RFuzzy: An Expressive Simple Fuzzy Compiler

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

Universidad Politécnica de Madrid*
{susana,vpablos}@fi.upm.es, hannes.strass@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 the representation of fuzzy concepts is complex or the results of the fuzzy queries are difficult to interpret.

In this paper we present a modern framework, *RFuzzy*, that is modeling multi-adjoint logic in a practical way. It provides some extensions as default values (to represent missing information), partial default values (for a subset of data) and typed variables. *RFuzzy* represents the truth value of predicates using facts, rules and also can define fuzzy predicates as continuous functions. Queries are answered with direct results (instead of providing complex constraints), so it is easy to use for any person that wants to represent a problem using fuzzy reasoning in a simple way (just using the classical fuzzy representation with real numbers). The most promising characteristic of *RFuzzy* is that the user can obtain constructive answers to queries that restrict the truth value.

Keywords: Fuzzy reasoning, Implementation tool, Fuzzy Logic, Multi-adjoint logic, Logic Programming Implementation, Fuzzy Logic 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 the sense of fuzziness) in logic programming.

* 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.

J. Cabestany et al. (Eds.): IWANN 2009, Part I, LNCS 5517, pp. 270–277, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Despite of the multitude of theoretical approaches to this issue, few of them resulted in current usable tools. Since Logic Programming is traditionally used in Knowledge Representation and Reasoning, we argue (as in [15]) that it is perfectly well-suited to implement a fuzzy reasoning tool as ours.

1.1 Fuzzy Approaches in Logic Programming

Introducing Fuzzy Logic into Logic Programming has provided the development of several fuzzy systems over Prolog. These systems replace its inference mechanism, 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 [4], as the Prolog-Elf system, the FRIL Prolog system and the F-Prolog language. However, there is no common method for fuzzifying Prolog, as noted in [11].

Some of these Fuzzy Prolog systems only consider fuzziness on predicates 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 Łukasiewicz logic.

There is also an extension of constraint logic programming [2], 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 Vojtáš in [14], which deals with many-valued implications.

1.2 Fuzzy Prolog

One of the most promising fuzzy tools for Prolog was the "Fuzzy Prolog" system [13,3]. The most important advantages against the other approaches are:

1. A truth value is represented as 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 (a real number) 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 like min, prod, etc.), disjunctive operators (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).
3. Crisp and fuzzy reasoning are consistently combined [10].

Fuzzy Prolog adds fuzziness to a Prolog compiler using CLP($\mathcal{R}$) instead of implementing a new fuzzy resolution method, 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 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. ([6,7,5]) about multi-adjoint programming, which describe a theoretical model, but no means of serious implementations apart from promising prototypes [1] and recently the FLOPER tool [9,8].

FLOPER implementation is inspired in Fuzzy Prolog [3] and adds the modellization of multi-adjoint logic. On one side Fuzzy Prolog is more expressive in the sense that can represent continuous fuzzy functions and its truth value is more general (union of intervals of real numbers), on the other side Fuzzy Prolog syntax is so flexible and general that can be complex for non-expert programmers just interested in modelling simple fuzzy problems.

This is the reason why we propose here the *RFuzzy* approach that is simpler than Fuzzy Prolog for the user because the truth values are simple real numbers instead of the general structures of Fuzzy Prolog. *RFuzzy* also models multi-adjoint logic and moreover provides some interesting improvements with respect to FLOPER: default values, partial default values (just for a subset of data), types for variables, and a useful sugar-syntax (for representing facts, rules and functions). Additionally *RFuzzy* inherits Fuzzy Prolog characteristics that are more expressive than other tools (uses Prolog-like syntax, has flexibility in the queries syntax, combines crisp and fuzzy predicates, uses general aggregation operators and provides constructive answers querying data and querying truth values).

Besides, *RFuzzy* implements multi-adjoint logic with a simple representation of the concept of credibility of the rules of multi-adjoint logic[1].

