Information Sciences 181 (2011) 1951–1970

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

RFuzzy: Syntax, semantics and implementation details of a simple and expressive fuzzy tool over Prolog

Susana Munoz-Hernandez *, Víctor Pablos-Ceruelo, Hannes Strass

L-3302, Campus de Montegancedo s/n, Facultad de Informática, Universidad Politécnica de Madrid, Boadilla del Monte 28660, Spain

ARTICLE INFO

Article history:
Available online 5 August 2010

Keywords:
Fuzzy logic
Logic programming application
Knowledge representation and reasoning
Semantics
Implementation

ABSTRACT

We present the RFuzzy framework, a Prolog-based tool for representing and reasoning with fuzzy information. The advantages of our framework 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.

In this approach 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. (It can be downloaded from http://babel.ls.fi.upm.es/software/rfuzzy/.) Besides implementation details, we also discuss some actual applications using RFuzzy.

1. Introduction

Logic programming [18] has been successfully used in knowledge representation and reasoning for decades. However, a serious drawback is the lack of an easy and user-friendly way to represent vague world information. Indeed, world data is not always perceived in a crisp way. Information that we gather might be imperfect, uncertain, or fuzzy in some other way. Hence the management of uncertainty and fuzziness is very important in knowledge representation.[1]

Over time, a great quantity of frameworks for incorporating uncertainty (in the sense of fuzziness) in logic programming have been proposed. Alas, only few of them resulted in actually usable tools. Since logic programming is traditionally used in knowledge representation and reasoning, we argue (as in [40]) that it is perfectly well-suited to implement a fuzzy reasoning tool as ours.

1.1. Fuzzy approaches in logic programming

Introducing fuzzy logic into logic programming resulted in the development of several fuzzy systems over Prolog. These systems replace its inference mechanism, SLD-resolution, with a fuzzy variant that is able to handle partial truth. Most of these systems implement the fuzzy resolution introduced by Lee [16], as the Prolog-Elf system [10], the FRIL Prolog system [2] and the F-Prolog language [17]. However, there is no common method for fuzzifying Prolog, as noted in [39].

Some of these Fuzzy Prolog systems only consider fuzziness on predicates whereas other systems consider fuzzy facts or fuzzy rules. There is no agreement about which fuzzy logic should be used. Most of them use min–max logic (for modelling the conjunction and disjunction operations), other systems just use Łukasiewicz logic [12].[2]

* Corresponding author. Tel.: +34 646465267; fax: +34 913363669.
E-mail addresses: susana@fi.upm.es (S. Munoz-Hernandez), vpablos@fi.upm.es (V. Pablos-Ceruelo), hannes.strass@alumnos.upm.es (H. Strass).

[1] "Is there a need for fuzzy logic" is an issue with a long history of spirited discussions and debate, as pointed out by Zadeh [52]. A better explanation on why it is needed can be found there.

[2] A good survey on many-valued logics (Gödel logics, Łukasiewicz logic, Product logic) can be found in [6,7].

0020-0255/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2010.07.033

There is also an extension of constraint logic programming [3], which can model logics based on semiring structures. This framework can model min–max fuzzy logic, which is the only logic with semiring structure. Another theoretical model for fuzzy logic programming without negation has been proposed by Vojtáš [49], which deals with many-valued implications.

1.2. Fuzzy Prolog

One of the most promising fuzzy tools for Prolog was the "Fuzzy Prolog" system [48,5]. The most important advantages in comparison to the other approaches are:

1. A truth value is represented as a finite union of sub-intervals on [0,1]. An interval is a particular case of union of one element, and a unique truth value (a real number) is a particular case of having an interval with only one element.
2. A truth value is propagated through the rules by means of an *aggregation operator*. The definition of this *aggregation operator* is general and it subsumes conjunctive operators (triangular norms [14] like min, prod, etc.), disjunctive operators [46] (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 [38]).
3. Crisp and fuzzy reasoning are consistently combined [24].

Fuzzy Prolog adds fuzziness to a Prolog compiler using CLP $(\mathcal{R})$ instead of implementing a new fuzzy resolution 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. RFuzzy approach motivation

Over the last few years several papers have been published by Medina et al. [31,32,30] about multi-adjoint programming. They describe a theoretical model, but no means of serious implementations apart from promising prototypes [1] and recently the FLOPER tool [34,27].

The FLOPER implementation is inspired by Fuzzy Prolog [5] 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[3] 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 multi-adjoint logic (see Section 5) and moreover provides some interesting improvements with respect to FLOPER: default values, partial default values (just for a subset of data), typed predicates and useful syntactic sugar (for representing facts, rules and functions). 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.

Some previous works have been published to introduce details about the syntax [36,22] and the operational semantics [37,42] of RFuzzy. The present work combines all these theoretical and practical details and additionally provides the declarative semantics and some application examples.

We start by introducing the formal syntax of RFuzzy (Section 2) and its declarative and operational semantics (Sections 3 and 4, respectively), to arrive at the justification of the sentence "RFuzzy allows to model multi-adjoint logic" (Section 5). Afterwards we board RFuzzy implementation and usage (Section 6) and provide some real application cases (Section 7). The last section (Section 8) is in charge of presenting the conclusions of this work.

2. Syntax

We will use a signature Σ of function symbols and a set of variables V to "build" the *term universe* $\mathrm{TU}_{\Sigma,\mathrm{V}}$ (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*$\mathrm{TB}_{\Pi\Sigma,\mathrm{V}}$ (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 vari-

[3] 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" [51,50] that means rough fuzzy set.

ables. As usual, the *Herbrand universe* 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*[4]:

- conjunctions $\&_1, \&_2, \ldots, \&_k$
- disjunctions $\vee_1, \vee_2, \ldots, \vee_l$
- implications $\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m$
- aggregations $@_1, @_2, \ldots, @_n$
- 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 $\widehat{F}$, respectively.

