Electronic Communications of the EASST
Volume 64 (2013)

Proceedings of the
XIII Spanish Conference on Programming
and Computer Languages
(PROLE 2013)

Automatic Proving of Fuzzy Formulae with Fuzzy Logic Programming and SMT

Miquel Bofill, Ginés Moreno, Carlos Vázquez and Mateu Villaret

19 pages

Guest Editors: Clara Benac Earle, Laura Castro, Lars-$\rangle AkeFredlund$
$ManagingEditors : TizianaMargaria, JuliaPadberg, GabrieleTaentzer$
$ECEASSTHomePage : http : //www.easst.org/eceasst/$ $ISSN 1863-2122$

Automatic Proving of Fuzzy Formulae with Fuzzy Logic Programming and SMT

Miquel Bofill[1], **Ginés Moreno**[2], **Carlos Vázquez**[2] **and Mateu Villaret**[1]

[1] Miquel.Bofill@udg.edu, Mateu.Villaret@udg.edu
Department of Computer Science, Applied Mathematics and Statistics
University of Girona
17071 Girona (Spain)

[2] Gines.Moreno@uclm.es, Carlos.Vazquez@uclm.es
Department of Computing Systems
University of Castilla-La Mancha
02071 Albacete (Spain)

Abstract: In this paper we deal with propositional fuzzy formulae containing several propositional symbols linked with connectives defined in a lattice of truth degrees more complex than *Bool*. We firstly recall an SMT (*Satisfiability Modulo Theories*) based method for automatically proving theorems in relevant infinitely-valued (including *Łukasiewicz* and *Gödel*) logics. Next, instead of focusing on satisfiability (i.e., proving the existence of at least one model) or unsatisfiability, our interest moves to the problem of finding the whole set of models (with a finite domain) for a given fuzzy formula. We propose an alternative method based on fuzzy logic programming where the formula is conceived as a goal whose derivation tree contains on its leaves all the models of the original formula, by exhaustively interpreting each propositional symbol in all the possible forms according the whole set of values collected on the underlying lattice of truth-degrees.

Keywords: Fuzzy Logic Programming; Automatic Theorem Proving; SMT

1 Introduction

Research on SAT (Boolean Satisfiability) and SMT (Satisfiability Modulo Theories) [BSST09] represents a successful and large tradition in the development of highly efficient automatic theorem solvers for classic logic. More recently there also exist attempts for covering fuzzy logics, as occurs with the approaches presented in [ABMV12, VBG12]. Moreover, if automatic theorem solving supposes too an starting point for the foundations of logic programming as well as one of its important application fields [Llo87, Sti88, Fit96, Apt90, Bra00], in this work we will show some preliminary guidelines about how fuzzy logic programming can face the automatic proving of fuzzy theorems.

Let us start our discussion with an easy motivating example. Assume that we have a very simple digital chip with just a single input port and just one output port, such that it reverts on *Out* the signal received from *In*. The behaviour of such chip can be represented by the following propositional formula $F : (In' \wedge Out) \vee (In \wedge Out')$. Depending on how we interpret

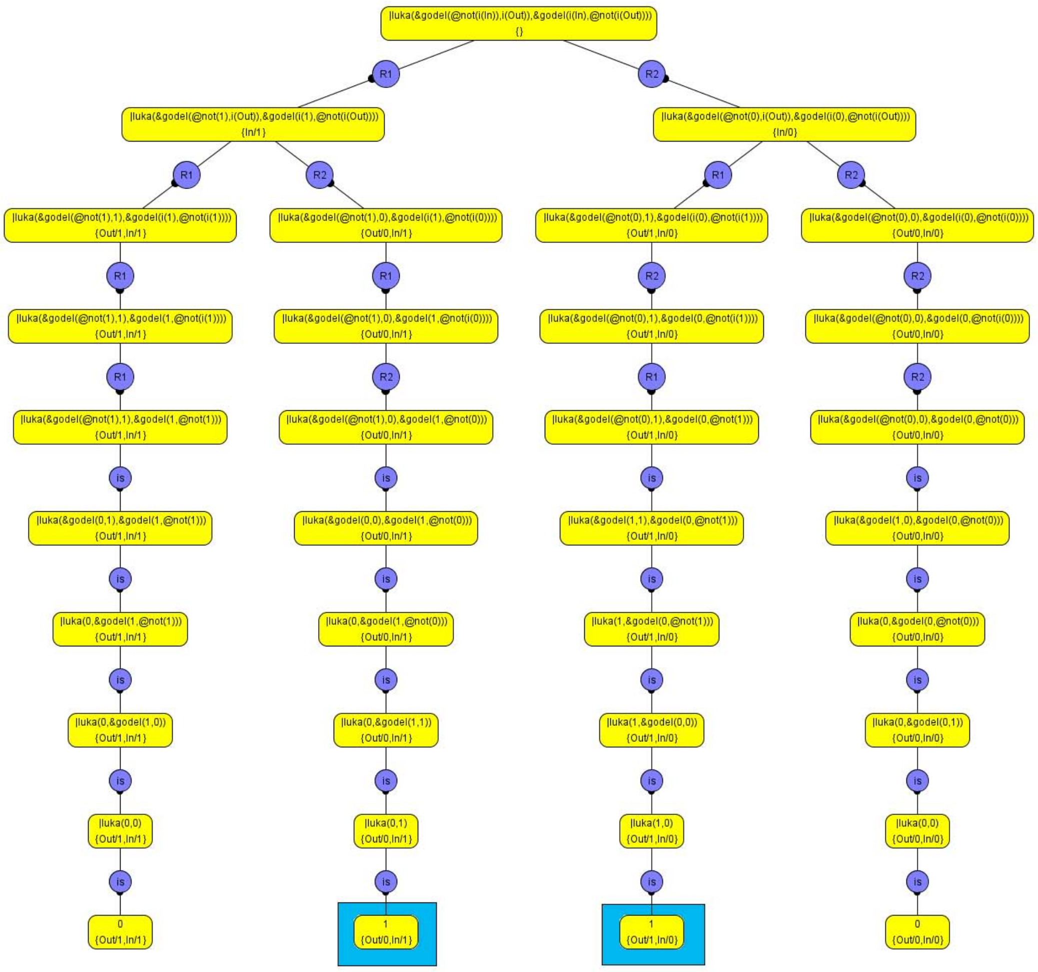

|luka(&godel(@not(i(In)),i(Out)),&godel(i(In),@not(i(Out))))
{}
R1
R2
|luka(&godel(@not(1),i(Out)),&godel(i(1),@not(i(Out))))
{In/1}
|luka(&godel(@not(0),i(Out)),&godel(i(0),@not(i(Out))))
{In/0}
R1
R2
R1
R2
|luka(&godel(@not(1),1),&godel(i(1),@not(i(1))))
{Out/1,In/1}
|luka(&godel(@not(1),0),&godel(i(1),@not(i(0))))
{Out/0,In/1}
|luka(&godel(@not(0),1),&godel(i(0),@not(i(1))))
{Out/1,In/0}
|luka(&godel(@not(0),0),&godel(i(0),@not(i(0))))
{Out/0,In/0}
R1
R1
R2
R2
|luka(&godel(@not(1),1),&godel(1,@not(i(1))))
{Out/1,In/1}
|luka(&godel(@not(1),0),&godel(1,@not(i(0))))
{Out/0,In/1}
|luka(&godel(@not(0),1),&godel(0,@not(i(1))))
{Out/1,In/0}
|luka(&godel(@not(0),0),&godel(0,@not(i(0))))
{Out/0,In/0}
R1
R2
R1
R2
|luka(&godel(@not(1),1),&godel(1,@not(1)))
{Out/1,In/1}
|luka(&godel(@not(1),0),&godel(1,@not(0)))
{Out/0,In/1}
|luka(&godel(@not(0),1),&godel(0,@not(1)))
{Out/1,In/0}
|luka(&godel(@not(0),0),&godel(0,@not(0)))
{Out/0,In/0}
is
is
is
is
|luka(&godel(0,1),&godel(1,@not(1)))
{Out/1,In/1}
|luka(&godel(0,0),&godel(1,@not(0)))
{Out/0,In/1}
|luka(&godel(1,1),&godel(0,@not(1)))
{Out/1,In/0}
|luka(&godel(1,0),&godel(0,@not(0)))
{Out/0,In/0}
is
is
is
is
|luka(0,&godel(1,@not(1)))
{Out/1,In/1}
|luka(0,&godel(1,@not(0)))
{Out/0,In/1}
|luka(1,&godel(0,@not(1)))
{Out/1,In/0}
|luka(0,&godel(0,@not(0)))
{Out/0,In/0}
is
is
is
is
|luka(0,&godel(1,0))
{Out/1,In/1}
|luka(0,&godel(1,1))
{Out/0,In/1}
|luka(1,&godel(0,0))
{Out/1,In/0}
|luka(0,&godel(0,1))
{Out/0,In/0}
is
is
is
is
|luka(0,0)
{Out/1,In/1}
|luka(0,1)
{Out/0,In/1}
|luka(1,0)
{Out/1,In/0}
|luka(0,0)
{Out/0,In/0}
is
is
is
is
0
{Out/1,In/1}
1
{Out/0,In/1}
1
{Out/1,In/0}
0
{Out/0,In/0}

