cplint on SWISH Manual

April 26, 2019

1 Syntax

cplint permits the definition of discrete probability distributions and continuous probaility densities.

1.1 Discrete Probability Distributions

LPAD and CP-logic programs consist of a set of annotated disjunctive clauses. 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. A general CP-logic clause has the form

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

where Body is a conjunction of goals as in Prolog. 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 :- Body.
```

stands for

```
h1:1 :- Body.
```

The coin example of [21] is represented as (file coin.pl)

```
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.

Moreover, the bodies of rules may contain built-in predicates, predicates from the libraries `lists`, `apply` and `clpr/nf_r` plus the predicate

```
average/2
```

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

The body of rules may also contain the predicate `prob/2` that computes the probability of an atom, thus allowing nested probability computations. For example (`meta.pl`)

```
a:0.2:-
  prob(b,P),
  P>0.2.
```

is a valid rule.

Moreover, the probabilistic annotations can be variables, as in (`flexprob.pl`))

```
red(Prob):Prob.

draw_red(R, G):-
  Prob is R/(R + G),
  red(Prob).
```

Variables in probabilistic annotations must be ground when resolution reaches the end of the body, otherwise an exception is raised.

Alternative ways of specifying probability distribution include

```
A:discrete(Var,D):-Body.
```

or

```
A:finite(Var,D):-Body.
```

where `A` is an atom containg variable `Var` and `D` is a list of couples `Value:Prob` assigning probability `Prob` to `Value`. Moreover, you can use

```
A:uniform(Var,D):-Body.
```

where `A` is an atom containg variable `Var` and `D` is a list of values each taking the same probability (1 over the length of `D`).

1.1.1 ProbLog Syntax

You can also use ProbLog [6] syntax, so a general clause takes the form

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

where the `pi` are numeric expressions.

1.1.2 PRISM Syntax

You can also use PRISM [19] syntax, so a program is composed of a set of regular Prolog rules whose body may contain calls to the `msw/2` predicate (multi-ary switch). A call `msw(term,value)` means that a random variable associated to `term` assumes value `value`. The admissible values for a discrete random variable are specified using facts for the `values/2` predicate of the form

```
values(T,L).
```

where `T` is a term (possibly containing variables) and `L` is a list of values. The distribution over values is specified using directives for `set_sw/2` of the form

```
:- set_sw(T,LP).
```

where `T` is a term (possibly containing variables) and `LP` is a list of probability values. Remember that usually in PRISM each call to `msw/2` refers to a different random variable, i.e., no memoing is performed, differently from the case of LPAD/CP-Logic/ProbLog. This behavior can be changed with the setting `prism_memoization`: if set to `true` then memoization is performed. Its default value is `false`, i.e., no memoization.

For example, the coin example above in PRISM syntax becomes (`coinmsw.pl`)

```
values(throw(_),[heads,tails]).
:- set_sw(throw(fair),[0.5,0.5]).
:- set_sw(throw(biased),[0.6,0.4]).
values(fairness,[fair,biased]).
:- set_sw(fairness,[0.9,0.1]).
res(Coin,R):- toss(Coin),fairness(Coin,Fairness),msw(throw(Fairness),R).
fairness(_Coin,Fairness) :- msw(fairness,Fairness).
toss(coin).
```

1.2 Continuous Probability Densities

`cplint` handles continuous or integer random variables as well with its sampling inference module. To specify a probability density on an argument `Var` of an atom `A` you can used rules of the form

```
A:Density:- Body
```

where `Density` is a special atom identifying a probability density on variable `Var` and `Body` (optional) is a regular clause body. Allowed `Density` atoms are

- `uniform(Var,L,U)`: `Var` is uniformly distributed in $[L, U]$
- `gaussian(Var,Mean,Variance)`: `Var` follows a Gaussian distribution with mean `Mean` and variance `Variance`. The distribution can be multivariate if `Mean` is a list and `Variance` a list of lists representing the mean vector and the covariance matrix. In this case the values of `Var` are lists of real values with the same length as that of `Mean`
- `dirichlet(Var,Par)`: `Var` is a list of real numbers following a Dirichlet distribution with α parameters specified by the list `Par`
- `gamma(Var,Shape,Scale)` `Var` follows a gamma distribution with shape parameter `Shape` and scale parameter `Scale`.
- `beta(Var,Alpha,Beta)` `Var` follows a beta distribution with parameters `Alpha` and `Beta`.
- `poisson(Var,Lambda)` `Var` follows a Poisson distribution with parameter `Lambda` (rate).
- `binomial(Var,N,P)` `Var` follows a binomial distribution with parameters `N` (number of trials) and `P` (success probability).
- `geometric(Var,P)` `Var` follows a geometric distribution with parameter `P` (success probability).
- `exponential(Var,Lambda)` `Var` follows an exponential distribution with parameter `Lambda` (ate, or inverse scale).
- `pascal(Var,R,P)` `Var` follows an exponential distribution with parameters `R` (number of failures) and `P` (success probability).

For example

```
g(X): gaussian(X,0, 1).
```

states that argument `X` of `g(X)` follows a Gaussian distribution with mean 0 and variance 1, while

```
g(X): gaussian(X,[0,0], [[1,0],[0,1]]).
```

states that argument `X` of `g(X)` follows a Gaussian multivariate distribution with mean vector $[0, 0]$ and covariance matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

.

For example, `gaussian_mixture.pl` defines a mixture of two Gaussians:

```
heads:0.6;tails:0.4.
g(X): gaussian(X,0, 1).
h(X): gaussian(X,5, 2).
mix(X) :- heads, g(X).
mix(X) :- tails, h(X).
```

The argument `X` of `mix(X)` follows a distribution that is a mixture of two Gaussian, one with mean 0 and variance 1 with probability 0.6 and one with mean 5 and variance 2 with probability 0.4.

The parameters of the distribution atoms can be taken from the probabilistic atom, the example `gauss_mean_est.pl`

```
val(I,X) :-
  mean(M),
  val(I,M,X).
mean(M): gaussian(M,1.0, 5.0).
val(_,M,X): gaussian(X,M, 2.0).
```

states that for an index `I` the continuous variable `X` is sampled from a Gaussian whose variance is 2 and whose mean is sampled from a Gaussian with mean 1 and variance 5.

Any operation is allowed on continuous random variables. The example below (`kalman_filter.pl`) encodes a Kalman filter:

```
kf(N,O, T) :-
  init(S),
  kf_part(0, N, S,O,T).
kf_part(I, N, S,[V|RO], T) :-
  I < N,
  NextI is I+1,
  trans(S,I,NextS),
  emit(NextS,I,V),
  kf_part(NextI, N, NextS,RO, T).
kf_part(N, N, S, [],S).
trans(S,I,NextS) :-
  {NextS =:= E + S},
  trans_err(I,E).
emit(NextS,I,V) :-
  {NextS =:= V+X},
  obs_err(I,X).
init(S):gaussian(S,0,1).
trans_err(_,E):gaussian(E,0,2).
obs_err(_,E):gaussian(E,0,1).
```

Continuous random variables are involved in arithmetic expressions (in `trans/3` and `emit/3`). It is often convenient, as in this case, to use CLP(R) constraints (by including the directive `:- use_module(library(clpr)).`) as in this way the expressions can be

used in multiple directions and the same clauses can be used both to sample and to evaluate the weight of the sample on the basis of evidence, otherwise different clauses have to be written. In case random variables are not sufficiently instantiated to exploit expressions for inferring the values of other variables, inference will return an error.

1.2.1 Distributional Clauses Syntax

You can also use the syntax of Distributional Clauses (DC) [13]. Continuous random variables are represented in this case by term whose distribution can be specified with density atoms as in

```
T~Density' := Body.
```

Here `:=` replaces the implication symbol, `T` is a term and `Density'` is one of the density atoms above without the `Var` argument, because `T` itself represents a random variables. In the body of clauses you can use the infix operator `~=` to equate a term representing a random variable with a logical variable or a constant as in `T ~= X`. Internally `cplint` transforms the terms representing random variables into atoms with an extra argument for holding the variable. DC can be used to represent also discrete distributions using

```
T~uniform(L) := Body.
T~finite(D) := Body.
```

where `L` is a list of values and `D` is a list of couples `P:V` with `P` a probability and `V` a value. If `Body` is empty, as in regular Prolog, the implication symbol `:=` can be omitted.

