Rfuzzy framework

Victor Pablos Ceruelo, Susana Munoz-Hernandez, and Hannes Strass

Universidad Politécnica de Madrid [**]
{vpablos,susana} at fi.upm.es, hannes.strass at alumnos.upm.es
http://babel.ls.fi.upm.es/

Abstract. Fuzzy reasoning is a very productive research field that during the last years has provided a number of theoretical approaches and practical implementation prototypes. Nevertheless, the classical implementations, like Fril, are not adapted to the latest formal approaches, like multi-adjoint logic semantics.
Some promising implementations, like Fuzzy Prolog, are so general that the regular user/programmer does not feel comfortable because either representation of fuzzy concepts is complex or the results difficult to interpret.
In this paper we present a modern framework, *Rfuzzy*, that is modelling multi-adjoint logic. It provides some extensions as default values (to represent missing information, even partial default values) and typed variables. Rfuzzy represents the truth value of predicates through facts, rules and functions. Rfuzzy answers queries with direct results (instead of constraints) and it is easy to use for any person that wants to represent a problem using fuzzy reasoning in a simple way (by using the classical representation with real numbers).

Key words: Fuzzy reasoning, Implementation tool, Fuzzy Logic, Multi-adjoint logic, Logic Programming Application

1 Introduction

One of the reasoning models that is more useful to represent real situations is fuzzy reasoning. Indeed, world information is not represented in a crisp way. Its representation is imperfect, fuzzy, etc., so that the management of uncertainty is very important in knowledge representation. There are multiple frameworks for incorporating uncertainty in logic programming:

– fuzzy set theory [5, 38, 32],
– probability theory [8, 16, 20, 25, 26],
– multi-valued logic [7, 11, 12, 15, 17, 29],
– possibilistic logic [6, 36, 37]

[**] This work is partially supported by the project DESAFIOS - TIN 2006-15660-C02-02 from the Spanish Ministry of Education and Science and project PROMESAS - S-0505/TIC/0407 from the Madrid Regional Government.

Despite of the multitude of theoretical approaches to this issue, few of them resulted in actual practically usable tools. Since Logic Programming is traditionally used in Knowledge Representation and Reasoning, we argue it is perfectly well-suited to implement a fuzzy reasoning tool as ours.

1.1 Fuzzy Approaches in Logic Programming

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 [18], as the Prolog-Elf system [10], the FRIL Prolog system [2] and the F-Prolog language [19]. However, there is no common method for fuzzifying Prolog, as has been noted in [28].

Some of these Fuzzy Prolog systems only consider the predicates' fuzziness whereas other systems consider fuzzy facts or fuzzy rules. There is no agreement about which fuzzy logic should be used. Most of them use min-max logic (for modelling the conjunction and disjunction operations) but other systems just use Lukasiewicz logic [13].

Furthermore, logic programming is considered a useful tool for implementing methods for reasoning with uncertainty in [38].

There is also an extension of constraint logic programming [3], which can model logics based on semiring structures. This framework can model min-max fuzzy logic, which is the only logic with semiring structure.

Another theoretical model for fuzzy logic programming without negation has been proposed by Vojtas in [35], which deals with many-valued implications.

1.2 Fuzzy Prolog

One of the most promising fuzzy tools for Prolog was the "Fuzzy Prolog" system [33, 9]. The most important advantages against the other approaches are:

1. A truth value will be a finite union of sub-intervals on $[0, 1]$. 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.
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 [14] like min, prod, etc.), disjunctive operators [31] (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 [27]).
3. Crisp and fuzzy reasoning are consistently combined [24].

Fuzzy Prolog adds fuzziness to a Prolog compiler using CLP($\mathcal{R}$) instead of implementing a new fuzzy resolution, as other former fuzzy Prologs do. It represents intervals as constraints over real numbers and *aggregation operators* as operations with these constraints, so it uses Prolog's built-in inference mechanism to handle the concept of partial truth.

