Bousi∼Prolog: a Prolog extension language for flexible query answering ⋆

Pascual Julián-Iranzo[1] Clemente Rubio-Manzano[2]
Juan Gallardo-Casero[3]

Dep. of Information Technologies and Systems,
University of Castilla-La Mancha,
Spain.

Abstract

In this paper we present the main features an implementation details of a programming language that we call Bousi∼Prolog. It can be seen as an extension of Prolog able to deal with similarity-based fuzzy unification ("Bousi" is the Spanish acronym for "fuzzy unification by similarity"). The main goal is the implementation of a declarative programming language well suited for flexible query answering.
The operational semantics of Bousi∼Prolog is an adaptation of the SLD resolution principle where classical unification has been replaced by an algorithm based on similarity relations defined on a syntactic domain. A similarity relation is an extension of the crisp notion of equivalence relation and it can be useful in any context where the concept of equality must be weakened. Hence, the syntax of Bousi∼Prolog is an extension of the Prolog's language: in general, a Bousi∼Prolog program is a set of Prolog clauses plus a set of similarity equations.

Keywords: Fuzzy Logic Programming, Fuzzy Prolog, Unification by Similarity, Weak SLD Resolution.

1 Introduction

Fuzzy Logic Programming integrates fuzzy logic and pure logic programming in order to provide these languages with the ability of dealing with uncertainty and approximate reasoning. There is no common method for this integration (See for instance: [7,8,12,2,15] and [20]; as well as [3,4,5,6] and [18]). A possible way to go, if we want to grapple with the issue of flexible query answering [4], is to follow the conceptual approach introduced in [18] where the notion of "approximation" is managed at a syntactic level by means of similarity relations. A similarity relation is an extension of the crisp notion of equivalence relation and it can be useful in any

⋆ This work has been partially supported by FEDER and the Spanish Science and Education Ministry (MEC) under grants TIN 2004-07943-C04-03 and TIN 2007-65749.

[1] Email: Pascual.Julian@uclm.es

[2] Email: Clemente.Rubio@alu.uclm.es

[3] Email: Juan.Gallardo@alu.uclm.es

[4] If you are pursuing a different objective, other approaches are preferable.

context where the concept of equality must be weakened. In [18] a new modified version of the **L**inear resolution strategy with **S**election function for **D**efinite clauses (SLD resolution) is defined, which is named *similarity-based* SLD resolution (or *weak* SLD resolution —WSLD—). This operational mechanism can be seen as a variant of the SLD resolution procedure where the classical unification algorithm has been replaced by the weak unification algorithm formally described in [18] (and reformulate in terms of a transition system in [10]). Informally, Maria Sessa's weak unification algorithm states that two terms $f(t_1, \ldots, t_n)$ and $g(s_1, \ldots, s_n)$ weak unify if the root symbols f and g are considered similar and each of their arguments t_i and s_i weak unify. Therefore, the weak unification algorithm does not produce a failure when there is a clash of two syntactical distinct symbols whenever they are similar.

In this paper we present the main features an implementation details of a programming language that we call Bousi∼Prolog (BPL for short), with an operational semantics based on the weak SLD resolution principle of [18]. Hence, Bousi∼Prolog computes answers as well as approximation degrees. Essentially, the Bousi∼Prolog syntax is just the Prolog syntax but enriched with a built-in symbol "∼∼" used for describing similarity relations by means of *similarity equations* of the form:

```
<alphabet symbol> ~~ <alphabet symbol> = <similarity degree>.
```

Although, formally, a similarity equation represents an arbitrary fuzzy binary relation, its intuitive reading is that two constants, n-ary function symbols or n-ary predicate symbols are similar with a certain degree. Informally, we use the built-in symbol "∼∼" as a compressed notation for the symmetric closure of an arbitrary fuzzy binary relation (that is, a similarity equation $a \sim\sim b = \alpha$ can be understood in both directions: a is similar to b and b is similar to a with degree α). Therefore, a Bousi∼Prolog program is a sequence of Prolog facts and rules followed by a sequence of similarity equations.

The structure of the paper is as follows. Some motivating examples are given in Section 2. The examples serve to introduce syntactical aspects of the Bousi∼Prolog language as well as to sustain the usefulness of the proposal. Section 3 presents the Bousi∼Prolog system structure, briefly describing its main components. Section 4, after recalling the definition of a similarity relation, gives some insight about its internal representation and how it is computed. The rest of this section is devoted to the implementation of the weak unification algorithm, which is the basis of the similarity-based SLD resolution principle. Section 5 presents the formal definition of Sessa's Weak SLD resolution principle and details of its concrete implementation in our system. In Section 6, information about distinct classes of cuts and negations are given. Section 7 discusses the relation of our work to other research lines on fuzzy logic programming. Finally, in Section 8 we give our conclusions and some lines of future research.

