A FUNCTIONAL TOOL FOR FUZZY FIRST ORDER LOGIC EVALUATION

J. MIGUEL CLEVA[1], VICTORIA LÓPEZ[2], JAVIER MONTERO[3]
[1]*Fac. Informatics, Complutense University, Madrid, Spain. Email:jcleva@sip.ucm.es*
[2]*Dept. Computer Science, Nebrija University, Madrid, Spain. Email: vlopez@mat.ucm.es*
[3]*Fac. Mathematics, Complutense University, Madrid, Spain. Email: monty@mat.ucm.es*

In this paper we present an automatic evaluation tool for fuzzy first order logic formulas. Since different semantics can be considered for every logical symbol, we allow for such formulas the appearance of syntactic modifiers, in such away that our tool is designed not only to evaluate formulas in existing fuzzy semantics, but also to evaluate the properties in any other semantic framework given by the user. Such generalization is performed using Haskell functional programming language.

1. Introduction

Verification and software quality measures are important fields nowadays. There exist many different approaches for the verification of software, which requires a logical specification of prerequisites and results of each program under consideration. Such verification mechanisms are considered in many different paradigms, such as imperative[8], functional[7] or functional-logic languages[4]. According to the specific program characteristics and the properties to be verified, different techniques can be taken into account. Main verification techniques are model-checking[1], theorem proving[10] and testing[12], but alternative combinations between them can be considered, together with other formal methods (like abstract interpretation[5], for example). Model checking verifies that a program, formalized as a transition system, satisfies a given temporal logic formula. Model checking is a very efficient technique to verify such temporal formulas from an initial state (starting point of our computation procedure). Theorem proving consists of verifying a given logical formula over a system, which is being specified as a program. Theorem provers can be distinguished by the language in

which systems are specified (see, e.g., Isabelle[10], Coq[3] or PVS[11]). Testing is commonly used for huge systems in which the previous approaches cannot give a result in reasonable time (it is also used to speed up decisions about system specifications).

From a formalization and verification point of view, the classical approach for verification of systems is the Hoare alternative[8], where the specification of the system is done by a pair of first order logical (FOL) formulas, and the verification of the imperative system uses the Hoare's deduction rules (see Hoare[8]). Nevertheless, this approach is not enough to deal with other programs executed in parallel. In this case, considered specification is temporal logic, which reflects the idea of system evolution in time. But systems nowadays are even more complex, since they evolve in space too. Hence, spatial-temporal logics are introduced in order to specify programs. Still, specification requisites may be inexact, not fitting standard crisp formalism. For this reason we have introduced a fuzzy logic approach for the specification of program properties[9]. But such a fuzzy logic approach requires a certain level of certainty of a given formula, to be chosen from different interpretations.

The tool we present in this paper has been initially developed to assist an expert to select a given semantics for a given situation, based upon Haskell[2], a lazy functional programming language which seems appropriate for a general but efficient tool. In particular, with this tool we calculate the values of a fuzzy first order formula for a collection of possible semantics given by the user. In this way, experts can get a better knowledge for their decisions.

The paper is structured as follows: section 2 is devoted to a survey on specification of software; in section 3 we develop our tool for evaluating fuzzy logic formulas, followed by a section 4 with several examples and a final section for conclusions and future research work.

2. Software specification

Software verification requires to formalize characteristics of the system. This is the main objective of software specification, where the properties each algorithm must verify are given, by means of a precondition and a postcondition. Precondition describes the situation in which the algorithm can be applied (otherwise we may get undesired results). Postcondition describes the relations between the input data and the output of the given algorithm, at the end of the computation process. Such statements are for-

malized as First Order Logic (FOL) formulas (see[6] for a survey on classical logic). From those formulas and the Hoare's deduction rules[8] for verification of programs, a formal verification of the algorithm can be developed, by applying some appropriate deduction rules in order to deduce from the precondition the final postcondition.

