Bousi∼Prolog: Design and Implementation of a Proximity-based Fuzzy Logic Programming Language

Pascual Julián-Iranzo[1] Fernando Sáenz-Pérez[2]

[1]Department of Information Technologies and Systems,
University of Castilla-La Mancha, Spain.
(**Pascual.Julian@uclm.es**)

[2]Department of Software Engineering and Artificial Intelligence,
Complutense University of Madrid, Spain.
(**fernan@sip.ucm.es**)

Outline

Outline

Fuzzy Logic Programming and Bousi Prolog

- **Fuzzy Logic Programming** = Logic Prog. + Fuzzy Logic
- **Bousi~Prolog** (BPL) is a fuzzy logic programming language whose main objective is to make flexible the query answering process.
- BPL is a conservative extension of Prolog, introducing as many fuzzy characteristics while maintaining the Prolog syntax.

Bousi Prolog features and syntax

- **One distinguished feature** of Bousi~Prolog is that it makes a separate treatment of Vague Knowledge.

  Algorithm = Logic + Vague Knowledge + Control.

- **Logic**: is specified by (possibly graded) facts and rules (most of which respect the Prolog syntax).
- **Vague Knowledge**: is specified by proximity equations (and/or directives defining fuzzy subsets).
- **Control**: is implemented by an operational semantics based on Weak SLD Resolution (= SLD Resolution + Weak Unification).

Bousi Prolog features and syntax

```
% FACTS
likes_teaching(john, physics).
likes_teaching(mary, chemistry).
has_degree(john, physics).
has_degree(mary, chemistry).

% RULE
can_teach(X,M):-has_degree(X, M), likes_teaching(X, M).
```

```
?- can_teach(X,math).
```

Bousi Prolog features and syntax

```
% FACTS
likes_teaching(john, physics).
likes_teaching(mary, chemistry).
has_degree(john, physics).
has_degree(mary, chemistry).
```

```
% PROXIMITY EQUATIONS
physics ~ math = 0.8.
physics ~ chemistry = 0.8.
chemistry ~ math = 0.6.
```

```
% RULE
can_teach(X,M):-has_degree(X, M), likes_teaching(X, M).
```

```
?- can_teach(X,math).
```

Bousi Prolog features and syntax

```
% FACTS
likes_teaching(john, physics) with 0.8.
likes_teaching(mary, chemistry) with 0.7.
has_degree(john, physics).
has_degree(mary, chemistry).

% PROXIMITY EQUATIONS
physics ~ math = 0.8.
physics ~ chemistry = 0.8.
chemistry ~ math = 0.6.

% RULE
can_teach(X,M):-has_degree(X, M), likes_teaching(X, M) with 0.9.
```

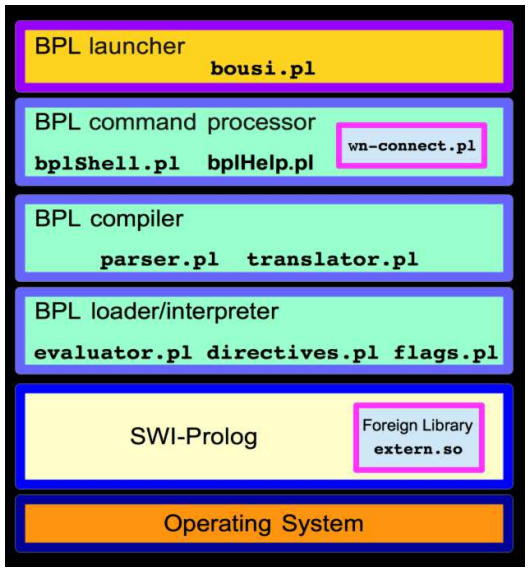
```
?- can_teach(X,math).
```

Aim of the Work

- **The goal of this work was to summarize our accumulated experience during the last decade in the development of the BPL system.**
- In this talk we focus our attention into the following main points:
  - The fundamentals of the BPL language.
  - How its operational semantics is implemented by translating a BPL program into Prolog clauses able to emulate WSLD-resolution.
  - Some special BPL implementation features, including those that make it more applicable.

Architecture of the Bousi∼Prolog system.

The Bousi∼Prolog system is composed of three subsystems which are integrated by a total of nine modules.



- The **bousi** module initializes the system.
- The **bplShell** module: command processing functionalities.
- The **parser** module: lexical, syntactic and semantic analysis of the BPL programs and queries.
- The **translator** module: translates the BPL source files and queries into TPL code. It relies on the **parser** module.
- The **evaluator** module: executes the TPL code. Implements the loader/interpreter of the BPL system.
- Modules with specific tasks: **bplHelp**, **directives**, **flags** and **foreign**.