1.3 Motivation and RFuzzy Approach

Over the last few years several papers have been published by Medina et al. ([22, 23, 21]) about multi-adjoint programming, which describe a theoretical model, but no means of serious implementations apart from promising prototypes [1].

Indeed, that was the reason for developing Fuzzy Prolog [9]. Fuzzy Prolog is a very expressive tool which allows the user [1] to program almost everything, but we have to pay for this expressiveness. The cost is a complex syntax difficult to understand.

The motivation for developing Rfuzzy is mainly focused on reducing this complex syntax. This reduction is based on three ideas:

1. Use real numbers instead of intervals between real numbers to represent truth values. Fuzzy Prolog answers to user queries are intervals like *it_will_rain (tonight, [0, 1])*. This is a bit difficult to understand by normal users. Truth value of this example is between 0 and 1, and this means that program can not conclude anything about the predicate truth value.
2. Whenever it is possible, do not answer user queries using constraints. A Fuzzy Prolog answer to an user query can be a constraint, like *it_will_rain (tomorrow, [X, Y]),* $X > 0$, $X < 1$, $Y > 0$, $Y < 1$. The meaning of this example is exactly the same as the meaning of the previous one, but it is slightly more difficult to understand it.
3. Truth value variables do not need to be coded. Taking care of variables to manage the predicates truth value introduces errors and makes the code illegible, without giving us any advantage.

Rfuzzy uses real numbers to represent truth values and its replies are never constraints. Besides, it is able to distinguish between crisp and fuzzy predicates and it manages the introduction of truth value variables, so the user does not need to take care of them. Truth variables are always introduced at the end of the predicate arguments list, so it can be seen as some syntactic sugar. We explain this in subsection 2.6.

From the point of view of expressiveness, we can remark that RFuzzy offers to the user the ability to define types, general and conditioned default values and truth value representations by means of facts, functions or rules. Besides, it implements multi-adjoint logic with representation of the concept of credibility of the rules, so it is one of the first tools that are actually modelling multi-adjoint logic [2].

[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] A complete formalization of the semantics of RFuzzy with a description of a least model semantics, a least fixpoint semantics, an operational semantics and the prove of their equivalence can be downloaded at `http://babel.ls.fi.upm.es/software/rfuzzy/`. This paper has been submitted and is pending of acceptance in an international conference.

2 Rfuzzy syntax

In this section we are going to describe RFuzzy's syntax. Rfuzzy defines the syntax of a new subset of Prolog predicates to work with truth values and to assign credibility to rules. The extensions that we have added to provide fuzziness of predicates are: type information, truth values (for facts, functions and rules) and default truth values.

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, constructivity of the answers, etc. Nevertheless, RFuzzy is simpler than Fuzzy Prolog for the user because the truth values are simple real numbers instead of the general structures of Fuzzy Prolog.

2.1 Type definition

Prolog does not have types. Prolog code are formulas and at execution time it looks for all of them to be true. To do that, it generates a Herbrand Universe and tries to substitute every variable with a Herbrand term. As we do not want programs to look for an answer infinitely, we offer the user a facility to restrict the set of possible solutions. This extension is named "types" and its syntax is shown in (1).

$$\textbf{:- set\_prop}\ pred/ar \Rightarrow type\_pred\_1/1\ [,\ type\_pred\_2/1\ ]^*\ . \qquad (1)$$

where *set_prop* is a reserved word, *pred* is the name of the typed predicate, *ar* is its arity and *type_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. Predicate *type_pred_*$\{n\}$ must have arity 1. The example below shows that the two arguments of the predicate *has_lower_price/2* have to be of type *car/1* and which individuals belong to that type.

```
:- set_prop has_lower_price/2 => car/1, car/1.
car(vw_caddy).
car(alfa_romeo_gt).
car(aston_martin_bulldog).
car(lamborghini_urraco).
```

2.2 Fact truth value

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

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

Arguments (*args*) should be ground and the truth value (*truth_val*) must be a real number between 0 and 1. The example below defines that the car *alfa_romeo_gt* is an *expensive_car* with a truth value 0.6.

```
expensive_car(alfa_romeo_gt) value 0.6 .
```

2.3 Functional truth value

Fact truth value definition (see subsection 2.2) is worth for a finite (and relative small) number of individuals. As we may want to define a big amount of individuals, we need more than this.