2 *RFuzzy* Expressiveness

RFuzzy enhances regular Prolog with truth values and with credibility values. In this section we enumerate and describe some of the most interesting characteristics of *RFuzzy* expressiveness through its syntax.

2.1 Types Definition

Prolog does not have types. It assigns values to the variables taking terms from the Herbrand Universe that can be created from the set of constants and constructors defined in a program. Nevertheless if we use types, then we can constraint the domain of values of the variables and this help us to return finite constructive answers (semantically correct). In *RFuzzy* types are defined according to (1) syntax.

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

[1] 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 proof of their equivalence can be downloaded at
`http://babel.ls.fi.upm.es/software/rfuzzy/`

where *set_prop* is a reserved word, *pred* is the name of the typed predicate, *ar* is its arity and *type_pred_1*, *type_pred_n* ($n \in 2, 3, \ldots, ar$) are predicates used to define types for each argument of *pred*. They must have arity 1. The definition is constraining the values of the n_th argument of *pred* to the values of the type *type_pred_n*. This definition of types ensures that the values assigned to the arguments of *pred* are correctly typed.

The example below shows that the arguments of predicates *has_lower_price/2* and *expensive_car/1* have to be of type *car/1*. The domain of type *car* is enumerated.

$$:-set\_prop\ has\_lower\_price/2\ =>\ car/1,\ car/1.$$
$$:-set\_prop\ expensive\_car/1\ =>\ car/1.$$
$$car(vw\_caddy). \qquad car(alfa\_romeo\_gt).$$
$$car(aston\_martin\_bulldog). \qquad car(lamborghini\_urraco).$$

2.2 Simple Truth Value Assignment

It is possible to assign a truth value to an individual using fuzzy facts. Their syntax, that we can see in (2), is different than regular Prolog facts syntax.

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

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 Continuous Function to Represent Truth Values

Facts definition (see subsection 2.2) is worth for a finite (and relative small) number of individuals. Nevertheless, it is very common to represent fuzzy truth using continuous functions. Fig. 1 shows an example in which the continuous function assigns the truth value of being *teenager* to each age.

Functions used to define the truth value of some group of individuals 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. We have to take into account that the domain can be infinite.

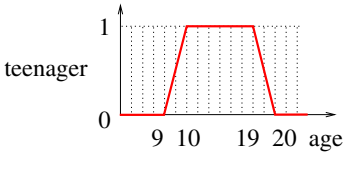

Fig. 1. Teenager truth value continuous representation

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. Predicate *pred* has arity 1, *val1, ..., valN* should be ground terms representing numbers of the domain (they are possible values of the argument of *pred*) and *truth_val1, ..., truth_valN* should be the truth values associated to these numbers. The truth value of the rest of the elements is obtained by interpolation.

$$pred \textbf{ :\#} \ ([(val1, truth\_val1),\ (val2, truth\_val2)\ [,\ (valn, truth\_valn)\ ]^*]) \ . \quad (3)$$

The *RFuzzy* syntax for the predicate *teenager/1* (represented in Fig.1) is:

$$teenager : \#([(9,0),(10,1),(19,1),(20,0)]).$$

2.4 Rule Definition with Truth Values and Credibility

A tool which only allows the user to define truth values through functions and facts lacks on allowing him to combine those truth values for representing more complex situations. A rule is the 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 *minimum*, *maximum*, *product*, etc.). The aggregation operator combines the truth values of the subgoals of the body of the rule to obtain the truth value of the head of the rule.

Appart from this, rules are assigned a credibility value to obtain the final truth value for the head of the clause. Credibility is used to express how much we trust a rule. It is used another opperator to aggregate the truth value obtained (from the aggregation of the subgoals of the body) with the rule's credibility.

