`cplint` Version 2.0 Manual

Fabrizio Riguzzi
fabrizio.riguzzi@unife.it

July 27, 2010

1 Introduction

`cplint` is a suite of programs for reasoning with LPADs [11, 12] and CP-logic programs [10, 13].

It consists of three Prolog modules for answering queries using goal-oriented procedures plus three Prolog modules for answering queries using the definition of the semantics of LPADs and CP-logic.

The modules for answering queries using using goal-oriented procedures are `lpadsld.pl`, `lpad.pl` and `cpl.pl`:

- `lpadsld.pl`: computes the probability of a query using the top-down procedure described in in [7] and [8]. It is based on SLDNF resolution and is an adaptation of the interpreter for ProbLog [4].

  It was proved correct [8] with respect to the semantics of LPADs for range restricted acyclic programs [1] without function symbols.

  It is also able to deal with extensions of LPADs and CP-logic: the clause bodies can contain `setof` and `bagof`, the probabilities in the head may be depend on variables in the body and it is possible to specify a uniform distribution in the head with reference to a `setof` or `bagof` operator. These extended features have been introduced in order to represent CLP(BN) [9] programs and PRM models [6]: `setof` and `bagof` allow to express dependency of an attribute from an aggregate function of another attribute, as in CLP(BN) and PRM, while the possibility of specifying a uniform distribution allows the use of the reference uncertainty feature of PRM.

- `lpad.pl`: computes the probability of a query using a top-down procedure based on SLG resolution [3]. As a consequence, it works for any

sound LPADs, i.e., any LPAD such that each of its instances has a two valued well founded model.

- `cpl.pl`: computes the probability of a query using a top-down procedure based on SLG resolution and moreover checks that the CP-logic program is valid, i.e., that it has at least an execution model.

The modules for answering queries using the definition of the semantics of LPADs and CP-logic are `semlpadsld.pl`, `semlpad.pl` and `semcpl.pl`:

- `semlpadsld.pl`: given an LPAD P, it generates all the instances of P. The probability of a query Q is computed by identifying all the instances where Q is derivable by SLDNF resolution.

- `semlpad.pl`: given an LPAD P, it generates all the instances of P. The probability of a query Q is computed by identifying all the instances where Q is derivable by SLG resolution.

- `semlcpl.pl`: given an LPAD P, it builds an execution model of P, i.e., a probabilistic process that satisfy the principles of universal causation, sufficient causation, independent causation, no deus ex machina events and temporal precedence. It uses the definition of the semantics given in [13].

2 Installation

`cplint` is distributed in source code in the git version of Yap. It includes Prolog and C files. Download it by following the instruction in `http://www.ncc.up.pt/~vsc/Yap/downloads.html`.

`cplint` requires cudd and glib-2.0. You can download cudd from `http://vlsi.colorado.edu/~fabio/CUDD/`. You can download glib-2.0 (version $\geq$ 2.0) from `http://www.gtk.org/`. This is a standard GNU package so it is easy to install it using the package management software of your Linux or Cygwin distribution.

Compile cudd:

1. downlad `cudd-2.4.2.tar.gz`
2. decompress it
3. `cd cudd-2.4.2`
4. check makefile options

5. `make`

Install Yap together with `cplint`: when compiling Yap following the instuction of the `INSTALL` file in the root of the Yap folder, use

```
configure --enable-cplint=DIR
```

Under Windows, you have to use Cygwin (glu does not compile under MinGW), so

```
configure --enable-cplint=DIR --enable-cygwin
```

where `DIR` is the path to the directory `cudd-2.4.2` (including `cudd-2.4.2`).

After having performed `make install` you can do `make installcheck` that will execute a suite of tests of the various programs. If no error is reported you have a working installation of `cplint`.

3 Syntax

Disjunction in the head is represented with a semicolon and atoms in the head are separated from probabilities by a colon. For the rest, the usual syntax of Prolog is used. For example, the CP-logic clause

$$h_1 : p_1 \vee \ldots \vee h_n : p_n \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_l$$

is represented by

```
h1:p1 ; ... ; hn:pn :- b1,...,bm,\+ c1,....,\+ cl
```

No parentheses are necessary. The `pi` are numeric expressions. It is up to the user to ensure that the numeric expressions are legal, i.e. that they sum up to less than one.

