UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

MEJORAS DE LA EXPRESIVIDAD DE LENGUAJES LÓGICOS CON EL MANEJO DE INFORMACIÓN NEGATIVA Y DIFUSA

PHD THESIS

Víctor Pablos Ceruelo

Ingeniero en Informática

June 2015

10

2015

Mejoras de la expresividad de lenguajes lógicos con el manejo de información negativa y difusa

Departamento de Lenguajes y
Sistemas Informáticos e Ingeniería de Software
Facultad de Informática
Universidad Politécnica de Madrid

Mejoras de la expresividad de lenguajes lógicos con el manejo de información negativa y difusa

presented in partial fulfillment of the degree of
Doctor in Software and Sistemas

Author: **Víctor Pablos Ceruelo**
Ingeniero en Informática
Universidad Politécnica de Madrid

Advisor: **Susana Muñoz Hernández**
Profesor Contratado Doctor
Universidad Politécnica de Madrid

Madrid, June 2015

Thesis Committee:

- **Prof. Juan José Moreno-Navarro**
  Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software, Universidad Politécnica de Madrid

- **Prof. Manuel Ojeda Aciego**
  Departamento de Matemática Aplicada, ETSI Informática, Universidad de Málaga

- **Prof. João Paulo Carvalho**
  Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa

- **Prof. Nicolas Madrid**
  Centre of Excellence IT4Innovations,
  University of Ostrava

- **Prof. Pedro López-García**
  Madrid Institute for Advanced Studies in Software Development Technologies (IMDEA Software Institute)

- **Prof. José Júlio Alferes**
  Departamento de Informática Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

- **Prof. Miguel Delgado Calvo-Flores**
  Departamento de Ciencias de la Computación e Inteligencia Artificial, Universidad de Granada

To my girlfriend,
Sara,
with love.

Contents

List of Figures

List of Tables

List of Listings

Resumen

Just the spanish translation of next chapter. To be done. 4000 chars max.

Summary

In this work we provide a theoretical basis (syntax and semantics) and a practical implementation of a framework for encoding the reasoning and the fuzzy representation of the world (as human beings understand it). The interest for this work comes from two sources: removing the existing complexity when doing it with a general purpose programming language (one developed without focusing in providing special constructions for representing fuzzy information) and providing a tool intelligent enough to answer in a constructive way, expressive queries over conventional data.

The framework, RFuzzy, allows to encode rules and queries in a syntax very close to the natural language used by human beings to express their thoughts, but it is more than that. It allows to encode very interesting concepts, as fuzzifications (functions to easily fuzzify crisp concepts), default values (used for providing results less adequate but still valid when the information needed to provide results is missing), similarity between attributes (used to search for individuals with a characteristic similar to the one we are looking for), synonyms or antonyms and it allows to extend the number of connectives and modifiers (even negation) we can use in the rules. The personalization of the definition of fuzzy concepts (very useful for dealing with the subjective character of fuzziness, in which a concept like tall depends on the height of the person performing the query) is another of the facilities included.

Besides, RFuzzy implements the multi-adjoint semantics. The interest in them is that in addition to obtaining the grade of satisfaction of a consequent from a rule, its credibility and the grade of satisfaction of the antecedents we can determine from a set of data how much credibility we must assign to a rule to model the behaviour of the set of data. So, we can determine automatically the credibility of a rule for a particular situation.

Although the theoretical contribution is interesting by itself, specially the inclusion of the negation modifier, the practical usage of it is equally important. Between the different uses given to the framework we highlight emotion recognition, robocup control, granularity control in parallel/distributed computing and flexible searches in databases.

Chapter 1

Introduction

The information available on the internet is huge, and it is increasing day by day. This is on one side good. The availability makes it possible to search for almost anything. But not on the other side, in which search engines are not able to return the expected answers.

Just suppose we are looking for a fast red car. We just open a web browser, go to the search form of our favourite search engine, write “fast red car” and press the button “search”. At this point our web browser communicates our query to a server and this one answers it with a list of web pages in which the words “fast”, “red” and “car” appear at some point. This could be the expected answer for our query, but it is not. We want as answers only those web pages talking about fast red cars, so the list of results must be reduced.

Leaving apart that the information about web pages is non-structured, we could think about a database with a list of words grouped by sentences. Even in that case the results might be erroneous, so pages containing sentences like “Mary bought a fast red car” could lead us to having in the result’s list a web page about news and gossip.

All this problems could be solved by modelling the information we have about the world as we see it, in a fuzzy way, and having tools for querying the systems in the same way. The existing tools when we started with this work had a limited set of capabilities, which is why we begun trying to increase them. The goal of this thesis was providing tools expressive enough to allow us to achieve this task. To achieve it is why we have developed the theoretical and practical parts of a framework for representing the world as we, human beings, see it.

In what remains of this chapter we focus in all the previous knowledge needed to understand the following ones. We start from an informal introduction to the representation of the real world in the way that we, as human beings, see it (Sec. 1.1). After it we enter in how all this information can be represented and treated by a computer, starting by the paradigm chosen, logic programming

(Sec. 1.2), and continuing by the representation of fuzziness in logic (Sec. 1.3). And finally we explain in detail the semantics we have chosen for representing the world and why we decided to use them (Sec. 1.4). All this is complemented by an explanation of the structure chosen for the document (Sec. 1.5).

1.1 A Fuzzy World

The world around us is clearly non-fuzzy. What we mean with this sentence is that we can always quantify it and determine exact values if we find first the unit measure and how to ascertain the size, amount or degree of the goal of our measures. Fuzziness comes to life when we, as human beings, try (maybe unconsciously) to reduce the amount of data that we have to retain in our memory. In this way, instead of keeping that Clara is 70cm we keep that Clara is a baby, sentence that offers us much more information at a minor storage cost.

In maths and in computer science we can use the extended version, but then the translation between the information (and knowldege) we have in mind and the one used in formulas and automatisms must be done by experts. This is why fuzzy logic has been so important in the recent years, because it allows the final user not only to understand easily but to check and extend the existing information (and knowledge). We provide now an example to illustrate what we mean with our previous sentences.

Suppose we are taking a bath and we have a shower mixer tap (not one of those which are thermostatic) and that when turning it to the left cold water is dispensed, while when turning it to the right hot water is dispensed. This means that if the water is not as hot as we want we usually turn it to the right, while if it is too hot we turn it to left. The question is: how much do we turn it to the left or to the right? 10 degrees? 20 degrees? 2 and a half? Obviously we never talk using this measures. We just remember that turning it a little bit changes the temperature a little bit, while turning it completely changes it completely and use a set of rules similar to:

- If the water is too hot, give the tap a quarter-turn to the left. Wait until the water changes its temperature and decide again.
- If it is hot, turn the tap to the left a little bit (less than a half-quarter-turn). Wait until the water changes its temperature and decide again.
- If it is ok, do not touch the tap. Enjoy ! Wait until the water changes its temperature (it usually does) and decide again.
- If it is cold, turn the tap to the right a little bit (less than a half-quarter-turn). Wait until the water changes its temperature and decide again.

- If it is very cold, give the tap a quarter-turn to the right. Wait until the water changes its temperature and decide again.

Now suppose that we want to encode this "intellegence" into an automatism. Encoding it using numbers makes it difficult to understand, and complicates the very simple idea we have in mind, as we see now. Being too hot over 50 degrees, hot over 45, cold under 30, very cold under 20 and "ok" between 30 and 40 we can encode the previous rules as

- if (temperature > 50) then turn_to_the_left(25) and recheck.
- if (46 < temperature <= 50) then turn_to_the_left(12) and recheck.
- if (41 < temperature <= 45) then turn_to_the_left(6) and recheck.
- if (30 <= temperature <= 40) then do nothing. Recheck after a while.
- if (25 <= temperature < 30) then turn_to_the_right(6) and recheck.
- if (20 <= temperature < 25) then turn_to_the_right(12) and recheck.
- if (temperature < 20) then turn_to_the_right(25) and recheck.

The previous example shows that the resulting program is difficult to read and the more pieces (or sentences) we use to define the actions from the inputs the more variables we have. This is even more problematic if we want to make changes or if we need to personalize the temperatures (maybe you prefer the water temperature to be hotter in winter). With fuzzy logic we can develop programs that behave more as humans and less as a complex algorithm. Take the previous example. If we try to encode it with fuzzy logic we start creating a linguistic variable for the temperature. We will use the function in Fig. 1.1.1 to fuzzify the real world value.

By using the linguistic variable temperature we can encode our knowledge in a different way. The one we propose in this case is

- if (warm(temperature)) then turn_to_the_left and recheck.
- if (ok(temperature)) then do nothing.
- if (cold(temperature)) then turn_to_the_right and recheck.

In the previous example we are obviously using fuzzy logic, so when the conditions are not fully satisfied the conclusions are not fully satisfied. Even more, note that we have not encoded how much the mixer tap needs to be turned in any direction. We consider that the defuzzification should not

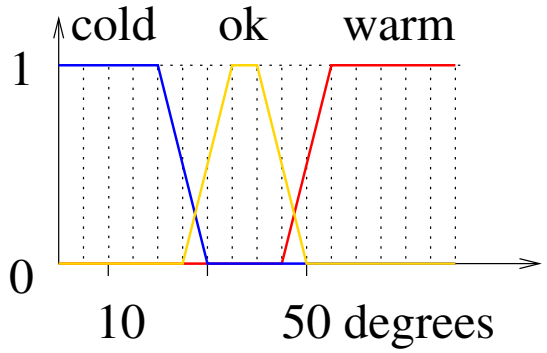


Figure 1.1.1: The linguistic variable temperature takes the values cold, ok and hot depending of the water temperature.

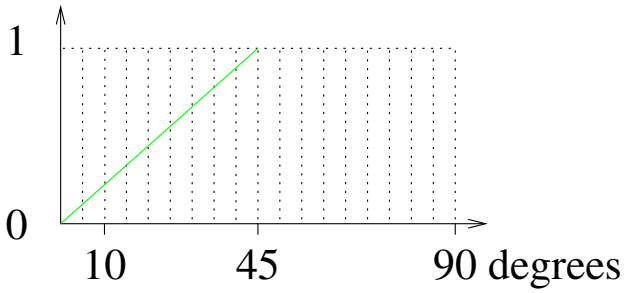


Figure 1.1.2: How much we turn to the left or right (or function to defuzzify), from the decision taken by the expert system.

interfere in the decision making. Nevertheless, the function used to defuzzify the decision taken is shown in Fig. 1.1.2.

As example of how this works, if the water has a temperature of 25 degrees the satisfaction of the previous rules is { 0.5, 0, 0 } and the result is that the mixer tap turns to the left with a satisfaction of 0.5, which results in 25 degrees (applying defuzzification to the satisfaction value 0.5).

1.2 Logic Programming and Logic

At the inception of computers construction-related difficulties were clearly so dominant that the languages for expressing problems and instructing the computer how to solve them were designed from the perspective of computer engineering alone. This languages were called *Imperative Programming Languages*, because they allow the programmer to impose the machine what to do now and what to do next. As the problems of building computers were gradually

understood and solved, the problem of using the programming languages mounted: a human could not instruct, or program, the computer to do complex tasks because he was not able to sum up the tasks done by each single code line into the task done by the whole program. This problem was first solved by the programmers, who had a repertory of pieces of code for well know tasks and just copy and paste them into the program under development, but this became unmanageable. The necessity of programming languages hiding the pieces of code doing well-known tasks forced the creation of "high level" programming languages. A "high level" programming language is basically that, a set of syntactic constructions corresponding to well known pieces of source code that the programmer can use to develop programs. It started from very simple schemas and nowadays programming languages include constructions as conditional loops, procedures, functions, abstract data types, objects, agents, aspects, and some others appearing day after day.

The existence of the schemas of source code reduce the time spent in coding programs, but do not remove the possibility of developing programs with errors. The anonymous quote

> To err is human, to really foul things up requires a computer.

says it all: we (humans) introduce errors when coding (because of our natural being) and computers mess things up because they just obey orders. To avoid this situations the high level programming include debuggers (to see at execution time what the computer is doing and determine where the error is) and checks. This checks are usually limitations in the available syntactical structures that can be applied to the data structures, so that we cannot add a number to a string of characters and things like that. This kind of checks can be static or dynamic, depending on if they are performed at compilation time (when the program written in the high-level programming language syntax is translated into the machine language) or at evaluation time (when the computer runs the machine language program). All this checks reduce the amounts of errors in programs, but there are some cases where the tests are not definitive: they can only warn the user to check by hand that the constructions are used as they should be. When software is not critical we can live with this solution, but there are some cases in which it is critical and we are forced to ensure that it is completely correct. Some solutions have been proposed to this problem: (1) to curtail the expressiveness of the language, (2) to introduce heuristics in the compilation process, (3) to build software systems "correct by construction" and (4) to use programming languages with syntax and semantics closer to the way a human being formalizes a task. In the first group we have as example the Ada language [The15a], which forbids using constructions as "goto" in order to improve the quality of the programs. In the second one there

are a lot of ongoing works, as [Mar+09], but most of them finally derive into curtailing the expressiveness of the language or allowing the programmer to introduce a "I checked this by hand" message for the compiler (so it does not ask for doing it again). The third one has to be with removing the necessity to translate by hand the "what we want the computer to do" into the operations that it must perform to achieve the task. The idea is writing an specification and letting a translator (a compiler) traduce it into machine code. Some examples are the B-method [Rob97], VDM [Jon87; Jon90] or the Z notation [Spi89]. The last one tries to reduce the distance between the task to be done and how to do it, by using *Declarative Programming Languages* instead of *Imperative Programming Languages*.

Declarative Programming Languages allow the programmer to represent the relation between the inputs and the outputs by using maths and/or logic, but it is a compiler task to decide the operations that the computer must perform to obtain the outputs from the inputs. There are mainly two subgroups, depending on if the language departs from a mathematical point of view of the world (use of functions to model the world and reason about it) or from a logic point of view (use of logic to model assumptions or premises and derive conclusions): *Functional Programming Languages* and *Logic Programming Languages*. Although we know that there are hybrids too, Functional Programming Languages and hybrids are out of the scope of this contribution. We just point out that Functional Programming Languages have their roots in lambda calculus and aim the reader to start by [Gol96] for more information on this class of declarative programming languages.

Logic Programming languages are programming languages based on logic, and the beginning of logic is tied with that of scientific thinking [SS94]. Logic provides a precise language for representing a goal, knowledge and assumptions. Moreover, logic allows consequences to be deduced from premises to study the truth or falsity of statements from the truth or falsity of others. This basis allows to develop a programming language with a clear distinction between declarative and operational semantics. Informally, the declarative interpretation (or semantics) of a program is concerned with the meaning, the results, while the procedural or operational interpretation is concerned with the method used to get the results. While in the declarative interpretation a program is viewed as a formula, and one can reason about its correctness without any reference to the underlying computational mechanism, in the procedural one it is viewed as a description of an algorithm that can be executed. The operational semantics of logic programs correspond to logical inference, while their declarative semantics are derived from the term model commonly referred to as the Herbrand base. This, in principle, makes declarative programs

easier to understand and to develop because you can focus on what to do without regarding on how to do it, but all that glitters is not gold: you have to ensure that the operational and the declarative semantics are equivalent and this is not always possible. In fact, the existing operational semantics of Prolog impose restrictions to some program constructions for which the declarative semantics do not, and this must be taken into account when developing programs. We enter this problem after describing the historical circumstances that facilitate the creation of the first logic programming compiler, "Marseille Prolog". We owe the historical material in the following two paragraphs to [PS87].

The basic ideas of logic programming emerged in the late 1960s and early 1970s from work on automated deduction. Proof procedures based on Robinson's resolution principle [Rob65] operate by building values for unknowns that make a problem statement a consequence of the given premises. Green [Gre69] observed that resolution proof procedures could thus in principle be used for computation. Resolution on its own is not a sufficient basis for logic programming, because resolution proof procedures may not be sufficiently goal-directed. Thus, Green's observations linking computation to deduction [Gre69] had no effective realization until the development of more goal-oriented linear resolution proof procedures, in particular Kowalski and Kuehner's SL resolution [KK71]. This development allowed Kowalski [Kow74a] to suggest a general approach to goal-directed deductive computation based on appropriate control mechanisms for resolution theorem provers and the further specialization of SL resolution to Horn clauses, and the corresponding procedural interpretation of Horn clauses, was first described in principle by Kowalski [Kow74a; Kow74b].

Even the SL resolution procedure and related theorem-proving methods were not efficient enough for practical computation, mainly because they had to cope with the full generality of first-order logic, in particular disjunctive conclusions. Further progress required the radical step of deliberately weakening the language to one that could be implemented with efficiency comparable to that of procedural languages. This step was mainly due to Colmerauer and his colleagues at Marseille in the early 1970s. Their work proceeded in parallel with (and in interaction with) the theoretical developments from the automated theorem-proving community. Inspired by his earlier Q-systems [Col70], a tree-matching phrase-structure grammar formalism, Colmerauer started developing a language that could at the same time be used for language analysis and for implementing deductive question-answering mechanisms. It eventually became clear that a particular kind of linear resolution restricted to definite clauses had just the right goal-directness and efficiency, and also enough expressive power for linguistic rules and some important aspects of

the question-answering problem. Their approach was first described as a tool for natural-language processing applications [AC73]. The resulting deductive system, supplemented with a few other computational devices, was the first Prolog system, known as "Marseille Prolog". The first detailed description of Prolog was the language manual for the Marseille Prolog interpreter [Rou75].

From the previous paragraphs we can deduce that Prolog is not as powerful as they wanted it to be: they had to weaken the language to obtain an efficiency comparable to that of procedural languages. The resulting system has two important restrictions. The first: programs must be formed by definite clauses and the query must be a Horn clause without a positive goal. The second: the operational semantics depend on encoding the "appropriate" control mechanisms.

A Horn clause is a disjunction of literals with at most one positive literal. Horn clauses are usually written as

$$L \leftarrow L_1,\ L_2,\ ...,\ L_n \qquad \left(L \vee \neg L_1 \vee \neg L_2 \vee ... \vee \neg L_n \right) \tag{1.2.1}$$

or

$$\leftarrow L_1,\ L_2,\ ...,\ L_n \qquad \left(\neg L_1 \vee \neg L_2 \vee ... \vee \neg L_n \right) \tag{1.2.2}$$

where $n \geq 0$ and L is the only positive literal. A *definite clause* is a Horn clause that has exactly one positive literal (Eq. 1.2.1). A Horn clause without a positive literal is called a goal (Eq. 1.2.2). Horn clauses express a subset of statements of first-order logic where there is no bi-implication, disjunction is represented by having more than one clause with the same head and the variables appearing in the head are universally quantified while the ones occurring only in the body are existentially quantified. The reason for this design decisions is that Prolog has an efficiency comparable with Imperative Programming Languages by restricting the language to Horn languages, but in some cases we might need something more: normal logic programs.

A normal logic program clause is a (first order) formula of the form

$$L \leftarrow L_1,\ L_2,\ ...,\ L_n,\ \neg L_{n+1},\ ....\ L_m \tag{1.2.3}$$

or

$$\leftarrow L_1,\ L_2,\ ...,\ L_n,\ \neg L_{n+1},\ ....\ L_m \tag{1.2.4}$$

where $n \geq 0$, $m \geq 0$ and L is always a positive literal. The positive literal to the left of $\leftarrow$ is called the clause's head and the formula to its right is called the clause's body. As in horn clauses, there is no bi-implication, disjunction is represented by having more than one clause with the same head and the variables appearing in the head are universally quantified while the ones occurring only in the body are existentially quantified. The only

difference is the allowance of the negation symbol in the syntax of the clauses' bodies. Although it seems to be a very small improvement, the inclusion of the negation symbol in the clause's bodies incredible increases the expressiveness of the language. As an example, consider we need to encode the universal quantification of a variable in the body. We can use the semantic equivalence $\forall\, \bar{X}\, .\, Formula \;\equiv\; \neg\, \exists\, \bar{X}\, .\, \neg\, Formula$ to convert the universal quantification into a existential one, and splitting the formula into four clauses we overcome the syntactic limitations:

$$\begin{aligned} p1 &\leftarrow \neg p2 & \qquad p3(X) &\leftarrow \neg p4(X) \\ p2 &\leftarrow p3(X) & \qquad p4(X) &\leftarrow Formula \end{aligned} \tag{1.2.5}$$

A normal logic program is a set of normal logic program clauses. A query is a clause without head, and solving a query consists in finding a substitution for the variables existentially quantified such that the program entails the query.

The "appropriate" control mechanisms on which the operational semantics depend to ensure efficiency, have the cost of converting LP into a not-fully declarative language. The difference between logic and control is attributed to Kowalski's paper "Algorithm = Logic + Control" [Kow79] although we prefer Kunen's definition in [Kun87]:

> The logic component should give us a declarative semantics of the language which tells us whether or not $\forall\phi\sigma$[1] indeed follows semantically from DB.
>
> The control component consists of guides to the Prolog compiler to help it decide whether or not a query is a logical consequence of DB.

Suppose the almost identical LP programs in Listings 1.2.1 and 1.2.2, representing the Peano (natural) numbers, and the query "$nat(Y)$", representing our request for a natural number to the interpreter. In a fully declarative LP system we should get the same answers for both programs, but we get $0, s(0), s(s(0)), s(s(s(0))), \ldots$ for the program in Listing 1.2.2 while for the one in Listing 1.2.1 the interpreters fail to halt or halt and print an error.

```
nat(s(X)) ← nat(X).
nat(0).
```

Listing 1.2.1: Peano Numbers (I)

[1] ϕ is a formula in Prolog's syntax, σ is a substitution and $\forall$ is the universal quantification.

```
nat(0).
nat(s(X)) ← nat(X).
```

Listing 1.2.2: Peano Numbers (II)

This incorrect result is caused by the fixed strategy that LP Interpreters implement. This strategy defines

(i) the order (usually Top-Down) in which the different program clauses are selected for solving the current goal (only necessary when the goal can be unified with the head of multiple clauses)

(ii) (after choosing a clause) which one is the literal in the clauses' body that we try to solve first (it is usually Left-To-Right, and only used if the clause's body has more than one literal) and

(iii) if our search for answers exhausts the selected clause before trying a different one (Depth-First, the usual one) or if it exhausts the current depth's level before trying the next one (Breadth-First).

LP interpreters suppose that the programmer writes programs taking into account the fixed strategy and, when the control guides are wrong, they loop until they reach the machine limits. An example of this is the query "$nat(Y)$" against the program in Listing 1.2.1. A LP interpreter

1. chooses the first clause in the program;
2. unifies the query with the clauses' head $nat(s(X))$, so now $Y = s(X)$;
3. selects the first atom in the clause, $nat(X)$;
4. tries to solve the new query, $nat(X)$.

As this one is a renaming of the original one it loops (repeats indefinitely the same four steps) until it reaches the machine limits. If this never happens then the LP interpreters fail to halt, but if this occurs then they halt and print an error.

In a nutshell, the logic programming paradigm regards a computation as automated reasoning over a corpus of knowledge, instead of actions that change the machine state in some way. Facts about the problem domain are expressed as logic formulas, and programs are executed by applying inference rules over them until an answer to the problem is found, or the collection of formulas is proved inconsistent. Nevertheless, Prolog is not a fully declarative programming language. It is more declarative than others, so it removes the necessity to specify the flow control in most cases, but the programmer still

needs to know if the interpreter or compiler implements depth or breadth-first search strategy and left-to-right or any other literal selection rule. Consider reading [SS94], [Llo87], [Kow88] or [CR93] for more information on this topic.

The human being has tried to mathematize everything around him since his origins. When representing the world as the human being understands it and how he takes decisions and interacts with the first one, we encounter the problem of representing fuzzy characteristics (it is hot), fuzzy rules (if it is hot, turn on the fan) and fuzzy actions (since it is not too hot, turn on the fan at medium speed). So, a machine needs all this information (or knowledge) if we want it to understand the world as the human being does and take decisions as the human being does. It is when carrying out this task that Lofti A. Zadeh found the necessity of fuzzy sets [Zad65] and, some years later, the necessity of linguistic variables [Zad75a; Zad75b; Zad75c]. A very good justification of their necessity can be found in his paper named “Is there a need for fuzzy logic?” [Zad08].

1.3 Fuzzy Logic

At first sight fuzziness and logic might be seen as very different things. Logic is usually understood as sentences which can be true or false and fuzziness as sentences which can take any valid value. It is not like that. On one hand logic is more than bi-valued logic. In fact many-valued logic is part of logic and allows infinite truth values. On the other one fuzziness measures the range of values of truthness that we can assign to some sentence and this range can be given a credibility so there is always a set of values that we can trust. Together we talk about fuzzy logic, logic that can be used to reason about fuzzy information and provide results with a credibility.

Take, for example, a database with the contents shown in Table. 1.3.1, the definition for the function “close” in Fig. 1.3.1 and the question “Is restaurant X close to the center?” with FL we can deduce that Il tempietto is “definitely” close to the center, Tapasbar is “almost” close, Ni Hao is “hardly” close and Kenzo is “not” close to the center. We highlight the words “definitely”, “almost”, “hardly” and “not” because the usual answers for the query are “1”, “0.9”, “0.1” and “0” for the individuals Il tempietto, Tapasbar, Ni Hao and Kenzo and the humanization of the numeric values is done in a subsequent step by defuzzification.

It was Lotfi Zadeh in 1965 who introduced fuzzy set theory [Zad65], and its existence was justified in his paper "Is there a need for fuzzy logic?" [Zad08]. Zadeh contributed much more to the fuzzy set theory, as the distinction between fuzzy sets without uncertainty and fuzzy sets with different levels of uncertainty

Table 1.3.1: Restaurants database

name	distance	price avg.	food type
Il_tempietto	100	30	italian
Tapasbar	300	20	spanish
Ni Hao	900	10	chinese
Kenzo	1200	40	japanese

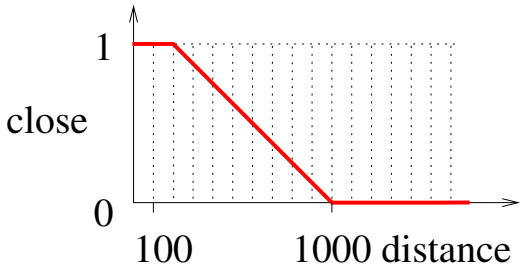


Figure 1.3.1: Close fuzzification function.

(see [Zad75a; Zad75b; Zad75c]), but we outline here just the ideas needed to make the contribution self contained.

It is usual when modeling real-world problems the necessity to represent not only if an individual belongs or not to a set, but the grade in which it belongs. This grade is what Zadeh tried to model by using a linguistic variable, a variable which can be assigned real-world adjectives[2] as values. For example, age takes the values young and old and temperature takes the values cold, warm and hot (Fig. 1.3.2).

But this linguistic variables are no more than part of the fuzzy systems

[2]Please take into account that values for a linguistic variable are not always adjectives.

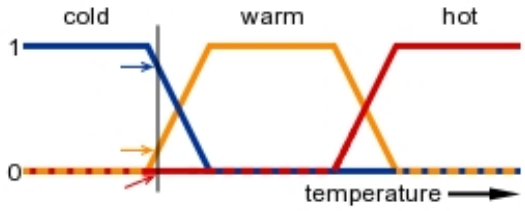


Figure 1.3.2: Temperature is a linguistic variable and (here) takes the values cold, warm and hot.

which, aimed at encoding in a computation the human way of solving problems, abstract the real-world facts into fuzzy facts by means of the fuzzification process, infer fuzzy solutions to the real-world problems by using fuzzy rules and defuzzify them to work in real-life scenarios.

To illustrate this description, suppose that temperature is 25 degrees and this measure is fuzzified into temperature(warm) by using the description for the linguistic variable temperature in Fig. 1.3.2. This fuzzy fact is then taken into account by the fuzzy rule if temperature(warm) then fan_speed(normal) and it is concluded fan_speed(normal). At last the conclusion is defuzzified by a linguistic variable description similar to the one in Fig. 1.3.2, obtaining the real-world decision to feed the fan with 5 volts[3].

Another example is trying to determine if a train will stop because of its speed decrease and we should take our baggage, but without knowing if the speed reduction is high, normal or low[4]. We could say that speed_reduction is normal, but we are not completely sure about it and this should be taken into account. For that purpose we measure how much we trust our perception by means of a real number in the interval $[0,\ 1]$[5], 0.6 here. This number is called the credibility of the rule. The fuzzification process result ("speed_reduction(normal) with cred 0.6") is then taken into account by the fuzzy rule "if train_speed_reduction(normal) then train_stops(soon)" to infer the conclusion "train_stops(soon) with cred 0.6", which is still not the solution expected by the deffuzification process. We need to apply another rule to this result[6], "if train_stops(soon) then take_your_baggage(now)" and the result here is "take_your_baggage(now) with cred 0.6". Finally the defuzzification process tries to deffuzify the last conclusion, but it is not strong enough to fire a real-world action. So, it does not suggest us to take our baggage.

1.3.1 Fuzzy Approaches in Logic Programming

Introducing Fuzzy Logic into Logic Programming resulted in the development of several fuzzy systems over Prolog. As Shapiro argues in [Sha83], Logic Programming has traditionally been used in knowledge representation and reasoning, which is why it is perfectly well-suited to implement fuzzy reasoning

[3] suppose that the linguistic variable fan_speed has the following description: when its value is fast we feed the fan with 10 volts, when normal with 5 volts, when low with 2.5 volts and when stop with 0 volts.

[4] Suppose we are just passengers that feel something but can not measure it.

[5] As usually, 0 means we do not trust the rule and 1 we trust it completely. We could use here a linguistic variable again, but we think a number between 0 and 1 makes it easy to understand.

[6] This second inference step is included to highlight that we can model problems much more complex than the previous one, solved in only three inference steps.

tools. It is just interesting to highlight that there is no common method for fuzzifying Prolog, as noted by Shen in [SDM89]. The only common denominator of all the existing fuzzy systems is that they implement in some way the fuzzy set theory introduced by Lotfi Zadeh in 1965 [Zad65].

Some of these systems replace its inference mechanism, SLD-resolution, with a fuzzy variant that is able to handle partial truth, introduced by Lee [Lee72].

Some 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), other systems just use Łukasiewicz logic [KK94] [7].

There is also an extension of constraint logic programming [BR01], 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 [Voj01], which deals with many-valued implications.

Leaving apart the theoretical frameworks, as [JMP05; Voj01], and focusing on frameworks having an implementation, the ones we knew about when we started this work were

- the Prolog-Elf system [IK85],
- the FRIL Prolog system [BMP95],
- the F-Prolog language [LL90],
- the FuzzyDL reasoner [BS08],
- the Fuzzy Logic Programming Environment for Research (FLOPER) [MM08a; MM08b], and
- the Fuzzy Prolog system [GMHV04; VGMH02].

1.3.2 Fuzzy Prolog

One of the most promising fuzzy tools for Prolog was the "Fuzzy Prolog" system [GMHV04; VGMH02]. The most important advantages in comparison to the other approaches are:

[7] A good survey on many-valued logics (Gödel logics, Łukasiewicz logic, Product logic) can be found in [Got01; Got05].

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 [KMP00] like min, prod, etc.), disjunctive operators [TCC95] (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 [PTC02]).

3. Crisp and fuzzy reasoning are consistently combined [MHVG02].

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

1.3.3 RFuzzy Approach Motivation

Over the last few years several papers have been published by Medina et al. ([MOAV01a; MOAV01b; MOAV01c]) about multi-adjoint programming. They describe a theoretical model, but no means of serious implementations apart from promising prototypes [AMM07] and recently the FLOPER tool [MM08a; MM08b; Mor06].

The FLOPER implementation is inspired by Fuzzy Prolog [GMHV04] and adds the possibility to use multi-adjoint logic. On the one hand, Fuzzy Prolog is more expressive in the sense that it can represent continuous fuzzy functions and its truth values are more general (unions of intervals of real numbers as opposed to real numbers); on the other hand, Fuzzy Prolog's syntax is so flexible and general that can be complex for non-expert programmers that are just interested in modelling simple fuzzy problems.

This is the reason why we herein propose the RFuzzy[8] approach. It is simpler for the user than Fuzzy Prolog because the truth values are simple real numbers instead of the general structures of Fuzzy Prolog. RFuzzy allows to model

[8] In RFuzzy's name, the "R" means Real, because the truth value that it uses is a real number instead of an interval or union of intervals as in Fuzzy Prolog. RFuzzy should not be confused with the term "R-Fuzzy set" [Wan+07; YH10] that means rough fuzzy set.

multi-adjoint logic and moreover provides some interesting improvements with respect to FLOPER:

- default values,
- partial default values (just for a subset of data),
- typed predicates,
- useful syntactic sugar (for representing facts, rules and functions),
- similarity between fuzzy predicates,
- similarity between attributes of non-fuzzy predicates,
- modifiers (even negation modifiers) and
- definition of new connectives and modifiers.

Additionally, RFuzzy inherits Fuzzy Prolog characteristics that makes it more expressive than other tools. Examples of this are:

- RFuzzy uses Prolog-like syntax, providing flexibility in the queries' syntax,
- it allows the programmer to combine crisp and fuzzy predicates,
- it provides general connectives to combine truth values and
- it provides constructive answers when querying data or truth values.

1.4 Multi-Adjoint Semantics

The structure used to give semantics to the programs written in the syntax accepted by RFuzzy is a particular case of the multi-adjoint (MA) algebra, presented by Medina, Ojeda-Aciego and Vojtáš in [MOAV01a; MOAV01b; MOAV01c; MOAV02; MOAV04; MO02]. The interest in using this structure is twofold: on one hand generalised logic programs' semantics require a notion of consequence (implication) different from the usual one[9]. The multi-adjoint algebra satisfies a generalised modus ponens rule and allows us to manage the rules' consequences in a very natural way. On the other one the multi-adjoint algebra allows us to obtain the credibility for the rules that we write from real-world data. There are other mechanisms for this purpose, as the one proposed

[9]The usual one is $(a \leftarrow b) \equiv \mathbf{t}$ iff $a \equiv \mathbf{t}$ whenever $b \equiv \mathbf{t}$.

by Palacios, Gacto and Alcalá-Fdez in [PGAF12], but we think that the one offered by the multi-adjoint algebra is more natural than the other ones.

As stated in the papers cited before, the multi-adjoint semantics are a common denominator of the residuated lattices and fairly general conjunctors and their adjoints. It allows several adjoint pairs in the residuated lattice, leading what Vojtáš calls multi-adjoint algebra in [Voj01] and what Medina, Ojeda-Aciego and Vojtáš call multi-adjoint (semi-) lattice in [MOAV04]. We copy here the definitions of these three concepts together with the definitions needed to understand them (all of them taken from [MOAV01c]). The definition of the multi-adjoint algebra (the basis of the multi-adjoint semantics) is straightforward from the definitions of adjoint pair and multi-adjoint lattice.

Definition 1.4.1 (Graded set, from [MOAV01c]). *A graded set is a set Ω with a function which assigns to each element $\omega \in \Omega$ a number $n \geq 0$, called the arity of ω.*

Definition 1.4.2 (Ω-Algebra, from [MOAV01c]). *Given a graded set Ω, an Ω-Algebra $\mathcal{U}$ is a pair $< A, I >$ where A is a nonempty set called the carrier, and I is a function which assigns maps to the elements of Ω as follows:*

1. *Each element $\omega \in \Omega_n$, $n > 0$, is interpreted as a map $I(\omega) : A^n \to A$, denoted by $\Omega_{\mathcal{U}}$.*

2. *Each element $c \in \Omega_0$ (i.e., c is a constant) is interpreted as an element $I(c)$ in A, denoted by $c_{\mathcal{U}}$.*

Definition 1.4.3 (Adjoint Pair, from [MOAV01c]. Firstly introduced in a logical context by Pavelka [Pav79]). *Let $<P, \preceq>$ be a partially ordered set and $(\leftarrow, \&)$ a pair of binary operations in P such that:*

1. *Operation $\&$ is increasing in both arguments, i.e. if $x_1, x_2, y \in P$ such that $x_1 \preceq x_2$ then $(x_1 \,\&\, y) \preceq (x_2 \,\&\, y)$ and $(y \,\&\, x_1) \preceq (y \,\&\, x_2)$;*

2. *Operation $\leftarrow$ is increasing in the first argument (the consequent) and decreasing in the second argument (the antecedent), i.e. if $x_1, x_2, y \in P$ such that $x_1 \preceq x_2$ then $(x_1 \leftarrow y) \preceq (x_2 \leftarrow y)$ and $(y \leftarrow x_2) \preceq (y \leftarrow x_1)$;*

3. *For any $x, y, z \in P$, we have that $x \preceq (y \leftarrow z)$ holds if and only if $(x \,\&\, z) \preceq y$ holds.*

Then $(\leftarrow, \&)$ is called an adjoint pair in <P, $\preceq$>.

Definition 1.4.4 (Multi-Adjoint Lattice, from [MOAV01c]). *Let $< L, \preceq >$ be a lattice. A multi-adjoint lattice L is a tuple*

$$(L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n) \tag{1.4.1}$$

satisfying the following items:

1. *$< L, \preceq >$ is bounded, i.e. it has bottom ($\bot$) and top ($\top$) elements;*
2. *$(\leftarrow_i, \&_i)$ is an adjoint pair in $< L, \preceq >$ for $i = 1, \ldots, n$;*
3. *$\top \,\&_i\, v \;=\; v \,\&_i\, \top \;=\; v$ for all $v \in L$ for $i = 1, \ldots, n$.*

Remark. A lattice with only one adjoint pair is called somewhere a residuated lattice (see [DP00; DP01]), and when more that one pair is introduced we get to a multi-adjoint lattice.

Definition 1.4.5 (Multi-Adjoint Ω-Algebra, from [MOAV01c]). *Let Ω be a graded set containing operators $\leftarrow_i$ and $\&_i$ for $i = 1, \ldots, n$ and possibly some extra operators, and let $\mathcal{L} = (L, I)$ be an Ω-algebra whose carrier set L is a lattice under $\preceq$.*

We say that $\mathcal{L}$ is a multi-adjoint Ω-algebra with respect to the pairs $(\leftarrow_i, \&_i)$ for $i = 1, \ldots, n$ if

$$\mathcal{L} = (L, \preceq, I(\leftarrow_1), I(\&_1), \ldots, I(\leftarrow_n), I(\&_n)) \tag{1.4.2}$$

is a multi-adjoint lattice.

From this definitions the important part is the relation between adjoint pairs $(\leftarrow_i, \&_i)$, which makes it possible to evaluate in the many-valued modus ponens the truth value (y) of the head (A) of a rule $(A \leftarrow_i B)$ from the truth value (z) of the body (B) and the weight (v) of the rule:

$$\frac{(B,\ z),\ (A \leftarrow_i B,\ v)}{(A,\ y)}, \tag{1.4.3}$$

$$v \preceq (y \leftarrow_i z) \quad \text{iff} \quad (v \,\&\, z) \preceq y \tag{1.4.4}$$

This relation is used to define *satisfaction* (Def. 1.4.7), and the *immediate consequences operator* (Def. 1.4.8). We include first the definition of multi-adjoint logic program (Def. 1.4.6), needed to understand them[10].

[10] This definition is not the one in [MOAV01c]. We just changed some syntactic constructions to avoid explaining the syntax used in [MOAV01c].