In the following, we assume some familiarity on the basic concepts around the field of logic programming [1].

2 Motivating examples

Our first example serves to illustrate BPL syntax as well as some features of its operational behavior in a very simple context.

Example 2.1 Consider the program `Autumn` that consists of the following clauses and similarity equations:

```
% FACTS        % RULES               % SIMILARITY EQUATIONS
autumn.        warm :- summer.       spring ~~ autumn = 0.7
               warm :- sunny.        spring ~~ summer = 0.5
               rainy :- spring.      autumn ~~ winter = 0.5
               cold :- winter.
               happy :- warm.
```

In an standard Prolog system a query as "`?- happy`" fails, since we are specifying that it is `warm` if it is `summer` time (first rule) and, actually, it is `autumn`. Similarly, the query "`?- rainy`" fails also.

However, the BPL system is able to compute the following successful derivations [5] :

- $\langle \leftarrow happy, id, 1\rangle \Longrightarrow_{\text{WSLD}} \langle \leftarrow warm, id, 1\rangle \Longrightarrow_{\text{WSLD}} \langle \leftarrow summer, id, 1\rangle$

  $\Longrightarrow_{\text{WSLD}} \langle \Box, id, 0.5\rangle.$

  Here, the last step is possible because `summer` weak unifies with the fact `autumn`, since there is a transitive connection between `summer` and `autumn` with approximation degree 0.5 (the minimum of 0.7 and 0.5). Therefore, the system answers "`Yes, with approximation degree 0.5`".
- $\langle \leftarrow rainy, id, 1\rangle \Longrightarrow_{\text{WSLD}} \langle \leftarrow spring, id, 1\rangle \Longrightarrow_{\text{WSLD}} \langle \Box, id, 0.7\rangle.$

  In this case, the system answers "`Yes, with approximation degree 0.7`" because `spring` and `autumn` weak unify with approximation degree 0.7 and the last step is possible.

In general, the Bousi∼Prolog computes answers as well as approximation degrees which are the minimum of the approximation degrees obtained in each step.

The second example shows how Bousi∼Prolog is well suited for flexible query answering.

Example 2.2 This BPL program fragment specify features and preferences on books stored in a data base. The preferences are specified by means of similarity equations:

```
% FACTS
adventures(treasure_island).
adventures(the_call_of_the_wild).
mystery(murders_in_the_rue_morgue).
horror(dracula).
science_fiction(the_city_and_the_stars).
```

[5] The symbol "id" denotes the identity substitution and "□" the empty clause.

```
science_fiction(the_martian_chronicles).

% RULES
good(X) :- interesting(X).

% SIMILARITY EQUATIONS
adventures ~~ mystery = 0.5
adventures ~~ science_fiction = 0.8
adventures ~~ interesting = 0.9
mystery ~~ horror = 0.9
mystery ~~ science_fiction = 0.5
science_fiction ~~ horror = 0.5
```

When this program is loaded an internal procedure constructs a similarity relation (i.e. a reflexive, symmetric, transitive, fuzzy binary relation) on the syntactic domain of the program alphabet. Therefore, all kind of books considered as `interesting` are retrieved by the query "`BPL> sv good(X)`".

The third and last example shows how similarity equations can be used to obtain a clean separation between logic and control in a pattern matching program.

Example 2.3 The following program gives the number of occurrences of a pattern `[e1,e2]` in a list of elements, where `e1` must be `a` and `e2` may be `b` or `c`.

```
% SIMILARITY EQUATIONS
e1~~a=1.
e2~~b=1.
e2~~c=1.

% FACTS and RULES
search([ ],0).
search([X|R],N):-search1([X|R],N).

search1([ ],0).
search1([X|R],N):-X~~e1 -> search2(R,N);
                           search1(R,N).
search2([ ],0).
search2([X|R],N):-X~~e2 -> search(R,N1),N is N1+1;
                           search(R,N).

occurrences(N):-search([a,b,c,a,c,b,d,a,c,d,b,b,a,b,c,c,a,c,a,b],N).
```

Here, "`~~`" is the weak unification operator and the expression "`X~~e1`" means that (the value bound to) "`X`" and "`e1`" weak unify with approximation degree greater than zero. Since the programmer wrote the similarity equation "`e1~~a=1`" in the program, the expression will success when `X` will be instantiated to "a". The same can be said for the expression "`X~~e1`".

