Operational Semantics for a Fuzzy Logic Programming System with Defaults and Constructive Answers

Hannes Strass Susana Munoz-Hernandez Victor Pablos Ceruelo

Technical University of Madrid (Spain) *
E-mail: hannes.strass@alumnos.upm.es, {susana, vpablos}@fi.upm.es

Abstract— *In this paper we present the operational semantics of RFuzzy, a fuzzy Logic Programming framework that represents thruth values using real numbers from the unit interval. RFuzzy provides some useful extensions: default values to represent missing information, and typed terms to intuitively restrict predicate domains. Together, they allow the system to give constructive answers in addition to truth values. RFuzzy does not confine to a particular Fuzzy Logic, but aims at being as general as possible by using the notion of aggregation operators.*

Keywords— *Operational Semantics, Logic Programming Application*

1 Introduction

For many real-world problems, crisp knowledge representation is not perfectly adequate. Information that we handle might be imprecise, uncertain, or even both. Classical two-valued logic cannot easily represent these qualitative aspects of information. To address this issue, multiple frameworks for incorporating uncertainty in logic have been developed over the years: fuzzy set theory, probability theory, multi-valued logic, or possibilistic logic; to mention only some.

From the point of view of practical tools to support this reasoning the field is not so rich. Logic programming is however a perfect candidate for the implementation of these tools because it is traditionally used for problem solving and knowledge representation.

1.1 Fuzzy Logic approaches

The result of introducing Fuzzy Logic into Logic Programming has been the development of several fuzzy systems over Prolog. These systems replace the inference mechanism, SLD-resolution, of Prolog with a fuzzy variant that is able to handle partial truth [1]. Most of these systems implement the fuzzy resolution introduced by Lee in [2], examples being the Prolog-Elf system [3], the Fril Prolog system [4] and the F-Prolog language [5]. However, there is no common method for fuzzifying Prolog as has been noted in [6].

1.2 Fuzzy Prolog

One of the most promising fuzzy tools for Prolog was the "Fuzzy Prolog" system [7, 8]. This approach is more general than others in some respects:

1. A truth value is a finite union of sub-intervals on $[0, 1]$.
2. A truth value is propagated through the rules by means of an *aggregation operator*. The definition of *aggregation operator* is general.
3. Crisp and fuzzy reasoning are consistently combined in a Prolog compiler [9].

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

There are other proposals, e.g. in [10], that provide an interpretation of truth values as intervals, but Fuzzy Prolog proposed to generalise this concept to unions of intervals for the first time.

1.3 Multi-adjoint logic

Over the last few years several papers have been published by Medina et al. [11, 12, 13] about multi-adjoint programming. The theoretical model described in these works led to the development of FLOPER [14], another Fuzzy Logic Programming system. It has a Logic-Programming-inspired syntax and provides free choice of aggregation operators and credibility of rules just as RFuzzy does. There are however some things that FLOPER cannot do: (1) deal with missing information (which RFuzzy does by default truth value declarations), (2) type atoms and predicates to give constructive answers, and (3) provide syntactic sugar to express truth value functions.

1.4 Motivation

The generality of the approach pursued in [7, 8] turned out to make many users feel uncomfortable: Fuzzy Prolog is rather expressive, so it is not always clear how knowledge should be represented. Furthermore, interpreting the output that comes as a sequence of constraints maybe possible for a human but is very hard to do for a computer program – especially basing a decision upon it may not be straightforward.

To address these issues, we propose the RFuzzy framework. It is considerably simpler to use than the above-mentioned Fuzzy Prolog, but still contains many of its nice features. In RFuzzy, truth values will be represented by real numbers from the unit interval $[0, 1]$. This simplifies making modelling decisions and interpreting the output of the system. We still use the general concept of aggregation operators to be able to model different Fuzzy Logics. The rules of RFuzzy programs will have attached a *credibility* value to them: it allows the author

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

of the rule to express how much they confide in the relation expressed by the rule. In addition, RFuzzy offers features that are very useful for knowledge representation, namely default values and types.

