TABLE OF CONTENTS

Page

CHAPTER Page

Chapter 1

SYSTEM INSTALLATION

1.1 Installation Instructions

The following installation instructions are for a Debian based Linux distribution. First install the following software that are required to build this software

```
sudo apt-get install git
sudo apt-get install uuid-dev
sudo apt-get install autotools-dev
```

After verifying that git is installed, also verify that you have GNU auto tools installed for your system. Download the lpmln repository

```
git clone https://github.com/samidhtalsania/lpmln
cd lpmln
git checkout LPNMR-draft
```

This will create a repository in the existing directory. Use the following sequence of commands to compile and install the software.

```
aclocal
autoconf
automake --add-missing
./configure
make
sudo make install
```

This will install two binaries in the system. lpmln2mln and lpmln2asp.

1.2 Prerequisites

Since the system uses existing tools to compute lpmln programs like clingo, Alchemy, Tuffy and Rockit, these systems need to be installed by the users and also their respective prerequisites.

1.2.1 Clingo

Clingo is available from https://sourceforge.net/projects/potassco/files/clingo/. The system uses clingo v4.5.4. Note that clingo needs to be built with its python module. The python module is used for marginal computation.

1.2.2 Alchemy

Alchemy is available from https://code.google.com/archive/p/alchemy-2/. The system uses Alchemy version 2.0.

1.2.3 Tuffy

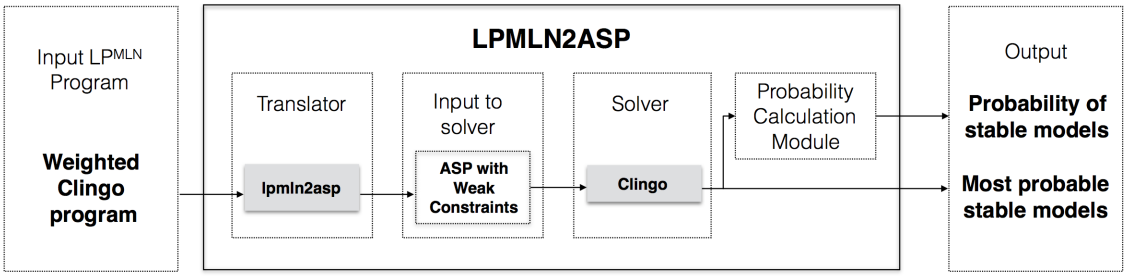
Tuffy is available from http://i.stanford.edu/hazy/tuffy/download/. The system uses Tuffy version 0.4. Tuffy requires postgres to be installed and configured correctly for its usage. More details can be found on the tuffy website.

1.2.4 Rockit

Rockit is available from https://github.com/jnoessner/rockIt. The system uses Rockit version 0.5.227. Rockit requires MySql and Gurobi to be installed and configured correctly for its use. Gurobi is a commercial software for which a free academic license can be acquired.

Chapter 2

LPMLN2ASP



2.1 Introduction

lpmln2asp is the package to compute MAP inference, marginal probability and conditional probability using ASP solvers.

The module is useful for converting legacy clingo code to weighted program with LPMLN semantics. Traditionally a clingo program comprises of a set of rules (referred to as hard rules from here onwards). A hard rule can be converted to a soft rule by appending a decimal value before the rule.

2.2 lpmln2asp Syntax

The system lpmln2asp is an elaborate macro processor which takes in soft clingo rules. Any valid clingo file that can be compiled by clingo(safe programs) can be used as input for lpmln2asp. Do note that in certain cases a safe program can also cause lpmln2asp to fail. A detailed example is described later.

The lpmln semantics allow rules of the form

```
w: head :- body.
```

where W is the weight given to the rules. The weight can be any decimal value or infinity(denoted by α). To denote infinite weight (hard rule) W is not written.

Here are a few examples,

```
10 a(X) :- b(Y). % soft rule

a(X) :- b(X). % hard rule - infinite weight

10.01 a(X) :- b(X). % Decimals are allowed

a(jo). % hard rule with body top

-10 a(jo). % soft rule with body top.
%Negative weights are allowed

:- b(X). % hard constraint

10 :- b(X). % soft constraint

log(10) a(jo). % Inbuilt functions like log(x) can be used.
```

2.2.1 Rules

The system accepts any rules that is also accepted by clingo. Every valid rule for clingo is also a valid rule for lpmln2asp system. This implementation was designed to easily convert rules to weak constraints and allow for MAP inference and performing probabilistic query on atoms and also calculating the probability of stable models.

For the accepted syntax of clingo, please refer to the clingo manual.

2.3 lpmln2asp usage

The basic usage of lpmln2asp is

```
lpmln2asp -i <input_file> [-e <evid_file>] [-r <output_file>]
[-q <query>]
```

lpmln2asp is an encompassing system on top of lpmln2cl which provides an interface similar to MLN tools like alchemy, tuffy, rockit.

lpmln2asp can compute conditional probability, marginal probability and MAP inference depending on the arguments provided.