Outline

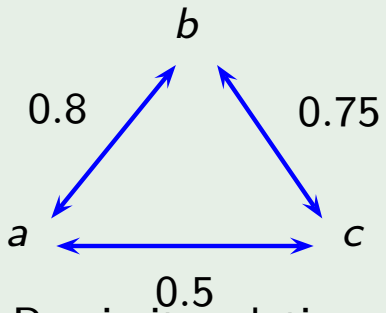
Proximity Relations

- A binary fuzzy relation $\mathcal{R}$ on U is a mapping $\mathcal{R} : U \times U \to [0, 1]$.
- Some important properties fuzzy relations may have:
  1. **(Reflexive)** $\mathcal{R}(x, x) = 1$ for any $x \in U$;
  2. **(Symmetric)** $\mathcal{R}(x, y) = \mathcal{R}(y, x)$ for any $x, y \in U$;
  3. **(Transitive)** $\mathcal{R}(x, z) \geq \mathcal{R}(x, y) \wedge \mathcal{R}(y, z)$ for any $x, y, z \in U$;
- Proximity relations are fuzzy binary relations holding the reflexive and symmetric properties.
- A proximity relation holding the transitive property is a similarity relation.

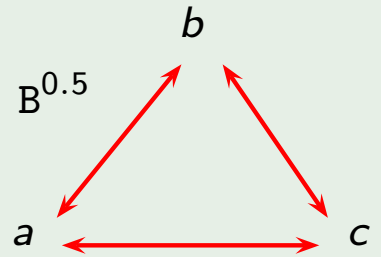
Proximity Blocks

- Given a proximity relation $\mathcal{R}$ on a set U, a λ-cut $\mathcal{R}_\lambda = \{\langle x, y\rangle \mid \mathcal{R}(x, y) \geq \lambda\}$
- Proximity block of level λ (or λ-block):
  - Given a proximity relation $\mathcal{R}$ on a set U,
  - is a subset of U such that the restriction of $\mathcal{R}_\lambda$ to this subset is a maximal total relation.

Example (1)



Proximity relation

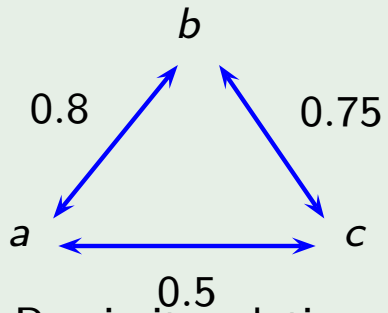


$B^{0.5} = \{a, b, c\}$

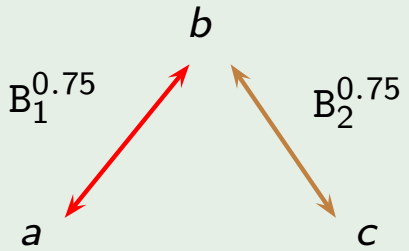
Proximity Blocks

- Given a proximity relation $\mathcal{R}$ on a set U, a λ-cut $\mathcal{R}_\lambda = \{\langle x, y\rangle \mid \mathcal{R}(x, y) \geq \lambda\}$
- Proximity block of level λ (or λ-block):
  - Given a proximity relation $\mathcal{R}$ on a set U,
  - is a subset of U such that the restriction of $\mathcal{R}_\lambda$ to this subset is a maximal total relation.

Example (1)



Proximity relation



$B_1^{0.75} = \{a, b\}$ $B_2^{0.75} = \{b, c\}$

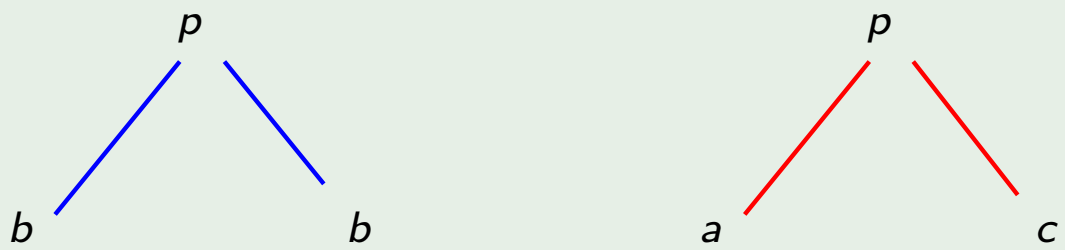
Proximity Relations on Syntactic Domains