The Indian GPA problem from `http://www.robots.ox.ac.uk/~fwood/anglican/examples/viewer/?worksheet=indian-gpa`in distributional clauses syntax (`https://github.com/davidenitti/DC/blob/master/examples/indian-gpa.pl`) takes the form (`indian_gpadc.pl`):

```
is_density_A:0.95;is_discrete_A:0.05.
% the probability distribution of GPA scores for American students is
% continuous with probability 0.95 and discrete with probability 0.05

agpa(A): beta(A,8,2) :- is_density_A.
% the GPA of American students follows a beta distribution if the
% distribution is continuous

american_gpa(G) : finite(G,[4.0:0.85,0.0:0.15]) :- is_discrete_A.
% the GPA of American students is 4.0 with probability 0.85 and 0.0
% with
% probability 0.15 if the
% distribution is discrete
american_gpa(A):- agpa(A0), A is A0*4.0.
% the GPA of American students is obtained by rescaling the value of
% agpa
```

```
% to the (0.0,4.0) interval
is_density_I : 0.99; is_discrete_I:0.01.
% the probability distribution of GPA scores for Indian students is
% continuous with probability 0.99 and discrete with probability
% 0.01
igpa(I): beta(I,5,5) :- is_density_I.
% the GPA of Indian students follows a beta distribution if the
% distribution is continuous
indian_gpa(I): finite(I,[0.0:0.1,10.0:0.9]):-  is_discrete_I.
% the GPA of Indian students is 10.0 with probability 0.9 and 0.0
% with
% probability 0.1 if the
% distribution is discrete
indian_gpa(I) :- igpa(I0), I is I0*10.0.
% the GPA of Indian students is obtained by rescaling the value
% of igpa
% to the (0.0,4.0) interval
nation(N) : finite(N,[a:0.25,i:0.75]).
% the nation is America with probability 0.25 and India with
% probability 0.75
student_gpa(G):- nation(a),american_gpa(G).
% the GPA of the student is given by american_gpa if the nation is
% America
student_gpa(G) :- nation(i),indian_gpa(G).
% the GPA of the student is given by indian_gpa if the nation
%is India
```

See

2 Semantics

The semantics of LPADs for the case of programs without functions symbols can be given as follows. An LPAD defines a probability distribution over normal logic programs called *worlds*. A world is obtained from an LPAD by first grounding it, by selecting a single head atom for each ground clause and by including in the world the clause with the selected head atom and the body. The probability of a world is the product of the probabilities associated to the heads selected. The probability of a ground atom (the query) is given by the sum of the probabilities of the worlds where the query is true.

If the LPAD contains function symbols, the definition is more complex, see [15, 20, 17].

For the semantics of programs with continuous random variables, see [10] that defines the probability space for N continuous random variables by considering the Borel σ-algebra over $\mathbb{R}^N$ and defines a Lebesgue measure on this set as the probability measure. The probability space is lifted to cover the entire program using the least model

semantics of constraint logic programs. Alternatively, [13] defines the semantics of distributional clauses by resorting to a stochastic Tp operator. `cplint` allows more freedom than distributional clauses in the use of continuous random variables in expressions, for example `kalman_filter.pl` would not be allowed by distributional clauses.

3 Inference

`cplint` answers queries using the module `pita` or `mcintyre`. The first performs the program transformation technique of [18]. Differently from that work, techniques alternative to tabling and answer subsumption are used. The latter performs approximate inference by sampling using a different program transformation technique and is described in [16]. Only `mcintyre` is able to handle continuous random variables.

For answering queries, you have to prepare a Prolog file where you first load the inference module (for example `pita`), initialize it with a directive (for example `:- pita`) and then enclose the LPAD clauses in `:-begin_lpad.` or `:-begin_plp.` and `:-end_lpad.` or `:-end_plp.` For example, the coin program above can be stored in `coin.pl` for performing inference with `pita` as follows

```
:- use_module(library(pita)).
:- pita.
:- begin_lpad.
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).
:- end_lpad.
```

The same program for `mcintyre` is

```
:- use_module(library(mcintyre)).
:- mc.
:- begin_lpad.
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).
:- end_lpad.
```

You can have also (non-probabilistic) clauses outside `:-begin/end_lpad.` These are considered as database clauses. In `pita` subgoals in the body of probabilistic clauses can query them by enclosing the query in `db/1`. For example (`testdb.pl`)

```
:- use_module(library(pita)).
:- pita.
:- begin_lpad.
sampled_male(X):0.5:-
  db(male(X)).
:- end_lpad.
male(john).
male(david).
```

You can also use `findall/3` on subgoals defined by database clauses (`persons.pl`)

```
:- use_module(library(pita)).
:- pita.
:- begin_lpad.
male:M/P; female:F/P:-
  findall(Male,male(Male),LM),
  findall(Female,female(Female),LF),
  length(LM,M),
  length(LF,F),
  P is F+M.
:- end_lpad.
male(john).
male(david).
female(anna).
female(elen).
female(cathy).
```

Aggregate predicates on probabilistic subgoals are not implemented due to their high computational cost (if the aggregation is over n atoms, the values of the aggregation are potentially 2^n). The Yap version of `cplint` includes reasoning algorithms that allows aggregate predicates on probabilistic subgoals, see `http://ds.ing.unife.it/~friguzzi/software/cplint/manual.html`.

In `mcintyre` you can query database clauses in the body of probabilistic clauses without any special syntax. You can also use `findall/3`.

3.1 Unconditional Queries

The unconditional probability of an atom can be asked using `pita` with the predicate

```
prob(:Query:atom,-Probability:float) is nondet
```

as in

```
?- prob(heads(coin),P).
```

If the query is non-ground, `prob/2` returns in backtracking the successful instantiations together with their probability.

When using `mcintyre`, the predicate for querying is

```
mc_prob(:Query:atom,-Probability:float,+Options:list) is det
```

where `Options` is a list of options, the following are recognised by `mc_prob/3`:

- `bar(-BarChar:dict)` BarChart is a dict for rendering with c3 as a bar chart with a bar for the number of successes and a bar for the number of failures.

For example

```
?- mc_prob(heads(coin),P,[]).
```

You can also use

```
mc_prob(:Query:atom,-Probability:float) is det
```

which is equivalent to `mc_prob/3` with an empty option list. In general, all the predicates that admit a list of options as an argument have a corresponding version without the list of options that is equivalent to calling the first with an empty option list.

With `mcintyre`, you can also take a given number of samples with

```
mc_sample(:Query:atom,+Samples:int,-Probability:float,
  Options:list) is det
```

where `Options` is a list of options, the following are recognised by `mc_sample/4`:

- `successes(-Successes:int)` Number of succesess
- `failures(-Failures:int)` Number of failures
- `bar(-BarChar:dict) BarChart` is a dict for rendering with c3 as a bar chart with a bar for the number of successes and a bar for the number of failures.

For example (`coinmc.pl`)

```
?- mc_sample(heads(coin),1000,P,[successes(S),failures(F)]).
```

that samples `heads(coin)` 1000 times and returns in `S` the number of successes, in `F` the number of failures and in `P` the estimated probability (`S/1000`). As another example, the call

```
?- mc_sample(heads(coin),1000,Prob).
```

samples `heads(coin)` 1000 times and returns the estimated probability that a sample is true.

You can also sample using Gibbs sampling with

```
mc_gibbs_sample(:Query:atom,+Samples:int,-Probability:float,
  +Options:list) is det
```

where `Options` is a list of options, the following are recognised by `mc_gibbs_sample/4`:

- `block(+Block:int)` Perform blocked Gibbs: `Block` variables are sampled together, default value 1
- `mix(+Mix:int)` The first `Mix` samples are discarded (mixing time), default value 0
- `successes(-Successes:int)` Number of succesess
- `failures(-Failures:int)` Number of failures

`mc_gibbs_sample/3` is equivalent to `mc_gibbs_sample/4` with an empty option list.

Moreover, you can sample arguments of queries with

```
mc_sample_arg(:Query:atom,+Samples:int,?Arg:var,
  -Values:list,+Options:list) is det
```

The predicate samples `Query` a number of `Samples` times. `Arg` should be a variable in `Query`. The predicate returns in `Values` a list of couples `L-N` where `L` is the list of values of `Arg` for which `Query` succeeds in a world sampled at random and `N` is the number of samples returning that list of values. If `L` is the empty list, it means that for that sample the query failed. If `L` is a list with a single element, it means that for that sample the query is determinate. If, in all couples `L-N`, `L` is a list with a single element, it means that the clauses in the program are mutually exclusive, i.e., that in every sample, only one clause for each subgoal has the body true. This is one of the assumptions taken for programs of the PRISM system [20]. For example `pcfglr.pl` and `plcg.pl` satisfy this constraint while `markov_chain.pl` and `var_obj.pl` doesn't.

`Options` is a list of options, the following are recognised by `mc_sample_arg/5`:

- `successes(-Successes:int)` Number of succeses
- `failures(-Failures:int)` Number of failueres
- `bar(-BarChar:dict) BarChart` is a dict for rendering with c3 as a bar chart with with a bar for each possible value of `L`, the list of values of `Arg` for which the query succeeds in a world sampled at random. The size of the bar is the number of samples returning that list of values.

An example of use of `mc_sample_arg/4` is

```
?- mc_sample_arg(reach(s0,0,S),50,S,Values).
```

of `markov_chain.pl` that takes 50 samples of `L` in `findall(S,(reach(s0,0,S),L)`.

You can sample arguments of queries also with

```
mc_sample_arg_raw(:Query:atom,+Samples:int,?Arg:var,
  -Values:list) is det
```

that samples `Query` a number of `Samples` times The predicate returns in `Values` a list of values of `Arg` returned as the first answer by `Query` in a world sampled at random. The value is `failure` if the query fails.

The predicate

```
mc_sample_arg_first(:Query:atom,+Samples:int,?Arg:var,
  -Values:list,+Options:list) is det
```

samples `Query` a number of `Samples` times and returns in `Values` a list of couples `V-N` where `V` is the value of `Arg` returned as the first answer by `Query` in a world sampled at random and `N` is the number of samples returning that value. `V` is failure if the query fails. `mc_sample_arg_first/5` differs from `mc_sample_arg/5` because the first just computes the first answer of the query for each sampled world.