The rest of the paper is organised as follows: Section 2 introduces the abstract syntax of RFuzzy. In Section 3 we present an operational semantics for RFuzzy and illustrate it with an example. The last but one section shortly sketches the key points of the implementation and Section 5 concludes.

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

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

Immediate examples for aggregation operators that come to mind are typical examples of t-norms and t-conorms: minimum $\min(a,b)$, maximum $\max(a,b)$, product $a \cdot b$, and probabilistic sum $a + b - a \cdot b$.

The above general definition of aggregation operators subsumes however all kinds of minimum, maximum or mean operators.

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

A *fuzzy clause* is written as

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

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

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

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

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

Example. 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:

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

The credibility value of the rule is 1.0, which means that we have no doubt about this relationship. The aggregation operator used here in both cases is the product "$\cdot$". We enrich the knowledge base with facts about some cities and their continents:

$$\begin{gathered}
\text{nice-weather(madrid)} \leftarrow 0.8,\\
\text{nice-weather(istanbul)} \leftarrow 0.7,\\
\text{nice-weather(moscow)} \leftarrow 0.2,\\
\text{many-sights(madrid)} \leftarrow 0.6,\\
\text{many-sights(istanbul)} \leftarrow 0.7,\\
\text{many-sights(sydney)} \leftarrow 0.6,\\
\text{city-continent(madrid, europe)} \leftarrow 1.0,\\
\text{city-continent(moscow, europe)} \leftarrow 1.0,\\
\text{city-continent(sydney, australia)} \leftarrow 1.0,\\
\text{city-continent(istanbul, europe)} \leftarrow 0.5,\\
\text{city-continent(istanbul, asia)} \leftarrow 0.5.
\end{gathered}$$

Some queries to this program could ask if Madrid is a good destination,
$\langle \text{good-destination(madrid)}, \text{v}\rangle$. Another query could ask if Istanbul is the perfect destination, $\langle \text{good-destination(istanbul)}, 1.0\rangle$.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. As this is rather the norm than the exception, we would like to have a mechanism that can handle non-present information.

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

Definition. A *default value declaration* for a predicate $p \in \Pi^{(n)}$ is written as $\texttt{default}(p(X_1,\ldots,X_n)) =$

$[\delta_1 \text{ if } \varphi_1, \dots, \delta_m \text{ if } \varphi_m]$ where $\delta_i \in [0,1]$ for all i. The φ_i are first-order formulas restricted to terms from $\mathrm{TU}_{\Sigma,\{X_1,\dots,X_n\}}$, the predicates $=$ and $\neq$, the symbol true, and the junctors $\wedge$ and $\vee$ in their usual meaning. ⌟

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

$$\texttt{default}(\textsf{nice-weather(X)}) = 0.5,$$
$$\texttt{default}(\textsf{many-sights(X)}) = 0.2,$$
$$\texttt{default}(\textsf{good-destination(X)}) = 0.3$$

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: $\texttt{default}(\textsf{city-continent(X, Y)}) = 0.0$. Notice that in this example $m = 1$ and $\varphi_1 = \mathsf{true}$ 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 can be viewed as inherent properties of terms – each term can have zero or more types. We use them to restrict the domains of predicates.

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

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

$$\textsf{madrid} : \textsf{City}, \textsf{istanbul} : \textsf{City},$$
$$\textsf{sydney} : \textsf{City}, \textsf{moscow} : \textsf{City};$$
$$\textsf{africa} : \textsf{Continent}, \textsf{america} : \textsf{Continent},$$
$$\textsf{antarctica} : \textsf{Continent},$$
$$\textsf{asia} : \textsf{Continent}, \textsf{europe} : \textsf{Continent}.$$

We also type the predicates in the obvious way:

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

◇

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

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

Example (continued). With respect to the given type declarations, city-continent(moscow, antarctica) is well-typed whileas 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 aggregation operator symbols. For Ω we will furthermore require that all operators from $\hat{\Omega}$ be monotonic.