Figure 1: Interpreting a formula with two values for signals/propositions *In* and *Out*.

each propositional symbol, we obtain the following final set of interpretations for the whole formula:

$$
\begin{array}{llllllll}
I1: & \{In = 0, Out = 0\} & \Rightarrow & F = 0 & \qquad I2: & \{In = 0, Out = 1\} & \Rightarrow & F = 1 \\
I3: & \{In = 1, Out = 0\} & \Rightarrow & F = 1 & \qquad I4: & \{In = 1, Out = 1\} & \Rightarrow & F = 0
\end{array}
$$

A SAT solver easily proves that F is satisfiable since, in fact, it has two models (i.e., interpretations of the propositional variables *In* and *Out* that assign 1 to the whole formula) represented by $I2$ and $I3$. An alternative way for explicitly obtaining such interpretations consists of using the fuzzy logic environment FLOPER developed in our research group [MMPV10, MMPV11, MMPV12, JMM+13] (http://dectau.uclm.es/floper/). As we will explain in the rest of the paper, when FLOPER executes the following goal (representing formula F) "`(@not(i(In)) & i(Out)) | (i(In) & @not(i(Out)))`" with respect to a fuzzy logic program com-

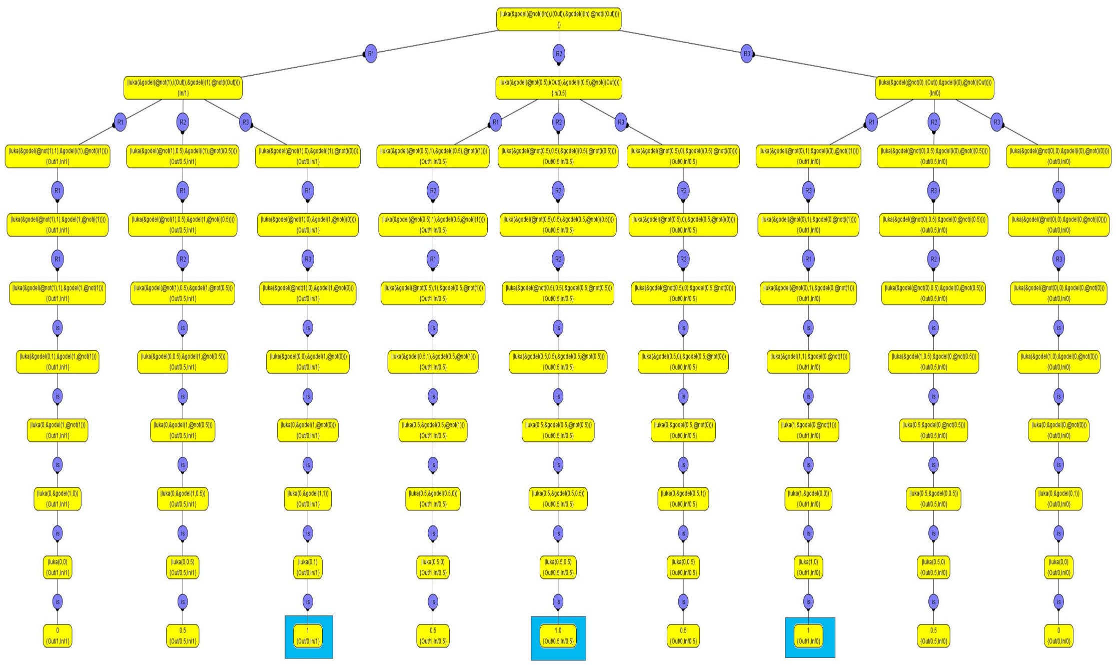


Figure 2: Interpreting a formula with three values for signals/propositions In and Out.

posed by just two rules: "`i(1) with 1`" and "`i(0) with 0`", it draws the tree shown in Figure 1, where models $I2$ and $I3$ appear in the two central leaves of the tree inside a blue box. Each branch in the tree starts by interpreting variables In and Out and continues with the evaluation of operators appearing in F.

Note that whereas formula F describes the behaviour of our chip in an "implicit way", the whole set of models $I2$ and $I3$ "explicitly" describes how the chip successfully works (any other interpretation not being a model, represents an abnormal behaviour of the chip), hence the importance of finding the whole set of models for a given formula.

Assume now that we plan to model an "analogic" version of the chip, where both the input and output signals might vary in an infinite range of values between 0 and 1, such that Out will simply represent the "complement" of In. The new behaviour of the chip can be expressed again by the same previous formula, but taking into account now that connectives involved in F could be defined in a fuzzy way as follows (see also Figure 3 afterwards):

x'	$=$	$1-x$	Product logic's negation
$x \wedge y$	$=$	$min(x,y)$	Gödel logic's conjunction
$x \vee y$	$=$	$min(x+y,1)$	Łukasiewicz logic's disjunction

Here we could use an SMT solver to prove that F is satisfiable. Following the approach of the work in [ABMV12], it can be easily checked that F is satisfiable[1] with the following SMT-LIB script, encoding the previous connectives into SAT modulo linear real arithmetic:

[1] The formula has infinite models of the form $\{In = x, Out = y\}$ such that $x + y = 1$.

```
; Set logic: Quantifier Free Linear Real Arithmetic
(set-logic QF_LRA)

; min(x,y)
(define-fun min ((x Real) (y Real)) Real
  (ite (> x y) y x))

; x' = 1 - x
(define-fun agr_not ((x Real)) Real
  (- 1 x))

; &G(x,y) = min{x,y}
(define-fun and_godel ((x Real) (y Real)) Real
  (min x y))

; |L(x,y) = min{x+y,1}
(define-fun or_luka ((x Real) (y Real)) Real
  (min (+ x y) 1))

; Declaration of variables
(declare-fun x () Real)
(declare-fun y () Real)

; Ordering relation
(assert (>= x 0))
(assert (<= x 1))
(assert (>= y 0))
(assert (<= y 1))

; Formula to check
(assert (= (or_luka (and_godel (agr_not x) y)
                    (and_godel x (agr_not y)))
            1))

; Check for satisfiability
(check-sat)
```

It is easy to understand the SMT-LIB syntax of the previous code. Just in case, we recall that the `ite` expression corresponds to the *if-then-else* construct. Note that all necessary connectives are easily encoded as is done in [ABMV12]. It is worth noting that, apart from proving satisfiability of a formula, SMT solvers can be used to prove that a formula is a theorem, by checking unsatisfiability of its negation.