It is easy to adapt the former program permitting the search of more complex combinations of patterns. For instance, introducing the similarity equation:

e1~~b=1.

In order to reach our goal, in this case, it is mandatory not to generate the transitive closure of the fuzzy relation defined by the set of similarity equations. This can be done by means of the BPL directive ":- transitivity(no), which inhibits the construction of the transitive closure, during the translating phase. The idea is to avoid the ascription of "e1" and "e2" to the same equivalence class, what will be a problem for the intended behavior of the new program.

Summarizing, the following program will count the number of occurrences of a pattern [e1,e2] in a list of elements, where e1 may be a or b and e2 may be b or c.

```
% BPL DIRECTIVE
:- transitivity(no)

% SIMILARITY EQUATIONS
e1~~a=1.
e1~~b=1.
e2~~b=1.
e2~~c=1.

% FACTS and RULES
search([ ],0).
search([X|R],N):-search1([X|R],N).

search1([ ],0).
search1([X|R],N):-X~~e1 -> search2(R,N);
                            search1(R,N).
search2([ ],0).
search2([X|R],N):-X~~e2 -> search(R,N1),N is N1+1;
                            search(R,N).

occurrences(N):-search([a,b,c,a,c,b,d,a,c,d,b,b,a,b,c,c,a,c,a,b],N).
```

3 Bousi∼Prolog Structure

The Bousi∼Prolog system we are presenting is a prototype, high level implementation written on top of SWI-Prolog [21] and is publicly available[6]. The complete implementation consists of about 900 lines of code. Figure 1 shows the structure of the BPL system through a functional dependency graph.

The bousi module contains the bpl_shell/0 main predicate which implements a command shell. Hence, providing the interface for the user. The relevants command are:

- ld -> (load) reads a file containing the source program for loading;
- lt -> (list) displays the current loaded program;

[6] The prototype implementation of the Bousi∼Prolog system can be found at the URL address http://www.inf-cr.uclm.es/www/pjulian/bousi.html.

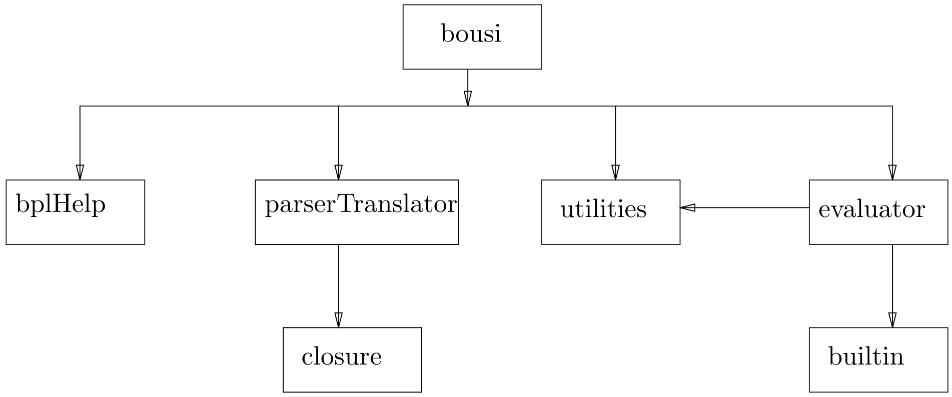

Fig. 1. Functional dependency graph of the Bousi∼Prolog system

- `sv -> (solve)` solves a (possibly conjunctive) query;
- `lc -> (lambda-cut)` reads or sets the lower bound for the approximation degree in a weak unification process (see later for a more detailed explanation of this feature).

The rest of commands are implemented as interface to the (unix) system environment.

The `bplHelp` module provides on-line explanation about the syntax of the commands and how they work.

The `parserTranslator` module contains the `parseTranslate/2` predicate This predicate parses a BPL InputFile and translates (compiles) it into an OutputFile which contains an intermediate Prolog representation of the source BPL code. The intermediate Prolog code is called "TPL code" (Translated BPL code). The parser phase is delegated to standard Prolog predicates. This is an imperfect solution because we lost the control of the whole parsing process and it imposes some real limitations [7]. However this is the cheapest solution. The improvement of the parser phase is let for future work.

The `evaluator` module implements the weak unification algorithm and the weak SLD resolution principle, which is the operational semantics of the language. Weak SLD resolution is implemented by means of a meta-interpreter [19]. The next two sections are devoted to precise the details of this implementation. The `evaluator` module uses the `builtin` module, which contains a relation of predicates which are sent directly to the SWI-Prolog interpreter.

The `utillities` module contains a repository of predicates used by other modules.