- Proximity relations can be defined on the alphabet of a first order language and extended to terms and atomic formulas.
- Declarative notion of proximity: two expressions of a first-order language $\mathcal{L}$ are λ-approximate
  1. When their symbols, at their corresponding positions, belong to the same λ-block and
  2. A certain symbol is always assigned to the same λ-block (i.e., it is playing the same role)

Proximity Between Expressions

Example (2: Proximity between $A_1 \equiv p(b,b)$ and $A_2 \equiv p(a,c)$)

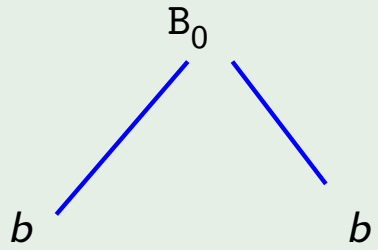
- Assume that $\mathcal{R} = \{\mathcal{R}(a,b) = 0.8, \mathcal{R}(b,c) = 0.75\}$,
- 0.75-blocks: $\mathrm{B}_0 = \{p\}$, $\mathrm{B}_1 = \{a,b\}$, $\mathrm{B}_2 = \{b,c\}$

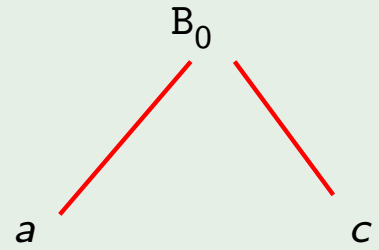


Proximity Between Expressions

Example (2: Proximity between $A_1 \equiv p(b,b)$ and $A_2 \equiv p(a,c)$)

- Assume that $\mathcal{R} = \{\mathcal{R}(a,b) = 0.8, \mathcal{R}(b,c) = 0.75\}$,
- 0.75-blocks: $\mathrm{B}_0 = \{p\}$, $\mathrm{B}_1 = \{a,b\}$, $\mathrm{B}_2 = \{b,c\}$

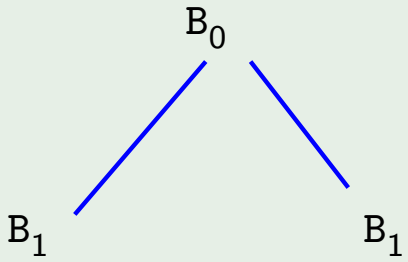


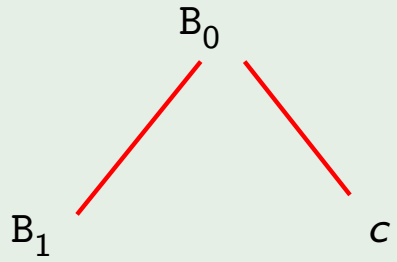


Proximity Between Expressions

Example (2: Proximity between $A_1 \equiv p(b,b)$ and $A_2 \equiv p(a,c)$)

- Assume that $\mathcal{R} = \{\mathcal{R}(a,b) = 0.8, \mathcal{R}(b,c) = 0.75\}$,
- 0.75-blocks: $\mathrm{B}_0 = \{p\}$, $\mathrm{B}_1 = \{a,b\}$, $\mathrm{B}_2 = \{b,c\}$

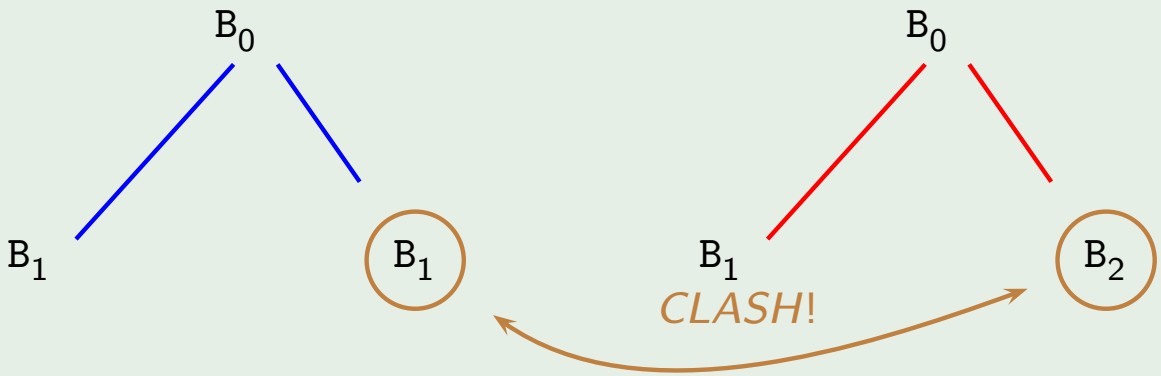




Proximity Between Expressions

Example (2: Proximity between $A_1 \equiv p(b, b)$ and $A_2 \equiv p(a, c)$)