`Options` is a list of options, the following are recognised by `mc_sample_arg_first/5`:

- `bar(-BarChar:dict) BarChart` has a bar for each value of `Arg` returned as a first answer for the query in a world sampled at random. The size of the bar is the number of samples that returned that value.

The predicate

```
mc_sample_arg_one(:Query:atom,+Samples:int,?Arg:var,
  -Values:list,+Options:list) is det
```

samples `Query` a number of `Samples` times and returns in `Values` a list of couples `V-N` where `V` is a value sampled with uniform probability from those returned by `Query` in a world sampled at random and `N` is the number of samples returning that value. `V` is failure if the query fails.

`Options` is a list of options, the following are recognised by `mc_sample_arg_one/5`:

- `bar(-BarChar:dict) BarChart` has a bar for each value of `Arg` returned by sampling with uniform probability one answer from those returned by the query in a world sampled at random. The size of the bar is the number of samples.

The predicate

```
mc_gibbs_sample_arg(:Query:atom,+Samples:int,?Arg:var,-
  Values:list,+Options:list) is det
```

samples an argument of the query using Gibbs sampling. The same options as those of `mc_gibbs_sample/4` are recognized.

Finally, you can compute expectations with

```
mc_expectation(:Query:atom,+N:int,?Arg:var,-Exp:float) is det
```

that computes the expected value of `Arg` in `Query` by sampling. It takes `N` samples of `Query` and sums up the value of `Arg` for each sample. The overall sum is divided by `N` to give `Exp`.

An example of use of the above predicate is

```
?- mc_expectation(eventually(elect,T),1000,T,E).
```

of `pctl_slep.pl` that returns in `E` the expected value of `T` by taking 1000 samples.

The predicate

```
mc_gibbs_expectation(:Query:atom,+N:int,?Arg:var,
  -Exp:float) is det
```

computes an expectation with Gibbs sampling.

3.1.1 Drawing BDDs

With `pita`, you can obtain the BDD for a query with the predicates

```
bdd_dot_file(:Query:atom,+FileName:string,-Var:list) is nondet
bdd_dot_string(:Query:atom,-DotString:string,-Var:list) is nondet
```

The first write the BDD to a file, the latter returns it as a string. The BDD is represented in the dot format of graphviz. Solid edges indicate 1-children, dashed edges indicate 0-children and dotted edges indicate 0-children with negation applied to the sub BDD. Each level of the BDD is associated to a variable of the form XI_J indicated on the left: I indicates the multivalued variable index and J the index of the Boolean variable of rule I. The hexadecimal number in each node is part of its address in memory and is not significant. The table `Var` contains the associations between the rule groundings and the multivalued variables: the first column contains contains the multivalued variable index, the second column contains the rule index, corresponding to its position in the program, and the last column contains the list of constants grounding the rule, each replacing a variable in the order of appearance in the rule.

The BDD can be drawn in `cplint` on SWISH by using the `graphviz` renderer by including

```
:- use_rendering(graphviz).
```

before `:- pita.`

For example (`coin.pl`)

```
?- bdd_dot_string(heads(coin),BDD,Var).
```

returns the BDD for the query `heads(coin)` and the list of associations between rule groundings and multivalued variables.

3.2 Conditional Queries on Discrete Variables

The conditional probability of an atom query given another atom evidence can be asked using `pita` with the predicate

```
prob(:Query:atom,:Evidence:atom,-Probability:float) is nondet
```

as in

```
?- prob(heads(coin),biased(coin),P).
```

If the query/evidence are non-ground, `prob/3` returns in backtracking ground instantiations together with their probability. The query and the evidence can be conjunctions of literals (positive or negative).

You also have

```
prob(:Query:atom,:Evidence:atom,-Probability:float,
  +Options:list) is nondet
```

where `Options` is a list of options, the following are recognised by `prob/4`:

- `bar(-BarChar:dict) BarChart` is a dict for rendering with c3 as a bar chart with a bar for the number of successes and a bar for the number of failures.

as in

```
?- prob(heads(coin),biased(coin),P,[bar(Chart)]).
```

`prob/3` is equivalent to `prob/4` with an empty option list.

When using `mcintyre`, you can ask conditional queries with rejection sampling, Metropolis-Hastings Markov Chain Monte Carlo or Gibbs sampling. In rejection sampling [22], you first query the evidence and, if the query is successful, query the goal in the same sample, otherwise the sample is discarded. In Metropolis-Hastings MCMC, `mcintyre` follows the algorithm proposed in [12] (the non adaptive version). A Markov chain is built by building an initial sample and by generating successor samples.

The initial sample is built by randomly sampling choices so that the evidence is true. This is done with a backtracking meta-interpreter that starts with the goal and randomizes the order in which clauses are selected during the search so that the initial sample is unbiased. Each time the meta-interpreter encounters a probabilistic choice, it first checks whether a value has already been sampled, if not, it takes a sample and records it. If a failure is obtained, the meta-interpreter backtracks to other clauses but without deleting samples. Then the goal is queries using regular MCINTYRE.

A successor sample is obtained by deleting a fixed number (parameter `Lag`) of sampled probabilistic choices. Then the evidence is queried using regular MCINTYRE starting with the undeleted choices. If the query succeeds, the goal is queried using regular MCINTYRE. The sample is accepted with a probability of $\min\{1, \frac{N_0}{N_1}\}$ where N_0 is the number of choices sampled in the previous sample and N_1 is the number of choices sampled in the current sample. In [12] the lag is always 1 but the proof in [12] that the above acceptance probability yields a valid Metropolis-Hastings algorithm holds also when forgetting more than one sampled choice, so the lag is user defined in `cplint`.

Then the number of successes of the query is increased by 1 if the query succeeded in the last accepted sample. The final probability is given by the number of successes over the total number of samples.

You can take a given number of sample with rejection sampling using

```
mc_rejection_sample(:Query:atom,:Evidence:atom,+Samples:int,
  -Probability:float,+Options:list) is det
```

where `Options` is a list of options, the following are recognised by `mc_sample_arg/5`:

- `successes(-Successes:int)` Number of succeses
- `failures(-Failures:int)` Number of failueres

as in (`coinmc.pl`)

```
?- mc_rejection_sample(heads(coin),biased(coin),1000,P,
  [successes(S),failures(F)]).
```

that takes 1000 samples where `biased(coin)` is true and returns in `S` the number of samples where `heads(coin)` is true, in `F` the number of samples where `heads(coin)` is false and in `P` the estimated probability (`S/1000`).

The query and the evidence can be conjunctions of literals.

You can take a given number of sample with Metropolis-Hastings MCMC using

```
mc_mh_sample(:Query:atom,:Evidence:atom,+Samples:int,
  -Probability:float,+Options:list) is det
```

where `Lag` (that is set with the options, default value 1) is the number of sampled choices to forget before taking a new sample.

`Options` is a list of options, the following are recognised by `mc_mh_sample/5`:

- `mix(+Mix:int)` The first Mix samples are discarded (mixing time), default value 0
- `lag(+Lag:int)` lag between each sample, Lag sampled choices are forgotten, default value 1
- `successes(-Successes:int)` Number of succeses
- `failures(-Failures:int)` Number of failueres
- `bar(-BarChar:dict)` BarChart is a dict for rendering with c3 as a bar chart with a bar for the number of successes and a bar for the number of failures.

With `Mix` specified it takes `Mix+Samples` samples and discards the first `Mix`.

For example (`arithm.pl`)

```
?- mc_mh_sample(eval(2,4),eval(1,3),10000,P,
  [successes(T), failures(F)]).
```

takes 10000 accepted samples and returns in `T` the number of samples where `eval(2,4)` is true, in `F` the number of samples where `eval(2,4)` is false and in `P` the estimated probability (`T/10000`).

The predicate

```
mc_gibbs_sample(:Query:atom,:Evidence:atom,+Samples:int,
  -Probability:float,+Options:list) is det
```

performs Gibbs sampling. `Options` is a list of options, the following are recognised by `mc_gibbs_sample/5`:

- `block(+Block:int)` Perform blocked Gibbs: `Block` variables are sampled together, default value 1
- `mix(+Mix:int)` The first `Mix` samples are discarded (mixing time), default value 0
- `successes(-Successes:int)` Number of succesess
- `failures(-Failures:int)` Number of failures

Moreover, you can sample arguments of queries with rejection sampling, Metropolis-Hastings MCMC or Gibbs sampling using

```
mc_rejection_sample_arg(:Query:atom,:Evidence:atom,
  +Samples:int,?Arg:var,-Values:list,+Options:list) is det
mc_mh_sample_arg(:Query:atom,:Evidence:atom,
  +Samples:int,?Arg:var,-Values:list,+Options:list) is det
mc_gibbs_sample_arg(:Query:atom,+Samples:int,
  ?Arg:var,-Values:list,+Options:list) is det
```

that return the distribution of values for `Arg` in `Query` in `Samples` of `Query` given that `Evidence` is true. `Options` is a list of options, the following are recognised:

- `mix(+Mix:int)` The first `Mix` samples are discarded (mixing time), default value 0 (only MH and GIbbs)
- `lag(+Lag:int)` lag between each sample, `Lag` sampled choices are forgotten, default value 1 (only MH)
- `block(+Block:int)` Perform blocked Gibbs: `Block` variables are sampled together, default value 1 (only Gibbs)
- `bar(-BarChar:dict)` `BarChart` is a bar chart of the possible values

The predicates return in `Values` a list of couples `L-N` where `L` is the list of values of `Arg` for which `Query` succeeds in a world sampled at random where `Evidence` is true and `N` is the number of samples returning that list of values.

```
mc_gibbs_sample_arg(:Query:atom,+Samples:int,
  ?Arg:var,-Values:list,+Options:list) is det
```

An example of use of the above predicates is

```
?- mc_mh_sample_arg(eval(2,Y),eval(1,3),1000,Y,V,[]).
```

of `arithm.pl`.