A schematic overview of the translation, load and execution of BPL programs is shown in Figure 2. In this figure, boxes denote different components of the system and names in boldface denote (intermedite) files. The source code of the BPL program must be stored in a file with the suffix "`.bpl`" (e.g., `prog.bpl`). The `parserTranslator` parses the BPL source file and translates (compiles) it into an intermediate Prolog representation of the source BPL code, which is stored in a file with the suffix "`.tpl`". Finally, the clauses in the TPL file are loaded into the

[7] For instance, we cannot use operators defined by the user, that is the "`:- op(_, _, _)`" directive.

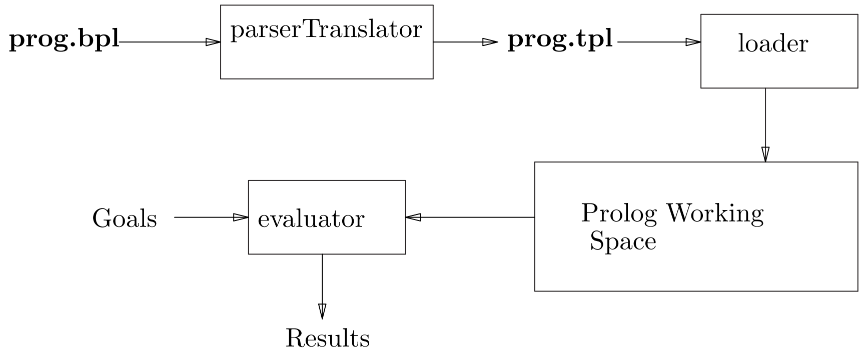

Fig. 2. Flow diagram overview of the Bousi~Prolog system

Prolog workspace. Then, the system is ready to admit queries which are solved by the `evaluator` meta-interpreter.

4 Similarity Equations and Weak Unification

The weak unification algorithm operates on the basis of a similarity relation. A *similarity relation* on a set U is a fuzzy binary relation on $U \times U$, that is, a mapping $\mathcal{R} : U \times U \rightarrow [0,1]$, holding the following properties: reflexive; symmetric and transitive. In this context, "transitive" means that $\mathcal{R}(x,z) \geq \mathcal{R}(x,y) \triangle \mathcal{R}(y,z)$ for any $x, y, z \in U$; where the operator '$\triangle$' is an arbitrary t-norm. Following [18], in the sequel, we restrict ourselves to similarity relations on a syntactic domain where the operator $\triangle = \wedge$ (that is, it is the minimum of two elements).

Similarity equations of the form "<symbol> ~~ <symbol> = <degree>" are used to represent an arbitrary fuzzy binary relation $\mathcal{R}$. A similarity equation $a \sim\sim b = \alpha$ is representing the entry $\mathcal{R}(a,b) = \alpha$. Internally, a similarity equation like the last one is coded as: `sim(a, b,` α`)`.

The user supplies an initial subset of similarity equations and then, the system automatically generates a reflexive, symmetric, transitive closure to obtain, by default, a similarity relation. However, if the BPL directive "`:- transitivity(no)`" is included at the beginning of a BPL program, only the reflexive, symmetric closure is computed. Therefore, a similarity equation $a \sim\sim b = \alpha$ can be understood in both directions: a is similar to b and b is similar to a with degree α.

An foreign predicate, `closure/3`, written in the C programming language [11], implements the algorithm for the construction of the similarity relation. This algorithm, has three steps. The first step computes the reflexive closure of the initial relation; the second the symmetric closure. The third step is an extension of the well-known Warshall's algorithm for computing the transitive closure of a binary relation, where the classical *meet* and *joint* operators on the set $\{0,1\}$ have been changed by the *maximum* (`MAX`) and the *minimum* (`MIN`) operators on the real interval $[0,1]$ respectively:

```
for(k = 0; k < nTotal; k++) {
    for(i = 0; i < nTotal; i++) {
        for(j = 0; j < nTotal; j++) {
            dMatriz[i][j] = MAX(dMatriz[i][j],
```

```
                              MIN(dMatriz[i][k], dMatriz[k][j]));
        }
    }
}
```

Here, initially, `dMatrix` is the adjacency matrix representing the reflexive, symmetric closure of the original fuzzy binary relation on a syntactic set. An interesting property of this algorithm is that it preserves the approximation degrees provided by the programmer in the similarity equations. See [9] for more details about the construction of a similarity relation. How to link a foreign predicate into the Prolog environment is explained in the SWI-Prolog reference manual [21].