Fuzzy truth values are usually represented using continuous functions. Fig. 1 shows an example in which the truth value function assigns the truth value of being *teenager* from the person's age value.

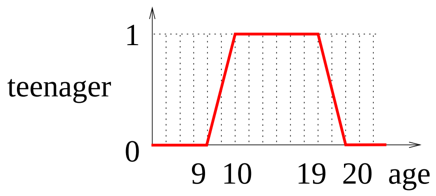

Fig. 1. Teenager credibility.

A function can be defined in several ways, but the easiest one is via a sequence of ordered pairs whose first element is the fact and the second element is the truth value assigned to that fact.

Functions used to define the truth value of some group of facts are usually continuous and linear over intervals. To define those functions there is no necessity to write down the value assigned to each element in their domains. A better way to define them is by means of their inflexion points, so function values for the elements between the inflexion points are determined by means of interpolation.

RFuzzy provides the syntax for defining functions by stretches. This syntax is shown in (3). 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. The truth value of the rest of the elements (apart from the border elements) is obtained by interpolation.

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

```
:- set_prop teenager/1 => people_age/1.
:- default(teenager/1, 0).
teenager :# ([ (9, 0), (10, 1), (19, 1), (20, 0) ]) .
```

2.4 Rule truth value

A tool which only allows the user to define truth values through functions and facts leaks on allowing him to combine those truth values for representing more complex situations. A rule is the perfect tool to combine the truth values of facts, functions, and other rules.

Rules allow the user to combine truth values in the correct way (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.

Credibility is used to express how much we trust the rule we write. Suppose a small weather example in which we have the rule *it will rain if it is cloudy and it is hot*. As it might rain but it might not, we can assign the rule a credibility of 0.7. As expected, the truth value obtained from the body is combined with the credibility value of the rule to obtain a final truth value.

Rfuzzy offers the user a concrete syntax to define combinations of truth values by means of aggregation operations, and assign to that rules a credibility. This syntax extension is defined in (4). Indeed, the user can choose two aggregation operators [3]: *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.

$$pred(arg1\ [,\ arg2]^*\ )\ [\ \mathbf{cred}\ \mathbf{(}op1, value1\mathbf{)}\ ]^{0,1}\mathbf{:}\sim op2\ pred1(arg1\ [,\ arg2]^*\ ) \\ [,\ pred2(arg1\ [,\ arg2]^*\ )]\ . \quad (4)$$

The following examples show its usage. The second one uses the operator *prod* for aggregating truth values of the subgoals of the body and the operator *min* to aggregate the result with the credibility of the rule, 0.8. As can be deduced from syntax and examples, *cred* and *:∼* are reserved words.

```
tempting_restaurant(R) :~ prod low_distance(R), cheap(R),
                                traditional(R).

good_player(J) cred (min,0.8) :~ prod swift(J), tall(J),
                                      experience(J).
```

2.5 General and Conditioned Default Truth Values

Unfortunately, information 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. The RFuzzy extension to define a default truth value for a predicate when applied to individuals for which the user has not defined an explicit truth value is named *general default truth value*.

Conditioned default truth value is used when the default truth value only applies to a subset of the function's domain. This subset is defined by a membership predicate which is true only when an individual belongs to the subset.

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

The membership predicate (*membership_predicate/ar*) and the predicate to which it is applied (*pred/ar*) need to be have the same arity (*ar*). If not, an error message will be shown at compilation time.