- Assume that $\mathcal{R} = \{\mathcal{R}(a, b) = 0.8, \mathcal{R}(b, c) = 0.75\}$,
- 0.75-blocks: $\mathtt{B}_0 = \{p\}$, $\mathtt{B}_1 = \{a, b\}$, $\mathtt{B}_2 = \{b, c\}$



- **The atoms A_1 and A_2 are not approximate.**

An Efficient Proximity-based Unification Algorithm

- Our weak unification algorithm relies on the notion of proximity just introduced.
- **The weak unification algorithm has three stages**.
- **Stage 1**: we analyze the proximity relation $\mathcal{R}$ extracting the set of proximity blocks (Bron-Kerbosch algorithm).
- **Stage 2**: we extend the proximity relation $\mathcal{R}$ into a new relation $\mathcal{RB}$, enhancing $\mathcal{R}$ with specific λ-block information.
- These two previous steps are implemented by a foreign predicate coded in C (SWI-Prolog Foreign Language Interface).
- **Stages 1 and 2 are done at compile time !!**

An Efficient Proximity-based Unification Algorithm

- **Stage 3**: weak unification, formalized by a transition system. It is characterized by a notion of unification state and a set of transition rules.
- A weak unification state is a tuple $\langle P, S, C, \alpha \rangle$ where:
  1. P is a (multi-)set of weak unification problems or failure;
  2. S is a set of equations in solved form;
  3. C is a set of block constraints of level λ;
  4. α is a unification degree.

An Efficient Proximity-based Unification Algorithm

- A block constraint is an ordered pair that links a symbol with a proximity λ-block label. We denote these constraints as bindings "$<$ `symbol`$>$:$<$ λ-`block_label`$>$" .
- Block constraints of level λ are used to detect inconsistencies in "block assignments" for an alphabet symbol.
- A satisfaction function, $\mathcal{S}at$, is used for block constraint satisfaction.
  - Implement as a Prolog predicate, `sat/3`, which essentially performs a membership test on an association list **and can be done efficiently at runtime!!**

An Efficient Proximity-based Unification Algorithm

- A weak unification process is formalized as a sequence of transition steps performed using the proximity-based unification relation "$\Rightarrow$".
- The proximity-based unification relation, "$\Rightarrow$", is defined by a set of transition rules:
  Term decomposition:

(a) $\langle \{f(\overline{t_n}) \approx f(\overline{s_n})\} \cup E, S, C, \alpha\rangle \Rightarrow \langle \{\overline{t_n \approx s_n}\} \cup E, S, C, \alpha)\rangle,$

(b) $\langle \{f(\overline{t_n}) \approx g(\overline{s_n})\} \cup E, S, C, \alpha\rangle \Rightarrow$
$\langle \{\overline{t_n \approx s_n}\} \cup E, S, \{(f{:}\mathrm{B}^{\lambda}_{\mathcal{R}}), (g{:}\mathrm{B}^{\lambda}_{\mathcal{R}})\} \cup C, \alpha \triangle \beta\rangle,$
if $\mathcal{RB}(f, g, \mathrm{B}^{\lambda}_{\mathcal{R}}) = \beta \geq \lambda$ and
$\mathcal{S}at(\{(f{:}\mathrm{B}^{\lambda}_{\mathcal{R}}), (g{:}\mathrm{B}^{\lambda}_{\mathcal{R}})\}, C) \neq \mathit{failure}.$

where $\mathcal{RB}$ is the extension of $\mathcal{R}$ with block information.

An Efficient Proximity-based Unification Algorithm

- A weak unification process is formalized as a sequence of transition steps performed using the proximity-based unification relation "$\Rightarrow$".
- The proximity-based unification relation, "$\Rightarrow$", is defined by a set of transition rules:

  Failure rule:

  $$\langle \{f(\overline{t_n}) \approx g(\overline{s_m})\} \cup E, S, C, \alpha\rangle \Rightarrow \langle \mathit{fail}, S, C, \alpha\rangle,$$

  if $n \neq m$, $\mathcal{RB}(f, g, \mathrm{B}^{\lambda}_{\mathcal{R}}) < \lambda$ or $\mathcal{S}at(\{(f{:}\mathrm{B}^{\lambda}_{\mathcal{R}}), (g{:}\mathrm{B}^{\lambda}_{\mathcal{R}})\}, C) = \mathit{failure}$

  where $\mathcal{RB}$ is the extension of $\mathcal{R}$ with block information.