On the other hand, Figure 2 shows too three models in the tree depicted by FLOPER when considering only three kinds of values (that is, 0, 0.5 and 1) for interpreting In and Out. Such models include, apart of $I2$ and $I3$ seen before, the interpretation $\{In = 0.5, Out = 0.5\}$ since, as we can see in the detailed computations performed along the middle branch of the tree, we have:

$$\begin{array}{lll}(0.5' \wedge 0.5) \vee (0.5 \wedge 0.5') & = ((1-0.5) \wedge 0.5) \vee (0.5 \wedge 0.5') & = (0.5 \wedge 0.5) \vee (0.5 \wedge 0.5') \\ = min(0.5, 0.5) \vee (0.5 \wedge 0.5') & = 0.5 \vee (0.5 \wedge 0.5') & = 0.5 \vee (0.5 \wedge (1-0.5)) \\ = 0.5 \vee (0.5 \wedge 0.5) & = 0.5 \vee min(0.5, 0.5) & = 0.5 \vee 0.5 \\ = min(0.5+0.5, 1) & = min(1, 1) & = 1\end{array}$$

Similarly, we can check for instance in the second branch of the tree that $\{In = 1, Out = 0.5\}$ is not a model (in fact, our chip can not return a signal of value 0.5 when its input is 1) since:

$$\begin{array}{lllllll}(1' \wedge 0.5) \vee (1 \wedge 0.5') & = & ((1-1) \wedge 0.5) \vee (1 \wedge 0.5') & = & (0 \wedge 0.5) \vee (1 \wedge 0.5') & = \\ min(0, 0.5) \vee (1 \wedge 0.5') & = & 0 \vee (1 \wedge 0.5') & = & 0 \vee (1 \wedge (1-0.5)) & = \\ 0 \vee (1 \wedge 0.5) & = & 0 \vee min(1, 0.5) & = & 0 \vee 0.5 & = \\ min(0+0.5, 1) & = & min(0.5, 1) & = & 0.5 & \end{array}$$

To finish this section, let us comment some connections between the two main topics of this work, i.e. Fuzzy Logic Programming and Fuzzy SMT, with *Answer Set Programming (ASP)*, a well-known declarative programming paradigm oriented towards combinatorial search problems which has been recently combined with fuzzy logic [VDV07]. In ASP, *answer sets* are models computed according to the stable model semantics of logic programming. This introduces non-monotonic reasoning into logic programming, and gives raise to a paradigm that is different from the proof-derivation approach of PROLOG. In [MO09], it is presented an ASP semantics for a kind of fuzzy logic programs very close to MALP, based on the idea of finding models that we also use in this paper when analyzing fuzzy logic formulae with our FLOPER tool. On the other hand, *Constraint Satisfaction Problems (CSPs)* are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations and hence, SMT and ASP can be roughly thought of as certain forms of CSPs. The translation of answer-set programs into SMT has been considered in [Nie08, JNS09, NJN13]. Also, ASP has been recently combined with SMT in [WZ11]. Although its nonmonotonic form of reasoning is out of the scope of this paper, we are interested in coping with this topic in future extensions of our proposal.

The structure of this paper is as follows. Section 2 constitutes the core of our paper and it is divided in three blocks. In sub-section 2.1 we present the main features of our fuzzy logic programming environment FLOPER, which are used in sub-section 2.2 for analyzing several fuzzy formulae, whereas in sub-section 2.3 we provide some interesting hints on cost measures associated to our method. Finally, we conclude in Section 3 by also describing a few number of challenging lines for future research.

2 MALP, FLOPER and Automatic Theorem Proving

In what follows we describe a very simple *subset of the* MALP[2] *language* (see [MOV04, JMP09] for a complete formulation of this framework), which in essence consists of a first-order language, $\mathscr{L}$, containing variables, constants, function symbols, predicate symbols, and several

[2] Multi-Adjoint Logic Programming.

$\&_{\texttt{P}}(x,y) \triangleq x * y$	$\vert_{\texttt{P}}(x,y) \triangleq x+y-x*y$	$\leftarrow_{\texttt{P}}(x,y) \triangleq \min(1,x/y)$
$\&_{\texttt{G}}(x,y) \triangleq \min(x,y)$	$\vert_{\texttt{G}}(x,y) \triangleq \max\{x,y\}$	$\leftarrow_{\texttt{G}}(x,y) \triangleq \begin{cases} 1 & \text{if } y \leq x \\ x & \text{otherwise} \end{cases}$
$\&_{\texttt{L}}(x,y) \triangleq \max(0,x+y-1)$	$\vert_{\texttt{L}}(x,y) \triangleq \min\{x+y,1\}$	$\leftarrow_{\texttt{L}}(x,y) \triangleq \min\{x-y+1,1\}$

Figure 3: Conjunctors, disjunctors and implications from *Product*, *Gödel* and *Łukasiewicz* logics.

(arbitrary) connectives to increase language expressiveness: implication connectives (denoted by $\leftarrow_1, \leftarrow_2, \ldots$); conjunctive connectives ($\wedge_1, \wedge_2, \ldots$), disjunctive connectives ($\vee_1, \vee_2, \ldots$), and hybrid operators (usually denoted by $@_1, @_2, \ldots$), all of them are grouped under the name of "aggregators". Although these connectives are binary operators, we usually generalize them as functions with an arbitrary number of arguments. So, we often write $@(x_1,\ldots,x_n)$ instead of $@(x_1,\ldots,@(x_{n-1},x_n))$. By definition, the truth function for an n-ary aggregation operator $[\![@]\!] : L^n \rightarrow L$ is required to be monotonous.

Additionally, our language $\mathscr{L}$ contains the values of a lattice $(L,\leq)$ and a set of connectives interpreted over such lattice. In general, L may be the carrier of any complete bounded lattice where a L-expression is a well-formed expression composed by values of L, as well as variable symbols, connectives and *primitive operators* (i.e., arithmetic symbols such as $*, +, min$, etc.). In what follows, we assume that the truth function of any connective @ in L is given by its corresponding *connective definition*, that is, an equation of the form $@(x_1,\ldots,x_n) \triangleq E$, where E is a L-expression not containing variable symbols apart from $x_1,\ldots,x_n$. For instance, some fuzzy connective definitions in the lattice $([0,1],\leq)$ are presented in Figure 3 (from now on, this lattice will be called $\mathscr{V}$ along this paper), where labels L, G and P mean respectively *Łukasiewicz logic*, *Gödel logic* and *product logic* (with different capabilities for modeling *pessimistic*, *optimistic* and *realistic scenarios*, respectively).

This subset of MALP is intended to cope with fuzzy propositional formulae like $P \wedge Q \rightarrow P \vee Q$, where propositions P and Q are interpreted as values of the lattice. To this end, a *program* is defined as a set of rules (also called "facts") of the form "H *with* v", where H is an atomic formula or atom (usually called *head*), and v is its associated *truth degree* (i.e., a value of L). More precisely, in our application, heads have always the form "$i(v)$" and each program rule looks like "$i(v)$ *with* v". It is noteworthy to point out that even when we use the same names for constants (building data terms) and truth degrees, the Herbrand Universe of each program and the carrier set of its associated lattice should never be confused, since they are in fact disjoint sets.

A *goal* is a formula built from atomic formulas $B_1,\ldots,B_n$ ($n \geq 0$), truth values of L, conjunctions, disjunctions and aggregations, submitted as a query to the system. In this subset of MALP, the atomic formulas of a *goal* have always the form "$i(P)$", being P a variable symbol. In this way, when running a simple goal like "$i(P)$" (as done in Figure 4), we could obtain several answers meaning something like "when $P = v$, then the resulting truth degree is v", representing all possible interpretations in L for proposition P in the original formula.

The procedural semantics of this subset of the MALP language consists of an operational phase (based on admissible steps that exploits the atoms in the goal), followed by an interpretive phase (that performs arithmetic operations to interpret the resulting formula on the lattice). In the following, $\mathscr{C}[A]$ denotes a formula where A is a sub-expression which occurs in the –possibly empty– context $\mathscr{C}[]$. Moreover, $\mathscr{C}[A/A']$ means the replacement of A by A' in context $\mathscr{C}[]$.

