The Bousi∼Prolog User Manual In A Nutshell (Version 1.04 — Release April 2008).

Pascual Julián-Iranzo, Clemente Rubio-Manzano, and Juan Gallardo-Casero

Department of Information Technologies and Systems,
University of Castilla-La Mancha.
{Pascual.Julian}@uclm.es,{Clemente.Rubio}{Juan.Gallardo}@alu.uclm.es

1 Introduction

Bousi∼Prolog (BPL for short) is an extension of the Prolog language with an operational semantics based on the weak SLD resolution principle (WSLD) of [3]. WSLD is an adaptation of the SLD resolution principle where classical unification has been replaced by an algorithm based on similarity relations defined on a syntactic domain. The weak unification algorithm provides a weak most general unifier as well as a numerical value, called the *unification degree*. Hence, Bousi∼Prolog computes answers as well as approximation degrees. At this point, it is worth to say that "Bousi" is the Spanish acronym for "fuzzy unification by similarity".

The BPL system we are presenting is a prototype, high level implementation written on top of SWI-Prolog [5]. The complete implementation consists of about 900 lines of Prolog code. The parser and translator 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). Then, following standard techniques [4], a meta-interpreter executes the TPL code according to the weak SLD resolution principle. More information about the structure and main features of the BPL system can be found in [2].

Since the BPL system is a prototype implementation some bugs may arise during its execution. Please, send your comments to Pascual.Julian@uclm.es

In the following, we assume some familiarity with the field of logic programming [1]. So, some basic concepts (such as: term, atomic formula, clause, substitution, etc.) are not introduced.

2 Requirements and Instalation Procedure

Bousi∼Prolog version 1.04 (release April 2008) is available in two distribution formats: the executable distribution and the complete source code distribution.

The current version of the low level implementation of Bousi∼Prolog runs on Unix-based platforms and has been tested under Linux (Red Hat/Fedora and Debian/Ubuntu distributions).

If you want to install the BPL system on your computer, starting from the complete source code distribution you need SWI-Prolog. Also, for compiling the source code, in order to generate an executable (state) file you may need the following packages installed on your computer:

- Red Hat/Fedora distributions: gmp-devel, ncurses-devel, readline-devel.
- Debian/Ubuntu distributions: libgmp3-dev, libncurses5-dev, libncursesw5-dev, libreadline5-dev.

On the other hand if you want to install and run the executable distribution, you do not need SWI-Prolog and, in order to run the executable (state) file you only may need the following packages installed on your computer:

- Red Hat/Fedora distributions: gmp, ncurses, readline.
- Debian/Ubuntu distributions: libgmp3c2, libncurses5, libncursesw5, libreadline5.

The following **installation procedure** is only for the executable distribution.

- Go into the directory where you want to download the executable distribution and download it.
- Decompress the file `bousi-installer.tgz` by the command:

  ```
  tar -xzf bousi-installer.tgz
  ```

  Then, a subdirectory (called "`bousi-installer`") is created containing the files of the executable distribution.
- Go into the `bousi-installer` directory, launching the installation process by the command

  ```
  . installer.sh
  ```

  Then, follow the installation instructions. If you are an ordinary user and you choose the default installation option, the executable file "`bousi`" and the shared library "`closure.so`" are installed into the directory

  `$HOME/bousi-prolog`.

  If you are root, the default directory where those files are installed is

  `/usr/local/bousi-prolog`.
- Once the installation is ended you can start the BPL system via the command "`bousi`".

WARNING: If you try to launch the BPL system by double clicking the executable file `bousi` from a file browser window, it might happen that the BPL system is not open but it remains resident in memory consuming a great amount of CPU resources. Always it is preferable to launch the BPL system from a (unix shell) terminal by means of the command "`bousi`".

```
                                                     Universidad de
|O)               |D)                              Castilla - La Mancha.
|O)(O)\U(S)|I| ~~ || |R (O) |L (O) (G|.      (Version 1.04 ~~ February 2008)
----------------------------------------------------------------------------
Welcome to Bousi~Prolog. This software is for research and educational
purposes only and it is distributed with NO WARRANTY.
Please, report bugs and send your suggestions to: Pascual.Julian@uclm.es
----------------------------------------------------------------------------

----------------------------------------------------------------------------
ld   -> (load) reads a file containing the source program for loading
lt   -> (list) displays the current loaded program
sv   -> (solve) solves a (possibly conjunctive) query
lc   -> (lambdacut) reads or sets the lower bound for the approximation
        degree in a weak unification process
hp   -> (help) shows this help info
ls   -> (list) lists the files in the working directory
pwd  -> (print working directory) shows information of the working directory
cd   -> (change Dir) changes the working directory
sh   -> (shell) starts a new interactive shell process
qt   -> (quit) leaves the system
----------------------------------------------------------------------------
---          Write  "hp Topic "  to obtain more help on that Topic       ---
----------------------------------------------------------------------------
```

Fig. 1. Welcome screen and shell commands of the Bousi∼Prolog system

3 The BPL Environment

In this section we give a brief tutorial on how to use the BPL environment wich is the interface between the user and this high level implementation.