Command line options for lpmln2asp,

```
  -h, --help      show this help message and exit
  -i I            input file. [REQUIRED]
  -e E            evidence file
  -r R            output file. If not provided output would be to STDOUT
  -q Q            query predicate constants. Multiple comma separated
     predicate constants can be provided.
  -clingo CLINGO  clingo options passed as it is to the solver. Pass all
      clingo options in "single quotes"
  -hr             [FALSE] Translate hard rules
  -prob           [MAP] Use this option to activate probabilistic
     inference.
  -quiet          [DEFAULT] Quiet mode for computation.
                  Suppresses output of probability of all models.
  -mf MF          [1000] Multiplying factor for weak constraints value
  -d              [FALSE] Debug. Print all debug information
```

The implementation is safe to use with files containing python code and does not affect the syntax of the python code. Do note that this implementation does not check for the correctness of syntax of input and in case of syntactically incorrect input, the implementation would output the translated rules which would be syntactically wrong as well.

query argument *-q* is best used without *-hr*. The option *-hr* translates all the hard rules as well. While this is useful for debugging, it also results in an increase of unsat atoms (one unsat atom corresponding to every rule in the encoding) which makes the post processing with python slower. Since hard rules always need to be satisfied this option is not much useful except for debugging.

-mf X option is the multiplying factor and the default value is 1. Consider a rule

$$\text{w : a :- b.}$$

where $0 < \text{w} < 1$. Since clingo does not allow decimals as a part of its weight scheme, the above rule would generate the weak constraint like,

$$:\sim \text{unsat}(0, \text{"w"}).[0@0]$$

All rules with weight $0 < \text{w} < 1$ would generate an unsat atom with weight 0 which may not be desirable if performing MAP inference. Setting the value of *-mf* to X would result in,

$$:\sim \text{unsat}(0, \text{"w"}).[\text{w} * \text{X}@0]$$

The multiplying factor multiplies the decimal weights by the factor and then uses the floor of the value as integral weights for weak constraints. This allows for more fine grained control of the weight scheme for MAP inference. For probabilistic inference the *-mf* option is ignored.

Usage Examples

MAP Inference

```
lpmln2asp -i prog_file
```

Marginal probability

```
lpmln2asp -i prog_file -q predicate_constant -prob
```

Conditional Probability

```
lpmln2asp -i prog_file -q predicate_constant -e evidence_file -prob
```

Probability of stable models (Verbose mode)

```
lpmln2asp -i prog_file -q predicate_constant -e evidence_file -prob -v
```

Internal functions in lpmln2asp

lpmln2asp allows for using functions like log(x) or exp(x) in the input language. The syntax is simple in the sense, instead of using weights you can use the function. The syntax to use such functions is,

```
@function_name(expression)
```

Examples,

```
@exp(2) bird(X) :- migratorybird(X).
@exp(2/10.0) :- residentbird(X) , migratorybird(X).


@log(10/1.0) residentbird(bob).
@log((0.75*10)+1-(2*(3/6))) migratorybird(bob).
```

2.4 lpmln2asp Examples

These are some domains on which lpmln2asp was tried. The input for the domains, the command line and thee expected output for each are mentioned in this section.

2.4.1 Example 1

Consider the simple LP^{MLN} program with atoms P,Q and R

Input

```
1 p :- q.
1 q :- p.
2 p :- not r.
3 r :- not p.
```

Command Line

```
lpmln2asp -i ex1.lp -q p -prob
```

Output

```
p 0.576042038598



OPTIMUM FOUND

Models       : 4
  Optimum    : yes
Calls        : 1
Time         : 0.698s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.560s
```

2.4.2 Taxi Problem

Consider a program where you know that you are going to be Late if you don't get a Taxi.

Input

```
1 late :- notaxi.
2 notaxi.
:- late.
```

Comamnd Line

```
lpmln2asp -i taxi.lp -q notaxi -prob
```

Output

```
Solving...


notaxi 0.73105857863


OPTIMUM FOUND


Models       : 2
  Optimum    : yes
Calls        : 1
Time         : 3.269s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.620s
```

2.4.3 Example 2 from **?**

Consider an instance where we are certain that we booked a concert and that we have a long drive ahead of us unless the concert is cancelled. However, there is a 20% chance that the concert maybe cancelled. This can be encoded as,

Input

```
concertbooked.
longdrive :- concertbooked , not cancelled.
```

```
-1609 cancelled.
-2231 :- cancelled.
```

Command Line

```
lpmln2asp -i ex2.lp -q concertbooked -prob
```

Output

```
Solving...
concertbooked 1.0


OPTIMUM FOUND


Models       : 5
  Optimum    : yes
Calls        : 1
Time         : 2.944s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.600s
```

2.4.4 Bayes Net

Bayes net can be easily encoded in the language of LP^{MLN}. We will consider the classic example of Burglary, Earthquake, Alarm, John Calls and Mary Calls. The problem statement is,

include graphic for bayes net example

I'm at work, neighbor John calls to say my alarm is ringing, but neighbor Mary doesn't call.Sometimes it's set on by minor earthquakes.Is there a burglar? We want to find the probability of p(Burglar|JohnCalls, –MaryCalls)

Input

```
ab :- pf(b).

ae :- pf(e).

aa :- ab ,ae , pf(a00).
aa :- ab ,not ae , pf(a01).
aa :- not ab ,ae , pf(a10).
aa :- not ab ,not ae , pf(a11).

aj :- aa ,pf(j0).
aj :- not aa ,pf(j1).

am :- aa ,pf(m0).
am :- aa ,pf(m1).



-6.9077 pf(b).

-6.2146 pf(e).

2.9444 pf(a00).

2.7515 pf(a01).

-0.8953 pf(a10).

-6.9067 pf(a11).

0.8472 pf(m0).

-4.5951 pf(m1).
```

```
2.1972 pf(j0).

-2.9444 pf(j1).
```

Evidence

```
:- not aj.
:- am.
```

Command line

```
lpmln2asp -i bn_draft.lp -e bn_draft_evid.db -q ab
```

Output

```
ab 0.00502573986437

OPTIMUM FOUND

Models       : 320
  Optimum    : yes
Calls        : 1
Time         : 0.222s (Solving: 0.01s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.220s
```

2.4.5 Pearl's Causal Model

The Firing Squad example is a well-known example that illustrates the idea of probabilistic causal model[Pearce et al.2000], proposed by Judea Pearl. The probabilistic causal model for this domain can be encoded in this domain.

include graphic for pearls causal model

In the figure we see the firing squad scenario expressed as a causal model. U and W are

exogenous variables as in their value does not depend on any external factors. C,A,B and D are endogenous variables as their values are determined by other variables.