Definition 1 (Admissible Step) Let $\mathscr{Q}$ be a goal and let σ be a substitution. The pair $\langle \mathscr{Q}; \sigma \rangle$ is a *state*. Given a program $\mathscr{P}$, an *admissible computation* is formalized as a state transition system, whose transition relation $\rightarrow_{AS}$ is defined as the least one satisfying:

$$\langle \mathscr{Q}[A]; \sigma \rangle \quad \rightarrow_{AS} \quad \langle (\mathscr{Q}[A/v])\theta; \sigma\theta \rangle$$

where A is the selected atom in $\mathscr{Q}$, $\theta = mgu(\{H = A\})$[3] and "H *with* v" in $\mathscr{P}$. An *admissible derivation* is a sequence $\langle \mathscr{Q}; id \rangle \rightarrow_{AS} \cdots \rightarrow_{AS} \langle \mathscr{Q}'; \theta \rangle$.

If we exploit all atoms of a given goal, by applying admissible steps as much as needed during the operational phase, then it becomes a formula with no atoms (a L-expression) which can be then interpreted w.r.t. lattice L as follows.

Definition 2 (Interpretive Step and Fuzzy Computed Answer) Let $\mathscr{P}$ be a program, $\mathscr{Q}$ a goal and σ a substitution. Assume that $[\![@]\!]$ is the truth function of connective $@$ in the lattice $(L, \leq)$ associated to $\mathscr{P}$, such that, for values $r_1, \ldots, r_n, r_{n+1} \in L$, we have that $[\![@]\!](r_1, \ldots, r_n) = r_{n+1}$. Then, we formalize the notion of *interpretive computation* as a state transition system, whose transition relation $\rightarrow_{IS}$ is defined as the least one satisfying:

$$\langle \mathscr{Q}[@(r_1, \ldots, r_n)]; \sigma \rangle \quad \rightarrow_{IS} \quad \langle \mathscr{Q}[@(r_1, \ldots, r_n)/r_{n+1}]; \sigma \rangle$$

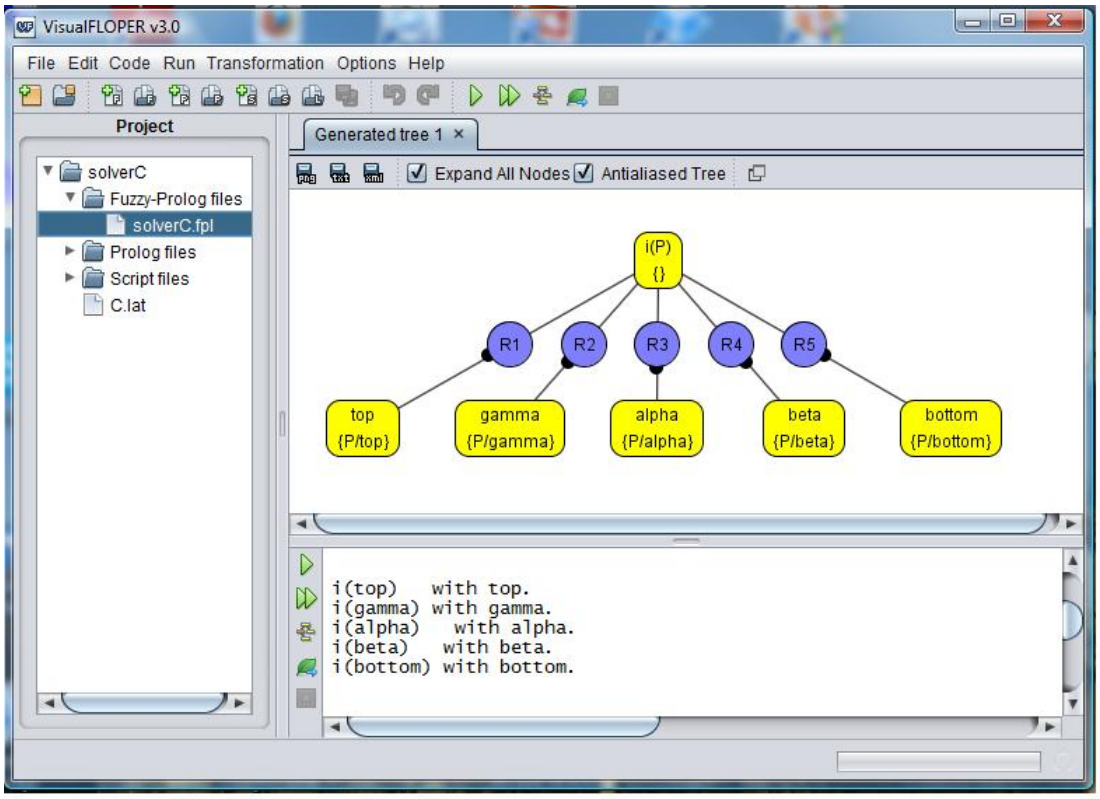
An *interpretive derivation* is a sequence $\langle \mathscr{Q}; \sigma \rangle \rightarrow_{IS} \cdots \rightarrow_{IS} \langle \mathscr{Q}'; \sigma \rangle$. When $\mathscr{Q}' = r \in L$, the state $\langle r; \sigma \rangle$ is called a *fuzzy computed answer* (f.c.a.) for that derivation.

2.1 The Fuzzy Logic Programming Environment FLOPER

The parser of our FLOPER tool [MMPV10, MMPV11] has been implemented by using the Prolog language. Once the application is loaded inside a Prolog interpreter, it shows a menu which includes options for loading/compiling, parsing, listing and saving MALP programs, as well as for executing/debugging fuzzy goals. Moreover, in [MMPV10] we explain that FLOPER has been recently equipped with new options, called "`lat`" and "`show`", for allowing the possibility of respectively changing and displaying the lattice associated to a given program.

A very easy way to model truth-degree lattices for being included into the FLOPER tool is also described in [MMPV10], according the following guidelines. All relevant components of each lattice are encapsulated inside a Prolog file which must necessarily contain the definitions of a minimal set of predicates defining the set of valid elements (`member/1` predicate), the top and bottom elements (`top/1` and `bot/1` predicates), the full or partial ordering established among them (`leq/2` predicate), as well as the repertoire of fuzzy connectives which can be used for their subsequent manipulation. If we have, for instance, some fuzzy connectives of the form $\&_{label_1}$ (conjunction), $|_{label_2}$ (disjunction) or $@_{label_3}$ (aggregation) with arities n_1, n_2 and n_3 respectively, we must provide clauses defining the *connective predicates* "`and_`$label_1$`/(`$\mathtt{n_1}$`+1)`",

[3] Here $mgu(E)$ denotes the *most general unifier* of an equation set E [LMM88].



```
i(top)    with top.
i(gamma)  with gamma.
i(alpha)   with alpha.
i(beta)   with beta.
i(bottom) with bottom.
```

Figure 4: A work-session with FLOPER solving goal `i(P)`.

"`or`_*label*$_2$`/(`n$_2$`+1)`" and "`agr`_*label*$_3$`/(`n$_3$`+1)`", where the extra argument of each predicate is intended to contain the result achieved after the evaluation of the proper connective. Finally, for the purposes of the current work, we also require for each lattice a Prolog fact of the form `members(L)` being the `L` a list containing the set of truth degrees belonging to the modeled lattice (or at least a representative subset of them when working with infinite lattices) for being used when interpreting propositional variables of fuzzy formulae. For instance, a lattice defining the simplest notion of binary lattice should implement predicate `member/1` with facts `member(0)` and `member(1)` (including also `members([0,1])`) and the Boolean conjunction could be defined by the pair of facts `and_bool(0,_,0)` and `and_bool(1,X,X)`.