If the clause has an empty body, it can be represented like this

```
h1:p1 ; ... ;hn:pn.
```

If the clause has a single head with probability 1, the annotation can be omitted and the clause takes the form of a normal prolog clause, i.e.

```
h1:- b1,...,bm,\+ c1,...,\+ cl.
```

stands for

```
h1:1 :- b1,...,bm,\+ c1,...,\+ cl.
```

The coin example of [12] is represented as (see file `coin.cpl`)

```
heads(Coin):1/2 ; tails(Coin):1/2:-
     toss(Coin),\+biased(Coin).

heads(Coin):0.6 ; tails(Coin):0.4:-
     toss(Coin),biased(Coin).

fair(Coin):0.9 ; biased(Coin):0.1.

toss(coin).
```

The first clause states that if we toss a coin that is not biased it has equal probability of landing heads and tails. The second states that if the coin is biased it has a slightly higher probability of landing heads. The third states that the coin is fair with probability 0.9 and biased with probability 0.1 and the last clause states that we toss a coin with certainty.

4 Commands

All six modules accept the same commands for reading in files and answering queries. The LPAD or CP-logic program must be stored in a text file with extension `.cpl`. Suppose you have stored the example above in file `coin.cpl`. In order to answer queries from this program, you have to run Yap, load one of the modules (such as for example `lpad.pl`) by issuing the command

```
use_module(library(lpad)).
```

at the command prompt. Then you must parse the source file `coin.cpl` with the command

```
p(coin).
```

if `coin.cpl` is in the current directory, or

```
p('path_to_coin/coin').
```

if `coin.cpl` is in a different directory. At this point you can pose query to the program by using the predicate `s/2` (for solve) that takes as its first argument a conjunction of goals in the form of a list and returns the computed probability as its second argument. For example, the probability of the conjunction `head(coin),biased(coin)` can be asked with the query

```
s([head(coin),biased(coin)],P).
```

For computing the probability of a conjunction given another conjunction you can use the predicate `sc/3` (for solve conditional) that take takes as input the query conjunction as its first argument, the evidence conjunction as its second argument and returns the probability in its third argument. For example, the probability of the query `heads(coin)` given the evidence `biased(coin)` can be asked with the query

```
sc([heads(coin)],[biased(coin)],P).
```

After having parsed a program, in order to read in a new program you must restart Yap when using `semlpadsld.pl` and `semlpad.pl`. With the other modules, you can directly parse a new program.

When using `lpad.pl`, the system can print the message "Uunsound program" in the case in which an instance with a three valued well founded model is found. Moreover, it can print the message "It requires the choice of a head atom from a non ground head": in this case, in order to answer the query, all the groundings of the culprit clause must be generated, which may be impossible for programs with function symbols.

When using `semcpl.pl`, you can print the execution process by using the command `print.` after `p(file).` Moreover, you can build an execution process given a context by issuing the command `parse(file).` and then `build(context).` where `context` is a list of atoms that are true in the context. `semcpl.pl` can print "Invalid program" in the case in which no execution process exists.

When using `cpl.pl` you can print a partial execution model including all the clauses involved in the query issued with `print.` `cpl.pl` can print the messages "Uunsound program", "It requires the choice of a head atom from a non ground head" and "Invalid program".

The modules make use of a number of parameters in order to control their behavior. They that can be set with the command

```
set(parameter,value).
```

from the Yap prompt after having loaded the module. The current value can be read with

```
setting(parameter,Value).
```

from the Yap prompt. The available parameters are:

- `epsilon_parsing` (valid for all six modules): if (1 - the sum of the probabilities of all the head atoms) is smaller than `epsilon_parsing` then `cplint` adds the null events to the head. Default value 0.00001