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

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

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

⌟

3 Operational 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. There are 3 different cases of how a truth value can be determined:

- 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

ISBN: 978-989-95079-6-8

computation. This is formalised by setting the ordering $<_a$ on truth value attributes such that $\blacktriangle <_a \blacklozenge <_a \blacktriangledown$.

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

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

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. 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:

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

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

A *fuzzy Herbrand interpretation* (or short, *interpretation*) of a fuzzy logic program is a mapping $I : B \to \mathbb{T}$ that assigns truth values to ground atoms.

The *domain* of an interpretation is the set of all atoms to which a "proper" truth value is assigned: $\mathsf{Dom}(I) := \{A \mid A \in B, I(A) \succ \bot\}$.

For two interpretations I and J, we say I *is less than or equal to* J, written $I \sqsubseteq J$, if $I \sqsubseteq J$ iff $I(A) \preccurlyeq J(A)$ for all $A \in B$.

Accordingly, the infimum (or intersection) and supremum (or union) of interpretations are, for all $A \in B$, 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. [Model] Let $P = (R, D, T)$ be a fuzzy logic program.

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

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

iff for all valuations σ, we have: if $I(\sigma(B_i)) = z_i v_i \succ \bot$ for all i, then $I(\sigma(A)) \succcurlyeq z'v'$ where $z' = z_1 \circ \cdots \circ z_n$ and $v' = \hat{F}_c(c, \hat{F}(v_1, \ldots, v_n)))$.

For a 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: if $\sigma(p(X_1, \ldots, X_n))$ is well-typed (w.r.t. T), then there exists an $1 \le j \le m$ such that $\sigma(\varphi_j)$ holds and $I(\sigma(p(X_1, \ldots, X_n))) \succcurlyeq \blacktriangle\delta_j$.

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 $I \Vdash D$. ⌟

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

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

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

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

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

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

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

⌟

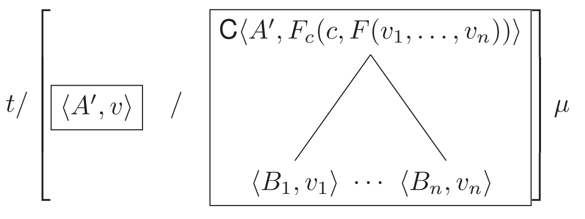
Computation nodes are essentially generalizations of queries that keep track of aggregation operator usage.

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

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

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

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



If there is a (variable disjoint instance of a) program clause $A \overset{c,F_c}{\longleftarrow}_F B_1, \ldots, B_n \in R$ 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

ISBN: 978-989-95079-6-8

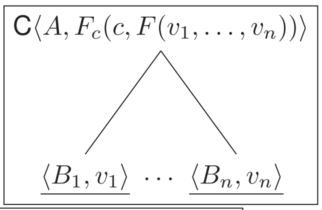
the node atom and the clause head to all the atoms in the tree.)

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

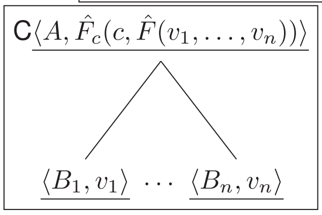
Default: $\boxed{t\,[\langle A,x\rangle]} \quad \rightharpoonup \quad \boxed{t\left[\langle A,x\rangle/\mathsf{D}\underline{\langle A,\delta_j\rangle}\right]\mu}$

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

Finalize:


$\mathsf{C}\langle A, F_c(c, F(v_1,\dots,v_n))\rangle$
$\underline{\langle B_1, v_1\rangle}\ \cdots\ \underline{\langle B_n, v_n\rangle}$