Following the Prolog style regulated by the previous guidelines, in Figure 5, we show the set of clauses modeling the more flexible lattice $\mathscr{V}$, which enables the possibility of working with truth degrees in the infinite space of the real numbers between 0 and 1, offering too a wide range of conjunction and disjunction operators recasted from the three typical fuzzy logics described before (i.e., the *Łukasiewicz*, *Gödel* and *product* logics), as well as a useful description for the hybrid aggregator *average* and the negation connective. Note that we have included the fact "`members([0,0.5,1]).`" which implies that in our application, propositional variables involved in fuzzy formulae to be proved, only three values (i.e., real numbers 0, 0.5 and 1, collected from the infinite unit interval) should be considered for interpreting such formulas, as

```
member(X)  :- number(X),0=<X,X=<1.                members([0,0.5,1]).
leq(X,Y)  :- X=<Y.                  bot(0).                   top(1).

and_luka(X,Y,Z)  :- pri_add(X,Y,U1),pri_sub(U1,1,U2),pri_max(0,U2,Z).
and_godel(X,Y,Z):- pri_min(X,Y,Z).
and_prod(X,Y,Z)  :- pri_prod(X,Y,Z).

or_luka(X,Y,Z)   :- pri_add(X,Y,U1),pri_min(U1,1,Z).
or_godel(X,Y,Z)  :- pri_max(X,Y,Z).
or_prod(X,Y,Z)   :- pri_prod(X,Y,U1),pri_add(X,Y,U2),
                    pri_sub(U2,U1,Z).

agr_aver(X,Y,Z)  :- pri_add(X,Y,U),pri_div(U,2,Z).
agr_not(X,Y)     :- pri_sub(1,X,Y).

pri_add(X,Y,Z)   :- Z is X+Y.   pri_min(X,Y,Z)  :- (X=<Y,Z=X;X>Y,Z=Y).
pri_sub(X,Y,Z)   :- Z is X-Y.   pri_max(X,Y,Z)  :- (X=<Y,Z=Y;X>Y,Z=X).
pri_prod(X,Y,Z)  :- Z is X * Y. pri_div(X,Y,Z)  :- Z is X/Y.
```

Figure 5: Prolog code for representing lattice $\mathscr{V}$, which models truth degrees in the real interval $[0,1]$ with standard fuzzy connectives.

shown in Figure 2.

Note also that we have included definitions for auxiliary predicates, whose names always begin with the prefix "pri_". All of them are intended to describe primitive/arithmetic operators (in our case $+$, $-$, $*$, $/$, *min* and *max*) in a Prolog style, for being appropriately called from the bodies of clauses defining predicates with higher levels of expressivity (this is the case for instance, of the three kinds of fuzzy connectives we are considering: conjunctions, disjunctions and aggregations).

Since till now we have considered two classical, fully ordered lattices (with a finite and infinite number of elements, respectively), we wish now to introduce a different case coping with a very simple lattice where not always any pair of truth degrees are comparable. So, consider the following partially ordered lattice $\mathscr{F}$ in the diagram of Figure 6, which is equipped with conjunction, disjunction and implication connectives based on the *Gödel* logic described in Figure 3, but with the particularity that now, in the general case, the *Gödel*'s conjunction must be expressed as $\&_{\mathtt{G}}(x,y) \triangleq inf(x,y)$, where it is important to note that we must replace the use of "*min*" by "*inf*" in the connective definition (and similarly for the disjunction connective, where "*max*" must be substituted by "*sup*").

To this end, observe in the Prolog code accompanying the graphic in Figure 6 that we have introduced clauses defining the primitive operators "pri_inf/3" and "pri_sup/3" which are intended to return the *infimum* and *supremum* of two elements. Related with this fact, we must point out the following aspects:

- Note that since truth degrees α and β are incomparable, then any call to goals of the form "?- leq(alpha,beta)." or "?- leq(beta,alpha)." will always fail.

$$\begin{tikzcd}
 & \top \arrow[d, dash] & \\
 & \gamma \arrow[dl, dash] \arrow[dr, dash] & \\
\alpha \arrow[dr, dash] & & \beta \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

```
member(bottom).     member(alpha).
member(beta).       member(gamma).
member(top).
members([bottom,alpha,beta,gamma,top]).
leq(bottom,X).     leq(X,X).    leq(alpha,gamma).
leq(beta,gamma).   leq(X,top).   l(X,X) :- !.
l(X,Y):-leq(X,Y),!.   l(X,Y):-leq(X,Z),l(Z,Y).
and_godel(X,Y,Z)  :- pri_inf(X,Y,Z).
or_godel(X,Y,Z)  :- pri_sup(X,Y,Z).
agr_impl(X,Y,Z)  :-  (l(Y,X),!,Z=top ;  Z=X).
pri_inf(X,Y,X):- l(X,Y),  !.
pri_inf(X,Y,Y):- l(Y,X),  !.
pri_inf(_,_,bottom).
pri_sup(X,Y,X):- l(Y,X),  !.
pri_sup(X,Y,Y):- l(X,Y),  !.
pri_sup(_,_,gamma).
```

Figure 6: The finite, partially ordered lattice $\mathscr{F}$ modeled in Prolog.

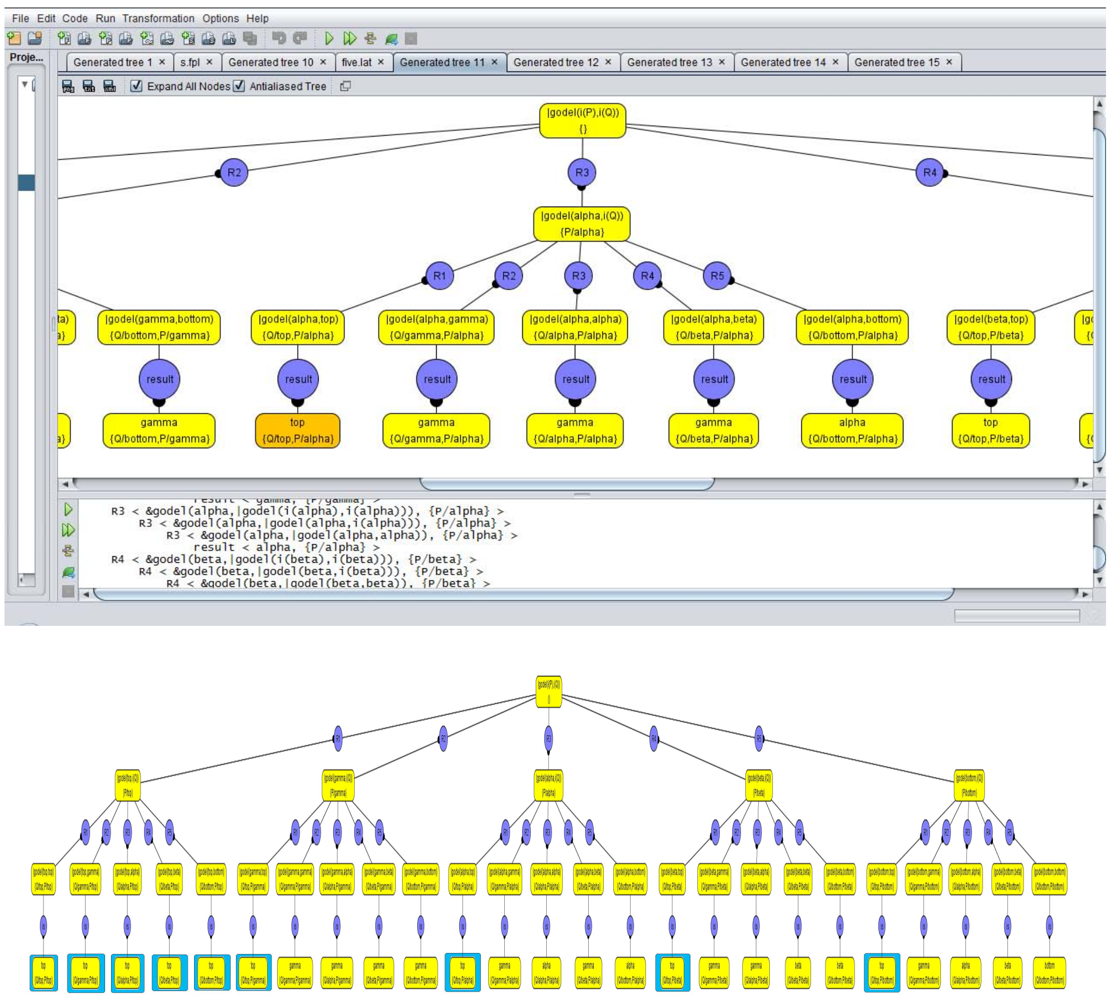
- A goal of the form "`?- pri_inf(alpha,beta,X).`", instead of failing, successfully produces the desired result "`X=bottom`".
- Note anyway that the implementation of the "`pri_inf/3`" predicate is mandatory for coding the general definition of "`and_godel/3`" (a similar reasoning follows for "`pri_sup/3`" and "`or_godel/3`").

