operators from $\hat{\Omega}$ be monotonic.

Lastly, we introduce the important notion of a "well-defined" program.

Definition. A fuzzy logic program $P = (R, D, T)$ is called *well-defined* iff

- for each predicate symbol p/n occurring in R, there exist both a predicate type declaration and a default value declaration;
- all clauses in R are well-typed;
- for each default value declaration $\texttt{default}(p(X_1,\ldots,X_n)) = [\delta_1 \text{ if } \varphi_1,\ldots,\delta_m \text{ if } \varphi_m]$, the formulas φ_i are pairwise contradictory and $\varphi_1 \vee \cdots \vee \varphi_m$ is a tautology, i.e. exactly one default truth value applies to each element of p/n's domain.

$\lrcorner$

The operational semantics will be formalized by a transition relation that operates on (possibly only partially instantiated) computation trees. Here, we will not need to keep track of default value attributes $\{\blacktriangle, \blacklozenge, \blacktriangledown\}$ explicitly, it will be encoded into the computations.

Definition. Let Ω be a signature of aggregation operator symbols and W a set of variables with $W \cap V = \varnothing$.

A *computation node* is a pair $\langle A, e\rangle$, where $A \in \mathrm{TB}_{\Pi,\Sigma,V}$ and e is a term over $[0,1]$ and W with function symbols from Ω. We say that a computation node is *ground* if e does not contain variables. A computation node is called *final* if $e \in [0,1]$. A final computation node will be indicated as $\underline{\langle A, e\rangle}$.

We distinguish two different types of computation nodes: C-nodes, that correspond to applications of program clauses, and D-nodes, that correspond to applications of default value declarations.

A *computation tree* is a directed acyclic graph whose nodes are computation nodes and where any pair of nodes has a unique (undirected) path connecting them. We call a computation tree ground or final if all its nodes are ground or final respectively.

For a given computation tree t we define the *tree attribute*

$$z_t = \begin{cases} \blacktriangledown & \text{if } t \text{ contains no } \mathsf{D}\text{-node} \\ \blacklozenge & \text{if } t \text{ contains both } \mathsf{C}\text{- and } \mathsf{D}\text{-nodes} \\ \blacktriangle & \text{if } t \text{ contains only } \mathsf{D}\text{-nodes} \end{cases}$$

$\lrcorner$

Computation nodes are essentially generalizations of queries that keep track of aggregation operator usage.

Computation trees as defined here should not be confused with the usual notion of SLD-trees. While SLD-trees describe the whole search space for a given query and thus give rise to different derivations and different answers, computation trees describe just a state in a single computation.

The computation steps that we perform on computation trees will be modelled by a relation between computation trees.

Definition.[Transition relation] For a given fuzzy logic program $P = (R, D, T)$, the transition relation $\rightharpoonup$ is characterized by the following transition rules:

Clause: $t/\left[\ \langle A', v\rangle\ \right] \rightharpoonup$

$$t/\left[\ \langle A', v\rangle \quad / \quad \begin{tikzcd} & \mathsf{C}\langle A', F_c(c, F(v_1,\dots,v_n))\rangle \arrow[dl, dash] \arrow[dr, dash] & \\ \langle B_1, v_1\rangle & \cdots & \langle B_n, v_n\rangle \end{tikzcd}\ \right]\mu$$

If there is a (variable disjoint instance of a) program clause $A \overset{c,F_c}{\longleftarrow}_F B_1, \dots, B_n \in R$ and $\mu = \mathrm{mgu}(A', A)$. (Take a non-final leaf node and add child nodes according to a program clause; apply the most general unifier of the node atom and the clause head to all the atoms in the tree.)

Note that we immediately finalize a node when applying this rule for a fuzzy fact.

Default: $t\,[\langle A, x\rangle] \quad \rightharpoonup \quad t\left[\langle A, x\rangle / \mathsf{D}\underline{\langle A, \delta_j\rangle}\right]\mu$

If A does not match with any program clause head, there is a default value declaration $\texttt{default}(p(X_1,\dots,X_n)) = [\delta_1 \text{ if } \varphi_1, \dots, \delta_m \text{ if } \varphi_m] \in D$, μ is a substitution such that $p(X_1,\dots,X_n)\mu = A\mu$ is a well-typed ground atom, and there exists a $1 \le j \le m$ such that $\varphi_j\mu$ holds. (Apply a default value declaration to a non-final leaf node thus finalizing it.)