The specific weak unification algorithm is implemented following closely Martelli and Montanari's unification algorithm for syntactic unification [14], but as usual in Prolog systems we do not use occur check:

```
% Term decomposition
unify(T1,T2,D) :- compound(T1), compound(T2), !,
                  functor(T1, F1, Aridad1),
                  functor(T2, F2, Aridad2),
                  Aridad1 =:= Aridad2,
                  sim(F1, F2, D1),
                  T1 =.. [F1| ArgsT1],
                  T2 =.. [F2| ArgsT2],
                  unifyArgs(ArgsT1, ArgsT2, D2), min(D1, D2, D).

unify(C1, C2, D) :- atomic(C1), atomic(C2), !, sim(C1, C2, D).

% Swap
unify(T,X, D) :- nonvar(T), var(X), !, unify(X,T, D).

% Variable elimination
unify(X,T, 1) :- var(X), X = T.
```

The predicate `unifyArgs(ArgsT1, ArgsT2, D)` checks if the terms (arguments) in the lists ArgsT1 and ArgsT2 can unify one with each other, obtaining a certain approximation degree `D`.

In order to understand the behavior of the predicate `unify/3`, the following comments are useful:

- As stated by the first clause defining the predicate `unify/3` the weak unification algorithm does not produce a failure when there is a clash of two syntactical distinct symbols `F1` and `F2` whenever they are similar. That is, the goal `sim(F1, F2, D1)` success with approximation degree `D1`, because there exists a similarity equation linking `F1` and `F2`.
- The third clause defining the predicate `unify/3` is the point where variables are instantiated, generating the bindings of the weak most general unifier.

Hence, this algorithm provides a weak most general unifier as well as a numerical value, called the *unification degree* in [18]. Intuitively, the unification degree will represent the truth degree associated with the (query) computed instance.

Bousi∼Prolog implements a weak unification operator, denoted by "∼∼", which is the fuzzy counterpart of the syntactical unification operator "=" of standard Prolog. It can be used, in the source language, to construct expressions like "Term1 ~~ Term2 =:= Degree" which is interpreted as follows: The expression is true if Term1 and Term2 are unifiable by similarity with approximation degree AD equal to Degree. In general, we can construct expressions "Term1 ~~ Term2 <op> Degree" where "<op>" is a comparison arithmetic operator (that is, an operator in the set {=:=, =\=, >, <, >=, =<}). Observe that the expression "Term1 ~~ Term2" is syntactic sugar of "Term1 ~~ Term2 > 0". Also it is possible the following construction: Term1 ~~ Term2 = Degree which success if Term1 and Term2 are weak unifiable with approximation degree Degree; otherwise fails. When Degree is a variable it is bound to the unification degree of Term1 and Term2. These expressions may be introduced in a query as well as in the body of a clause.

Example 4.1 Assume that the BPL program of Example 2.2 is load. The following is a simple session with the BPL system:

```
BPL> sv adventures(X) ~~ interesting(Y) > 0.5
With approximation degree: 1
X = _G1248
Y = _G1248

Yes

BPL> sv adventures ~~ mystery
With approximation degree: 1

Yes
```

Both goals success with approximation degree 1 because: adventures(X) and interesting(Y) weak unify with unification degree 0.9, greater than 0.5; adventures and mystery trivially weak unify with unification degree 0.5, greater than 0; and the comparison operator is a crisp one.

```
BPL> sv adventures(X) ~~ mystery(Y) = D
With approximation degree: 1
X = _G1714
Y = _G1714
D = 0.5;

No answers
```

This goal success with approximation degree 1 because it is completely true that adventures(X) and mystery(Y) weak unify with unification degree 0.5. There are not more answers since only a weak unifier representative is returned.

Note that the last goal is equivalent to the following one:

```
BPL>  sv unify(adventures(X), mystery(Y), D)
```

```
With approximation degree: 1
X = _G2522
Y = _G2522
D = 0.5

Yes
```

Finally observe that Bousi∼Prolog also provides the standard syntactic unification operator "=". The operator symbol "=" is overloaded and it can be used in different contexts with different meanings: i) it behaves as an identity when it is used inside a similarity equation or inside the construction ""; ii) it behaves as the syntactic unification operator when it is used dissociated of the weak unification operator "~~".