2.2 Some Examples

This subset of the MALP language suffices for developing a simple fuzzy theorem prover, where it is important to remark that our tool can cope with different lattices (not only the real interval [0,1]) containing a finite number of elements -marked in "`members`"- maintaining full or partial ordering among them. Hence, we can use FLOPER for enumerating the whole set of interpretations and models of fuzzy formulae. To this end, only a concrete lattice L is required in order to automatically build a program composed by a set of facts of the form "$i(\alpha)$ *with* α", for each $\alpha \in L$. For instance, the MALP program associated to lattice $\mathscr{F}$ in Figure 6 looks like:

```
i(top)          with    top.
i(gamma)        with    gamma.
i(alpha)        with    alpha.
i(beta)         with    beta.
i(bottom)       with    bottom.
```

Once the lattice and the residual program have been loaded into FLOPER, the formula to be evaluated is introduced as a goal to the system following some conventions:



```
R3 < &godel(alpha,|godel(i(alpha),i(alpha))), {P/alpha} >
    R3 < &godel(alpha,|godel(alpha,i(alpha))), {P/alpha} >
        R3 < &godel(alpha,|godel(alpha,alpha)), {P/alpha} >
            result < alpha, {P/alpha} >
R4 < &godel(beta,|godel(i(beta),i(beta))), {P/beta} >
    R4 < &godel(beta,|godel(beta,i(beta))), {P/beta} >
        R4 < &godel(beta,|godel(beta,beta)), {P/beta} >
```

Figure 7: A work-session with FLOPER solving formula $P \vee Q$ (25 interpretations, 9 models).

- If P is a propositional variable in the original formula, then it is denoted as "`i(P)`" in the goal F.
- If & is a conjunction of a certain logic "label" in the original formula, then it is denoted as "`&label`" in goal F.
- For disjunctions, negations and implications, use respectively "`|label`", "`@no_label`" and "`@im_label`" in F.
- For other aggregators use "@$_{\texttt{label}}$" in F.

In what follows we discuss some examples related with the lattice shown in Figure 6 and its residual MALP program just seen before. Firstly, if we execute goal "`i(P)`" into FLOPER, we obtain the five interpretations for `P` shown in Figure 4. On the other hand, consider now the

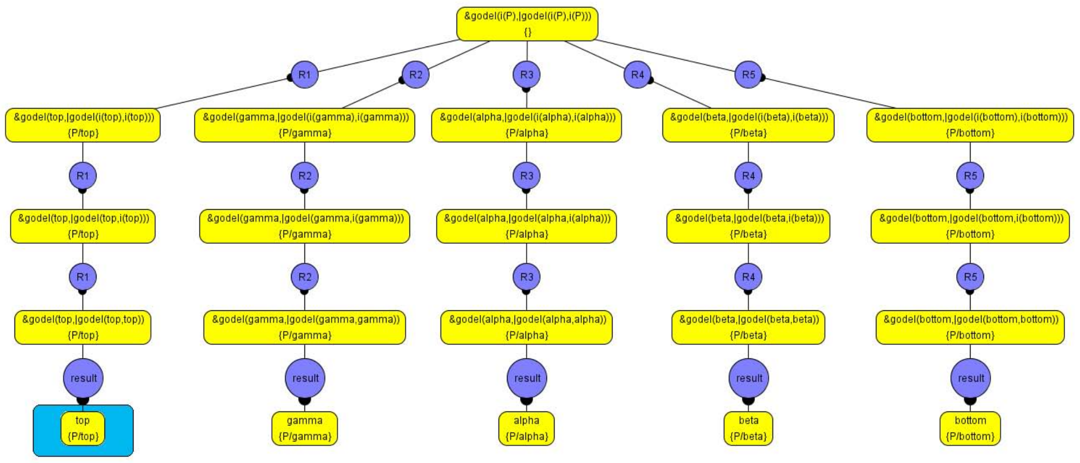

&godel(i(P),|godel(i(P),i(P)))
{}
R1
R2
R3
R4
R5
&godel(top,|godel(i(top),i(top)))
{P/top}
&godel(gamma,|godel(i(gamma),i(gamma)))
{P/gamma}
&godel(alpha,|godel(i(alpha),i(alpha)))
{P/alpha}
&godel(beta,|godel(i(beta),i(beta)))
{P/beta}
&godel(bottom,|godel(i(bottom),i(bottom)))
{P/bottom}
&godel(top,|godel(top,i(top)))
&godel(gamma,|godel(gamma,i(gamma)))
&godel(alpha,|godel(alpha,i(alpha)))
&godel(beta,|godel(beta,i(beta)))
&godel(bottom,|godel(bottom,i(bottom)))
&godel(top,|godel(top,top))
&godel(gamma,|godel(gamma,gamma))
&godel(alpha,|godel(alpha,alpha))
&godel(beta,|godel(beta,beta))
&godel(bottom,|godel(bottom,bottom))
result
top
gamma
alpha
beta
bottom

Figure 8: Full proof tree for formula $P \wedge (P \vee P)$ with 1 model among 5 interpretations.

propositional formula $P \vee Q$, which is translated into the MALP goal "`(i(P) | i(Q))`" and after being executed with FLOPER, the tool returns a tree[4] whose 25 leaves represent the whole set of interpretations (9 of them -inside blue clouds- are models) as seen in Figure 7. See also Figure 8 associated to formula $P \wedge (P \vee P)$.

Consider now the more involved formula $P \wedge Q \rightarrow P \vee Q$ which becomes into the MALP goal "`(i(P) & i(Q)) @impl (i(P) | i(Q))`". When interpreted by FLOPER, the system returns the list of answers displayed in Figure 9, having all them the maximum truth degree "top", which proves that this formula is a tautology, as wanted.

2.3 Some Hints on Cost Measures

We wish to finish this section by providing some comments about cost measures and efficiency. So, given a lattice L, a formula F and its associated proof tree T, we define the following values:

- v is the number of distinct variables in F.
- v' is the number of occurrences (including repetitions) of variables in F.
- c is the number of connectives in F.
- r is the number of (marked) elements in lattice L given by predicate "`members`".

And now we have that:

- The width of the tree T, or total number of interpretations of F, is r^v.

[4] Each state contains its corresponding goal and substitution components and they are drawn inside yellow circles. Admissible steps, colored in blue, are labeled with the program rule they exploit. Finally, those blue circles annotated with word "*is*", correspond to interpretive steps. Sometimes we include blue circles labeled with "*result*" which represents a chained sequence of interpretive steps.

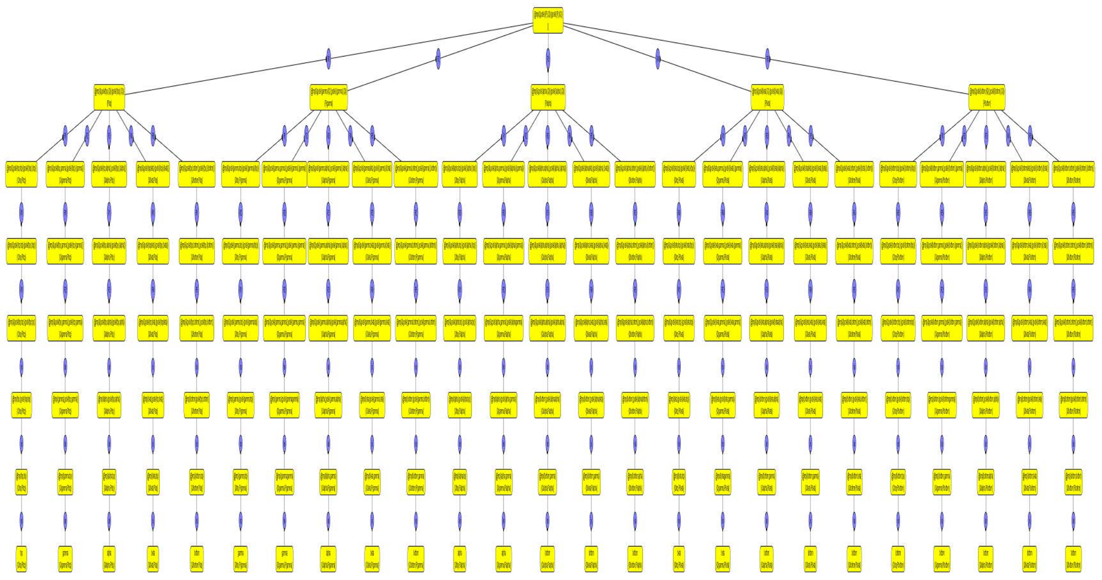


Figure 9: Full proof tree for tautology $P \wedge Q \rightarrow P \vee Q$ (25 models).