The Proximity-based Unification Algorithm in Action

Example (3: $A_1 \equiv p(b, b)$ and $A_2 \equiv p(a, c)$)

- $\mathcal{R}(a, b)$=0.8, $\mathcal{R}(b, c)$=0.75
- **Stage 1**: B_1=$\{a, b\}$, B_2=$\{b, c\}$
- **Stage 2**: $\mathcal{RB}(a, b, \mathrm{B}_1) = 0.8, \mathcal{RB}(b, c, \mathrm{B}_2) = 0.75, \ldots$
- **Stage 3**: **The atoms A_1 and A_2 do not weakly unify**.

$$\langle \{\underline{p(b,b) \approx p(a,c)}\}, id, \emptyset, 1\rangle$$
$$\Rightarrow_{1a} \langle \{\underline{b \approx a}, b \approx c\}, id, \emptyset, 1\rangle$$
$$\Rightarrow_{1b} \langle \{\underline{b \approx c}\}, id, \{(b{:}\,\mathrm{B}_1), (a{:}\,\mathrm{B}_1)\}, 0.8 \wedge 1\rangle$$
$$\Rightarrow_{5} \langle \mathit{failure}, id, \{(b{:}\,\mathrm{B}_1), (a{:}\,\mathrm{B}_1)\}, 0.8\rangle$$

An Efficient Proximity-based Unification Algorithm: Some Implementation Details

- The proximity-based weak unification algorithm is implemented by the Prolog predicate `weak_unify_a3/6`.

```
weak_unify_a3(Term1, Term2, Lambda, Cin, Cout, Degree) :-
% Term decomposition
    compound(Term1), compound(Term2), !,
    Term1 =.. [Functor1|Args1], Term2 =.. [Functor2|Args2],
    length(Args1, Arity), length(Args2, Arity),
    (Functor1==Functor2
    → Cin1=Cin, DegreeFunctor=1.0
    ;
    sim(Functor1, Functor2, Block, DegreeFunctor),
    DegreeFunctor >= Lambda,
    sat_a3([Functor1:Block, Functor2:Block], Cin, Cin1),
    ),
    weak_unify_args_a3(Args1,Args2,Lambda, Cin1,Cout,DegreeArgs),
    t_norm_op(DegreeFunctor, DegreeArgs, Degree).
```

Weak SLD Resolution (WSLD) (of level λ)

- Let Π be a program, $\mathcal{R}$ be a proximity relation, $\triangle$ a fixed t-norm and a λ cut value.
- Weak SLD (WSLD) resolution is defined as a transition system $\langle E, \Rightarrow_{\text{WSLD}} \rangle$ where:
  - E is a set of tuples $\langle \mathcal{G}, \theta, \alpha, C \rangle$ (the state of a computation)
  - $\Rightarrow_{\text{WSLD}} \subseteq (E \times E)$ is the transition relation, defined as:

$$\langle (\leftarrow A' \wedge \mathcal{Q}'), \theta, \alpha, C \rangle \Rightarrow_{\text{WSLD}} \langle \leftarrow (\mathcal{Q} \wedge \mathcal{Q}')\sigma, \theta\sigma, \beta \triangle \alpha \triangle \mu, C' \cup C \rangle$$

if
1. $R \equiv (A \leftarrow \mathcal{Q} \text{ with } \mu) \ll \Pi$,
2. $\text{wmgu}^{\lambda}_{\mathcal{R}}(A, A') = \langle \sigma, C', \beta \rangle$,
3. $\mathcal{S}at(C', C) \neq \textit{failure}$,
4. $(\beta \triangle \alpha \triangle \mu) \geq \lambda$.

Where β and μ are truth degrees (in $[0, 1]$), $\mathcal{Q}$ and $\mathcal{Q}'$ are conjunctions of atoms.

Weak SLD Resolution (WSLD) (of level λ)

- A **WSLD derivation** (of level λ) for $\Pi \cup \{\mathcal{G}_0\}$ is a sequence of WSLD resolution steps

$$\langle \mathcal{G}_0, id, 1, \emptyset \rangle \Rightarrow_{\mathsf{WSLD}} \langle \mathcal{G}_1, \theta_1, \alpha_1, C_1 \rangle \Rightarrow_{\mathsf{WSLD}} \ldots \Rightarrow_{\mathsf{WSLD}} \langle \mathcal{G}_n, \theta_n, \alpha_n, C_n \rangle$$

- **WSLD refutation** is a WSLD derivation (of level λ):

$$\langle \mathcal{G}, id, 1, \emptyset \rangle \Rightarrow_{\mathsf{WSLD}}{}^{*} \langle \square, \sigma, \alpha, C \rangle$$

- **output of the computation**: $\langle \sigma, \alpha \rangle$
  - $\sigma = \theta \restriction \mathcal{V}ar(\mathcal{G}_0)$ is a computed answer and α is its computed approximation degree.
- Block constraints are used to guarantee the consistency of the final answer (although it is not part of it).