Definition 1.4.6 (Multi-Adjoint Logic Program). *A multi-adjoint logic program is a set of clauses of the form*

$$< A \leftarrow_i F(B_1, \ldots, B_n), c > \tag{1.4.5}$$

where $c \in [0, 1]$ *is the credibility assigned to the rule,* i *a conjunctor* &*,* F *an aggregator* $@_i$ *and* A *and* B_i*,* $i \in [1..n]$*, atoms. When the satisfiability of* A *depends on the satisfiability of only one atom* B *the aggregator* F *is omitted, resulting in* $< A \leftarrow_i B, v >$*. This representation is used too when referring to a rule in which we do not know if the atom* A *depends on only one atom* B *or more than one.*

Definition 1.4.7 (Satisfaction, from [MOAV01c]). *Given an interpretation* $I \in \mathcal{I}_{\mathcal{L}}$*, where* $\mathcal{I}_{\mathcal{L}}$ *is the set of all interpretations of the formulas defined by the* Ω*-algebra* $\mathcal{F}$ *in the* Ω*-algebra* $\mathcal{L}$*, a weighted rule* $< A \leftarrow_i B, v >$ *is satisfied by I if and only if (iff)* $v \preceq \hat{I}(A \leftarrow_i B)$*.*

Definition 1.4.8 (Immediate consequences operator $\mathcal{T}_{\mathcal{P}}^{\mathcal{L}}$, from [MOAV01c]). *Let* $\mathcal{P}$ *be a multi-adjoint logic program. The immediate consequences operator* $\mathcal{T}_{\mathcal{P}}^{\mathcal{L}} : \mathcal{I}_{\mathcal{L}} \rightarrow \mathcal{I}_{\mathcal{L}}$*, mapping interpretations to interpretations, is defined by considering*

$$\mathcal{T}_{\mathcal{P}}^{\mathcal{L}}(I)(A) = sup \{ \; v \,\hat{\&}_i \, \hat{I}(B) \;\mid\; < A \leftarrow_i B, v > \in \mathcal{P} \; \}$$

The definitions of *satisfaction* (Def. 1.4.7), and *immediate consequences operator* (Def. 1.4.8) are used to give meaning the programs, as usually. Suppose a program in the syntax the papers cited define and a query. In order to give an answer to the query the program is instantiated or grounded, the atoms are given an interpretation and this interpretation is extended to the formulas in the language. In bi-valued logic the interpretations are just $\{ true, false \}$ while here it is (usually) a number $v \in [0, 1]$, but in both cases the expected result is the maximum (or supreme) of the values obtained by the different rules. So, we select all the rules whose head unify with the query, take their respective interpretations and compute the maximum. This is the result for the query.

We provide the following examples to illustrate the theoretical concepts introduced before.

Example 1.4.1 Suppose we have four fuzzy facts, A, B, C and D. D is always satisfied with at least the maximum truth value of A, B, and C. From this knowledge we can extract the following rule:

$$D \xleftarrow{1,\ Gödel} max(A, B, C) \tag{1.4.6}$$

The interesting point is that the rule's credibility value, 1, has been computed from the real-world data. From Gödel's implication operator definition

$$b \xleftarrow{X,\ Gödel} a = \left\{ \begin{array}{lll} 1 & \text{if} & a \leq b \\ b & \text{if} & b < a \end{array} \right\} \tag{1.4.7}$$

and knowing that the satisfaction of D is always higher than the satisfaction of A, B and C we can obtain the rule's credibility value.

Example 1.4.2 Suppose we have four fuzzy facts, A, B, C and D and the rule in Eq. 1.4.6. Knowing that A, B and C are satisfied with, at least, the values 0.3, 0.4 and 0.5 we can compute how much satisfied is D:

$$\hat{D} = min(1,\ max(0.3, 0.4, 0.5)\ )\ =\ 0.5 \tag{1.4.8}$$

1.5 Structure of the Work

In the following pages we present the contributions of this thesis, published in a set of research papers. Chapter. 2 corresponds to [MHPCS11], Chapter. 3 to [PCMH11] and Chapter. 4 to [PCMH15]. Chapter. 5 corresponds to the real applications developed using RFuzzy and their impact. It contains material belonging to [MHPCS11] and [PCMH15].

The papers [MHPCS11] and [PCMH15] are respectively improved versions of the publications [MHPCS09; PCMHS08; PCSMH09; SMHPC09] and [PCMH14a; PCMH14b; PCMH14c; PCMH14d; PCMH14e], which is why we have taken them instead of the original ones.

The three research papers chosen to build the thesis [MHPCS11; PCMH11; PCMH15] belong to the research idea of increasing the expressiveness of fuzzy logic languages. To achieve it we have developed a framework, RFuzzy. RFuzzy is a framework with a clear and easy syntax that allows us to encode fuzzy logic in programs and obtain results to fuzzy queries.

The first part of the work (Chapter. 2) corresponds to the first version of RFuzzy. In this part RFuzzy allows to encode fuzzy rules, to obtain default truth values when no better value can be computed and to encode the translation between crisp and fuzzy values by using piecewise functions. In this part we solve problems as obtaining the correct truth value when more than one rule is applicable. This is needed mainly due to the existence of rules for encoding "default truth values", truth values that are returned by the system when no better value can be computed. After an informal introduction we include too the syntax and semantics developed, in which an incipient priorities system helps to provide plenty of facilities, including the ones mentioned before.

In the second part (Chapter. 3) we develop a priorities system for RFuzzy much more powerful than the previous one. The main difference with respect to the previous one is that this one is not restricted to the usage of just three symbols, allowing the introduction of an infinite amount of priorities and the assignment of different priorities to each construction. With its inclusion RFuzzy gain, between others, the capability to define personalized rules, rules that work

differently depending on the user posing the query. In this chapter we provide too the syntax and semantics developed, focusing in the mechanism used to translate each one of the constructions created using the new priorities system.

In the third part of the work (Chapter. 4) we include syntactic structures to model synonyms, antonyms, similarity between attributes (allowing to search for a Mediterranean restaurant and get Spanish restaurants as valid answers), and the use and definition of modifiers (very, too much or even negation). In this part we have focused in providing a negation mechanism with a clear syntax and semantics, which is why the semantics have changed so much from the original ones.

In the three chapters mentioned (Chapter. 2, Chapter. 3 and Chapter. 4) we provide detailed information about the three releases of the framework developed: syntax, semantics and links to the source code publicly available.

In Chapter. 5 we present some applications that have been developed using RFuzzy. We highlight two between all of them: Emotion Recognition and FleSe (Flexible Searches in databases). Emotion recognition had a lot of impact and some companies show interest on how it works. FleSe is an easy-to-use web interface framework to pose fuzzy queries in almost natural language. It makes use of all the facilities provided by RFuzzy. It is a good example to show all the capabilities that RFuzzy offers to the final user.

At last it is worth to highlight that no source code is included to increase the weight of this document. In contrast and since everything is free and public available, we provide links to it. The links point to the open-source code available and we include a link to a working installation of FleSe. This working installation is automatically updated from FleSe's last release and allows users to pose fuzzy queries to (their, our or somebody's) non-fuzzy databases.

Chapter 2

The initial version of the fuzzy logic framework: RFuzzy v.1

The RFuzzy framework is a Prolog-based tool for representing and reasoning with fuzzy information. Its advantages in comparison to previous tools along this line of research are its easy, user-friendly syntax, and its expressivity through the availability of default values and types.

Here we describe the formal syntax, the operational semantics and the declarative semantics of RFuzzy (based on a lattice). A least model semantics, a least fixpoint semantics and an operational semantics are introduced and their equivalence is proven. We provide a real implementation that is free and available.[1]

We start by introducing the formal syntax of RFuzzy (Section 2.1) and its declarative and operational semantics (Sections 2.2 and 2.3, respectively), to arrive at the justification of the sentence "RFuzzy allows to model multi-adjoint logic" (Section 2.4). Afterwards we board RFuzzy implementation and usage (Section 2.5).

2.1 Syntax

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 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

[1] It can be downloaded from [PC15d]

arguments are elements of $TU_{\Sigma,V}$. Atoms and terms are called *ground* if they do not contain variables. As usual, the *Herbrand universe* $\mathbb{HU}$ is the set of all ground terms, and the *Herbrand base* $\mathbb{HB}$ is the set of all atoms with arguments from the Herbrand universe. To capture different interdependencies between predicates, we will make use of a signature Ω of *many-valued connectives*[2]:

- conjunctions $\&_1, \&_2, ..., \&_k$
- disjunctions $\vee_1, \vee_2, ..., \vee_l$
- implications $\leftarrow_1, \leftarrow_2, ..., \leftarrow_m$
- aggregations $@_1, @_2, ..., @_n$
- real numbers $v \in [0,\ 1] \subset \mathbb{R}$. These connectives are of arity 0 ($v \in \Omega^{(0)}$) and symbolise dependency on no other predicate.

While Ω denotes the set of connective symbols, $\hat{\Omega}$ denotes the set of their respective associated truth functions. Instances of connective symbols and truth functions are denoted by F and $\hat{F}$ respectively.

Truth functions for conjunctions are conjunctors $\hat{F} : [0,1]^2 \to [0,1]$ monotone and non-decreasing in both coordinates. Truth functions for disjunctions are disjunctors $\hat{F} : [0,1]^2 \to [0,1]$ monotone in both coordinates. Truth functions for implications are implicators $\hat{F} : [0,1]^2 \to [0,1]$ non-increasing in the first and non-decreasing in the second coordinate. Truth functions for aggregation operators are functions $\hat{F} : [0,1]^n \to [0,1]$ that verify $\hat{F}(0,\ \ldots,\ 0) = 0$ and $\hat{F}(1,\ \ldots,\ 1) = 1$. At last, truth functions for connectives $v \in \Omega^{(0)}$ are functions of arity 0 (constants) that coincide with the connectives ($\hat{F} = F$).

A n-ary truth function for a connective is called *monotonic in the i-th argument* ($i \leq n$), if $x \leq x'$ implies

$$\hat{F}(x_1,\ \ldots,\ x_{i-1},\ x,\ x_{i+1},\ \ldots,\ x_n) \leq \hat{F}(x_1,\ \ldots,\ x_{i-1},\ x',\ x_{i+1},\ \ldots,\ x_n).$$

A truth function is called *monotonic* if it is monotonic in all arguments.

Immediate examples for connectives that come to mind are

- for conjunctors: Łukasiewicz logic ($\hat{F}(x,y) = max(0, x+y-1)$), Gödel logic ($\hat{F}(x,y) = min(x,y)$), product logic ($\hat{F}(x,y) = x \cdot y$).
- for disjunctors: Łukasiewicz logic ($\hat{F}(x,y) = min(1, x+y)$), Gödel logic ($\hat{F}(x,y) = max(x,y)$), product logic ($\hat{F}(x,y) = x \cdot y$).

[2]Note that in Fuzzy Prolog (and in previous RFuzzy works) the term "aggregation operator" subsumes conjunctions, disjunctions and aggregations. In this work we distinguish between them and include a new one (implications).

- for implicators: Łukasiewicz logic ($\hat{F}(x,y) = min(1, 1-x+y)$), Gödel logic ($\hat{F}(x,y) = y$ if $x > y$ else 1), product logic ($\hat{F}(x,y) = x \cdot y$).
- for aggregation operators[3]: arithmetic mean, weighted sum or a monotone function learned from data.

Definition 2.1.1. *Let Ω be a connective signature, Π a predicate signature, Σ a term signature and V a set of variables.*

A fuzzy clause *is written as*

$$A \xleftarrow{c,F_c} F(B_1, \ldots, B_n)$$

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 $F_c \in \{\&_1, \ldots, \&_k\} \subset \Omega^{(2)}$ and $F \in \Omega^{(n)}$ are connectives symbols (for the credibility value and the body, respectively)

A fuzzy fact *is a special case of a clause where $c = 1$, F_c is the usual multiplication of real numbers "$\cdot$" and $n = 0$ (thus $F \in \Omega^{(0)}$). It is written as $A \longleftarrow F$ (we omit c and F_c).*

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 connective associated to the clause to yield the truth value for the head atom.

Example 1. Consider the following clause, that models to what extent cities can be deemed good travel destinations – the quality of the destination depends on the weather and the availability of sights:

$$\text{good-destination(X)} \xleftarrow{1.0,\cdot} \cdot\,(\text{nice-weather(X)}, \text{many-sights(X)})\ .$$

The credibility value of the rule is 1.0, which means that we have no doubt about this relationship. The connective used here in both cases is the usual multiplication of real numbers. We enrich the knowledge base with facts about some cities and their continents:

$$\begin{array}{ll} \text{nice-weather(madrid)} \longleftarrow 0.8\ , & \text{many-sights(madrid)} \longleftarrow 0.6\ , \\ \text{nice-weather(istanbul)} \longleftarrow 0.7\ , & \text{many-sights(istanbul)} \longleftarrow 0.7\ , \\ \text{nice-weather(moscow)} \longleftarrow 0.2\ , & \text{many-sights(sydney)} \longleftarrow 0.6\ , \end{array}$$

[3]Note that the above definition of aggregation operators subsumes all kinds of minimum, maximum or mean operators.

city-continent(madrid, europe) $\longleftarrow$ 1.0 ,
city-continent(moscow, europe) $\longleftarrow$ 1.0 ,
city-continent(sydney, australia) $\longleftarrow$ 1.0 ,
city-continent(istanbul, europe) $\longleftarrow$ 0.5 ,
city-continent(istanbul, asia) $\longleftarrow$ 0.5 .

Some queries to this program could ask if Madrid is a good destination, $\langle$good-destination(madrid), v$\rangle$. Another query could check if Istanbul is the perfect destination, $\langle$good-destination(istanbul), 1.0$\rangle$.The result of the first query will be the real value 0.48 and the second one will fail. It can be seen that no information about the weather in Sydney or sights in Moscow is available although these cities are "mentioned". $\diamond$

In the above example, the knowledge that we represented using fuzzy clauses and facts was not only vague but moreover incomplete. Indeed, this is the general case in real applications. Information is sometimes missing for some or all of the cases and it is necessary to provide an alternative semantics for these situations. The open world assumption (OWA) introduces the concept of unknown or absent information. It is very common in logic programming (for example in Prolog programming) to use the closed world assumption (CWA) for supposing false information that is not explicitly present or not derivable from the program. There are other works (like [LS05]) that allow *any* assumption.

We have chosen to use CWA enriched with a mechanism of default rules for assigning default values when information about truth values is absent in a program. In particular, we have been able to provide a rich syntax for defining default values to subsets of elements satisfying a particular condition.

Definition 2.1.2. *A* default value declaration *for a predicate* $p \in \Pi^{(n)}$ *is written as*

$$\texttt{default}(p(X_1, \ldots, X_n)) \;=\; [\delta_1 \textit{ if } \varphi_1,\; \ldots,\; \delta_m \textit{ if } \varphi_m]$$

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* $\mathbf{t}$ *and the conjunction* $\wedge$ *and disjunction* $\vee$ *in their usual meaning.*

Example (continued). Let us add the following default value declarations to the knowledge base and thus close the mentioned gaps.

$$\texttt{default}(\text{nice-weather(X)}) \;=\; 0.5,$$
$$\texttt{default}(\text{many-sights(X)}) \;=\; 0.2,$$

$$\texttt{default}(\text{good-destination(X)}) \;=\; 0.3$$

They could be interpreted as: when visiting an arbitrary city of which nothing further is known, it is likely that you have nice weather but you will less likely find many sights. Irrespective of this, it will only to a small extent be a good travel destination.

To model the fact that a city is not on a continent unless stated otherwise, we add another default value declaration for city-continent:

$$\texttt{default}(\text{city-continent(X, Y)}) \;=\; 0.0.$$

Notice that in this example $m = 1$ and $\varphi_1 = \mathbf{t}$ for all the default value declarations. ◇

The default values allow our knowledge base to answer arbitrary questions about predicates that occur in it. But will the answers always make sense? To stay in the above example, if we ask a question like "What is the truth value of nice-weather(australia)?" we will get the answer "0.5" which does not make too much sense since Australia is not a city, but a continent.

To address this issue, we introduce types into the language. Types in RFuzzy are subsets of terms of the Herbrand base of the program. When we assign types to the arguments of a predicate, we are restricting their domains. This contrasts with [Sch+08] and similar works, most of them variants of the proposal of Mycroft and O'Keefe [MO84] that was an adaption of the type system of Hindley and Milner [Mil78]. These works are more oriented to input/output type checking while in RFuzzy types are used for the constructive generation of answers (see Section 2.5.2).

Definition 2.1.3. *Types are built from terms $t \in \mathbb{HU}$. A* term type declaration *assigns a type $\tau \in \mathcal{T}$ to a term $t \in \mathbb{HU}$ 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. The set composed by all term type declarations and predicate type declarations is denoted by T.*

Example (continued). Using the set of types $\mathcal{T} = \{\text{City}, \text{Continent}\}$, we add some term type declarations to our knowledge base:

madrid : City,
istanbul : City,
sydney : City,
moscow : City;
africa : Continent,

$$\begin{aligned} \text{america} &: \text{Continent}, \\ \text{antarctica} &: \text{Continent}, \\ \text{asia} &: \text{Continent}, \\ \text{europe} &: \text{Continent}. \end{aligned}$$

We also type the predicates in the obvious way (i.e. providing the predicate type declarations):

$$\begin{aligned} \text{nice-weather} &: (\text{City}), \\ \text{many-sights} &: (\text{City}), \\ \text{good-destination} &: (\text{City}), \\ \text{city-continent} &: (\text{City}, \text{Continent}). \end{aligned}$$

◇

For a ground atom $A = p(t_1, \ldots, t_n) \in \mathbb{HB}$ we say that it is *well-typed with respect to* T iff $p : (\tau_1, \ldots, \tau_n) \in T$ implies $(t_i : \tau_i) \in T$ for $1 \leq i \leq n$. For a ground clause $A \xleftarrow{c,F_c} F(B_1, \ldots, B_m)$ we say that it is well-typed w.r.t. T iff A is well-typed whenever all B_i are well-typed for $1 \leq i \leq m$ (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.

Example (continued). With respect to the given type declarations, city-continent(moscow, antarctica) is well-typed, city-continent(asia, europe) is not. ◇

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 connective symbols.

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

Definition 2.1.4. *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*

$$\texttt{default}(p(X_1,\ldots,X_n)) \;=\; [\delta_1 \textit{ if } \varphi_1,\; \ldots,\; \delta_m \textit{ if } \varphi_m],$$

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.

Besides, a well-defined fuzzy logic program can not have clauses that depend, directly or indirectly (i.e. via other clauses) on themselves. This is considered a program error and neither our semantics nor our implementation is capable of dealing with this kind of programs.

2.2 Declarative Semantics

The possibility to define default truth values for predicates offers us a great deal of flexibility and expressivity. But it also has its drawbacks: reasoning with defaults is inherently non-monotonic – we might have to withdraw some conclusions that have been made in an earlier stage of execution. To capture this formally, we attach to each truth value an attribute that indicates how this value has been concluded. These attributes could be ordered numbers (as in [TB04] where real numbers in $[0,1]$ are used) but we decided, for clarity reasons, to restrict the possible scenarios to three cases that are characterised by three different symbols depending on the origin of the truth values:

- exclusively by application of program facts and clauses, represented by the symbol ▼ denoting the attribute value *safe*,
- by indirect use of default values, represented by the symbol ⧫ denoting the attribute value *unsafe (mixed)*, or
- directly via a default value declaration, represented by the symbol ▲ denoting the attribute value *unsafe (pure)*.

We need to be able to compare the attributes (in order to be able to prefer one conclusion over another) and to combine them to keep track of default value usage in the course of computation. This is formalised by setting the ordering $<_a$ on truth value attributes such that ▲ $<_a$ ⧫ $<_a$ ▼.

The operator $\circ : \{▲, ⧫, ▼\} \times \{▲, ⧫, ▼\} \to \{⧫, ▼\}$ is then defined as:

$$x \circ y := \begin{cases} ▼ & \text{if } x = y = ▼ \\ ⧫ & \text{otherwise} \end{cases}$$

The operator $\circ$ is designed to keep track of attributes during computation: only when two "safe" truth values are combined, the result is known to be "safe", in all other cases it is "unsafe". It should be noted that "$\circ$" is monotonic.

The truth values that we use in the description of the semantics will be real values $v \in [0,1]$ with an attribute (i.e. a $z \in \{\blacktriangle, \blacklozenge, \blacktriangledown\}$) attached to it. We will write them as zv, and the set of possible truth values as $\mathbb{T}$. So, $zv \in \mathbb{T}$. The ordering $\preccurlyeq$ on the truth values will be the lexicographic product of $<_a$, the ordering on the attributes, and the standard ordering $<$ of the real numbers. The set of truth values is thus totally ordered as follows:

$$\bot \prec \blacktriangle 0 \prec \cdots \prec \blacktriangle 1 \prec \blacklozenge 0 \prec \cdots \prec \blacklozenge 1 \prec \blacktriangledown 0 \prec \cdots \prec \blacktriangledown 1.$$

We remark that the pair $(\mathbb{T}, \preccurlyeq)$ forms a complete lattice.

A *valuation* $\sigma : V \to \mathbb{HU}$ is an assignment of ground terms to variables. Each valuation σ uniquely constitutes a mapping $\hat{\sigma} : \mathrm{TB}_{\Pi,\Sigma,V} \to \mathbb{HB}$ that is defined in the obvious way.

A *fuzzy Herbrand interpretation* (or short, *interpretation*) of a fuzzy logic program is a mapping $I : \mathbb{HB} \to \mathbb{T}$ that assigns truth values to ground atoms. The *domain* of an interpretation is the set of all atoms in the Herbrand Base, although for readability reasons we usually omit those atoms to which the truth value $\bot$ is assigned (interpretations are total functions). This mapping can be seen as a set of pairs (A, zv) such that $A \in \mathbb{HB}$ and $zv \in \mathbb{T} \setminus \{\bot\}$, meaning for an atom not being in the set that its truth value is $\bot$.

For two interpretations I and J, we say I *is less than or equal to* J, written $I \sqsubseteq J$, iff $I(A) \preccurlyeq J(A)$ for all $A \in \mathbb{HB}$. Two interpretations I and J are *equal*, written $I = J$, iff $I \sqsubseteq J$ and $J \sqsubseteq I$. Minimum and maximum for interpretations are defined from $\preccurlyeq$ as usually.

Accordingly, the infimum (or intersection) and supremum (or union) of interpretations are, for all $A \in \mathbb{HB}$, defined as $(I \sqcap J)(A) := \min(I(A), J(A))$ and $(I \sqcup J)(A) := \max(I(A), J(A))$.

The pair $(\mathcal{I}_P, \sqsubseteq)$ of the set of all interpretations of a given program with the interpretation ordering forms a complete lattice. This follows readily from the fact that the underlying truth value set $\mathbb{T}$ forms a complete lattice with the truth value ordering $\preccurlyeq$.

Definition 2.2.1 (Model)**.** *Let $P = (R, D, T)$ be a fuzzy logic program.*

For a clause $r \in R$ of the form $A \xleftarrow{c,F_c} F(B_1, \ldots, B_n)$ we say that I is a model of the clause *r and write*

$$I \Vdash A \xleftarrow{c,F_c} F(B_1, \ldots, B_n)$$

iff for all valuations σ, we have:

$$\textit{if } I(\sigma(B_i)) = z_i v_i \succ \bot \textit{ for all } i \textit{ then}^{4} I(\sigma(A)) \succcurlyeq z'v'$$

where $z' = z_1 \circ \cdots \circ z_n$ *and* $v' = \hat{F}_c(c, \hat{F}(v_1, \ldots, v_n))$.

For a clause $r \in R$ *of the form* $A \longleftarrow v$ *we say that* I is a model of the clause r *and write*

$$I \Vdash A \longleftarrow v$$

iff for all valuations σ, *we have* $I(\sigma(A)) \succcurlyeq \blacktriangledown v$.

For a default value declaration $d \in D$ *we say that* I is a model of the default value declaration d *and write*

$$I \Vdash \texttt{default}(p(X_1, \ldots, X_n)) \;=\; [\delta_1 \textit{ if } \varphi_1, \ \ldots, \ \delta_m \textit{ if } \varphi_m]$$

iff for all valuations σ, *we have:*

$$I(\sigma(p(X_1, \ldots, X_n))) \succcurlyeq \blacktriangle \delta_j$$

where φ_j *is the only formula that holds for* $\sigma(\varphi_j)$ $(1 \leq j \leq m)$. *Notice that, as* $\delta_i \in [0,1]$ *for all* i *(see Definition 2.1.2) and* $zv = \blacktriangle \delta_j$, *we force the truth values defined in defaults to be different from* $\bot$ *and lower than* $\blacklozenge 0$.

We write $I \Vdash R$ *if* $I \Vdash r$ *for all* $r \in R$ *and similarly* $I \Vdash D$ *if* $I \Vdash d$ *for all* $d \in D$. *Finally, we say that* I is a model of the program P *and write* $I \Vdash P$ *iff* $I \Vdash R$ *and* $I \Vdash D$.

Proposition 2.2.1. *Let* $P = (R, D, T)$ *be a well-defined fuzzy logic program and* I *a model of* P, $I \Vdash P$. *For all ground atoms* A *in* $\mathrm{TB}_{\Pi,\Sigma,V}$ *we can assume that*

$$I(A) \succ \bot$$

Proof. Trivial from the definitions of well-defined fuzzy logic program and model: For being a well-defined fuzzy logic program every predicate has a default value declaration for every ground atom A in $\mathrm{TB}_{\Pi,\Sigma,V}$, so we have a default value declaration

$$\texttt{default}(p(X_1, \ldots, X_n)) \;=\; [\delta_1 \text{ if } \varphi_1, \ \ldots, \ \delta_m \text{ if } \varphi_m]$$

such that $\sigma(p(X_1, \ \ldots, \ X_n)) = A$ and $\sigma(\varphi_i)$ is the only φ_i that holds. By $I \Vdash D$, this implies that

$$I(A) = \blacktriangle \delta_i$$

As for $I \Vdash P$ we need $I \Vdash D$ and this implies $I \Vdash d$ for every d, we have that $I(A) \succcurlyeq \blacktriangle \delta_i$, and $\blacktriangle \delta_i \succ \bot$. □

[4] Note that here we use 'then' and not 'iff', so if $I(\sigma(B_i)) = z_i v_i = \bot$ then the value of $I(\sigma(A))$ is not restricted but $I \Vdash A \overset{c,F_c}{\longleftarrow} F(B_1, \ldots, B_n)$.

Note that this proposition allows us to assure that a computation will not stop when the truth value of some atom is unknown in some interpretation. It is not possible (due to default value definitions) to have a truth value unknown for an atom. The following proposition affirms that a default value declaration is not used when a clause can be used to determine the truth value of an atom.

Proposition 2.2.2. *Let $P = (R, D, T)$ be a well-defined fuzzy logic program, I a model of P, $I \Vdash P$ and A' a ground atom such that $A' \in \mathrm{TB}_{\Pi,\Sigma,V}$. For a clause $r \in R$ of the form*

$$A \xleftarrow{c,F_c} F(B_1, \ldots, B_n) \in R \qquad or \qquad A \longleftarrow v \in R$$

and a default declaration

$$\texttt{default}(p(X_1, \ldots, X_n)) = [\delta_1 \textit{ if } \varphi_1, \ \ldots, \ \delta_m \textit{ if } \varphi_m] \in D$$

such that for some valuations σ_1 and σ_2 we have

$$\sigma_1(A) = A' = \sigma_2(p(X_1, \ \ldots, \ X_n))$$

and the rule $r \in R$ (if r is of the form $A \xleftarrow{c,F_c} F(B_1, \ldots, B_n)$) fulfills

$$I(\sigma_1(B_i)) = z_i v_i \succ \bot \textit{ for all } i$$

and the default declaration fulfills

$$\sigma_2(\varphi_i) \textit{ holds}$$

then

$$I(A') \succcurlyeq z'v' \succ \blacktriangle\delta_i$$

where $z'v'$ are obtained as in Definition 2.2.1.

Proof. Trivial from the definition of model of well-defined fuzzy logic program and the definition of the operator $\circ$. As the image of the operator $\circ$ is $\{\blacklozenge, \blacktriangledown\}$ and for rules $r \in R$ of the form $A \longleftarrow v$ we have that $z' = \blacktriangledown$ it is easy to see that $z' \in \{\blacklozenge, \blacktriangledown\}$. So, truth values obtained from clauses (when applicable) have always a truth value higher than the one obtained by the application of defaults:

$$\blacklozenge \succ \blacktriangle \quad \text{and} \quad \blacktriangledown \succ \blacktriangle$$

□

This is the intended meaning, so the lack of information on the truth value of some atom never stops a computation if an approximate value for it (obtained from a default declaration) can be used, but the approximate value is never preferred over a truth value determined by a rule.

2.2.1 Least Model Semantics

Every program has a least model, which is usually regarded as the intended interpretation of the program, since it is the most conservative model. The following proposition will be an important prerequisite to define the least model semantics. It states that the infimum (or intersection) of a non-empty set of models of a program will again be a model. The existence of a least model is then obvious and easily defined as the intersection of all models.

Proposition 2.2.3 (Model intersection property)**.** *Let $P = (R, D, T)$ be a well-defined fuzzy logic program and $\mathcal{I}$ be a non-empty set of interpretations. Then*

$$I \Vdash P \text{ for all } I \in \mathcal{I} \text{ implies } \bigsqcap_{I \in \mathcal{I}} I \Vdash P$$

Proof. We start by defining $J = \bigsqcap_{I \in \mathcal{I}} I$. From Definition 2.2.1 we know that we have three cases to check:

1. Let $A \xleftarrow{c,F_c} F(B_1, \ldots, B_n) \in R$ be a fuzzy clause and σ a valuation.

   From Proposition 2.2.1, the premise that for each $I \in \mathcal{I}$ we have that $I \Vdash P$ and the definition of $\bigsqcap$ it is clear that for all ground atoms C in $\mathrm{TB}_{\Pi,\Sigma,V}$ we have $J(C) = \bigsqcap_{I \in \mathcal{I}} I(C) \succ \bot$. As $\sigma(B_i)$ is a ground atom in $\mathrm{TB}_{\Pi,\Sigma,V}$, $J(\sigma(B_i))_{\,i \in 1\ldots n} \succ \bot$.

   As $J(\sigma(B_i))_{\,i \in 1\ldots n} \succ \bot$, for J to be model of P it has to be true that

   $$J(\sigma(A)) \succcurlyeq z'_J v'_J,$$

   where $z'_J = z_1 \circ \cdots \circ z_n$, $v'_J = \hat{F}_c(c, \hat{F}(v_1, \ldots, v_n))$ and $z_i v_i = J(\sigma(B_i)) = \bigsqcap_{I \in \mathcal{I}} I(\sigma(B_i))$.

   As each $I \in \mathcal{I}$ makes true $I \Vdash P$ we have

   $$I(\sigma(A)) \succcurlyeq z'_I v'_I,$$

   where $z'_I = z_1 \circ \cdots \circ z_n$, $v'_I = \hat{F}_c(c, \hat{F}(v_1, \ldots, v_n)))$ and $z_i v_i = I(\sigma(B_i))$[5].

   It is easy to see from $J(\sigma(B_i)) \preccurlyeq I(\sigma(B_i))$ and the fact that F_c, F and "$\circ$" are non-decreasing functions that $z'_J v'_J \preccurlyeq z'_I v'_I$, so

   $$J(\sigma(A)) \succcurlyeq z'_I v'_I \succcurlyeq z'_J v'_J$$

[5]Note that we use two different definitions for $z_i v_i$ in the same proof. There is no clash as one of them is used for calculating $z'_J v'_J$ and the other one for calculating $z'_I v'_I$.

$z_i v_i = \bigsqcap_{I \in \mathcal{I}} I(\sigma(B_i))$ makes $\bot \preccurlyeq z_i v_i \preccurlyeq I(\sigma(A))$ for every $I \in \mathcal{I}$ since $\bot \preccurlyeq J(\sigma(B_i)) \preccurlyeq I(\sigma(B_i))$ for every $I \in \mathcal{I}$ and from the fact that F_c, F and "$\circ$" are non-decreasing functions we can assure that $I(A) \succcurlyeq J(A) \succcurlyeq z'v'$, so $J \Vdash P$.

2. Let $A \longleftarrow v \in R$ be a fuzzy clause and σ a valuation.

   From the premise that for each $I \in \mathcal{I}$ we have that $I \Vdash P$ we know that

   $$I(A)_{I \in \mathcal{I}} \succcurlyeq \blacktriangledown v$$

   and, from the definition of $\bigsqcap$, it is clear that

   $$J(A) = \bigsqcap_{I \in \mathcal{I}} I(A) \succcurlyeq \blacktriangledown v$$

   so $J \Vdash P$.

3. Let $\texttt{default}(p(X_1, \ldots, X_n)) = [\delta_1 \text{ if } \varphi_1, \ldots, \delta_m \text{ if } \varphi_m] \in D$ be a default declaration and σ a valuation such that for a ground atom A in $\mathrm{TB}_{\Pi,\Sigma,V}$ we have $A = \sigma(p(X_1, \ldots, X_n))$.

   From the premise that for each $I \in \mathcal{I}$ we have that $I \Vdash P$ we know that for exactly one φ_i it must be that $\sigma(\varphi_i)$ holds and this implies that

   $$I(\sigma(p(X_1, \ldots, X_n)))_{I \in \mathcal{I}} \succcurlyeq \blacktriangle \delta_i$$

   and, from the definition of $\bigsqcap$, it is clear that

   $$J(A) \succcurlyeq \bigsqcap_{I \in \mathcal{I}} I(A) \succcurlyeq \blacktriangle \delta_i$$

   so $J \Vdash P$.

□

Definition 2.2.2. *Let P be a well-defined fuzzy logic program. The* least model *of P is defined as* $\mathsf{lm}(P) := \bigsqcap_{I \Vdash P} I$.

2.2.2 Least Fixpoint Semantics

Definition 2.2.3 (T_P operator)**.** *Let $P = (R, D, T)$ be a well-defined fuzzy logic program, I an interpretation, and recall that* $\mathsf{ground}(R)$ *denotes the set of all ground instances of clauses in R. The operator T_P is defined as follows:*

$$T_P(I) \quad := \quad T_R(I) \quad \sqcup \quad T_D(I)$$

where

$T_D(I) :=$

$$\left\{ A \mapsto \blacktriangle\delta_j \;\middle|\; \begin{array}{c} \texttt{default}(p(X_1,\ldots,X_n)) \;=\; [\delta_1 \textit{ if } \varphi_1,\ \ldots,\ \delta_m \textit{ if } \varphi_m] \in D, \\ A = \sigma(p(t_1,\ldots,t_n)) \;\; \textit{and} \\ \textit{there exists a } \; 1 \leq j \leq m \textit{ such that } \sigma(\varphi_j) \;\; \textit{holds.} \end{array} \right\}$$

$T_R(I) := (\bigsqcup_{r\in R} \{A \mapsto zv \mid \textit{condition}\})$

where $\{A \mapsto zv \mid \textit{condition}\}$ *can be*

$$\left\{ A \mapsto zv \;\middle|\; \begin{array}{c} A \xleftarrow{c,F_c} \; F(B_1,\ldots,B_n) \;\in \mathsf{ground}(R), \\ I(B_i) = z_i v_i \succ \bot \textit{ for all } i, \\ z = z_1 \circ \cdots \circ z_n \;\; \textit{and} \;\; v = \hat{F}_c(c, \hat{F}(v_1,\ldots,v_n)) \end{array} \right\}$$

or

$$\left\{ A \mapsto zv \;\middle|\; \begin{array}{c} A \longleftarrow v \in \mathsf{ground}(R) \;\; \textit{and} \\ z = \blacktriangledown \end{array} \right\}$$

Note that if for a ground atom A both a program clause with body atoms from I's domain and a default value declaration exist, the truth value that comes from the clause is preferred, since $T_R(I) \succ T_D(I)$[6].

As it is usual in the logic programming framework, the semantics of a program is characterised by the pre-fixpoints of T_P.

Theorem 2.2.1. *Let P be a well-defined fuzzy logic program and I an interpretation.*

$$I \Vdash P \quad \textit{iff} \quad T_P(I) \;\sqsubseteq\; I.$$

[6] $T_R(I) \succ T_D(I)$ comes from the fact that $T_R(I) \succ \blacktriangle 1$, $T_D(I) \prec \blacklozenge 0$ and $\blacktriangledown \succ \blacklozenge \succ \blacktriangle$.

Proof. "if": Let $T_P(I) \sqsubseteq I$ and $A' \in \mathbb{HB}$ be an arbitrary ground atom. For $I \Vdash P$ we need $I \Vdash R$ and $I \Vdash D$, so we have to check both cases.

Before starting, note that for all ground atom $C \in \mathbb{HB}$ we have $T_P(I)(C) \succ \bot$ since $T_P(I)(C) = T_R(I)(C) \sqcup T_D(I)(C)$ and for a well-defined fuzzy logic program $T_D(I)(C) \succ \bot$. As $T_P(I)(C) \sqsubseteq I(C)$, $I(C) \succcurlyeq T_P(I)(C) \succ \bot$.

$I \Vdash R$: We need $I \Vdash r$ for every $r \in R$, and r can be of the form $A \xleftarrow{c,F_c} F(B_1, \ldots, B_n) \in R$ or $A \longleftarrow v \in R$.

Let $A \xleftarrow{c,F_c} F(B_1, \ldots, B_n) \in R$ be a fuzzy clause and σ a valuation such that $A' = \sigma(A)$. The ground clause used to evaluate $T_P(I)(A')$ will be $\sigma(\, A \xleftarrow{c,F_c} F(B_1, \ldots, B_n)\,) \in \mathsf{ground}(R)$.
As $I(C) \succ \bot$ for all $C \in \mathbb{HB}$, $I(\sigma(B_i)) = z_i v_i \succ \bot$ for all i.
So, for $I \Vdash R$ we need $I(A') \succcurlyeq z'v'$, where $z' = z_1 \circ \cdots \circ z_n$, $v' = \hat{F}_c(c, \hat{F}(v_1, \ldots, v_n))$ and $z' \in \{ \blacktriangledown, \blacklozenge \}$.
By definition of T_P we have $T_P(I)(A') = T_R(I)(A') = z'v'$, where $z' = z_1 \circ \cdots \circ z_n$, $v' = \hat{F}_c(c, \hat{F}(v_1, \ldots, v_n))$ and $z' \in \{ \blacktriangledown, \blacklozenge \}$.
From the premise $T_P(I) \sqsubseteq I$ we get $I(A') \succcurlyeq z'v'$, so $I \Vdash R$.

Let $A \longleftarrow v \in R$ be a fuzzy clause and σ a valuation such that $A' = \sigma(A)$. The ground clause used to evaluate $T_P(I)(A')$ will be $\sigma(\, A \longleftarrow v\,) \in \mathsf{ground}(R)$.
So, for $I \Vdash R$ we need $I(A') \succcurlyeq z'v'$, where $z' = \blacktriangledown$ and $v' = v$.
By definition of T_P we have $T_P(I)(A') = \blacktriangledown v \succ \bot$.
From the premise $T_P(I) \sqsubseteq I$ we get $I(A') \succcurlyeq \blacktriangledown v$.

$I \Vdash D$: Let $\mathtt{default}(p(X_1, \ldots, X_n)) = [\delta_1 \text{ if } \varphi_1, \ldots, \delta_m \text{ if } \varphi_m] \in D$, σ be a valuation, and let $A' := \sigma(p(X_1, \ldots, X_n))$ be well-typed. Since P is well-defined, there exists a j such that $\sigma(\varphi_j)$ holds and thus $T_P(I)(A') = \blacktriangle\delta_j$. From the premise $T_P(I) \sqsubseteq I$, we again get $I(A') \succcurlyeq \blacktriangle\delta_j$.

"only if": Let $I \Vdash P$ and $A' \in \mathbb{HB}$ be an arbitrary ground atom. From $I \Vdash P$ we have that $I \Vdash R$ and $I \Vdash D$.

from $I \Vdash D$: for A' we know that there are a default declaration $\mathtt{default}(p(X_1, \ldots, X_n)) = [\delta_1 \text{ if } \varphi_1, \ldots, \delta_m \text{ if } \varphi_m] \in D$, a valuation σ such that $A' := \sigma(p(X_1, \ldots, X_n))$ is well-typed and,

since P is well-defined, a j such that $\sigma(\varphi_j)$ holds and makes $I(A') \succcurlyeq \blacktriangle\delta_j$.

From definition of T_D in T_P it is easy to see that $T_D = \blacktriangle\delta_j$, but for T_P we can only say (for now, we continue below) that $T_P(I)(A') = T_R(I)(A') \sqcup T_D(I)(A') \succcurlyeq \blacktriangle\delta_j$.

The important fact that is derived from $I \Vdash D$ is that for every A' we have $I(A') \succcurlyeq \blacktriangle\delta_j \succcurlyeq \bot$.

from $I \Vdash R$: for A' we know that if there is a rule $r \in R$ of the form $A \xleftarrow{c,F_c} F(B_1,\ldots,B_n) \in R$ or $A \longleftarrow v \in R$ and there exists a σ such that $A' = \sigma(A)$ then

if r is of the form $A \xleftarrow{c,F_c} F(B_1,\ldots,B_n) \in R$: From $I \Vdash D$ we know that for every $\sigma(B_i)$ we have $I(\sigma(B_i)) \succcurlyeq \bot$, so $I(A') \succcurlyeq z'v'$, where $z' = z_1 \circ \cdots \circ z_n$, $z' \in \{ \blacktriangledown, \blacklozenge \}$ and $v' = \hat{F}_c(c, \hat{F}(v_1, \ldots, v_n))$.