Through such a model we can answer queries such as:

- Given that the court has not ordered the execution, what is the probability that the prisoner is dead?
- Given that the prisoner is dead, what is the probability that the court has ordered the execution?
- Given that Rifleman A shot, what is the probability that Rifleman B shot as well?
- Given that the captain has not given the signal and Rifleman A decided to shoot, what is the probability that the prisoner is dead? What is the probability that Rifleman B shot?
- Given that the prisoner is dead, what is the probability that the prisoner were not dead if Rifleman A had not shoot?

Through the encoding shown below we will try to answer the the first query.

Input

```
@log(0.2)  u.
@log(0.7)  w.


c :- u.
a :- c.
a :- w.
b :- c.
d :- a.
d :- b.
```

```
cs :- u, not do(c1), not do(c0).
as :- cs, not do(a1), not do(a0).
as :- w, not do(a1), not do(a0).
bs :- cs, not do(b1), not do(b0).
ds :- as, not do(d1), not do(d0).
ds :- bs, not do(d1), not do(d0).

cs :- do(c1).
as :- do(a1).
bs :- do(b1).
ds :- do(d1).
```

Evidence

```
do(a0).
:- not d.
```

Command Line

```
lpmln2asp -i pcm1.lp -e evid.db -prob -q ds
```

Output

```
ds 0.326921718908

OPTIMUM FOUND

Models       : 3
  Optimum    : yes
Calls        : 1
Time         : 0.217s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.200s
```

2.4.6 Inconsistent Databases

Consider an instance where you have conflicting information from different data sources. One data source may mention that an entity Jo is a resident bird however some other data source may say that Jo is a migratory bird. Although we cannot say with certainty that Jo is a migratory bird or a resident bird but we can say with certainty that Jo is a Bird. Suppose we want to find out what is the probability of Jo being a Bird given that one KB says it is ResidentBird and other says it is a MigratoryBird. Such a domain can be easily encoded in this language.

Input

```
bird(X) :- residentbird(X).
bird(X) :- migratorybird(X).
:- residentbird(X) , migratorybird(X).
2 residentbird(jo).
1 migratorybird(jo).
```

Command Line

```
lpmln2asp -i bird.lpmln -q bird,residentbird,migratorybird -prob
```

Output

```
migratorybird(jo) 0.244728471055
residentbird(jo) 0.665240955775
bird(jo) 0.90996942683



OPTIMUM FOUND


Models       : 7
```

```
  Optimum    : yes
Calls        : 1
Time         : 0.956s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.610s
```

2.4.7 Probabilistic States

Consider an example where there are two states S_1 and S_2 and action A_1. The initial state is S_1. A_1 can be true or false. A_1 switches state from S_1 to S_2 if true or stays in the same state. A_1 does not affect S_2 and the state does not change thereafter. A_1 has certain probability of being true. The question is to find the probability of a model in which state is S_2.

Input

```
step(0).
step(1).
boolean(t).
boolean(f).



next(0,1).

-0.6931 pf(0).
-0.6931 pf(1).

p(t,T1) :- a(t,T) , next(T,T1) , pf(t,T) , p(f,T).

p(t,T1) :- a(f,T) , next(T,T1) , p(t,T).

p(f,T1) :- a(f,T) , next(T,T1) , p(f,T).
```

```
p(B,0) :- p(B,0).

a(B,T) :- a(B,T).


p(t,T) :- not p(f,T), step(T).
p(f,T) :- not p(t,T), step(T).

:- p(t,T) , p(f,T).


a(t,T) :- not a(f,T), step(T).
a(f,T) :- not a(t,T), step(T).
:- a(t,T) , a(f,T).

p(B,T) :- p(B,T1) , p(B,T) , next(T1,T).

p(f,0).

a(t,0).
```

Command Line

```
lpmln2asp -i bc.lp -prob -q p
```

Clingo Output

```
p(f, 0) 1.0
p(t, 1) 0.5
p(f, 1) 0.5
```

```
OPTIMUM FOUND


Models       : 16
  Optimum    : yes
Calls        : 1
Time         : 0.210s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.200s
```

2.4.8 Monty Hall Problem

This is an example of a complicated bigger domain that this framework can solve. A player is given an opportunity to select one of the three closed doors, behind one of which there is a prize. Behind the other two doors are empty rooms. Once the player has chosen Monty opens one of the remaining closed doors which does not contain the prize. The player is then asked to switch his selection to other unopened door. We query the probability of the player winning if he switches.