$\rightharpoonup$


$\mathsf{C}\underline{\langle A, \hat{F}_c(c, \hat{F}(v_1,\dots,v_n))\rangle}$
$\underline{\langle B_1, v_1\rangle}\ \cdots\ \underline{\langle B_n, v_n\rangle}$

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

⌟

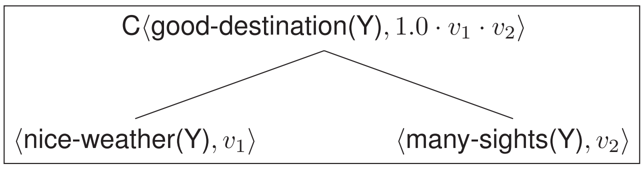
Here, the notation $t[A]$ means "the tree t that contains the node A somewhere". Likewise, $t[A/B]$ is to be read as "the tree t where the node A has been replaced by the node B".

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

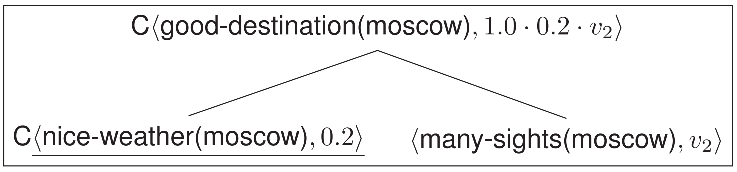
Example (continued). We start with the tree

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

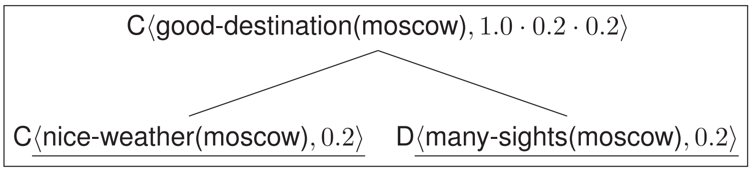
Applying the **Clause**-transition to the initial tree with the program clause good-destination(X)$\overset{1.0,\cdot}{\longleftarrow}$.nice-weather(X), many-sights(X) yields


$\mathsf{C}\langle \mathsf{good\text{-}destination(Y)}, 1.0 \cdot v_1 \cdot v_2\rangle$
$\langle \mathsf{nice\text{-}weather(Y)}, v_1\rangle$
$\langle \mathsf{many\text{-}sights(Y)}, v_2\rangle$

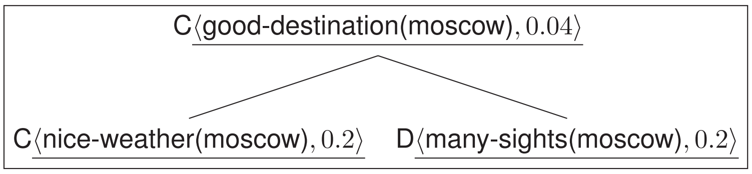
Now we apply **Clause** to the left child with nice-weather(moscow) $\leftarrow$ 0.2:


$\mathsf{C}\langle \mathsf{good\text{-}destination(moscow)}, 1.0 \cdot 0.2 \cdot v_2\rangle$
$\mathsf{C}\underline{\langle \mathsf{nice\text{-}weather(moscow)}, 0.2\rangle}$
$\langle \mathsf{many\text{-}sights(moscow)}, v_2\rangle$

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


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

In the last step, we finalise the root node.


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

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

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

⌟

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

4 Implementation

RFuzzy is implemented as a package of the Ciao Prolog System [15]. It consists essentially of a set of rules that translate the RFuzzy Syntax to ANSI Prolog using the expansion of code of the packages in Ciao Prolog. The predicates of the program that have been declared as fuzzy get an additional argument that makes the truth value explicit. The resulting program can then be interpreted and executed "as usual".

Example (continued). We have no space to describe the implementation syntax used at RFuzzy, but it can be easily deduced from the implementation of our running example (not all the clauses are shown here).