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

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

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

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 ($\widehat{F}(x,y) = max(0, x+y-1)$), Gödel logic ($\widehat{F}(x,y) = min(x,y)$), product logic ($\widehat{F}(x,y) = x \cdot y$).
- for disjunctors: Łukasiewicz logic ($\widehat{F}(x,y) = min(1, x+y)$), Gödel logic ($\widehat{F}(x,y) = max(x,y)$), product logic ($\widehat{F}(x,y) = x \cdot y$).
- for implicators: Łukasiewicz logic ($\widehat{F}(x,y) = min(1, 1-x+y)$), Gödel logic ($\widehat{F}(x,y) = y$ if $x > y$ else 1), product logic ($\widehat{F}(x,y) = x \cdot y$).
- for aggregation operators[5]: arithmetic mean, weighted sum or a monotone function learned from data.

Definition 2.1. 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

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

where $A \in TB_{\Pi,\Sigma,V}$ is called the head, $B_1,\ldots,B_n \in TB_{\Pi,\Sigma,V}$ 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 $A \leftarrow F$ (we omit c and F_c).

A *fuzzy query* is a pair $\langle A, v\rangle$, where $A \in TB_{\Pi,\Sigma,V}$ 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:

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

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:

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

[4] 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).

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

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

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 [19]) 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 2.2. A *default value declaration* for a predicate $p \in \Pi^{(n)}$ is written as

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

where $\delta_i \in [0,1]$ for all *i*. The φ_i are first-order formulas restricted to terms from $\mathrm{TU}_{\Sigma,\{X_1,\ldots,X_n\}}$, the predicates = and $\neq$, the symbol true and the 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 $\varphi_1 = \text{true}$ 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 [43] and similar works, most of them variants of the proposal of Mycroft and O'Keefe [28] that was an adaption of the type system of Hindley and Milner [25]. These works are more oriented to input/output type checking while in RFuzzy types are used for the constructive generation of answers (see Section 6.2).

Definition 2.3. Types are built from terms $t \in \mathbb{HU}$. A assigns a type $\tau \in \mathcal{T}$ to a term $t \in \mathbb{HU}$ and is written as t: τ. A 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 ith 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, america: Continent, antarctica: Continent,
asia : Continent, europe: Continent.

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

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

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 $p\colon (\tau_1, \ldots, \tau_n) \in T$ implies $(t_i\colon \tau_i) \in T$ for $1 \leqslant i \leqslant n$. For a ground clause $A \overset{c,F_c}{\leftarrow} F(B_1, \ldots, B_m)$ we say that it is well-typed w.r.t. *T* iff *A* is well-typed whenever all B_i are well-typed for $1 \leqslant i \leqslant m$ (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 2.4. 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

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

the formulas φ_i are pairwise contradictory and $\varphi_1 \vee \cdots \vee \varphi_m$ is a tautology, i.e. exactly one default truth value applies to each element of p/n's domain.

Besides, a well-defined fuzzy logic program cannot 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.

3. 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 [45] 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 $\blacktriangle <_a \blacklozenge <_a \blacktriangledown$.

The operator $\circ : \{\blacktriangle, \blacklozenge, \blacktriangledown\} \times \{\blacktriangle, \blacklozenge, \blacktriangledown\} \to \{\blacklozenge, \blacktriangledown\}$ is then defined as:

$$x \circ y := \begin{cases} \blacktriangledown & \text{if } x = y = \blacktriangledown \\ \blacklozenge & \text{otherwise.} \end{cases}$$

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:

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

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

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

A *fuzzy Herbrand interpretation* (or short, *interpretation*) of a fuzzy logic program is a mapping $I : \mathbb{HB} \rightarrow \mathbb{T}$ that assigns truth values to ground atoms. The 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 $\perp$ 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 $zv \in \mathbb{T} \setminus \{\perp\}$, meaning for an atom not being in the set that its truth value is $\perp$.

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

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

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

iff for all valuations σ, we have[6]:

$$\text{if } I(\sigma(B_i)) = z_i v_i \succ \perp \quad \text{for all } i \text{ then}^6\ I(\sigma(A)) \succcurlyeq z' v',$$

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

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

$$I \Vdash A \leftarrow v$$

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

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

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

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 $\sigma(\varphi_j)(1 \leqslant j \leqslant m)$. Notice that, as $\delta_i \in [0,1]$ for all i (see Definition 2.2) and $zv = \blacktriangle\delta_j$, we force the truth values defined in defaults to be different from $\perp$ 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 $I \Vdash P \Leftrightarrow I \Vdash R$ and $I \Vdash D$.

Proposition 3.2. *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 $TB_{\Pi,\Sigma,V}$ we can assume that*

$$I(A) \succ \perp .$$

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 $I(A) \succcurlyeq \blacktriangledown \delta_i$, and $\blacktriangle \delta_i \succ \perp$. □

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.

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

Proposition 3.3. *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 $A' \in TB_{\Pi,\Sigma,V}$. For a clause $r \in R$ of the form*

$$A \overset{c,F_c}{\leftarrow} F(B_1,\dots,B_n) \in R \quad \text{or} \quad A \leftarrow v \in R$$

and a default declaration

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

such that for some valuations σ_1 and σ_2 we have

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

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

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

and the default declaration fulfills

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

then

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

where $z'v'$ are obtained as in Definition 3.1.

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 $A \leftarrow v$ 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:

$$\blacklozenge \succ \blacktriangle \quad \text{and} \quad \blacktriangledown \succ \blacktriangle. \qquad \square$$

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.