Input

```
door(d1;d2;d3).
constant(prize;selected;open).
number(2;3).
boolean(t;f).



canopen(D,f) :- selected(D),door(D).
canopen(D,f) :- prize(D),door(D).
canopen(D,t) :- not canopen(D,f),door(D),door(D).

%//constrints
 :- canopen(D,t) , canopen(D,f).
```

```
 :- prize(D1) , prize(D2) , D1!=D2.
 :- selected(D1) , selected(D2) , D1!=D2.
 :- open(D1) , open(D2) , D1!=D2.
%// **** RandomSelection ****
prize(d1) :- not intervene(prize) , not prize(d2) , not prize(d3).
prize(d2) :- not intervene(prize) , not prize(d1) , not prize(d3).
prize(d3) :- not intervene(prize) , not prize(d2) , not prize(d1).
selected(d1) :- not selected(d2) , not selected(d3) , not intervene(
   selected).
selected(d2) :- not selected(d3) , not selected(d1) , not intervene(
   selected).
selected(d3) :- not selected(d2) , not selected(d1) , not intervene(
   selected).
open(d3) :- not open(d1) , not open(d2) , not intervene(open).
open(d2) :- not open(d1) , not open(d3) , not intervene(open).
open(d1) :- not open(d3) , not open(d2) , not intervene(open).
 :- open(D) , not canopen(D,t) , not intervene(open).
posswithdefprob(selected,D) :- not posswithassprob(selected,D) , not
   intervene(selected),door(D),door(D).
posswithdefprob(prize,D) :- not posswithassprob(prize,D) , not intervene
   (prize),door(D),door(D).
posswithdefprob(open,D) :- not posswithassprob(open,D) , canopen(D,t) ,
   not intervene(open),door(D) , door(D),door(D).

numdefprob(prize,X) :- X= #count{D:posswithdefprob(prize,D)} , prize(Y)
   , posswithdefprob(prize,Y),number(X).

numdefprob(selected,X) :- X= #count{D:posswithdefprob(selected,D)} ,
   selected(Y) , posswithdefprob(selected,Y),number(X).
```

```
numdefprob(open,X) :- X= #count{D:posswithdefprob(open,D)} , open(Y) ,
   posswithdefprob(open,Y),number(X).



%// **** Do ****
do(selected,d1).
selected(d1) :- do(selected,d1).
intervene(selected) :- do(selected,d1).

%// **** Obs ****
obs(open,d2).
 :- obs(open,d2) , not open(d2).

% The only soft rules in the program
-0.6931 :- not numdefprob(C,2),constant(C).
-0.4054 :- not numdefprob(C,3),constant(C).
```

Command line

```
lpmln2asp -i input.lp -prob -q prize
```

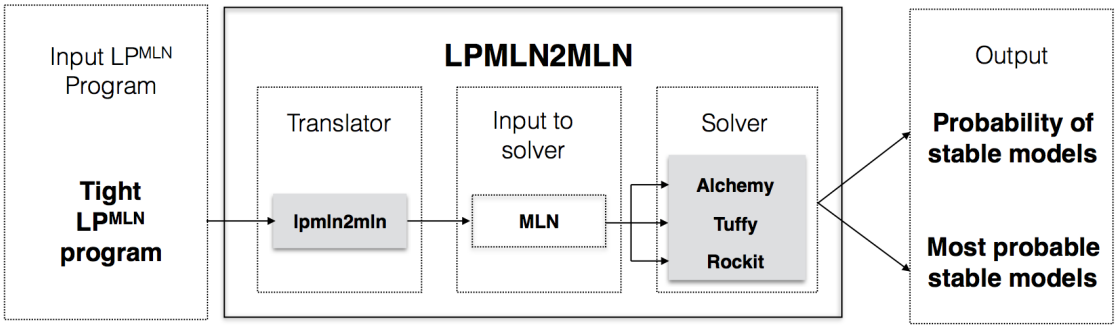
Clingo Output

```
prize(d1) 0.333343817985
prize(d3) 0.666656182015



OPTIMUM FOUND

Models       : 2
  Optimum    : yes
Calls        : 1
```

```
Time         : 0.210s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.200s
```

Chapter 3

LPMLN2MLN



3.1 Introduction

LP^{MLN} is an extension of answer set programs by adopting the concept of weights, whose weight scheme is adopted from Markov Logic Network. This document describes an implementation of LP^{MLN}, which compiles the language of LP^{MLN} into the input language of ALCHEMY, TUFFY or ROCKIT.

This chapter explains the syntax for the input language of lpmln2mln, the completion algorithm implemented, tseytin's transformation usage in completion and the usage of the system. The chapter also has a section explaining the differences among the underlying solvers and their respective weaknesses and capabilities. Currently this implementation of lpmln language is restricted to tight programs only.

3.2 Input Syntax

Identifiers

A legal identifier is a sequence of letters and digits that begin with a letter. Identifier for objects should begin with an uppercase letter.

Variables

All variables begin with lowercase letters. All constants begin with uppercase letter (Do not confuse between constant values mentioned here and constants section explained in the next section).

A variable is

- An identifier.
- Has not been declared as either a sort, object constant or predicate constant. The type of variable is determined by the position p in which it appears in the predicate Pr. It's type is the same as the sort present in location p in the predicate declaration of Pr.

Declarations

Declarations are similar to Alchemy syntax.

A sort is a collection of objects. Each sort s_i is a collection of objects $o_1, ..., o_n$. Each sort is declared as,

$$s = \{o_1, ..., o_n\}$$

A predicate is declared using a predicate constant P with terms $t_1, ..., t_n$. Each predicate is declared as,

$$P(t_1, ..., t_n)$$

where each t_i is a sort. A predicate with 0 terms can be declared. All the predicates and sorts that are used in the input program have to be declared before its usage in the program. A sort with no objects cannot be declared.