To compute conditional expectations, use

```
mc_rejection_expectation(:Query:atom,:Evidence:atom,+N:int,
  ?Arg:var,-Exp:float) is det
mc_mh_expectation(:Query:atom,:Evidence:atom,+N:int,
  ?Arg:var,-Exp:float,+Options:list) is det
mc_gibbs_expectation(:Query:atom,:Evidence:atom,+N:int,
  ?Arg:var,-Exp:float,+Options:list) is det
```

where `Options` is a list of options, the same as those of the predicates for conditional argument sampling are recognised. For example

```
?- mc_mh_expectation(eval(2,Y),eval(1,3),1000,Y,E,[]).
```

of `arithm.pl` computes the expectation of argument `Y` of `eval(2,Y)` given that `eval(1,3)` is true by taking 1000 samples using Metropolis-Hastings MCMC.

Note that conditional inference is not allowed for PRISM programs with the setting `prism_memoization` set to `false`, as sampled values are not stored in that case and conditioning would have no effect.

3.3 Conditional Queries on Continuous Variables

When you have continuous random variables, you may be interested in sampling arguments of goals representing continuous random variables. In this way you can build a probability density of the sampled argument. When you do not have evidence or you have evidence on atoms not depending on continuous random variables, you can use the above predicates for sampling arguments.

For example

```
?- mc_sample_arg(val(0,X),1000,X,L).
```

from (`gauss_mean_est.pl`)) samples 1000 values for `X` in `value(0,X)` and returns them in `L`.

When you have evidence on ground atoms that have continuous values as arguments, you cannot use rejection sampling or Metropolis-Hastings, as the probability of the evidence is 0. For example, the probability of sampling a specific value from a Gaussian is 0. Continuous variables have probability densities instead of distributions as discrete variables. In this case, you can use likelihood weighting or particle filtering [9, 11, 13] to obtain samples of continuous arguments of a goal.

For each sample to be taken, likelihood weighting uses a meta-interpreter to find a sample where the goal is true, randomizing the choice of clauses when more than one resolves with the goal in order to obtain an unbiased sample. This meta-interpreter is similar to the one used to generate the first sample in Metropolis-Hastings.

Then a different meta-interpreter is used to evaluate the weight of the sample. This meta-interpreter starts with the evidence as the query and a weight of 1. Each time the meta-interpreter encounters a probabilistic choice over a continuous variable, it first checks whether a value has already been sampled. If so, it computes the probability density of the sampled value and multiplies the weight by it. If the value has not been sampled, it takes a sample and records it, leaving the weight unchanged. In this way,

each sample in the result has a weight that is 1 for the prior distribution and that may be different from the posterior distribution, reflecting the influence of evidence.

In particle filtering, the evidence is a list of atoms. Each sample is weighted by the likelihood of an element of the evidence and constitutes a particle. After weighting, particles are resampled and the next element of the evidence is considered.

The predicate

```
mc_lw_sample(:Query:atom,:Evidence:atom,+Samples:int,
  -Prob:float) is det
```

samples `Query` a number of `Samples` times given that `Evidence` (a conjunction of atoms is allowed here) is true. The predicate returns in `Prob` the probability that the query is true. It performs likelihood weighting: each sample is weighted by the likelihood of evidence in the sample. For example

```
?- mc_lw_sample(nation(a),student_gpa(4.0),1000,P).
```

from `indian_gpa.pl` samples 1000 times the query `nation(a)` given that `student_gpa(4.0)` has been observed.

The predicate

```
mc_lw_sample_arg(:Query:atom,:Evidence:atom,+N:int,?Arg:var,
  -ValList) is det
```

returns in `ValList` a list of couples `V-W` where `V` is a value of `Arg` for which `Query` succeeds and `W` is the weight computed by likelihood weighting according to `Evidence` (a conjunction of atoms is allowed here). For example

```
?- mc_lw_sample_arg(val(0,X),(val(1,9),val(2,8)),100,X,L).
```

from `gauss_mean_est.pl` samples 100 values for `X` in `val(0,X)` given that `val(1,9)` and `val(2,8)` have been observed.

You can compute conditional expectations using likelihood weighting with

```
mc_lw_expectation(:Query:atom,Evidence:atom,+N:int,?Arg:var,
  -Exp:float) is det
```

that computes the expected value of `Arg` in `Query` given that `Evidence` is true. It takes `N` samples of `Arg` in `Query`, weighting each according to the evidence, and returns their weighted average.

The predicate

```
mc_particle_sample_arg(:Query:atom,+Evidence:list,
  +Samples:int,?Arg:var,-Values:list) is det
```

samples argument `Arg` of `Query` using particle filtering given that `Evidence` is true. `Evidence` is a list of goals and `Query` can be either a single goal or a list of goals. When `Query` is a single goal, the predicate returns in `Values` a list of couples `V-W` where `V` is a value of `Arg` for which `Query` succeeds in a particle in the last set of particles and `W` is

the weight of the particle. For each element of `Evidence`, the particles are obtained by sampling `Query` in each current particle and weighting the particle by the likelihood of the evidence element.

When `Query` is a list of goals, `Arg` is a list of variables, one for each query of `Query` and `Arg` and `Query` must have the same length of `Evidence`. `Values` is then list of the same length of `Evidence` and each of its elements is a list of couples `V-W` where `V` is a value of the corresponding element of `Arg` for which the corresponding element of `Query` succeeds in a particle and `W` is the weight of the particle. For each element of `Evidence`, the particles are obtained by sampling the corresponding element of `Query` in each current particle and weighting the particle by the likelihood of the evidence element.

For example

```
?-[O1,O2,O3,O4]=[-0.133, -1.183, -3.212, -4.586],
mc_particle_sample_arg([kf_fin(1,T1),kf_fin(2,T2),kf_fin(3,T3),
  kf_fin(4,T4)],
  [kf_o(1,O1),kf_o(2,O2),kf_o(3,O3),kf_o(4,O4)],100,
  [T1,T2,T3,T4],[F1,F2,F3,F4]).
```

from `kalman_filter.pl` performs particle filtering for a Kalman filter with four observations. For each observation, the value of the state at the same time point is sampled. The list of samples is returned in `[F1,F2,F3,F4]`, with each element being the sample for a time point.

The predicate

```
mc_particle_sample(:Query:atom,:Evidence:list,
  +Samples:int,-Prob:float) is det
```

samples `Query` a number of `Samples` times given that `Evidence` is true using particle filtering. `Evidence` is a list of goals. The predicate returns in `Prob` the probability that the query is true.

You can compute conditional expectations using particle filtering with

```
mc_particle_expectation(:Query:atom,Evidence:atom,+N:int,
  ?Arg:var,-Exp:float) is det
```

that computes the expected value of `Arg` in `Query` given that `Evidence` is true. It uses `N` particles.

3.4 Causal Inference

`pita` and `mcintyre` support causal reasoning, i.e., computing the effect of actions using the do-calculus [14].

Actions in this setting are represented as literals of action predicates, that must be declared as such with the directive

```
:- action predicate1/arity1,...,predicaten/arityn.
```

When performing causal reasoning, action literals must be enclosed in the `do/1` functor and included in the evidence conjunction. More than one action can be included (each with in a separate `do/1` term) and actions and observations can be freely mixed. All conditional inference goals can be used except those for particle filtering.

For example

```
?- prob(recovery,do(drug),P).
```

from `simpson.swinb` computes the probability of recovery of a patient given that the action of administering a drug has been performed.

3.5 Graphing the Results

In `cplint` on SWISH you can draw graphs for visualizing the results either with C3.js or with R. Similar predicates are avaiiable for the two methods. There are two types of graphs: those that represent individual probability values with a bar chart and those that visualize the results of sampling arguments.

3.5.1 Using C3.js

You can draw the probability of a query being true and being false as a bar chart using the predicates

```
bar1(+Probability:float,-Chart:dict) is det
bar(+Probability:float,-Chart:dict) is det
bar(+Successes:int,+Failures:int,-Chart:dict) is det
argbar(+Values:list,-Chart:dict) is det
```

They return a dict for rendering with C3.js as a bar chart: the first returns bar chart with a single bar for the probability, the second a chart with bar for the probability and a bar for one minus the probability, the third a chart with a bar for the number of successes and a bar for the number of failures, and the fourth a chart with a for bar each value, where `Values` is a list of couples `V-N` where `V` is the value and `N` is the number of samples returning that value.

To render C3.js charts you have to include

```
:- use_rendering(c3).
```

before `:- pita.`

You can also use the `bar(-Chart:dict)` option of many predicates as in

```
?- prob(heads(coin),biased(coin),P,[bar(Chart)]).
```

`P` will be instantiated with a chart with a bar for the probability of `heads(coin)` true and a bar for the probability of `heads(coin)` false, given that `biased(coin)` is true.