The BPL environment implements a command shell. Figure 1 shows the welcome screen and a summary of the shell commands. To start the BPL environment execute the command "`bousi`". Once the BPL environment is running you can type in the following commands:

– `ld -> (LOAD)` The Load command can be used to show what program has been loaded, when used without an argument. In this case it shows the name of the loaded program or the notice "No program loaded". Also, when used with a single argument, it can be used to load a file containing the source program.

  Syntax:

`ld`	shows if there is a current loaded program.
`ld <FileName>` ...	reads the file FileName, containing the source program, translates it into a TPL file and consults the TPL file. `<FileName>` is a BPL file name without extension.

– `lt -> (LIST)` The List command shows the rules in the current loaded program. Note that similarity equations are not shown.

  Syntax:

  `lt` .... displays the current loaded program.

– `sv -> (SOLVE)` The Solve command sends a (possibly conjunctive) query to be solved by the Evaluator. It requires a single argument that must be an atom or a conjunction of atoms. Conjunctive queries must be enclosed by curved parenthesis signs.

Syntax:

```
sv <Atom>
sv (<Atom_1>, <Atom_2>, ..., <Atom_N>)
```

– `lc -> (LAMBDACUT)` We can impose a limit to the expansion of the search space in a computation by what we called a "lambdacut". When Lambdacut is set, 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. The Lambdacut command can be used to show what is the current Lambdacut value or to set a new Lambdacut value.

Syntax:

`lc`	reads the current value of the lower bound for the approximation degree in a weak unification process.
`lc <Value>`	sets the lower bound for the approximation degree in a weak unification process. `Value` is a real between 0 and 1.

– `hp -> (HELP)` The Help command gives a summary of the BPL shell commands or specific information on a BPL shell command.

Syntax:

`hp`	gives a summary of BPL shell commands.
`hp <ShellCmnd>` ..	gives information on a BPL shell command.

– `ls -> (LIST)` Executes the command "ls -B –color" on the operating system. That is, this command shows a list of the files in the working directory, ordered by columns, ignoring backups and distinguishing plane files from directories, executable files and so on, by color.

Syntax:

`ls`	lists the files in the working directory.

– `pwd -> (PRINT WORKING DIRECTORY)` Print the full filename of the current working directory.

Syntax:

`pwd`	shows the filename of the working directory.

– `cd -> (CHANGE DIR)` Changes the working directory.

Syntax:

`cd <Dir>`	Changes the current directory to `<Dir>`, where "`<Dir>`" is a pathname.

– `sh -> (SHELL)` Starts an interactive Unix shell. Default is /bin/sh. The environment variable SHELL overrides this default.

Syntax:

`sh`	starts a new interactive shell process.

– `qt -> (QUIT)` leaves the system.

4 Edition and execution of a Program and a Query

The BPL system does not provide an specific text editor. This is not a real inconvenient in a unix-based implementation. If you wish to create a new BPL program or open an old one for edition, you can use your favorite text editor.

In order to launch a BPL program, you must follow these steps:

1. Once a BPL program has been created, use the `ld` command for loading the program into the working space.
2. Use the `sv` command to send a query to the BPL system.
3. The BPL systems returns an answer or responds "`no answers`" if there is not any. In the first case, you can type the semi-colon plus the new-line keys "`;<new-line>`", if you want another solution, or return if you are satisfied, after which Bousi∼Prolog will say "`Yes`". If Bousi∼Prolog answers "`no answers`", this indicates it cannot find any (more) answers to the query.

5 The Bousi∼Prolog programming language

In this section we briefly summarize the features of Bousi∼Prolog. We concentrate on the syntactical aspects.

The programming language we call Bousi∼Prolog is an extension of the standard Prolog language with a similarity relation defined on a syntactic domain. Therefore, the syntax is mainly the Prolog syntax but enriched with a built-in symbol used for describing similarity relations (actually, fuzzy binary relations which are automatically converted into similarity relations) by means of *similarity equations* of the form:

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

meaning that two constants, n-ary function symbols or n-ary predicate symbols are similar with a certain degree.

A BPL program is a sequence of Prolog facts and rules plus a sequence of similarity equations. A *rule* is a conditional formula of the form:

$\mathcal{A}{:}{-}L_1, L_2, \ldots, L_n.$

where the $\mathcal{A}$ is an atomic formula (i.e., a relational symbol applied to a tuple of rst-order terms) and the L_i are literals (e.i., either atomic formulas or negation of atomic formulas). The atomic formula "$\mathcal{A}$" is called the *head* of the rule and the conjunction of literals "$L_1, L_2, \ldots, L_n$" is called the *body*. In general, the body may be an aggregation of literals connected by conjunctions (`,`), disjunctions (`;`) and if-then-else (`->`) operators. A *fact* is a rule without a body.