From definition of T_R it is easy to see that if there is such a rule, there will be a ground version of that rule $\sigma(A \xleftarrow{c,F_c} F(B_1,\ldots,B_n)) \in \mathsf{ground}(R)$ such that we can assure that $T_R(I)(A') \succcurlyeq z'v'$.

if r is of the form $A \longleftarrow v \in R$: $I(A') \succcurlyeq z'v'$, where $z' = \blacktriangledown$ and $v' = v$.

From definition of T_R it is easy to see that if there is such a rule, there will be a ground version of that rule $\sigma(A \longleftarrow v) \in \mathsf{ground}(R)$ such that we can assure that $T_R(I)(A') \succcurlyeq z'v'$.

Note that we have only proved that for $d \in D$ a default declaration $T_D(I)(A') \succcurlyeq \blacktriangle\delta_j$ and that if there is a rule $r \in R$ then $T_R(I)(A') \succcurlyeq z'v'$. This is caused by the fact that in model definition we have to check that for every rule $r \in R$ it holds that $I(A') \succcurlyeq z'v'$, while in T_P definition we collect each one of the values of each rule and evaluate the maximum value between them.

So, if there is no rule $r \in R$ of the form $A \xleftarrow{c,F_c} F(B_1,\ldots,B_n) \in R$ or $A \longleftarrow v \in R$ such that for some valuation σ we have $A' = \sigma(A)$ then $T_P(I)(A') = T_R(I)(A') \sqcup T_D(I)(A') = \bot \sqcup \blacktriangle\delta_j = \blacktriangle\delta_j$, and this results in $T_P(I)(A') = \blacktriangle\delta_j \preccurlyeq I(A')$, so $T_P(I)(A') \sqsubseteq I(A')$.

But if there are k clauses $r \in R$ of the form $A \xleftarrow{c,F_c} F(B_1,\ldots,B_n) \in R$ or $A \longleftarrow v \in R$ such that for some valuations $\sigma_1 \ldots \sigma_k$ we have $A' = \sigma_l(A)$ for $1 \leq l \leq k$,

then from $I \Vdash P$ we have $I(A') \succcurlyeq max_{\,l \in [\,1\,..\,k\,]}\,(\,z'_l\,v'_l\,)$, where $z'_l = z_1 \circ \cdots \circ z_n$, $z'_l \in \{\,\blacktriangledown, \blacklozenge\,\}$, $v'_l = \hat{F}_c(c, \hat{F}(v_1, \ldots, v_n))$ and $I(\,B_i\,) = z_i v_i$.

Instead of this procedure of finding the maximum value in model definition, in T_P this is achieved by using the join operator $\sqcup$:

$$\begin{aligned} T_P(I)(A') &= T_R(I)(A') \sqcup T_D(I)(A') = \\ &\quad (\bigsqcup_{l\,=\,1\ldots k} z'_l\,v'_l\,) \sqcup \blacktriangle\delta_j = \\ &\quad \bigsqcup_{l\,=\,1\ldots k} z'_l\,v'_l\,. \end{aligned}$$

As $\bigsqcup_{l\,=\,1\ldots k} z'v'_l \preccurlyeq I(A')$, we have $T_P(I)(A') \sqsubseteq I(A')$.

□

Thus, if I is a model of P, then for every A occurring in the program we have that $T_P(I)(A) \preccurlyeq I(A)$, which means that I is a pre-fixpoint of T_P.

Proposition 2.2.4 (T_P is monotonic)**.** *Let P be a well-defined fuzzy logic program and I_i and I_{i+1} two interpretations.*

$$\textit{if} \quad I_i \sqsubseteq I_{i+1} \quad \textit{then} \quad T_P(I_i) \sqsubseteq T_P(I_{i+1})$$

Proof. From definition of $I_i \sqsubseteq I_{i+1}$ we have that $I_i(A) \preccurlyeq I_{i+1}(A)$ for all $A \in \mathbb{HB}$, and from definition of T_P we have

$$T_P(I) \;:=\; T_R(I) \bigsqcup T_D(I).$$

Since the operator $\sqcup$ is monotonic and the value of $T_D(I)$ depends only on the default declarations in the program (it is shared by both interpretations I_i and I_{i+1}), $T_D(I_i) = T_D(I_{i+1})$. For the value of $T_R(I)$ it is rather different, since

$$T_R(I) := \left(\bigsqcup_{r \in R} \{A \mapsto zv \mid \text{condition}\} \right)$$

where $\{A \mapsto zv \mid \text{condition}\}$ can be

$$\left\{ A \mapsto zv \;\middle|\; \begin{array}{c} A \xleftarrow{c,F_c} F(B_1, \ldots, B_n) \in \mathsf{ground}(R), \\ I(B_i) = z_i v_i \succ \bot \text{ for all } i, \\ z = z_1 \circ \cdots \circ z_n \text{ and } v = \hat{F}_c(c, \hat{F}(v_1, \ldots, v_n)) \end{array} \right\}$$

or

$$\left\{ A \mapsto zv \;\middle|\; \begin{array}{c} A \longleftarrow v \in \mathsf{ground}(R) \text{ and} \\ z = \blacktriangledown \end{array} \right\}$$

For those clauses in the program with the form $A \longleftarrow v \in R$ the mapping $A \mapsto zv$ remains untouched, but for clauses with the form $A \xleftarrow{c,F_c} F(B_1,\ldots,B_n) \in R$ the mapping depends on the values $I_i(B_j)$ and $I_{i+1}(B_j)$, $j \in [1 \,..\, n]$.

As $\sqcup$ is monotone, for those clauses we need to justify that $I_i(A) \preccurlyeq I_{i+1}(A)$ for all $A \in \mathbb{HB}$ implies that the mappings obtained for I_i, $A \mapsto (zv)_{I_i}$, and I_{i+1}, $A \mapsto (zv)_{I_{i+1}}$, fulfill $(zv)_{I_i} \preccurlyeq (zv)_{I_{i+1}}$.

We use for that the knowledge about the truth functions for the connectives F and F_C. We know that both are monotone and non-decreasing, so from the premise $I_i(B_j) \preccurlyeq I_{i+1}(B_j)$ for all $B_j \in \mathbb{HB}$ we can deduce $(zv)_{I_i} \preccurlyeq (zv)_{I_{i+1}}$. □

Due to the monotonicity of the immediate consequences operator, the semantics of P is given by its least model which, as shown by the Knaster-Tarski fixpoint theorem [Kna28; Tar55], is exactly the least fixpoint of T_P.[7]

We now prove that our T_P is continuous, so Kleene's fixpoint theorem [Kle52] can be used to prove that the least fixed point can be reached in ω steps.

Proposition 2.2.5 (T_P is continuous)**.** *Let P be a well-defined fuzzy logic program and $I_0 \sqsubseteq I_1 \sqsubseteq \ldots$ a countably infinite increasing sequence of interpretations. Then*

$$T_P\left(\bigsqcup_{n=0}^{\infty} I_n\right) = \bigsqcup_{n=0}^{\infty} T_P(I_n).$$

Proof. We use the following facts:

1. Since $I_0 \sqsubseteq I_1 \sqsubseteq \ldots$ and from definition of $\sqsubseteq$ we have that $I_i(A) \preccurlyeq I_{i+1}(A)$ for every ground term $A \in \mathbb{HB}$. As $\bigsqcup$ takes by definition the maximum interpretation, $\bigsqcup_{i=0}^{n} I_i = I_n$.

2. We have that $T_P(I_0) \sqsubseteq T_P(I_1) \sqsubseteq \ldots$ since $I_0 \sqsubseteq I_1 \sqsubseteq \ldots$ and T_P is monotonic (Proposition 2.2.4). Again by using definitions of $\sqsubseteq$ and $\bigsqcup$ we obtain $\bigsqcup_{i=0}^{n} T_P(I_i) = T_P(I_n)$.

[7] As usually, the least fixpoint of T_P can be obtained by transfinitely iterating T_P from the least interpretation $\bot$.

$$T_P\left(\bigsqcup_{n=0}^{\infty} I_n\right) \quad =^{1} \quad T_P\left(I_\infty\right) \quad =^{2} \quad \bigsqcup_{n=0}^{\infty} T_P(I_n)$$

□

We now recall the definition of ordinal powers of an operator. Let T be an operator and α be an ordinal number. The *ordinal power* α *of* T is defined as follows:

$$T{\uparrow}\alpha := \begin{cases} T(T{\uparrow}\alpha - 1) & \text{if } \alpha \text{ is a successor ordinal} \\ \bigsqcup_{\alpha' < \alpha} T{\uparrow}\alpha' & \text{if } \alpha \text{ is a limit ordinal} \end{cases}$$

Theorem 2.2.2. *Let P be a well-defined fuzzy logic program. Then the least fixpoint of T_P exists and is equal to $T_P{\uparrow}\omega$.*

Proof. The existence of the least fixpoint of T_P follows from the facts that $(\mathcal{I}_P, \sqsubseteq)$ forms a complete lattice, T_P is monotone (Proposition 2.2.4), and the Knaster-Tarski fixpoint theorem [Kna28; Tar55]. Its equality to $T_P \uparrow \omega$ follows from the facts that $(\mathcal{I}_P, \sqsubseteq)$ forms a complete lattice, T_P is continuous (Proposition 2.2.5), and the Kleene fixpoint theorem [Kle52]. □

Since the least fixpoint always exists, we can define a semantics based on it.

Definition 2.2.4. *Let P be a well-defined fuzzy logic program. Then the* least fixpoint semantics of P *is defined as* $\mathsf{lfp}(P) = T_P{\uparrow}\omega(\bot)$. *Here, $\bot$ denotes the interpretation mapping everything to $\bot$ (thus being the least element of the lattice $(\mathcal{I}_P, \sqsubseteq)$).*

Theorem 2.2.3. *For a well-defined fuzzy logic program P, we have*

$$\mathsf{lm}(\ P\ ) \quad = \quad \mathsf{lfp}(\ P\ ).$$

Proof.

$$\begin{aligned} \mathsf{lm}(P) &= \bigsqcap_{I \ \Vdash\ P} I && \text{(by definition)} \\ &= \bigsqcap_{T_P(I) \sqsubseteq I} I && \text{(by Lemma 2.2.1)} \\ &= T_P{\uparrow}\omega(\bot) && \text{(by the Kleene fixpoint theorem)} \\ &= \mathsf{lfp}(P) && \text{(by definition)} \end{aligned}$$

□

2.3 Operational Semantics

The operational semantics will be formalised 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; they will be encoded into the computations.

Definition 2.3.1. *Let Ω be a signature of connectives and W a set of variables with $W \cap V = \varnothing$.*[8]

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 the set of variables 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 (final) if all its nodes are ground (final).*

For a given computation tree t we define the tree attribute

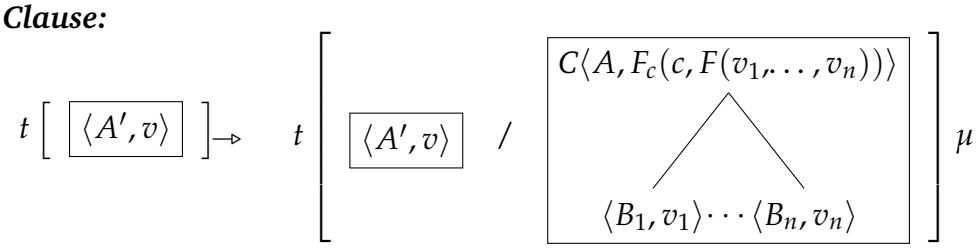
$$z_t = \begin{cases} \blacktriangledown & \text{if } t \text{ contains no D-node} \\ \blacklozenge & \text{if } t \text{ contains both C- and D-nodes} \\ \blacktriangle & \text{if } t \text{ contains only D-nodes} \end{cases}$$

Computation nodes are essentially generalisations of queries that keep track of connective 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 2.3.2 (Transition relation)**.** *For a given fuzzy logic program $P = (R, D, T)$, the transition relation $\rightarrowtriangle$ is characterised by the transition rules below. In these rules 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".*

[8] Note that V is the set of variables used when building the term base $\mathrm{TB}_{\Pi,\Sigma,V}$ of our program.

- ***Clause:***



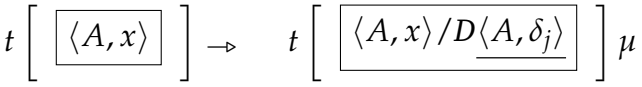
If there is a (variable disjoint instance of a) program clause

$$A \xleftarrow{c,F_c} F(B_1, \ldots, B_n) \in R \text{ and } \mu = \mathsf{mgu}(A', A).$$

(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.)

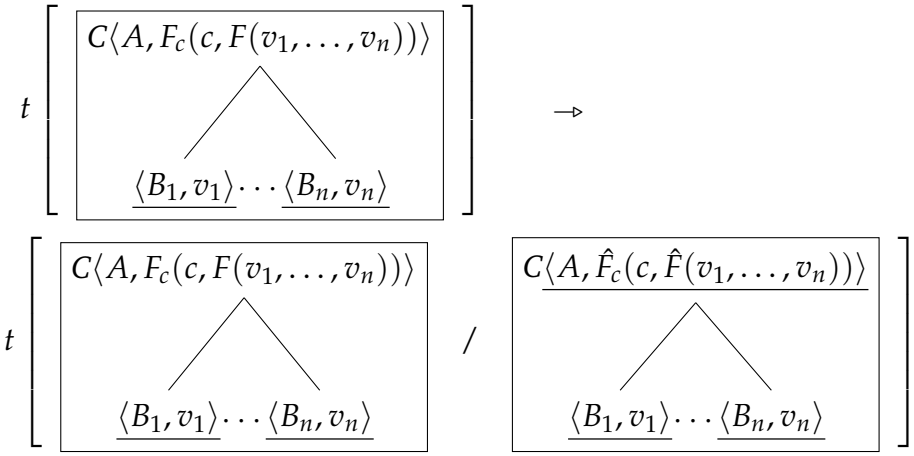
Note that we immediately finalise a node when applying this rule for a fuzzy fact.

- ***Default:***



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

- ***Finalise:***



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

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[9]. We illustrate this with an example computation.

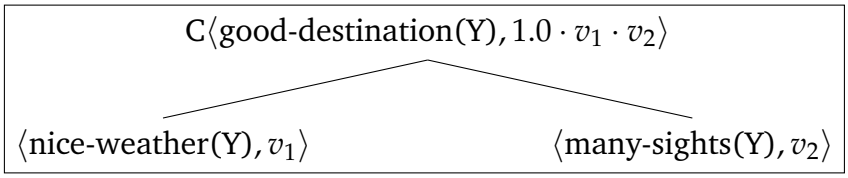
Example (continued). We start with the tree

$$\boxed{\langle \text{good-destination(Y)}, v\rangle}.$$

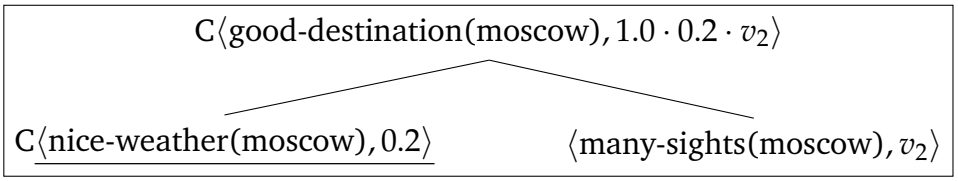
Applying the **Clause**-transition to the initial tree with the program clause

$$\text{good-destination(X)} \overset{1.0,\cdot}{\longleftarrow} \cdot(\text{nice-weather(X)}, \text{many-sights(X)})$$

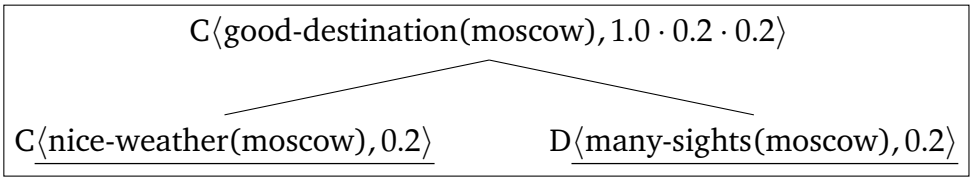
yields



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

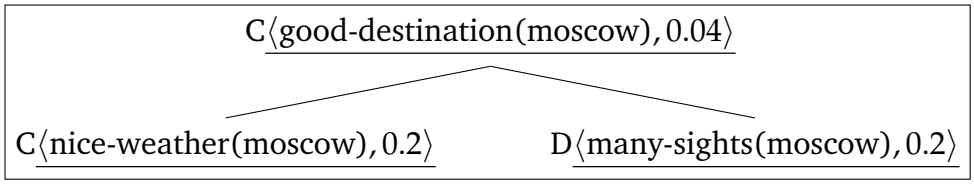


Since there exists no clause whose head matches many-sights(moscow), we apply the **Default**-rule for many-sights to the right child.



In the last step, we finalise the root node.

[9] Note that infinite computations lead to no answer as in the operational semantics of SLD resolution.



The calculated truth value for good-destination(moscow) is thus 0.04. ◇

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 generalisation of the success set of a program.

Definition 2.3.3. *Let P be a well-defined fuzzy logic program. The canonical model of P for $A \in \mathbb{HB}$ is defined as follows:*

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

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

2.4 About multi-adjoint logic programming

Generalised logic programs' semantics require a different notion of consequence (implication) which satisfies a generalised modus ponens rule. It is described as natural in [Voj01] to look for a semantic basis as a common denominator of the residuated lattices and fairly general conjunctors and their adjoints. Different implications and several modus ponens-like inference rules are used. So, the principal point of this extension naturally leads to considering several adjoint pairs in the residuated lattice, leading what [Voj01] calls multi-adjoint algebra and [MOAV04] calls multi-adjoint (semi-)lattice. During the last years there have been many theoretical publications about multi-adjoint framework semantics [MOAV01c; MOAV02; MOAV04; Voj01].

RFuzzy is able to model multi-adjoint logic; but, as its algebraic structure seems to be difficult to understand for a not-so-theoretical reader (potential RFuzzy programmer), we have not used multi-adjoint definitions in Sections 2.2 and 2.3 when describing RFuzzy semantics for the sake of simplicity. In this section we explain in an intuitive way the relation of RFuzzy with the multi-adjoint framework.

Multi-adjoint logic is a theoretical framework developed to give support to the use of a number of different implications in program rules. The underlying

structure used to capture such information is a multi-adjoint algebra, which is straightforward from the definitions of adjoint pair and multi-adjoint lattice. The definitions of these three concepts are taken from [MOAV01c].

Definition 2.4.1 (Adjoint Pair, from [MOAV01c]. Firstly introduced in a logical context by Pavelka [Pav79]). *Let $<P, \preceq>$ be a partially ordered set and $(\leftarrow, \&)$ a pair of binary operations in P such that:*

(i) Operation & is increasing in both arguments, i.e. if $x_1, x_2, y \in P$ such that $x_1 \preceq x_2$ then $(x_1 \& y) \preceq (x_2 \& y)$ and $(y \& x_1) \preceq (y \& x_2)$;

(ii) Operation $\leftarrow$ is increasing in the first argument (the consequent) and decreasing in the second argument (the antecedent), i.e. if $x_1, x_2, y \in P$ such that $x_1 \preceq x_2$ then $(x_1 \leftarrow y) \preceq (x_2 \leftarrow y)$ and $(y \leftarrow x_2) \preceq (y \leftarrow x_1)$;

(iii) For any $x, y, z \in P$, we have that $x \preceq (y \leftarrow z)$ holds if and only if $(x \& z) \preceq y$ holds.

Then $(\leftarrow, \&)$ is called an adjoint pair in $<$P, $\preceq>$.

Definition 2.4.2 (Multi-Adjoint Lattice, from [MOAV01c]). *Let $< L, \preceq>$ be a lattice. A multi-adjoint lattice L is a tuple*

$$(L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n) \tag{2.4.1}$$

satisfying the following items:

(i) $< L, \preceq>$ is bounded, i.e. it has bottom ($\bot$) and top ($\top$) elements;

(ii) $(\leftarrow_i, \&_i)$ is an adjoint pair in $< L, \preceq>$ for $i = 1, \ldots, n$;

(iii) $\top \&_i v = v \&_i \top = v$ for all $v \in L$ for $i = 1, \ldots, n$.

Definition 2.4.3 (Multi-Adjoint Ω-Algebra, from [MOAV01c]). *Let Ω be a graded set containing operators $\leftarrow_i$ and $\&_i$ for $i = 1, \ldots, n$ and possibly some extra operators, and let $\mathcal{L} = (L, I)$ be an Ω-algebra whose carrier set L is a lattice under $\preceq$.*

We say that $\mathcal{L}$ is a multi-adjoint Ω-algebra with respect to the pairs $(\leftarrow_i, \&_i)$ for $i = 1, \ldots, n$ if

$$\mathcal{L} = (L, \preceq, I(\leftarrow_1), I(\&_1), \ldots, I(\leftarrow_n), I(\&_n)) \tag{2.4.2}$$

is a multi-adjoint lattice.

There are similar definitions in [MOAV01b; MOAV01c; MOAV02]. They are used to build the framework of multi-adjoint logic programming in [JMP05; MM08a], but the relevant fact is the relation between adjoint pairs $(\leftarrow_i, \&_i)$, which makes it possible to evaluate in the many-valued modus ponens the truth value of the head of a rule from the truth value of the body and the weight of the rule:

$$\frac{(B,\ z),\ (B \rightarrow A,\ x)}{(A,\ y)},$$

$$x \preceq (y \leftarrow z) \quad \text{iff} \quad (x \& z) \preceq y$$

So, this can be used to define from *satisfaction the immediate consequences operator*, as illustrated below. Note that $\hat{I}(B) = z$, $\hat{I}(B \rightarrow A) = x$, $\hat{I}(A) = y$, so

$$v \preceq \hat{I}(A \leftarrow_i B) \quad \text{iff} \quad v \,\&_i\, \hat{I}(B) \preceq \hat{I}(A) \tag{2.4.3}$$

Definition 2.4.4 (Satisfaction, from [MOAV01c])**.** *Given an interpretation $I \in \mathcal{I}_{\mathcal{L}}$, where $\mathcal{I}_{\mathcal{L}}$ is the set of all interpretations of the formulas defined by the Ω-algebra $\mathcal{F}$ in the Ω-algebra $\mathcal{L}$, a weighted rule $< A \leftarrow_i B, v >$ is satisfied by I iff $v \preceq \hat{I}(A \leftarrow_i B)$.*

Definition 2.4.5 (Immediate consequences operator $\mathcal{T}_{\mathcal{P}}^{\mathcal{L}}$, from [MOAV01c])**.** *Let $\mathcal{P}$ be a multi-adjoint logic program. The immediate consequences operator $\mathcal{T}_{\mathcal{P}}^{\mathcal{L}} : \mathcal{I}_{\mathcal{L}} \rightarrow \mathcal{I}_{\mathcal{L}}$, mapping interpretations to interpretations, is defined by considering*

$$\mathcal{T}_{\mathcal{P}}^{\mathcal{L}}(I)(A) = sup\ \{\ v\ \hat{\&}_i\ \hat{I}(B) \quad | \quad A \leftarrow_i^v B \in \mathcal{P}\ \} \tag{2.4.4}$$

We provide an example from FLOPER [MM08b] (Fuzzy LOgic Programming Environment for Research) to illustrate an execution in which this relation between the adjoint pairs is used to calculate the truth value of the head of the rules.

```
R1 : p(X) ←P q(X,Y) &G r(Y) with 0.8
R2 : q(a,Y) ←P s(Y) with 0.7
R3 : q(b,Y) ←L r(Y) with 0.8
R4 : r(Y) ← with 0.7
R5 : s(b) ← with 0.9
```

Listing 2.4.1: Example taken from [MM08b]

The labels P, G and L in the program mean Product logic, Gödel intuitionistic Logic and Łukasiewicz logic, respectively. The goal for the previous program is $\leftarrow p(X)\&_G r(a)$. Each underlined expression in the execution below is the one selected in each admissible step (Definition 2.1 in [MM08b])

```
<p(X)&Gr(a); id>
 →AS1R1 <(0.8 &P (q(X1,Y1) &G r(Y1))) &G r(a); σ1>
 →AS1R2 <(0.8 &P ((0.7 &P s(Y2)) &G r(Y1))) &G r(a); σ2>
 →AS1R2 <(0.8 &P ((0.7 &P 0.9) &G r(b))) &G r(a); σ3>
 →AS1R2 <(0.8 &P ((0.7 &P 0.9) &G 0.7)) &G r(a); σ4>
 →AS1R2 <(0.8 &P ((0.7 &P 0.9) &G 0.7)) &G 0.7; σ5>
```

Listing 2.4.2: Execution process, taken from [MM08b]

where

$$\sigma_1 = \{X/X_1\}, \tag{2.4.5}$$
$$\sigma_2 = \{X/a, X_1/a, Y_1/Y_2\}, \tag{2.4.6}$$
$$\sigma_3 = \{X/a, X_1/a, Y_1/b, Y_2/b\}, \tag{2.4.7}$$
$$\sigma_4 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b\}, \tag{2.4.8}$$
$$\sigma_5 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b, Y_4/a\} \tag{2.4.9}$$

and, as the only variable appearing in the query is X, the only substitution associated with the result is $\{X/a\}$. So, the admissible computed answer (Definition 2.2 in [MM08b]) is

$$< \ (0.8 \ \&_P \ ((0.7 \ \&_P \ 0.9) \ \&_G \ 0.7)) \ \&_G \ 0.7 \ ; \ \{X/a\} \ >, \tag{2.4.10}$$

that can be simplified, after evaluating the arithmetic expression, to

$$< 0.504 \ ; \ \{X/a\} \ > . \tag{2.4.11}$$

Being a rule of the form "$A \ \leftarrow_i \ B \ \ with \ \ v$", where A is the head, B the body, v the weight of the rule and " i " is the used logic (e.g. P, G, L), the interesting point is that to calculate the truth value of the head of a rule we use the relation between the adjoint pairs previously exposed,

$$v \preceq \hat{I}(A \leftarrow_i B) \quad \text{iff} \quad v\&_i\hat{I}(B) \preceq \hat{I}(A)$$

Notice that the elements of the adjoint pair $(\leftarrow_i, \ \&_i)$ have different uses in an evaluation process. While $\leftarrow_i$ is used to obtain the weight of a rule from the

truth values of the body and the head of the rule (e.g. for inducing fuzzy rules weights from data in a data mining process), $\&_i$ is used to obtain the truth value of the head of a rule from the truth values of the elements of the body and the weight of the rule.

This second deductive process is the one that is modelled by RFuzzy. To this end, we let RFuzzy programmers code different $\&_i$ while $\leftarrow_i$ is not explicit in our syntax. In our rule syntax, that is used in definitions 2.2.1 and 2.2.3,

$$A \xleftarrow{c,F_c} F(B_1,\ldots,B_n) \ \in \mathsf{ground}(R)$$

it is always deduced that v, the resultant truth value for the clause head, is

$$v = \hat{F}_c(c, \hat{F}(v_1,\ldots,v_n))$$

where instead of " $\leftarrow_i$ ... *with* c " we use " $\xleftarrow{c,F_C}$ " and F_C is $\&_i$ if $(\leftarrow_i, \&_i)$ is an adjoint pair.

The previous example from FLOPER [MM08b] translated to the RFuzzy syntax used in the semantics sections is below.

R_1 : p(X) $\xleftarrow{0.8,P}$ q(X,Y) $\&_G$ r(Y)
R_2 : q(a,Y) $\xleftarrow{0.7,P}$ s(Y)
R_3 : q(b,Y) $\xleftarrow{0.8,L}$ r(Y)
R_4 : r(Y) $\longleftarrow$ 0.7
R_5 : s(b) $\longleftarrow$ 0.9

Let us, finally, explain the adequateness of RFuzzy default rules to the multi-adjoint framework. We are going to provide a clarifying example where we define a type *city* and a predicate *nice-weather/1* with one argument of type *city*.

Example 2.

```
madrid : City
sydney : City
moscow : City

nice-weather : ( City )
```

A program with a default truth value 0.5 for cities whose information about nice weather is not explicit

```
default(nice-weather(X)) = 0.5

nice-weather(madrid) ← 0.8
nice-weather(moscow) ← 0.2
```

is semantically equivalent to the following plain program

```
nice-weather(madrid) ← 0.8
nice-weather(moscow) ← 0.2
nice-weather(sydney) ← 0.5
```

◇

This works in the general case: default value declarations in RFuzzy programs are indeed just syntactic sugar and can be compiled away. Hence, also general RFuzzy programs can be considered to model multi-adjoint semantics.

2.5 Using the Framework. Implementation Details

Obviously, when coding in Prolog it is not possible to write the symbols we have used in Section 2.1. Here we expose the syntax used to write programs and some details of the implementation of the framework.

2.5.1 The programs syntax

Types are coded according to the following syntax:

:- set_prop *pred*/*ar* **=>** *type_pred*_1/1 [, *type_pred_n*/1]* . (2.5.1)

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. This definition of types constrains the values of the n-th argument of *pred* to the values accepted by the predicate *type_pred_n*. This definition of types ensures that the values assigned to the arguments of *pred* are correctly typed.

The example below requires 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.

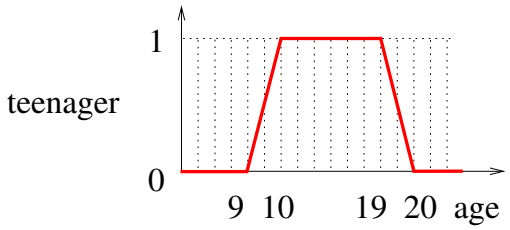


Figure 2.5.1: Teenager truth value continuous representation

```
← set_prop has_lower_price/2 ⇒ car/1, car/1.
← set_prop expensive_car/1 ⇒ car/1.
car(vw_caddy).
car(alfa_romeo_gt).
car(aston_martin_bulldog).
car(lamborghini_urraco).
```

The syntax for fuzzy facts is

$$pred(args) \textbf{ value } truth\_val. \tag{2.5.2}$$

where 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.
```

Fuzzy facts are worth for a finite (and relatively small) number of individuals. Nevertheless, it is very common to represent fuzzy truth using continuous functions. Figure 2.5.1 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.

RFuzzy provides the syntax for defining functions by stretches. This syntax is shown in (2.5.3). 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.

$$pred \texttt{ :\# } ([(val1, truth\_val1),\ (val2, truth\_val2)\ [,\ (valn, truth\_valn)\ ]^*]) \ . \tag{2.5.3}$$

The RFuzzy syntax for the predicate *teenager/1* (represented in Figure2.5.1) is:

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

Fuzzy clauses or rules have a simple syntax, defined in (2.5.5). There are two connectives, *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 a connective from the list of the available ones[10] or define his/her own connective.

$$pred(arg1\ [,\ argn]^*\ )\ [\ \textbf{cred (}op1, value1\textbf{)}\ ]\ \texttt{:}\sim\ op2\ pred1(args\_pred\_1)\ [,\ predm(args\_pred\_m)]\ . \tag{2.5.4}$$

The following example uses the operator *prod* for combining truth values of the subgoals of the body and *min* (Gödel logic is used here) to combine the result with the credibility of the rule (which is 0.8). "**cred (***op*1, *value*1**)**" can only appear 0 or 1 times.

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

Default truth values syntax is defined in (2.5.5) and (2.5.6),

:- default(*pred/ar, truth_value***) .** (2.5.5)

:- default(*pred/ar, truth_value***) => *membership_predicate/ar*.** (2.5.6)

where *pred/ar* is in both cases the predicate to which we are defining default values. As expected, when defining the three cases (explicit, conditional and 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.

[10] Connectives 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*.

The code from the example below added to the code from the previous examples 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* (conditional 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).
```

2.5.2 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*?

```
?- expensive_car(vw_caddy,V).
V = 0.5 ? ;
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
```

2.5.3 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 [The15b] package: 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($\mathbb{R}$) constraints by means of the RFuzzy package and

(2) the program with constraints is translated into ISO Prolog by using the CLP($\mathbb{R}$) package.

Figure 2.5.2 shows the sequential process of program transformation.

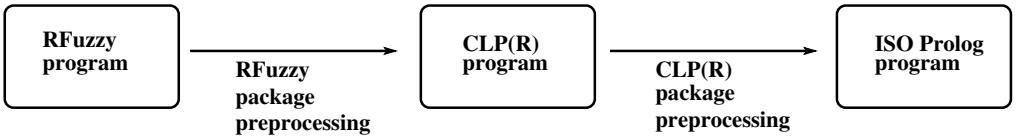


Figure 2.5.2: RFuzzy architecture.

Chapter 3

Management of priorities in the framework: RFuzzy v.2

The number of problems we can represent by using fuzzy logic is huge but there still some that can not be simulated by just using it, as the one we present and try to overcome here: the existence of rule priorities that overwrite the normal ordering of results obtained from fuzzy logic inferences. Just suppose we want to have default values for some rules. This can not be coded in the current multi-adjoint framework because all rules have the same priority there. We try here to overcome this limitation.

In the previous chapter we have presented three symbols, an order between them ($\blacktriangle <_a \blacklozenge <_a \blacktriangledown$) and an operator ($\circ$) to combine them for this purpose. For working with small programs this is perfectly adequate, but when dealing with larger programs the intentions get lost due to the assignation of identical priority weights to clauses that depend on a small amount of default information and clauses that depend on a large amount of default information.

Our goal in this chapter is to differentiate between them by using priorities so the inference process gets even closer to the human way of reasoning and solving problems.

There are many proposals on how to introduce priorities in logic programming (LP) [AP95; DST03; JGM07; LV90; MNR97; WZL00] but, as far as we know, there is no existing work on fuzzy logic programming, although it seems to be rather necessary its inclusion.

We start by the syntax used in this chapter (3.1) and go next for the semantics of our proposal (3.2).

3.1 Syntax

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 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* $\mathbb{HU}$ is the set of all ground terms, and the *Herbrand base* $\mathbb{HB}$ is the set of all atoms with arguments from the Herbrand universe. A substitution σ or ξ is (as usual) a mapping from variables from V to terms from $\mathrm{TU}_{\Sigma,V}$[1].

To capture different interdependencies between predicates, we will make use of a signature Ω of *many-valued connectives*[2] formed by *conjunctions* $\&_1, \&_2, ..., \&_k$, *disjunctions* $\vee_1, \vee_2, ..., \vee_l$, *implications* $\leftarrow_1, \leftarrow_2, ..., \leftarrow_m$, *aggregations* $@_1, @_2, ..., @_n$ and tuples of real numbers in the interval $[0,\ 1]$ represented by $(\mathsf{p},\ \mathsf{v})$.

While Ω denotes the set of connective symbols, $\hat{\Omega}$ denotes the set of their respective associated truth functions. Instances of connective symbols and truth functions are denoted by $\&_i$ and $\hat{\&}_i$ for conjunctors, $\vee_i$ and $\hat{\vee}_i$ for disjunctors, $\leftarrow_i$ and $\hat{\leftarrow}_i$ for implicators, $@_i$ and $\hat{@}_i$ for aggregators and $(\mathsf{p},\ \mathsf{v})$ and $(\hat{\mathsf{p}},\ \hat{\mathsf{v}})$ for the tuples.

Truth functions for the connectives are then defined as $\hat{\&} : [0,1]^2 \to [0,1]$ monotone[3]and non-decreasing in both coordinates, $\hat{\vee} : [0,1]^2 \to [0,1]$ monotone in both coordinates, $\hat{\leftarrow} : [0,1]^2 \to [0,1]$ non-increasing in the first and non-decreasing in the second coordinate, $\hat{@} : [0,1]^n \to [0,1]$ as a function that verifies $\hat{@}(0,\ \ldots,\ 0) = 0$ and $\hat{@}(1,\ \ldots,\ 1) = 1$ and $(\mathsf{p},\ \mathsf{v}) \in \Omega^{(0)}$ are functions of arity 0 (constants) that coincide with the connectives.

Immediate examples for connectives that come to mind for conjunctors are:

- in Łukasiewicz logic ($\hat{F}(x,y) = max(0, x+y-1)$),
- in Gödel logic ($\hat{F}(x,y) = min(x,y)$),

[1] Although we prefer using suffix notation ($(Term)\sigma$), note that it is equivalent to prefix notation ($\sigma(Term)$).

[2] In some works the term "aggregation operator" subsumes conjunctions, disjunctions and aggregations. In this work we distinguish between them and include a new one (implications).

[3] As usually, a n-ary function $\hat{F}$ is called *monotonic in the i-th argument* ($i \leq n$), if $x \leq x'$ implies $\hat{F}(x_1,\ \ldots,\ x_{i-1},\ x,\ x_{i+1},\ \ldots,\ x_n) \leq \hat{F}(x_1,\ \ldots,\ x_{i-1},\ x',\ x_{i+1},\ \ldots,\ x_n)$ and a function is called *monotonic* if it is monotonic in all arguments.

- in product logic ($\hat{F}(x,y) = x \cdot y$),

for disjunctors:

- in Łukasiewicz logic ($\hat{F}(x,y) = min(1, x+y)$),
- in Gödel logic ($\hat{F}(x,y) = max(x,y)$),
- in product logic ($\hat{F}(x,y) = x \cdot y$),

for implicators:

- in Łukasiewicz logic ($\hat{F}(x,y) = min(1, 1-x+y)$),
- in Gödel logic ($\hat{F}(x,y) = y$ if $x > y$ else 1),
- in product logic ($\hat{F}(x,y) = x \cdot y$)

and for aggregation operators[4]: arithmetic mean, weighted sum or a monotone function learned from data.

3.2 Semantics

The main idea behind our semantics is that if a rule has more priority than the other ones then the intended truth value for an inference where this rule is involved is the one it obtains.

For this purpose we attach to the usual truth value $\mathsf{v} \in [0,1]$ a real number $\mathsf{p} \in [0,1]$ denoting the (accumulated) priority, resulting in the tuple of real numbers between 0 and 1 symbolized by $(\mathsf{p}, \mathsf{v}) \in \Omega^{(0)}$. As it can be noted from the symbols used, the first element indicates the priority and second one the "old" truth value. We represent the tuple by (p,v), although in some cases we use (pv) to highlight that the variable is only one and it can take the value $\perp$. The union between the set containing all possible combinations of two real numbers between 0 and 1 and $\{\perp\}$ is symbolized by $\mathbb{KT}$ and we define the ordering between elements from $\mathbb{KT}$ as follows:

Definition 3.2.1 ($\preccurlyeq_{\mathbb{KT}}$)**.**

$$\perp \preccurlyeq_{\mathbb{KT}} \perp$$
$$\perp \preccurlyeq_{\mathbb{KT}} (\mathsf{p}, \mathsf{v})$$

[4]Note that the above definition of aggregation operators subsumes all kinds of minimum, maximum or mean operators.

$$(\mathsf{p}_1,\ \mathsf{v}_1) \preccurlyeq_{\mathbb{KT}} (\mathsf{p}_2,\ \mathsf{v}_2) \quad \leftrightarrow \quad (\ \mathsf{p}_1 < \mathsf{p}_2\ ) \quad \textit{or} \quad (\ \mathsf{p}_1 = \mathsf{p}_2 \quad \textit{and} \quad \mathsf{v}_1 \leq \mathsf{v}_2\ ) \tag{3.2.1}$$

where $<$ is defined as usually (note that v_i and p_j are just real numbers between 0 and 1). It is obvious that the pair $(\ \mathbb{KT}, \preccurlyeq_{\mathbb{KT}})$ forms a complete lattice.

The structure used to give semantics to our programs is the multi-adjoint algebra, presented in [MOAV01a; MOAV01b; MOAV01c; MOAV02; MOAV04; MO02] and somewhere else. The basic idea is that a multi-adjoint $\Omega-$algebra can be seen as an extension of a multi-adjoint lattice containing a number of extra operators provided by the signature Ω, and a multi-adjoint lattice is just a lattice with more than one pair of operations obeying the adjoint property. We start from the definition of adjoint property.

Definition 3.2.2 (Adjoint property)**.** *Departing from a Poset (a partially ordered set) $<\ P,\ \leq >$ and introducing a pair of operations $(\ \&,\ \leftarrow\ )$, we say that the operations form an adjoint pair if*

(i) $\&$ *is increasing in both arguments,*

(ii) $\leftarrow$ *is increasing in its first argument and decreasing in the second one and*

(iii) *(*the adjoint property*)*[5]

for any $x, y, z \in P$ we have that $z \leq (x \leftarrow y)$ holds if and only if $z \ \& \ y \ \leq \ x$.