WSLD Resolution: Implementation details

- Bousi~Prolog implements WSLD resolution by compiling BPL programs into a set of Prolog clauses that are able of emulating it.
- It uses a program translation that we call **BPL expansion**:
  1. Each BPL program rule is replaced by the set of rules which are approximate (w.r.t. $\mathcal{R}$) to the rule being transformed.
  2. The head of those approximate rules are linearised to facilitate the crisp unification of the defined predicate with a goal, while the weak unification of their arguments are carried out explicitly in the body of the transformed rules

WSLD Resolution: Implementation details

Definition (BPL expansion)

- Let $\mathcal{RB}$ be the extension of $\mathcal{R}$, $\triangle$ the fixed t-norm and $\lambda \in [0,1]$ a cut value.
- Let $p(t1,\ldots,t_n) \leftarrow \mathcal{Q}$ with δ be a graded rule in Π.

Then, for each entry $\mathcal{RB}(p,q,\mathrm{B}^{\lambda}_{\mathcal{R}}) = \alpha \geq \lambda$ add to the transformed program Π' the e-clause:

$$\langle q(x_1,\ldots,x_n) \leftarrow x_1 \approx t_1 \wedge \cdots \wedge x_n \approx t_n \wedge \mathcal{Q};\ (\delta \triangle \alpha);\ [p : \mathrm{B}^{\lambda}_{\mathcal{R}}, q : \mathrm{B}^{\lambda}_{\mathcal{R}}] \rangle$$

where each x_i is a fresh variable and $x_i \approx t_i$ forces weak unification, i.e, the evaluation of $\mathrm{wmgu}^{\lambda}_{\mathcal{R}}(x_i, t_i)$.

WSLD Resolution: Implementation details

Example (5)

% PROXIMITY EQUATIONS
p $\sim$ q = 0.9.
% FACTS RULE
p(a).

% PROXIMITY RELATION
$\mathcal{RB}$(p,q,0) = 0.9.
$\mathcal{RB}$(q,p,0) = 0.9.
% E-CLAUSES
<p(X1) :- X1$\approx$a; 1 ; []>
<q(X1) :- X1$\approx$a; 0.9; [(p,0), (q,0)]>

WSLD Resolution: Implementation details

Example (6)

% PROXIMITY EQUATIONS
a $\sim$ b = 0.7.
b $\sim$ c = 0.8.
p $\sim$ q = 0.9.
% FACTS RULE
p(X) :- r(X) with 0.75.
r(a).

% PROXIMITY RELATION
$\mathcal{RB}$(a,b,2)=0.7. $\mathcal{RB}$(c,b,1)=0.8.
$\mathcal{RB}$(b,a,2)=0.7. $\mathcal{RB}$(p,q,0)=0.9.
$\mathcal{RB}$(b,c,1)=0.8. $\mathcal{RB}$(q,p,0)=0.9.
% E-CLAUSES
<p(X1) :- X1$\approx$X, r(X); 0.75 ; []>
<q(X1) :- X1$\approx$X, r(X);
$0.9 \wedge 0.75$; [(p,0), (q,0)]>
<r(X1) :- X1$\approx$a; 1; []>

WSLD Resolution: Implementation details

Definition (operational semantics for expanded programs)

Defined as a transition system $\langle E, \Rightarrow_{\text{EXP}} \rangle$ where

- E is a set of tuples $\langle \mathcal{G}, \alpha, C \mid \theta \rangle$ (goal, approximation degree, block constraints, substitution),
- $\Rightarrow_{\text{EXP}} \subseteq (E \times E)$ is a transition relation which satisfies:

Rule 1: if $\text{wmgu}^{\lambda}_{\mathcal{R}}(A, B) = \langle \sigma, \beta, C' \rangle$, $\mathcal{S}at(C \cup C') \neq \textit{failure}$ and $(\beta \triangle \alpha) \geq \lambda$,

$$\langle (\leftarrow \underline{A \approx B} \wedge \mathcal{Q}), \alpha, C \mid \theta \rangle \Rightarrow_{\text{EXP}} \langle \leftarrow \mathcal{Q}\sigma, \beta \triangle \alpha, C \cup C' \mid \theta\sigma \rangle$$

WSLD Resolution: Implementation details

Definition (operational semantics for expanded programs)

Defined as a transition system $\langle E, \Rightarrow_{\text{EXP}} \rangle$ where