The syntax for defining a general default truth value is shown in (5), and the syntax for assigning a conditioned default truth value is shown in (6). *pred/ar* is in both cases the predicate to which we are defining default values. As expected, when defining the three cases (explicit, conditioned and default truth value) only one will be given back when doing a query. The precedence when looking for the truth value goes from the most concrete to the least one.

$$\textbf{:- default(}pred/ar,\ truth\_value\textbf{)}\ \textbf{.} \tag{5}$$

$$\textbf{:- default(}pred/ar,\ truth\_value\textbf{)}\ =>\ membership\_predicate/ar\textbf{.} \tag{6}$$

The example below shows how to assign a default truth value of 0.5 to all cars that do not have an explicit truth value nor have a default conditioned truth value. Besides, it shows how to assign a conditioned default truth value to all cars belonging to a small subset and not having an explicit truth value. This subset is determined by the membership predicate *expensive_type/1*, and default truth value for its elements is 0.9. So *lamborghini_urraco* is an *expensive_car* with truth value 0.9 but *vw_caddy* is an *expensive_car* with truth value 0.5. Both values are default approximations because we have no direct declaration (as for *alfa_romeo_gt* that is an *expensive_car* with a truth value 0.6 as we show above).

```
:- set_prop expensive_car/1 => car/1.
:- default(expensive_car/1, 0.9) => expensive_type/1.
:- default(expensive_car/1, 0.5).

expensive_type(lamborghini_urraco).
expensive_type(aston_martin_bulldog).
```

2.6 Doing queries with truth values

Indeed the program has to be run. When compiling, *Rfuzzy* adds a new argument to the arguments list of each fuzzy predicate. This argument serves for querying about the predicate truth value. It can be seen as syntactic sugar, as truth value is not part of the predicate arguments, but metadata information.

Truth value argument is added to the predicates in a uniform way: it is always a new argument at the end of the arguments list of the predicate. In the previous example we wrote *expensive_car/1*, so to query the system we have to give the predicate two arguments instead of only one where the second one will represent the query's truth value. This can be seen in the first example of subsection 2.7.

2.7 Constructive Answers

A fuzzy tool should be able to provide constructive answers for queries. The regular (easy) questions are asking for the truth value of an element. For example, how expensive is an *alfa_romeo_gt*?

```
?- expensive_car(alfa_romeo_gt,V).
V = 0.6 ? ;
no
```

But the really interesting queries are the ones that ask for values that satisfy constraints over the truth value. For example, which cars are very expensive?. RFuzzy provides this constructive functionality.

```
?- expensive_car(X,V), V > 0.8.
V = 0.9, X = aston_martin_bulldog ? ;
V = 0.9, X = lamborghini_urraco ? ;
no
```

The RFuzzy package implements a meta-translation of the RFuzzy syntax to ISO Prolog, via CLP(R), this is the reason for its constructivity.

3 Applications

Rfuzzy is mainly suitable for expert systems applications. As mentioned before, its main advantages in comparison to Fuzzy Prolog are its simpler syntax, the use of real numbers instead of intervals between them and the implicit handling of truth values. Besides, it presents facts' truth values (explicit, default or conditioned default truth value), functions' truth values and rules (with or without credibility) which simplifies the user development process a lot.