Another example is

```
?- mc_prob(heads(coin),P,[bar(Chart)]).
```

that returns a chart representation of the probability.

```
?- mc_sample(heads(coin),1000,P,[bar(Chart)]).
```

returns in `Chart` a diagram with one bar for the number of successes and one bar for the number of failures.

The options of `mc_sample_arg/5`, `mc_sample_arg_first/5`, `mc_mh_sample_arg/6`, `mc_rejection_sample_arg/6`, can be used for visualizing the results of sampling arguments.

An example is

```
?- mc_sample_arg(reach(s0,0,S),50,S,ValList,[bar(Chart)]).
```

of `markov_chain.pl`.

The same result can be achieved with

```
?- mc_sample_arg(reach(s0,0,S),50,S,ValList),argbar(ValList,Chart)
```

Drawing a graph is particularly interesting when sampling values for continuous arguments of goals. In this case, you can use the samples to draw the probability density function of the argument. The predicate

```
histogram(+List:list,-Chart:dict,+Options:list) is det
```

draws a histogram of the samples in `List` that must be a list of couples of the form `[V]-W` or `V-W` where `V` is a sampled value and `W` is its weight. This is the format of the list of samples returned by argument sampling predicates.

The predicate

```
density(+List:list,-Chart:dict,+Options:list) is det
```

draws a line chart of the density of the samples in `List` that must take the same form as for `histogram/3`.

In `histogram/3` and `density/3` `Options` is a list of options, the following are recognised:

- `min(+Min:float)` the minimum value of domain, default value the minimum in `List`
- `max(+Max:float)` the maximum value of domain, default value the maximum in `List`
- `nbins(+NBins:int)` the number of bins for dividing the domain, default value 40

In this way you can specify the limits and the number of intervals of the X.

The predicate

```
densities(+PriorList:list,+PostList:list,-Chart:dict,
  +Options:list) is det
```

draws a line chart of the density of two sets of samples, usually prior and post observations. The samples in `PriorList` and `PostList` can be either couples `[V]-W` or `V-W` where `V` is a value and `W` its weight. The same options as for `histogram/3` and `density/3` are recognized.

For example, the query

```
?-  mc_sample_arg(value(0,X),1000,X,L0,[]),
    histogram(L0,Chart,[]).
```

from `gauss_mean_est.pl`, takes 1000 samples of argument `X` of `value(0,X)` and draws the density of the samples using an histogram.

Instead

```
?- mc_sample_arg(value(0,Y),1000,Y,L0,[]),
   mc_lw_sample_arg(value(0,X),
    (value(1,9),value(2,8)),1000,X,L),
   densities(L0,L,Chart).
```

from `gauss_mean_est.pl` takes 1000 amples of argument `X` of `value(0,X)` before and after observing `(value(1,9),value(2,8)` and draws the prior and posterior densities of the samples using a line chart.

Predicates `histogram/3`, `density/3` and `densities/4` each have a version with one argument less that is equivalent to the predicate called with an empty option list.

3.5.2 Using R

You have to load library `cplint_r` (a SWI-Prolog pack) with

```
:- use_module(library(cplint_r)).
```

Then you can use predicates

```
bar_r/1
bar_r/2
argbar_r/1
```

that work as their C3.js counterpart but do not return the graph as an argument as the graph is printed with a different mechanism.

You also have

```
histogram_r(+List:list,+Options:list) is det
```

that works as `histogram/3`.

```
density_r(+List:list) is det
```

is like `density/3` with the number of bins is determined by R.

```
densities_r(+PriorList:list,+PostList:list) is det
```

is like `densities/3` with the number of bins is determined by R.

See `gauss_mean_est_R.pl` for an example of use of these predicates.

ml

3.6 Parameters

The inference modules have a number of parameters in order to control their behavior. They can be set with the directive

```
:- set_pita(<parameter>,<value>).
```

or

```
:- set_mc(<parameter>,<value>).
```

after initialization (`:-pita.` or `:-mc.`) but outside `:-begin/end_lpad.` The current value can be read with

```
?- setting_pita(<parameter>,Value).
```

or

```
?- setting_mc(<parameter>,Value).
```

from the top-level. The available parameters common to both `pita` and `mcintyre` are:

- `epsilon_parsing`: if (1 - the sum of the probabilities of all the head atoms) is larger than `epsilon_parsing`, then `pita` adds the null event to the head. Default value `0.00001`.
- `single_var`: determines how non ground clauses are treated: if `true`, a single random variable is assigned to the whole non ground clause, if `false`, a different random variable is assigned to every grounding of the clause. Default value `false`.

Moreover, `pita` has the parameters

- `depth_bound`: if `true`, the depth of the derivation of the goal is limited to the value of the `depth` parameter. Default value `false`.
- `depth`: maximum depth of derivations when `depth_bound` is set to `true`. Default value `5`.
- `prism_memoization`: `false`: original prism semantics, `true`: semantics with memoization

If `depth_bound` is set to `true`, derivations are depth-bounded so you can query also programs containing infinite loops, for example programs where queries have an infinite number of explanations. However the probability that is returned is guaranteed only to be a lower bound, see for example `markov_chaindb.pl`

`mcintyre` has the parameters

- `min_error`: minimal width of the binomial proportion confidence interval for the probability of the query. When the confidence interval for the probability of the query is below `min_error`, the computation stops. Default value `0.01`.

- `k`: the number of samples to take before checking whether the the binomial proportion confidence interval is below `min_error`. Default value `1000`. `max_samples`: the maximum number of samples to take. This is used when the probability of the query is very close to 0 or 1. In fact `mcintyre` also checks for the validity of the the binomial proportion confidence interval: if less than 5 failures or successes are sampled, even if the width of the confidence interval is less than `min_error`, the computation continues. This would lead to non-termination in cases where the probability is 0 or 1. `max_samples` ensures termination. Default value `10e4`.

- `prism_memoization`: `false`: original prism semantics, `true`: semantics with memoization

The example `markov_chain.pl` shows that `mcintyre` can perform inference in presence of an infinite number of explanations for the goal. Differently from `pita`, no depth bound is necessary, as the probability of selecting the infinite computation branch is 0. However, also `mcintyre` may not terminate if loops not involving probabilistic predicates are present.

If you want to set the seed of the random number generator, you can use SWI-Prolog predicates `setrand/1` and `getrand/1`, see SWI-Prolog manual.

3.7 Tabling

You can also use tabling in inference to speed up the computation and/or avoid loops, see the SWI-Prolog manual.

To do so you have to use the `tabling` library module and declare some of the predicates as tabled. The tabling declarations go after the `:-pita.` or `:- mc.` directives.

For example, to compute the probability of paths in undirected graphs you can use the program (`path_tabling.swinb`)

```
:- use_module(library(pita)).
:- use_module(library(tabling)).
:- pita.
:- table path/2.
:- begin_lpad.
path(X,X).
path(X,Y):-
  path(X,Z),edge(Z,Y).
edge(X,Y):-arc(X,Y).
edge(X,Y):-arc(Y,X).
arc(a,b):0.2.
arc(b,e):0.5.
arc(a,c):0.3.
arc(c,d):0.4.
arc(d,e):0.4.
arc(a,e):0.1.
```

```
:- end_lpad.
```

Then you can compute the probability that `a` and `e` are connected with

```
prob(path(a,e),Prob).
```

This programs has loops so if you run the above query without tabling `pita` would loop forever.

You can use tabling with both `pita` and `mcintyre`.

4 Learning

The following learning algorithms are available:

- EMBLEM (EM over Bdds for probabilistic Logic programs Efficient Mining): an implementation of EM for learning parameters that computes expectations directly on BDDs [3], [1], [2]
- SLIPCOVER (Structure LearnIng of Probabilistic logic programs by searChing OVER the clause space): an algorithm for learning the structure of programs by searching the clause space and the theory space separately [4]
- LEMUR (LEarning with a Monte carlo Upgrade of tRee search): an algorithm for learning the structure of programs by searching the clase space using Monte-Carlo tree search [8]

4.1 Input

To execute the learning algorithms, prepare a Prolog file divided in five parts

- preamble
- background knowledge, i.e., knowledge valid for all interpretations
- LPAD/CPL-program for you which you want to learn the parameters (optional)
- language bias information
- example interpretations

The preamble must come first, the order of the other parts can be changed.

For example, consider the Bongard problems of [7]. `bongard.pl` and `bongardkeys.pl` represent a Bongard problem for SLIPCOVER. `bongard.pl` and `bongardkeys.pl` represent a Bongard problem for LEMUR.

4.1.1 Preamble

In the preamble, the SLIPCOVER library is loaded with (see `bongard.pl`):

```
:- use_module(library(slipcover)).
```

Now you can initialize SLIPCOVER with

```
:- sc.
```

At this point you can start setting parameters for SLIPCOVER such as for example

```
:- set_sc(megaex_bottom,20).
:- set_sc(max_iter,2).
:- set_sc(max_iter_structure,5).
:- set_sc(verbosity,1).
```

We will see later the list of available parameters.