3.1. 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 3.4 (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*

$$I \Vdash P \quad \textit{for all } I \in \mathcal{I} \textit{ implies } \textstyle\bigsqcap_{I \in \mathcal{I}} I \Vdash P.$$

Proof. We start by defining $J = \bigsqcap_{I\in\mathcal{I}} I$. From Definition 3.1 we know that we have three cases to check:

1. Let $A \overset{c,F_c}{\leftarrow} F(B_1,\dots,B_n) \in R$ be a fuzzy clause and σ a valuation. From Proposition 3.2, 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 $\mathrm{TB}_{\Pi,\Sigma,V}$, $J(\sigma(B_i))_{i\in 1\dots n} \succ \bot$.
   As $J(\sigma(B_i))_{i\in 1,\dots,n} \succ \bot$, for J to be model of P it has to be true that

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

   where $z'_J = z_1 \circ \cdots \circ z_n$, $v'_J = \widehat{F_c}(c, \widehat{F}(v_1,\dots,v_n)))$ 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 $z'_I = z_1 \circ \cdots \circ z_n$, $v'_I = \widehat{F_c}(c, \widehat{F}(v_1,\dots,v_n)))$ and $z_i v_i = I(\sigma(B_i))$.[7]
   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

[7] 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 $z'_I v'_I$.

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

$z_i v_i = \bigsqcap_{I\in\mathcal{I}} I(\sigma(B_i))$ 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 $A \leftarrow v \in R$ 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 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

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

so $J \Vdash P$.

Definition 3.5. Let P be a well-defined fuzzy logic program. The *least model of* P is defined as $\mathrm{lm}(P) := \bigsqcap_{I \Vdash P} I$.

3.2. Least fixpoint semantics

Definition 3.6 (T_P *operator*). Let $P = (R,D,T)$ be a well-defined fuzzy logic program, I an interpretation, and recall that ground(R) denotes the set of all ground instances of clauses in R. The operator T_P is defined as follows:

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

where

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

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

where $\{A \mapsto z\ v|\text{condition}\}$ can be

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

or

$$\left\{ A \mapsto zv \;\middle|\; \begin{array}{l} A \leftarrow v \in \mathrm{ground}(R) \text{ and} \\ z = \blacktriangledown \end{array} \right\}.$$

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 $T_R(I) \succ T_D(I)$[8].

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

Theorem 3.7. *Let P be a well-defined fuzzy logic program and I an interpetation.*

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

[8] $T_R(I) \succ T_D(I)$ 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

$A \overset{c,F_c}{\leftarrow} F(B_1,\ldots,B_n) \in R$ or $A \leftarrow v \in R$.

Let $A \overset{c,F_c}{\leftarrow} F(B_1,\ldots,B_n) \in R$ 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 $\sigma(A \overset{c,F_c}{\leftarrow} F(B_1,\ldots,B_n)) \in \text{ground}(R)$.
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 $z' = z_1 \circ \cdots \circ z_n$, $v' = \widehat{F_c}(c, \widehat{F}(v_1,\ldots,v_n)))$ and $z' \in \{\blacktriangledown, \blacklozenge\}$. By definition of T_P we have $T_P(I)(A') = T_R(I)(A') = z'v'$, where $z' = z_1 \circ \cdots \circ z_n$, $v' = \widehat{F_c}(c, \widehat{F}(v_1,\ldots,v_n)))$ 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 $A \leftarrow v \in R$ 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 $\sigma(A \leftarrow v) \in \text{ground}(R)$. 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 $\texttt{default}(p(X_1,\ldots,X_n)) = [\delta_1 \text{ if } \varphi_1,\ldots,\delta_m \text{ if } \varphi_m] \in D$, 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 $\texttt{default}(p(X_1,\ldots,X_n)) = [\delta_1 \text{ if } \varphi_1,\ldots,\delta_m \text{ if } \varphi_m] \in D$, 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 $A \overset{c,F_c}{\leftarrow} F(B_1,\ldots,B_n) \in R$ or $A \leftarrow v \in R$ and there exists a σ such that $A' = \sigma(A)$ then
if r is of the form $A \overset{c,F_c}{\leftarrow} F(B_1,\ldots,B_n) \in R$: 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 $v' = \widehat{F_c}(c, \widehat{F}(v_1,\ldots,v_n)))$.
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 $\sigma(A \overset{c,F_c}{\leftarrow} F(B_1,\ldots,B_n)) \in \text{ground}(R)$ such that we can assure that $T_R(I)(A') \succcurlyeq z'v'$.
if r is of the form $A \leftarrow v \in R$: $I(A') \succcurlyeq z'v'$, where $z' = \blacktriangle$ 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 $\sigma(A \leftarrow v) \in \text{ground}(R)$ 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 $A \overset{c,F_c}{\leftarrow} F(B_1,\ldots,B_n) \in R$ or $A \leftarrow v \in R$ 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 $A \overset{c,F_c}{\leftarrow} F(B_1,\ldots,B_n) \in R$ or $A \leftarrow v \in R$ such that for some valuations $\sigma_1,\ldots,\sigma_k$ we have $A' = \sigma_l(A)$ for $1 \leqslant l \leqslant k$, then from $I \Vdash P$ we have $I(A') \succcurlyeq max_{l \in [1,\ldots,k]}(z'_l v'_l)$, where $z'_l = z_1 \circ \cdots \circ z_n, z'_l \in \{\blacktriangle, \blacklozenge\}$, $v'_l = \widehat{F_c}(c, \widehat{F}(v_1,\ldots,v_n)))$ 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 $\sqcup$:

$$T_P(I)(A') = T_R(I)(A') \sqcup T_D(I)(A') = \left(\bigsqcup_{l=1,\ldots,k} z'_l v'_l \right) \sqcup \blacktriangle\delta_j = \bigsqcup_{l=1,\ldots,k} z'_l v'_l.$$

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

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

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

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

where $\{A \mapsto z\ v | \text{condition}\}$ can be

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

or

$$\left\{ A \mapsto zv \left| \begin{array}{l} A \leftarrow v \in \mathsf{ground}(R) \text{ and} \\ z = \blacktriangledown \end{array} \right. \right\}$$

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

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 [44,15], is exactly the least fixpoint of T_P.[9]

We now prove that our T_P is continuous, so Kleene's fixpoint theorem [13] can be used to prove that the least fixed point can be reached in ω steps.

Proposition 3.9 (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 3.8). Again by using definitions of $\sqsubseteq$ and $\bigsqcup$ we obtain $\bigsqcup_{i=0}^{n} T_P(I_i) = T_P(I_n)$.