Although a medical expert system development were the best example of using Rfuzzy, due to lack of space we prefer to show here one in which we decide which is the best restaurant for going out.

```
:- module(restaurant,_,[rfuzzy, clpq]).

:- prop restaurant/1.

:- set_prop tempting_restaurant/1 => restaurant/1.
:- default(tempting_restaurant/1, 0.1).
tempting_restaurant(R) :~ prod low_distance(R), cheap(R),
                               traditional(R).

restaurant(kenzo).
restaurant(burger_king).
restaurant(pizza_jardin).
restaurant(subway).
restaurant(derroscas).
restaurant(il_tempietto).
restaurant(kono_pizza).
restaurant(paellador).
restaurant(tapasbar).
```

```
crisp_distance(kenzo, 50).
crisp_distance(burguer_king, 100).
crisp_distance(pizza_jardin, 70).
crisp_distance(subway, 85).
crisp_distance(derroscas, 120).
crisp_distance(il_tempietto, 150).
crisp_distance(kono_pizza, 65).
crisp_distance(paellador, 55).
crisp_distance(tapasbar, 40).

low_distance(R, TV) :- crisp_distance(R, D),
                       low_distance_aux(D, TV).

:- set_prop low_distance_aux/1 => distance/1.
:- default(low_distance_aux/1, 0).
low_distance_aux :# ([ (0, 1), (50, 0.9), (100, 0.8), (200, 0.6),
                       (300, 0.5), (500, 0.4), (1000, 0.1),
                       (2000, 0) ]).

very_low_distance(X) :- crisp_distance(X, D), D < 100.

:- set_prop cheap/1 => restaurant/1.
:- default(cheap/1, 0.2) => very_low_distance/1.
:- default(cheap/1, 0.5).

cheap(kenzo) value 0.2.
cheap(el_rincon) value 1.
cheap(el_reventaero) value 1.

:- set_prop traditional/1 => restaurant/1.
:- default(traditional/1, 0.8) => very_low_distance/1.
:- default(traditional/1, 1).

traditional(kenzo) value 0.5.
traditional(el_reventaero) value 0.87.

distance(0).
distance(X) :- distance(Y), number(Y),
X .=. Y + 1,
(   X < 5000 ; ( X >= 5000, !, fail) ).
```

In the example we can see that we know the distance to all the restaurants in a crisp way. This crisp value is translated by means of *low_distance* and *low_distance_aux* into a fuzzy one which is used into *tempting_restaurant* to determine its truth value. This allows us to ask which is the truth value of each tempting restaurant, which restaurant is a tempting restaurant with a truth value of, for example, 0.7 or list all tempting restaurants with their truth values.

The Rfuzzy module with installation instructions and examples can be downloaded from `http://babel.ls.fi.upm.es/software/rfuzzy/`.

4 Implementation details

RFuzzy is a logic programming language that is able to model all the extensions that are described in section 2. It is implemented as a Ciao Prolog [30] package because Ciao Prolog offers the possibility of dealing with a higher order compilation through the implementation of Ciao packages. Those packages serve as input for the *"Ciao System Preprocessor"* (CiaoPP) [4], a tool able to perform source-to-source transformations.

The reason beyond the implementation of *Rfuzzy* as a Ciao Prolog package is that the resultant code has to deal with two kinds of queries:

- queries in which the user asks for the truth value of an individual, and
- queries in which the user asks for an individual with a concrete truth value.

As can be seen in the following example, this is not an easy task.

```
?- A is 1, B is 2, C is A + B.

A = 1, B = 2, C = 3 ? .
yes
?- C is 3, C is A + B.
{ERROR: illegal arithmetic expression}
{ERROR: illegal arithmetic expression}
no
?-
```

Formula $C\ is\ A + B$ only works if variables A and B are bound. Almost all predicates that are problematic with non-bound variables have inside comparisons and/or assignments. This aims us to translate Rfuzzy programs into CLP($\mathcal{R}$) programs. CLP($\mathcal{R}$) is a Ciao Prolog Package which translates real number operations into constraints applied to the variables involved in those operations.

Taking advantage of Rfuzzy and CLP($\mathcal{R}$) transformations, our tool compiles Rfuzzy programs into ISO Prolog programs, so the interpreter is able to work with them as it normally does. As a result, the global compilation process has two preprocessor steps: (1) the Rfuzzy program is translated into CLP($\mathcal{R}$) constraints by means of the Rfuzzy package and (2) those constraints are translated into ISO Prolog by using the CLP($\mathcal{R}$) package. Fig. 2 shows the whole process.