RFuzzy offers a simple syntax for representing these rules, defined in (5). There are two aggregation operators, *op2* for combining the truth values of the subgoals of the rule body and *op1* for combining the previous result with the rule's credibility. The user can choose for any of them an aggregation operator from the list of the available ones[2] or define his/her own aggregation operator.

$$pred(arg1\ [,\ argn]^*\ )\ [\ \textbf{cred}\ (op1, value1)\ ]\ \textbf{:}\sim op2 \quad (4)$$
$$pred1(args\_pred\_1)\ [,\ predm(args\_pred\_m)]\ .$$

The following example uses the operator *prod* for aggregating truth values of the subgoals of the body and *min* to aggregate the result with the credibility of the rule (which is 0.8). "**cred** $(op1, value1)$" can only appear 0 or 1 times.

$$good\_player(J)\ cred(min, 0.8)\ \textbf{:}\sim prod\ swift(J),\ tall(J),\ has\_experience(J).$$

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

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 for a fuzzy predicate 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. A 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*. The syntax for defining a general default truth value is shown in (5).

Conditioned default truth value is used when the default truth value only applies to a subset of the domain. This subset is defined by a membership predicate which is true only when an individual belongs to the subset. The membership predicate (*membership_predicate/ar*) and the predicate to which it is applied (*pred/ar*) need to have the same arity (*ar*). The syntax is shown in (6).

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

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

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.

The code from the example below joint with the code from examples in subsections 2.1 and 2.2 assigns to the predicate *expensive_car* a truth value of 0.5 when the car is *vw_caddy* (default truth value), 0.9 when it is *lamborghini_urraco* or *aston_martin_bulldog* (conditioned default truth value) and 0.6 when it is *alfa_romeo_gt* (explicit truth value).

$$:-default(expensive\_car/1, 0.9) => expensive\_make/1.$$
$$:-default(expensive\_car/1, 0.5).$$
$$expensive\_make(lamborghini\_urraco).$$
$$expensive\_make(aston\_martin\_bulldog).$$

2.6 Constructive Answers

A very interesting characteristic for a fuzzy tool is being able to provide constructive answers for queries. The regular (easy) questions ask for the truth value of an element. For example, how expensive is an *Volkswagen Caddy*? (See left hand side example below)

$?-expensive\_car(vw\_caddy, V).$	$?-expensive\_car(X, V), V > 0.8.$
$V = 0.5?;$	$V = 0.9, X = aston\_martin\_bulldog?;$
no	$V = 0.9, X = lamborghini\_urraco?;$
	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? (See right hand side example above). *RFuzzy* provides this constructive functionality.

3 Implementation Details

RFuzzy has to deal with two kinds of queries, (1) queries in which the user asks for the truth value of an individual, and (2) queries in which the user asks for an individual with a concrete or a restricted truth value. For this reason *RFuzzy* is implemented as a Ciao Prolog [12] package because Ciao Prolog offers the possibility of dealing with a higher order compilation through the implementation of Ciao packages.

The compilation process of a *RFuzzy* program has two pre-compilation steps: (1) the *RFuzzy* program is translated into CLP($\mathcal{R}$) constraints by means of the *RFuzzy* package and (2) the program with constraints is translated into ISO Prolog by using the CLP($\mathcal{R}$) package. Fig. 2 shows the whole process.

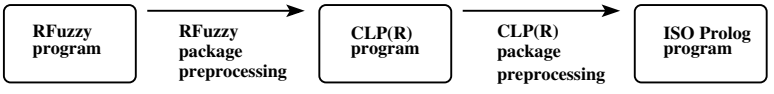

Fig. 2. *RFuzzy* architecture

4 Conclusions

RFuzzy is not a work in progress. It is an available implementation[3] that offers to the users/programmers a new framework to represent fuzzy problems over real numbers. Main *RFuzzy* advantages over *Fuzzy Prolog* are a simpler syntax and the elimination of answers with constraints. Moreover *RFuzzy* is one of the first tools modelling multi-adjoint logic, as explained in subsection 1.3. All the advanced characteristics of *RFuzzy* are missing in other tools as FLOPER[9,8].