A lattice with only one adjoint pair is called somewhere a residuated lattice (see [DP00; DP01]), and when more that one pair is introduced we get to a multi-adjoint lattice.

Definition 3.2.3 (Multi-Adjoint Lattice)**.** *A multi-adjoint lattice $\mathcal{L}$ is a tuple $(L, \leq,\ \leftarrow_1,\ \&_1,\ \ldots,\ \leftarrow_n,\ \&_n)$ satisfying*

(i) $<\ L,\ \leq >$ *is a bounded lattice,*

(ii) $(\ \leftarrow_i,\ \&_i)$ *is an adjoint pair in $<\ L,\ \leq >$, for $i = 1, \ldots, n$ and*

(iii) $\top\ \&_i\ \mathsf{v}\ =\ \mathsf{v}\ \&_i\ \top\ =\ \mathsf{v}$ *for all $\mathsf{v} \in L$ and $i = 1, \ldots, n$.*

Definition 3.2.4 (Multi-Adjoint Algebra)**.** *Let $(L, \leq,\ \leftarrow_1,\ \&_1,\ \ldots,\ \leftarrow_n,\ \&_n)$ be a multi-adjoint lattice. The implication algebra Ω defining the operators $(\ \leftarrow_i,\ \&_i\ )$ for $i = 1, \ldots, n$ with respect to $\mathcal{L}$ is a multi-adjoint algebra.*

[5] Note that the adjoint property offers us a way to evaluate inference rules because $z \leq (x \leftarrow y)$ iff $z \ \& \ y \leq x$ defines the inference rule $\frac{(B,y) \quad (A \leftarrow B, z)}{(A,x)}$

It is usual to define a multi-adjoint logic program as a set of weighted rules $A \xleftarrow{F_c,\ c} F(B_1,\ \ldots,\ B_n)$ where $c \in [0,\ 1]$ and F_c is a conjunctor &, but the semantics associated with this syntax is not capable to manage the priority issues we want to encode. To overcome this restriction we enrich this syntax by changing c by $(\mathsf{p},\ \mathsf{v}) \in \mathbb{KT}$ and adding a condition $COND(A)$ that can be used to encode a truth value to a subset of individuals fulfilling the condition.

Definition 3.2.5 (Multi-Adjoint Logic Program)**.** *A multi-adjoint logic program is a set of clauses of the form*

$$A \overset{(\mathsf{p},\ \mathsf{v}),\&_i}{\longleftarrow} @_i(B_1,\ldots,B_n) \ \textit{if}\ COND(A) \tag{3.2.2}$$

where $(\mathsf{p},\ \mathsf{v}) \in \mathbb{KT}$*,* $\&_i$ *is a conjunctor,* $@_i$ *an aggregator,* A *and* B_i*,* $i \in [1..n]$*, are atoms and* $COND(A)$ *is a first-order formula (a condition that needs to be satisfied for* p *to get the truth value* v*) formed by the predicates in* $\mathrm{TB}_{\Pi,\Sigma,V}$*, the predicates* $=$*,* $\geq$*,* $\leq$*,* $>$ *and* $<$ *restricted to terms from* $\mathrm{TU}_{\Sigma,V}$*, the symbol* $\boldsymbol{t}$ *and the conjunction* $\wedge$ *and disjunction* $\vee$ *in their usual meaning.*

Definitions needed to understand the semantics are given in advance, as usually.

Definition 3.2.6 (Valuation, Interpretation)**.** *A* valuation *or* instantiation $\sigma : V \to \mathbb{HU}$ *is an assignment of ground terms to variables and uniquely constitutes a mapping* $\hat{\sigma} : \mathrm{TB}_{\Pi,\Sigma,V} \to \mathbb{HB}$ *that is defined in the obvious way.*

A fuzzy Herbrand interpretation *(or short,* interpretation*) of a fuzzy logic program is a mapping* $I : \mathbb{HB} \to \mathbb{KT}$ *that assigns an element in our lattice to ground atoms. The* domain *of an interpretation is the set of all atoms in the Herbrand Base, although for readability reasons we omit those atoms to which the truth value* $\bot$ *is assigned (interpretations are total functions). This mapping can be seen as a set of pairs* $(A, (\mathsf{p},\ \mathsf{v}))$ *such that* $A \in \mathbb{HB}$ *and* $(\mathsf{p},\ \mathsf{v}) \in \mathbb{KT} \setminus \{\bot\}$*.*

It is possible to extend uniquely the mapping I *defined on* $\mathbb{HB}$ *to the set of all ground formulas of the language by using the unique homomorphic extension. This extension is denoted* $\hat{I}$ *and the set of all interpretations of the formulas in a program* $\mathbb{P}$ *is denoted* $\mathcal{I}_{\mathbb{P}}$*.*

Definition 3.2.7 (Interpretation Ordering, Minimum, Maximum, Infimum and Supremum)**.** *For two interpretations* I *and* J*, we say* I is less than or equal to J*, written* $I \sqsubseteq J$*, iff* $I(A) \preccurlyeq_{\mathbb{KT}} J(A)$ *for all* $A \in \mathbb{HB}$*. Two interpretations* I *and* J *are* equal*, written* $I = J$*, iff* $I \sqsubseteq J$ *and* $J \sqsubseteq I$*. For all* $A \in \mathbb{HB}$ *minimum is defined as* $min(I,J)(A) = I(A)$ *if* $I(A) \preccurlyeq_{\mathbb{KT}} J(A)$ *and* $min\ (I,\ J)\ (A) = J(A)$ *if* $J(A) \preccurlyeq_{\mathbb{KT}} I(A)$*, maximum as* $max\ (I,\ J)\ (A) = I(A)$ *if* $I(A) \succcurlyeq_{\mathbb{KT}} J(A)$

and $max\ (I,\ J)\ (A) = J(A)$ *if* $J(A) \succcurlyeq_{\mathbb{KT}} I(A)$, *infimum (or intersection) as* $(I \sqcap J)(A) := \min(I(A),\ J(A))$ *and supremum (or union) as* $(I \sqcup J)(A) := \max(I(A),\ J(A))$.

Lemma 3.2.1. *The pair* $(\mathcal{I}_P, \sqsubseteq)$ *of the set of all interpretations of a given program with the interpretation ordering forms a complete lattice.*

Proof. This follows readily from the fact that the underlying lattice set $\mathbb{KT}$ forms a complete lattice with the lattice values ordering $\preccurlyeq_{\mathbb{KT}}$. □

Up to now we have defined the underlying lattice we use for choosing the best interpretation between the available ones, but we still have not defined which is the expected one. For that purpose we define the model of a program, but before it we need to define a operator to combine the knowledge grades, $\circ$.

Definition 3.2.8 (The operator $\circ$)**.** *The aim of taking into account the knowledge quality of every single root involved in the inference process removes the possibility to use mathematical operators in which the result remains unchanged when some input does not (i.e. min, max, etc).*

Besides, the operator must be formed by a pair of functions ($\circ_{\&}$, $\circ_{\leftarrow}$) *where the former is used when combining the knowledge under the application of a conjunction function and the latter when combining it under the implication function.*

This is why we decided to use as operator $\circ_{\&}$ *the mean, defining*

$$x \circ_{\&} y = \frac{x + y}{2} \quad \text{and} \quad z \circ_{\leftarrow} y = 2 * z - y\,.$$

Remark. From here afterwards, the application of some conjunctor $\bar{\&}$ (resp. implicator $\bar{\leftarrow}$, aggregator $\bar{@}$) to elements $(\mathsf{p},\ \mathsf{v}) \in \mathbb{KT} \setminus \{\bot\}$ refers to the application of the truth function $\hat{\&}$ (resp. $\hat{\leftarrow}$, $\hat{@}$) to the second elements of the tuples while $\circ_{\&}$ (resp. $\circ_{\leftarrow}$, $\circ_{\&}$) is the one applied to the first ones.[6]. When applied to the element $\bot$ all of them ($\bar{\&}$, $\bar{\leftarrow}$ and $\bar{@}$) return $\bot$.

Definition 3.2.9 (Multi-Adjoint Satisfaction)**.** *(Modified from the definition in [MO02]) Let* $\mathbb{P}$ *be a multi-adjoint logic program,* $I \in \mathcal{I}_{\mathbb{P}}$ *an interpretation and* $A \in \mathbb{HB}$ *a ground atom. A clause* $Cl_i \in \mathbb{P}$ *of the form* $\{\ A \overset{(\mathsf{p},\ \mathsf{v}),\&_i}{\longleftarrow} @_i(B_1, \ldots, B_n)$ *if* $COND(A)\ \}$ *is satisfied by I iff*

$$(\mathsf{p},\ \mathsf{v}) \preccurlyeq_{\mathbb{KT}} inf\ \{\ \hat{I}\ ((\ A \leftarrow_{\&_i} @_i(B_1,\ \ldots,\ B_n)\ )\ \xi)\ |$$

[6]Note that this new operators $\bar{\&}$, $\bar{\leftarrow}$ and $\bar{@}$ still keep the properties exposed in Sec. 3.1, i.e. the first one is non-decreasing in both coordinates, the second one is non-increasing in the first and non-decreasing in the second coordinate and the last one verifies $\bar{F}(0,\ \ldots,\ 0) = 0$ and $\bar{F}(1,\ \ldots,\ 1) = 1$.

$$\xi \text{ any ground instantiation and } COND(A) \text{ is satisfied } \} \quad (3.2.3)$$

which, by means of the adjoint property, is equivalent to

$$\hat{I}(A) \succcurlyeq_{\mathbb{KT}} sup\{(\mathsf{p}, \mathsf{v}) \ \bar{\&}_{\&_i} \ \hat{I}((@_i(B_1, \ldots, B_n)) \ \xi) \mid \xi \text{ any ground instantiation and } COND(A) \text{ is satisfied } \} \quad (3.2.4)$$

Definition 3.2.10 (Satisfaction, Model)**.** *Let $\mathbb{P}$ be a multi-adjoint logic program, $I \in \mathcal{I}_{\mathbb{P}}$ an interpretation and $A \in \mathbb{HB}$ a ground atom. We say that a clause $Cl_i \in \mathbb{P}$ is satisfied by I or I* is a model of the clause *Cl_i ($I \Vdash Cl_i$) iff for all ground atoms $A \in \mathbb{HB}$ and for all instantiations σ for which $B\sigma \in \mathbb{HB}$ (note that σ can be the empty substitution) it is true that*

$$\hat{I}(A) \succcurlyeq_{\mathbb{KT}} (\mathsf{p}, \mathsf{v}) \ \bar{\&}_i \ \bar{@}_i(\hat{I}(B_1\sigma), \ldots, \hat{I}(B_n\sigma)) \quad \text{if} \quad COND(A) \quad (3.2.5)$$

Note that eq. 3.2.5 is equivalent to eq. 3.2.4. Finally, we say that I is a model of the program $\mathbb{P}$ *and write $I \Vdash \mathbb{P}$ iff $I \Vdash Cl_i$ for all clauses in our multi-adjoint logic program $\mathbb{P}$.*

Every program has a least model, which is usually regarded as the intended interpretation of the program, since it is the most conservative model. The following proposition will be an important prerequisite to define the least model semantics. It states that the infimum (or intersection) of a non-empty set of models of a program will again be a model. The existence of a least model is then obvious and easily defined as the intersection of all models.

Proposition 3.2.1 (Model intersection property)**.** *Let $\mathbb{P}$ be a multi-adjoint logic program and $\mathcal{I}_{\mathbb{P}}$ be a non-empty set of interpretations. Then*

$$I \Vdash \mathbb{P} \text{ for all } I \in \mathcal{I}_{\mathbb{P}} \text{ implies } \prod_{I \in \mathcal{I}_{\mathbb{P}}} I \Vdash \mathbb{P}$$

Proof. Suppose that for all $I \in \mathcal{I}_{\mathbb{P}}$ it is true that $I \Vdash \mathbb{P}$. We define $J = \prod_{I \in \mathcal{I}_{\mathbb{P}}} I$.

(step. 1) from the definition of model of a program (Def. 3.2.10) we have that $I \Vdash \mathbb{P}$ iff $I \Vdash Cl_i$ for all clauses in our program $\mathbb{P}$, and this results in

$$\hat{I}(A) \succcurlyeq_{\mathbb{KT}} (\mathsf{p}, \mathsf{v}) \ \bar{\&}_i \ \bar{@}_i(\hat{I}(B_1\sigma), \ldots, \hat{I}(B_n\sigma)) \quad \text{if} \quad COND(A)$$

for all atoms $A \in \mathbb{HB}$ and for all instantiations σ for which $B\sigma$ is ground.

(step. 2) from $J = \prod_{I \in \mathcal{I}_{\mathbb{P}}} I$ and the definition of $\prod$ as the minimum between interpretations (Def. 3.2.7) we have that for all $I \in \mathcal{I}_{\mathbb{P}}$ it is true that for all $L \in \mathbb{HB}$ $(I \sqcap J)(L) := \min(I(L), J(L)) = J(L)$.

(step. 3) define for some ground atom $A \in \mathbb{HB}$ and some ground instantiation σ such that $B\sigma \in \mathbb{HB}$ the variables $(kv)_{I,\sigma}$ and $(kv)_{J,\sigma}$ as follows:

$$\begin{aligned} \hat{I}(A) \succcurlyeq_{\mathbb{KT}} (kv)_{I,\sigma} &= (\mathsf{p},\ \mathsf{v})\ \bar{\&}_i\ \bar{@}_i(\ \hat{I}(B_1\sigma),\ \ldots,\ \hat{I}(B_n\sigma)\ ) \quad \text{if} \quad COND(\ A\ ) \\ (kv)_{J,\sigma} &= (\mathsf{p},\ \mathsf{v})\ \bar{\&}_i\ \bar{@}_i(\ \hat{J}(B_1\sigma),\ \ldots,\ \hat{J}(B_n\sigma)\ ) \quad \text{if} \quad COND(\ A\ ) \end{aligned}$$

from the definition of $\bar{\&}_i$ and $\bar{@}_i$ as non decreasing functions and $\hat{I}(B_i\sigma) \succcurlyeq_{\mathbb{KT}} \hat{J}(B_i\sigma)$ (step. 2) it is clear that $(kv)_{I,\sigma} \succcurlyeq_{\mathbb{KT}} (kv)_{J,\sigma}$.

(step. 4) from $J = \bigsqcap_{I \in \mathcal{I}_{\mathbb{P}}} I$ and the definition of $\bigsqcap$ as the minimum between interpretations (Def. 3.2.7) we have that for some $I \in \mathcal{I}_{\mathbb{P}}$ and some $L \in \mathbb{HB}$ it is true that $(I \sqcap J)(L) := \min(I(L), J(L)) = J(L) = I(L)$.

(step. 5) from $(kv)_{I,\sigma} \succcurlyeq_{\mathbb{KT}} (kv)_{J,\sigma}$ (step. 3) and the fact that $\hat{J}(A)$ gets its value from some $\hat{I}(A)$ (step. 4) we can fix for some atom A and any substitution σ the order $(1,\ 1) \succcurlyeq_{\mathbb{KT}} \hat{I}(A) \succcurlyeq_{\mathbb{KT}} \hat{J}(A) \succcurlyeq_{\mathbb{KT}} (kv)_{I,\sigma} \succcurlyeq_{\mathbb{KT}} (kv)_{J,\sigma} \succcurlyeq_{\mathbb{KT}} \bot$,

(step. 6) The order in step. 5 defines $\hat{J}(A) \succcurlyeq_{\mathbb{KT}} (kv)_{J,\sigma}$, so

$$\hat{J}(A) \succcurlyeq_{\mathbb{KT}} (kv)_{J,\sigma} = (\mathsf{p},\ \mathsf{v})\ \bar{\&}_i\ \bar{@}_i(\ \hat{J}(B_1\sigma),\ \ldots,\ \hat{J}(B_n\sigma)\ ) \quad \text{if} \quad COND(\ A\ )$$

which proves Prop. 3.2.1. □

Definition 3.2.11. *Let $\mathbb{P}$ be a well-defined fuzzy logic program. The* least model of $\mathbb{P}$ *is defined as* $\mathsf{lm}(\mathbb{P}) := \bigsqcap_{I \Vdash \mathbb{P}} I$.

Definition 3.2.12 (T_P Operator)**.** *Let $\mathbb{P}$ be a multi-adjoint logic program, $L \in \mathbb{HB}$ an atom and $I \in \mathcal{I}_{\mathbb{P}}$ an interpretation. The immediate consequences operator $T_P : \mathcal{I}_{\mathbb{P}} \to \mathcal{I}_{\mathbb{P}}$ is defined as follows:*

$$\begin{aligned} T_P(I)(A) \doteq \ sup\ \{\ & (\mathsf{p},\ \mathsf{v})\ \bar{\&}_i\ \bar{@}_i(\ \hat{I}(B_1\sigma),\ \ldots,\ \hat{I}(B_n\sigma)\ ) \quad \textit{if} \quad COND(\ A\ )\ \mid \\ & \{\ A \xleftarrow{(\mathsf{p},\ \mathsf{v}),\&_i} @_i(B_1,\ldots,B_n)\ \textit{if}\ COND(A)\ \}\ \in \mathbb{P}\ \} \end{aligned} \tag{3.2.6}$$

As it is usual in the logic programming framework, the semantics of a program $\mathbb{P}$ is characterized by the post-fixpoints of T_P.

Proposition 3.2.2. *Let $\mathbb{P}$ be a multi-adjoint logic program and $I \in \mathcal{I}_{\mathbb{P}}$ an interpretation.*

$$I \Vdash \mathbb{P} \quad \Leftrightarrow \quad T_P(I) \sqsubseteq I. \tag{3.2.7}$$

Proof. "if": Let $T_P(I) \sqsubseteq I$ and L be some arbitrary ground atom. Define for any ground instantiation σ the variable $(kv)_{I,\sigma}$ as follows:

$$(kv)_{I,\sigma} \doteq \quad (\mathsf{p},\ \mathsf{v})\ \bar{\&}_i\ \bar{@}_i(\ \hat{I}(B_1\sigma),\ \ldots,\ \hat{I}(B_n\sigma)\ ) \quad \text{if} \quad COND(\ A\ )\ \mid$$

$$\{\ A \xleftarrow{(\mathsf{p},\ \mathsf{v}),\&_i} @_i(B_1,\ldots,B_n)\ \text{ if }\ COND(A)\ \} \in \mathbb{P} \quad (3.2.8)$$

so we can say that for any ground instantiation σ

$$\hat{I}(A) \succcurlyeq_{\mathbb{KT}} (kv)_{I,\sigma} \quad (3.2.9)$$
$$T_P(I)(A) \doteq= sup\ \{\ (kv)_{I,\sigma}\ \} \quad (3.2.10)$$

and from this and the definition of the symbols sup and $\succcurlyeq_{\mathbb{KT}}$ we can fix the order $\hat{I}(A) \succcurlyeq_{\mathbb{KT}} T_P(I)(A)$, proving that $I \Vdash \mathbb{P}$.

"only if": Let $I \Vdash \mathbb{P}$ and L be some arbitrary ground atom. We define for any ground instantiation σ the variable $(kv)_{I,\sigma}$ as in Eq. 3.2.8. Since $I \Vdash \mathbb{P}$, for any ground instantiation σ Eq. 3.2.9 has to be true. If we define our T_P operator from the variable $(kv)_{I,\sigma}$, as in Eq. 3.2.10, we know from the definition of the symbols sup and $\succcurlyeq_{\mathbb{KT}}$ that $\hat{I}(A) \succcurlyeq_{\mathbb{KT}} T_P(I)(A)$, so $T_P(I) \sqsubseteq I$. □

Proposition 3.2.3 (T_P is monotonic)**.** *Let $\mathbb{P}$ be a multi-adjoint logic program and $I_i \in \mathcal{I}_{\mathbb{P}}$ and $I_{i+1} \in \mathcal{I}_{\mathbb{P}}$ two interpretations. if $I_i \sqsubseteq I_{i+1} \Rightarrow T_P(I_i) \sqsubseteq T_P(I_{i+1})$.*

Proof. Suppose that $I_i \sqsubseteq I_{i+1}$. By definition of $\sqsubseteq$ this implies that for all atoms L $\hat{I}_i(L) \preccurlyeq_{\mathbb{KT}} \hat{I}_{i+1}(L)$. In the definition of T_P operator (Def. 3.2.12) $T_P(I)(L)$ is related to $I(L)$ by means of the operations sup, $\&_i$ and $@_i$. Since all of them are non-decreasing and monotone, we can assure that $T_P(I_i)(L) \preccurlyeq_{\mathbb{KT}} T_P(I_{i+1})(L)$ and conclude $T_P(I_i) \sqsubseteq T_P(I_{i+1})$ □

Proposition 3.2.4 (T_P is continuous)**.** *Let $\mathbb{P}$ be a multi-adjoint logic program and $I_0 \sqsubseteq I_1 \sqsubseteq \ldots$ a countable infinite increasing sequence of interpretations. Then $T_P\left(\bigsqcup_{n=0}^{\infty} I_n\right) = \bigsqcup_{n=0}^{\infty} T_P(I_n)$.*

Proof. We use the following facts:

(fact. 1) Since $I_0 \sqsubseteq I_1 \sqsubseteq \ldots$ and from definition of $\sqsubseteq$ we have that $I_i(A) \preccurlyeq_{\mathbb{KT}} I_{i+1}(A)$ for every ground term $A \in \mathbb{HB}$. As $\bigsqcup$ takes by definition the maximum interpretation, $\bigsqcup_{i=0}^{n} I_i = I_n$.

(fact. 2) We have that $T_P(I_0) \sqsubseteq T_P(I_1) \sqsubseteq \ldots$ since $I_0 \sqsubseteq I_1 \sqsubseteq \ldots$ and T_P is monotonic (Prop. 3.2.3). Again by using definitions of $\sqsubseteq$ and $\bigsqcup$ we obtain $\bigsqcup_{i=0}^{n} T_P(I_i) = T_P(I_n)$.

$$T_P\left(\bigsqcup_{n=0}^{\infty} I_n\right) \overset{fact.\ 1}{=} T_P\left(I_\infty\right) \overset{fact.\ 2}{=} \bigsqcup_{n=0}^{\infty} T_P(I_n)$$

□

Theorem 3.2.2. *Let $\mathbb{P}$ be a multi-adjoint logic program. Then the least fixpoint of T_P exists and is equal to $T_P{\uparrow}\omega$.*

Proof. The existence of the least fixpoint of T_P follows from the facts that $(\mathcal{I}_{\mathbb{P}}, \sqsubseteq)$ forms a complete lattice, T_P is monotone (Proposition 3.2.3), and the Knaster-Tarski fixpoint theorem [Kna28; Tar55]. Its equality to $T_P{\uparrow}\omega$ follows from the facts that $(\mathcal{I}_{\mathbb{P}}, \sqsubseteq)$ forms a complete lattice, T_P is continuous (Proposition 3.2.4), and the Kleene fixpoint theorem [Kle52]. □

Since the least fixpoint always exists, we can define a semantics based on it.

Definition 3.2.13. *Let $\mathbb{P}$ be a multi-adjoint logic program. Then the* least fixpoint semantics of $\mathbb{P}$ *is defined as* $\mathsf{lfp}(\mathbb{P}) = T_P{\uparrow}\omega(\bot)$. *Here, $\bot$ denotes the interpretation mapping everything to $\bot$ (thus being the least element of the lattice $(\mathcal{I}_P, \sqsubseteq)$).*

Theorem 3.2.3. *For a multi-adjoint logic program $\mathbb{P}$, we have*

$$\mathsf{lm}(\,\mathbb{P}\,) \;=\; \mathsf{lfp}(\,\mathbb{P}\,). \tag{3.2.11}$$

Proof.

$$\mathsf{lm}(\mathbb{P}) \overset{1}{=} \bigsqcap_{I \,\Vdash\, \mathbb{P}} I \overset{2}{=} \bigsqcap_{T_P(I) \sqsubseteq I} I \overset{3}{=} T_P{\uparrow}\omega(\bot) \overset{4}{=} \mathsf{lfp}(\mathbb{P})$$

where (1) is by definition of least model of a program, (2) is by Prop. 3.2.2, (3) is by the Kleene fixpoint theorem [Kle52] and (4) is by definition of least fixpoint semantics. □

Chapter 4

Extending the framework with similarity and negation: RFuzzy v.3

In RFuzzy [MHPCS11; PCMH11] the authors define a particular case of the multi-adjoint semantics [MOAV01a; MOAV01b; MOAV01c; MOAV02; MOAV04; MO02] in which the variable v is substituted by a tuple (p, v) for taking into account the rule's priority value p in the computations. In addition to this they slightly modify the rules' syntax and semantics to include the possibility to decide if a rule is evaluated or not (at evaluation time) from the values introduced in its arguments. In this work we depart from the improvements achieved in their work, fix a small error they have in keeping the multi-adjoint semantics[1] and introduce some interesting novelties. We have introduced the use of modifiers, even the negation modifier, similarity between attributes, similarity between fuzzy predicates, syntax for the inclusion of new modifiers and connectives, connection to non-fuzzy databases and links between the non-fuzzy characteristics in the databases and fuzzy predicates.

4.1 Syntax

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 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

[1] The operator $\circ_{\leftarrow}$ defined in Def. 3.2.8 can obtain results not in the set $[\,0,\,1\,]$, which is not allowed in the multi-adjoint semantics.

Herbrand universe $\mathbb{HU}$ is the set of all ground terms, and the *Herbrand base* $\mathbb{HB}$ is the set of all atoms with arguments from the Herbrand universe. A substitution σ or ξ is (as usual) a mapping from variables from V to terms from $\mathrm{TU}_{\Sigma,V}$ and can be represented in suffix ($(Term)\sigma$) or in prefix notation ($\sigma(Term)$).

To capture different interdependencies between predicates, we will make use of a signature Ω of *many-valued connectives* formed by *conjunctions* $\&_1, \&_2, ..., \&_k$, *disjunctions* $\vee_1, \vee_2, ..., \vee_l$, *implications* $\leftarrow_1, \leftarrow_2, ..., \leftarrow_m$, *aggregations* $@_1, @_2, ..., @_n$, *modifiers* $\diamond_1, \diamond_2, ..., \diamond_o$, *negation operators* $\neg_1, \neg_2, ..., \neg_p$, real numbers in the interval $[0,\ 1]$ represented by v when we talk about the truth value or credibility of a rule and tuples of real numbers in the interval $[0,\ 1]$ represented by $(\mathsf{p},\ \mathsf{v})$ when we talk about the truth value or credibility of a rule with priority.

While Ω denotes the set of connective symbols, $\hat{\Omega}$ denotes the set of their respective associated truth functions. Instances of connective symbols and truth functions are denoted by $\&_i$ and $\hat{\&}_i$ for conjunctors, $\vee_i$ and $\hat{\vee}_i$ for disjunctors, $\leftarrow_i$ and $\hat{\leftarrow}_i$ for implicators, $@_i$ and $\hat{@}_i$ for aggregators, $\diamond_i$ and $\diamond_i$ for modifiers, $\neg_i$ and $\hat{\neg}_i$ for negation operators, v and $\hat{\mathsf{v}}$ for the truth value or the credibility and $(\mathsf{p},\ \mathsf{v})$ and $(\mathsf{p},\hat{\ }\mathsf{v})$ for the tuples of truth values or credibility values and priority values.

Truth functions for the connectives are then defined as $\hat{\&} : [0,1]^2 \to [0,1]$ monotone[2] and non-decreasing in both coordinates, $\hat{\vee} : [0,1]^2 \to [0,1]$ monotone in both coordinates, $\hat{\leftarrow} : [0,1]^2 \to [0,1]$ non-increasing in the first and non-decreasing in the second coordinate, $\hat{@} : [0,1]^n \to [0,1]$ as a function that verifies $\hat{@}(0,\ \ldots,\ 0) = 0$ and $\hat{@}(1,\ \ldots,\ 1) = 1$, $\diamond : [0,1] \to [0,1]$ as a function without special restrictions, $\hat{\neg} : [0,1] \to [0,1]$ non-increasing and satisfying $\hat{\neg}(0) = 1$ and $\hat{\neg}(1) = 0$ and $\mathsf{v} \in \Omega^{(0)}$ and $(\mathsf{p},\ \mathsf{v}) \in \Omega^{(0)}$ are functions of arity 0 (constants) that coincide with the symbols used to represent them. More properties and a more specific and detailed classification of connectives (based on their properties) can be found in the work of De Cock and Kerre [CK04].

Immediate examples for connectives that come to mind for conjunctors, disjunctors and implicators are shown in Table. 4.1.1, where Ł stands for Łukasiewicz logic, Gö for Gödel logic and prod for product logic. For aggregation operators[3] the usual ones are arithmetic mean, weighted sum or a monotone function learned from data, for modifiers the "very" function,

[2] As usually, a n-ary function $\hat{F}$ is called *monotonic in the i-th argument* ($i \leq n$), if $x \leq x'$ implies $\hat{F}(x_1,\ \ldots,\ x_{i-1},\ x,\ x_{i+1},\ \ldots,\ x_n) \leq \hat{F}(x_1,\ \ldots,\ x_{i-1},\ x',\ x_{i+1},\ \ldots,\ x_n)$ and a function is called *monotonic* if it is monotonic in all arguments.

[3] Note that the above definition of aggregation operators subsumes all kinds of minimum, maximum or mean operators.

$\hat{very}(x) = x^2$, and for negation the function $\hat{\neg}(x) = 1 - x$.

Table 4.1.1: Examples of conjunctors, disjunctors and implicators

	conjunctor	disjunctor	implicator
synt.	A $\&_i$ B	A $\vee_i$ B	A $\rightarrow$ B
	$\hat{A} = x \ ; \ \hat{B} = y$		
Ł	$max(0, x + y - 1)$	$min(1, x + y)$	$min(1, 1 - x + y)$
Gö	$min(x, y)$	$max(x, y)$	y if $x > y$ else 1
prod	$x \cdot y$	$x \cdot y$	$x \cdot y$

4.2 Syntactic constructions for writing programs

In this section we explain all the syntactic structures available for writing programs. Since an exposition of syntactic structures turns out to be a little bit difficult to read we do it from a practical point of view. FleSe is a web interface for RFuzzy. It offers the final user a human-oriented environment for querying RFuzzy and shows its answers using a more human-oriented representation too. We take examples from it to facilitate the understanding of the syntactic constructions presented here.

Once the user has chosen a program (or configuration file) FleSe asks him what he is looking for (Fig. 4.2.1). The things we can look for depend on the virtual databases' tables defined in the program file loaded. Before showing how to define them we show how to relate them to one or more database tables.

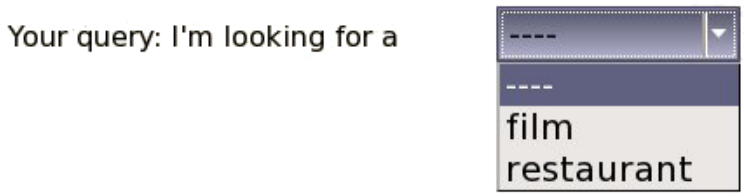


Figure 4.2.1: Dialog to select what you are looking for.

The database tables storing the information of what we call a virtual database table can be more than one, which is why we decided to allow the programmer to use the Prolog facilities for mixing all the information into a

Prolog predicate. This Prolog predicate is what we use to define the virtual database table. The syntax used to define where is the database table (host name or ip and port), the user name and password and the protocol for connecting to it (SQL) is shown in Eq. 4.2.1, while the one for defining the table structure (the columns' names) and how we link them to the predicate arguments is in Eq. 4.2.2. Suppose that we have two tables for storing the information of a restaurant, one for the "food type" (*restaurant_food_type*) and another for the "distance to the city center" (*restaurant_dist_to_tcc*). We can do the operations in Eqs. 4.2.3, 4.2.4, 4.2.5 and 4.2.6 to obtain all the information about a restaurant. If instead of that we have all the information of a restaurant in just one table we can make use of the code in Eq. 4.2.7. The removal of columns (in case we do not need the information stored there) can be done in a similar way to the union of columns from different tables (Eq. 4.2.6), as it is shown in Eq. 4.2.8.

$$sql\_persistent\_location(database\_id, \\ db('SQL', user, pass, 'host' : port)). \tag{4.2.1}$$

$$:-sql\_persistent(\\ predicate\_name(Prolog\ type\ for\ each\ column), \\ database\_table\_name(columns'\ names), \\ database\_id). \tag{4.2.2}$$

$$sql\_persistent\_location(myDatabase, \\ db('SQL', 'me', 'myPass', 'localhost' : 1521)). \tag{4.2.3}$$

$$:-sql\_persistent(\\ rest\_food\_type(integer, string), \\ restaurant\_food\_type(id, food\_type), \\ myDatabase). \tag{4.2.4}$$

$$:-sql\_persistent(\\ rest\_dist\_to\_tcc(integer, integer), \\ restaurant\_dist\_to\_tcc(id, dist\_to\_tcc), \\ myDatabase). \tag{4.2.5}$$

$$
\begin{aligned}
&restaurant(Id,\ Food\_type,\ Dist\_to\_tcc) :- \\
&\quad rest\_food\_type(Id,\ Food\_type), \\
&\quad rest\_dist\_to\_tcc(Id,\ Dist\_to\_tcc).
\end{aligned} \tag{4.2.6}
$$

$$
\begin{aligned}
&:- sql\_persistent(\\
&\quad restaurant(integer,\ string,\ integer), \\
&\quad restaurant(id,\ food\_type,\ dist\_to\_tcc), \\
&\quad myDatabase).
\end{aligned} \tag{4.2.7}
$$

$$
\begin{aligned}
&restaurant(Id,\ Food\_type,\ Dist\_to\_tcc) :- \\
&\quad rest\_food\_type(Id,\ Food\_type, \\
&\qquad\qquad Dist\_to\_tcc,\ Avr\_price).
\end{aligned} \tag{4.2.8}
$$

Once we have defined a predicate for accessing all the information we use the syntax in Eq. 4.2.9 to define the virtual database table (vdbt). In eq. 4.2.9 pT is the name of the vdbt (the individual or subject of our searches, which corresponds to the name of the predicate defined for accessing the database), pA is the arity of the predicate used to define the vdbt (and the number of columns the vdbt has), pN is the name assigned to a column of the vdbt pT and pT' is a basic type, one of $\{boolean\_type,\ enum\_type,\ integer\_type,\ float\_type,\ string\_type\}$. We provide an example in Eq. 4.2.10 to clarify, in which the restaurant vdbt has seven columns (or the predicate has seven arguments), the first for the name of the restaurant, the second for the restaurant type, the third for the food type served in the restaurant, the fourth for the number of years since its opening, the fifth for the distance to the city center from that restaurant, the sixth for the restaurant's price average and the last one for the price of the restaurant's menu. Please note that none of the previous definitions of predicates for accessing the database can be used in conjunction with the definition of vdbt in Eq. 4.2.10. We provide a valid one in Eqs. 4.2.11 and 4.2.12.

$$define\_database(pT/pA, [(pN, pT')]). \tag{4.2.9}$$

$$\begin{aligned}
&define_database(restaurant/7, [\\
&\quad (name, string_type),\\
&\quad (restaurant_type, enum_type),\\
&\quad (food_type, enum_type),\\
&\quad (years_since_opening, integer_type),\\
&\quad (distance_to_the_city_center, integer_type),\\
&\quad (price_average, integer_type),\\
&\quad (menu_price, integer_type)]).
\end{aligned} \tag{4.2.10}$$

$$\begin{aligned}
&:-sql_persistent(\\
&\quad restaurantAux(integer, string, string, string,\\
&\qquad\qquad integer, integer\ integer, integer),\\
&\quad restaurant(restaurant_id, name,\\
&\qquad\qquad restaurant_type, food_type,\\
&\qquad\qquad years_since_opening,\\
&\qquad\qquad distance_to_the_city_center,\\
&\qquad\qquad price_average, menu_price),\\
&\quad myDatabase).
\end{aligned} \tag{4.2.11}$$

$$\begin{aligned}
&restaurant(Name, Restaurant_type,\\
&\qquad Food_type, Years_since_opening,\\
&\qquad Distance_to_the_city_center,\\
&\qquad Price_average, Menu_price) :-\\
&\quad restaurant(Restaurant_id,\\
&\qquad\qquad Name, Restaurant_type,\\
&\qquad\qquad Food_type, Years_since_opening,\\
&\qquad\qquad Distance_to_the_city_center,\\
&\qquad\qquad Price_average, Menu_price).
\end{aligned} \tag{4.2.12}$$

As told before, this syntactical construction provides a value for the combo of things that we can look for (the values given to pT, "restaurant" this time, are the values shown in this combo), but this is not its only function. When

we choose what we are looking for FleSe shows us a combo of characteristics of the object that we can use to filter our search, and a plus sign to its right (Fig. 4.2.2). The plus sign serves to add more conditions. We focus on it later. The combo (Fig. 4.2.3) allows us to chose a characteristic of the thing we are looking for and, depending if we chose one defined as fuzzy or not, FleSe shows us the possibility to use negation and/or a modifier for it (Fig. 4.2.4) or an operator to compare the non-fuzzy value stored in the database for the thing and the non-fuzzy value we want to compare to (Figs. 4.2.4 and 4.2.5).

Figure 4.2.2: Dialog to filter our search

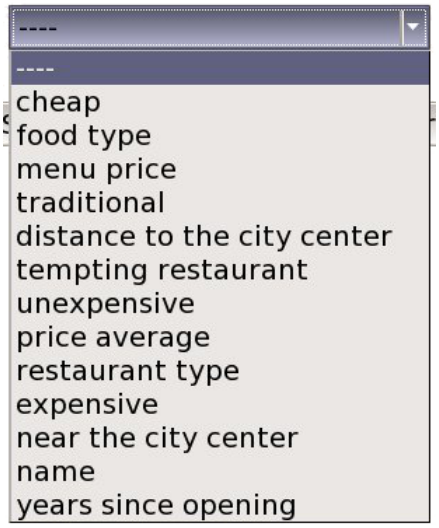


Figure 4.2.3: Available characteristics for the thing we are looking for.

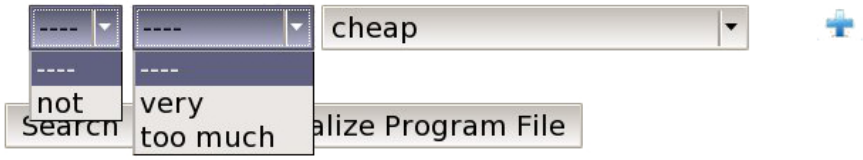


Figure 4.2.4: Available negation and modifier operators for the fuzzy characteristic chosen.

The characteristics shown in Fig. 4.2.3 are fuzzy and non-fuzzy characteristics of the thing we are looking for (a restaurant this time). The non-fuzzy ones

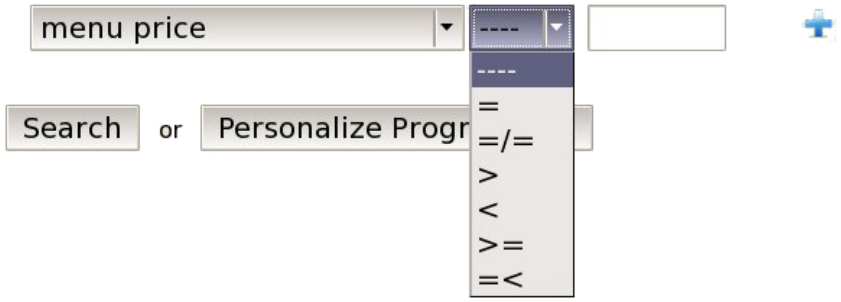


Figure 4.2.5: Available comparison operators and input field for the value we want to use in the comparison, for the non-fuzzy characteristic chosen.

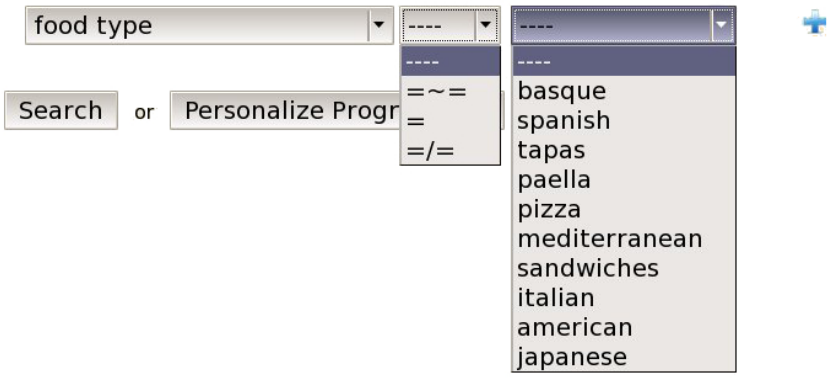


Figure 4.2.6: Available comparison operators and values for the non-fuzzy characteristic chosen, of type “enum_type”.

are just the virtual database table columns (See Eq. 4.2.10) because in our work we have not focused into allowing the definition of non-fuzzy characteristics from other characteristics. When the user chooses one we use the “types” assigned to each column (*boolean_type*, *enum_type*, *integer_type*, *float_type* or *string_type*) to determine the type of the comparison operator and if we can offer a list of values for the comparison (Fig. 4.2.6). We present in Table. 4.2.1 the comparison operators defined for each type. The list of values shown in Fig. 4.2.6 is only available when the type of the column is *enum_type* and depends on the existing values in the database.