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

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* 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 3.10. *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 3.8), and the Knaster–Tarski fixpoint theorem [44,15].

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 3.9), and the Kleene fixpoint theorem [13]. □

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

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

Definition 3.11. 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 3.12. *For a well-defined fuzzy logic program P, we have*

$$\mathbf{lm}(P) = \mathbf{l}fp(P).$$

Proof

$$\begin{aligned}\mathsf{lm}(P) &= \textstyle\prod_{I \Vdash P} I \quad \text{(by definition)}\\ &= \textstyle\prod_{T_P(I) \sqsubseteq I} I \quad \text{(by Lemma (3.7)}\\ &= T_P \uparrow \omega(\bot) \qquad \text{(by the Kleene fixpoint theorem)}\\ &= \mathsf{lfp}(P) \qquad \text{(by definition).} \qquad \square\end{aligned}$$

4. 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 {▲,♦,▼} explicitly; they will be encoded into the computations.

Definition 4.1. Let Ω be a signature of connectives and W a set of variables with $W \cap V = \emptyset$.[10]

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 if e does not contain variables. A computation node is called if $e \in [0,1]$. A final computation node will be indicated as $\langle A, e\rangle$.

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.

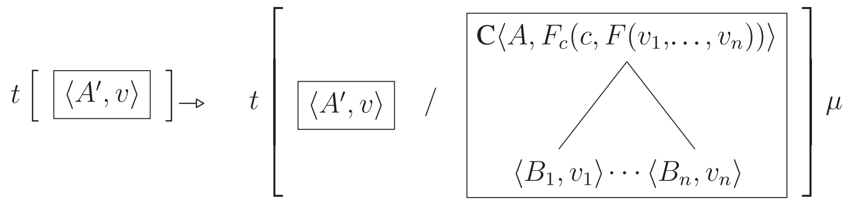
A *computation tree* is a directed acyclic graph whose nodes are computation nodes and where any pair of nodes has a unique (undirected) path connecting them. We call a computation tree ground (final) if all its nodes are ground (final). For a given computation tree t we define the

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

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 4.2 (*Transition relation*). For a given fuzzy logic program $P = (R, D, T)$, the transition relation $\rightarrow$ 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".

- **Clause:**


$t\left[\boxed{\langle A', v\rangle}\right] \rightarrow t$
$\langle A', v\rangle$
/
$\mathrm{C}\langle A, F_c(c, F(v_1,\ldots,v_n))\rangle$
$\langle B_1, v_1\rangle \cdots \langle B_n, v_n\rangle$
μ

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

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

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

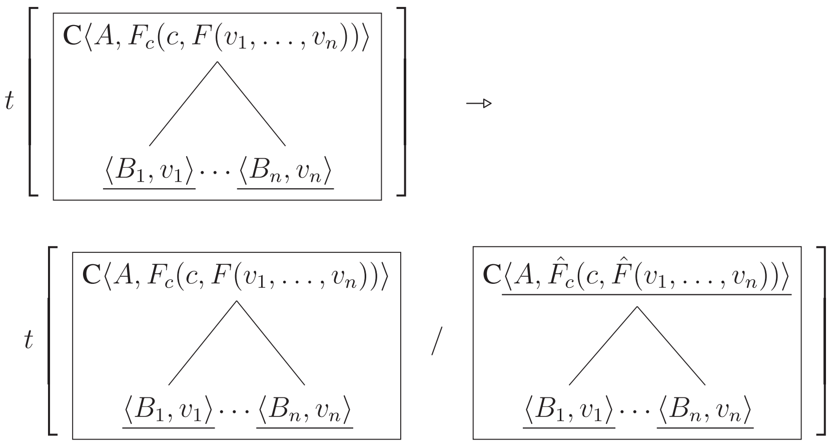
(Take a non-final leaf node and add child nodes according to a program clause; apply the most general unifier of the node atom and the clause head to all the atoms in the tree.)
Note that we immediately finalise a node when applying this rule for a fuzzy fact.

- **Default:**

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

If A does not match with any program clause head, there is a default value declaration `default`$(p(X_1,\ldots,X_n)) = [\delta_1$ if$\varphi_1,\ldots,\delta_m$ 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 $1 \leqslant j \leqslant m$ such that $\varphi_j\mu$ holds. (Apply a default value declaration to a non-final leaf node thus finalising it.)

- **Finalise:**



t
$\mathrm{C}\langle A, F_c(c, F(v_1,\ldots,v_n))\rangle$
$\underline{\langle B_1, v_1\rangle}\cdots\underline{\langle B_n, v_n\rangle}$
$\rightharpoonup$
t
$\mathrm{C}\langle A, F_c(c, F(v_1,\ldots,v_n))\rangle$
$\underline{\langle B_1, v_1\rangle}\cdots\underline{\langle B_n, v_n\rangle}$
/
$\mathrm{C}\underline{\langle A, \hat{F}_c(c, \hat{F}(v_1,\ldots,v_n))\rangle}$
$\underline{\langle B_1, v_1\rangle}\cdots\underline{\langle B_n, v_n\rangle}$

(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 if a final computation tree is created; the truth value of the instantiated query can then be read off the root node.[11] We illustrate this with an example computation.