Extensions added to *Prolog* by *RFuzzy* are: types (subsection 2.1), default truth values conditioned or general (subsection 2.5), assignment of truth values to individuals by means of facts (subsection 2.2), functions (subsection 2.3) or rules with credibility (subsection 2.4).

One of the most important consequences of these extensions is the constructivity of the answers with the possibility of constraining the truth value in the queries as we describe in section 2.6.

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, Coding Rules, etc.

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

References

1. Abietar, J.M., Morcillo, P.J., Moreno, G.: Designing a software tool for fuzzy logic programming. In: Simos, T.E., Maroulis, G. (eds.) Proc. of the Int. Conf. of Computational Methods in Sciences and Engineering. ICCMSE 2007. Computation in Mordern Science and Engineering, vol. 2, pp. 1117–1120. American Institute of Physics (2007) (Distributed by Springer)
2. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint Logic Programming: syntax and semantics. In: ACM TOPLAS, vol. 23, pp. 1–29 (2001)
3. Guadarrama, S., Munoz-Hernandez, S., Vaucheret, C.: Fuzzy Prolog: A new approach using soft constraints propagation. Fuzzy Sets and Systems 144(1), 127–150 (2004)
4. Lee, R.C.T.: Fuzzy Logic and the resolution principle. Journal of the Association for Computing Machinery 19(1), 119–129 (1972)
5. Medina, J., Ojeda-Aciego, M., Votjas, P.: A completeness theorem for multi-adjoint Logic Programming. In: International Fuzzy Systems Conference, pp. 1031–1034. IEEE, Los Alamitos (2001)
6. Medina, J., Ojeda-Aciego, M., Votjas, P.: Multi-adjoint Logic Programming with continuous semantics. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS, vol. 2173, pp. 351–364. Springer, Heidelberg (2001)
7. Medina, J., Ojeda-Aciego, M., Votjas, P.: A procedural semantics for multi-adjoint Logic Programming. In: Brazdil, P.B., Jorge, A.M. (eds.) EPIA 2001. LNCS, vol. 2258, pp. 290–297. Springer, Heidelberg (2001)
8. Morcillo, P.J., Moreno, G.: Floper, a fuzzy logic programming environment for research. In: Proceedings of the Spanish Conference on Programming and Computer Languages, PROLE 2008, Gijón, Spain (2008)
9. Moreno, G.: Building a fuzzy transformation system. In: SOFtware SEMinar 2006: Theory and Practice of Computer Science, pp. 409–418 (2006)
10. Munoz-Hernandez, S., Vaucheret, C., Guadarrama, S.: Combining crisp and fuzzy Logic in a prolog compiler. In: Moreno-Navarro, J.J., Mariño, J. (eds.) Joint Conf. on Declarative Programming: APPIA-GULP-PRODE 2002, Madrid, Spain, pp. 23–38 (September 2002)
11. Shen, Z., Ding, L., Mukaidono, M.: Fuzzy resolution principle. In: Proc. of 18th International Symposium on Multiple-valued Logic, vol. 5 (1989)
12. The CLIP Lab. The Ciao Prolog Development System WWW Site, `http://www.clip.dia.fi.upm.es/Software/Ciao/`
13. Vaucheret, C., Guadarrama, S., Munoz-Hernandez, S.: Fuzzy prolog: A simple general implementation using clp(r). In: Baaz, M., Voronkov, A. (eds.) LPAR 2002. LNCS (LNAI), vol. 2514, pp. 450–463. Springer, Heidelberg (2002)
14. Vojtas, P.: Fuzzy logic programming. Fuzzy Sets and Systems 124(1), 361–370 (2001)
15. Ehud, Y., Shapiro: Logic programs with uncertainties: A tool for implementing rule-based systems. In: International Joint Conference on Artificial Intelligence, pp. 529–532 (1983)