5 Operational Semantics

Let Π be a set of Horn clauses and $\mathcal{R}$ a similarity relation on the first order alphabet induced by Π. We define *Weak SLD* (WSLD) *resolution* as a transition system $\langle E, \Longrightarrow_{\text{WSLD}}\rangle$ where E is a set of triples $\langle \mathcal{G}, \theta, \alpha\rangle$ (goal, substitution, approximation degree), that we call the *state* of a computation, and whose transition relation $\Longrightarrow_{\text{WSLD}} \subseteq (E \times E)$ is the smallest relation which satisfies:

$$\frac{\mathcal{C} = (\mathcal{A} \leftarrow \mathcal{Q}) \ll \Pi, \sigma = wmgu(\mathcal{A},\mathcal{A}') \neq fail,\ \lambda = \mathcal{R}(\sigma(\mathcal{A}),\sigma(\mathcal{A}'))}{\langle(\leftarrow \mathcal{A}',\mathcal{Q}'), \theta, \alpha\rangle \ \Longrightarrow_{\text{WSLD}}\ \langle \leftarrow \sigma(\mathcal{Q}, \mathcal{Q}'), \sigma \circ \theta, \lambda \wedge \alpha\rangle}$$

where $\mathcal{Q}$, $\mathcal{Q}'$ are conjunctions of atoms and the notation "$\mathcal{C} \ll \Pi$" is representing that $\mathcal{C}$ is a standardized apart clause in Π.

A WSLD derivation for $\Pi \cup \{\mathcal{G}_0\}$ is a sequence of steps

$$\langle \mathcal{G}_0, id, 1\rangle \Longrightarrow_{\text{WSLD}} \ldots \Longrightarrow_{\text{WSLD}} \langle \mathcal{G}_n, \theta_n, \lambda_n\rangle.$$

And a WSLD refutation is a WSLD derivation $\langle \mathcal{G}_0, id, 1\rangle \Longrightarrow_{\text{WSLD}}{}^{*} \langle \Box, \sigma, \lambda\rangle$, where σ is a computed answer and λ is its *approximation degree*. Certainly, a WSLD refutation computes a family of answers, in the sense that, if $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ then whatever substitution $\theta' = \{x_1/s_1, \ldots, x_n/s_n\}$, holding that $s_i \equiv_{\mathcal{R},\lambda} t_i$ (i.e., $\mathcal{R}(s_i, t_i) \geq \lambda$), for any $1 \leq i \leq n$, is also a computed answer with approximation degree λ. However, in practice, we only return a representative of the family of answers.

As it was commented in Section 3, the `parseTranslate/2` predicate of the `parserTranslator` module translates (compiles) rules and facts of the source BPL code into an intermediate Prolog code representation which is called "TPL code" (Translated BPL code). More precisely, a rule "`Head :- Body`" is translated to "`rule(Head, Body)`" and a fact "`Head`" to "`rule(Head, true)`"

A meta-interpreter executes the BPL code according to the WSLD resolution principle. Figure 3 shows the implementation of the meta-interpreter.

The following clauses are the core of the WSLD resolution principle implementation:

```
solve(true,1):- !.
```

```
% solve(Goal): solve Goal giving a computer answer
% and its approximation degree.
solve(Goal) :- solve(Goal, Degree),
        write('With approximation degree: '),
        write(Degree),
        nl.


% solve(Goal, Degree): true if there is a refutation
% for 'Goal' with approximation degree 'Degree'.
solve(true,1):- !.
% Crisp Negation As Failure
solve(\+(A), D) :- !, (solve(A, DA) -> (DA = 1 -> fail;
                                                  D = 1);
                                       D = 1).
solve((A,B), D) :- !,
                   solve(A, DA),
                   solve(B, DB),
                   min(DA, DB, D).
solve((C -> A), D):- !, (solve(C, DC) ->
    solve(A, DA), min(DC, DA, D)).
solve((C -> A;B), D):- !, (solve(C, DC)  ->
    solve(A, DA), min(DC, DA, D) ;
            solve(B, DB), D = DB).
solve((A;B), D) :- !, (solve(A, DA), D = DA ;
        solve(B, DB), D = DB).
% Weak Negation As Failure
solve(not(A), D) :- !, (solve(A, DA) -> (DA = 1 -> fail;
                                                   D is 1 - DA);
                                        D = 1).
solve(A, 1) :- built(A), !, call(A).
solve(A, D) :- rule(H,B),
        unify(A, H, AD),
        lambdaCut(L),
        AD >= L,
        solve(B, DB),
        min(AD, DB, D).
```

Fig. 3. A meta-interpreter for executing BPL code

```
solve((A,B), D) :- !,
                   solve(A, DA),
                   solve(B, DB),
                   min(DA, DB, D).
solve(A, D) :- rule(H,B),
       unify(A, H, AD),
       lambdaCut(L),
       AD >= L,
       solve(B, DB),
       min(AD, DB, D).
```

This clauses asserts that:

- The goal `true` is solved with approximation degree 1.
- In order to solve a conjunctive goal `(A, B)` firts solve the atom `A`, obtaining an approximation degree `DA`, and then the remaining conjunctive goal `B`, obtaining an approximation degree `DB`. The approximation degree of the hole conjunctive goal is the minimum of `DA` and `DB`.
- In order to solve the atom `A`, select a rule whose head `H` and `A` weak unify with approximation degree `AD`. If `AD` is greater or equal than the current `LambdaCut` value `L` (see below), solve the body `B` of the rule, obtaining an approximation degree `DB`. Then, the approximation degree of the goal is the minimum of `AD` and `DB`.