Literal

A literal is a predicate constant which may or may not be followed by terms. Literals are used in rules. Each literal may be prefixed by double or single negation. Literal used in head of a rule (defined later) cannot have any negation. Literal without negation are called positive literals. A literal is used in the input as,

$$[\text{not} \mid \text{not not}] \quad P(t_1, ..., t_n)$$

where t_i can be a variable or an object constant and $i >= 0$.

Examples of Literals:

```
male(a,b,c)
not male(a,b,c)
not not male(a,b,c)
```

or of type,

```
male
```

where a,b,c are variables but can also be a *object constant*.

Negation

Negation can be single negation denoted by not or double negation denoted by not not. Negation can be used throughout the program in front of any literals in the body part of any rule.

Comparison operators

Comparison operators { = , != } are allowed only between variables of the same type or between a variable and an object constant whose sort is the same as that represented by the variable. Comparison operators can only be used in body of the rules.

Weights

The weight can be positive or negative decimal.

Head of a formula

Head is a disjunction of literals or empty($\perp$). Head only allows positive literals.

$$A_1 \quad v \quad A_2 \quad v \quad ... \quad v \quad A_N \tag{3.1}$$

where A_i is any positive literal(a literal without negation) and i >= 0. Choice rules can also be expressed in head as,

$$\{A_i\} \tag{3.2}$$

where A_i is any positive literal. Comparison operators cannot be used in head.

Body of a formula

Body is a conjunction of literals or empty ($\top$).

$$A_1 \wedge ... \wedge A_N \tag{3.3}$$

where A_i is any literal or a comparison literal and i >= 0. In addition comparison operators can be used in the body of a rule as,

$$j \quad = \quad k$$

$$j \quad != \quad k \tag{3.4}$$

where j is a variable and k can be a variables and or an object constant. The variables getting compared need to be of the same type. The object constant getting compared to a variable has to be one of the object constants of the sort represented by the variable.

Rules

Rules are of two types, soft rules and hard rules. A soft rule is written as,

$$w \quad \text{head} <= \text{body} \tag{3.5}$$

where head is the head of the rule, body is the body of the rule and w is the weight of the rule. A hard rule is written as,

$$\text{head} <= \text{body}. \tag{3.6}$$

Notice the period at end.

Rule examples

Each of the rules below is a *Hard rule* (notice the period at the end). Each of these hard rules can be written as a soft rule by removing the period and adding a weight at the starting of the rule.

- Simple formula:

```
male(T) <= not  intervene(Male).
```

- Disjunction in Head:

```
male(T) v male(F) <= not  intervene(Male).
```

- Conjunction in Body (note that b are variables):

```
human(Male, C2) <= male(b) ^ human(Male, b).
```

- Constraint:

```
<= male(a) ^ male(b).
```

- Double negation:

```
<= not  male(T).
```

- Comparison Operators:

```
female(a) <= male(a) ^ male(b) ^ a!=b.
```

```
female(a) <= male(a) ^ male(b) ^ a=b.
```

- Choice Rules:

```
{load(a, b)} <= step(a, b).
```

  which is equivalent to

```
load(a, b) v not load(a, b)  <= step(a, b).
```

  which is a case of disjunction in head.

Comments

All comments start with // and start on a new line.

3.3 Usage

Running lpmln2mln is very much like running traditional MLN solvers.

Usage:

```
lpmln2mln [-alchemy | -tuffy | -rockit] -i <input_file> -e <
   evidence_file> -r <output_file> -mln <solver specific options in
   quotes> -q <query>


-alchemy        Compile to Alchemy version 2.0 syntax [DEFAULT]

-tuffy          Compile to Tuffy version 0.3 syntax.

-rockit         Compile to rockit version 0.5.227 syntax.

-i              Input file. [REQUIRED]
-r              Output file. [REQUIRED]
-e              Evidence file. [REQUIRED for Rockit]
-mln            Arguments passed to the respective solvers selected by
   first argument.
-q              Query.[REQUIRED]

Alchemy Options:
Call infer -h from command prompt to view alchemy options.

Tuffy Options:
Call java -jar tuffy.jar

Rockit Options:
Call java -jar rockit.jar
```

Internally lpmln2mln calls the compiler LPMLNCOMPILER which is another binary distributed with the package.

There are a lot of options for lpmlncompiler which can be used for fine grain control over the process of compilation. One such option is the optimization flag. There are three

levels of optimization available o0, o1, o2. The optimization that this does is with respect to the completion of literals when the input program is compiled for either of the solvers. While completion, the conjunction of clauses are distributed over disjunction. The optimization process replaces the conjunctive clauses with a single literal which effectively translates conjunction over disjunction to a formula in disjunctive normal form. The first level of optimization o1 translates conjunctive clauses with "EXIST" keyword and the second level of optimization o2 translates all conjunctive clauses. Internally o1 is the default optimization. o2 is not required for most use cases and o0 should be used with caution because in some cases alchemy would not be able to compute lpmln inputs compiled with o0 due to memory constraints.

3.4 Target Languages

The system translates the input program into weighted First order logic programs, which can be run on Alchemy, Tuffy and Rockit. The following section describes the respective systems and there limitations.