- The number of admissible steps performed on a single branch of T is v'.
- The number of interpretive steps performed on a single branch of T is c.
- The depth of T, or number of computational (admissible/interpretive) steps for each possible interpretation of F is $v' + c$.
- An upper bound for the total number of admissible steps in T is $|as| \leq (v' - v)r^v + \sum_{i=1}^{v} r^i$.
- An upper bound for the total number of interpretive steps in T is $|is| \leq cr^v$.
- An finally, an upper bound for the total number of computational (admissible and interpretive) steps is $|T| \leq (c + v' - v)r^v + \sum_{i=1}^{v} r^i$.

Let us come back again to tautology $P \wedge Q \rightarrow P \vee Q$ for which FLOPER displays the whole set of models seen in Figure 9, and assume now a more general version with the following shape $P_1 \wedge \ldots \wedge P_n \rightarrow P_1 \vee \ldots \vee P_n$ for which we have studied its behaviour in the table of Figure 10. In the horizontal axis we represent the number n of different propositional variables appearing in the formula, whereas the vertical axis refers to the number of seconds[5] needed to obtain the whole set of interpretations (all them are models in this case) for the formula. Both the red and blue lines refers to the method just commented along this paper, but whereas the red line indicates that

[5] The benchmarks have been performed using a computer with processor Intel Core Duo, with 2 GB RAM and Windows Vista.

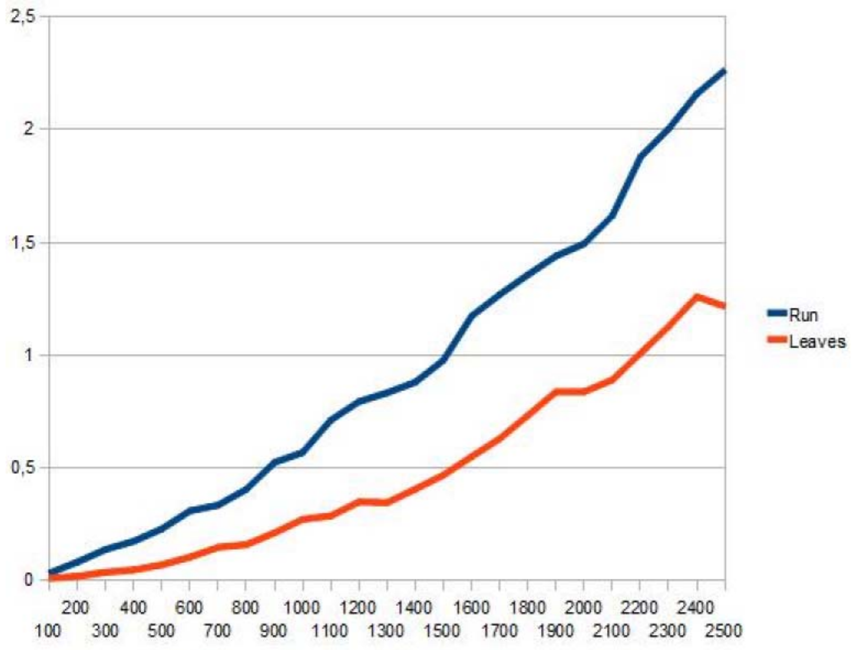


Figure 10: Behaviour of the method.

the derivation tree has been produced by performing admissible and interpretive steps according Definitions 1 and 2, respectively, the blue line refers to the execution of the Prolog code obtained after compiling with FLOPER the MALP program and goal associated to our intended formula. The results achieved in the figure show that our method has a nice behaviour in both cases, even for formulae with a big number of propositional variables. Of course, the method does not try to compete with SAT techniques (which are always faster and can deal with more complex formulae containing many more propositional variables), but it is important to remark again that in our case, we face the problem of finding the whole set of models for a given formula, instead of only focusing on satisfiability.

3 Conclusions and Future Work

In this paper we have focused on two different, but related problems regarding the automatic proving of propositional fuzzy formulae. In particular, whereas an SMT solver has been used for checking satisfiability, an alternative technique based on fuzzy logic programming has been introduced for finding the whole set of interpretations which are models for a given formula. In the future we plan to introduce improvements on both methods, regarding the set of truth degrees collected in the lattice used for interpreting a given formula. In the case of SMT, we plan to investigate how to cope with lattices equipped with partial ordering among their elements, whereas for the method based on fuzzy logic programming, it is important to design a much more flexible technique for dealing with infinite lattices than the one we have used in this paper

```
<node>
    <rule>R0</rule>
    <goal>or_godel(i(P),i(Q))</goal>
    <substitution>{}</substitution>
    <children>
        <node>
            <rule>R1</rule>
            <goal>or_godel(bottom,i(Q))</goal>
            <substitution>{P/bottom}</substitution>
            <children>
                <node>
                    <rule>R1</rule>
                    <goal>or_godel(bottom,bottom)</goal>
                    <substitution>{Q/bottom,P/bottom}</substitution>
                    <children>
                        <node>
                            <rule>result</rule>
                            <goal>bottom</goal>
                            <substitution>{Q/bottom,P/bottom}</substitution>
                            <children>
                            </children>
                        </node>
                    </children>
                </node>
                ...
```

Figure 11: Part of the XML file representing the execution tree shown in Figure 7.

(based on "pointing out" just a few number of truth degrees in the infinite space). In this last sense, some *halting rules* and *branch cuts* should be needed (maybe through *alfa cuts*) or even it could be interesting to study how to obtain all models of a formula by a constraint (as $x+y=1$ for the example given in the Introduction about the analogical chip) or a set of constraints. Moreover, we are also interested in reinforcing our techniques by making use of recent advances produced in the field of (fuzzy variants of) ASP.

Anyway, we are nowadays planning to make use of the standard "*XML Path Language*" XPath [BBC+07] in order to automate the process of directly extracting the set of models contained on proof trees once they have been exported by FLOPER in XML format. In what follows, we simply draft some key-point ideas which can be very useful when dealing with complex formulae beyond the simpler examples seen along this paper since, as formally stated in sub-section 2.3, the number of interpretations grows exponentially w.r.t. the set of propositions and connectives included on fuzzy formulae to be tested. Moreover, in the near future, the method will be reinforced by means of the *"fuzzy XPath"* tool developed too with FLOPER as described in [ALM11, ALM12, ALMV13] and which is freely available from http://dectau.uclm.es/fuzzyXPath. So, let us recall that XPath was designed as a query language for XML text in which the path of the underlying tree of any XML document is used to describe the query (subsequent nodes on XPath expressions are separated by a simple slash `/` or a double slash `//`, being this last case useful for overriding several nodes). Moreover, XPath expressions can be adorned with Boolean

```
<root>
  <substitution>{Q/top,P/bottom}</substitution>
  <substitution>{Q/top,P/alpha}</substitution>
  <substitution>{Q/top,P/beta}</substitution>
  <substitution>{Q/top,P/gamma}</substitution>
  <substitution>{Q/bottom,P/top}</substitution>
  <substitution>{Q/alpha,P/top}</substitution>
  <substitution>{Q/beta,P/top}</substitution>
  <substitution>{Q/gamma,P/top}</substitution>
  <substitution>{Q/top,P/top}</substitution>
</root>
```