In the following example the predicate *tempting_restaurant* is translated from Rfuzzy syntax into ISO Prolog syntax. In the first step, the Rfuzzy package inserts truth value variables, the *inject* metapredicate call (one of its arguments is the aggregation operator to be used, *prod*) and inserts Rfuzzy comparisons to

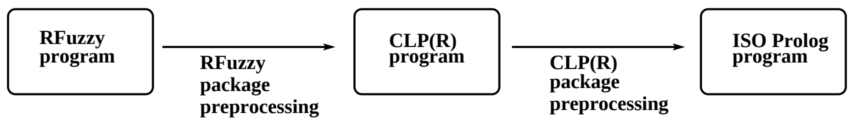

Fig. 2. Rfuzzy architecture.

take care at execution time that the rule's truth value is always between zero and one. In the second step, CLP($\mathcal{R}$) converts comparisons into constraints (via predicate calls).

```
% Rfuzzy program
tempting_restaurant(R) :~ prod low_distance(R), cheap(R),
                               traditional(R).

% CLP(R) program
rfuzzy_rule_tempting_restaurant(R,_1) :-
        low_distance(R,_2),
        cheap(R,_3),
        traditional(R,_4),
        inject([_2,_3,_4],prod,_1),
        _1 .>=. 0,
        _1 .=<. 1.

% ISO Prolog program
rfuzzy_rule_tempting_restaurant(R,_1) :-
        low_distance(R,_2),
        cheap(R,_3),
        traditional(R,_4),
        inject([_2,_3,_4],prod,_1),
        solve_generic_1(le,0,_1,-1),
        solve_generic_1(le,-1,_1,1) .
```

Internally, Rfuzzy package unifies and translates all the information given by the user to each predicate (Types, default values with and without condition, truth values defined in facts and rules with and without credibility) into a single predicate body. A simplified version of the skeleton used for that predicate is shown below.

Rfuzzy package simplified skeleton

```
Main :- Types, ( Normal ; Default)

Normal :- ( Fact ;
            (\+(Fact_Aux), Function) ;
            (\+(Fact_Aux), \+(Function_Aux), Rule)
          )
```

```
Default :- \+(Fact_Aux), \+(Function_Aux), \+(Rule_Aux),
           ( Cl_With_Cond ;
             (\+(Cl_With_Cond_Aux), Cl_With_No_Cond)
           )
```

The skeleton has three different parts: the one which takes care of allowing only queries or answers with the expected individuals, the one which looks for a concrete truth value (it can be defined by means of a fact, a function or a rule) and the one which looks for a default truth value (conditioned or not). Predicates ending in _aux do not take care on the truth value argument.

The first part is obtained from the type definitions (see 2.1), translating all types into a predicate which is called at first (Types) so we assure we only work with the expected individuals.

The second part looks for a concrete value whose arguments have to unify with the parameters the user has given. Precedence when looking for it is:

1. A fact (see subsection 2.2)
2. A function (see subsection 2.3)
3. A rule (see subsection 2.4)

The third part is only called when the second one (searching for a truth value) fails, and looks for a conditioned or default truth value.

5 Conclusions and Current Work

Rfuzzy offers to the users a new framework to represent fuzzy problems over real numbers. *Fuzzy Prolog* [34, 33, 9] is an existing framework for dealing with Fuzzy problems representation. Main *Rfuzzy* advantages over *Fuzzy Prolog* are a simpler syntax and the elimination of answers with constraints, and *Rfuzzy* is one of the first tools modelling multi-adjoint logic, as explained in subsection 1.3.

Rfuzzy syntax is simpler that *Fuzzy Prolog* syntax. Its fuzzy values are simple 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 behaviour of fuzzy variables, this syntax reduction is an advantage. Programs written in *Rfuzzy* syntax are more legible and more easy to understand than *Fuzzy Prolog* programs.