Bousi∼Prolog uses the similarity-based SLD principle [3] (also called weak SLD resolution) as operational semantics. This operational mechanism is well suited for flexible query answering.

Example 1. 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).
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)`".

5.1 The weak unification operator

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 succeeds 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 2. Assume that the BPL program of Example 1 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 succeed 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 succeeds 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 "`Term1 ~~ Term2 = Degree`"; ii) it behaves as the syntactic unification operator when it is used dissociated of the weak unification operator "`~~`".

5.2 Distinct Classes of Negations

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 negation is as follows:

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

where the predicate `solve/2` is used internally by the BPL meta-interpreter to solve BPL queries.

5.3 Directives

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 (explained before).

On the other hand, in a BPL program, the user supplies an initial subset of similarity equations for defining a fuzzy relation and then, the system automatically generates its reflexive, symmetric, transitive closure to obtain, by default, a similarity relation. In order to resolve certain problems sometimes is preferable not to generate the similarity relation from the initial set of similarity equations. This effect can be achieved by means of the BPL directive "`:- transitivity(no)`", which inhibits the construction of the transitive closure, during the translation phase. If the BPL directive "`:- transitivity(no)`" is included at the beginning of a BPL program, only the reflexive, symmetric closure is computed. By default, the transitivity flag is set to "`yes`"

5.4 Built-in predicates

In this section we give a list of the built-in predicates provided by Bousi∼Prolog. As for the current implementation the core of the system is a meta-interpreter, built-in predicates are sent directly to the Prolog interpreter.

Control and meta-call predicates Control predicates are useful to manage repetition through backtracking. We provide two simple predicates for control:

- `fail`: Always produces a fail.
- `repeat` Always succeed, providing an infinite number of choice points.

Meta-call predicates are used to call terms constructed at run time. The meta-call mechanism offered by Bousi∼Prolog is:

- `call(Goal)`: Invokes `Goal` as a goal.

Arithmetic operators The basic operator for the evaluation of arithmetic expressions is:

- `Exp1 is Exp2`: Forces the evaluation of `Exp1` and `Exp2`. It will be successful if `Exp1` and `Exp2` unify after their evaluation.

The following arithmetic comparison operators are available:

- `Exp1 =:= Exp2`: Succeeds if expression Exp1 evaluates to a number equal to Exp2.
- `Exp1 =\= Exp2`: Succeeds if expression Exp1 evaluates to a number different to Exp2.
- `Exp1 > Exp2`: True if expression Exp1 evaluates to a number greater than Exp2.
- `Exp1 < Exp2`: True if expression Exp1 evaluates to a number less than Exp2.
- `Exp1 >= Exp2`: True if expression Exp1 evaluates to a number greater than or equal to Exp2.
- `Exp1 =< Exp2`: True if expression Exp1 evaluates to a number less than or equal to Exp2.

Comparison operators Terms are ordered in the so called "*standard order*". This order is defined as follows:

1. `Variables` are less than `Numbers`, `Numbers` are less than `Atoms`, `Atoms` are less than `Strings`, `Strings` are less than `Comound Terms`.
2. `Variables` are sorted by address.
3. `Atoms` are compared alphabetically.
4. `Strings` are compared alphabetically.
5. `Numbers` are compared by value.
6. `Compound` terms are first checked on their arity, then on their functor-name (alphabetically) and finally recursively on their arguments, leftmost argument first.

To compare the standard order between two terms the following operators are provided:

– `Term1 == Term2`: Succeeds if term `Term1` is syntactically equivalent to term `Term2`.
– `Term1 \== Term2`: Succeeds if term `Term1` is not syntactically equivalent to term `Term2`.
– `Term1 =@= Term2`: True if `Term1` is "structurally equal" to `Term2`.
– `Term1 \=@= Term2`: True if `Term1` is not "structurally equal" to `Term2`.
– `Term1 @< Term2`: True if `Term1` is greater than `Term2`, in the standard order of terms.
– `Term1 @> Term2`: True if `Term1` is less than `Term2`, in the standard order of terms.
– `Term1 @>= Term2`: True if `Term1` is greater than or equal to `Term2`, in the standard order of terms.
– `Term1 @=< Term2`: True if expression `Term1` is less than or equal to `Term2`, in the standard order of terms.

Syntactic unification

– `Term1 = Term2`: Unifies `Term1` with `Term2`. True if the unification succeeds.
– `Term1 \= Term2`: True if `Term1` does not unify with `Term2`.