- `save_dot` (valid for all goal-oriented modules): if `true` a graph representing the BDD is saved in the file `cpl.dot` in the current directory in dot format. The variables names are of the form `Xn_m` where `n` is the number of the multivalued variable and `m` is the number of the binary variable. The correspondence between variables and clauses can be evinced from the message printed on the screen, such as

  ```
  Variables: [(2,[X=2,X1=1]),(2,[X=1,X1=0]),(1,[])]
  ```

  where the first element of each couple is the clause number of the input file (starting from 1). In the example above variable `X0` corresponds to clause `2` with the substitutions `X=2,X1=1`, variable `X1` corresponds to clause `2` with the substitutions `X=1,X1=0` and variable `X2` corresponds to clause `1` with the empty substitution. You can view the graph with `graphviz` (`www.graphviz.org`) using the command

  ```
  dotty cpl.dot &
  ```

- `ground_body` (valid for `lpadsld.pl` and all semantic modules): determines how non ground clauses are treated: if `true`, ground clauses are obtained from a non ground clause by replacing each variable with a constant, if `false`, ground clauses are obtained by replacing only variables in the head with a constant. In the case where the body contains variables not in the head, setting it to false means that the body represents an existential event.

5 Semantic Modules

The three semantic modules need to produce a grounding of the program in order to compute the semantics. They require an extra file with extension `.uni` (for universe) in the same directory where the `.cpl` file is.

There are two ways to specify how to ground a program. The first consists in providing the list of constants to which each variable can be instantiated. For example, in our case the current directory will contain a file `coin.uni` that is a Prolog file containing facts of the form

```
universe(var_list,const_list).
```

where `var_list` is a list of variables names (each must be included in single quotes) and `const_list` is a list of constants. The semantic modules generate the grounding by instantiating in all possible ways the variables of `var_list`

with the constants of `const_list`. Note that the variables are identified by name, so a variable with the same name in two different clauses will be instantiated with the same constants.

The other way to specify how to ground a program consists in using mode and type information. For each predicate, the file `.uni` must contain a fact of the form

```
mode(predicate(t1,...,tn)).
```

that specifies the number and types of each argument of the predicate. Then, the list of constants that are in the domain of each type `ti` must be specified with a fact of the form

```
type(ti,list_of_constants).
```

The file `.uni` can contain both universe and mode declaration, the ones to be used depend on the value of the parameter `grounding`: with value `variables`, the universe declarations are used, with value `modes` the mode declarations are used.

With `semcpl.pl` only mode declarations can be used.

6 Extensions

In this section we will present the extensions to the syntax of LPADs and CP-logic programs that `cplint` can handle.

The first is the use of some standard Prolog predicates. The bodies can contain the built-in predicates:

```
is/2
>/2
</2
>=/2
=</2
=:=/2
=\=/2
true/0
false/0
=/2
==/2
\=/2
\==/2
length/2
```

The bodies can also contain the following library predicates:

```
member/2
max_list/2
min_list/2
nth0/3
nth/3
```

plus the predicate

```
average/2
```

that, given a list of numbers, computes its arithmetic mean.

When using `lpadsld.pl`, the bodies can contain the predicates `setof/3` and `bagof/3` with the same meaning as in Prolog. Existential quantifiers are allowed in both, so for example the query

```
setof(Z, (term(X,Y))^foo(X,Y,Z), L).
```

returns all the instantiations of `Z` such that there exists an instantiation of `X` and `Y` for which `foo(X,Y,Z)` is true.

An example of the use of `setof` and `bagof` is in the file `female.cpl`:

```
male(C):M/P ; female(C):F/P:-
    person(C),
    setof(Male,known_male(Male),LM),
    length(LM,M),
    setof(Female,known_female(Female),LF),
    length(LF,F),
    P is F+M.

person(f).

known_female(a).

known_female(b).

known_female(c).

known_male(d).

known_male(e).
```

The disjunctive rule expresses the probability of a person of unknown sex of being male or female depending on the number of males and females that are known. This is an example of the use of expressions in the probabilities in the head that depend on variables in the body. The probabilities are well defined because they always sum to 1 (unless `P` is 0).

Another use of `setof` and `bagof` is to have an attribute depend on an aggregate function of another attribute, similarly to what is done in PRM and CLP(BN).