In the preamble, the LEMUR library is loaded with (see `bongard.pl`):

```
:- use_module(library(lemur)).
```

Now you can initialize LEMUR with

```
:- lemur.
```

At this point you can start setting parameters for LEMUR such as for example

```
:- set_lm(verbosity,1).
```

A parameter that is particularly important for both SLIPCOVER and LEMUR is `verbosity`: if set to 1, nothing is printed and learning is fastest, if set to 3 much information is printed and learning is slowest, 2 is in between. This ends the preamble.

4.1.2 Background and Initial LPAD/CPL-program

Now you can specify the background knowledge with a fact of the form

```
bg(<list of terms representing clauses>).
```

where the clauses must be deterministic. Alternatively, you can specify a set of clauses by including them in a section between `:- begin_bg.` and `:- end_bg.` For example

```
:- begin_bg.
replaceable(gear).
replaceable(wheel).
replaceable(chain).
not_replaceable(engine).
not_replaceable(control_unit).
component(C):-
  replaceable(C).
component(C):-
  not_replaceable(C).
:- end_bg.
```

from the `mach.pl` example. If you specify both a `bg/1` fact and a section, the clauses of the two will be combined.

Moreover, you can specify an initial program with a fact of the form

```
in(<list of terms representing clauses>).
```

The initial program is used in parameter learning for providing the structure. Remember to enclose each clause in parentheses because `:-` has the highest precedence.

For example, `bongard.pl` has the initial program

```
in([(pos:0.197575 :-
       circle(A),
       in(B,A)),
    (pos:0.000303421 :-
       circle(A),
       triangle(B)),
    (pos:0.000448807 :-
       triangle(A),
       circle(B))]).
```

Alternatively, you can specify an input program in a section between `:- begin_in.` and `:- end_in.`, as for example

```
:- begin_in.
pos:0.197575 :-
  circle(A),
  in(B,A).
pos:0.000303421 :-
  circle(A),
  triangle(B).
pos:0.000448807 :-
  triangle(A),
  circle(B).
:- end_in.
```

If you specify both a `in/1` fact and a section, the clauses of the two will be combined.

The annotations of the head atoms of the initial program can be probabilities, as in the example above, in this case the parameters do not matter as they are first randomized. The type of randomization depends on the setting `alpha`. If it takes value 0, a truncated Dirichlet process is used to initialize the parameters: the probability of being true of each Boolean random variable used to represent multivalued random variables is sampled and independently uniformly in [0,1].

If it takes a value ≥ 0, the parameters are sampled from a symmetric Dirichlet distribution, i.e. a Dirichlet distribution with vector of parameters $(\alpha, \ldots, \alpha)$.

The annotations of the head atoms of the initial program can also be `p(<prob>)` with `<prob>` a probability, in this case the parameter is fixed so it is not tuned by learning, as in

```
in([(pos:0.5 :-
  circle(A),
  in(B,A)),
     (pos:p(0.5) :-
  circle(A),
  triangle(B))]).
```

The annotations of the head atoms of the initial program can also be `t(<prob>,<args>)` with `<prob>` either a probability, in this case it is the initial value of the parameter, or a variable, in this case the parameter is initially randomized, and `<args>` a tuple of variables that also appear in the clause. In this case a different parameter is learned for every grounding of `<args>` that make the body true.

For example, we can set the initial value of the parameter of the second clause to 0.9 with

```
in([(pos:0.5 :-
  circle(A),
  in(B,A)),
     (pos:t(0.9) :-
  circle(A),
  triangle(B))]).
```

With the program below we learn a different parameter for every instantiation of `C` in the second clause:

```
in([(pos:0.5 :-
  circle(A),
  in(_B,A)),
     (pos:t(_,C) :-
  triangle(A),
  config(A,C))]).
```

4.1.3 Language Bias

The language bias part contains the declarations of the input and output predicates. Output predicates are declared as

```
output(<predicate>/<arity>).
```

and indicate the predicate whose atoms you want to predict. Derivations for the atoms for this predicates in the input data are built by the system. These are the predicates for which new clauses are generated.

Input predicates are those whose atoms you are not interested in predicting. You can declare closed world input predicates with

```
input_cw(<predicate>/<arity>).
```

For these predicates, the only true atoms are those in the interpretations and those derivable from them using the background knowledge, the clauses in the input or in the hypothesized program are not used to derive atoms for these predicates. Moreover, clauses of the background knowledge that define closed world input predicates and that call an output predicate in the body will not be used for deriving examples.

Open world input predicates are declared with

```
input(<predicate>/<arity>).
```

In this case, if a subgoal for such a predicate is encountered when deriving a subgoal for the output predicates, both the facts in the interpretations, those derivable from them and the background knowledge, the background clauses and the clauses of the input program are used.

Then, you have to specify the language bias by means of mode declarations in the style of Progol.

```
modeh(<recall>,<predicate>(<arg1>,...)).
```

specifies the atoms that can appear in the head of clauses, while

```
modeb(<recall>,<predicate>(<arg1>,...)).
```

specifies the atoms that can appear in the body of clauses. `<recall>` can be an integer or `*`. `<recall>` indicates how many atoms for the predicate specification are retained in the bottom clause during a saturation step. `*` stands for all those that are found. Otherwise the indicated number is randomly chosen.

For SLIPCOVER, two specialization modes are available: `bottom` and `mode`. In the first, a bottom clause is built and the literals to be added during refinement are taken from it. In the latter, no bottom clause is built and the literals to be added during refinement are generated directly from the mode declarations. LEMUR has only specialization `mode`.

Arguments of the form

```
+<type>
```

specifies that the argument should be an input variable of type `<type>`, i.e., a variable replacing a `+<type>` argument in the head or a `-<type>` argument in a preceding literal in the current hypothesized clause.

Another argument form is

```
-<type>
```

for specifying that the argument should be a output variable of type `<type>`. Any variable can replace this argument, either input or output. The only constraint on output variables is that those in the head of the current hypothesized clause must appear as output variables in an atom of the body.

Other forms are

```
#<type>
```

for specifying an argument which should be replaced by a constant of type `<type>` in the bottom clause but should not be used for replacing input variables of the following literals when building the bottom clause or

`-#<type>`

for specifying an argument which should be replaced by a constant of type `<type>` in the bottom clause and that should be used for replacing input variables of the following literals when building the bottom clause.

`<constant>`

for specifying a constant.

Note that arguments of the form `#<type> -#<type>` are not available in specialization mode `mode`, if you want constants to appear in the literals you have to indicate them one by one in the mode declarations.

An example of language bias for the Bongard domain is

```
output(pos/0).

input_cw(triangle/1).
input_cw(square/1).
input_cw(circle/1).
input_cw(in/2).
input_cw(config/2).

modeh(*,pos).
modeb(*,triangle(-obj)).
modeb(*,square(-obj)).
modeb(*,circle(-obj)).
modeb(*,in(+obj,-obj)).
modeb(*,in(-obj,+obj)).
modeb(*,config(+obj,-#dir)).
```

SLIPCOVER and LEMUR also require facts for the `determination/2` Aleph-style predicate that indicate which predicates can appear in the body of clauses. For example

```
determination(pos/0,triangle/1).
determination(pos/0,square/1).
determination(pos/0,circle/1).
determination(pos/0,in/2).
determination(pos/0,config/2).
```

state that `triangle/1` can appear in the body of clauses for `pos/0`.

SLIPCOVER and LEMUR also allow mode declarations of the form

`modeh(<r>,[<s1>,...,<sn>],[<a1>,...,<an>],[<P1/Ar1>,...,<Pk/Ark>]).`

These mode declarations are used to generate clauses with more than two head atoms. In them, `<s1>,...,<sn>` are schemas, `<a1>,...,<an>` are atoms such that `<ai>` is obtained from `<si>` by replacing placemarkers with variables, `<Pi/Ari>` are the predicates admitted in the body. `<a1>,...,<an>` are used to indicate which variables should be shared by the atoms in the head. An example of such a mode declaration (from `uwcselearn.pl`) is

```
modeh(*,
[advisedby(+person,+person),tempadvisedby(+person,+person)],
[advisedby(A,B),tempadvisedby(A,B)],
[professor/1,student/1,hasposition/2,inphase/2,
publication/2,taughtby/3,ta/3,courselevel/2,yearsinprogram/2]).
```

If you want to specify negative literals for addition in the body of clauses, you should define a new predicate in the background as in

```
not_worn(C):-
  component(C),
  \+ worn(C).
one_worn:-
  worn(_).
none_worn:-
  \+ one_worn.
```

from `mach.pl` and add the new predicate in a `modeb/2` fact

```
modeb(*,not_worn(-comp)).
modeb(*,none_worn).
```

Note that successful negative literals do not instantiate the variables, so if you want a variable appearing in a negative literal to be an output variable you must instantiate before calling the negative literals. The new predicates must also be declared as input

```
input_cw(not_worn/1).
input_cw(none_worn/0).
```

Lookahead can also be specified with facts of the form

```
lookahead(<literal>,<list of literals>).
```

In this case when a literal matching `<literal>` is added to the body of clause during refinement, then also the literals matching `<list of literals>` will be added. An example of such declaration (from `muta.pl`) is

```
lookahead(logp(_),[(_=_)]).
```

Note that `<list of literals>` is copied with `copy_term/2` before matching, so variables in common between `<literal>` and `<list of literals>` may not be in common in the refined clause.