6 Distinct Classes of Cuts and Negations

We can impose a limit to the expansion of the search space in a computation by what we called a "lambda-cut". When the `LambdaCut` flag is set to a value different to zero, the weak unification process fails if the computed approximation degree goes below the stored `LambdaCut` value. Therefore, the computation also fails and all possible branches starting from that choice point are discarded. By default the `LambdaCut` value is zero (that is, no restriction to a computation is imposed). However, the `LambdaCut` flag can be set to a different value by means of a `lambdaCut` directive introduced inside of a BPL program or the `lc` command of the BPL shell. The `lc` command can be used to show which is the current `Lambdacut` value or to set a new `Lambdacut` value.

Bousi∼Prolog can use the standard cut predicate, "`!`" of the Prolog language, but, in an indirect way, embedded into more declarative predicates and operators, such as: `not` (weak negation as failure —see below—), `\+` (crisp negation as failure —see below—) and `->` (if-then and if-then-else operators).

On the other hand Bousi∼Prolog provides an operator, "`\+`", for crisp negation as failure and a predicate "`not`" for weak negation as failure. The implementation of these distinct classes of Negations is as follows:

- A goal `\+(A)` fails only if `solve(A, DA)` successes with approximation degree `DA =1`. Otherwise `\+(A)` is true with approximation degree `1`. That is "`\+`" operates as the classical negation as failure.

```
% Crisp negation as failure
solve(\+(A), D) :- !, (solve(A, DA) -> (DA = 1 -> fail;
  D = 1);
D = 1).
```

- A goal `not(A)` fails only if `solve(A, DA)` success with approximation degree `DA =1`. When `solve(A, DA)` success, but the approximation degree `DA` is lesser than `1`, `not(A)` also success with approximation degree `D = 1 - DA`. If it is the case that `solve(A, DA)` fails, `not(A)` success with approximation degree `D = 1`.

```
% Weak negation as failure
solve(not(A), D) :- !, (solve(A, DA) -> (DA = 1 -> fail;
   D is 1 - DA);
D = 1).
```

7 Related Work

Several fuzzy extensions of the resolution rule [16], used in classical logic programming, with similarity relations have been proposed during the last decade. Although all these approaches relay in the replacement of the classical syntactic unification algorithm by a similarity-based unification algorithm, we can distinguish two main lines of research:

- The first one is represented by the theoretical works [5,6] and [4], where the concept of unification by similarity was first developed. However they use the cumbersome notions of *clouds*, *systems of clouds* and *closures operators* in its definition. From our point of view, these notions endangers the efficiency of the operational semantics which uses them, because they are costly to compute. The main practical realization of this line of work is the fuzzy logic language LIKELOG [3]: it is mainly implemented in Prolog using the aforementioned concepts and rather direct techniques.

- The second line of research is represented by the theoretical works [17] and [18], where the concept of weak unification was developed. The proposed algorithm is a clean extension of the Martelli and Montanari's unification algorithm for syntactic unification [14]. From our point of view, the weak unification algorithm is better suited for computing. As it was commented, the combination of the weak unification algorithm with the SLD resolution rule produces the weak SLD operational semantics we use in our Bousi∼Prolog implementation. In [13], an implementation based the weak SLD operational semantics is refered.

Despite the interest of systems like LIKELOG and the one described in [13], less implementations details are provided. Also, at the best of our knowledge, the implementation of these systems are not publicly available and, therefore, it is difficult an experimental comparison with our system.

8 Conclusions and Further Research

In this paper we present the main features an implementation details of a programming language that we call Bousi∼Prolog ("Bousi" is the spanish acronym for "fuzzy unification by similarity"). It can be seen as an extension of Prolog which incorporates similarity-based fuzzy unification, leading to a system well suited to be used for approximate reasoning and flexible query answering.