So, in the classical school example (available in `student.cpl`) you can find the following clauses:

```
student_rank(S,h):0.6 ; student_rank(S,l):0.4:-
    bagof(G,R^(registr_stu(R,S),registr_gr(R,G)),L),
    average(L,Av),Av>1.5.

student_rank(S,h):0.4 ; student_rank(S,l):0.6:-
    bagof(G,R^(registr_stu(R,S),registr_gr(R,G)),L),
    average(L,Av),Av =< 1.5.
```

where `registr_stu(R,S)` expresses that registration `R` refers to student `S` and `registr_gr(R,G)` expresses that registration `R` reports grade `G` which is a natural number. The two clauses express a dependency of the rank of the student from the average of her grades.

Another extension can be used with `lpadsld.pl` in order to be able to represent reference uncertainty of PRMs. Reference uncertainty means that the link structure of a relational model is not fixed but is uncertain: this is represented by having the instance referenced in a relationship be chosen uniformly from a set. For example, consider a domain modeling scientific papers: you have a single entity, paper, and a relationship, cites, between paper and itself that connects the citing paper to the cited paper. To represent the fact that the cited paper and the citing paper are selected uniformly from certain sets, the following clauses can be used (see file `paper_ref_simple.cpl`):

```
uniform(cites_cited(C,P),P,L):-
    bagof(Pap,paper_topic(Pap,theory),L).

uniform(cites_citing(C,P),P,L):-
    bagof(Pap,paper_topic(Pap,ai),L).
```

The first clauses states that the paper `P` cited in a citation `C` is selected uniformly from the set of all papers with topic theory. The second clauses expresses that the citing paper is selected uniformly from the papers with topic ai.

These clauses make use of the predicate

```
uniform(Atom,Variable,List)
```

in the head, where `Atom` must contain `Variable`. The meaning is the following: the set of all the atoms obtained by instantiating `Variable` of `Atom` with a term taken from `List` is generated and the head is obtained by having a disjunct for each instantiation with probability $1/N$ where N is the length of `List`.

A more elaborate example is present in file `paper_ref.cpl`:

```
uniform(cites_citing(C,P),P,L):-
    setof(Pap,paper(Pap),L).

cites_cited_group(C,theory):0.9 ; cites_cited_group(C,ai):0.1:-
    cites_citing(C,P),paper_topic(P,theory).

cites_cited_group(C,theory):0.01;cites_cited_group(C,ai):0.99:-
    cites_citing(C,P),paper_topic(P,ai).

uniform(cites_cited(C,P),P,L):-
    cites_cited_group(C,T),bagof(Pap,paper_topic(Pap,T),L).
```

where the cited paper depends on the topic of the citing paper. In particular, if the topic is theory, the cited paper is selected uniformly from the papers about theory with probability 0.9 and from the papers about ai with probability 0.1. if the topic is ai, the cited paper is selected uniformly from the papers about theory with probability 0.01 and from the papers about ai with probability 0.99.

PRMs take into account as well existence uncertainty, where the existence of instances is also probabilistic. For example, in the paper domain, the total number of citations may be unknown and a citation between any two paper may have a probability of existing. For example, a citation between two paper may be more probable if they are about the same topic:

```
cites(X,Y):0.005 :-
    paper_topic(X,theory),paper_topic(Y,theory).

cites(X,Y):0.001 :-
    paper_topic(X,theory),paper_topic(Y,ai).

cites(X,Y):0.003 :-
    paper_topic(X,ai),paper_topic(Y,theory).
```

```
cites(X,Y):0.008 :-
    paper_topic(X,ai),paper_topic(Y,ai).
```

This is an example where the probabilities in the head do not sum up to one so the null event is automatically added to the head. The first clause states that, if the topic of a paper `X` is theory and of paper `Y` is theory, there is a probability of 0.005 that there is a citation from `X` to `Y`. The other clauses consider the remaining cases for the topics.