Fuzzy Prolog answers to user queries are difficult to understand due to the existence of constraints. As normal replies to final users are ground terms, the programmer has to code by hand how to reach them. To eliminate those constraints and answer queries with ground terms the programmer tries to substitute variables with ground terms until one makes true all of them. *Rfuzzy* offers a powerful tool to deal with this task: *Type definition*. *Type definition* (see subsection 2.1) allows the user to define which terms are suitable for being substituted into a variable, so she does not have to code this behaviour again. Besides, the elimination of answers with constraints provides more human readable answers and more easy to test programs (because answers we test do not have constraints, just ground terms).

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 2.5). This allows us to make inference with default truth values when we do not know anything about the truth of some fact.

Extensions added to *Prolog* by *Rfuzzy* are: Types, default truth values (conditioned or not), assignment of truth values to individuals by means of facts, functions or rules, and assignment of credibility to the rules.

Besides, the possibility to provide constructive answers to the queries increase its usage, as can be seen in subsection 2.7.

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.), 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 apply constructive negation to the engine. RFuzzy needs to define types in a constructive way (by means of predicates that are able to generate all their individuals by backtracking) so we cannot use constraints. Future research will be done in this line for widening the definition of types.

References

1. J.M. Abietar, P.J. Morcillo, and G. Moreno. Designing a software tool for fuzzy logic programming. In T.E. Simos and G. Maroulis, editors, *Proc. of the Int. Conf. of Computational Methods in Sciences and Engineering. ICCMSE'07*, volume 2 of *Computation in Mordern Science and Engineering*, pages 1117–1120. American Institute of Physics, 2007. Distributed by Springer.
2. J. F. Baldwin, T. P. Martin, and B. W. Pilsworth. *Fril: Fuzzy and Evidential Reasoning in Artificial Intelligence*. John Wiley & Sons, 1995.
3. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint Logic Programming: syntax and semantics. In *ACM TOPLAS*, volume 23, pages 1–29, 2001.
4. F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP8/95.0.7.20, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, 1999.
5. T.H. Cao. Annotated fuzzy logic programs. *Fuzzy Sets and Systems*, 113(2):277–298, 2000.
6. D. Dubois, J. Lang, and H. Prade. Towards possibilistic logic programming. In *Proc. of ICLP-91*, pages 581–595. MIT Press, 1991.
7. M. Fitting. Bilattices and the semantics of logic programming. *Journal of Logic Programmig*, 11:91–116, 1991.
8. N. Fuhr. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science*, 51(2):95–110, 2000.