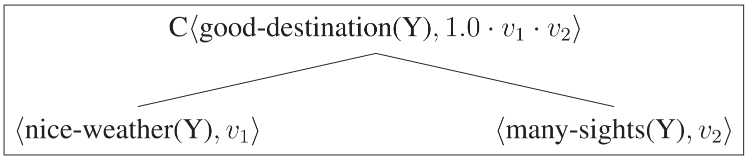
Example (continued). We start with the tree

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

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

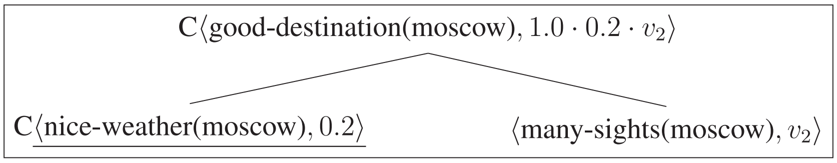
good-destination (X) $\overset{1.0,}{\leftarrow}$ (nice-weather (X), many-sights (X)),

yields

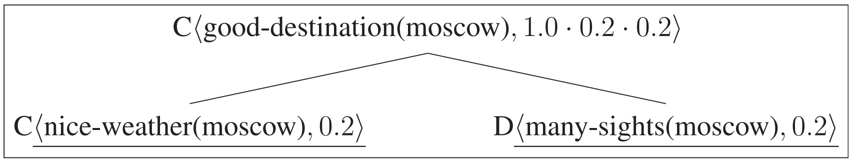


Now we apply **Clause** to the left child with nice-weather(moscow) $\leftarrow$ 0.2:

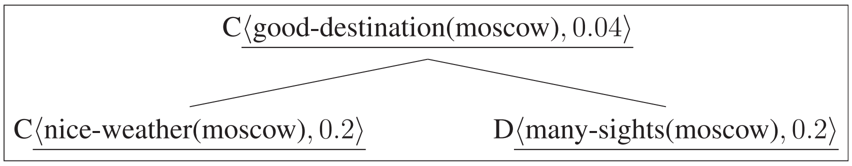
[11] Note that infinite computations lead to no answer as in the operational semantics of SLD-resolution.


C⟨good-destination(moscow), 1.0 · 0.2 · v₂⟩
C⟨nice-weather(moscow), 0.2⟩
⟨many-sights(moscow), v₂⟩

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


C⟨good-destination(moscow), 1.0 · 0.2 · 0.2⟩
C⟨nice-weather(moscow), 0.2⟩
D⟨many-sights(moscow), 0.2⟩

In the last step, we finalise the root node.


C⟨good-destination(moscow), 0.04⟩
C⟨nice-weather(moscow), 0.2⟩
D⟨many-sights(moscow), 0.2⟩

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

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

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

5. 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 [49] 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 [49] calls multi-adjoint algebra and [33] calls multi-adjoint (semi-)lattice.

During the last years there have been many theoretical publications about multi-adjoint framework semantics [49,33,31,29].

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 3 and 4 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 [31].

Definition 5.1 (*Adjoint Pair, from [31]. Firstly introduced in a logical context by Pavelka [35]*). 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 5.2 (*Multi-adjoint lattice, from [31]*). Let $<L, \preceq>$ be a lattice. A multi-adjoint lattice L is a tuple $(L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$ 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 5.3 (*Multi-adjoint Ω-algebra, from [31]*). 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 $\mathcal{L} = (L, \preceq, I(\leftarrow_1), I(\&_1), \ldots, I(\leftarrow_n), I(\&_n))$ is a multi-adjoint lattice.

There are similar definitions in [31,32,29]. They are used to build the framework of multi-adjoint logic programming in [11,26], 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:

$$\frac{(B,z), (B \to A, x)}{(A,y)},$$
$$x \preceq (y \leftarrow z) \iff (x\&z) \preceq y.$$

So, this can be used to define from *satisfaction the immediate consequences operator*, as illustrated below. Note that $\widehat{I}(B) = z, \widehat{I}(B \to A) = x, \widehat{I}(A) = y$, so $v \preceq \widehat{I}(A \leftarrow_i B) \iff v \&_i \widehat{I}(B) \preceq \widehat{I}(A)$.

Definition 5.4 (*Satisfaction, from [31]*). 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 $v \preceq \widehat{I}(A \leftarrow_i B)$.

Definition 5.5 (*Immediate consequences operator $\mathcal{T}_{\mathcal{P}}^{\mathcal{L}}$, from [31]*). Let $\mathcal{P}$ be a multi-adjoint logic program. The immediate consequences operator $\mathcal{T}_{\mathcal{P}}^{\mathcal{L}} : \mathcal{I}_{\mathcal{L}} \to \mathcal{I}_{\mathcal{L}}$, mapping interpretations to interpretations, is defined by considering

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

We provide an example from FLOPER [27] (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.

R_1 : p(X) $\leftarrow_P$ q(X,Y) $\&_G$ r(Y) with 0.8
R_2 : q(a,Y) $\leftarrow_P$ s(Y) with 0.7
R_3 : q(b,Y) $\leftarrow_L$ r(Y) with 0.8
R_4 : r(Y) $\leftarrow$ with 0.7
R_5 : s(b) $\leftarrow$ with 0.9

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 [27]).

$$\langle \mathtt{p(X)}\&_G\mathtt{r(a)}; \mathtt{id}\rangle$$
$$\to_{AS1^{R1}} \langle (0.8\&_P(\underline{q(X_1, Y_1)}\&_G r(Y_1)))\&_G r(a); \sigma_1\rangle$$
$$\to_{AS1^{R2}} \langle (0.8\&_P((0.7\&_P \underline{s(Y_2)})\&_G r(Y_1))\&_G r(a); \sigma_2\rangle$$
$$\to_{AS1^{R2}} \langle (0.8\&_P((0.7\&_P 0.9)\&_G \underline{r(b)}))\&_G r(a); \sigma_3\rangle$$
$$\to_{AS1^{R2}} \langle (0.8\&_P((0.7\&_P 0.9)\&_G 0.7))\&_G \underline{r(a)}; \sigma_4\rangle$$
$$\to_{AS1^{R2}} \langle (0.8\&_P((0.7\&_P 0.9)\&_G 0.7))\&_G 0.7; \sigma_5\rangle$$

where $\sigma_1 = \{X/X_1\}$, $\sigma_2 = \{X/a, X_1/a, Y_1/Y_2\}$, $\sigma_3 = \{X/a, X_1/a, Y_1/b, Y_2/b\}$, $\sigma_4 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b\}$, $\sigma_5 = \{X/a, X_1/a, Y_1/b, Y_2/b, Y_3/b, Y_4/a\}$ 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 [27]) is $<((0.8\&_P((0.7\&_P 0.9)\&_G 0.7))\&_G 0.7; \{X/a\}>$, that can be simplified, after evaluating the arithmetic expression, to $< 0.504; \{X/a\}>$.