Finalize:

$$\begin{tikzcd} & \mathsf{C}\langle A, F_c(c, F(v_1,\dots,v_n))\rangle \arrow[dl, dash] \arrow[dr, dash] & \\ \underline{\langle B_1, v_1\rangle} & \cdots & \underline{\langle B_n, v_n\rangle} \end{tikzcd} \quad \rightharpoonup$$

$$\begin{tikzcd} & \mathsf{C}\underline{\langle A, \hat{F}_c(c, \hat{F}(v_1,\dots,v_n))\rangle} \arrow[dl, dash] \arrow[dr, dash] & \\ \underline{\langle B_1, v_1\rangle} & \cdots & \underline{\langle B_n, v_n\rangle} \end{tikzcd}$$

(Take a non-final node whose children are all final and replace its truth expression by the corresponding truth value.)

$\lrcorner$

Here, the notation $t[A]$ means "the tree t that contains the node A somewhere". Likewise, $t[A/B]$ is to be read as "the tree t where the node A has been replaced by the node B".

Asking the query $\langle A, v\rangle$ corresponds to applying the transition rules to the initial computation tree $\langle A, v\rangle$. The computation ends *successfully* if a final computation tree is created, the truth value of the instantiated query can then be read off the root node. We will illustrate this with an example computation.

Example. We start with the tree $\langle\text{good-destination(Y)}, v\rangle$. Applying the **Clause**-transition to the initial tree with the program clause good-destination(X)$\overset{1.0,\cdot}{\longleftarrow}_{\cdot}$nice-weather(X), many-sights(X) yields

$$\begin{tikzcd} & \mathsf{C}\langle\text{good-destination(Y)}, 1.0\cdot v_1\cdot v_2\rangle \arrow[dl, dash] \arrow[dr, dash] & \\ \langle\text{nice-weather(Y)}, v_1\rangle & & \langle\text{many-sights(Y)}, v_2\rangle \end{tikzcd}$$

Now we apply **Clause** to the left child with nice-weather(moscow) $\leftarrow$ 0.2:

$$\begin{tikzcd} & \mathsf{C}\langle\text{good-destination(moscow)}, 1.0\cdot 0.2\cdot v_2\rangle \arrow[dl, dash] \arrow[dr, dash] & \\ \mathsf{C}\underline{\langle\text{nice-weather(moscow)}, 0.2\rangle} & & \langle\text{many-sights(moscow)}, v_2\rangle \end{tikzcd}$$

Since there exists no clause whose head matches many-

$\mathsf{sights(moscow)}$, we apply the **Default**-rule for $\mathsf{many\text{-}sights}$ to the right child.

$$\begin{array}{c} \mathsf{C}\langle \mathsf{good\text{-}destination(moscow)}, 1.0 \cdot 0.2 \cdot 0.2\rangle \\ \hline \mathsf{C}\langle \underline{\mathsf{nice\text{-}weather(moscow)}, 0.2}\rangle \quad \mathsf{D}\langle \underline{\mathsf{many\text{-}sights(moscow)}, 0.2}\rangle \end{array}$$

In the last step, we finalize the root node.

$$\begin{array}{c} \mathsf{C}\langle \underline{\mathsf{good\text{-}destination(moscow)}, 0.04}\rangle \\ \hline \mathsf{C}\langle \underline{\mathsf{nice\text{-}weather(moscow)}, 0.2}\rangle \quad \mathsf{D}\langle \underline{\mathsf{many\text{-}sights(moscow)}, 0.2}\rangle \end{array}$$

The calculated truth value for $\mathsf{good\text{-}destination(moscow)}$ is thus 0.04. $\diamond$

The actual operational semantics is now given by the truth values that can be derived in the defined transition system. This "canonical model" can be seen as a generalization of the success set of a program.

Definition. Let P be a well-defined fuzzy logic program. The canonical model of P for $A \in B$ is defined as follows:

$$\mathsf{cm}(P) :=$$
$$\left\{ A \mapsto z_t v \;\middle|\; \begin{array}{l} \text{there exists a computation starting} \\ \text{with } \langle A, w\rangle \text{ and ending with a final} \\ \text{computation tree } t \text{ with root node} \\ \underline{\langle A, v\rangle} \end{array} \right\}$$

$\lrcorner$

It can be verified that the canonical model $\mathsf{cm}(P)$ is indeed a model of P.

IV. Conclusions and Current Work

RFuzzy offers to the users a new framework to represent fuzzy problems over real numbers. It has some advantages over other fuzzy prolog approaches, like *Fuzzy Prolog* [6], [7], [13] or FLOPER [12]. They are a simpler syntax, the elimination of answers with constraints by means of constructive answers and the introduction of conditioned default truth values.