The so called weak unification algorithm [18] is based on similarity relations defined on a syntactic domain. At a syntactic level, Bousi∼Prolog represents similarity relations by means of similarity equations. The syntax of Bousi∼Prolog is an extension of the standard Prolog language: in general, a Bousi∼Prolog program is a set of Prolog clauses plus a set of similarity equations.

Bousi∼Prolog implements a weak unification operator, denoted by "∼∼", which is the fuzzy counterpart of the syntactical unification operator "=" of standard PrologṪhe weak unification operator can be included in a query or in the body of a rule.

The weak SLD resolution principle [18] used by Bousi∼Prolog as operational semantics, is implemented by means of a meta-interpreter. This is a cheap solution from the implementation point of view but expensive from the point of view of the efficient execution.

Although Bousi∼Prolog implements the main features of a standard Prolog other features, such as working with modules, are not covered. In the future we want to add these missing features to our language. Also we want to incorporate new non standard features and to improve certain modules of our system, such as the parser.

On the other hand, in order to solve the efficiency problem, we have investigated how to incorporate the weak unification algorithm into the Warren Abstract Machine. Some preliminary results for a pure subset of Prolog can be find in [10]. Also we want to develop this line of work to cover all the present and future features of Bousi∼Prolog in a more efficient implementation.

References

[1] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, Englewood Cliffs, NJ, 1997.

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

[3] Francesca Arcelli Fontana and Ferrante Formato. Likelog: A logic programming language for flexible data retrieval. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99), February 28 - March 2, 1999, San Antonio, Texas, USA*, pages 260–267, 1999.

[4] Francesca Arcelli Fontana and Ferrante Formato. A similarity-based resolution rule. *Int. J. Intell. Syst.*, 17(9):853–872, 2002.

[5] Ferrante Formato, Giangiacomo Gerla, and Maria I. Sessa. Extension of logic programming by similarity. In Maria Chiara Meo and Manuel Vilares Ferro, editors, *APPIA-GULP-PRODE*, pages 397–410, 1999.

[6] Ferrante Formato, Giangiacomo Gerla, and Maria I. Sessa. Similarity-based unification. *Fundam. Inform.*, 41(4):393–414, 2000.

[7] S. Guadarrama, S. Muñoz, and C. Vaucheret. Fuzzy Prolog: A new approach using soft constraints propagation. *Fuzzy Sets and Systems, Elsevier*, 144(1):127–150, 2004.

[8] M. Ishizuka and N. Kanai. Prolog-ELF Incorporating Fuzzy Logic. In Aravind K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI'85). Los Angeles, CA, August 1985.*, pages 701–703. Morgan Kaufmann, 1985.

[9] P. Julián-Iranzo. A procedure for the construction of a similarity relation. In M. Ojeda, editor, *In Proc. of the 12th International Conference on Information Processing and Management of Uncertainty in Knoledge-based Systems (IPMU 2008), June 22-27, 2008, Málaga, Spain*, page 8. U. Málaga, 2008. (Accepted for publication).

[10] P. Julián-Iranzo and C. Rubio-Manzano. A wam implementation for flexible query answering. In A.P. del Pobil, editor, *In Proc. of the 10th IASTED International Conference on Artificial Intelligence and Soft Computing (ASC 2006), August 28-30, 2006, Palma de Mallorca*, pages 262–267. ACTA Press, 2006.

[11] B.W. Kernighan and D.M. Ritchie. *The C Programming Language, 2nd Edition*. Prentice-Hall, 1988.

[12] R.C.T. Lee. Fuzzy Logic and the Resolution Principle. *Journal of the ACM*, 19(1):119–129, 1972.

[13] Vincenzo Loia, Sabrina Senatore, and Maria I. Sessa. Similarity-based sld rsolution and its implementation in an extended prolog system. In *FUZZ-IEEE*, pages 650–653, 2001.

[14] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[15] J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Similarity-based unification: a multi-adjoint approach. *Fuzzy Sets and Systems*, 146(1):43–62, 2004.

[16] J.A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[17] Maria I. Sessa. Flexible querying in deductive database. In A. Di Nola and G. Gerla, editors, *School on Soft Computing at Salerno University: Selected Lectures 1996-1999*, pages 257–276. Springer Verlag, 2000.

[18] Maria I. Sessa. Approximate reasoning by similarity-based sld resolution. *Theoretical Computer Science*, 275(1-2):389–426, 2002.

[19] L. Sterling and E. Shapiro. *The Art of Prolog (Second Edition)*. The MIT Press, Cambridge, MA, 1994.

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

[21] J. Wielemaker. SWI-Prolog 5.6 Reference Manual. Technical report: vesion 5.6.52, March 2008, University of Amsterdam, 2008.