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

```
many_sights(madrid) value 0.6.
many_sights(sydney) value 0.6.

good_destination(X) cred (prod,1.0):~ prod
        nice_weather(X),
        many_sights(X).
```

The default value declarations are also very similar to the abstract syntax.

```
:- default(nice_weather/1, 0.5).
:- default(many_sights/1, 0.2).
:- default(good_destination/1, 0.3).
```

We use crisp predicates to represent types.

```
city(madrid).
city(moscow).
city(sydney).

:- set_prop(nice_weather/1) => city/1.
:- set_prop(many_sights/1) => city/1.
:- set_prop(good_destination/1) => city/1.
```

To ask queries to the system, we add a variable that is going to be instantiated with the truth value.

```
?- good_destination(moscow, V).

V = 0.04 ?
yes
```

But we cannot only ask for truth values of fully instantiated atoms. The real power of RFuzzy lies in the ability to provide constructive answers. For example, if we want to know "What is the best travel destination according to the knowledge base?" we just ask the following query. It looks for a destination D with a truth value V for which no destination with a higher truth value V1 exists.

```
?- good_destination(D, V),
   \+ (good_destination(_, V1),
       V1 > V).

D = madrid,
V = 0.48 ?

yes
```

As we see, the system returns "Madrid" as best destination and thus answers a question with an object rather than a truth value. ◇

5 Conclusions and Future Work

We presented the operational semantics of the RFuzzy framework for Fuzzy Logic Programming and showed some features of the implementation [1] via an example. We finally remark that a least model semantics and a least fixpoint semantics for RFuzzy also have been defined and proven equivalent to the operational semantics shown here.

[1]A complete release of the implementation is available at `http://babel.ls.fi.upm.es/software/rfuzzy`

References

[1] P. Vojtas. Fuzzy logic programming. *Fuzzy Sets and Systems*, 124(1):361–370, 2001.

[2] R. C. T. Lee. Fuzzy Logic and the resolution principle. *Journal of the Association for Computing Machinery*, 19(1):119–129, 1972.

[3] M. Ishizuka and N. Kanai. Prolog-ELF incorporating fuzzy Logic. In *IJCAI*, pages 701–703, 1985.

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

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

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

[7] C. Vaucheret, S. Guadarrama, and S. Munoz-Hernandez. Fuzzy prolog: A simple general implementation using clp(r). In P.J. Stuckey, editor, *Int. Conf. in Logic Programming, ICLP 2002*, number 2401 in LNCS, page 469, Copenhagen, Denmark, July/August 2002. Springer-Verlag.

[8] S. Guadarrama, S. Munoz-Hernandez, and C. Vaucheret. Fuzzy Prolog: A new approach using soft constraints propagation. *Fuzzy Sets and Systems, FSS*, 144(1):127–150, 2004. ISSN 0165-0114.

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

[10] H. T. Nguyen and E. A. Walker. *A first Course in Fuzzy Logic*. Chapman & Hall/Crc, 2000.

[11] J. Medina, M. Ojeda-Aciego, and P. Votjas. Multi-adjoint Logic Programming with continuous semantics. In *LPNMR*, volume 2173 of *LNCS*, pages 351–364, Boston, MA (USA), 2001. Springer-Verlag.

[12] J. Medina, M. Ojeda-Aciego, and P. Votjas. A procedural semantics for multi-adjoint Logic Programming. In *EPIA*, volume 2258 of *LNCS*, pages 290–297, Boston, MA (USA), 2001. Springer-Verlag.

[13] J. Medina, M. Ojeda-Aciego, and P. Votjas. A completeness theorem for multi-adjoint Logic Programming. In *International Fuzzy Systems Conference*, pages 1031–1034. IEEE, 2001.

[14] P.J. Morcillo and G. Moreno. Floper, a fuzzy logic programming environment for research. In *Proceedings of the Spanish Conference on Programming and Computer Languages, PROLE 2008*, Gijn, Spain, 2008.

[15] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.