The fuzzy characteristics in Fig. 4.2.3, in contrast with the non-fuzzy ones, can be defined in the configuration file and we can do it using fuzzy and non-fuzzy characteristics. Before entering in how this is done we explain how the non-fuzzy characteristics are available for defining fuzzy characteristics. From a sentence of the form in Eq. 4.2.9 RFuzzy creates a predicate for each column. Each one of this predicates has as first argument the thing and as second

Table 4.2.1: Comparison operators available by the things characteristics' type

type	operator	meaning of operator
boolean_type	=	"is equal to"
	= / =	"is different from"
enum_type	=	"is equal to"
	= / =	"is different from"
	=~=	"is similar to"
interger_type	=	"is equal to"
float_type	= / =	"is different from"
	>	"is bigger than"
	<	"is lower than"
	>=	"is bigger than or equal to"
	=<	"is lower than or equal to"
string_type	=	"is equal to"
	= / =	"is different from"

argument the value for the characteristic, so it serves to obtain the value for a specific thing, to obtain all the things having some value for the characteristic or to obtain all the things and their respective value for the characteristic[4]. Taking the virtual database table definition example in Eq. 4.2.10, RFuzzy provides us with the predicates in Eq. 4.2.13.

We can use five syntactic constructions to define fuzzy characteristics. Fuzzy characteristics are, in principle, characteristics whose satisfiability depends on one or more non-fuzzy characteristics for which we know their non-fuzzy value (because it is in some column in the database), but not always. We can define their satisfiability from the satisfiability of other fuzzy characteristic and we can even define a fixed value for them in the configuration file. We start by the constructions or rules, in our humble opinion, simpler and explain one by one what they serve for.

[4]This predicates' behaviour is the usual in Prolog and we have kept it in RFuzzy. The facility that provides it is called backtracking.

$$
\begin{aligned}
&name(R,\ Value) :-\ ...\\
&restaurant\_type(R,\ Value) :-\ ...\\
&food\_type(R,\ Value) :-\ ...\\
&years\_since\_opening(R,\ Value) :-\ ...\\
&distance\_to\_the\_city\_center(R,\ Value) :-\ ...\\
&price\_average(R,\ Value) :-\ ...\\
&menu\_price(R,\ Value) :-\ ...
\end{aligned}
\tag{4.2.13}
$$

The first construction we can use to define fuzzy characteristics (Eq. 4.2.14) serves to define the rare situation in which for all the individuals in the virtual database table we have the same result for the fuzzy characteristic. The interest in having such a construction comes from the possibility to limit the individuals for which it returns the fixed value, by using the tail construction in Eq. 4.2.15 as tail of Eq. 4.2.14. Besides, RFuzzy allows us to use too the tail constructions in Eqs. 4.2.16 and 4.2.17, aimed respectively at assigning some credibility to the rule and limiting its use to some user name (this is useful for personalizing the rule). We explain the three tail constructions (Eqs. 4.2.15, 4.2.16 and 4.2.17) with more detail below. In the syntactic construction in Eq. 4.2.14 pT is the name of the vdbt (as in Eq. 4.2.9), TV is the truth value (a float number between 0 and 1) and $fPredName$ is the name of the fuzzy characteristic we are defining. In Eq. 4.2.18 we present an example in which we say that all the restaurants are cheap with a truth value of 0.5.

$$
\begin{aligned}
&fPredName(pT) :\sim value(TV) && (4.2.14)\\
&\quad if(\ pN(pT)\ \ comp\ \ value) && (4.2.15)\\
&\quad with\_credibility(credOp,\ credVal) && (4.2.16)\\
&\quad only\_for\_user\ 'UserName' && (4.2.17)
\end{aligned}
$$

$$
cheap(restaurant) :\sim value(0.5) \tag{4.2.18}
$$

The tail constructions in Eqs. 4.2.15, 4.2.16 and 4.2.17 serve as tails for the constructions in Eqs. 4.2.14, 4.2.22, 4.2.24, 4.2.27, 4.2.28, 4.2.30, 4.2.32 and 4.2.34. As told before, they serve, respectively, to slightly modify the behaviour of the "main" rule (or construction). Eq. 4.2.15 serves to limit the individuals in the virtual database table for which it is used, Eq. 4.2.16 to assign some credibility to it and 4.2.17 to limit its use to some user name. We explain them in detail in the following paragraphs.

The tail construction in Eq. 4.2.15 (not usable when the "main" construction is Eq. 4.2.34) serves to limit the individuals for which we wanna use the fuzzy clause or rule (limits its application to subsets of the set of individuals in the vdbt). In Eq. 4.2.15 pN and pT are the name of a column of a virtual database table and the vdbt name (as in Eq. 4.2.9), $comp$ is a comparison operator and can take the values "is_equal_to", "$is\_different\_from$", "is_bigger_than", "is_lower_than", "$is\_bigger\_than\_or\_equal\_to$" and "$is\_lower\_than\_or\_equal\_to$" and $value$ can be of type $integer\_type$, $enum\_type$ or $string\_type$. The only restrictions are that $value$ must be of the type assigned to the column pN of the vdbt pT and that if they are of type $enum\_type$ or $string\_type$ the only comparison operators available (the only values RFuzzy allows for $comp$) are "is_equal_to" and "$is\_different\_from$". We show an example in Eq. 4.2.19 in which we say that the restaurant Zalacain is cheap with a truth value of 0.1.

$$cheap(restaurant) :\sim value(0.1) \\ if(name(restaurant)\ is\_equal\_to\ zalacain). \tag{4.2.19}$$

The tail construction in Eq. 4.2.16 serves to define the credibility of a rule, together with the operator needed to combine it with the truth value resulting from evaluating the rule's body. In its syntactic definition in Eq. 4.2.16 $credVal$ is the credibility, a number of float type, and $credOp$ is the credibility operator. The credibility operator is a conjunctor, a mathematical function that must be monotone and non-decreasing in their coordinates. We are allowed to use any of the examples of conjunctors in Subsec. 4.1 ("prod" for the product conjunctor, "luka" for the Łukasiewicz conjunctor and "min" for the Gödel conjunctor) or we can define our own conjunctors, by using the syntactic construction in Eq. 4.2.37. We show an example in Eq. 4.2.20 in which we say that the restaurant Don Jamon is cheap with a truth value of 0.3 but this rule has a credibility of 0.8 and the operator that must be used to combine the credibility with the truth value is the minimum (called too Gödel conjunctor).

$$cheap(restaurant) :\sim value(0.3) \\ if(name(restaurant)\ is\_equal\_to\ don\_jamon) \\ with\_credibility(min,\ 0.8). \tag{4.2.20}$$

The tail construction in Eq. 4.2.17 is aimed at defining personalized rules, rules that only apply when the user logged in and the user in the rule are the same one. In the construction $Username$ is the name of the user, a string. We show an example in Eq. 4.2.21 in which we say that Lara considers that the

restaurant Zalacain is not close to the center. So, if it is she who poses a query to the system asking for restaurants close to the city center, she will obtain that the Zalacain restaurant is not close (or close with a value zero).

$$\begin{aligned} &close\_to\_the\_city\_center(restaurant) :\sim value(0) \\ &\quad if(name(restaurant)\ is\_equal\_to\ zalacain) \\ &\quad only\_for\_user\ \ 'Lara' \end{aligned} \tag{4.2.21}$$

The second construction we can use to define fuzzy characteristics (Eq. 4.2.22) is usually called fuzzification function and serves to define the link between a non-fuzzy characteristic for which we have numerical values in the database and the fuzzy characteristic we are defining. This relation is what RFuzzy uses to determine how much satisfied is a fuzzy characteristic for some individual stored in our database, from the non-fuzzy and numerical value that we have in the database for the non-fuzzy characteristic of the individual. In Eq. 4.2.22 $fPredName$ is the name of the fuzzy characteristic we are defining (and the name of the fuzzification), pT is the name of the vdbt, pN is the name of the non-fuzzy characteristic (or the name of the column in the vdbt) and $[(valIn,\ valOut)]$ is a list of pairs of values such that $valIn$ belongs to the set of values that the non-fuzzy characteristic can take (the domain of the fuzzification function) and $valOut$ to the set of values that the fuzzy characteristic can take (the fuzzification function image, always a subset of $[0,1]$)[5]. An example in which we compute how cheap is a restaurant from its average price is presented in Eq. 4.2.23. The graphical representation corresponding to this example is in Fig. 4.2.7.

$$\begin{aligned} &fPredName(pT) :\sim function(\ pN(\ pT\ ), \\ &\qquad\qquad [\ (valIn,\ valOut)\ ]). \end{aligned} \tag{4.2.22}$$

$$\begin{aligned} &cheap(restaurant) :\sim function(\\ &\quad price\_average(restaurant), \\ &\quad [(0,\ 1), (10,\ 1), (20,\ 0.9), (50,\ 0), (200, 0)]). \end{aligned} \tag{4.2.23}$$

When defining the satisfiability of a fuzzy characteristic using a fuzzification function we can get an unexpected behaviour if the database contains a null

[5] $[(valIn, valOut)]$ is basically a piecewise function definition, where each two contiguous points represent a piece.

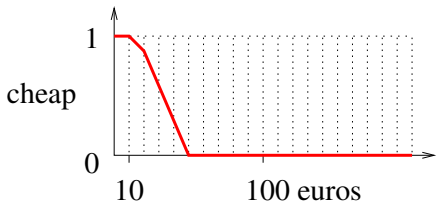


Figure 4.2.7: Cheap function (for restaurant).

value in the field that it uses. This malfunctioning of the system consists in the exclusion of the individual from the list of results, because RFuzzy cannot compute results for it. To avoid this behaviour RFuzzy allows us to define the satisfaction of the fuzzy concepts in this cases, by using the construction for entering default truth values, truth values that are used only when no better result can be computed. The idea of allowing the programmer to encode default truth values has been taken from the works of Muñoz-Hernández and Vaucheret [MHV06; MHV07]. The syntax to define default truth values is shown in Eq. 4.2.24, where $fPredName$ is the name of the fuzzy characteristic for which we are defining the default truth value, pT the name of the vdbt and TV the truth value we want to use in this cases. We provide two examples in Eqs. 4.2.25 and 4.2.26 in which we say that, in absence of information, we consider that a restaurant will not be close to the city center (this is what the zero value means) and that, in absence of information, a restaurant is considered to be medium cheap[6].

$$fPredName(pT) :\sim defaults\_to(TV) \tag{4.2.24}$$

$$close\_to\_the\_city\_center(restaurant) :\sim defaults\_to(0). \tag{4.2.25}$$

$$cheap(restaurant) :\sim defaults\_to(0.5). \tag{4.2.26}$$

There is another cause of malfunctioning of a Prolog system (remember that RFuzzy is developed in Prolog), which is the existence of two rules such that the satisfiability of each one of them depends on the satisfiability of the other one (this is informally called a loop). Although some authors, as Loyer and Stracia in [LS03], study this problem and propose solutions that we could have used, we decided to consider this kind of programs erroneous. Our humble opinion

[6]We include two examples here so if one builds a program by taking all the examples in the contribution the rule in Eq. 4.2.29 works for all the restaurants in our database.

is that they are more part of philosophical studies than of practical applications. We simply highlight that if the configuration file contains a program with loops RFuzzy will loop until its consumption of memory reaches the amount of memory available in the system. At that moment the program will just exit showing an error to the user.

The third construction we can use to define fuzzy characteristics is called fuzzy rule. A fuzzy rule allows us to define the satisfaction of a fuzzy characteristic from the satisfaction of other fuzzy characteristic. We have two syntactical forms for defining fuzzy rules, the first one used when the satisfiability of the fuzzy characteristic we are defining depends on only one fuzzy characteristic, shown in Eq. 4.2.27, and the second one when it depends on more than one, shown in Eq. 4.2.28. In Eq. 4.2.27 $fPredName2$ is a fuzzy characteristic previously defined, $fPredName$ is the name of the fuzzy characteristic we are defining and pT is the name of the vdbt, while in Eq. 4.2.28 $aggr$ is the aggregator used to combine the truth values of the fuzzy characteristics between parenthesis, $fPredName2$ and $fPredName3$ are fuzzy characteristics previously defined, $fPredName$ is the name of the fuzzy characteristic we are defining and pT is again the name of the vdbt. We show an example in Eq. 4.2.29 in which we say that a restaurant is a tempting restaurant depending on the worst value it has between being close to the center and being cheap, which means that a restaurant must be close to the center and cheap at the same time to consider it a tempting restaurant.

$$fPredName(pT) :\sim rule(fPredName2(pT)) \tag{4.2.27}$$

$$\begin{aligned} fPredName(pT) :\sim rule(aggr, (&fPredName2(pT),\\ &fPredName3(pT),\\ &\dots)) \end{aligned} \tag{4.2.28}$$

$$\begin{aligned} tempting\_restaurant(restaurant) :\sim rule(\ &min,\\ &(\ close\_to\_the\_city\_center(restaurant),\\ &\ cheap(restaurant)\ )) \end{aligned} \tag{4.2.29}$$

The fourth construction we can use to define fuzzy characteristics is for defining a fuzzy characteristic as a synonym of another one already defined. We present its form in Eq. 4.2.30, where $fPredName$ is the name of the fuzzy characteristic we are defining, $fPredName2$ the fuzzy characteristic from which we are defining it and pT the name of the vdbt. The value of $fPredName2$ must be different from the value of $fPredName$. We show an example in Eq. 4.2.31

in which we define an unexpensive restaurant as (almost) the synonym of a cheap one (something being unexpensive might not always be cheap).

$$fPredName(pT) :\sim synonym\_of(\ fPredName2(pT)) \tag{4.2.30}$$

$$unexpensive(restaurant) :\sim synonym\_of(cheap(restaurant)) \; with\_credibility(prod,\ 0.9). \tag{4.2.31}$$

The fifth construction we can use to define fuzzy characteristics is for defining a fuzzy characteristic as an antonym of another already defined. We present its form in Eq. 4.2.32, where $fPredName$ is the name of the fuzzy characteristic we are defining, $fPredName2$ the fuzzy characteristic from which we are defining it, and pT the name of the vdbt. The value of $fPredName2$ must be different from the value of $fPredName$. We show an example in Eq. 4.2.33 in which we define an expensive restaurant as the antonym of a cheap one.

$$fPredName(pT) :\sim antonym\_of(fPredName2(pT)). \tag{4.2.32}$$

$$expensive(restaurant) : antonym\_of(cheap(restaurant)). \tag{4.2.33}$$

In addition to the five constructions presented for defining fuzzy characteristics the framework offers us to enrich the knowledge by defining new aggregation operators, new modifiers, new negation operators and relating the values in the database that we see as synonyms by defining similarity between attributes. We introduce now each one of the constructions that RFuzzy offers us for each one of this facilities.

The similarity between attributes allows us to create a link between values in the database, and gives sense to the allowance of the operator "$=\sim=$" (similar to) in a query (see Fig.4.2.6). We start by an example to justify its necessity. Suppose we are looking for a "Mediterranean" food restaurant and we have in the database the values "Mediterranean", "Spanish", "Italian", "Portuguese", etc. It is clear that the previous values are all "Mediterranean", but the search engine does not know about the existing relation, and it will not return the restaurants with values different than "Mediterranean" as answers for the

query. The construction we present serves just to tell the framework about this relation, allowing the user to ask for restaurants serving food similar to the "Mediterranean" one and getting as answers the restaurants whose value for food served we define similar to the "Mediterranean" one. The syntax is shown in Eq.4.2.34, where pT is the name of a virtual database table, pN the name of a column of the vdbt, TV a truth value (a float number in $[0,1]$), and $value1$ and $value2$ two valid (and different) values for the vdbt column pN of the vdbt pT. In the example in Eq. 4.2.35 we say that the food type mediterranean is 0.8 similar to the spanish one.

$$similarity\_between(pT,\ pN(value1),\ pN(value2),\ TV). \tag{4.2.34}$$

$$similarity\_between(\ restaurant,\ food\_type(mediterranean),\ food\_type(spanish),\ \ 0.8). \tag{4.2.35}$$

It is important to highlight that in Eq. 4.2.35 we say that the food type mediterranean is 0.8 similar to the spanish one, but not in the other way. If we want to say that the spanish food is at some point similar to the mediterranean one we need to add another line of code saying that. RFuzzy does not add the inverse relation because, as told in the introduction (Sec. sec:intro), we do not force the similarity relation to be reflexive, symmetric and transitive, i.e., an equivalence relation. As some of the authors referenced in Sec. sec:intro mention, forcing it to be an equivalence relation is too restrictive for real-world applications and this is just what we present here. Suppose, for example, that we want to say that the spanish food is similar to the mediterranean one. Since looking for a spanish food restaurant and eating in a Mediterranean food restaurant is not the same as looking for a Mediterranean one and eating in a spanish one, we assign 0.6 to the similarity between spanish and Mediterranean, instead of the 0.8 assigned to the opposite relation. We provide another example in which (we think) the idea of how to use similarity is clarified. Suppose we want to encode that if we are looking for a spanish food restaurant the restaurants serving tapas are valid ones (tapas is clearly spanish food). One could think that a restaurant serving spanish food could be valid for someone looking for one serving tapas, but most of the spanish restaurants serving spanish food do not serve tapas at all. So, we cannot model such an idea. The lines of code representing this last example are shown in 4.2.36. The last line of the example is optional and corresponds to the tail in Eq. 4.2.16. It is there

just to show that we can assign credibility to the similarity between attributes (but in this case it is superfluous). We can use too the tail in Eq. 4.2.17 to personalize the rule (in case we want or need it).

$$\begin{aligned} &similarity\_between(\ restaurant,\\ &\qquad food\_type(spanish),\\ &\qquad food\_type(tapas),\quad 0.7)\\ &\qquad with\_credibility(prod,\ 1). \end{aligned} \tag{4.2.36}$$

The framework provides us with the connectives product (*prod*), arithmetic mean (*mean*), minimum (*min*) and Łukasiewicz product (*luka*), but we might need to define our own ones. The syntax for defining new connectives is shown in Eq. 4.2.37, where *Name* is the name of the connective we are defining and *prolog_code* is the prolog code that defines how the three variables that the predicate has as arguments are related. This three variables are the three last arguments of *define_connective*: *Var_In_1*, *Var_In_2* and *Var_Out*. The definition is done by using Prolog code, so any formula can be encoded. We present an example in Eq. 4.2.38 in which we define the connective *max_but_at_most_a_half*, which computes the maximum of the inputs and returns the value if it is under 0.5, or 0.5 if over. When defining new connectives programmers must ensure that they satisfy the properties of conjunctors, disjunctors, etc if they want to use them as conjunctors, disjunctors, etc. The framework does not check if they are satisfied to be as flexible as possible.

$$\begin{aligned} &define\_connective(Name/3,\ Var\_In\_1,\\ &\qquad Var\_In\_2,\ Var\_Out) :-\\ &\quad prolog\_code. \end{aligned} \tag{4.2.37}$$

$$\begin{aligned} &define\_connective(max\_but\_at\_most\_a\_half/3,\\ &\qquad TV\_In\_1,\ TV\_In\_2,\ TV\_Out) :-\\ &\quad max(TV\_In\_1,\ TV\_In\_2,\ TV\_Aux),\\ &\quad min(TV\_Aux,\ 0.5,\ TV\_Out). \end{aligned} \tag{4.2.38}$$

Modifiers serve, in principle, to slightly modify the meaning of a fuzzy characteristic. Zadeh called them hedges in [Zad72] and classified them into type I and type II. The difference between them is that type I are operators acting on a fuzzy set of individuals while type II require a description of how they act on the components. Typical examples[7] of type I hedges are “very”,

[7] We owe most of the examples to Zadeh’s paper [Zad72].

"more or less", "much", "too much", "slightly", "highly". Some of type II are "essentially", "technically", "actually", "strictly", "in a sense", "practically", "virtually", "regular". The modifiers or hedges we can use in RFuzzy belong (this might change in the near future) to type I. From the big set of available type I hedges we have included only two in the framework: "very" (X^2) and "too much" (X^3). The reason for not including more is that the selection of the function to apply when they are chosen (X^2, X^3, ...) depends on the author's preferences and is most of the times directed by the set of data chosen. Since the framework allows to define new operators this is not a drawback, but an advantage: it is not needed to use names for them like "my_very". The syntax for defining new modifiers is shown in Eq. 4.2.39, where Name is the name of the modifier we are defining and *Var_In* and *Var_Out* are the names of the variables that can be used in the Prolog code written in *prolog_code* to define the modifier behaviour. We show an example in Eq. 4.2.40 in which we define the output value of *a_little* as the square root of the input value.

$$define\_modifier(Name/2,\ Var\_In,\ Var\_Out) :- \\ prolog\_code. \tag{4.2.39}$$

$$define\_modifier(a\_little/2,\ TV\_In,\ TV\_Out) :- \\ TV\_Out * TV\_Out\ .=.\ TV\_In. \tag{4.2.40}$$

Negation is an operation that takes a value and obtains another one. Most people associates it with obtaining "no" from "yes" and "yes" from "no", but here it can be a little bit more complicated. In principle, from the fact that we are modeling fuzzy logic truth values with numbers between 0 and 1, the semantics for the negation operator could be $Output = 1 - Input$, but there are more proposals for the definition of this operator. For example, Esteva, Godo, Hájek and Navara study in [Est+00] the residuated fuzzy logics arising from continuous t-norms without non-trivial zero divisors and an involutive negation and Flaminio and Marchioni study in [FM06] the addition of arbitrary involutive negations to t-norm-based logics. Since the definition of the negation operator might depend on the environment and we want the framework to be as customizable as possible, we offer the possibility to define new negation operators, by means of the syntax shown in Eq. 4.2.41. We present in Eq. 4.2.42 the Gödel negation studied by Borgwardt and Peñaloza in [BP12]. The function implemented by this Gödel negation simply maps 0 to 1 and everything else to 0.

$$define\_negation\_op(Name/2,\ Var\_In,\ Var\_Out) :- \\ prolog\_code. \tag{4.2.41}$$

$$define\_negation\_op(godel\_neg/2,\ TV\_In,\ TV\_Out) :- \\ (\ (TV\_In. = .0,\ \ TV\_Out. = .1)\ ; \\ (\backslash + (TV\_In. = .0),\ \ TV\_Out. = .0)). \tag{4.2.42}$$

4.3 The semantics of the framework's configuration file

The semantics of a programming language (in this case the one defined for the syntax allowed in the framework's configuration file) can be developed in multiple ways. In our case we provide declarative and operational semantics, and prove that they are equivalent. The reason for doing it in this way is that the first ones allow us to easily reason and prove properties[8] while the other ones are more interested in providing a procedure to automatically (without human intervention) get the results[9]. By proving their equivalence we prove that the operational semantics are sound (that every result in the set of results of the operational semantics is in the set of the declarative ones) and complete (that every result in the declarative semantics is in the set of results of the operational ones).

The declarative and operational semantics of RFuzzy are really a particular case of the declarative and operational semantics of the Multi-Adjoint framework. This allows us to reuse most of the improvements that some authors have proposed for the Multi-Adjoint framework, as the ones presented in the works of Julian, Medina, Morcillo, Moreno and Ojeda-Aciego [Jul+09; Jul+11]. The particularization comes basically from the introduction of priorities in the evaluation process, needed to help the computation process to choose the result with the highest priority value or, when multiple results have the highest priority value, the one (between them) with the highest truth value. The works of Pablos-Ceruelo and Muñoz-Hernández [MHPCS11; PCMH11] served as basis for this purpose. In the first one the authors propose a framework with an incipient priorities system, only able to distinguish when a truth value has

[8] They have a strong mathematical base, which allows, for example, to prove that the interpretation chosen as the intended semantics of the program is a model.

[9] They show the mechanism used by the computer to obtain the results, allowing to reason about efficiency, resources consumption, etc.

been provided by a rule, by the combination of a rule and one or more default value assignments or just by one or more default value assignments. In the last one the authors introduce a priorities system closer to the one we have here, by taking ideas from some works in which the authors introduce priorities in logic programming (LP), as Wang, Zhou and Fangzhen in [WZL00], Analyti and Pramanik in [AP95], Laenens and Vermeir in [LV90], Marek, Nerode and Remmel in [MNR97], Jayaraman, Govindarajan and Mantha in [JGM07] and Delgrande, Schaub and Tompits in [DST03]. The differences between this work and the proposal of Pablos-Ceruelo and Muñoz-Hernández [PCMH11] are mainly due to the inclusion of modifiers and negation operators.

The semantics we present are divided in two. In the first part (Subsec. 4.3.1) we present a structure and its semantics and in the second one (Subsec. 4.3.2) we present the translation of the syntactical constructions we have introduced in Subsec. 4.2 to this one. We do it in this way because we consider that the presentation is much more simpler than if we present the semantics of each construction.

4.3.1 Low level semantics

Multi-adjoint logic programs (see Def. 1.4.6) are usually defined as a set of weighted rules

$$< A \leftarrow_i F(B_1, \ldots, B_n), c > \tag{4.3.1}$$

where $c \in [0, 1]$ is the credibility assigned to the rule, i a conjunctor $\&$, F an aggregator $@_i$ and A and B_i, $i \in [1..n]$, atoms. Sometimes the syntax used is

$$A \xleftarrow{F_c,\, c} F(B_1, \ldots, B_n) \tag{4.3.2}$$

where F_c corresponds to the i in the equation before (it is a conjunctor $\&$). In [PCMH11] Pablos-Ceruelo and Muñoz-Hernández enrich the framework with the capability to deal with priorities, changing $c \in [0, 1]$ by $(\mathsf{p}, \mathsf{v}) \in \mathbb{KT}$, and add the possibility to have conditional rules, rules that are only used when the individual A for which we are computing the truth value fulfills the condition $COND(A)$. The programs they can write are sets of weighted rules

$$A \xleftarrow{(\mathsf{p}, \mathsf{v}),\, \&_i} @_j(B_1, \ldots, B_i, \ldots, B_n) \text{ if } COND(A) \tag{4.3.3}$$

where $(\mathsf{p}, \mathsf{v}) \in \mathbb{KT}$, $\&_i$ is a conjunctor, $@_i$ an aggregator, A and B_i, $i \in [1..n]$, are atoms and $COND(A)$ is a first-order formula (a condition that needs to be satisfied for p to get the truth value v) formed by the predicates in $\mathrm{TB}_{\Pi,\Sigma,V}$, the

predicates $=$, $\neq$, $\geq$, $\leq$, $>$ and $<$ restricted to terms from $\mathrm{TU}_{\Sigma,V}$, the symbol **t** and the conjunction $\wedge$ and disjunction $\vee$ in their usual meaning.

Our definition for multi-adjoint programs is in Def. 4.3.4, but we introduce first some concepts needed to understand it. First goes $\mathbb{KT}$, the set of valid values for the tuples (p, v) and, just after, the ordering of values. Def. 4.3.3 is novel with respect to the work of Pablos-Ceruelo and Muñoz-Hernández [PCMH11] and is needed for the inclusion of modifiers and negation operators in the semantics.

Definition 4.3.1 ($\mathbb{KT}$)**.** *$\mathbb{KT}$ is the set of valid values for the variable (pv), which represents at the same time a (accumulated) priority (p) and a truth value (v). Although it is just one variable, we usually split its value in two for readability reasons, resulting in (p, v). We do that because p and v are just real numbers between 0 and 1: $\mathsf{v} \in [0,1]$ and $\mathsf{p} \in [0,1]$. Writing them together complicates unnecessarily the reading of their values. The reason for considering them a unique variable comes from sources: there is a value that represents that we could not find a valid value for it, $\{\bot\}$, and the ordering between values forms a complete lattice. So, the set of valid values for the variable (pv) is:*

$$\mathbb{KT} = \{\bot\} \cup \{\ (\mathsf{p},\ \mathsf{v}) \mid \mathsf{v} \in [0,\ 1] \wedge \mathsf{p} \in [0,\ 1]\ \} \tag{4.3.4}$$

Definition 4.3.2 ($\preccurlyeq_{\mathbb{KT}}$)**.** *The ordering between the values in $\mathbb{KT}$ is fixed by the definition of $\preccurlyeq_{\mathbb{KT}}$:*

$$\begin{aligned} \bot &\preccurlyeq_{\mathbb{KT}} \bot \\ \bot &\preccurlyeq_{\mathbb{KT}} (\mathsf{p},\ \mathsf{v}) \\ (\mathsf{p}_1,\ \mathsf{v}_1) \preccurlyeq_{\mathbb{KT}} (\mathsf{p}_2,\ \mathsf{v}_2) &\leftrightarrow (\ \mathsf{p}_1 < \mathsf{p}_2\ ) \textit{ or} \\ &\quad (\ \mathsf{p}_1 = \mathsf{p}_2 \textit{ and } \mathsf{v}_1 \leq \mathsf{v}_2\ ) \end{aligned} \tag{4.3.5}$$

where $<$ is defined as usually (remember that v_i and p_j are just real numbers between 0 and 1). It is obvious that the pair $(\mathbb{KT}, \preccurlyeq_{\mathbb{KT}})$ forms a complete lattice.

Remark. From the definitions of $\mathbb{KT}$ and $\preccurlyeq_{\mathbb{KT}}$, we get that $(\mathbb{KT}, \preccurlyeq_{\mathbb{KT}})$ forms a complete lattice and, assuming that we have a non-empty set of adjoint pairs (see Def. 1.4.3), we can get that our lattice is a Multi-Adjoint lattice (see Def. 1.4.4) and the algebra we define a Multi-Adjoint algebra (see Def. 1.4.5).

Definition 4.3.3 (Basic formula)**.** *A basic formula can be an atom (Eq. 4.3.6), the application of a modifier or a negation operator to an atom (Eqs. 4.3.7 and 4.3.8 resp.) or the application of a negation operator to the application of a modifier to an atom (Eq. 4.3.9).*

$$A \tag{4.3.6}$$

$$\Diamond(A) \tag{4.3.7}$$
$$\neg(A) \tag{4.3.8}$$
$$\neg(\Diamond(A)) \tag{4.3.9}$$

Definition 4.3.4 (Multi-Adjoint Logic Program)**.** *A multi-adjoint logic program is a set of clauses of the form*

$$\{\ A \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} @_j(D_1, \ldots, D_i, \ldots, D_n) \quad \textit{if}\ \ COND(A)\ \} \tag{4.3.10}$$

where $(\mathsf{p},\ \mathsf{v}) \in \mathbb{KT}$*,* $\&_i$ *is a conjunctor,* $@_i$ *an aggregator,* A *an atom, each* D_i*,* $i \in [1..n]$*, a basic formula (see Def. 4.3.3) and* $COND(A)$ *a first-order formula (a boolean condition that needs to be satisfied for* A *to get the value computed by the rule) formed by the predicates in* $\mathrm{TB}_{\Pi,\Sigma,V}$*, the predicates* $=$*,* $\neq$*,* $\geq$*,* $\leq$*,* $>$ *and* $<$ *restricted to terms from* $\mathrm{TU}_{\Sigma,V}$*, the symbol* $\mathbf{t}$ *and the conjunction* $\wedge$ *and disjunction* $\vee$ *in their usual meaning.*

If $n = 1$ *then* $@_j$ *is omitted (there is no need for an aggregator to combine the tuples of two or more basic formulas* D_i *because there is only one) and we represent it with the form*

$$A \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} D \tag{4.3.11}$$

If $n = 0$ *the clause is intended to be used for assigning a truth value to an atom, with more or less credibility. In this case there is no aggregator nor basic formulas in the clause's body and we represent it as follows*

$$A \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} (\mathsf{p}',\ \mathsf{v}') \tag{4.3.12}$$

where $(\mathsf{p}',\ \mathsf{v}')$ *is the truth value and priority assigned to the fact (Please note that* $(\mathsf{p},\ \mathsf{v})$ *is still the credibility assigned to the rule).*

Definition 4.3.5 (Valuation)**.** *A* valuation *or* instantiation $\sigma : V \to \mathbb{HU}$ *is an assignment of ground terms to variables and uniquely constitutes a mapping* $\hat{\sigma} : \mathrm{TB}_{\Pi,\Sigma,V} \to \mathbb{HB}$ *that is defined in the obvious way.*

Definition 4.3.6 (Interpretation)**.** *A* fuzzy Herbrand interpretation *(or short,* interpretation*) of a fuzzy logic program is a mapping* $I : \mathbb{HB} \to \mathbb{KT}$ *that assigns an element in our lattice to ground atoms. The* domain *of an interpretation is the set of all atoms in the Herbrand Base, although for readability reasons we omit those atoms to which the element assigned is* $\bot$ *(interpretations are total functions). This mapping can be seen as a set of pairs* $(A,\ \hat{(\mathsf{p},\ \mathsf{v})})$ *such that* $A \in \mathbb{HB}$ *and* $(\mathsf{p},\ \mathsf{v}) \in \mathbb{KT} \setminus \{\bot\}$*.*

It is possible to extend uniquely the mapping I defined on $\mathbb{HB}$ to the set of all ground formulas of the language by using the unique homomorphic extension. This extension is denoted $\hat{I}$ and the set of all interpretations of the formulas in a program $\mathbb{P}$ is denoted $\mathcal{I}_{\mathbb{P}}$.

Definition 4.3.7 (Interpretation Ordering, Minimum, Maximum, Infimum and Supremum)**.** *For two interpretations I and J, we say I* is less than or equal to *J, written $I \sqsubseteq J$, iff $I(A) \preccurlyeq_{\mathbb{KT}} J(A)$ for all $A \in \mathbb{HB}$. Two interpretations I and J are* equal*, written $I = J$, iff $I \sqsubseteq J$ and $J \sqsubseteq I$. Minimum (Eq. 4.3.13), maximum (Eq. 4.3.14), infimum (or intersection, Eq. 4.3.15) and supremum (or union, Eq. 4.3.16) for all $A \in \mathbb{HB}$ are defined as follows:*

$$min(I,J)(A) = \begin{cases} I(A) & \text{if } I(A) \preccurlyeq_{\mathbb{KT}} J(A) \\ J(A) & \text{if } I(A) \succ_{\mathbb{KT}} J(A) \end{cases} \tag{4.3.13}$$

$$max(I,J)(A) = \begin{cases} I(A) & \text{if } I(A) \succcurlyeq_{\mathbb{KT}} J(A) \\ J(A) & \text{if } I(A) \prec_{\mathbb{KT}} J(A) \end{cases} \tag{4.3.14}$$

$$(I \sqcap J)(A) = \min(I(A), J(A)) \tag{4.3.15}$$

$$(I \sqcup J)(A) = \max(I(A), J(A)) \tag{4.3.16}$$

Lemma 4.3.1. *The pair $(\mathcal{I}_P, \sqsubseteq)$ of the set of all interpretations of a given program with the interpretation ordering forms a complete lattice.*

Proof. This follows readily from the fact that the underlying lattice set $\mathbb{KT}$ forms a complete lattice with the lattice values ordering $\preccurlyeq_{\mathbb{KT}}$. □

Up to now we have defined the underlying lattice we use for choosing the best interpretation between the available ones, but we still need to define which ones are considered valid for our program and which one is the expected one. We define first which interpretations are considered to be model of a program and, before this one, the concepts needed to understand it.

One of the concepts we define is the operator used to combine (or modify) the priority of the tuples $(p, \hat{} \, v)$ when the programmer chooses a connective to combine (or modify) the truth values of one or more subgoals. This one is needed due to the fact that the truth functions associated to the connectives of arity bigger than zero in Ω (see Subsec. 4.1) are intended to be used with just truth values, and not with tuples $(p, \hat{} \, v)$. In order to overcome this limitation we provide the priority operator $\circ$.

Definition 4.3.8 (The operator $\circ$)**.** *The operator $\circ$ is intended to be used for taking care of the combination of the priorities while the truth function associated to the connective chosen takes care of the truth values.*

The aim of taking into account the priority of every single root (each D_i, $i \in 1 \ldots n$) involved in the inference process removes the possibility to use mathematical operators in which the result remains unchanged when some input does not (i.e. min, max, etc). Besides, the operator must be defined for each possible connective. The function we assign to the operator (dependent of the type of connective) is shown in Eqs. 4.3.17 - 4.3.22.

$$x \circ_{\&} y = \frac{x + y}{2} \tag{4.3.17}$$

$$z \circ_{\leftarrow} y = max(0, min(1, 2 * z - y)) \tag{4.3.18}$$

$$x \circ_{\vee} y = \frac{x + y}{2} \tag{4.3.19}$$

$$x \circ_{@} y = \frac{x + y}{2} \tag{4.3.20}$$

$$\circ_{\neg}(x) = x \tag{4.3.21}$$

$$\circ_{\diamond}(x) = x \tag{4.3.22}$$

The functions in Eqs. 4.3.17, 4.3.18, 4.3.19 and 4.3.20 are extended as usual when applied to more than two input values.

Remark. From here afterwards, the application of the function associated to any connective of arity bigger than zero refers to the application of the truth function defined for the connective to the truth value in the tuples $(p, \hat{v})$ and the application of the corresponding priority operator $\circ$ to the priorities in the tuples. When one of the input tuples takes the value $\bot$ the result will be always $\bot$.

Remark. Although the operators $\bar{\&}$, $\bar{\leftarrow}$, $\bar{\vee}$ and $\bar{@}$ still keep the properties exposed in Sec. 4.1,[10] the priority value and the priority operators are intended to be used for having distinct rules for the general situation and the exception(s). What we mean is that, while in the original multi-adjoint framework the rules get their credibility directly from real-world scenarios[11] the inclusion of the priority offers the possibility to get a high credibility for a rule representing

[10] The first one is non-decreasing in both coordinates, the second one non-increasing in the first and non-decreasing in the second coordinate, the third one monotone in both coordinates and the last one is a function that verifies $\bar{F}(0, \ldots, 0) = 0$ and $\bar{F}(1, \ldots, 1) = 1$.

[11] If whenever A occurs B occurs then the rule $B \leftarrow A$ gets a high credibility. See examples 1.4.1 and 1.4.2.

Table 4.3.1: Table with the credibility of the rule "if it is cloudy, it rains" for some cities

city	if it is cloudy, it rains
Barcelona	0.7
Valencia	0.7
A Coruña	0.9
Madrid	0.2
Sevilla	0.1

most of the individuals of a set and a low credibility for those elements of the set for which the rule is not satisfied up to the same degree. Suppose, for example, that we have two rules: "if it is cloudy, it rains" and "if it rains, it is cloudy". The second one is true with a credibility of 0.9 and we can model it without priorities, but for the first one and the set of data in Table. 4.3.1 the credibility of the first rule would be 0.1 (the minimum). What we offer is the possibility to say that the rule has a credibility of 0.1 for Sevilla, a credibility of 0.2 for Madrid and a credibility of 0.7 for the other spanish cities, by giving a higher priority to the first two rules.