9. S. Guadarrama, S. Munoz-Hernandez, and C. Vaucheret. Fuzzy Prolog: A new approach using soft constraints propagation. *Fuzzy Sets and Systems, FSS*, 144(1):127–150, 2004. ISSN 0165-0114.
10. M. Ishizuka and N. Kanai. Prolog-ELF incorporating fuzzy Logic. In *IJCAI*, pages 701–703, 1985.
11. M. Kifer and Ai Li. On the semantics of rule-based expert systems with uncertainty. In *Proc. of ICDT-88*, number 326 in LNCS, pages 102–117, 1988.
12. M. Kifer and V.S. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12:335–367, 1992.
13. F. Klawonn and R. Kruse. A Łukasiewicz logic based Prolog. *Mathware & Soft Computing*, 1(1):5–29, 1994.
14. E.P. Klement, R. Mesiar, and E. Pap. Triangular norms. Kluwer Academic Publishers.
15. L. Lakshmanan. An epistemic foundation for logic programming with uncertainty. *LNCS*, 880:89–100, 1994.
16. L. Lakshmanan and N. Shiri. Probabilistic deductive databases. *Int. Logic Programming Symposium*, pages 254–268, 1994.
17. L. Lakshmanan and N. Shiri. A parametric approach to deductive databases with uncertainty. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):554–570, 2001.
18. R. C. T. Lee. Fuzzy Logic and the resolution principle. *Journal of the Association for Computing Machinery*, 19(1):119–129, 1972.
19. D. Li and D. Liu. *A Fuzzy Prolog Database System.* John Wiley & Sons, New York, 1990.
20. T. Lukasiewicz. Fixpoint characterizations for many-valued disjunctive logic programs with probabilistic semantics. In *Proc. of LPNMR-01*, volume 2173, pages 336–350, 2001.
21. J. Medina, M. Ojeda-Aciego, and P. Votjas. A completeness theorem for multi-adjoint Logic Programming. In *International Fuzzy Systems Conference*, pages 1031–1034. IEEE, 2001.
22. J. Medina, M. Ojeda-Aciego, and P. Votjas. Multi-adjoint Logic Programming with continuous semantics. In *LPNMR*, volume 2173 of *LNCS*, pages 351–364, Boston, MA (USA), 2001. Springer-Verlag.
23. J. Medina, M. Ojeda-Aciego, and P. Votjas. A procedural semantics for multi-adjoint Logic Programming. In *EPIA*, volume 2258 of *LNCS*, pages 290–297, Boston, MA (USA), 2001. Springer-Verlag.
24. S. Munoz-Hernandez, C. Vaucheret, and S. Guadarrama. Combining crisp and fuzzy Logic in a prolog compiler. In J. J. Moreno-Navarro and J. Mariño, editors, *Joint Conf. on Declarative Programming: APPIA-GULP-PRODE 2002*, pages 23–38, Madrid, Spain, September 2002.
25. R. Ng and V.S. Subrahmanian. Stable model semantics for probabilistic deductive databases. In *Proc. of ISMIS-91*, number 542 in LNCS, pages 163–171, 1991.
26. R. Ng and V.S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1993.
27. 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*, 30(1):57–72, 2002.
28. Z. Shen, L. Ding, and M. Mukaidono. Fuzzy resolution principle. In *Proc. of 18th International Symposium on Multiple-valued Logic*, volume 5, 1989.

29. V.S. Subrahmanian. On the semantics of quantitative logic programs. In *Proc. of 4th IEEE Symp. on Logic Programming*, pages 173–182. Computer Society Press, 1987.
30. The CLIP Lab. The Ciao Prolog Development System WWW Site, `http://www.clip.dia.fi.upm.es/Software/Ciao/`.
31. E. Trillas, S. Cubillo, and J. L. Castro. Conjunction and disjunction on $([0, 1], <=)$. *Fuzzy Sets and Systems*, 72:155–165, 1995.
32. M.H. van Emden. Quantitative duduction and its fixpoint theory. *Journal of Logic Programming*, 4(1):37–53, 1986.
33. C. Vaucheret, S. Guadarrama, and S. Munoz-Hernandez. Fuzzy prolog: A simple general implementation using clp(r). In M. Baaz and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2002*, number 2514 in LNAI, pages 450–463, Tbilisi, Georgia, October 2002. Springer-Verlag.
34. C. Vaucheret, S. Guadarrama, and S. Munoz-Hernandez. Fuzzy prolog: A simple general implementation using clp(r). In P.J. Stuckey, editor, *Int. Conf. in Logic Programming, ICLP 2002*, number 2401 in LNCS, page 469, Copenhagen, Denmark, July/August 2002. Springer-Verlag.
35. P. Vojtas. Fuzzy logic programming. *Fuzzy Sets and Systems*, 124(1):361–370, 2001.
36. G. Wagner. A logical reconstruction of fuzzy inference in databases and logic programs. In *Proc. of IFSA-97*, Prague, 1997.
37. G. Wagner. Negation in fuzzy and possibilistic logic programs. In *Logic programming and Soft Computing*. Research Studies Press, 1998.
38. Ehud Y. and Shapiro. Logic programs with uncertainties: A tool for implementing rule-based systems. In *IJCAI*, pages 529–532, 1983.