It is also possible to specify that a literal can only be added together with other literals with facts of the form

```
lookahead_cons(<literal>,<list of literals>).
```

In this case `<literal>` is added to the body of clause during refinement only together with literals matching `<list of literals>`. An example of such declaration is

```
lookahead_cons(logp(_),[(_=_)]).
```

Also here `<list of literals>` is copied with `copy_term/2` before matching, so variables in common between `<literal>` and `<list of literals>` may not be in common in the refined clause.

Moreover, we can specify lookahead with

```
lookahead_cons_var(<literal>,<list of literals>).
```

In this case `<literal>` is added to the body of clause during refinement only together with literals matching `<list of literals>` and `<list of literals>` is not copied before matching, so variables in common between `<literal>` and `<list of literals>` are in common also in the refined clause. This is allowed only with `specialization` set to `bottom`. An example of such declaration is

```
lookahead_cons_var(logp(B),[(B=_)]).
```

4.1.4 Example Interpretations

The last part of the file contains the data. You can specify data with two modalities: models and keys. In the models type, you specify an example model (or interpretation or megaexample) as a list of Prolog facts initiated by `begin(model(<name>)).` and terminated by `end(model(<name>)).` as in

```
begin(model(2)).
pos.
triangle(o5).
config(o5,up).
square(o4).
in(o4,o5).
circle(o3).
triangle(o2).
config(o2,up).
in(o2,o3).
triangle(o1).
config(o1,up).
end(model(2)).
```

The interpretations may contain a fact of the form

```
prob(0.3).
```

assigning a probability (0.3 in this case) to the interpretations. If this is omitted, the probability of each interpretation is considered equal to $1/n$ where n is the total number of interpretations. `prob/1` can be used to set a different multiplicity for the interpretations.

The facts in the interpretation are loaded in SWI-Prolog database by adding an extra initial argument equal to the name of the model. After each interpretation is loaded, a fact of the form `int(<id>)` is asserted, where `id` is the name of the interpretation. This can be used in order to retrieve the list of interpretations.

Alternatively, with the keys modality, you can directly write the facts and the first argument will be interpreted as a model identifier. The above interpretation in the keys modality is

```
pos(2).
triangle(2,o5).
config(2,o5,up).
square(2,o4).
in(2,o4,o5).
circle(2,o3).
triangle(2,o2).
config(2,o2,up).
in(2,o2,o3).
triangle(2,o1).
config(2,o1,up).
```

which is contained in the `bongardkeys.pl` This is also how model `2` above is stored in SWI-Prolog database. The two modalities, models and keys, can be mixed in the same file. Facts for `int/1` are not asserted for interpretations in the key modality but can be added by the user explicitly.

Note that you can add background knowledge that is not probabilistic directly to the file writing the clauses taking into account the model argument. For example (`carc.pl`) contains

```
connected(_M,Ring1,Ring2):-
  Ring1 \= Ring2,
  member(A,Ring1),
  member(A,Ring2), !.

symbond(Mod,A,B,T):- bond(Mod,A,B,T).
symbond(Mod,A,B,T):- bond(Mod,B,A,T).
```

where the first argument of all the atoms is the model.

Example `registration.pl` contains for example

```
party(M,P):-
  participant(M,_, _, P, _).
```

that defines intensionally the target predicate `party/1`. Here `M` is the model and `participant/4` is defined in the interpretations. You can also define intensionally the negative examples with

```
neg(party(M,yes)):- party(M,no).
neg(party(M,no)):- party(M,yes).
```

Then you must indicate how the examples are divided in folds with facts of the form: `fold(<fold_name>,<list of model identifiers>)`, as for example

```
fold(train,[2,3,...]).
fold(test,[490,491,...]).
```

As the input file is a Prolog program, you can define intensionally the folds as in

```
fold(all,F):-
  findall(I,int(I),F).
```

`fold/2` is dynamic so you can also write (`registration.pl`)

```
:- fold(all,F),
   sample(4,F,FTr,FTe),
   assert(fold(rand_train,FTr)),
   assert(fold(rand_test,FTe)).
```

which however must be inserted after the input interpretations otherwise the facts for `int/1` will not be available and the fold `all` would be empty. This command uses `sample(N,List,Sampled,Rest)` exported from `slipcover` that samples `N` elements from `List` and returns the sampled elements in `Sampled` and the rest in `Rest`. If `List` has `N` elements or less, `Sampled` is equal to `List` and `Rest` is empty.

4.2 Commands

4.2.1 Parameter Learning

To execute EMBLEM, prepare an input file in the editor panel as indicated above and call

```
?- induce_par(<list of folds>,P).
```

where `<list of folds>` is a list of the folds for training and `P` will contain the input program with updated parameters.

For example `bongard.pl`, you can perform parameter learning on the `train` fold with

```
?- induce_par([train],P).
```

4.2.2 Structure Learning

To execute SLIPCOVER, prepare an input file in the editor panel as indicated above and call

```
induce(+List_of_folds:list,-P:list) is det
```

where `List_of_folds` is a list of the folds for training and `P` will contain the learned program.

For example `bongard.pl`, you can perform structure learning on the `train` fold with

```
?- induce([train],P).
```

A program can also be tested on a test set with `test/7` or `test_prob/6` as described below.

Between two executions of `induce/2` you should exit SWI-Prolog to have a clean database.

To execute LEMUR, prepare an input file in the editor panel as indicated above and call

```
induce_lm(+List_of_folds:list,-P:list) is det
```

where `List_of_folds` is a list of the folds for training and `P` will contain the learned program.

For example `bongard.pl`, you can perform structure learning on the `train` fold with

```
?- induce_lm([train],P).
```

Between two executions of `induce_lm/2` you should exit SWI-Prolog to have a clean database.

4.2.3 Testing

A program can also be tested on a test set in SLIPCOVER and LEMUR with

```
test(+Program:list,+List_of_folds:list,-LL:float,
  -AUCROC:float,-ROC:list,-AUCPR:float,-PR:list) is det
```

or

```
test(+Program:list,+List_of_folds:list,-NPos:int,-NNeg:int,
  -LL:float,-ExampleList:list) is det
```

where `Program` is a list of terms representing clauses and `List_of_folds` is a list of folds.

`test/7` returns the log likelihood of the test examples in `LL`, the Area Under the ROC curve in `AUCROC`, a dictionary containing the list of points (in the form of Prolog pairs `x-y`) of the ROC curve in `ROC`, the Area Under the PR curve in `AUCPR`, a dictionary containing the list of points of the PR curve in `PR`.

`test_prob/6` returns the log likelihood of the test examples in `LL`, the numbers of positive and negative examples in `NPos` and `NNeg` and the list `ExampleList` containing couples `Prob-Ex` where `Ex` is `a` for `a` a positive example and `\+(a)` for `a` a negative example and `Prob` is the probability of example `a`.

Then you can draw the curves in `cplint` on SWISH using C3.js using

```
compute_areas_diagrams(+ExampleList:list,
  -AUCROC:float,-ROC:dict,-AUCPR:float,-PR:dict) is det
```

(from pack `auc`) that takes as input a list `ExampleList` of pairs probability-literal of the form that is returned by `test_prob/6`.

For example, to test on fold `test` the program learned on fold `train` you can run the query

```
?- induce_par([train],P),
   test(P,[test],LL,AUCROC,ROC,AUCPR,PR).
```

Or you can test the input program on the fold `test` with

```
?- in(P),
   test(P,[test],LL,AUCROC,ROC,AUCPR,PR).
```

In `cplint` on SWISH, by including

```
:- use_rendering(c3).
:- use_rendering(lpad).
```

in the code before `:- sc.` the curves will be shown as graphs using C3.js and the output program will be pretty printed.

You can also draw the curves in `cplint` on SWISH using R by loading library `cplint_r` with

```
:- use_module(library(cplint_r)).
```

and using

```
test_r(+Program:list,+List_of_folds:list,-LL:float,
  -AUCROC:float,-AUCPR:float) is det
```

or predicate

```
compute_areas_diagrams_r(+ExampleList:list,
  -AUCROC:float,-AUCPR:float) is det
```

that takes as input a list `ExampleList` of pairs probability-literal of the form that is returned by `test_prob/6`.