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,

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

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 3.1 and 3.6,

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

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

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

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

The previous example from FLOPER [27] translated to the RFuzzy syntax used in the semantics sections is below.

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

6. Using the framework. Implementation details

Obviously, when coding in Prolog it is not possible to write the symbols we have used in Section 2. Here we expose the syntax used to write programs and some details of the implementation of the framework.

6.1. The programs syntax

Types are coded according to the following syntax:

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

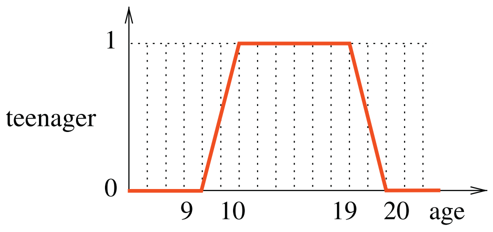


Fig. 1. Teenager truth value continuous representation.

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.

```
:- 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

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

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)value0.6.
```

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. Fig. 1 shows an example in which the continuous function assigns the truth value of being *teenager* to each age.

Functions used to define the truth value of some group of individuals are usually continuous and linear over intervals. To define those functions there is no necessity to write down the value assigned to each element in their domains. We have to take into account that the domain can be infinite.

RFuzzy provides the syntax for defining functions by stretches. This syntax is shown in (3). External brackets represent the Prolog list symbols and internal brackets represent cardinality in the formula notation. Predicate *pred* has arity 1, *val1*, …, *valN* should be ground terms representing numbers of the domain (they are possible values of the argument of *pred*) and *truth_val1*,…,*truth_valN* should be the truth values associated to these numbers. The truth value of the rest of the elements is obtained by interpolation.

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

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

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

Fuzzy clauses or rules have a simple syntax, defined in (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[12] or define his/her own connective.

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

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.

[12] 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*.

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

Default truth values syntax is defined in (5) and (6),

$$:\text{-}\ \mathsf{default}(pred/ar,\ truth\_value), \tag{5}$$
$$:\text{-}\ \mathsf{default}(pred/ar,\ truth\_value) = membership\_predicate/ar, \tag{6}$$

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.

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).
```

6.2. 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
```

6.3. 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 [47] 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($\mathcal{R}$) constraints by means of the RFuzzy package and
(2) the program with constraints is translated into ISO Prolog by using the CLP($\mathcal{R}$) package.

Fig. 2 shows the sequential process of program transformation.

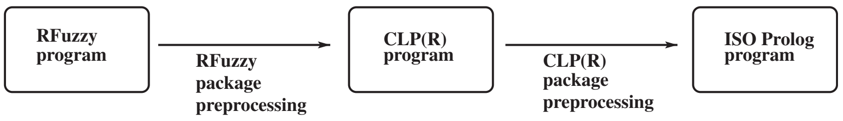


Fig. 2. RFuzzy architecture.

7. 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 two examples of applications that are currently using RFuzzy for modelling real-world problems.

7.1. 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 [4], 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, yet 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, other 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 Fig. 3.

7.2. 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 [9,8] 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 Fig. 4. Prolog is a perfect tool to design strategies for soccer players using simple rules close to human reasoning. Sometimes this reasoning

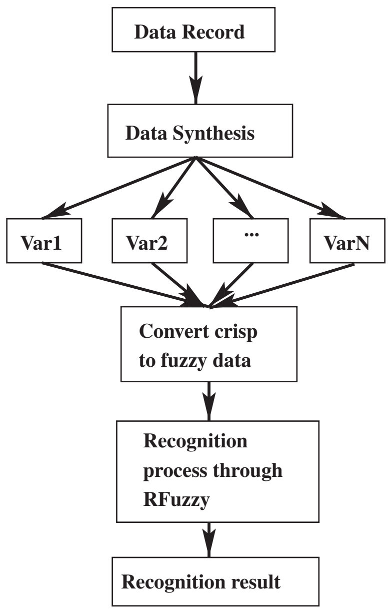

Data Record
Data Synthesis
Var1
Var2
...
VarN
Convert crisp to fuzzy data
Recognition process through RFuzzy
Recognition result

Fig. 3. Methodology of data recognition.

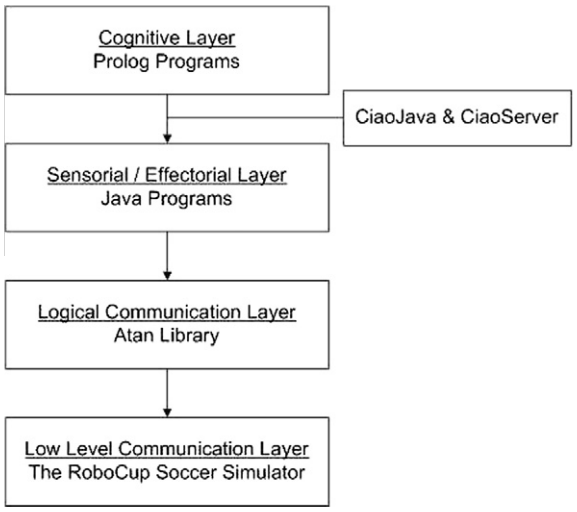

Cognitive Layer
Prolog Programs
CiaoJava & CiaoServer
Sensorial / Effectorial Layer
Java Programs
Logical Communication Layer
Atan Library
Low Level Communication Layer
The RoboCup Soccer Simulator

Fig. 4. System architecture for RoboCupSoccer server.

needs to deal with uncertainty, fuzziness or incompleteness of the information. These issues are solved in [9,8] by using Fuzzy Prolog [5,23,21,41]. 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.