Definition 4.3.9 (Multi-Adjoint Satisfaction)**.** *Let $\mathbb{P}$ be a multi-adjoint logic program, $I \in \mathcal{I}_{\mathbb{P}}$ an interpretation and $A \in \mathbb{HB}$ a ground atom. A clause $Cl_i \in \mathbb{P}$ of the form*

$$\{\ A \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} @_j(D_1,\ldots,D_i,\ldots,D_n)\ \textit{if}\ \ COND(A)\ \} \tag{4.3.23}$$

is satisfied by I iff

$$\begin{aligned}(\mathsf{p},\hat{}\ \mathsf{v}) \preccurlyeq_{\mathbb{KT}}\ \ inf\ \{\ & \hat{I}\ (\ A\ ) \leftarrow_i \hat{I}\ ((\ @_i(D_1,\ \ldots,\ D_n)\ )\ \xi)\ \mid \\ & \xi\ \ \textit{any ground instantiation and} \\ & COND(\ A\ )\ \ \textit{is satisfied}\ \}\end{aligned} \tag{4.3.24}$$

which, by means of the adjoint property (the tuple $(\&_i,\ \leftarrow_i)$ is an adjoint pair), is equivalent to

$$\begin{aligned}\hat{I}\ (\ A\ )\ \succcurlyeq_{\mathbb{KT}}\ \ sup\ \{\ & (\mathsf{p},\hat{}\ \mathsf{v})\ \hat{\&}_i\ \hat{I}((\ @_i(D_1,\ \ldots,\ D_n)\ )\ \xi)\ \mid \\ & \xi\ \ \textit{any ground instantiation and} \\ & COND(\ A\ )\ \ \textit{is satisfied}\ \ \}\end{aligned} \tag{4.3.25}$$

Remark. Although it is obvious, if the rule is of the form

$$A \xleftarrow{(\mathsf{p},\, \mathsf{v}),\, \&_i} D \tag{4.3.26}$$

then $\hat{I}((\ @_i(D_1,\ \ldots,\ D_n)\ )\ \xi\ )$ is replaced by $\hat{I}(\ D\ \xi\ )$ and if it is of the form

$$A \xleftarrow{(\mathsf{p},\, \mathsf{v}),\, \&_i} (\mathsf{p}',\, \mathsf{v}') \tag{4.3.27}$$

then by $\hat{I}(\ (\mathsf{p}',\, \mathsf{v}')\ )$.

Definition 4.3.10 (Satisfaction, Model)**.** *Let $\mathbb{P}$ be a multi-adjoint logic program, $I \in \mathfrak{I}_{\mathbb{P}}$ an interpretation and $A \in \mathbb{HB}$ a ground atom. We say that a clause $Cl_i \in \mathbb{P}$ of the form*

$$\{\ A \xleftarrow{(\mathsf{p},\, \mathsf{v}),\, \&_i} @_j(D_1, \ldots, D_j, \ldots, D_n)\ \textit{if}\ \ COND(A)\ \} \tag{4.3.28}$$

is satisfied by I or I is a model of the clause *Cl_i ($I \Vdash Cl_i$) iff for all ground atoms $A \in \mathbb{HB}$ and for all instantiations σ for which $D_j\sigma$ is a ground basic formula (note that σ can be the empty substitution) it is true that*

$$\begin{gathered}\hat{I}(A) \succcurlyeq_{\mathbb{KT}} (\mathsf{p}, \hat{}\ \mathsf{v})\ \hat{\&}_i\ \hat{@}_i(\ \hat{I}(D_1\sigma),\ \ldots,\ \hat{I}(D_n\sigma)\ ) \\ \textit{if}\ \ COND(A)\end{gathered} \tag{4.3.29}$$

Note that eq. 4.3.29 is equivalent to eq. 4.3.25. Finally, we say that I is a model of the program *$\mathbb{P}$ and write $I \Vdash \mathbb{P}$ iff $I \Vdash Cl_i$ for all clauses in our multi-adjoint logic program $\mathbb{P}$.*

Every program has a model[12] and it is usual to have more than one. Between all of them one must be chosen as the program declarative semantics. This is what we present now but, before doing it, we need to define stratification. Stratification is needed because the existence of negation (fuzzy, but negation) in a program forces us to determine the interpretation that we want to be the declarative semantics of our program strata by strata. More information about stratification can be found in the work of Przymusinski [Prz89b].

Definition 4.3.11 (Stratification, adapted and extended from the definition proposed by Przymusinski in [Prz89b])**.** *A general program $\mathbb{P}$ is stratified if and*

[12] This affirmation should be proved, but it is trivial to do it from the existence of an stratification of the program (for more details we refer to the work of Przymusinski [Prz89b]). The existence of an stratification can be proved easily from the fact that we do not allow loops in programs.

only if it is possible to decompose the set S of all predicates of $\mathbb{P}$ into disjoint sets $S_1, \ldots, S_r$, called strata, so that for every clause $Cl_i \in \mathbb{P}$ of the form

$$\{ A \xleftarrow{(\mathsf{p},\, \mathsf{v}),\, \&_i} @_j(D_1, \ldots, D_j, \ldots, D_n) \quad \text{if } COND(A) \}, \tag{4.3.30}$$

where every symbol means the same as in Eq. 4.3.10, we have that:

$$\forall\, i\ stratum(A) \geq stratum(D_i) \ \text{ if } D_i \text{ has the form } B \text{ or } \Diamond B \tag{4.3.31}$$

$$\forall\, i\ stratum(A) > stratum(D_i) \ \text{ if } D_i \text{ has the form } \neg B \text{ or } \neg\Diamond B \tag{4.3.32}$$

We define stratum of a program $\mathbb{P}$ *as the maximum stratum of the atoms A appearing in it:*

$$\forall\, A \in \mathbb{HB},\ stratum(\mathbb{P}) = maximum(stratum(A)) \tag{4.3.33}$$

We define the program $\mathbb{P}$ at stratum k, $\mathbb{P}_{str.k}$ *as the set of clauses $Cl_i \in \mathbb{P}$ such that the stratum assigned to their heads A is lower or equal to k.*

Remark. The stratification we will use from here on is defined as follows. If A appears as head of a clause Cl_i of the form

$$\{ A \xleftarrow{(\mathsf{p},\, \mathsf{v}),\, \&_i} (\mathsf{p}',\, \mathsf{v}') \} \tag{4.3.34}$$

we assign 0 to $stratum(A, Cl_i)$. If this is not the case, then A appears as head of a clause Cl_j of the form

$$\{ A \xleftarrow{(\mathsf{p},\, \mathsf{v}),\, \&_i} @_j(D_1, \ldots, D_k, \ldots, D_n) \text{ if } COND(A) \} \tag{4.3.35}$$

and then the value for $stratum(A,\ Cl_j)$ will be the maximum of $stratum(D_i)$ plus one. For a set of clauses $\{ Cl_1, \ldots, Cl_n \}$ in which the atom A appears as head, the stratum of A will be defined as

$$stratum(A) = maximum \begin{matrix} i = n \\ i = 1 \end{matrix} (\ stratum(A,\ Cl_i)\ ) \tag{4.3.36}$$

Definition 4.3.12 (Declarative semantics)**.** *Let $\mathbb{P}$ be a well-defined fuzzy logic program. The* interpretation chosen as the declarative semantics of $\mathbb{P}$ *is defined*

recursively, starting from the stratum zero and ending at stratum $stratum(\mathbb{P})$. *For each atom* $A \in \mathbb{HB}$

$$I_{str.0}(A) \doteq \bigsqcap_{I \Vdash \mathbb{P}} (A, (\mathsf{p}, \hat{\ } \mathsf{v})) \;\Big|\; \begin{array}{l} (A, (\mathsf{p}, \hat{\ } \mathsf{v})) \in I \\ \wedge \\ stratum(A) = 0 \end{array} \tag{4.3.37}$$

$$I_{str.k}(A) \doteq \; \bot \; if \; stratum(A) > k \tag{4.3.38}$$

$$I_{str.k}(A) \doteq \bigsqcap_{*} (A, (\mathsf{p}, \hat{\ } \mathsf{v})) \;\Big|\; \begin{array}{l} (A, (\mathsf{p}, \hat{\ } \mathsf{v})) \in I \\ \wedge \\ stratum(A) = k \end{array} \bigsqcup I_{str.(k-1)}(A) \tag{4.3.39}$$

where $*$ *is*

$$I \Vdash \mathbb{P} \;\Big|\; \begin{array}{l} \forall A' \in \mathbb{HB}.\, (stratum(A') < k) \implies \\ \qquad I_{str.(k-1)}(A') = I(A') \end{array}$$

Basically, the interpretation chosen must have at each strata the minimum interpretation value for each atom A *in that strata, starting from stratum zero. The interpretations not having this minimum value for the atoms at any stratum lower than the current one are not considered for computing the minimum of the current strata.*

Remark. The necessity to compute the declarative semantics in the way described in Def. 4.3.12 comes from the possibility to have a program $\mathbb{P}$ in which the interpretation of one atom depends negatively on the interpretation of the other one. Suppose a program $\mathbb{P}$ formed by the set of rules

$$A \leftarrow \neg(B) \tag{4.3.40}$$
$$B \leftarrow (1,\ 0.1) \tag{4.3.41}$$

and assume that the definition of the negation operator is

$$Output \;=\; 1 \;-\; Input \tag{4.3.42}$$

then we can take the following interpretations as examples of interpretations which are models of our program $\mathbb{P}$:

$$I_1 = \{ (B, (1, \hat{0}.1)), (A, (1, \hat{0}.9)) \} \tag{4.3.43}$$

$$I_2 = \{ (B, (1, \hat{0}.4)), (A, (1, \hat{0}.6)) \} \tag{4.3.44}$$

What we want to highlight with this example is that the declarative semantics of the program cannot be its least model (usually defined as the infimum of all its interpretations which are models) since the intersection of all of them is no more a model of $\mathbb{P}$:

$$I_3 = \{ (B, (1, \hat{0}.1)), (A, (1, \hat{0}.6)) \} \tag{4.3.45}$$

This is the main reason for not using the least model as the declarative semantics of our programs.

Lemma 4.3.2 (The interpretation chosen as the program declarative semantics is a model)**.**

$$I_{str.(stratum(\mathbb{P}))} \Vdash \mathbb{P} \tag{4.3.46}$$

Proof. We start calling J to $I_{str.(stratum(\mathbb{P}))}$. For stratum 0 there is no dependency on other(s) stratum(s), since it is defined as

$$J_{str.0}(A) \doteq \bigsqcap_{I \Vdash \mathbb{P}} I_{str.0}(A) \tag{4.3.47}$$

which makes

$$J_{str.0} \doteq \bigsqcap_{I \Vdash \mathbb{P}} I_{str.0} \tag{4.3.48}$$

Then, from the definition of infimum (Def. 4.3.7) we know that there must be (at least) an interpretation I, $I \Vdash \mathbb{P}$, such that $I_{str.0} = J_{str.0}$ and we know about this interpretation that it is part of I

$$I_{str.0} \subset I \tag{4.3.49}$$

and that I is a model of $\mathbb{P}$:

$$I \Vdash \mathbb{P} \tag{4.3.50}$$

so we can conclude that the stratum $stratum0$ of $I_{str.(stratum(\mathbb{P}))}$ is part of a model of $\mathbb{P}$.

Table 4.3.2: Table representing the situation in which the intersection of two models is not a model

stratum	$I_1(A)$ vs $I_2(A)$	$I_1(B)$ vs $I_2(B)$	I_{ds}
$str.0$			$I_1 \sqcap I_2$
$str.(stratum(A))$	$I_1(A) < I_2(A)$		$I_1 \sqcap I_2$
$str.(stratum(B))$	$I_1(A) < I_2(A)$	$I_1(B) > I_2(B)$	???

For stratum k, $k > 0$, we can have two models $I_1 \Vdash \mathbb{P}$ and $I_2 \Vdash \mathbb{P}$ such that $min(I_1, I_2)$, is not a model. This happens when for two atoms A and B the interpretation of B depends on the negation of the interpretation of A, so that $stratum(A) < stratum(B)$, $I_1(A) < I_2(A)$ and $I_1(B) > I_2(B)$. The table Table. 4.3.2 represents this situation. There, I_{ds} is the interpretation chosen as the declarative semantics (restricted to the corresponding strata).

Since the definition of the declarative semantics removes the interpretations causing this situations (See the condition $*$ in Eq. 4.3.39), I_2 is never taken into account for computing $I_{str.(stratum(B))}$ and we can use a proof analogous to the proof for stratum 0. □

Now we go for the operational semantics. Although the usual in the logic programming framework is characterizing the operational semantics of a program $\mathbb{P}$ by using the post-fixpoints of T_P, the T_P that we could define would be non-monotonic[13]. This means, basically, that it does not enjoy any of the interesting properties of the monotonic immediate consequences operators, so we define the operational semantics in a different way.

Definition 4.3.13. *Let $\mathbb{P}$ be a multi-adjoint logic program. Then the* operational semantics of $\mathbb{P}$ *is defined from the interpretation of an atom A at stratum k.*

$$I_{str.k}(A) \doteq \bot \ \ if \ \ stratum(A) > k$$

$$I_{str.k}(A) \doteq$$

$$sup \left\{ \begin{array}{l|l} (\mathsf{p}, \mathsf{v}) \ \hat{\&}_i \ \hat{@}_i(\ \hat{I_{str.(k)}}(D_1\sigma), \ \ldots, \ \hat{I_{str.(k)}}(D_n\sigma) \ ) & \\ \quad if & COND(A)) \ \wedge \ A = A'\sigma \ \wedge \\ & \{ \ A' \xleftarrow{(\mathsf{p}, \mathsf{v}), \ \&_i} @_j(D_1, \ldots, D_i, \ldots, D_n) \\ & \qquad if \ COND(A') \ \} \in \mathbb{P} \end{array} \right.$$

[13] For a good revision of non-monotonic formalisms and logic programming we recommend the work of Dix, Moniz Pereira and Przymusinski [DPP96] or the condensed version of Przymusinski [Prz89a].

$$\} \text{ if } stratum(A) \leq k \tag{4.3.51}$$

The interpretation of a program $\mathbb{P}$ that we call its operational semantics is the one resulting from putting together the interpretations of all the atoms in its Herbrand base $\mathbb{HB}$, at stratum $stratum(\mathbb{P})$,

$$I_{str.k} \doteq \cup_{A \in \mathbb{HB}} \{ \ I_{str.k}(A) \ \} \tag{4.3.52}$$

In order to get it we must compute the interpretation of all the atoms in the Herbrand base from stratum 0 to stratum $stratum(\mathbb{P})$.

$$\begin{array}{l} I_{str.0} \\ \ldots \\ I_{str.(stratum(\mathbb{P}))} \end{array} \tag{4.3.53}$$

Now that we have the declarative and the operational semantics we must prove that they are equivalent.

Theorem 4.3.3. *For a multi-adjoint logic program $\mathbb{P}$, the declarative semantics and the operational ones coincide.*

Proof. We use induction to prove that both semantics are equivalent. We start from stratum 0 and prove that at stratum $k+1$ both have the same interpretation for the same atom. Suppose some ground atom A.

case $k = 0$) we start by proving that they take the same value when $stratum(A) = 0$. If $k = 0$ and $stratum(A) = 0$ then the only clauses Cl_i in which A appears as head have the form $\{ \ A' \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} (\mathsf{p}',\ \mathsf{v}') \ \}$. We compute the interpretations that correspond to its declarative ($I_{str.0}(A)_{ds}$) and operational semantics ($I_{str.0}(A)_{os}$) in the following way.

$$I_{str.0}(A)_{ds} \doteq \prod_{I \ \Vdash \ \mathbb{P}} (A,\ (\mathsf{p}, \hat{}\ \mathsf{v})) \ \Big| \ (A,\ (\mathsf{p}, \hat{}\ \mathsf{v})) \ \in \ I \tag{4.3.54}$$

$$\begin{array}{l} I_{str.0}(A)_{os} \doteq \\ \left| \begin{array}{l} sup \ \{ \ (\mathsf{p}, \hat{}\ \mathsf{v}) \ \hat{\&}_i \ (\mathsf{p}',\ \mathsf{v}') \\ \qquad \text{if } \left| \begin{array}{l} COND(A)) \ \wedge \ A \ = \ A'\sigma \ \wedge \\ \{ \ A' \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} (\mathsf{p}',\ \mathsf{v}') \\ \qquad\qquad \text{if } COND(A') \ \} \ \in \ \mathbb{P} \end{array} \right. \\ \} \end{array} \right. \end{array} \tag{4.3.55}$$

Since $I_{str.0}(A)_{ds} \Vdash \mathbb{P}_{str.0}$ we know that

$$I_{str.0}(A)_{ds} \succcurlyeq_{\mathbb{KT}} (\mathsf{p}, \hat{}\ \mathsf{v}) \ \hat{\&}_i \ (\mathsf{p}', \mathsf{v}') \quad \text{if} \left| \begin{array}{l} COND(A) \ \wedge \ A = A'\sigma \ \wedge \\ \{ \ A' \xleftarrow{(\mathsf{p}, \mathsf{v}), \ \&_i} (\mathsf{p}', \mathsf{v}') \ \text{ if } \ COND(A') \ \} \in \mathbb{P} \end{array} \right. \tag{4.3.56}$$

which means that

$$I_{str.0}(A)_{ds} \succcurlyeq_{\mathbb{KT}} I_{str.0}(A)_{os} \tag{4.3.57}$$

By computing the intersection of all the interpretations which are models of $\mathbb{P}$ we are getting the minimal interpretation of $I_{str.0}(A)_{ds}$ and there must be at least one interpretation whose value for $I_{str.0}(A)$ coincides with the value $(\mathsf{p}, \hat{}\ \mathsf{v}) \ \hat{\&}_i \ (\mathsf{p}', \mathsf{v}')$ obtained by the operational semantics to determine $I_{str.0}(A)_{os}$. Since this value is the minimal interpretation considered to be a model, it will be the assigned to $I_{str.0}(A)_{ds}$. This proves that $I_{str.0}(A)_{ds} = I_{str.0}(A)_{os}$.

It is easy to see that, when $stratum(A) > 0$, $I_{str.0}(A)_{ds} = I_{str.0}(A)_{os}$, due to the fact that both formulas assign $\perp$ when $stratum(A) > k$ and $k = 0$.

case k+1) here we start by the case $stratum(A) > k+1$. As before, it is easy to see that $I_{str.k+1}(A)_{ds} = I_{str.k+1}(A)_{os}$, due to the fact that (again) both formulas assign $\perp$ to the interpretation of A when $stratum(A) > k$ and k here takes the value $k+1$.

When $stratum(A) = k+1$ we have the following formulas:

$$I_{str.k+1}(A)_{ds} \doteq \prod_{*} (A, \ (\mathsf{p}, \hat{}\ \mathsf{v})) \left| \begin{array}{l} (A, \ (\mathsf{p}, \hat{}\ \mathsf{v})) \in I \\ \wedge \\ stratum(A) = k+1 \end{array} \right.$$

where $*$ is

$$I \Vdash \mathbb{P} \left| \begin{array}{c} \forall A' \in \mathbb{HB}. \ (stratum(A') < k+1) \implies \\ I_{str.(k)}(A') = I(A') \end{array} \right. \tag{4.3.58}$$

$$
\begin{array}{l}
I_{str.k+1}(A)_{os} \doteq \\
\quad \left\{ sup \left\{ \begin{array}{l|l}
 & (\mathsf{p}, \hat{}\, \mathsf{v}) \,\hat{\&}_i \\
 & \quad \hat{@}_i(\, I_{str.\hat{(k+1)}}(D_1\sigma),\, \ldots,\, I_{str.\hat{(k+1)}}(D_n\sigma)\,)) \\
 & \quad COND(A)) \;\wedge\; A = A'\sigma \;\wedge \\
\text{if} & \{\, A' \xleftarrow{(\mathsf{p},\, \mathsf{v}),\, \&_i} @_j(D_1, \ldots, D_i, \ldots, D_n) \\
 & \qquad\qquad \text{if } COND(A')\, \} \in \mathbb{P}
\end{array} \right. \right. \\
\quad \} \text{ if } stratum(A) \leq k+1
\end{array}
\tag{4.3.59}
$$

Suppose now that for level k it has been proved that for every atom E we have $I_{str.k}(E)_{ds} = I_{str.k}(E)_{os}$[14]. This condition will help us in two ways. In the first one we use $I_{str.k}(E)$ to filter out all interpretations which, being models, have values for $I_{str.k}(E)$ different from it (condition $*$ in Eq. 4.3.58). In this way we get rid of the problematic interpretations mentioned before. In the second one it helps us to fix the values for the basic formulas in the bodies of the following formulas. The first formula comes from the definition of model. Since all the interpretations used to determine $I_{str.k+1}(A)_{ds}$ are models we know that for every substitution σ and every clause $Cl_i \in \mathbb{P}$ of the form

$$
\{\, A \xleftarrow{(\mathsf{p},\, \mathsf{v}),\, \&_i} @_j(D_1, \ldots, D_j, \ldots, D_n) \quad \text{if } COND(A)\, \},
\tag{4.3.60}
$$

it is true that

$$
I_{str.k+1}(A)_{ds} \succcurlyeq_{\mathbb{KT}} (\mathsf{p}, \hat{}\, \mathsf{v}) \,\hat{\&}_i\, \hat{@}_i(\, \hat{I}(D_1\sigma),\, \ldots,\, \hat{I}(D_n\sigma)\,) \quad \text{if } COND(A)
\tag{4.3.61}
$$

The second formula comes from Eq. 4.3.59, in which the premise $stratum(A) = k+1$ allows us to remove the last "if" condition, resulting in

$$
\begin{array}{l}
I_{str.k+1}(A)_{os} \doteq \\
\quad \left\{ sup \left\{ \begin{array}{l|l}
 & (\mathsf{p}, \hat{}\, \mathsf{v}) \,\hat{\&}_i \\
 & \quad \hat{@}_i(\, I_{str.\hat{(k+1)}}(D_1\sigma),\, \ldots,\, I_{str.\hat{(k+1)}}(D_n\sigma)\,)) \\
 & \quad COND(A)) \;\wedge\; A = A'\sigma \;\wedge \\
\text{if} & \{\, A' \xleftarrow{(\mathsf{p},\, \mathsf{v}),\, \&_i} @_j(D_1, \ldots, D_i, \ldots, D_n) \\
 & \qquad\qquad \text{if } COND(A')\, \} \in \mathbb{P}
\end{array} \right. \right.
\end{array}
\tag{4.3.62}
$$

[14] Please note that we use E instead of A. They are different atoms.

In the previous formulas the interpretations of the basic formulas in the clauses' bodies, $\hat{I}(D\sigma)$, depend on the interpretation of some atom $B \in \mathbb{HB}$ such that $stratum(B) \leq k+1$. Since $I_{str.k}(A)_{ds} = I_{str.k}(A)_{os}$ we can substitute in both of them and get, respectively,

$$\begin{aligned} I_{str.k+1}(A)_{ds} \succcurlyeq_{\mathbb{KT}} & (\mathsf{p}, \hat{\ } \mathsf{v}) \; \hat{\&}_i \\ & \hat{@}_i(\; \hat{I_{str.k}}(D_1\sigma),\; \ldots,\; \hat{I_{str.k}}(D_n\sigma)\;) \\ & \text{if } \; COND(A) \end{aligned} \tag{4.3.63}$$

$$I_{str.k+1}(A)_{os} \doteq$$

$$sup \left\{ \begin{array}{l|l} & (\mathsf{p}, \hat{\ } \mathsf{v}) \; \hat{\&}_i \\ & \quad \hat{@}_i(\; \hat{I_{str.k}}(D_1\sigma),\; \ldots,\; \hat{I_{str.k}}(D_n\sigma)\;)) \\ & COND(A)) \;\wedge\; A = A'\sigma \;\wedge \\ \text{if} & \{\; A' \xleftarrow{(\mathsf{p},\, \mathsf{v}),\, \&_i} @_j(D_1, \ldots, D_i, \ldots, D_n) \\ & \qquad \text{if } \; COND(A') \;\} \in \mathbb{P} \end{array} \right. \tag{4.3.64}$$

At this point is where the intersection in the formula in Eq. 4.3.58 and the supreme in Eq. 4.3.62 allow us to say that the declarative and the operational semantics are equivalent. On one hand computing the intersection is equivalent to determining the minimum $I_{str.k+1}(A)_{ds}$ satisfying the formula in Eq. 4.3.63. On the other one computing the supreme is equivalent to determining the maximum value

$$\begin{aligned} & (\mathsf{p}, \hat{\ } \mathsf{v}) \; \hat{\&}_i \; \hat{@}_i(\; \hat{I_{str.k}}(D_1\sigma),\; \ldots,\; \hat{I_{str.k}}(D_n\sigma)\;) \\ & \qquad \text{if } \; COND(A) \end{aligned} \tag{4.3.65}$$

We just need to need to highlight that we are computing the intersection of all the models, and there must be at least one such that

$$\begin{aligned} I_{str.k+1}(A)_{ds} \doteq & (\mathsf{p}, \hat{\ } \mathsf{v}) \; \hat{\&}_i \\ & \hat{@}_i(\; \hat{I_{str.k}}(D_1\sigma),\; \ldots,\; \hat{I_{str.k}}(D_n\sigma)\;) \\ & \text{if } \; COND(A) \end{aligned} \tag{4.3.66}$$

for some rule and some substitution (and it must satisfy Eq. 4.3.63 for the other ones). This one takes just the value of $I_{str.k+1}(A)_{os}$, obtained by Eq. 4.3.64. □

4.3.2 High level semantics

The semantics presented are sound and complete, but the syntactic constructions introduced in Subsec. 4.2 are very different from the ones used to present the declarative and operational semantics (Eqs. 4.3.10, 4.3.11 and 4.3.12). We copy them here once again and present now how each one of the syntactic constructions in Subsec. 4.2 is translated to the form of one of them.

$$A \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} @_j(D_1,\ldots,D_i,\ldots,D_n) \quad \text{if } COND(A) \tag{4.3.67}$$

$$A \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} D \text{ if } COND(A) \tag{4.3.68}$$

$$A \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} (\mathsf{p}',\ \mathsf{v}') \text{ if } COND(A) \tag{4.3.69}$$

We start by the construction used to define the virtual database table,

$$define\_database(pT/pA, [(pN, pT')]). \tag{4.3.70}$$

This structure has no translation into the syntactic one used to give semantics to our configuration file. The reason is that it does not define any fuzzy predicate, but the structure of the database table that we can use to store information about the individuals or subjects of our searches. Nevertheless, RFuzzy generates from it multiple predicates that simplify the programmer life when accessing the information stored in the database table, as explained in Subsec. 4.2.

The construction used to define the situation in which we define a truth value for some individuals in the database,

$$fPredName(pT) :\sim value(TV) \tag{4.3.71}$$
$$if(\ pN(pT)\ \ comp\ \ value) \tag{4.3.72}$$
$$with\_credibility(credOp,\ credVal) \tag{4.3.73}$$
$$only\_for\_user\ 'UserName' \tag{4.3.74}$$

is translated into

$$fPredName(Individual) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} (1,\ TV) \quad \text{if } COND \tag{4.3.75}$$

where the *by default*[15] values for p, v, $\&_i$ and $COND$ are 0.8, 1, *prod* (product) and *true*. The way we increase the priority of the rule when Eq. 4.3.72 appears

[15] We have different values for the variables p, v, $\&_i$ and $COND$ when the tails constructions in Eqs. 4.3.72, 4.3.73 and 4.3.74 do not appear as tails of the main clause, and when they appear. In the first case we call the values the *by default* ones.

as tail of the main clause is by adding 0.05 to p and, when Eq. 4.3.74 appears, by adding 0.1 to p. So, if both appear p gets the value 0.95. If the main clause has as tail the one in Eq. 4.3.72 the value for $COND$ is replaced by

$$COND \wedge pN(Individual,\ Value) \wedge comp(Value,\ value) \tag{4.3.76}$$

where $COND$ in Eq. 4.3.76 is the existing condition before the replacement. If Eq. 4.3.74 appears as tail $COND$ is replaced by

$$COND\ \wedge\ currentUser(Me)\ \wedge\ Me\ =\ 'UserName' \tag{4.3.77}$$

The occurrence of the tail in Eq. 4.3.73 does not change the value of the variables $COND$ or p, but the *by default* values for v and $\&_i$. The values they take in case it is used are the ones given to the variables *credVal* and *credOp* in Eq. 4.3.73.

The construction used to define fuzzifications,

$$fPredName(pT) :\sim function(\ pN(\ pT\ ),\ [\ (valIn,\ valOut)\ ]). \tag{4.3.78}$$

is translated into a set of clauses, one for each piece the programmer has defined for the piecewise function. Each one of the resulting clauses has the form

$$\begin{aligned} &fPredName(Individual) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} \\ &\quad (1,\quad pN(Individual)\ *\ \frac{(valOut\_2 - valOut\_1)}{(valIn\_2 - valIn\_1)}) \\ &\quad \text{if}\quad COND \end{aligned} \tag{4.3.79}$$

in which p, v, $\&_i$ and $COND$ take *by default* the values 0.6, 1, *prod* (product) and

$$(valIn\_1 < pN(Individual) \leq valIn\_2) \tag{4.3.80}$$

As before, this values change when the tails in Eqs. 4.3.72, 4.3.73 and 4.3.74 appear as tails of the main clause. The way in which they change is the same one as before.

The construction used to tell the computer which one is the truth value that must be assigned to a predicate when no better value can be computed,

$$fPredName(pT) :\sim defaults\_to(TV) \tag{4.3.81}$$

is translated into

$$fPredName(Individual) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} (1,\ TV)\ \text{ if }\ COND \tag{4.3.82}$$

where the *by default* value for the variables p, v, $\&_i$ and $COND$ are 0, 1, *prod* (product) and *true*. This values change when the tails in Eqs. 4.3.72, 4.3.73 and 4.3.74 appear as tails of the main clause, exactly the same way as for the previous constructions.

The constructions to define fuzzy rules,

$$fPredName(pT) :\sim rule(fPredName2(pT)) \tag{4.3.83}$$

$$\begin{aligned} fPredName(pT) :\sim rule(aggr, (&fPredName2(pT),\\ &fPredName3(pT),\\ &\ldots)) \end{aligned} \tag{4.3.84}$$

are (respectively) translated into

$$\begin{aligned} fPredName(Individual) \xleftarrow{(\mathsf{p},\, \mathsf{v}),\ \&_i}\ & fPredName2(Individual)\\ & \text{if } COND \end{aligned} \tag{4.3.85}$$

$$\begin{aligned} fPredName(Individual) \xleftarrow{(\mathsf{p},\, \mathsf{v}),\ \&_i}\ & @(\ fPredName2(Individual),\\ & fPredName3(Individual),\ \ldots)\\ & \text{if } COND \end{aligned} \tag{4.3.86}$$

where the *by default* value for the variables p, v, $\&_i$ and $COND$ are 0.4, 1, *prod* (product) and *true*. This values change when the tails in Eqs. 4.3.72, 4.3.73 and 4.3.74 appear as tails of the main clause, exactly the same way as in the previous constructions.

The construction used to define a fuzzy characteristic as a synonym of another one defined before,

$$\begin{aligned} &fPredName(pT) :\sim\\ &\quad synonym\_of(\ fPredName2(pT)\ ) \end{aligned} \tag{4.3.87}$$

is translated into

$$\begin{aligned} fPredName(Individual) \xleftarrow{(\mathsf{p},\, \mathsf{v}),\ \&_i}\ & fPredName2(Individual)\\ & \text{if } COND \end{aligned} \tag{4.3.88}$$

where the *by default* value for the variables p, v, $\&_i$ and $COND$ are 0.8, 1, *prod* and *true*. As before, this values change when the tails in Eqs. 4.3.72, 4.3.73 and 4.3.74 appear as tails of the main clause. The way in which they change is the same one as before.

The construction used to define a fuzzy characteristic as an antonym of another one defined before,

$$\begin{aligned} &fPredName(pT) :\sim \\ &\quad antonym\_of(fPredName2(pT)\ ) \end{aligned} \tag{4.3.89}$$

is translated into

$$\begin{aligned} fPredName(Individual) \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} & \\ & not(fPredName2(Individual)) \\ & \text{if } COND \end{aligned} \tag{4.3.90}$$

where the *by default* value for the variables p, v, $\&_i$ and $COND$ are 0.8, 1, *prod* and *true*. As before, this values change when the tails in Eqs. 4.3.72, 4.3.73 and 4.3.74 appear as tails of the main clause. The way in which they change is the same one as before.

The existence of the construction for defining the similarity between attributes comes from the necessity to look for individuals with a characteristic similar to the one we are looking for. The syntax exposed before,

$$similarity\_between(pT,\ pN(value1),\ pN(value2),\ TV). \tag{4.3.91}$$

is translated into

$$\begin{aligned} fuzzy\_comparator('=\sim=',\ pN,\ value1,\ Individual) & \\ \xleftarrow{(\mathsf{p},\ \mathsf{v}),\ \&_i} TV & \\ \text{if } pN(Individual,\ value2) & \end{aligned} \tag{4.3.92}$$

so that we can ask for the similarity between the characteristic entered in argument $value1$ and the value that the individual $Individual$ has for the same characteristic. The *by default* value for the variables p, v, $\&_i$ and $COND$ are 0.8, 1, *prod* and *true* and the way in which they change when the tails in Eqs. 4.3.72, 4.3.73 and 4.3.74 appear as tails of the main clause is the same one as before. The necessity to have the name of the column (or the characteristic name) used for the comparison in the arguments' list comes from the possibility to have the same value, $value1$, in different columns in the database. It allows

us to determine easily which one is the characteristic we are comparing the input value to.

As seen before, the syntax of the constructions used to define fuzzy characteristics have a translation into the syntax we use to give semantics to our programs. We summarize now the translations in two tables, Table. 4.3.3 and Table. 4.3.4. The first one shows the values for the variables when none of the tails in Eqs. 4.3.72, 4.3.73 and 4.3.74 appear as tails of a main clause, while the second one shows just the changes that their usage originates.

construction	p	v	$\&_i$	$@_j\ (B_1,\ldots,B_n)$	$COND$
fuzzy value	0.8	1	product	TV	true
synonym	0.8	1	product	TV	true
antonym	0.8	1	product	TV	true
fuzzification function	0.6	1	product	$pN(Individual) * \frac{(valOut\_2 - valOut\_1)}{(valIn\_2 - valIn\_1)}$	$valIn\_1 < pN(Individual) \leq valIn\_2$
fuzzy rule	0.4	1	product	$@_j\ (B_1,\ldots,B_n)$	true
default fuzzy value	0	1	product	TV	true

Table 4.3.3: Summary of the values given "by default" to the variables p, v, $\&_i$, $@_j\ (B_1,\ldots,B_n)$ and $COND$.

tail construction	p	v	$\&_i$	$COND$
eq. 4.2.15	p + 0.05	v	$\&_i$	$COND \wedge$ ($pN(Individual)$ $comp$ $value$)
eq. 4.2.17	p + 0.1	v	$\&_i$	$COND \wedge$ $currentUser(Me) \wedge$ $Me = 'UserName'$
eq. 4.2.16	p	$credVal$	$credOp$	$COND$

Table 4.3.4: Changes in the values given to the variables p, v, $\&_i$ and $COND$ when the tails' constructions in eqs. 4.2.15, 4.2.17, 4.2.16 are used.

Chapter 5

Real Application Cases

Although there are plenty of theoretical approaches related to fuzzy reasoning, just few of them are used by people different from the developers. It is a challenge to produce a tool that can be used by any person that is interested in modelling a real (fuzzy) problem. From the beginning, the goal of RFuzzy has been to combine expressivity and a simple syntax, thus facilitating its usage in different fields of application. In this section we show four examples of applications that are currently using RFuzzy for modelling real-world problems. The last one, FleSe (Sec. 5.4), is part of the contributions of this thesis.

5.1 Emotion Recognition

Emotion recognition is a very interesting field in modern science and technology but it is not an easy task to automate. Many researchers and engineers are working to recognise this prospective field, but the difficulty is that emotions are not clear, not crisp. In [FM09], fuzzy reasoning is used for emotion recognition.

After studying the specific characteristics of voice speech for each human emotion (speech rate, pitch average, intensity and voice quality), that paper presents a prototype that implements emotion recognition using a fuzzy model expressed in RFuzzy. The resulting prototype is simple, quite efficient, and is able to identify the emotion of a person from his/her voice speech characteristics. The studied emotions are sadness, happiness, anger, excitement and plain emotion. According to their experiments, the prototype has a 90% of success in its deductions. The tool inherited the constructivity of RFuzzy, so it can be used not only to identify emotions automatically but also to recognise the people that have an emotion through their different speeches. It analyses an emotional speech and obtains the percentage of each emotion that is detected. So, it can provide many constructive answers according to a query demand.

The prototype is an easy tool for emotion recognition that can be modified and improved by adding new rules from speech and face analysis. So, obtaining similar implementations using RFuzzy for any kind of recognition is very straightforward.

The methodology used for this application can be generalised for any deduction of information from data records. For example, emotion recognition could take data from face analysis plus speech record; deduction of age from physical data, deduction of interest in buying from shopping records, etc. The general methodology is represented in Figure 5.1.1.

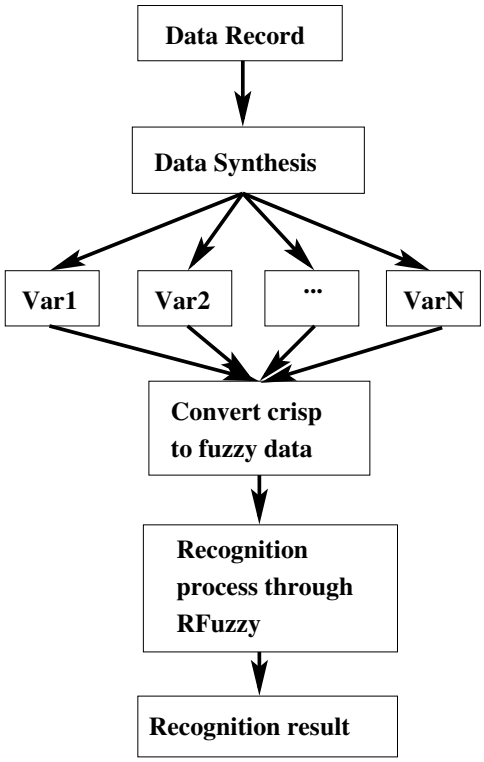


Figure 5.1.1: Methodology of data recognition.

The work of Farooque and Muñoz-Hernández has been object of interest of two Marketing companies and is nowadays a trending topic because of the recent appearance of emotion recognition applications for the Google Glasses, as the SHORE Google Glass app [Fra15] or the Emotient announce of a private beta for Sentiment Analysis Glassware for Google Glass [Emo14]. The first one [Emo12] was interested in including the voice emotion recognition into their proposal of facial emotion recognition. And the last one [Mar15] was interested in using it for doing an emotion recognition of the people calling

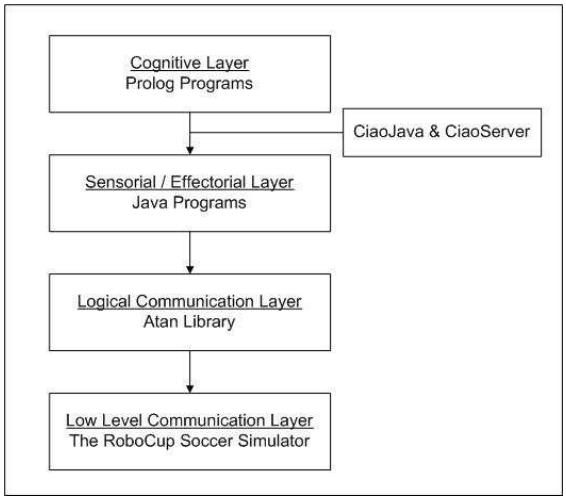


Figure 5.2.1: System Architecture for RoboCup Soccer Server.

their calling center, so they can redirect the incoming call to a more or less expert in human treatment depending on the emotion recognized.

Besides, Diego Reyes Prieto interviewed Muñoz-Hernández [Rey15] in the Nocturna program of RCN radio [RCN15] about the emotion recognition system developed. The interview is available at [PC15c].

5.2 Robocup Control Implementation

The RoboCupSoccer domain has several leagues which vary in the rules of play such as specification of players, number of players, field size and match duration. Nevertheless, each RoboCup league is a variant of a soccer league and therefore they are based on some basic rules of soccer.

In [MHW07a; MHW07b] a generalised architecture was proposed by offering flexibility to switch between leagues and programming language while maintaining Prolog as cognitive layer. The system architecture of this proposal is represented in Figure 5.2.1. Prolog is a perfect tool to design strategies for soccer players using simple rules close to human reasoning. Sometimes this reasoning needs to deal with uncertainty, fuzziness or incompleteness of the information. These issues were solved in [MHW07a; MHW07b] by using Fuzzy Prolog [GMHV04; MGP05; MV05; MV06]. Fuzzy Prolog is so expressive that the syntax of its answers is represented using constraints, so a great amount of information can be provided in a compact formula. Nevertheless, for the same reason it is a notation difficult to understand for the potential users.

The removal of the the difficulties to understand the notation came from

the combination of Prolog and RFuzzy to implement the cognitive layer in RoboCupSoccer in [MH10]. The advantage with respect to former approaches comes from the expressivity and simplicity of RFuzzy, that allows to represent all information of the problem and, at the same time, provides understandable results for the queries.