- E is a set of tuples $\langle \mathcal{G}, \alpha, C \mid \theta \rangle$ (goal, approximation degree, block constraints, substitution),
- $\Rightarrow_{\text{EXP}} \subseteq (E \times E)$ is a transition relation which satisfies:

Rule 2: if $\langle p(x_1, \ldots, x_n) \leftarrow x_1 \approx t_1 \wedge \cdots \wedge x_n \approx t_n \wedge \mathcal{Q}'; \beta; C' \rangle \ll \Pi'$ and $\mathcal{S}at(C \cup C') \neq \mathit{failure}$

$$\langle (\leftarrow \underline{p(s_1, \ldots, s_n)} \wedge \mathcal{Q}), \alpha, C \mid \theta \rangle \Rightarrow_{\text{EXP}} \langle (\leftarrow s_1 \approx t_1 \wedge \cdots \wedge s_n \approx t_n \wedge \mathcal{Q}' \wedge \mathcal{Q}), \beta \triangle \alpha, C \cup C' \mid \theta \rangle$$

in Rule 2, we perform a syntactic unification of the selected atom of the e-goal and the head of the e-clause.

WSLD Resolution: Implementation details

Example (8: e-clauses for the program of the last example 6)

```
p(X1,C0,C2,D):- unify_arguments_a3([[X1,X,C0,C1,D1]]),
                r(X,C1,C2,D2),
                degree_composition([0.75,D1,D2],D),
                over_lambdacut(D).
q(X1,C0,C3,D):- over_lambdacut(0.9),
                sat_a3([q:0,p:0],C0,C1),
                unify_arguments_a3([[X1,X,C1,C2,D1]]),
                r(X,C2,C3,D2),
                degree_composition([0.9,0.75,D1,D2],D),
                over_lambdacut(D).
r(X1,C0,C1,D):- unify_arguments_a3([[X1,a,C0,C1,D1]]),
                degree_composition([1,D1],D),
                over_lambdacut(D).
```

Outline

The foreign module and the foreign library

The module `foreign.pl` collects a set of predicates that connect programs written in C performing specific tasks which are critical:

- `ext_tokenize/2`: Lexically analyzes the string passed as a parameter.
- `ext_closure/6`: Calculates the reflexive, symmetric or transitive closure of the equation list. It uses a fuzzy variant of the Warshall algorithm.
- `ext_block_equs/4`: Computes the proximity blocks of level λ associated to a fuzzy relation $\mathcal{R}$ and generates the extended relation $\mathcal{RB}$. it uses an adaptation of the Bron-Kerbosch algorithm.
- `ext_translate_fuzzy_sets/5`: Transforms the information of a linguistic variable into a reflexive fuzzy relation that simulates it. It uses "*fuzzy matching*" techniques coming from FuzzyCLIPS.

Filtering techniques

- Imposing a λ-cut in a program discards answers below this threshold.
- The aim of Filtering is to reduce the search space eagerly **in the compilation stage**:
  - Only those $\mathcal{RB}$ entries with an approximation degree greater than or equal to the current λ-cut are generated.
  - Discarding the generation of extended rules for which their approximation degree in $\mathcal{RB}$ is below the current λ-cut.
  - Loading a rule only if its weight is above the cut.
- It can be activated/deactivated inside a program `:-filtering(+Boolean)`, or from the system prompt `fl +Boolean`.

Tail recursion

- The previous implementation of WSLD-resolution by BPL expansion breaks tail recursion of user programs.
- Because partial degrees are composed by means of a last call to the internal predicate `degree_composition/2`.
- This problem can be solved by computing the partial degrees first and passing them, on the fly, through an accumulator of each call in the body of a rule, keeping the last call as the recursive call.

Tail recursion

Example (9: e-clauses for the program of the example 6)

```
p(X1,C0,C2,D0,D):-
        unify_arguments_a3([[X1,X,C0,C1,D1]]),
        degree_composition([0.75,D0,D1],D2),
        r(X,C1,C2,D2,D).
q(X1,C0,C3,D0,D):-
        sat_a3([q:0,p:0],C0,C1),
        unify_arguments_a3([[X1,X,C1,C2,D1]]),
        degree_composition([0.75,0.9,D0,D1],D2),
        r(X,C2,C3,D2,D).
r(X1,C0,C2,D0,D):-
        unify_arguments_a3([[X1,a,C0,C2,D1]]),
        degree_composition([D0,D1],D).
```

Tail recursion

- Tail Recursion Optimisation (TRO) improves both time and space behaviour.
- We test both behaviours with **a simple example** (a program that sums the elements of a list with 20K integers):

```
my_sum([],S,S).
my_sum([X—Xs],C,S) :- C1 is C+X, my_sum(Xs,C1,S).
```

- The following table summarize the results of the experiment:

TRO	Time	Inferences	Global Stack	Local Stack
No	3,714	1,400,185	4,194,304	8,388,608
Yes	2,302	1,400,204	524,288	0

Time **speed-up when enabling TRO is** 1.6×.