The main problem on specification is that properties of a system are given by demand (clients provide the requisites our program should satisfy). Such requisites are usually given in natural language, so they use to be ambiguous. To cope with that possibility we have considered as specification framework the fuzzy logic approach[9]. Under this approach, systems are specified as triples (program, precondition and postcondition), specified as fuzzy FOL formulas.

For verification and evaluation purposes, a specific method has been developed[9] in order to evaluate the confidence level of our program, once a particular specification has been given. Hence, we need an evaluation of any given fuzzy logic formula, so we can interpret the relation between precondition and postcondition. An automatic system will be very useful for this purpose.

3. Evaluating Fuzzy FOL formulas

In this section we present the main characteristics of the evaluation tool we have developed. Our main goal was to provide a mechanism to evaluate fuzzy first order formula within a given interpretation semantic, to be considered by the expert for deciding about such semantics for the validation of the program.

The main characteristic of this tool is the possibility of dealing with any semantics for the interpretation of fuzzy logic formulas. For the implementation of our tool we have considered Haskell[2], a functional programming language allowing to deal with functions as arguments.

3.1. *The evaluation tool*

The function that evaluates the formula in a given scenario is called `eval`. The implementation of this function is shown in figure 1. This function makes use of many other functions that check the correctness of the input data, split the formula in different tokens to be evaluated, and auxiliary functions to calculate the partial values of different kinds of formulas.

The general form of this function is:

`eval` *form semlist univ intlist numcharlist cont modlist*

```
eval :: [Char] -> [(Char,Float->Float->Float)] -> [Char]
        -> [(Char,Float->Float)] -> [(Char,Char,Float)]
        -> [(Char,Char)] -> [(Char,Float->Float)] -> Float
eval xs fs us rs cs es ms = evalR xs fs 0 us rs cs es ms

evalR :: [Char] -> [(Char,Float->Float->Float)] -> Float ->
         [Char] -> [(Char,Float -> Float)] -> [(Char,Char,Float)]
         -> [(Char,Char)] -> [(Char,Float->Float)] -> Float
evalR [] _ v _ _ _ _ _= v
evalR ('(':xs) fs v us rs cs es ms=
    let (ys,zs)=formulaS('(':xs) in
      evalR zs fs (evalR ys fs v us rs cs es ms) us rs cs es ms
evalR ('&':xs) fs v us rs cs es ms =
    let (ys,zs)=formulaS(xs) in
       evalR zs fs ((funcion '&' fs) v
             (evalR ys fs v us rs cs es ms)) us rs cs es ms
evalR ('|':xs) fs v us rs cs es ms =
    let (ys,zs)=formulaS(xs) in
       evalR zs fs (dor v (evalR ys fs v us rs cs es ms)
             (funcion '&' fs) (funcion '~' fs)) us rs cs es ms
evalR ('>':xs) fs v us rs cs es ms =
    let (ys,zs)=formulaS(xs) in
        evalR zs fs (funcionI (funcion '>' fs) (funcion '~' fs)
             (funcion '&' fs) v (evalR ys fs v us rs cs es ms))
              us rs cs es ms
evalR ('~':xs) fs v us rs cs es ms =
    let (ys,zs)=formulaS(xs) in
        evalR zs fs ((funcion '~' fs)
             (evalR ys fs v us rs cs es ms) 0) us rs cs es ms
evalR ('A':xs) fs v us rs cs es ms =
    let (ys,zs,var)=formulaSC(xs) in
        evalR zs fs (valoraA ys var fs v us rs cs es ms)
              us rs cs es ms
evalR ('E':xs) fs v us rs cs es ms =
    let (ys,zs,var)=formulaSC(xs) in
        evalR zs fs (valoraE ys var fs v us rs cs es ms)
        us rs cs es ms
evalR (x:xs) fs v us rs cs es ms =
    let (y,m,zs) = formulaRel(xs) in
        evalR zs fs (valorRel x (head y) m us rs cs es ms)
        us rs cs es ms
```

Figure 1. The `eval` function implemented

where the following parameters appear:

- *form* is the fuzzy FOL formula. Fuzzy first order logic is a natural formalization of the system properties. Its syntax is the same as first order logic (FOL)[6], where we define the following translation between FOL formulas and formulas accepted in the system (we shall be able to write any fuzzy formula in our system by applying this translation mechanism):

Definition 3.1. Let φ be a FOL formula. Its syntactic translated formula $\bar{\varphi}$ is inductively defined as follows:

- Predicate symbols P have their syntactic counterpart `P` in the system.
- $\neg\varphi$ is translated into `~` $\bar{\varphi}$
- $\varphi \wedge \psi$ is translated int $\bar{\varphi}$ `&` $\bar{\psi}$
- The disjunction $\varphi \vee \psi$ is translated into $\bar{\varphi}$ `|` $\bar{\psi}$
- The implication $\varphi \Rightarrow \psi$ is translated into $\bar{\varphi}$ `>` $\bar{\psi}$
- The quantifications $\forall x.\varphi$ and $\exists x.\varphi$ are translated into `Ax.`$(\bar{\varphi})$ and `Ex.`$(\bar{\varphi})$ respectively.

- *semlist* is the semantics of the logic symbols used to evaluate the formula. The semantics in the fuzzy framework is defined as follows:

Definition 3.2. A 'fuzzy semantic' ***FS*** is a triple $(\wedge_f, \vee_f, \neg_f)$ with three consistent fuzzy operators for AND, OR and NEGATION. When necessary the tuple is extended to deal with the implication operator considering the tuple $(\wedge_f, \vee_f, \neg_f, \Rightarrow_f)$.

In our evaluation tool the fuzzy semantic is given as a list of pairs formed by a logic symbol (`~ & | >`) and the function associated to the corresponding symbol. For simplicity, we have considered in the program usual functions, like Zadeh's logic (`dmin,dmax,comp`) or Lukaszewicz's logic (`luka,comp`), but each user can introduce alternative functions. Lists in Haskell are written as sequences of elements between square brackets separated with commas (e.g., `[1,2,3,4]` is the list of naturales formed by such numbers).

- *univ* is universe of discourse of the logic. The universe is introduced as a list of characters not overlapping any other name neither variable nor predicate symbol. Such characters are written in Haskell between quotation marks. As we can only deal with finite lists, we restrict our universe of discourse to be a finite domain.

- *intlist* is the list of pairs of predicates and associated interpretation functions. This collection of predicates is inserted as a list of pairs formed

by the predicate symbol and the interpretation function.

- *numcharlist* is the list of numeric characteristic values. The numeric characteristic for each universe symbol is introduced as a list of triples `(S,P,V)`, where `S` is an element from the universe of discourse, `P` is a predicate symbol, and `V` is the numeric characteristic associated to `S` for the predicate symbol `P`. We use this numeric characteristic to obtain the interpretation of the element for the corresponding predicate.

- *cont* is the environment used to evaluate free variables. It is represented as a list of pairs formed by the variable and the value within a bounded universe. In almost every execution this environment is empty, i.e., there are no free variables.

- *modlist* is the list of modifiers of the logic. The list consists of pairs formed by the modifier symbol and its interpretation function. This modifier symbols are not defined in the program, but by the user avoiding to crash with any other previous symbol.

4. Example

Let us consider three individuals for this case named *John, Michael, Ann*, to be observed under the predicates *tall, short* and *old, young*. Lets for example assume that their respective heights are 1.7, 1.9, 1.65 meters and that their respective ages are 18, 20 and 35 years. The interpretation function for these values can be seen in the table below.