3.4.1 Alchemy

Alchemy syntax subsumes the First-Order Logic. Any FOL rule can be written in Alchemy. Unlike, Rockit and Tuffy, Alchemy does not use a database system anytime during computation and performs all computation in memory. This limits the usage of alchemy to smaller domains which can be computed in memory. Larger domains most likely will lead to a segment fault with this system.

3.4.2 Tuffy

Tuffy syntax is not as expressive as alchemy syntax and a bit different then alchemy therefore the translated code from lpmln2mln cannot be directly used with tuffy. However,

tuffy uses databases for grounding and inference and therefore it is much more scalable then alchemy and an LP^{MLN} to tuffy translation is desirable.

Tuffy Limitations

Here is a list of differences between Alchemy and Tuffy,

- Tuffy does not support bi-implication<=> which is supported by alchemy. lpmln2mln handles it by translating a rule with bi-implication

  ```
  a <=> b.
  ```

  to

  ```
  a => b.
  b => a.
  ```

- Tuffy uses different syntax when comparing variables or variables with constants.

  ```
  [x_i = y_i AND x_i = Y_i]
  [x_i = y_i OR x_i = Y_i]
  ```

  where x_i, y_i is a variable and Y_i is an object constant of the same type as x_i.

- Declaration of predicates in Tuffy has to be at the very beginning of the file. In Alchemy the predicate can be declared anywhere before it is used.

- Commas for conjunction.

- Exist clauses are used differently in Tuffy. A rule like,

  ```
  p(x) => Exist y q(x,y) v Exist z r(x,z)
  ```

  is written in Tuffy as

  ```
  Exist y,z   p(x) => q(x,y) v r(x,z)
  ```

- Tuffy does not accept predicates without any arguments.

```
P

Q

P => Q
```

  The above rules will not be accepted by Tuffy. Although the input language of lpmln2mln allows predicates without any terms, the user needs to make sure that the output program generated is runnable in tuffy by adding a dummy argument to all predicates with arity 0.

3.4.3 Rockit

Rockit has a more limited syntax then tuffy. This section lists down some of these limitations of Rockit syntax. The current implementation is limited in supporting rockit syntax.

One hard limitation while using Rockit as solver is to not use comparison operators in the input language and using rules with empty body in evidence file only.

Rockit limitations

- Tuffy syntax can be used for rocket after eliminating equality, $=>$ and $\wedge$.
- Exist syntax is different from tuffy, alchemy.

```
Exists y friends(x,y)
```

  Becomes

```
|y| friends(x,y)
```

- Cutting plane inference is disabled when existential formulas are used. The author warns that runtimes might be long and models might require large amount of RAM in presence of existential rules.

- All constants should be within “ ” . Rest all are considered variables in a formula.
- Gurobi is the internal solver. It is a commercially available ILP solver with free academic licenses and needs to be installed separately.
- Evidence cannot be empty.
- All predicate not preceded by * are hidden predicates and form the query atoms.

3.5 Examples

This section lists all examples that were tested with the lpmln2mln system. For each of these examples Alchemy was used as the solver. However either solver can be used for these examples. This section lists the input, command line, intermediate input for Alchemy and the expected output for each of these examples.

3.5.1 Example 1

Consider the simple LP^{MLN} program with atoms P,Q and R

LP^{MLN} Input

```
P
Q
R

1 P <= Q
1 Q <= P
2 P <= not R
3 R <= not P
```

Usage - MLN

```
lpmln2mln -i input.lpmln -q P,Q,R -r out.txt
```

The above usage instructs the compiler to output Alchemy compatible code and query for atom P,Q AND R. Note that if -q argument was not given then the program would terminate with an error. Alchemy needs an atom to query for otherwise it terminates and as a result lpmln2al terminates.

Intermediate Program - MLN

```
P
Q
R

//r1
1 Q => P
//r2
1 P => Q
//r3
2 !R => P
//r4
3 !P => R
//parsing complete!
P => (Q) v (!R).
Q => (P).
R => (!P).
```

Output - MLN

Models with interpretation P,Q and R have the highest probability. The marginal probabilities(which is what the software outputs) can thus be calculated from this table. P's probability would be the summation of the values from [2,3] and [5,3] which comes down to 0.57 and

```
P 0.568993
Q 0.410009
R 0.424008
```

3.5.2 Taxi Problem

Consider a program where you know that you are going to be Late if you don't get a Taxi.

LP^{MLN} Input

```
NoTaxi
Late

//You are going to be late if you don't get a taxi
//Our belief in this rule is denoted by the weight at the start
1 Late <= NoTaxi

//There is a chance for a taxi to be not available
2 NoTaxi

//But you certainly cannot be late no matter what situation
//Hence this constraint is encoded as a hard rule
<= Late.
```

Usage - MLN

```
lpmln2mln -i input.lpmln -q Late,NoTaxi -r out.txt
```

Intermediate Program - MLN

```

NoTaxi
```

```
Late

//You are going to be late if you don't get a taxi
//Our belief in this rule is denoted by the weight at the start
1 NoTaxi => Late

//There is a chance for a taxi to be not available
2 NoTaxi

//But you certainly cannot be late no matter what situation
//Hence this constraint is encoded as a hard rule
(!Late).
//parsing complete!
Late => (NoTaxi).
```