Indexing

- The previous implementation of WSLD-resolution by BPL expansion breaks the indexing of user programs.
- Because it is necessary to flatten and linearise the rules of the BPL source program.
- One possible solution: to mimicking the same technique that we apply to simulate the fuzzy unification of the symbols at the root of the head rule by means of a crisp unification.

Indexing

Example (10: Program expansion and indexing)

Consider the following program:

```
a∼b=1.0.
p(a).
```

whose standard translation is:

```
p(X1,C1,C2,D1,D):- unify_arguments_a3([[X1,a,C1,C2,D2]]),
                   degree_composition([D1,D2],D).
```

A posible translation for the fact preserving indexing could be:

```
p(a,C,C,D,D).
p(b,Ci,Co,D,D) :- sat_a3([a:0,b:0],Ci,Co).
```

This translation technique breaks our entended operational semantics consisting in returning a representative with the higher approximation degree of all unifiers.

Indexing

- Though this translation is not applicable in general in our setting, we can still retain it for all data symbols which do not participate in any proximity equation.
- An important data structure that benefit from this are **lists** (for which it is not expected to be involved in any proximity equation).
- This implies great performance improvements which amounts to a speed-up of 190× for the optimised version in some tests.

Tail recursion optimisation and indexing

- Combined effect of Tail Recursion Optimisation (TRO) and indexing solving a **linguistic application**.

Syst.	TRO	Indexing	Time	Inferences	Glo. Stack	Loc. Stack
BPL	No	No	1,284	2,926,462	2,097,152	2,064,384
BPL	Yes	No	1.291	2,926,481	2,097,152	0
BPL	No	Yes	0.589	2,606,458	2,097,152	4,161,536
BPL	Yes	Yes	0.597	2,606,469	2,097,152	0
SWI	Yes	Yes	0.250	2,370,737	2,097,152	0

- When TRO and indexing are enabled, we can spect speed-up of about 2×.

Outline

Comparison of different systems

- We tested the applicability of uncertainty systems to face crisp applications as a way to measure the **cost of processing weak unification and other features**: Queens benchmark.

N	BPL	FASILL	FLOPER	SQCLP	SQCLP+w	SWI
8	0.186	8.533	0.008	0.005	0.502	0.005
9	0.786	45.131	0.020	0.022	1.956	0.014
10	3.705	253.208	0.119	0.091	10.475	0.053
11	15.658	1259.431	0.577	0.424	63.245	0.345
12	89.820	7743.477	2.592	1.820	378.099	1.669

Comparison of different systems

- Testing weak unification and weights on different systems by means of an instrumented benchmark.

	BPL		FASILL		SQCLP+w	
N	Time	Inferences	Time	Inferences	Time	Inferences
200	0.014	136,037	2.642	17,632,711	0.067	397,786
400	0.041	511,437	18.411	70,225,711	0.341	1,475,377
600	0.089	1,126,837	31.067	157,778,711	0.566	3,232,796
800	0.155	1,982,237	63.281	280,291,711	1.036	5,670,368
1000	0.233	3,077,637	120.433	437,764,711	1.703	8,787,900

- Speed-up ranging from 4.8× to 7.3× between BPL and SQCLP+w.

Comparison of different systems

- Measure ratios between Bousi~Prolog and other systems for **the library example** (Caballero et al., 2014).

	FASILL				SQCLP+w			
N	Speed-up	Inferences	Global	Local	Speed-up	Inferences	Global	Local
150	124	80	67	8	2,352	1,169	16	1,021
300	136	94	99	8	2,116	1,233	24	4,093
450	161	107	264	8	1,615	1,277	66	4,680
600	195	121	195	8	1,629	1,308	49	7,021
750	215	134	390	17	1,810	1,324	97	9,361

Outline

Conclusions

- We have summarized our accumulated experience during more than a decade in the development of BPL.
- We described the structure of the BPLa system, and how its operational semantics is implemented by BPL expansion.
- We have introduced a number of translation techniques: Filtering; TRO; some indexing capabilities.
- Other techniques for efficiency are embodied into a foreign library written in C.
- Finally, these optimisations have been tested and the BPL system compared to others: The experiment results confirm considerable gains on efficiency.