Nowadays we are working on another proposal that uses a combination of Prolog and RFuzzy to implement the cognitive layer in RoboCupSoccer, which has the advantage of incorporating fuzzy logic as conventional logic in this layer [20]. The advantage with respect to former approaches is 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.

8. Conclusions

We presented a fuzzy logic programming framework called RFuzzy. RFuzzy is a serious attempt to provide a useful tool for modelling real 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 RFuzzy's characteristics are representation of default (and conditional default) values, types and syntax for functions and (continuous and discrete) rules. After some years of fertile research in this field, we provide in this work a complete compilation and reformulation of all previous publications related to syntax and operational semantics of RFuzzy. Interesting novel contributions are also included: we discuss some real applications for which our framework is currently being employed (Section 7). Most importantly, we provide the formal declarative semantics of our approach (Section 3) along with a soundness and completeness result that links operational and declarative semantics.

Acknowledgements

Work supported by the Research Staff Training Program BES-2008-008320 (Ministry of Science and Innovation) and projects DESAFIOS10 – TIN 2009-14599-C03-01 (Ministry of Science and Education) and PROMESAS – S-0505/TIC/0407 (Madrid Regional Government).

References

[1] J.M. Abietar, P.J. Morcillo, G. Moreno, Designing a software tool for fuzzy logic programming, in: G. Maroulis, T.E. Simos, (Eds.), Computational Methods in Science and Engineering, American Institute of Physics Conference Series, vol. 963, December 2007, pp. 1117–1120.
[2] J.F. Baldwin, T.P. Martin, B.W. Pilsworth, Fril-Fuzzy and Evidential Reasoning in Artificial Intelligence, John Wiley & Sons, Inc., New York, NY, USA, 1995.
[3] Stefano Bistarelli, Francesca Rossi, Semiring-based constraint logic programming: syntax and semantics, ACM Trans. Program. Lang. Syst. 23 (1) (2001) 1–29.
[4] Mahfuza Farooque, Susana Muñoz-Hernández, Easy fuzzy tool for emotion recognition: Prototype from voice speech analisys, in: J. Kacprzyk J. Filipe, A. Dourado, (Eds.), Proceeding of ICFC 2009 – First International Conference on Fuzzy Computation, Madeira, Portugal, INSTICC Press, October 2009.
[5] S. Guadarrama, S. Mu ñoz-Hernández, C. Vaucheret, Fuzzy prolog: a new approach using soft constraints propagation, Fuzzy Sets Syst. (FSS) 144 (1) (2004) 127–150. Possibilistic Logic and Related Issues.
[6] Siegfried Gottwald, A Treatise on Many-Valued Logics, Studies in Logic and Computation, vol. 9, Research Studies Press, 2001.
[7] Siegfried Gottwald, Mathematical fuzzy logic as a tool for the treatment of vague information, Inform. Sci. 172 (1–2) (2005) 41–71.
[8] Susana Muñoz-Hernández, 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, Springer, Cancún, México, 2007, pp. 635–645.