List manipulation Some standard predicates for list manipulation:

– `is_list(Term)`: True if `Term` is bound to the empty list "`[]`" or a term "`[_|L]`" where the second argument "L" is a list.
– `member(Elem, List)`: Succeeds when `Elem` is one of the members of `List`.
– `append(List1, List2, List)`: True when `List` is the concatenation of `List1` and `List2`.
– `reverse(List1, List2)`: Reverses the order of the elements in `List1` and unifies the result with the elements of `List2`.
– `last(List, Elem)`: Succeeds if `Elem` unifies with the last element of `List`.
– `length(List, Int)`: True if `Int` is the number of elements of `List`.

Analysing and Constructing Terms Some predicates for cataloging terms:

– `var(Term)`: True if `Term` currently is a free variable.
– `integer(Term)`: True if `Term` is bound to an integer.
– `number(Term)`: True if `Term` is bound to an integer or floating point number.
– `atom(Term)`: True if `Term` is bound to an atom.
– `atomic(Term)`: True if `Term` is bound to an atom, string, integer or floating point number.
– `compound(Term)`: True if `Term` is bound to a compound term.

Some predicates for constructing terms:

– `functor(Term, Functor, Arity)`: True if `Term` is a term with functor `Functor` and arity `Arity`. If `Term` is a variable it is unified with a new term holding only variables.
– `arg(Arg, Term, Value)`: `Term` should be instantiated to a term, `Arg` to an integer between `1` and the `arity` of `Term`. `Value` is unified with the Arg-th argument of `Term`.
– `Term =.. List`: `List` is a list which head is the functor of `Term` and the remaining arguments are the arguments of the term. Each of the arguments may be a variable, but not both.

Edinburgh-style I/O The I/O predicates refer to the *current input stream* and the *current output stream*. Initially these streams are connected to the terminal. The current output stream is changed using `tell/1`. The current input stream is changed using `see/1`. The streams current value can be obtained using `telling/1` for output and `seeing/1` for input streams.

The streams are connected to files or the terminal. The reserved stream name "`user`" refers to the terminal. The connection is made either by `tell/1` or by `see/1` which open a file for output or input respectively. A simple working session is as follows:

```
see(file_name). % open the file "file_name"
                % which is made the current input stream
.......
predicates for reading
the current input stream
.......
seen. % close the current input stream
```

The I/O predicates available on this implementation of Bousi∼Prolog are:

– `see(File)`: Open `File` for reading and makes it the current input stream.
– `tell(File)`: Open `File` for writing and makes it the current output stream.
– `seeing(Stream)`: Gets the current input stream.
– `telling(Stream)`: Gets the current output stream.
– `seen`: Closes the current input stream. The new input stream becomes `user`.

- `told`: Closes the current output stream. The new output stream becomes `user`.
- `read(Term)`: Reads the next term from the current input stream and unifies it with `Term`.
  When reaching end of file `Term` is unified with the atom `end_of_file`.
- `write(Term)`: Writes `Term` to the current output stream.
- `get(Char)`: Reads the current input stream and unifies the next non-blank character with `Char`. `Char` is unified with `-1` at end of file.
- `get0(Char)`: Reads the current input stream and unifies `Code` with the character code of the next character. `Code` is unified with `-1` at end of file.
- `put(Char)`: Writes `Char` to the current output stream.
- `nl`: Writes a new-line character to the current output stream.

Runtime Statistics

- `statistics(Key, Value)`: Unifies system statistics determined by Key with Value. Some possible keys are:
  - `cputime`: cpu time since Prolog was started in seconds
  - `inferences`: Total number of predicate calls since Prolog was started.

See the SWI-Prolog documentation for more details.

5.5 Some limitations

Although Bousi∼Prolog implements the main features of a standard Prolog other features, such as working with modules, are not covered. Also, because the parser phase is delegated to standard Prolog predicates we lost the control of the whole parsing process, what imposes some real limitations. For instance, we cannot use operators defined by the user, that is the "`:- op(_, _, _)`" directive. In the future we want to add these missing features to our language.

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

References

1. K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, Englewood Cliffs, NJ, 1997.
2. P. Julián-Iranzo, C. Rubio-Manzano, and J. Gallardo-Casero. Bousi∼prolog: a prolog extension language for flexible query answering. page 14, 2008. (Submitted for publication).
3. Maria I. Sessa. Approximate reasoning by similarity-based sld resolution. *Theoretical Computer Science*, 275(1-2):389–426, 2002.
4. L. Sterling and E. Shapiro. *The Art of Prolog (Second Edition).* The MIT Press, Cambridge, MA, 1994.
5. J. Wielemaker. SWI-Prolog 5.6 Reference Manual. Technical report: vesion 5.6.52, March 2008, University of Amsterdam, 2008.