Output - MLN

Since we have put a hard constraint that we cannot be late the probability of being Late is going to be 0. The probability of there being a NoTaxi comes out to be 71%. NoTaxi = T, Late = F satisfies the second soft rule and the hard constraint. Hence the probability for this model comes out to be $e^2/(e^2 + e^1)$ which comes out to be 73%

```
Late 4.9995e-05
NoTaxi 0.711979
```

3.5.3 Example 2 from **?**

Consider an instance where we are certain that we booked a concert and that we have a long drive ahead of us unless the concert is cancelled. However, there is a 20% chance that the concert maybe cancelled. This can be encoded as,

LP^{MLN} Input

```
ConcertBooked
LongDrive
Cancelled

//We know that the concert is booked
ConcertBooked.

//We know that if the concert is booked and not cancelled,
//there is going to be a long drive
LongDrive <= ConcertBooked ^ not Cancelled.

//Our belief in concert getting cancelled - 50%
-1.6094 Cancelled

//Our belief in concert not getting cancelled - 80%
-0.2231 <= Cancelled
```

Usage - MLN

```
lpmln2mln -i input.lpmln -q LongDrive,ConcertBooked,Cancelled -r out.txt
```

Intermediate Program - MLN

```
ConcertBooked
LongDrive
Cancelled

//We know that the concert is booked

//We know that if the concert is booked and not cancelled,
//there is going to be a long drive
```

```
ConcertBooked ^ !Cancelled => LongDrive.

//Our belief in concert getting cancelled - 50%
-1.6094 Cancelled

//Our belief in concert not getting cancelled - 80%
-0.2231 (!Cancelled)

//parsing complete!
LongDrive => (ConcertBooked ^ !Cancelled).
```

Output - MLN

This output clearly shows that there is an 80% chance of a LongDrive if the chances for show getting cancelled are close to 20%. These are not exact values since alchemy does not do exact inference.

```
ConcertBooked 0.99995
Cancelled 0.188031
LongDrive 0.811969
```

3.5.4 Bayes Net

Bayes net can be easily encoded in the language of LP^{MLN}. We will consider the classic example of Burglary, Earthquake, Alarm, John Calls and Mary Calls. The problem statement is,

include graphic for bayes net example

I'm at work,neighbor John calls to say my alarm is ringing,but neighbor Mary doesn't call.Sometimes it's set on by minor earthquakes.Is there a burglar?

LP^{MLN} Input

```
entity(B,E,A00,A01,A10,A11,J0,J1,M0,M1)



AB
AE
AA
AJ
AM
PF(entity)

//Here we are encoding the Conditional Probability Table
//Marginal B
//ln(0.001)
-6.9077 PF(B)

//Marginal E
//ln(0.002)
-6.2146 PF(E)

//Conditional A=1 given B=1, E=1
// ln(0.95/0.05)
2.9444 PF(A00)

//Conditional A=1hh given B=1, E=0
// ln(0.94/0.06)
2.7515 PF(A01)

//Conditional A=1 given B=0, E=1
// ln(0.29/0.71)
-0.8953 PF(A10)
```

```
//Conditional A=1 given B=0, E=0
// ln(0.001/0.999)
-6.9067 PF(A11)

//Conditional M=1 given A=1
// ln(0.7/0.3)
0.8472 PF(M0)

//Conditional M=1 given A=0
// ln(0.01/0.99)
-4.5951 PF(M1)

//Conditional J=1 given A=1
//ln(0.90/0.10)
2.1972 PF(J0)

//Conditional J=1 given A=0
//ln(0.05/0.95)
-2.9444 PF(J1)

//Here we encode the dependence relation from Bayes Net

//The Auxiliary literal PF(B) implies that Action B happened.
AB <= PF(B).

//The Auxiliary literal PF(E) implies that Action E happened.
AE <= PF(E).

//The encoding of the Alarm Node is done in a similar fashion
//Wherever in the CPT, B=1 we use AB in Body and not AB otherwise.
//For a particular assignment of values, ex. B=1, A=1 we also add to the
```

```
//body the corresponding auxiliary atom which in this case is PF(A00).
AA <= AB ^ AE ^ PF(A00).
AA <= AB ^ not AE ^ PF(A01).
AA <= not AB ^ AE ^ PF(A10).
AA <= not AB ^ not AE ^ PF(A11).

//Encoding of John Calls Node
AJ <= AA ^ PF(J0).
AJ <= not AA ^ PF(J1).

//Encoding of Mary Calls Node
AM <= AA ^ PF(M0).
AM <= AA ^ PF(M1).

//Instantiation of Bayes Net
//In the Question it says that John Calls,
//So we make AJ Literals's value as True.
AJ.
```

Usage - MLN

The question asks to query if there was a Burglar. So we query for literal AB. This example also demonstrates how to pass options to the respective solvers.

```
lpmln2mln -i input.lpmln -q AB -r out.txt -alchemy " -maxSteps 10000"
```