5.3 Fuzzy Granularity Control in Parallel/Distributed Computing

Any realistic approach to automatic program parallelization must take into account practical issues related to the resource usage of parallel executions, such as the overhead associated with parallel tasks creation, migration of tasks to remote processors, and communication. The aim of granularity control techniques is avoiding such overheads undermining the benefits of parallel executions. For example, sufficient conditions have been proposed to ensure that the parallel execution of some given tasks will not take longer that their corresponding sequential execution. However, when the goal is to optimize the average execution time of several runs, such conditions can be very conservative, causing a loss in parallelization oportunities. Aimed at solving this problem, T. Trigo de la Vega et al proposed in [TLM10; TLGMH10] novel conditions based on fuzzy logic. The presentation of this conditions is accompanied by an experimental assessment with real programs in which they show that such conditions select the optimal type of execution in most cases and behave much better than the conservative conditions.

5.4 FleSe (Flexible Searches in Databases)

Our starting point is assuming that it is nonsense to teach every search engine user how to translate the (almost always) fuzzy query he/she has in his/her mind into a query that a machine can understand and answer. FleSe is an user friendly interface for the search engine provided by RFuzzy, with the capability to evaluate the (possibly) fuzzy query against a database containing no fuzzy information. By using it we can answer queries like “I want a restaurant close to the center serving Mediterranean food or similar” from a database that contains a list of restaurants with the distance of each one to the center (100 meters, 1000 meters, etc) and the food they serve (Mediterranean, Spanish, Italian, etc) and some meta-information linking fuzzy and non-fuzzy information.

This work takes advantage of the improvements achieved in the theoretical

proposal of RFuzzy v.3 (Chapter. 4), as the inclusion of modifiers (even the negation modifier), similarity between predicates, similarity between attributes, etc.

5.4.1 Preliminaries

From the beginning the human being has tried to create machines with the capability to understand the real world as he does and help him to carry out tasks that he does not like to do. One of this tasks is searching in a set of individuals which ones satisfy some condition, and the automatism created for dealing with such task is called a search engine.

Most existing search engines are tools capable of performing only crisp searches. Some examples are looking for the web pages containing some words or looking for the flats whose price is not over a number we indicate. This means that we (human beings) need to convert our (usually) fuzzy queries into crisp ones, so the computer can look for the individuals satisfying it. Take, for example, the query "I want cheap flats". Since the computer does not understand the meaning of "cheap" we need to change it by "I want flats with a price lower than 200.000 €". In our humble opinion, this necessity to defuzzify the queries should be removed: existing search engines should allow the user to pose fuzzy queries. This is the task we try to fulfill here: providing a fuzzy (and non-fuzzy) search engine with an user-friendly interface. We enumerate the most important difficulties found when carrying it out in the following paragraphs.

The first one is that most of the times the information about the individuals is stored in a non-structured way (for example web pages) instead of in a structured one (databases, ontologies, or others). It is, instead of having the possibility to retrieve 200.300 € as the price of the flat number 3 (the individual id) we usually retrieve a string with the flat description (the flat is located in Madrid, near Sol's metro station, has three rooms and its price is 200.300 €). As we see it, this one is easily solved by structuring the information and we do not focus here on it.

The second one is that, when the individuals' information is stored in an structured way, it is most of the times crisp instead of fuzzy (for example the flat's price instead of the attributes "cheap", "expensive" or others). We could study the technical reasons for deciding to use a non-fuzzy database, but there is at least a non-technical one which justifies it: the inherent subjective character of fuzzy attributes. Take, for example, Victor's height: 1'81 cm. There is no problem in storing this crisp value (it is just a float number), but it is no so easy if we try to store if Victor is "tall", "very tall", "no tall at all" or any

other fuzzy value, because it might not be true for all the people retrieving the value from the database. Elsa, whose height is 1'41 cm, might consider him very tall, while Andrew, whose height is 1'75, might consider him just tall.

The third one is a drawback of assuming that the database should store crisp information: the retrieval is most of the times done in a crisp way. Suppose a database with flats and their prices. If we enter a query for retrieving the flats with a price lower than 200.000 € and there is a flat whose price is 200.300 € then it will not be retrieved, while it might be the one we are looking for (Is 300 € enough money for considering the flat too expensive?). Since we think that this should not be the case, we provide the possibility to navigate through all the results, whether they satisfy the query or not.

The fourth one is caused by the fact that one has to decide between limiting the user queries' syntax or add to the system the capability to understand queries in natural language. Regulating the queries' syntax has pros and cons. On the plus side we have the easy recognition of the query by the system. On the down side, the result of writing the query using the syntactical structures available is sometimes so complex that only a few human beings understand it. By trying to provide a user-friendly interface we fall into the necessity of recognizing the user query.

The recognition of the query has basically two parts: syntactic and semantic recognition. The first one has to deal with the lexicographic form of the set of words that compose the query and tries to find a query similar to the user's one but more commonly used. The objective with this operation is to pre-cache the answers for the most common queries and return them in less time, although sometimes it serves to remove typos in the user queries. An example of this is replacing "cars", "racs", "arcs" or "casr" by "car". The detection of words similar to one in the query is called fuzzy matching and the decision to propose one of them as the "good one" is based on statistics of usage of words and groups of words. The search engines usually ask the user if he/she wants to change the typed word(s) by this one(s). There are really interesting proposals for this procedure, as the one proposed by Carvalho in [PC13].

The semantic recognition is work still in progress and it is sometimes called "natural language processing". In the past search engines were tools used to retrieve the web pages containing the words typed in the query, but today they tend to include capabilities to understand the user query. An example is computing 4 plus 5 when the query is "4+5" or presenting a currency converter when we write "euro dollar". This is still far away from our proposal: retrieving web pages containing "fast red cars" instead of the ones containing the words "fast", "red" and "car". Our proposal can be seen as an hybrid between a search engine doing semantic recognition and one not doing it. The tool we present

has an interface intelligent enough to know what queries the search engine can answer, and it allows the user to build any of this queries with its human oriented easy to use interface.

The fifth one has to be with how do we retrieve the individuals in the database that satisfy the (fuzzy) conditions. Suppose a query like "I want a restaurant close to the center". If we assume that the computer is able to "understand" the query then it will look for a set of restaurants in the database satisfying it and return them as answer, but the database does not contain any information about "close to the center", just the "distance of a restaurant to the center". It needs to know the link between the concepts "distance to the center" and "close to the center" and how to obtain the second from the first one. As we will see, we allow to write this relations in the configuration file, so the framework can use them for translating our fuzzy queries and executing them against a non-fuzzy database.

The sixth (and last one) has to be with retrieving answers for a query in which some characteristic we ask for must be taken into account, but not as a restriction. Suppose we are looking for a bag to wear with a pair of shoes and a dress, both of brown color. If we ask for a brown color bag and the individuals in the database only have the color name (brown, red, orange, ...) the search engine will only retrieve the brown bags, not the ones maroon, sienna, saddle brown, chocolate, Peru, dark goldenrod, goldenrod, sandy brown, rosy brown, tan, burly wood, wheat, navajo white, bisque, blanched almond or cornsilk. By contrast, some of them might fit the request. An easy solution for solving this problem, that works for examples like the previous one, is replacing the attribute or characteristics by others that can be measured. For the previous example we could replace the color names assigned to each individual by the RGB (Red-Green-Blue) components of the color and translate the word "brown" in the query by its RGB components. With this change any fuzzy search engine could retrieve bags with a color similar to the one we are looking for as answers for the query, but it does not work for all the attributes or characteristics we know about. Suppose we want a Mediterranean food restaurant. This can not be split into pieces, or we do not know how to do it in such a way that it keeps its meaning. In this case we need to use a different tool: similarity.

Similarity, as we see it, is a relation between concepts that allows the search engine to get as answers individuals that are removed from the results set if we do not use it. There are some works in which the authors try to introduce similarity in fuzzy logic, as the ones of Jia-Bing Wang, Zheng-Quan Xu and Neng-Chao Wang [WXW02], Lluis Godo and Ricardo Oscar Rodriguez [GR99], Didier Dubois and Henri Prade [DP95] and F. Esteva, P. Garcia, L. Godo, E. Ruspini and L. Valverde [Est+94]. The main differences between our work and this ones

are (1) that we do not force the similarity relation to be reflexive, symmetric and transitive, i.e., an equivalence relation. As some of they mention, forcing it to be an equivalence relation is too restrictive for real-world applications. And (2) that we do not try to measure the closeness (or similarity) between two fuzzy propositions. Our work goes in the other direction: we take the similarity value computed and return the elements considered to be similar to the one we are looking for.

5.4.2 Comparison with other approaches

We include here a comparison of our work and some works we know about. We divide them in three groups, one dedicated to tools for accessing regular databases (databases with non-fuzzy information) in a fuzzy way, another dedicated to information retrieval and a final one dedicated to providing easy-to-use interfaces for providing fuzzy searches.

In the first group, tools for accessing regular databases (databases with non-fuzzy information) in a fuzzy way, we mention SQLF, presented by P. Bosc and O. Pivert [BP95], FQL, presented by Takahashi [Tak91], FIIS, presented by M. Zemankova [Zem89], FIRST, presented by D. Lucarella and R. Morara [LM91] and the tool proposed by Chen and Jong [CJ97]. A very good revision of this ones and some other proposals is the work of Herrera-Viedma and López-Herrera [HVLH10]. Although maybe a little bit outdated, we recommend the work of Leonid Tineo [Tin05] too. Most of this works focus in improving the efficiency of the existing procedures, in including new syntactic constructions, in allowing to introduce in the query that the system accepts the conversion between the fuzzy values in the original query and the non-fuzzy values needed to execute it or in improving the translation of the fuzzy query into the SQL syntax (so any regular database can answer it).

In the second group, works focusing in information retrieval, we mention the work of Ropero, Gómez, Carrasco and León [Rop+12], in which they use a logic-based term weighting procedure to create an index for answering queries, and the revision of works dedicated to information retrieval done by Zadrożny [ZN09]. Most of the works mentioned in the revision and the one by Ropero, Gómez, Carrasco and León focus in creating an index for answering queries with enough information to answer any query, by using different term weighting procedures (even logic-based ones). Once they have it the entered query can be analyzed by using natural language processing or its syntax might be reduced to a sometimes slightly complicated query syntax (it differs from some works to others).

In the last group we have the works interested in providing an easy-to-use

interface. The work of Ribeiro and Moreira [RM03] goes in this line but we think that it should not be considered a search engine project, since the natural language interface they provide only answers queries of the types (1) does X (some individual) have some fuzzy property, for example "Is it true that IBM is productive?", and (2) do an amount of elements have some fuzzy property, for example "Do most companies in central Portugal have sales_profitability?".

Our work is clearly a search engine, but we have given more weight to provide an easy to use interface allowing the user to represent any query that can be answered from the information available in the configuration file and the database than to the generation of an index or to providing an efficient mechanism for searching the database. It is clear that if we want to make it usable with large amounts of data we will need to go in one of the directions (or maybe both), but up to know we are more interested in providing the final user with all the tools that he/she might need to ask a database the fuzzy queries he/she has in mind. This is the reason why we do not even try to compare response time nor resources consumption.

5.4.3 The Architecture of the FleSe Application

FleSe, Flexible Searches in Databases, is a web interface for querying the search engine provided by RFuzzy. On one side the web interface is intelligent enough to know the queries that the search engine can answer and presents the user a form to introduce any of this queries. On the other one the search engine allows to query a database about the concepts it stores in each column or about the concepts defined in its configuration file, which can be fuzzy and non-fuzzy concepts.

A picture paints a thousand words, so we start by a painting showing the architecture of the application we have developed (Fig. 5.4.1). In Fig. 5.4.1 the computer on the top symbolizes the application user. He connects via a web browser, through the internet, to FleSe's web interface. FleSe's web interface is an application written Java that runs on a Tomcat server, on a machine with a Linux operating system. The Tomcat server listens to http(s) requests and, when it receives one for the FleSe application, it redirects it to the Servlet in the FleSe's application in charge of processing and answering it. In order to process and answer the request the Servlet evaluates what it is asking for and decides whether it needs to call the core for answering it or not. When we press the search button most of the times it calls the core, but if the query has been posed recently it will use cached results to decrease the response time.

The core is RFuzzy, a Prolog program that reads a configuration file to (1) determine to which database connect to (and how), (2) link multiple tables into

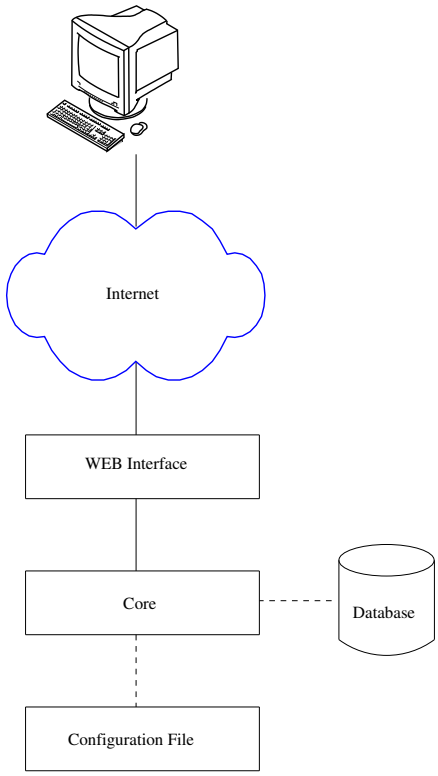


Figure 5.4.1: Application's architecture.

a virtual database table or get rid of columns with not needed information, (3) give meaning to each column in the virtual database table and (4) add concepts (fuzzy or not) to the list of concepts with which the user can build queries.

The four tasks performed by the core can be configured by modifying the configuration file, and the syntax allowed by the core for each one of them has been presented in Subsec. 4.2. Once the configuration file is chosen by the user the web interface asks the core to load it. From this moment on the web interface is fed by all the knowledge that the core extracts from the configuration file and from the database it is told to connect to. In this way the web interface is able to determine the queries that the core can answer and provides the user with a form general enough to allow him to introduce any of this queries. This produces on the final user the illusion of being working with an intelligent interface, at the times that allowing him to introduce in an easy and flexible way the query he has in mind.

5.4.4 Example of Usage of The Framework User Interface

The framework user interface gets a lot of information from the framework core, providing the final user an interface "intelligent" enough to allow him to perform any query that the framework can answer.

Suppose, for example, that we are looking for a restaurant near the city center and with a menu price under 25 €. To get the answers to our query we start choosing the configuration file (or program) that contains the information needed to connect to the database with information about restaurants and how to fuzzify it, "db_leisure" (Fig. 5.4.2).

It is worth to highlight that FleSe offers the possibility to have multiple configuration files, so developers can have multiple environments for the user queries. This is very interesting from at least two different points of view. The first, to have a configuration file under testing at the same time that we have a configuration file in production. The second, to have different environments for the queries. Maybe we have different databases and we want to perform our queries to each one of them. We could use different names to distinguish them (*car_db*_1, *car_db*_2) but we think that it has more sense to have different configuration files ans keep the original object name (*car*). Each time we upload a configuration file in FleSe it is added to the list of the available ones, becoming available for the final users. We show in Fig. 5.4.2 the available ones.

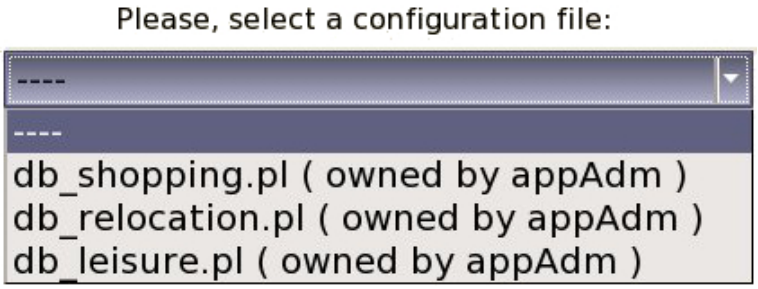


Figure 5.4.2: Select configuration file dialog.

After that we select what are we looking for (a restaurant this time, Fig. 5.4.3).

Now we can enter the restrictions with which we want the search engine to filter the results. To do it the interface presents us a combo with the fuzzy and non-fuzzy attributes available and a plus sign to the right of the combo (Fig. 5.4.4). The attributes are the names we give to the columns by using the construction in Eq. 4.2.9 and the fuzzy predicates defined by using the

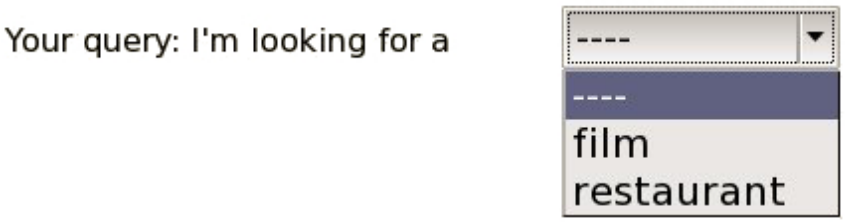


Figure 5.4.3: Choosing what we are looking for.

constructions in Eqs. 4.2.14, 4.2.22, 4.2.24, 4.2.27, 4.2.28, 4.2.30 and 4.2.32.[1] The plus sign serves to add more conditions to the query (it only has one line at the beginning) and the "show options" label can be used to change the connective used to combine the truth values from minimum to product, Łukasiewicz or any other one (Figs. 5.4.5 and 5.4.6). If the connective required is not one of the previously mentioned then it needs to be previously defined in the framework using the syntax in Eq. 4.2.37.

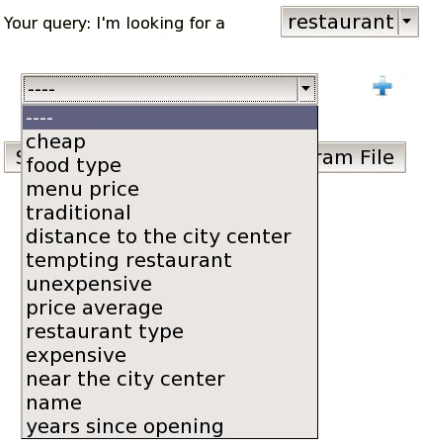


Figure 5.4.4: The available attribute(s) for writing the query.

After choosing an attribute the user interface interacts with the framework core and determines if the attribute selected is fuzzy or not. In case the predicate is fuzzy it shows to its left two combos, one for choosing (or not) a negation modifier and the other one for choosing (or not) a modifier (Fig. 5.4.7), while if it is a non-fuzzy one it shows a combo for selecting a comparison operator and, depending on the operator selected, a combo with the available values or a free text field for entering a value (Fig. 5.4.8).

[1] It is worth to remark that we can use multiple sentences to define a fuzzy predicate and it

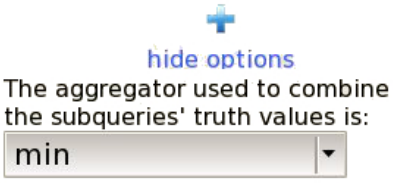


Figure 5.4.5: Available options when pressing “show options”.

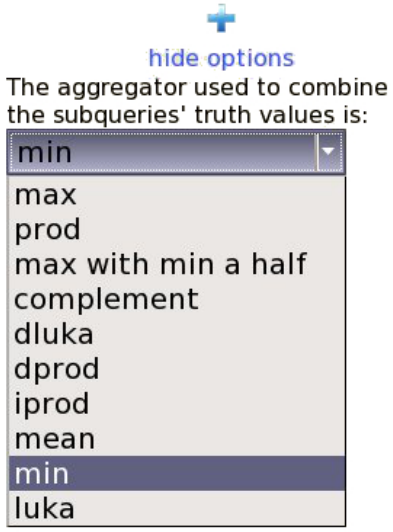


Figure 5.4.6: Available connectives to combine the subqueries results.

Figure 5.4.7: Available modifiers for the fuzzy attribute.

By using the combos and the text fields shown we can enter the query we had in mind, as can be seen in Fig. 5.4.9. After posing the query and pressing the button labelled “search” the search engine shows the query results grouped in five tabs: “10 best results”, “results over 70%”, “results over 50%”, “results

will appear only one time in the list of attributes.

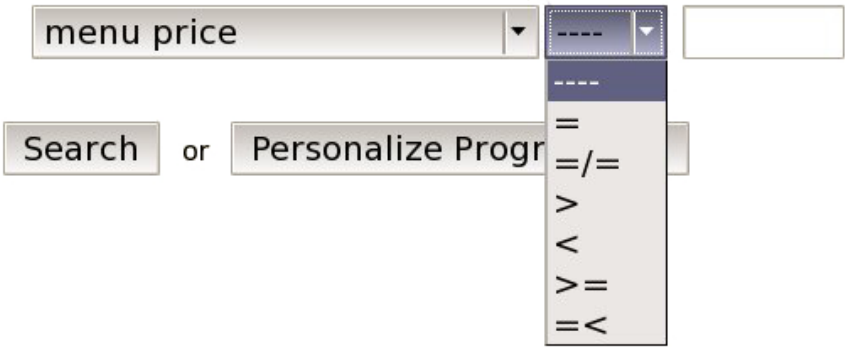


Figure 5.4.8: Available comparison operators for the non-fuzzy attribute.

over 0%" and "all results". This allows the user to navigate through all the results, even if they do not satisfy the query at all. We show in Fig. 5.4.10 the results for the query entered in Fig. 5.4.9. The data in the first column corresponds to the information in the virtual database table. The user can choose between seeing it or not.

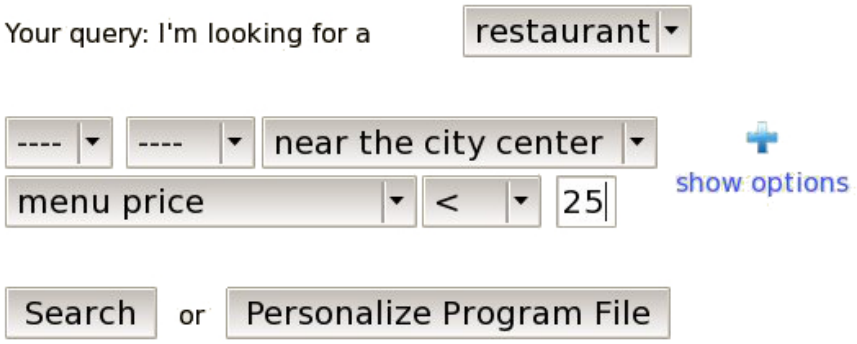


Figure 5.4.9: Query example.

10 best results | Results over 70% | Results over 50% | Results over 0% | All results

restaurant	name	restaurant type	food type	years since opening	distance to the city center	price average	menu price	Truth Value
restaurant(meson del jamon, fast food, spanish, 8, 100, 20, 15)	meson del jamon	fast food	spanish	8	100	20	15	1.00
restaurant(museo del jamon, fast food, spanish, 8, 150, 20, 15)	museo del jamon	fast food	spanish	8	150	20	15	0.95

Figure 5.4.10: Answers returned for the query example in Fig. 5.4.9

In case the fuzzification functions defined in the configuration file do not suit our expectations, we can personalize them using the button "Personalize

Program File". This mechanism allows to personalize how the framework translates the non-fuzzy attributes stored in the database into the fuzzy ones used for querying the database. When pressing the button the interface shows a pop-up window (Fig.5.4.11) in which it asks the user which fuzzy predicate he wants to personalize and his preferences for the fuzzification of the values stored in the database.

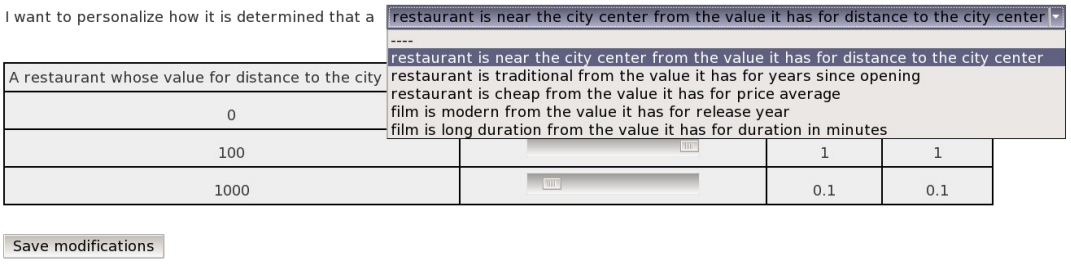


Figure 5.4.11: Selection of the fuzzy attribute the user wants to personalize and introduction of the user definition

5.5 Chapter final notes

As has been show in this chapter, RFuzzy is not just a theoretical framework. It is a framework with theoretical background (it has declarative and operational semantics) which can be used in practice to manage fuzziness.

Almost each day somebody ask us if RFuzzy could be used for implementing this or that idea they have in mind. We have been encouraged to use it in fields like the semantic web, robot control or non-crisp compilation decisions. In Sec. 6.2 we mention some of them in which we are working now.

Before this section ends we want to highlight again that the source code of RFuzzy is available for downloading at [PC15d]. The one of FleSe at [PC15b] and there is a running version of FleSe available at [PC15a]. Extra material can be found in [PC15c], as interviews, presentation slides, or usage manuals.

Chapter 6

Conclusions

It is a reality that we, as human beings, understand the world as a fuzzy entity. The cinema is far, and it is even strange to hear that you have to drive ten kilometers to go there. It is far. This is all we say. Representing it in the way we understand it (and not as a machine needs to proceed to behave and return the answers we want) was an existing necessity. We present here a framework that allows to represent relations between fuzzy and non-fuzzy concepts in a syntax very close to the human way of thinking. As far as we know, it is the first time that a work on extending the expressiveness of fuzzy languages includes together the large amount of extensions that we have included here.

The work we have presented is the result of a very fertile research. We have published our results in a workshop, ([PCMHS08]), eight conferences ([MHPCS09; PCMH11; PCMH14b; PCMH14c; PCMH14d; PCMH14e; PCSMH09; SMHPC09]. and a journal paper ([MHPCS11]) and we have sent a second journal paper that we expect to be published soon ([PCMH15]).

Besides, both the RFuzzy framework and some of the applications developed using it have had a lot of impact. An American university ([Duk15]) was trying to determine if the emotion of a CEO CFO, manager, analyst, ... could affect the stock market and they took ideas from RFuzzy for their proposal. An Spanish company ([Inn12]) was interested in using RFuzzy as we do it in FleSe (see Sec. 5.4), to represent fuzzy concepts and translate fuzzy queries into queries that can be run against a non-fuzzy database. Between the applications developed using the RFuzzy framework (we have presented some of them in Chapter. 5), some companies ([Emo12; Mar15]) showed interest for the emotion recognition application presented by Farooque and Muñoz-Hernández. More details are provided in Sec. 5.1.

The work presented here can be seen as the sum of three parts. The first one is in chapter 2 and is a serious attempt to provide an useful tool for modelling real world problems and applying fuzzy reasoning to them. It is a

very expressive tool with a simple syntax that makes it adequate for modelling real application cases. Some of its characteristics are representation of default (and conditional default) values, types and syntax for functions and (continuous and discrete) rules. This initial version of RFuzzy comes with formal declarative semantics (Section 2.2) along with a soundness and completeness result that links operational and declarative semantics.

The second part of the work (chapter 3) is in which we increased the expressiveness of the language by introducing the usage of a real number between zero and one (included) for encoding the priority. In the first part (chapter 2) RFuzzy uses three symbols for encoding it (▲, ♦ *and* ▼) and this is sometimes not enough (mainly when we have more than three rules giving answers to our query and we want to assign them more than three different priority values). Priorities are used to define which rule is the one that we will use to get results to our query, a facility whose most important use is allowing to define personalized rules. Personalized rules allow the final user to reuse existing programs while modifying the parts whose behaviour they do not want. This gets the development of fuzzy logic programs a little bit closer to the human way of thinking, where the use of priorities is a reality. Take, for example, the rule saying "if it is late, go to the bed". This rule applies when you live with your parents but, if you live on your own, you can forget it or change its priority. You assign a different priority to the rule. In this part we also provide syntax, semantics and proofs between declarative and operational semantics.

In the third part (chapter 4) we have presented the most recent version of RFuzzy, in which the expressiveness has been increased even more. One of the improvements has been including the possibility to link non-fuzzy information stored in a database with fuzzy information, offering the final user the possibility to pose fuzzy and non-fuzzy queries in almost natural language against non-fuzzy databases. The other ones can be clustered in two groups. In the first one we include the capability to increase the number of connectives and modifiers available for developing. In the second one we have the new constructions included for modelling fuzzy concepts (Sec. 4.2), which allow to:

- define fuzzy concepts from synonyms (unexpensive from cheap) or antonyms (expensive from unexpensive),
- define the similarity between two non-fuzzy characteristics (Mediterranean restaurant similar to Spanish restaurant), allowing to ask for the individuals having a characteristic similar to the one we are looking for (I want a Mediterranean restaurant or similar),
- use modifiers (i.e. quite, rather, very, very much, little, much, hardly, ...), even the negation modifier.

The syntax goes, as usually, with their informal, declarative and operational semantics and proofs of soundness and completeness between the last two. In this part the semantics are much more interesting than in the previous ones due to the inclusion of negation modifiers.

In Chapter 5 we present some of the applications developed using RFuzzy. Although the more attractive one is the Emotion Recognition application (see Sec. 5.1) the one that benefits more from the improvements done to improving the expressiveness of fuzzy languages is FleSe (see Sec. 5.4). FleSe (Flexible Searches in databases) is a web interface for querying a fuzzy search engine over conventional (non-fuzzy) databases. The main difference with respect to other approaches is that FleSe does not provide a complex syntax for querying the database nor a free text area field to enter the query. It evaluates the information in the configuration file and in the database to determine all the possible queries that a user can perform and provides a form to enter any of this queries. The queries can be fuzzy and we can even ask for the individuals in the database having some attribute with a value similar to the one we are looking for, which makes the combination of FleSe and RFuzzy a really useful tool for modelling any scenario and providing the final user with a search engine intelligent enough to help him pose any of the queries that the framework can solve.

In a nutshell, the combination of FleSe and RFuzzy

i) provides a user-friendly interface to query the search engine in almost natural language,

ii) takes advantage of the information structured in databases, but allowing the user to pose fuzzy and flexible queries,

iii) allows to define fuzzy concepts by using

- non-fuzzy information stored in databases
- other fuzzy concepts (using rules, synonyms and/or antonyms)
- fixed values written in configuration files
- default values (to solve the problem of missing information)

iv) allows to define similarity between the attributes stored in a database, allowing to ask for individuals with an attribute similar to the one we are looking for.

v) allows to define connectives (as product or minimum) and modifiers (as little, very, very much, etc), including negation modifiers (as not), and use them in the rules and in the queries,

vi) allows to personalize the definition of the relations between the crisp information in databases and the fuzzy concepts we need to perform fuzzy queries, even from the web interface.

At last, the but not least important than the previous, it is worth to highlight that the applications developed are free software. The source code of RFuzzy is available for downloading at [PC15d]. The one of FleSe at [PC15b] and there is a (beta) running version of FleSe available at [PC15a] This version of Flese allows to upload any configuration file and test the facilities it provides[1] and uses any open authentication provider (google, facebook, ...) for authentication purposes, so it does not require users to create accounts for using it. It can connect to any database and use it, being its only requisite for testing to upload and/or choose an existing configuration file written in the syntax explained in Sec. 4.2.

6.1 About the authorship of the contents

The work here presented is composed by the work of multiple people. Sometimes they collaborated with PhD. Candidate Victor Pablos Ceruelo, sometimes with Dr. Susana Muñoz Hernández and sometimes with both. In this section we make clear each one of the situations.

The works [MHPCS09; MHPCS11; PCMHS08; PCSMH09; SMHPC09] were the result of a very fertile collaboration with now Dr. Hannes Straß. In this collaboration Dr. Hannes Straß worked in the development of the semantics of RFuzzy while PhD. Candidate Victor Pablos Ceruelo worked in the syntax and its practical implementation. Nevertheless, Sec. 2.4 (a section about the link between RFuzzy semantics and the multi-adjoint semantics) corresponds to PhD. Candidate Victor Pablos Ceruelo.

In Sec.5.1 Sec.5.2 and Sec.5.3 we describe results obtained by Dr. Susana Muñoz Hernández, some times working alone and some others with some other researcher(s). Their work appears here to highlight that the work of PhD. Candidate Victor Pablos Ceruelo is much more than just a theoretical framework.

Chapter. 3, Chapter. 4 and Sec. 5.4 have been developed by the author of this thesis, under the supervision of Dr. Susana Muñoz Hernández.

[1] Please remember that the databases need to be publicly available. If not, FleSe will not be able to retrieve answers for the queries. For testing purposes you can export the database contents, copy them into a configuration file (in the proper syntax) and replace the database declaration lines by the corresponding lines.

6.2 Current Work

We are today working in different directions.

- In the inclusion of new connectives and modifiers, as the "all the more as" connective presented by Bosc and Pivert [BP11] or the "and possibly" of Bordogna and Pasi [BP94]. We are interested too in type II modifiers (see Subsec. 4.2 or Zadeh's paper [Zad72]).
- In the inclusion of an editor in FleSe for the configuration files, allowing to do it directly from the web interface. In this way we want to remove the necessity to use a Ciao Prolog [Bue+97] source code editor for that purpose.
- In the connection of FleSe to information stored in ontologies, to get not only data (as we do with databases) but relations between concepts.
- In automatic learning the fuzzy function definition from the personalizations introduced by the users in FleSe, so the ones provided by us are more close to the subset of users that use our application.
- In the improvement of the efficiency of our search engine by using some of the mechanisms revised in Subsec. 5.4.2.

Bibliography

[AMM07] J. M. Abietar, P. J. Morcillo, and G. Moreno. “Designing a Software Tool for Fuzzy Logic Programming”. In: *Computational Methods in Science and Engineering*. Ed. by G. Maroulis & T. E. Simos. Vol. 963. American Institute of Physics Conference Series. Dec. 2007, pp. 1117–1120. DOI: 10.1063/1.2835940 (cit. on p. 15).

[AC73] Philippe Roussel et Robert Pasero Alain Colmerauer Henry Kanoui. *Un système de communication homme-machine en Français (rapport de recherche)*. Tech. rep. Marseille, France: Universite Aix-Marseille II, 1973. URL: http://alain.colmerauer.free.fr/ (cit. on p. 8).

[AP95] Anastasia Analyti and Sakti Pramanik. “Reliable Semantics for Extended Logic Programs with Rule Prioritization.” In: *J. Log. Comput.* (1995), pp. 303–324 (cit. on pp. 55, 84).

[BMP95] J. F. Baldwin, T. P. Martin, and B. W. Pilsworth. *Fril- Fuzzy and Evidential Reasoning in Artificial Intelligence*. New York, NY, USA: John Wiley & Sons, Inc., 1995. ISBN: 047195523X (cit. on p. 14).

[BR01] Stefano Bistarelli and Francesca Rossi. “Semiring-based constraint logic programming: syntax and semantics”. In: *ACM Trans. Program. Lang. Syst.* 23.1 (2001), pp. 1–29. ISSN: 0164-0925. DOI: http://doi.acm.org/10.1145/383721.383725 (cit. on p. 14).

[BS08] Fernando Bobillo and Umberto Straccia. “fuzzyDL: An Expressive Fuzzy Description Logic Reasoner”. In: *2008 International Conference on Fuzzy Systems (FUZZ-08)*. IEEE Computer Society, 2008, pp. 923–930 (cit. on p. 14).

[BP94] Gloria Bordogna and Gabriella Pasi. “A fuzzy query language with a linguistic hierarchical aggregator”. In: *Proceedings of the 1994 ACM symposium on Applied computing*. SAC ’94. Phoenix,

Arizona, USA: ACM, New York (NY), USA, 1994, pp. 184–187. ISBN: 0-89791-647-6. DOI: 10.1145/326619.326693. URL: http://doi.acm.org/10.1145/326619.326693 (cit. on p. 125).

[BP12] Stefan Borgwardt and Rafael Peñaloza. "Non-Gödel negation makes unwitnessed consistency undecidable". In: *Proceedings of the 25th International Workshop on Description Logics (DL'2012)*. 2012 (cit. on p. 82).

[BP95] P. Bosc and O. Pivert. "SQLf: a relational database language for fuzzy querying". In: *Fuzzy Systems, IEEE Transactions on* 3.1 (1995), pp. 1 –17. ISSN: 1063-6706. DOI: 10.1109/91.366566 (cit. on p. 112).

[BP11] P. Bosc and O. Pivert. "On a strengthening connective for flexible database querying". In: *Fuzzy Systems (FUZZ), 2011 IEEE International Conference on*. 2011, pp. 1233–1238. DOI: 10.1109/FUZZY.2011.6007308 (cit. on p. 125).

[Bue+97] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. *The Ciao Prolog System. Reference Manual*. Tech. rep. CLIP3/97.1. System and manual at `http://www.cliplab.org/Software/Ciao/`. School of Computer Science, Technical University of Madrid (UPM), 1997 (cit. on p. 125).

[CJ97] Shyi-Ming Chen and Woei-Tzy Jong. "Fuzzy query translation for relational database systems". In: *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on* 27.4 (1997), pp. 714–721. ISSN: 1083-4419. DOI: 10.1109/3477.604117 (cit. on p. 112).

[CK04] Martine De Cock and Etienne E. Kerre. "Fuzzy modifiers based on fuzzy relations". In: *Information Sciences* 160.1–4 (2004), pp. 173 –199. ISSN: 0020-0255. DOI: http://dx.doi.org/10.1016/j.ins.2003.09.002. URL: http://www.sciencedirect.com/science/article/pii/S0020025503002974 (cit. on p. 66).

[Col70] Alain Colmerauer. *Les systèmes Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur*. Mimeo, Montréal, 1970. URL: http://alain.colmerauer.free.fr/ (cit. on p. 7).

[CR93] Alain Colmerauer and Philippe Roussel. "The birth of Prolog". In: *SIGPLAN Not.* 28 (3 1993), pp. 37–52. ISSN: 0362-1340. DOI: http://doi.acm.org/10.1145/155360.155362. URL: http://doi.acm.org/10.1145/155360.155362 (cit. on p. 11).

[DP00] Carlos Viegas Damásio and Luís Moniz Pereira. "Hybrid Probabilistic Logic Programs as Residuated Logic Programs". In: *Proceedings of the European Workshop on Logics in Artificial Intelligence*. JELIA '00. London, UK: Springer-Verlag, 2000, pp. 57–72. ISBN: 3-540-41131-3. URL: http://portal.acm.org/citation.cfm?id=646332.687616 (cit. on pp. 18, 58).

[DP01] Carlos Viegas Damásio and Luís Moniz Pereira. "Monotonic and Residuated Logic Programs". In: *Proceedings of the 6th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*. ECSQARU '01. London, UK: Springer-Verlag, 2001, pp. 748–759. ISBN: 3-540-42464-4. URL: http://portal.acm.org/citation.cfm?id=646564.696073 (cit. on pp. 18, 58).

[DST03] James P. Delgrande, Torsten Schaub, and Hans Tompits. "A framework for compiling preferences in logic programs". In: *Theory Pract. Log. Program.* 3 (2 2003), pp. 129–187. ISSN: 1471-0684. DOI: 10.1017/S1471068402001539. URL: http://portal.acm.org/citation.cfm?id=986825.986826 (cit. on pp. 55, 84).

[DPP96] Jürgen Dix, Luís Moniz Pereira, and Teodor C. Przymusinski. "Prolegomena to Logic Programming for Non-monotonic Reasoning". In: *Non-Monotonic Extensions of Logic Programming, NMELP '96, Bad Honnef, Germany, September 5-6, 1996, Selected Papers*. Ed. by Jürgen Dix, Luís Moniz Pereira, and Teodor C. Przymusinski. Vol. 1216. Lecture Notes in Computer Science. Springer, 1996, pp. 1–36. ISBN: 3-540-62843-6. DOI: 10.1007/BFb0023799. URL: http://dx.doi.org/10.1007/BFb0023799 (cit. on p. 94).

[DP95] Didier Dubois and Henri Prade. "Comparison of two fuzzy set-based logics: similarity logic and possibilistic logic". In: *Fuzzy Systems, 1995. International Joint Conference of the Fourth IEEE International Conference on Fuzzy Systems and The Second International Fuzzy Engineering Symposium., Proceedings of 1995 IEEE Int*. Vol. 3. 1995, pp. 1219–1226. DOI: 10.1109/FUZZY.1995.409838 (cit. on p. 111).

[Duk15] Duke University. *Duke University site*. [Online; accessed March 23, 2015]. 2015. URL: http://www.duke.edu/ (cit. on p. 121).