7 Additional Files

In the directory where Yap keeps the library files (usually `/usr/local/share/Yap`) you can find the directory `cplint` that contains the files:

- `testlpadsld_gbtrue.pl, testlpadsld_gbfalse.pl, testlpad.pl, testcpl.pl, testsemlpadsld.pl, testsemlpad.pl testsemcpl.pl`: Prolog programs for testing the modules. They are executed when issuing the command `make installcheck` during the installation. To execute them afterwords, load the file and issue the command `t.`
- Subdirectory `examples`:
  - `alarm.cpl`: representation of the Bayesian network in Figure 2 of [12].
  - `coin.cpl`: coin example from [12].
  - `coin2.cpl`: coin example with two coins.
  - `dice.cpl`: dice example from [12].
  - `twosideddice.cpl, threesideddice.cpl` game with idealized dice with two or three sides. Used in the experiments in [8].
  - `ex.cpl`: first example in [8].
  - `exapprox.cpl`: example showing the problems of approximate inference (see [8]).
  - `exrange.cpl`: example showing the problems with non range restricted programs (see [8]).
  - `female.cpl`: example showing the dependence of probabilities in the head from variables in the body (from [12]).
  - `mendel.cpl, mendels.cpl`: programs describing the Mendelian rules of inheritance, taken from [2].

- `paper_ref.cpl, paper_ref_simple.cpl`: paper citations examples, showing reference uncertainty, inspired by [6].
- `paper_ref_not.cpl`: paper citations example showing that negation can be used also for predicates defined by clauses with `uniform` in the head.
- `school.cpl`: example inspired by the example `school_32.yap` from the source distribution of Yap in the `CLPBN` directory.
- `school_simple.cpl`: simplified version of `school.cpl`.
- `student.cpl`: student example from Figure 1.3 of [5].
- `win.cpl, light.cpl, trigger.cpl, throws.cpl, hiv.cpl, invalid.cpl`: programs taken from [13]. `invalid.cpl` is an example of a program that is invalid but sound.

The files `*.uni` that are present for some of the examples are used by the semantical modules. Some of the example files contain in an initial comment some queries together with their result.

- Subdirectory `doc`: contains this manual in latex, html and pdf.

8 License

`cplint`, as Yap, follows the Artistic License 2.0 that you can find in Yap CVS root dir. The copyright is by Fabrizio Riguzzi.

The program uses the library CUDD for manipulating BDDs that is included in glu. For the use of CUDD, the following license must be accepted:

Copyright (c) 1995-2004, Regents of the University of Colorado
All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the University of Colorado nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

References

[1] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Comput.*, 9(3/4):335–364, 1991.

[2] H. Blockeel. Probabilistic logical models for mendel's experiments: An exercise. In *Inductive Logic Programming (ILP 2004), Work in Progress Track*, 2004.

[3] Weidong Chen and David Scott Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[4] L. De Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2462–2467, 2007.

[5] L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In Saso Dzeroski and Nada Lavrac, editors, *Relational Data Mining*. Springer-Verlag, Berlin, 2001.

[6] L. Getoor, N. Friedman, D. Koller, and B. Taskar. Learning probabilistic models of relational structure. *Journal of Machine Learning Research*, 3:679–707, December 2002.

[7] Fabrizio Riguzzi. A top down interpreter for lpad and cp-logic. In *10th Congress of the Italian Association for Artificial Intelligence*. Springer, 2007. http://www.ing.unife.it/docenti/FabrizioRiguzzi/Papers/Rig-AIIA07.pdf.

[8] Fabrizio Riguzzi. A top down interpreter for lpad and cp-logic. In *The 14th RCRA workshop Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 2007. http://pst.istc.cnr.it/RCRA07/articoli/P19-riguzzi-RCRA07.pdf.

[9] V. Santos Costa, D. Page, M. Qazi, and J. Cussens. CLP(BN): Constraint logic programming for probabilistic knowledge. In *Uncertainty in Artificial Intelligence (UAI 2003)*, 2003.

[10] J. Vennekens, M. Denecker, and M. Bruynooghe. Representing causal information about a probabilistic process. In *10th European Conference on Logics in Artificial Intelligence, JELIA 2006*, LNAI. Springer, September 2006.

[11] J. Vennekens and S. Verbaeten. Logic programs with annotated disjunctions. Technical Report CW386, K. U. Leuven, 2003. http://www.cs.kuleuven.ac.be/∼joost/techrep.ps.

[12] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *The 20th International Conference on Logic Programming (ICLP 2004)*, 2004. http://www.cs.kuleuven.ac.be/∼joost/.

[13] Joost Vennekens, Marc Denecker, and Maurice Bruynooge. Extending the role of causality in probabilistic modeling. http://www.cs.kuleuven.ac.be/∼joost/cplogic.pdf, 2006.