Figure 12: XML file obtained after evaluating an XPath query.

conditions (between square brackets `'[]'`) on nodes and leaves to restrict the number of answers of the query. For instance, we have used the XPath online tool http://www.xpathtester.com/test for executing the query "`//node[goal='top']/substitution`" against the XML file shown in Figure 11, which was generated by FLOPER when producing the proof tree drawn in Figure 7, thus returning as output the new XML document listed in Figure 12. As illustrated in Figure 11, note that the XML files representing proof trees exported by FLOPER, are always rooted with the `node` label, whose children are based on four finds of 'tags' (this structure is nested as much as needed):

- `rule`, which indicates the program rule evaluated to reach the current node (the virtual rule `R0` is pointed out only in the initial node),
- `goal`, which contains the MALP expression under evaluation, that is, the formula that the system is trying to prove on its current initial/intermediate/final step. Note that, when in our example such value is `top`, then we have found a model, where the values assigned to the propositional symbols of the formula are collected in the following tag...
- `substitution`, which accumulates the variable bindings performed along a fuzzy logic derivation (i.e., chain of computational steps along a branch of the execution tree) and whose meaning in our target setting, reveals the way of interpreting the propositions contained on a formula for obtaining its models (see Figure 12, where the nine solutions labeled with this tag and reported in the output XML document, indicate each one the truth values for the propositional variables that satisfy the formula with the maximum truth degree), and finally
- `children`, which contains the set of underlying nodes of the tree in a nested way.

As we have just revealed, the combined use of fuzzy logic programming together with the standard XPath language for XML data retrieval, admits a challenging feedback applicable to the automatic search of models of fuzzy formulae for which we plan to introduce extra capabilities by using the flexible resources of our *"fuzzy XPath"* system developed with FLOPER.

Acknowledgements: This work has been partially supported by the EU (FEDER), and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grants TIN2012-33042 and TIN2013-45732-C4-2-P. Moreover, Carlos Vázquez and Ginés Moreno received grants for International mobility from the University of Castilla-La Mancha (CYTEMA project and *Vicerrectorado de Profesorado*).

Bibliography

[ABMV12] C. Ansótegui, M. Bofill, F. Manyà, M. Villaret. Building Automated Theorem Provers for Infinitely-Valued Logics with Satisfiability Modulo Theory Solvers. In *Proceedings of the 42nd IEEE International Symposium on Multiple-Valued Logic, ISMVL 2012*. Pp. 25–30. 2012.

[ALM11] J. Almendros-Jiménez, A. Luna, G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In Bassiliades et al. (eds.), *Proceedings of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11. Barcelona, Spain, July 19–21*. Pp. 186–193. Springer Verlag, LNCS 6826, 2011.

[ALM12] J. Almendros-Jiménez, A. Luna, G. Moreno. Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language. *Electronic Notes in Theoretical Computer Science* 282:3–18, 2012.

[ALMV13] J. Almendros-Jiménez, A. Luna, G. Moreno, C. Vázquez. Analyzing Fuzzy Logic Computations with Fuzzy XPath. In Fredlund and Castro (eds.), *Actas de las XIII Jornadas sobre Programación y Lenguajes, PROLE'13, Jornadas SISTEDES, Madrid, Spain, September 18-20*. Pp. 136–150 ("work in progress" track, extended version submitted to ECEASST). Universidad Complutense de Madrid (ISBN: 978-84-695-8331-9), 2013.

[Apt90] K. R. Apt. Logic Programming. In Leewen (ed.), *Handbook of Theoretical Computer Science*. Volume B: Formal Models and Semantics, chapter 10, pp. 493–574. MIT Press, Massachusetts Institute of Technology, USA, 1990.

[BBC$^+$07] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, J. Siméon. XML path language (XPath) 2.0. *W3C*, 2007.

[Bra00] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison Wesley, September 2000.

[BSST09] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications 185, pp. 825–885. IOS Press, 2009.

[Fit96] M. Fitting. *First-order logic and automated theorem proving*. Springer Verlag, 1996.

[JMM$^+$13] P. Julián, J. Medina, P. Morcillo, G. Moreno, M. Ojeda-Aciego. An Unfolding-Based Preprocess for Reinforcing Thresholds in Fuzzy Tabulation. In *Advances in Computational Intelligence - Proc of the 12th International Work-Conference on Artificial Neural Networks, IWANN 2013. Tenerife, Spain, June 12-14*. Pp. 647–655. Springer Verlag, LNCS 7902, PART I, 2013.

[JMP09] P. Julián, G. Moreno, J. Penabad. On the Declarative Semantics of Multi-Adjoint Logic Programs. In *Proceedings of the 10th International Work-Conference on Artificial Neural Networks, IWANN'09. Salamanca, Spain, June 10-12*. Pp. 253–260. Springe Verlag, LNCS 5517, 2009.

[JNS09] T. Janhunen, I. Niemelä, M. Sevalnev. Computing Stable Models via Reductions to Difference Logic. In Erdem et al. (eds.), *Proc. of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2009, Potsdam, Germany, September 14-18*. Pp. 142–154. Springer Verlag, LNCS 5753, 2009.

[Llo87] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[LMM88] J. L. Lassez, M. J. Maher, K. Marriott. Unification Revisited. In Minker (ed.), *Foundations of Deductive Databases and Logic Programming*. Pp. 587–625. Morgan Kaufmann, Los Altos, California, 1988.

[MMPV10] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In al. (ed.), *Proceedings of 4th Intl Symposium on Rule Interchange and Applications, RuleML'10. Washington, USA, October 21–23*. Pp. 119–126. Springer Verlag, LNCS 6403, 2010.

[MMPV11] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Fuzzy Computed Answers Collecting Proof Information. In al. (ed.), *Advances in Computational Intelligence. Proceedings of the 11th International Work-Conference on Artificial Neural Networks, IWANN 2011. Torremolinos, Spain, June 8-10*. Pp. 445–452. Springer Verlag, LNCS 6692, 2011.

[MMPV12] P. J. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Dedekind-MacNeille completion and Cartesian product of multi-adjoint lattices. *Int. J. Comput. Math.* 89(13-14):1742–1752, 2012.

[MO09] N. Madrid, M. Ojeda-Aciego. On Coherence and Consistence in Fuzzy Answer Set Semantics for Residuated Logic Programs. In Gesù et al. (eds.), *Proc. of the 8th International Workshop on Fuzzy Logic and Applications, WILF 2009. Palermo, Italy, June 9-12*. Pp. 60–67. Springer Verlag, LNCS 5571, 2009.

[MOV04] J. Medina, M. Ojeda-Aciego, P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems* 146:43–62, 2004.

[Nie08] I. Niemelä. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence* 53(1-4):313–329, 2008.

[NJN13] M. Nguyen, T. Janhunen, I. Niemelä. Translating Answer-Set Programs into Bit-Vector Logic. In Tompits et al. (eds.), *Proc. of the 19th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2011. Vienna, Austria, September 28-30*. Pp. 95–113. Springer Verlag, LNCS 7773, 2013.

[Sti88] M. E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated reasoning* 4(4):353–380, 1988.

[VBG12] A. Vidal, F. Bou, L. Godo. An SMT-Based Solver for Continuous t-norm Based Logics. In *Proceedings of the 6th International Conference on Scalable Uncertainty Management, SUM 2012. Marburg, Germany, September 17-19*. Pp. 633–640. Springer Verlag, LNCS 7520, 2012.

[VDV07] D. Van Nieuwenborgh, M. De Cock, D. Vermeir. An introduction to fuzzy answer set programming. *Annals of Mathematics and Artificial Intelligence* 50(3-4):363–388, 2007.

[WZ11] Y. Wang, M. Zhang. Answer Set Programming Modulo Theories. In Zhang (ed.), *Applied Informatics and Communication*. Communications in Computer and Information Science 228, pp. 655–661. Springer Berlin Heidelberg, 2011.