[Emo14] Emotient. *Emotient Announces Private Beta For "Sentiment Analysis" Glassware For Google Glass*. [Online; accessed March 23, 2015]. 2014. URL: http://54.148.116.96/about/press/emotient-

announces-private-beta-for-sentiment-analysis-glassware-for-google-glass/ (cit. on p. 106).

[Emo12] Emotion Explorer Lab. *Emotion Explorer Lab site*. [Online; accessed October 14, 2012]. 2012. URL: http://www.emotionexplorerlab.com/ (cit. on pp. 106, 121).

[Est+94] F. Esteva, P. Garcia, L. Godo, E. Ruspini, and L. Valverde. "On similarity logic and the generalized modus ponens". In: *Fuzzy Systems, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the Third IEEE Conference on*. 1994, 1423–1427 vol.2. DOI: 10.1109/FUZZY.1994.343609 (cit. on p. 111).

[Est+00] Francesc Esteva, Lluís Godo, Petr Hájek, and Mirko Navara. "Residuated fuzzy logics with an Involutive Negation". In: *Archive for Mathematical Logic* 39.2 (2000), pp. 103–124. DOI: 10.1007/s001530050006. eprint: http://dx.doi.org/10.1007/s001530050006. URL: http://link.springer.com/article/10.1007\%2Fs001530050006 (cit. on p. 82).

[FM09] Mahfuza Farooque and Susana Muñoz-Hernández. "Easy Fuzzy Tool for Emotion Recognition: Prototype from Voice Speech Analisys". In: *Proceeding of ICFC 2009 - First International Conference on Fuzzy Computation*. Ed. by J. Kacprzyk J. Filipe and A. Dourado. Madeira, Portugal: INSTICC Press, 2009 (cit. on p. 105).

[FM06] Tommaso Flaminio and Enrico Marchioni. "T-norm-based logics with an independent involutive negation". In: *Fuzzy Sets and Systems* 157.24 (2006), pp. 3125 –3144. ISSN: 0165-0114. DOI: http://dx.doi.org/10.1016/j.fss.2006.06.016. URL: http://www.sciencedirect.com/science/article/pii/S016501140600265X (cit. on p. 82).

[Fra15] Fraunhofer IIS. *SHORE Google Glass app*. [Online; accessed March 23, 2015]. 2015. URL: http://www.iis.fraunhofer.de/en/ff/bsy/tech/bildanalyse/shore-gesichtsdetektion.html (cit. on p. 106).

[GR99] Lluis Godo and Ricardo Oscar Rodriguez. "A Fuzzy Modal Logic for Similarity Reasoning". In: *Fuzzy Logic and Soft Computing*. Ed. by Kai-Yuan Cai Guoqing Chen Mingsheng Ying. Kluwer Academic, 1999. URL: http://publicaciones.dc.uba.ar/Publications/1999/GR99 (cit. on p. 111).

[Gol96] Benjamin Goldberg. “Functional programming languages”. In: *ACM Comput. Surv.* 28.1 (Mar. 1996), pp. 249–251. ISSN: 0360-0300. DOI: 10.1145/234313.234414. URL: http://doi.acm.org/10.1145/234313.234414 (cit. on p. 6).

[Got01] Siegfried Gottwald. *A Treatise on Many-Valued Logics*. Vol. 9. Studies in Logic and Computation. Research Studies Press, 2001 (cit. on p. 14).

[Got05] Siegfried Gottwald. “Mathematical fuzzy logic as a tool for the treatment of vague information”. In: *Information Sciences* 172.1-2 (2005), pp. 41 –71. ISSN: 0020-0255. DOI: DOI : 10 . 1016 / j . ins . 2005 . 02 . 004. URL: http : / / www . sciencedirect . com / science / article / B6V0C - 4FNN9YV - 1 / 2 / 17326222f60ffd6f67401986f00adfc6 (cit. on p. 14).

[Gre69] Cordell Green. “Theorem-proving by resolution as a basis for question-answering systems, in”. In: *Machine Intelligence, B. Meltzer and D. Michie, eds* (1969), pp. 183–205 (cit. on p. 7).

[GMHV04] S. Guadarrama, Susana Muñoz-Hernández, and C. Vaucheret. “Fuzzy Prolog: a new approach using soft constraints propagation”. In: *Fuzzy Sets and Systems (FSS)* 144.1 (2004). Possibilistic Logic and Related Issues, pp. 127 –150. ISSN: 0165-0114. DOI: DOI : 10 . 1016 / j . fss . 2003 . 10 . 017. URL: http : / / www . sciencedirect . com / science / article / B6V05 - 49YH3XJ - C / 2 / 77e1ea993165ce225e7a954dc324a92e (cit. on pp. 14, 15, 107).

[HVLH10] E. Herrera-Viedma and A.G. López-Herrera. “A Review on Information Accessing Systems Based on Fuzzy Linguistic Modelling”. In: *International Journal of Computational Intelligence Systems* 3.4 (2010), pp. 420–437. DOI: 10 . 1080 / 18756891 . 2010 . 9727711. eprint: http://www.tandfonline.com/doi/pdf/10.1080/18756891.2010.9727711. URL: http://www.tandfonline.com/doi/abs/10.1080/18756891.2010.9727711 (cit. on p. 112).

[Inn12] Innovation4Information. *Innovation4Information site*. [Online; accessed October 14, 2012]. 2012. URL: http : / / www . innovation4information.com/ (cit. on p. 121).

[IK85] Mitsuru Ishizuka and Naoki Kanai. “Prolog-ELF incorporating fuzzy logic”. In: *IJCAI’85: Proceedings of the 9th international joint conference on Artificial intelligence*. Los Angeles, California: Morgan Kaufmann Publishers Inc., 1985, pp. 701–703. ISBN: 0-934613-02-8, 978-0-934-61302-6 (cit. on p. 14).

[JGM07] Bharat Jayaraman, Kannan Govindarajan, and Surya Mantha. *Logic Programming with Preferences and Constraints*. http://www.scientificcommons.org/42844415, 2007. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.33.462 (cit. on pp. 55, 84).

[Jon87] C. B. Jones. "Program specification and verification in VDM". In: *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*. Marktoberdorf, Germany: Springer-Verlag New York, Inc., 1987, pp. 149–184. ISBN: 0-387-18003-6. URL: http://dl.acm.org/citation.cfm?id=42675.42682 (cit. on p. 6).

[Jon90] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990. ISBN: 0-13-880733-7 (cit. on p. 6).

[JMP05] Pascual Julián, Ginés Moreno, and Jaime Penabad. "On fuzzy unfolding: A multi-adjoint approach". In: *Fuzzy Sets and Systems* 154.1 (2005), pp. 16 –33. ISSN: 0165-0114. DOI: DOI : 10 . 1016 / j . fss . 2005 . 03 . 013. URL: http : / / www . sciencedirect . com / science / article / B6V05 - 4G1G93S - 4 / 2 / 15299df2b9433e8cf410a0c48a97f249 (cit. on pp. 14, 46).

[Jul+09] Pascual Julián, Jesús Medina, Ginés Moreno, and Manuel Ojeda-Aciego. "Thresholded Tabulation in a Fuzzy Logic Setting". In: *Electronic Notes in Theoretical Computer Science* 248.0 (2009). Proceedings of the Eighth Spanish Conference on Programming and Computer Languages (PROLE 2008), pp. 115 –130. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2009.07.063. URL: http://www.sciencedirect.com/science/article/pii/S1571066109002862 (cit. on p. 83).

[Jul+11] Pascual Julián, Jesús Medina, Pedro J. Morcillo, Ginés Moreno, and Manuel Ojeda-Aciego. "A Static Preprocess for Improving Fuzzy Thresholded Tabulation." In: *IWANN (2)*. Ed. by Joan Cabestany, Ignacio Rojas, and Gonzalo Joya. Vol. 6692. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 429–436 (cit. on p. 83).

[KK94] F. Klawonn and R. Kruse. "A Łukasiewicz logic based Prolog". In: *Mathware & Soft Computing* 1.1 (1994), pp. 5–29. URL: citeseer.nj.nec.com/227289.html (cit. on p. 14).

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. Bibl. Matematica. New York / Amsterdam: D. Van Nostrand / North-Holland, 1952 (cit. on pp. 39, 40, 64).

[KMP00] Erich Peter Klement, Radko Mesiar, and Endre Pap. *Triangular Norms (Volume 8 Trends in Logic)*. Kluwer Academic Publishers. 2000 (cit. on p. 15).

[Kna28] B. Knaster. "Un théorème sur les fonctions d'ensembles." French. In: *Annales Soc. Polonaise* 6 (1928), pp. 133–134 (cit. on pp. 39, 40, 64).

[Kow74a] R. Kowalski. *Logic for Problem Solving*. DCL memo. Department of Computational Logic, Edinburgh University, 1974. URL: http://books.google.es/books?id=f0I-PgAACAAJ (cit. on p. 7).

[Kow79] Robert Kowalski. "Algorithm = logic + control". In: *Comunication of the ACM* 22 (7 1979), pp. 424–436. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/359131.359136. URL: http://doi.acm.org/10.1145/359131.359136 (cit. on p. 9).

[Kow74b] Robert A. Kowalski. "Predicate Logic as Programming Language". In: *Proceedings IFIP Congress*. North Holland, 1974, pp. 569–574 (cit. on p. 7).

[Kow88] Robert A. Kowalski. "The early years of logic programming". In: *Commun. ACM* 31 (1 1988), pp. 38–43. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/35043.35046. URL: http://doi.acm.org/10.1145/35043.35046 (cit. on p. 11).

[KK71] Robert A. Kowalski and Donald Kuehner. "Linear Resolution with Selection Function." In: *Artificial Intelligence* 2.3/4 (1971), pp. 227–260. URL: http://dblp.uni-trier.de/db/journals/ai/ai2.html#KowalskiK71 (cit. on p. 7).

[Kun87] Kenneth Kunen. "Negation in Logic Programming". In: *Journal of Logic Programming* 4.4 (1987), pp. 289–308 (cit. on p. 9).

[LV90] Els Laenens and Dirk Vermeir. "A Fixpoint Semantics for Ordered Logic". In: *Journal of Logic and Computation* 1 (1990), pp. 159–185. DOI: 10.1093/logcom/1.2.159 (cit. on pp. 55, 84).

[Lee72] R. C. T. Lee. "Fuzzy Logic and the Resolution Principle". In: *Journal of the Association for Computing Machinery* 19.1 (1972), pp. 119–129 (cit. on p. 14).

[LL90] Deyi Li and Dongbo Liu. *A fuzzy Prolog database system*. New York, NY, USA: John Wiley & Sons, Inc., 1990. ISBN: 0-471-92762-7 (cit. on p. 14).

[Llo87] John Wylie Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987. ISBN: 3-540-18199-7 (cit. on p. 11).

[LS03] Yann Loyer and Umberto Straccia. "Default Knowledge in Logic Programs with Uncertainty". In: *Logic Programming. Proceedings of the 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003*. 2916 (LNCS). Springer Berlin Heidelberg, 2003, pp. 466–480. ISBN: 978-3-540-20642-2. DOI: 10.1007/978-3-540-24599-5_32 (cit. on p. 77).

[LS05] Yann Loyer and Umberto Straccia. "Any-world assumptions in logic programming". In: *Theoretical Computer Science* 342.2-3 (Sept. 2005), pp. 351–381. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2005.04.005. URL: http://dx.doi.org/10.1016/j.tcs.2005.04.005 (cit. on p. 26).

[LM91] D. Lucarella and R. Morara. "FIRST: Fuzzy Information Retrieval SysTem". In: *Journal of Information Science* 17.2 (1991), pp. 81–91. DOI: 10.1177/016555159101700202. eprint: http://jis.sagepub.com/content/17/2/81.full.pdf+html. URL: http://jis.sagepub.com/content/17/2/81.abstract (cit. on p. 112).

[MNR97] V. Wiktor Marek, Anil Nerode, and Jeffrey B. Remmel. "Basic Forward Chaining Construction for Logic Programs". In: *Proceedings of the 4th International Symposium on Logical Foundations of Computer Science*. London, UK: Springer-Verlag, 1997, pp. 214–225. ISBN: 3-540-63045-7. URL: http://portal.acm.org/citation.cfm?id=645682.664436 (cit. on pp. 55, 84).

[Mar15] Marketing de Servicios. *Marketing de Servicios (MDS) site*. [Online; accessed March 23, 2015]. 2015. URL: http://www.marketingdeservicios.com/ (cit. on pp. 106, 121).

[Mar+09] Guillem Marpons, Julio Mariño, Manuel Carro, Ángel Herranz, Lars-Åke Fredlund, Juan José Moreno-Navarro, and Álvaro Polo. "A Coding Rule Conformance Checker Integrated into GCC". In: *ENTCS* 248 (2009), pp. 149–159. ISSN: 1571-0661. URL: http://dx.doi.org/10.1016/j.entcs.2009.07.065 (cit. on p. 6).

[MOAV01a] Jesús Medina, Manuel Ojeda-Aciego, and Peter Vojtáš. "A Completeness Theorem for Multi-Adjoint Logic Programming". In: *FUZZ-IEEE*. 2001, pp. 1031–1034 (cit. on pp. 15, 16, 58, 65).

[MOAV01b] Jesús Medina, Manuel Ojeda-Aciego, and Peter Vojtáš. "A Procedural Semantics for Multi-adjoint Logic Programming". In: *EPIA*. Ed. by Pavel Brazdil and Alípio Jorge. Vol. 2258. Lecture Notes in Computer Science. Springer, 2001, pp. 290–297. ISBN: 3-540-43030-X (cit. on pp. 15, 16, 46, 58, 65).

[MOAV01c] Jesús Medina, Manuel Ojeda-Aciego, and Peter Vojtáš. "Multi-adjoint Logic Programming with Continuous Semantics". In: *LPNMR*. Ed. by Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski. Vol. 2173. Lecture Notes in Computer Science. Springer, 2001, pp. 351–364. ISBN: 3-540-42593-4 (cit. on pp. 15–19, 44–46, 58, 65).

[MOAV02] Jesús Medina, Manuel Ojeda-Aciego, and Peter Vojtáš. "A Multi-Adjoint Approach to Similarity-Based Unification". In: *Electronic Notes in Theoretical Computer Science* 66.5 (2002). UNCL'2002, Unification in Non-Classical Logics (ICALP 2002 Satellite Workshop), pp. 70 –85. ISSN: 1571-0661. DOI: DOI:10.1016/S1571-0661(04)80515-2. URL: http://www.sciencedirect.com/science/article/B75H1-4DDWJ13-37/2/bdc92744d6ddc8e888ea314efb711107 (cit. on pp. 16, 44, 46, 58, 65).

[MOAV04] Jesús Medina, Manuel Ojeda-Aciego, and Peter Vojtáš. "Similarity-based unification: a multi-adjoint approach". In: *Fuzzy Sets and Systems* 146.1 (2004), pp. 43–62 (cit. on pp. 16, 17, 44, 58, 65).

[MO02] Jesús Medina Moreno and Manuel Ojeda-Aciego. "On First-Order Multi-Adjoint Logic Programming". In: *11th Spanish Congress on Fuzzy Logic and Technology*. 2002. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.3800 (cit. on pp. 16, 58, 60, 65).

[Mil78] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375 (cit. on p. 27).

[MM08a] Pedro J. Morcillo and Gines Moreno. "Programming with Fuzzy Logic Rules by Using the FLOPER Tool". In: *RuleML '08: Proceedings of the International Symposium on Rule Representation, Interchange and Reasoning on the Web*. Orlando, Florida: Springer-Verlag, 2008, pp. 119–126. ISBN: 978-3-540-88807-9. DOI: http://dx.doi.org/10.1007/978-3-540-88808-6_14 (cit. on pp. 14, 15, 46).

[MM08b] Pedro José Morcillo and Ginés Moreno. "FLOPER, a Fuzzy Logic Programming Environment for Research". In: *Proceedings of VIII Jornadas sobre Programación y Lenguajes (PROLE'08)*. Ed. by Fundación Universidad de Oviedo. Gijón, Spain, 2008, pp. 259–263. ISBN: 978-84-612-5819-2 (cit. on pp. vii, 14, 15, 46–48).

[Mor06] Ginés Moreno. "Building a Fuzzy Transformation System". In: *SOFSEM*. Ed. by Jirí Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, and Julius Stuller. Vol. 3831. Lecture Notes in Computer Science. Springer, 2006, pp. 409–418. ISBN: 3-540-31198-X (cit. on p. 15).

[MGP05] Susana Muñoz-Hernández and Jose Manuél Gómez-Pérez. "Solving Collaborative Fuzzy Agents Problems with CLP(FD)". In: *PADL*. Ed. by Manuel V. Hermenegildo and Daniel Cabeza. Vol. 3350. Lecture Notes in Computer Science. Springer, 2005, pp. 187–202. ISBN: 3-540-24362-3 (cit. on p. 107).

[MV05] Susana Muñoz-Hernández and Claudio Vaucheret. "Extending Prolog with Incomplete Fuzzy Information". In: *Proceedings of the 15th International Workshop on Logic Programming Environments. CoRR* abs/cs/0508091 (2005) (cit. on p. 107).

[MV06] Susana Muñoz-Hernández and Claudio Vaucheret, eds. *Default values to handel Incomplete Fuzzy Information*. Vol. 14. IEEE Computational Intelligence Society Electronic Letter, ISSN 0-7803-9489-5. IEEE, 2006, pp. 216–223 (cit. on p. 107).

[MH10] Susana Muñoz-Hernández. "Robot Soccer". In: InTech, 2010. Chap. RFuzzy: an Easy and Expressive Tool for Modelling the Cognitive Layer in RoboCupSoccer, pp. 267–284. ISBN: 978-953-307-036-0 (cit. on p. 108).

[MHPCS09] Susana Muñoz-Hernández, Víctor Pablos-Ceruelo, and Hannes Strass. "RFuzzy: An Expressive Simple Fuzzy Compiler". In: *IWANN (1)*. Ed. by Joan Cabestany, Francisco Sandoval, Alberto Prieto, and Juan M. Corchado. Vol. 5517. Lecture Notes in Computer Science. Springer, 2009, pp. 270–277. ISBN: 978-3-642-02477-1. DOI: http://dx.doi.org/10.1007/978-3-642-02478-8_34 (cit. on pp. 20, 121, 124).

[MHPCS11] Susana Muñoz-Hernández, Víctor Pablos-Ceruelo, and Hannes Strass. "RFuzzy: Syntax, Semantics and Implementation Details of a Simple and Expressive Fuzzy Tool over Prolog". In: *Information Sciences* 181.10 (2011). Special Issue on Information

Engineering Applications Based on Lattices, pp. 1951 –1970. ISSN: 0020-0255. DOI: 10.1016/j.ins.2010.07.033. URL: http://www.sciencedirect.com/science/article/B6V0C-50PJWYR-2/2/26d8ff890f0effc98aa1c12225a5fb87 (cit. on pp. 20, 65, 83, 121, 124).

[MHV06] Susana Muñoz-Hernández and Claudio Vaucheret. "Default Values to Handle Incomplete Fuzzy Information". In: *FUZZ-IEEE. 2006 IEEE International Conference on Fuzzy Systems*. IEEE, 2006, pp. 14 –21. ISBN: 0-7803-9488-7. DOI: 10.1109/FUZZY.2006.1681688 (cit. on p. 77).

[MHV07] Susana Muñoz-Hernández and Claudio Vaucheret. "Fuzzy Prolog: Default Values to Represent Missing Information". In: *Computational Intelligence Based on Lattice Theory*. Ed. by Vassilis G. Kaburlasos and Gerhard X. Ritter. Vol. 67. Studies in Computational Intelligence. Springer, 2007, pp. 287–308. ISBN: 978-3-540-72686-9 (cit. on p. 77).

[MHVG02] Susana Muñoz-Hernández, Claudio Vaucheret, and Sergio Guadarrama. "Combining Crisp and Fuzzy Logic in a Prolog Compiler". In: *Joint Conference on Declarative Programming: APPIA-GULP-PRODE 2002*. Ed. by J.J. Moreno-Navarro and J. Mariño. Madrid, Spain, 2002, pp. 23–38 (cit. on p. 15).

[MHW07a] Susana Muñoz-Hernández and Wiratna Sari Wiguna. "Fuzzy Cognitive Layer in RoboCupSoccer". In: *12th International Fuzzy Systems Association World Congress (IFSA 2007). Foundations of Fuzzy Logic and Soft Computing*. Cancún, México: Springer, 2007, pp. 635–645 (cit. on p. 107).

[MHW07b] Susana Muñoz-Hernández and Wiratna Sari Wiguna. "Fuzzy Prolog as Cognitive Layer in RoboCupSoccer". In: *IEEE Symposium on Computational Intelligence and Games (2007 IEEE Symposia Series in Computational Intelligence)*. IEEE. Honolulu, Hawaii, 2007, pp. 340–345 (cit. on p. 107).

[MO84] Alan Mycroft and Richard A. O'Keefe. "A polymorphic type system for PROLOG." In: *Artificial Intelligence* 23.3 (1984), pp. 295–307. ISSN: 0004-3702. DOI: http://dx.doi.org/10.1016/0004-3702(84)90017-1 (cit. on p. 27).

[PC15a] Víctor Pablos-Ceruelo. *FleSe framework working installation*. [Online; accessed April 24, 2015]. 2015. URL: https://moises.ls.fi.upm.es/java-apps/flese/ (cit. on pp. 119, 124).

[PC15b] Víctor Pablos-Ceruelo. *FleSe source code location*. [Online; accessed April 24, 2015]. 2015. URL: http://babel.ls.fi.upm.es/software/FleSe/ (cit. on pp. 119, 124).

[PC15c] Víctor Pablos-Ceruelo. *RFuzzy applications and more location*. [Online; accessed April 24, 2015]. 2015. URL: http://babel.ls.fi.upm.es/software/RFuzzy/applications/ (cit. on pp. 107, 119).

[PC15d] Víctor Pablos-Ceruelo. *RFuzzy source code location*. [Online; accessed April 24, 2015]. 2015. URL: http://babel.ls.fi.upm.es/software/RFuzzy/ (cit. on pp. 23, 119, 124).

[PCMH11] Víctor Pablos-Ceruelo and Susana Muñoz-Hernández. "Introducing priorities in RFuzzy: Syntax and Semantics". In: *CMMSE 2011 : Proceedings of the 11th International Conference on Mathematical Methods in Science and Engineering*. Vol. 3. Benidorm (Alicante), Spain, 2011, pp. 918–929. ISBN: 978-84-614-6167-7. URL: http://gsii.usal.es/~CMMSE/index.php?option=com_content{\&}task=view{\&}id=15{\&}Itemid=16 (cit. on pp. 20, 65, 83–85, 121).

[PCMH14a] Víctor Pablos-Ceruelo and Susana Muñoz-Hernández. "A Framework for Modelling Real-World Knowledge Capable of Obtaining Answers to Fuzzy and Flexible Searches". In: *Computational Intelligence - Revised and Selected Papers of the International Joint Conference IJCCI 2013 held in Vilamoura, Portugal, August 2013*. Springer Verlag – accepted, 2014, to appear (cit. on p. 20).

[PCMH14b] Víctor Pablos-Ceruelo and Susana Muñoz-Hernández. "Enriching Traditional Databases with Fuzzy Definitions to Allow Flexible and Expressive Searches". In: *Proceedings of the International Conference on Fuzzy Computation Theory and Applications*. Ed. by António Dourado, José M. Cadenas, and Joaquim Filipe. Rome, Italy, 2014, pp. 111–118. ISBN: 978-989-758-053-6. DOI: http://dx.doi.org/10.5220/0005074101110118 (cit. on pp. 20, 121).

[PCMH14c] Víctor Pablos-Ceruelo and Susana Muñoz-Hernández. "FleSe: A Tool for Posing Flexible and Expressive (Fuzzy) Queries to a Regular Database". In: *Distributed Computing and Artificial Intelligence, 11th International Conference*. Ed. by Sigeru Omatu, Hugues Bersini, Juan M. Corchado Rodríguez, Sara Rodríguez, Pawel Pawlewski, and Edgardo Bucciarelli. Vol. 290. Advances in Intelligent Systems and Computing. DCAI 2014. Salamanca, Spain: Springer, 2014, pp. 157–164. ISBN: 978-3-319-07592-1.

URL: http://dx.doi.org/10.1007/978-3-319-07593-8_20 (cit. on pp. 20, 121).

[PCMH14d] Víctor Pablos-Ceruelo and Susana Muñoz-Hernández. "Introducing Similarity Relations in a Framework for Modelling Real-World Fuzzy Knowledge". In: *IPMU (3)*. Ed. by Anne Laurent, Olivier Strauss, Bernadette Bouchon-Meunier, and Ronald R. Yager. Vol. 444. Communications in Computer and Information Science. online isbn: 978-3-319-08852-5, series issn: 1865-0929. Springer International Publishing, 2014, pp. 51–60. ISBN: 978-3-319-08851-8. DOI: 10.1007/978-3-319-08852-5_6 (cit. on pp. 20, 121).

[PCMH14e] Víctor Pablos-Ceruelo and Susana Muñoz-Hernández. "On modelling real-world knowledge to get answers to fuzzy and flexible searches without human intervention". In: *Fuzzy Systems (FUZZ-IEEE), 2014 IEEE International Conference on*. Beijing, China, 2014, pp. 2329–2336. ISBN: 978-1-4799-2073-0. DOI: 10.1109/FUZZ-IEEE.2014.6891723 (cit. on pp. 20, 121).

[PCMH15] Víctor Pablos-Ceruelo and Susana Muñoz-Hernández. "FleSe". In: *unknown* 0.0 (2015). to be published, pp. 0 –0 (cit. on pp. 20, 121).

[PCMHS08] Víctor Pablos-Ceruelo, Susana Muñoz-Hernández, and Hannes Strass. "RFuzzy framework". In: *18th Workshop on Logic-based methods in Programming Environments, WLPE 2008*. Ed. by Puri Arenas and Damiano Zanardini. Vol. abs/0903.2188. Udine, Italy, 2008, pp. 62–76. URL: http://arxiv.org/pdf/0903.2188v1 (cit. on pp. 20, 121, 124).

[PCSMH09] Víctor Pablos-Ceruelo, Hannes Strass, and Susana Muñoz-Hernández. "RFuzzy—A framework for multi-adjoint Fuzzy Logic Programming". In: *Fuzzy Information Processing Society, 2009. NAFIPS 2009. Annual Meeting of the North American Fuzzy Information Processing Society Annual Conference*. Cincinnati, Ohio, USA, 2009, pp. 1–6. ISBN: 978-1-4244-4575-2. DOI: 10.1109/NAFIPS.2009.5156427 (cit. on pp. 20, 121, 124).

[PGAF12] Ana M. Palacios, María José Gacto, and Jesús Alcalá-Fdez. "Mining fuzzy association rules from low-quality data". In: *Soft Comput.* 16.5 (2012), pp. 883–901 (cit. on p. 17).

[PC13] João Paulo Carvalho. "On the semantics and the use of fuzzy cognitive maps and dynamic cognitive maps in social sciences". In: *Fuzzy Sets and Systems* 214 (2013), pp. 6–19 (cit. on p. 110).

[Pav79] Jan Pavelka. "On fuzzy logic I, II and III". In: *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 25.5 (1979), pp. 45–52, 119–134, 447–464. ISSN: 0044-3050 (cit. on pp. 17, 45).

[PS87] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis (Center for the Study of Language and Information - Lecture Notes)*. 1st ed. Center for the Study of Language and Inf, June 1987. ISBN: 0937073172. URL: http://www.mtome.com/Publications/PNLA/pnla-digital.html (cit. on p. 7).

[PTC02] Ana Pradera, Enric Trillas, and Tomasa Calvo. "A general class of triangular norm-based aggregation operators: quasi-linear T-S operators". In: *International Journal of Approximate Reasoning* 30.1 (2002), pp. 57 –72. ISSN: 0888-613X. DOI: DOI:10.1016/S0888-613X(02)00064-6. URL: http://www.sciencedirect.com/science/article/B6V07-45BHK72-2/2/d67dbc30d5491e0df09600a3d771f3fc (cit. on p. 15).

[Prz89a] Teodor C. Przymusinski. "Non-Monotonic Formalisms and Logic Programming". In: *Logic Programming, Proceedings of the Sixth International Conference, Lisbon, Portugal, June 19-23, 1989*. Ed. by Giorgio Levi and Maurizio Martelli. MIT Press, 1989, pp. 655–674. ISBN: 0-262-62065-0 (cit. on p. 94).

[Prz89b] Teodor C. Przymusinski. "On the Declarative and Procedural Semantics of Logic Programs". In: *Journal of Automated Reasoning* 5.2 (1989), pp. 167–205 (cit. on p. 90).

[RCN15] RCN Radio. *Nocturna*. [Online; accessed March 23, 2015]. 2015. URL: http://www.rcnradio.com/content/nocturna-rcn (cit. on p. 107).

[Rey15] Diego Reyes Prieto. *Radio Interview to Susana Muñoz-Hernández about emotion recognition*. 2015. URL: http://babel.ls.fi.upm.es/software/rfuzzy/applications/ (cit. on p. 107).

[RM03] Rita A. Ribeiro and Ana M. Moreira. "Fuzzy query interface for a business database". In: *International Journal of Human-Computer Studies* 58.4 (2003), pp. 363 –391. ISSN: 1071-5819. DOI: 10.1016/S1071-5819(03)00010-7. URL: http://www.sciencedirect.com/science/article/pii/S1071581903000107 (cit. on p. 113).

[Rob65] J. A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *J. ACM* 12.1 (Jan. 1965), pp. 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253. URL: http://doi.acm.org/10.1145/321250.321253 (cit. on p. 7).

[Rob97] Ken Robinson. “The B Method and the B Toolkit”. In: *Algebraic Methodology and Software Technology*. Vol. 1349. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1997, pp. 576–580. DOI: 10.1007/bfb0000503. URL: http://dx.doi.org/10.1007/bfb0000503 (cit. on p. 6).

[Rop+12] Jorge Ropero, Ariel Gómez, Alejandro Carrasco, and Carlos León. “A Fuzzy Logic intelligent agent for Information Extraction: Introducing a new Fuzzy Logic-based term weighting scheme”. In: *Expert Systems with Applications* 39.4 (2012), pp. 4567 –4581. ISSN: 0957-4174. DOI: http://dx.doi.org/10.1016/j.eswa.2011.10.009. URL: http://www.sciencedirect.com/science/article/pii/S0957417411014941 (cit. on p. 112).

[Rou75] Philippe Roussel. *Prolog: Manuel de Référence et Utilisation (Technical Report)*. Marseille, France, 1975 (cit. on p. 8).

[Sch+08] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. “Towards Typed Prolog”. In: *ICLP ’08: Proceedings of the 24th International Conference on Logic Programming*. Udine, Italy: Springer-Verlag, 2008, pp. 693–697. ISBN: 978-3-540-89981-5. DOI: http://dx.doi.org/10.1007/978-3-540-89982-2_59 (cit. on p. 27).

[Sha83] Ehud Y. Shapiro. “Logic programs with uncertainties: a tool for implementing rule-based systems”. In: *IJCAI’83: Proceedings of the Eighth international joint conference on Artificial intelligence*. Karlsruhe, West Germany: Morgan Kaufmann Publishers Inc., 1983, pp. 529–532 (cit. on p. 13).

[SDM89] Z. Shen, L. Ding, and M. Mukaidono. “Fuzzy Resolution Principle”. In: *Proc. of 18th International Symposium on Multiple-valued Logic*. Vol. 5. 1989 (cit. on p. 14).

[Spi89] J. M. Spivey. *The Z notation: a reference manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989. ISBN: 0-13-983768-X (cit. on p. 6).

[SS94] Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. Cambridge, MA, USA: MIT Press, 1994. ISBN: 0-262-19338-8 (cit. on pp. 6, 11).

[SMHPC09] Hannes Strass, Susana Muñoz-Hernández, and Víctor Pablos-Ceruelo. "Operational Semantics for a Fuzzy Logic Programming System with Defaults and Constructive Answers". In: *IFSA/EUSFLAT Conf.* Ed. by João Paulo Carvalho, Didier Dubois, Uzay Kaymak, and João Miguel da Costa Sousa. 2009, pp. 1827–1832. ISBN: 978-989-95079-6-8 (cit. on pp. 20, 121, 124).

[Tak91] Y. Takahashi. "A fuzzy query language for relational databases". In: *Systems, Man and Cybernetics, IEEE Transactions on* 21.6 (1991), pp. 1576–1579. ISSN: 0018-9472. DOI: 10.1109/21.135699 (cit. on p. 112).

[Tar55] Alfred Tarski. "A Lattice-Theoretical fixpoint theorem and its applications". English. In: *Pacific Journal of Mathematics* 5 (1955), pp. 285–309. URL: http://projecteuclid.org/DPubS?service=UI{\&}version=1.0{\&}verb=Display{\&}handle=euclid.pjm/1103044538 (cit. on pp. 39, 40, 64).

[The15a] The AdaCore company. *The Ada programming language*. [Online; accessed March 23, 2015]. 2015. URL: http://www.adacore.com/adaanswers/about/ada/ (cit. on p. 5).

[The15b] The CLIP Lab. *The Ciao Prolog Development System WWW Site*. [Online; accessed March 23, 2015]. 2015. URL: http://ciaohome.org/ (cit. on p. 53).

[TB04] George Theodorakopoulos and John S. Baras. "Trust evaluation in ad-hoc networks". In: *WiSe '04: Proceedings of the 3rd ACM workshop on Wireless security*. Philadelphia, PA, USA: ACM, 2004, pp. 1–10. ISBN: 1-58113-925-X. DOI: http://doi.acm.org/10.1145/1023646.1023648 (cit. on p. 29).

[Tin05] Leonid José Tineo. *A contribution to Database Flexible Querying: Fuzzy Quantified Queries Evaluation (PhD. Thesis)*. 2005 (cit. on p. 112).

[TLM10] Teresa Trigo de la Vega, Pedro Lopez-García, and Susana Muñoz-Hernández. "A Fuzzy Approach to Resource Aware Automatic Parallelization". In: *Computational Intelligence - Revised and Selected Papers of the International Joint Conference, IJCCI 2010, Valencia, Spain, October 2010*. Ed. by Kurosh Madani, António Dourado Correia, Agostinho C. Rosa, and Joaquim Filipe. Vol. 399. Studies in Computational Intelligence. Springer, 2010, pp. 229–245. ISBN: 978-3-642-27533-3. DOI: 10.1007/978-3-642-27534-0_15. URL: http://dx.doi.org/10.1007/978-3-642-27534-0_15 (cit. on p. 108).

[TLGMH10] Teresa Trigo de la Vega, Pedro Lopez-García, and Susana Muñoz-Hernández. "Towards Fuzzy Granularity Control in Parallel/Distributed Computing". In: *IJCCI (ICFC-ICNC)*. Ed. by Joaquim Filipe and Janusz Kacprzyk. Best Student Paper Award ICFC 2010. SciTePress, 2010, pp. 43–55. ISBN: 978-989-8425-32-4 (cit. on p. 108).

[TCC95] Enric Trillas, Susana Cubillo, and Juan Luis Castro. "Conjunction and disjunction on ([0,1], $\leq$)". In: *Fuzzy Sets and Systems* 72.2 (1995), pp. 155–165. ISSN: 0165-0114. DOI: http://dx.doi.org/10.1016/0165-0114(94)00348-B (cit. on p. 15).

[VGMH02] Claudio Vaucheret, Sergio Guadarrama, and Susana Muñoz-Hernández. "Fuzzy Prolog: A Simple General Implementation Using CLP(R)". In: *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2002*. Ed. by M. Baaz and A. Voronkov. Vol. 2514. LNAI. Tbilisi, Georgia: Springer, 2002, pp. 450–463. ISBN: 3-540-00010-0 (cit. on p. 14).

[Voj01] Peter Vojtáš. "Fuzzy logic programming". In: *Fuzzy Sets and Systems* 124.3 (2001), pp. 361–370 (cit. on pp. 14, 17, 44).

[WXW02] Jia-Bing Wang, Zheng-Quan Xu, and Neng-Chao Wang. "A fuzzy logic with similarity". In: *Proceedings of the 2002 International Conference on Machine Learning and Cybernetics*. Vol. 3. 2002, pp. 1178 –1183. DOI: 10.1109/ICMLC.2002.1167386. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1167386{\&}tag=1 (cit. on p. 111).

[WZL00] Kewen Wang, Lizhu Zhou, and Fangzhen Lin. "Alternating Fixpoint Theory for Logic Programs with Priority". In: *Proceedings of the First International Conference on Computational Logic*. CL '00. London, UK: Springer-Verlag, 2000, pp. 164–178. ISBN: 3-540-67797-6. URL: http://portal.acm.org/citation.cfm?id=647482.728274 (cit. on pp. 55, 84).

[Wan+07] Xizhao Wang, Eric C.C. Tsang, Suyun Zhao, Degang Chen, and Daniel S. Yeung. "Learning fuzzy rules from fuzzy samples based on rough set technique". In: *Information Sciences* 177.20 (2007), pp. 4493 –4514. ISSN: 0020-0255. DOI: DOI:10.1016/j.ins.2007.04.010. URL: http://www.sciencedirect.com/science/article/B6V0C-4NN0WFB-2/2/47188b1f12603fcacf0e46ddb9f20904 (cit. on p. 15).

[YH10] Yingjie Yang and Chris Hinde. "A new extension of fuzzy sets using rough sets: R-fuzzy sets". In: *Information Sciences* 180.3 (2010), pp. 354 –365. ISSN: 0020-0255. DOI: DOI : 10 . 1016 / j . ins . 2009 . 10 . 004. URL: http : / / www . sciencedirect . com / science / article / B6V0C - 4XFFJSC - 2 / 2 / 1b4c9840729bace77020a1e7bcf812cc (cit. on p. 15).

[Zad72] L. A. Zadeh. "A Fuzzy-Set-Theoretic Interpretation of Linguistic Hedges". In: *Journal of Cybernetics* 2.3 (1972), pp. 4–34. DOI: 10.1080/01969727208542910. eprint: http://dx.doi.org/10.1080/01969727208542910. URL: http://dx.doi.org/10.1080/01969727208542910 (cit. on pp. 81, 125).

[Zad65] Lotfi A. Zadeh. "Fuzzy Sets". In: *Information and Control* 8.3 (1965), pp. 338–353 (cit. on pp. 11, 14).

[Zad75a] Lotfi A. Zadeh. "The concept of a linguistic variable and its application to approximate reasoning - I". In: *Information Sciences* 8.3 (1975), pp. 199–249. DOI: 10.1016/0020-0255(75)90036-5. URL: http://dx.doi.org/10.1016/0020-0255(75)90036-5 (cit. on pp. 11, 12).

[Zad75b] Lotfi A. Zadeh. "The concept of a linguistic variable and its application to approximate reasoning - II". In: *Information Sciences* 8.4 (1975), pp. 301–357. DOI: 10.1016/0020-0255(75)90046-8. URL: http://dx.doi.org/10.1016/0020-0255(75)90046-8 (cit. on pp. 11, 12).

[Zad75c] Lotfi A. Zadeh. "The concept of a linguistic variable and its application to approximate reasoning - III". In: *Information Sciences* 9.1 (1975), pp. 43–80. DOI: 10.1016/0020-0255(75)90017-1. URL: http://dx.doi.org/10.1016/0020-0255(75)90017-1 (cit. on pp. 11, 12).

[Zad08] Lotfi A. Zadeh. "Is there a need for fuzzy logic?" In: *Information Sciences* 178.13 (2008), pp. 2751–2779. ISSN: 0020-0255. DOI: DOI: 10.1016/j.ins.2008.02.012. URL: http://www.sciencedirect.com/science/article/B6V0C-4S03RJC-1/2/109dfdae2551800f42203a199f55504e (cit. on p. 11).

[ZN09] Sławomir Zadrożny and Katarzyna Nowacka. "Fuzzy information retrieval model revisited". In: *Fuzzy Sets and Systems* 160.15 (2009). Special Issue: The Application of Fuzzy Logic and Soft Computing in Information Management, pp. 2173 –2191. ISSN: 0165-0114. DOI: http://dx.doi.org/10.1016/j.fss.2009.02.

012. URL: http://www.sciencedirect.com/science/article/pii/S0165011409001080 (cit. on p. 112).

[Zem89] Maria Zemankova. “FIIS: A Fuzzy Intelligent Information System”. In: *IEEE Data Eng. Bull.* 12.2 (1989), pp. 11–20 (cit. on p. 112).