4.3 Parameters

Parameters are set with commands of the form

```
:- set_sc(<parameter>,<value>).
```

The available parameters are:

- `specialization`: (values: `{bottom,mode}`, default value: `bottom`, valid for SLIPCOVER) specialization mode.
- `depth_bound`: (values: `{true,false}`, default value: `true`) if `true`, the depth of the derivation of the goal is limited to the value of the `depth` parameter.
- `depth` (values: integer, default value: 2): depth of derivations if `depth_bound` is set to `true`
- `single_var` (values: `{true,false}`, default value: `false`): if set to `true`, there is a random variable for each clause, instead of a different random variable for each grounding of each clause
- `epsilon_em` (values: real, default value: 0.1): if the difference in the log likelihood in two successive parameter EM iteration is smaller than `epsilon_em`, then EM stops
- `epsilon_em_fraction` (values: real, default value: 0.01): if the difference in the log likelihood in two successive parameter EM iteration is smaller than `epsilon_em_fraction`*(-current log likelihood), then EM stops
- `iter` (values: integer, defualt value: 1): maximum number of iteration of EM parameter learning. If set to -1, no maximum number of iterations is imposed
- `iterREF` (values: integer, defualt value: 1, valid for SLIPCOVER and LEMUR): maximum number of iteration of EM parameter learning for refinements. If set to -1, no maximum number of iterations is imposed.
- `random_restarts_number` (values: integer, default value: 1, valid for EMBLEM, SLIPCOVER and LEMUR): number of random restarts of parameter EM learning
- `random_restarts_REFnumber` (values: integer, default value: 1, valid for SLIPCOVER and LEMUR): number of random restarts of parameter EM learning for refinements
- `seed` (values: seed(integer) or seed(random), default value `seed(3032)`): seed for the Prolog random functions, see SWI-Prolog manual
- `c_seed` (values: unsigned integer, default value 21344)): seed for the C random functions

- `logzero` (values: negative real, default value log(0.000001)): value assigned to $\log 0$
- `max_iter` (values: integer, default value: 10, valid for SLIPCOVER): number of interations of beam search
- `max_var` (values: integer, default value: 4, valid for SLIPCOVER and LEMUR): maximum number of distinct variables in a clause
- `beamsize` (values: integer, default value: 100, valid for SLIPCOVER): size of the beam
- `megaex_bottom` (values: integer, default value: 1, valid for SLIPCOVER): number of mega-examples on which to build the bottom clauses
- `initial_clauses_per_megaex` (values: integer, default value: 1, valid for SLIPCOVER): number of bottom clauses to build for each mega-example (or model or interpretation)
- `d` (values: integer, default value: 1, valid for SLIPCOVER): number of saturation steps when building the bottom clause
- `mcts_beamsize` (values: integer, default value: 3, valid for LEMUR): size of the Monte-Carlo tree search beam
- `mcts_visits` (values: integer, default value: +1e20, valid for LEMUR): maximum number of visits
- `max_iter_structure` (values: integer, default value: 10000, valid for SLIPCOVER): maximum number of theory search iterations
- `background_clauses` (values: integer, default value: 50, valid for SLIPCOVER): maximum numbers of background clauses
- `maxdepth_var` (values: integer, default value: 2, valid for SLIPCOVER and LEMUR): maximum depth of variables in clauses (as defined in [5]).
- `mcts_max_depth` (values: integer, default value: 8, valid for LEMUR): maximum depth of default policy search
- `mcts_c` (values: real, default value: 0.7, valid for LEMUR): value of parameter C in the computation of UCT
- `mcts_iter` (values: integer, default value: 20, valid for LEMUR): number of Monte-Carlo tree search iterations
- `mcts_maxrestarts` (values: integer, default value: 20, valid for LEMUR): maximum number of Monte-Carlo tree search restarts

- `neg_ex` (values: `given`, `cw`, default value: `cw`): if set to `given`, the negative examples in testing are taken from the test folds interpretations, i.e., those examples `ex` stored as `neg(ex)`; if set to `cw`, the negative examples are generated according to the closed world assumption, i.e., all atoms for target predicates that are not positive examples. The set of all atoms is obtained by collecting the set of constants for each type of the arguments of the target predicate.
- `alpha` (values: floating point ≥ 0, default value: 0): parameter of the symmetric Dirichlet distribution used to initialize the parameters. If it takes value 0, a truncated Dirichlet process is used to sample parameters: the probability of being true of each Boolean random variable used to represent multivalued random variables is sampled uniformly and independently in [0,1]. If it takes a value ≥ 0, the parameters are sampled from a symmetric Dirichlet distribution, i.e. a Dirichlet distribution with vector of parameters $(\alpha, \ldots, \alpha)$.
- `verbosity` (values: integer in [1,3], default value: 1): level of verbosity of the algorithms.

5 Download Query Results through an API

The results of queries can also be downloaded programmatically by directly approaching the Pengine API. Example client code is available. For example, the `swish-ask.sh` client can be used with bash to download the results for a query in CSV. The call below downloads a CSV file for the coin example.

```
$ bash swish-ask.sh --server=http://cplint.eu \
  example/inference/coin.pl Prob "prob(heads(coin),Prob)"
```

The script can ask queries against Prolog scripts stored in `http://cplint.eu` by specifying the script on the commandline. User defined files stored in `cplint` on SWISH (locations of type `http://cplint.eu/p/coin_user.pl`) can be directly used, for example:

```
$ bash swish-ask.sh --server=http://cplint.eu \
  coin_user.pl Prob "prob(heads(coin),Prob)"
```

Example programs can be used by specifying the folder portion of the url of the example, as in the first coin example above where the url for the program is `http://cplint.eu/example/inference/coin.pl`.

You can also use an url for the program as in

```
$ bash swish-ask.sh --server=http://cplint.eu \
  https://raw.githubusercontent.com/friguzzi/swish/\
  master/example/inference/coin.pl Prob "prob(heads(coin),Prob)"
```

Results can be downloaded in JSON using the option `--json-s` or `--json-html`. With the first the output is in a simple string format where Prolog terms are sent using quoted write, the latter serialize responses as HTML strings. E.g.

```
$ bash swish-ask.sh --json-s --server=http://cplint.eu \
  coin_user.pl Prob "prob(heads(coin),Prob)"
```

The JSON format can also be modified. See `http://www.swi-prolog.org/pldoc/doc_for?object=pengines%3Aevent_to_json/4`.

Prolog can exploit the Pengine API directly. For example, the above can be called as:

```
?- [library(pengines)].
?- pengine_rpc('http://cplint.eu',
     prob(heads(coin),Prob),
     [ src_url('https://raw.githubusercontent.com/friguzzi/swish/\
master/example/inference/coin.pl'),
     application(swish)
     ]).
Prob = 0.51.
?-
```

6 Manual in PDF

A PDF version of the manual is available at `https://github.com/friguzzi/cplint/blob/master/doc/help-cplint.pdf`.

7 Bibliography

References

[1] Elena Bellodi and Fabrizio Riguzzi. EM over binary decision diagrams for probabilistic logic programs. In *Proceedings of the 26th Italian Conference on Computational Logic (CILC2011), Pescara, Italy, 31 August 31-2 September, 2011*, 2011.

[2] Elena Bellodi and Fabrizio Riguzzi. EM over binary decision diagrams for probabilistic logic programs. Technical Report CS-2011-01, Dipartimento di Ingegneria, Università di Ferrara, Italy, 2011.

[3] Elena Bellodi and Fabrizio Riguzzi. Expectation Maximization over binary decision diagrams for probabilistic logic programs. *Intelligent Data Analysis*, 17(2):343–363, 2013.

[4] Elena Bellodi and Fabrizio Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming*, 15(2):169–212, 2015.

[5] William W. Cohen. Pac-learning non-recursive prolog clauses. *Artif. Intell.*, 79(1):1–38, 1995.

[6] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence*, pages 2462–2467, 2007.

[7] L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the 6th Conference on Algorithmic Learning Theory (ALT 1995)*, volume 997 of *LNAI*, pages 80–94, Fukuoka, Japan, 1995. Springer.

[8] Nicola Di Mauro, Elena Bellodi, and Fabrizio Riguzzi. Bandit-based Monte-Carlo structure learning of probabilistic logic programs. *Mach. Learn.*, 100(1):127–156, July 2015.

[9] Robert M Fung and Kuo-Chu Chang. Weighing and integrating evidence for stochastic simulation in bayesian networks. In *Fifth Annual Conference on Uncertainty in Artificial Intelligence*, pages 209–220. North-Holland Publishing Co., 1990.

[10] Muhammad Asiful Islam, CR Ramakrishnan, and IV Ramakrishnan. Inference in probabilistic logic programs with continuous random variables. *Theory and Practice of Logic Programming*, 12:505–523, 7 2012.

[11] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine learning. MIT Press, Cambridge, MA, 2009.

[12] Arun Nampally and CR Ramakrishnan. Adaptive mcmc-based inference in probabilistic logic programs. *arXiv preprint arXiv:1403.6036*, 2014.

[13] Davide Nitti, Tinne De Laet, and Luc De Raedt. Probabilistic logic programming for hybrid relational domains. *Mach. Learn.*, 103(3):407–449, 2016.

[14] J. Pearl. *Causality*. Cambridge University Press, 2000.

[15] David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2):7–56, 1997.

[16] Fabrizio Riguzzi. MCINTYRE: A Monte Carlo system for probabilistic logic programming. *Fundamenta Informaticae*, 124(4):521–541, 2013.

[17] Fabrizio Riguzzi. The distribution semantics is well-defined for all normal programs. In Fabrizio Riguzzi and Joost Vennekens, editors, *Proceedings of the 2nd International Workshop on Probabilistic Logic Programming (PLP)*, volume 1413 of *CEUR Workshop Proceedings*, pages 69–84, Aachen, Germany, 2015. Sun SITE Central Europe.

[18] Fabrizio Riguzzi and Terrance Swift. Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In *Technical Communications of the International Conference on Logic Programming*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 162–171. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[19] Taisuke Sato and Yoshitaka Kameya. Prism: A language for symbolic-statistical modeling. In *International Joint Conference on Artificial Intelligence*, pages 1330–1339, 1997.

[20] Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *J. Artif. Intell. Res.*, 15:391–454, 2001.

[21] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, volume 3131 of *LNCS*, pages 195–209. Springer, 2004.

[22] John Von Neumann. Various techniques used in connection with random digits. *Nat. Bureau Stand. Appl. Math. Ser.*, 12:36–38, 1951.