[9] Susana Muñoz-Hernández, 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.
[10] Mitsuru Ishizuka, Naoki Kanai, Prolog-elf incorporating fuzzy logic, in: IJCAI'85: Proceedings of the 9th international Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1985, pp. 701–703.
[11] Pascual Julián, Ginés Moreno, Jaime Penabad, On fuzzy unfolding: a multi-adjoint approach, Fuzzy Sets Syst. 154 (1) (2005) 16–33.
[12] F. Klawonn, R. Kruse, A Łukasiewicz logic based prolog, Math. Soft Comput. 1 (1) (1994) 5–29.
[13] Stephen Cole Kleene, Introduction to Metamathematics, D. Van Nostrand, New York, 1952.
[14] E.P. Klement, R. Mesiar, E. Pap, Triangular Norms, Kluwer Academic Publishers, 2000.
[15] B. Knaster, Un théorème sur les fonctions d'ensembles, Ann. Soc. Polonaise 6 (1928) 133–134.
[16] R.C.T. Lee, Fuzzy logic and the resolution principle, J. Assoc. Comput. Mach. 19 (1) (1972) 119–129.
[17] Deyi Li, Dongbo Liu, A Fuzzy Prolog Database System, John Wiley & Sons, Inc., New York, NY, USA, 1990.
[18] John Wylie Lloyd, Foundations of Logic Programming, second ed., Springer, 1987.
[19] Yann Loyer, Umberto Straccia, Any-world assumptions in logic programming, Theor. Comput. Sci. 342 (2–3) (2005) 351–381.
[20] Susana Munoz-Hernandez, Robot Soccer, Chapter RFuzzy: an easy and expressive tool for modelling the cognitive layer in RoboCupSoccer, INTECH, January 2010, pp. 267–284, ISBN: 978-953-307-036-0.
[21] Susana Muñoz-Hernández, Jose Manuél Gómez-Pérez, Solving collaborative fuzzy agents problems with CLP(FD), in: Manuel V. Hermenegildo, Daniel Cabeza (Eds.), PADL, Lecture Notes in Computer Science, vol. 3350, Springer, 2005, pp. 187–202.
[22] Susana Muñoz-Hernández, Víctor Pablos-Ceruelo, Hannes Strass, Rfuzzy: an expressive simple fuzzy compiler, in: Joan Cabestany, Francisco Sandoval, Alberto Prieto, Juan M. Corchado (Eds.), IWANN (1), Lecture Notes in Computer Science, vol. 5517, Springer, 2009, pp. 270–277.
[23] Susana Muñoz-Hernández, Claudio Vaucheret, Extending prolog with incomplete fuzzy information, in: Proceedings of the 15th International Workshop on Logic Programming Environments, CoRR, abs/cs/0508091, 2005.
[24] S. Muñoz-Hernández, C. Vaucheret, S. Guadarrama, Combining crisp and fuzzy logic in a prolog compiler, in: J.J. Moreno-Navarro, J. Mariño (Eds.), Joint Conference on Declarative Programming: APPIA-GULP-PRODE 2002, Madrid, Spain, September 2002, pp. 23–38.
[25] Robin Milner, A theory of type polymorphism in programming, J. Comput. Syst. Sci. 17 (1978) 348–375.
[26] Pedro J. Morcillo, 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, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 119–126.
[27] Pedro José Morcillo, Ginés Moreno, Floper, a fuzzy logic programming environment for research, in: Fundación Universidad de Oviedo (Eds.), Proceedings of VIII Jornadas Sobre Programación y Lenguajes (PROLE'08), Gijón, Spain, October 2008, pp. 259–263.
[28] Alan Mycroft, Richard A. O'Keefe, A polymorphic type system for prolog, Artif. Intell. 23 (3) (1984) 295–307.
[29] Jesús Medina, Manuel Ojeda-Aciego, Peter Vojtáš, A multi-adjoint approach to similarity-based unification, Electronic Notes in Theoretical Computer Science 66 (5) (2002) 70–85. UNCL'2002, Unification in Non-Classical Logics (ICALP 2002 Satellite Workshop).
[30] Jesús Medina, Manuel Ojeda-Aciego, Peter Vojtás, A completeness theorem for multi-adjoint logic programming, in: FUZZ-IEEE, 2001, pp. 1031–1034.
[31] Jesús Medina, Manuel Ojeda-Aciego, Peter Vojtás, Multi-adjoint logic programming with continuous semantics, in: Thomas Eiter, Wolfgang Faber, Miroslaw Truszczynski (Eds.), LPNMR, Lecture Notes in Computer Science, vol. 2173, Springer, 2001, pp. 351–364.
[32] Jesús Medina, Manuel Ojeda-Aciego, Peter Vojtás, A procedural semantics for multi-adjoint logic programming, in: Pavel Brazdil, Alípio Jorge (Eds.), EPIA, Lecture Notes in Computer Science, vol. 2258, Springer, 2001, pp. 290–297.
[33] Jesús Medina, Manuel Ojeda-Aciego, Peter Vojtás, Similarity-based unification: a multi-adjoint approach, Fuzzy Sets Syst. 146 (1) (2004) 43–62.
[34] Ginés Moreno, Building a fuzzy transformation system, in: Jirí Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, Julius Stuller (Eds.), SOFSEM, Lecture Notes in Computer Science, vol. 3831, Springer, 2006, pp. 409–418.
[35] Jan Pavelka, On fuzzy logic I, II and III, Z. Math. Logik Grundlagen der Math. 25 (5) (1979) 45–52. 119–134, 447–464.
[36] Víctor Pablos-Ceruelo, Susana Muñoz-Hernández, Hannes Strass, Rfuzzy framework, in: Paper Presented at the 18th Workshop on Logic-based Methods in Programming Environments (WLPE2008), CoRR, abs/0903.2188, 2009.
[37] Víctor Pablos-Ceruelo, Hannes Strass, Susana Mu noz Hernández, Rfuzzy—a framework for multi-adjoint fuzzy logic programming, in: Fuzzy Information Processing Society, 2009, NAFIPS 2009, Annual Meeting of the North American, pp. 1–6, June 2009.
[38] Ana Pradera, Enric Trillas, Tomasa Calvo, A general class of triangular norm-based aggregation operators: quasi-linear t-s operators, Int. J. Approx. Reason. 30 (1) (2002) 57–72.
[39] Z. Shen, L. Ding, M. Mukaidono, Fuzzy resolution principle, in: Proceedings of the 18th International Symposium on Multiple-valued Logic, vol. 5, 1989.
[40] 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, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1983, pp. 529–532.
[41] S. Mu noz-Hernández, C. Vaucheret, (Eds.), Default values to handel incomplete fuzzy information, IEEE Computational Intelligence Society Electronic Letter, vol. 14, IEEE, 2006, ISSN 0-7803-9489-5.
[42] Hannes Strass, Susana Muñoz-Hernández, Víctor Pablos-Ceruelo, Operational semantics for a fuzzy logic programming system with defaults and constructive answers, in: João Paulo Carvalho, Didier Dubois, Uzay Kaymak, João Miguel da Costa Sousa, (Eds.), IFSA/EUSFLAT Conference, 2009, pp. 1827–1832.
[43] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, Bart Demoen, Towards typed prolog, in: ICLP '08: Proceedings of the 24th International Conference on Logic Programming, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 693–697.
[44] Alfred Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific J. Math. 5 (1955) 285–309.
[45] George Theodorakopoulos, John S. Baras, Trust evaluation in ad-hoc networks, in: WiSe '04: Proceedings of the 3rd ACM Workshop on Wireless Security, ACM, New York, NY, USA, 2004, pp. 1–10.
[46] Enric Trillas, Susana Cubillo, Juan Luis Castro, Conjunction and disjunction on ($[0,1]$, $\leqslant$), Fuzzy Sets Syst. 72 (2) (1995) 155–165.
[47] The CLIP Lab. The Ciao Prolog Development System WWW Site. <http://www.clip.dia.fi.upm.es/Software/Ciao/>.
[48] Claudio Vaucheret, Sergio Guadarrama, Susana Muñoz-Hernández, Fuzzy prolog: a simple general implementation using CLP(R), in: Matthias Baaz, Andrei Voronkov (Eds.), LPAR, Lecture Notes in Artificial Intelligence, vol. 2514, Springer, 2002, pp. 450–464.
[49] Peter Vojtás, Fuzzy logic programming, Fuzzy Sets Syst. 124 (3) (2001) 361–370.
[50] Xizhao Wang, Eric C.C. Tsang, Suyun Zhao, Degang Chen, Daniel S. Yeung, Learning fuzzy rules from fuzzy samples based on rough set technique, Inform. Sci. 177 (20) (2007) 4493–4514.
[51] Yingjie Yang, Chris Hinde, A new extension of fuzzy sets using rough sets: R-fuzzy sets, Inform. Sci. 180 (3) (2010) 354–365.
[52] Lotfi A. Zadeh, Is there a need for fuzzy logic?, Inform Sci. 178 (13) (2008) 2751–2779.