RFuzzy syntax is simpler because its fuzzy values are real numbers instead of intervals between real numbers and it hides the management of truth value variables. As normal fuzzy problems do not use intervals to represent fuzziness and do not need to code an uncommon behavior of fuzzy variables, this syntax reduction is an advantage. Programs written in RFuzzy syntax are more legible and easier to be understood than programs written in other fuzzy frameworks.

Answers to user in other fuzzy prolog frameworks are constraints, and it is difficult (and sometimes impossible) use them in other programs (like web applications) due to the management of them. RFuzzy eliminates constraints in answers by giving to the user the ability to define types, enabling constructivity in answers (See subsection II-F).

There is also an extension to introduce default truth values. As world information is sometimes incomplete, RFuzzy offers to the user the possibility to define default truth values and default conditioned truth values (see subsection II-E). This allows us to make inference with default truth values when we do not know anything about the truth of some fact.

There are countless applications and research lines which can benefit from the advantages of using the fuzzy representations offered by RFuzzy. Some examples are: search engines, knowledge extraction (from databases, ontologies, etc.), the Semantic Web, business rules, and coding rules (where the violation of one rule can be given a truth value).

Current work on RFuzzy tries to implement dynamical truth values, so truth values can be loaded in the program from a database, enabling its use for dynamical applications in the web. Semantic Web is one of this applications which should be improved using fuzzy reasoning to represent fuzzy relations between objects, so the world they model actually is crisp. Another problem we are dealing with is that "Tweety is a bird" with a truth value of 1.0 but it flies with a truth value of 0 - non-monotonic reasoning. It can be deduced from here that in our programs defined truth values remove this kind of inconsistencies, but not all inconsistencies are removed. And in the end (but not less important) we are trying to apply constructive negation to the engine, so we can ask not only which cities are suitable to visit, (see the example) but which ones are not suitable to visit.

References

[1] R. C. T. Lee, "Fuzzy Logic and the resolution principle," *Journal of the Association for Computing Machinery*, vol. 19, no. 1, pp. 119–129, 1972.

[2] M. Ishizuka and N. Kanai, "Prolog-ELF incorporating fuzzy Logic," in *IJCAI*, 1985, pp. 701–703.

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

[4] D. Li and D. Liu, *A Fuzzy Prolog Database System*. New York: John Wiley & Sons, 1990.

[5] Z. Shen, L. Ding, and M. Mukaidono, "Fuzzy resolution principle," in *Proc. of 18th International Symposium on Multiple-valued Logic*, vol. 5, 1989.

[6] C. Vaucheret, S. Guadarrama, and S. Munoz-Hernandez, "Fuzzy prolog: A simple general implementation using clp(r)," in *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2002*, ser. LNAI, M. Baaz and A. Voronkov, Eds., no. 2514. Tbilisi, Georgia: Springer-Verlag, October 2002, pp. 450–463.

[7] S. Guadarrama, S. Munoz-Hernandez, and C. Vaucheret, "Fuzzy Prolog: A new approach using soft constraints propagation," *Fuzzy Sets and Systems, FSS*, vol. 144, no. 1, pp. 127–150, 2004, iSSN 0165-0114.

[8] E. Klement, R. Mesiar, and E. Pap, "Triangular norms," Kluwer Academic Publishers.

[9] E. Trillas, S. Cubillo, and J. L. Castro, "Conjunction and disjunction on $([0,1], <=)$," *Fuzzy Sets and Systems*, vol. 72, pp. 155–165, 1995.

[10] A. Pradera, E. Trillas, and T. Calvo, "A general class of triangular norm-based aggregation operators: quasi-linear t-s operators," *International Journal of Approximate Reasoning*, vol. 30, no. 1, pp. 57–72, 2002.

[11] S. Munoz-Hernandez, C. Vaucheret, and S. Guadarrama, "Combining crisp and fuzzy Logic in a prolog compiler," in *Joint Conf. on Declarative Programming: APPIA-GULP-PRODE 2002*, J. J. Moreno-Navarro and J. Mariño, Eds., Madrid, Spain, September 2002, pp. 23–38.

[12] G. Moreno, "Building a fuzzy transformation system." in *SOFSEM*, 2006, pp. 409–418.

[13] C. Vaucheret, S. Guadarrama, and S. Munoz-Hernandez, "Fuzzy prolog: A simple general implementation using clp(r)," in *Int. Conf. in Logic Programming, ICLP 2002*, ser. LNCS, P. Stuckey, Ed., no. 2401. Copenhagen, Denmark: Springer-Verlag, July/August 2002, p. 469.