	HEIGHT	tall	short	AGE	young	old
John	1.7	0.4	0.6	18	1	0
Michael	1.9	1	0	20	0.9	0.1
Ann	1.65	0.3	0.7	35	0.6	0.4

Lets then evaluate some specific formulas with our tool:

- *Michael is neither very old nor very young*: this property is specified as

$$\neg(P \uparrow (M) \wedge Q \uparrow (M))$$

where in this case M represents *Michael*, P and Q are the predicates *old* and *young* and the $\uparrow$ represents the syntactic modifier *very*. After performing the syntactic translation we obtain the following formula valid in the system:

```
~(P+(M)&Q+(M))
```

where + represents the modifier ↑. To evaluate the formula in our tool we have to give an element in the list of modifiers and its associated function. The call to the system in this case is the following:

```
evaluate "~(P+(M))&(Q+(M))" [('&',luka),('~',comp)]
['M'] [('P',young),('Q',old)] [('M','P',22),('M','Q',22)]
[] [('+',sqr)]
```

where `sqr` is the predefined function to calculate the square of a given number. The formula is evaluated in the system using the Lukaszewicz's logic together with the square function as interpretation of the modifier symbol for predicates.

- *Everybody is very tall and very young*: this property is formalized as follows.

$$\forall x.(P \uparrow (x) \wedge Q \uparrow (x))$$

where P and Q represents the predicates tall and young. And after the translation process we obtain the expression:

```
Ax.(P+(x)&Q+(x))
```

To ask the tool for the value of the expression, we need to give the universe of discourse as the list of elements `A, B, C`. The environment is again empty, and the expression introduced to the system will be

```
evaluate "Ax.((P+(x))&(Q+(x)))" [('&',dmin),('~',comp)]
 ['J','M','A'] [('P',tall),('Q',young)]
 [('J','P',1.7),('J','Q',18),,('M','P',1.9),('M','Q',22),
 ('A','P',1.65),('A','Q',35)] [] [('+',sqr)]
```

The semantics used to evaluate this expression is the Zadeh's logic and for the universal quantification we can use the aggregation of the conjunction of every instance of the formula, substituting the variable with a universal symbol (see table below with the values for expressions in this example).

Example	Zadeh	Product	Lukaszewicz
1	0.4	0.4	0.4
2	0.19	0.1881	0.18
3	0.09	0.0042	0

5. Conclusions and future work

We have presented in this paper a functional tool for the evaluation of fuzzy first order formulas. This tool is useful to assist an expert to make decisions about the convenience of different semantics. The functional programming language we chose, Haskell, allows to consider functions as parameters, producing a general but efficient tool. Nevertheless, it is still needed that the user writes down the whole expression to be evaluated.

In order to make this tool more friendly to users, we plan to improve it by embedding the system into another language with graphic interface capabilities. We also plan to extend the above tool to carry out the progress of the program, in terms of the involved logic formulas, so the user can obtain at any time the evaluation of the formula transformed by the program instruction.

Acknowledgments This Research has been partially supported by grants MTM2005-08982 and TIN2005-09207 from the Government of Spain.

References

1. B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci and Ph. Shnobelen. *Systems and Software verification: model-checking techniques and tools*, Springer, 2001.
2. R. Bird. *Introduction to functional programming using Haskell*, Prentice Hall, 1998.
3. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*, Springer, 2004.
4. J.M. Cleva, J. Leach and F.J.López-Fraguas. *A logic programming approach to the verification of functional-logic programs*, Proc. Principles and Practice of Declarative Programming (PPDP'04), ACM Press, 2004, pp. 9–19.
5. P. Cousot and R. Cousot. *Refining model checking by abstract interpretation*, Automated Software Engineering Journal 6:69–95, 1999.
6. H.B. Enderton. *A Mathematical Introduction to Logic*, Academic Press, 2001.
7. M.J.C. Gordon and T.F. Melham. *Introduction to HOL*, Cambridge Univ. Press, 1993.
8. C.A.R. Hoare. *An axiomatic basis for computer programming*, Comm. ACM 12:89–100, 1969.
9. V. López, J. Montero y L. Garmendia. *Fuzzy specification of algorithms*, Technical Report (www.mat.ucm.es/ fuzzycs/fsa.pdf).
10. T.Nipkow, L.C. Paulson and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higer-Order Logic*, Springer, 2002.
11. S. Owre, N. Shankar, J.M. Rushby and D.W.J. Stringer-Calvert. *PVS System Guide*, SRI International, 2001.
12. G.J. Myers. *The Art of Software Testing*, John Wiley, 1979.