Intermediate Program - MLN

```
entity={B,E,A00,A01,A10,A11,J0,J1,M0,M1}

AB
AE
AA
```

```
AJ
AM
PF(entity)

-6.9077 PF(B)
-6.2146 PF(E)

// ln(0.95/0.05)
2.9444 PF(A00)
// ln(0.94/0.06)
2.7515 PF(A01)
// ln(0.29/0.71)
-0.8953 PF(A10)
// ln(0.001/0.999)
-6.9067 PF(A11)

// ln(0.7/0.3)
0.8472 PF(M0)
// ln(0.01/0.99)
-4.5951 PF(M1)

//ln(0.90/0.10)
2.1972 PF(J0)
//ln(0.05/0.95)
-2.9444 PF(J1)

PF(B) => AB.
PF(E) => AE.

AB ^ AE ^ PF(A00) => AA.
AB ^ !AE ^ PF(A01) => AA.
```

```
!AB ^ AE ^ PF(A10) => AA.
!AB ^ !AE ^ PF(A11) => AA.

AA ^ PF(J0) => AJ.
!AA ^ PF(J1) => AJ.

AA ^ PF(M0) => AM.
AA ^ PF(M1) => AM.

//parsing complete!
AA => (AB ^ AE ^ PF(A00)) v (AB ^ !AE ^ PF(A01)) v (!AB ^ AE ^ PF(A10))
   v (!AB ^ !AE ^ PF(A11)).
AB => (PF(B)).
AE => (PF(E)).
AJ => (AA ^ PF(J0)) v (!AA ^ PF(J1)).
AM => (AA ^ PF(M0)) v (AA ^ PF(M1)).
PF(_a) => (_a=B) v (_a=E) v (_a=A00) v (_a=A01) v (_a=A10) v
(_a=A11) v (_a=M0) v (_a=M1) v (_a=J0) v (_a=J1).
```

Output - MLN

The answer shows the marginal probability of a Burglary happening if John Calls. The exact value for this inference is 0.01628. Using Alchemy we get an answer close to this. This demonstrates that LP^{MLN} can handle Bayes Net Inference.

```
AB 0.00934907
```

Need to replace this

3.5.5 Pearl's Causal Model

The Firing Squad example is a well-known example that illustrates the idea of probabilistic causal model[Pearce et al.2000], proposed by Judea Pearl. The probabilistic causal

model for this domain can be encoded in this domain.

include graphic for pearls causal model

In the figure we see the firing squad scenario expressed as a causal model. U and W are exogenous variables as in their value does not depend on any external factors. C,A,B and D are endogenous variables as their values are determined by other variables.

Through such a model we can answer queries such as:

- Given that the court has not ordered the execution, what is the probability that the prisoner is dead?
- Given that the prisoner is dead, what is the probability that the court has ordered the execution?
- Given that Rifleman A shot, what is the probability that Rifleman B shot as well?
- Given that the captain has not given the signal and Rifleman A decided to shoot, what is the probability that the prisoner is dead? What is the probability that Rifleman B shot?
- Given that the prisoner is dead, what is the probability that the prisoner were not dead if Rifleman A had not shoot?

Through the encoding shown below we will try to answer the the first query.

LP^{MLN} Input

```
world(0,1)
iv(PC,PA,PB,PD,NC,NA,NB,ND)


// Court orders the execution
U
```

```
// Captain gives a signal
C(world)
// Rifleman A shoots
A(world)
// Rifleman B shoots
B(world)
// Prisoner dies
D(world)
Do(iv, world)
// rifleman A is nervous
W

// Suppose the court has ordered the execution with probability 0.6
//ln(p/1-p)
// ln(0.6/0.4) = 0.4054 U
0.4054 U

// Suppose rifleman A has probability 0.1 of
//pulling the trigger out of nervousness
//ln(p/1-p)
// ln(0.1/0.9) = -2.1972
-2.1972 W

//The court has not ordered execution
<= U.



// Model Description
//For every causal rule A causes B in the Diagram we write the rule
//B(x) <= A(x), not Do(B, x), not Do(not B, x)
//Here B is represented by {Pc,Pa,PB,PD} and not B by {NA,NB,NC,ND}
```

```
//U causes C
C(x) <= not Do(PC, x) ^ not Do(NC, x) ^ U .
//C causes A
A(x) <= not Do(PA, x) ^ not Do(NA, x) ^ C(x) .
//C causes B
B(x) <= not Do(PB, x) ^ not Do(NB, x) ^ C(x) .
//A causes D
D(x) <= not Do(PD, x) ^ not Do(ND, x) ^ A(x) .
//B causes D
D(x) <= not Do(PD, x) ^ not Do(ND, x) ^ B(x) .
//W causes A
A(x) <= not Do(PA, x) ^ not Do(NA, x) ^ W .

//Semantics of Do(x,y) is specified by these rules
C(x) <= Do(PC, x) .
A(x) <= Do(PA, x) .
B(x) <= Do(PB, x) .
D(x) <= Do(PD, x) .
```

Usage - MLN

```
lpmln2mln -i input.lpmln -q D -r out.txt -alchemy" -maxSteps 10000"
```

Intermediate Program - MLN

```
world={0,1}
iv={PC,PA,PB,PD,NC,NA,NB,ND}

// Court orders the execution
U
// Captain gives a signal
C(world)
```

```
// Rifleman A shoots
A(world)
// Rifleman B shoots
B(world)
// Prisoner dies
D(world)
Do(iv,world)
// rifleman A is nervous
W

// Suppose the court has ordered the execution with probability 0.6
//ln(p/1-p)
// ln(0.6/0.4) = 0.4054 U
0.4054 U

// Suppose rifleman A has probability 0.1 of
//pulling the trigger out of nervousness
//ln(p/1-p)
// ln(0.1/0.9) = -2.1972
-2.1972 W

(!U).



// Model Description
!Do(PC,x) ^ !Do(NC,x) ^ U => C(x).
!Do(PA,x) ^ !Do(NA,x) ^ C(x) => A(x).
!Do(PB,x) ^ !Do(NB,x) ^ C(x) => B(x).
!Do(PD,x) ^ !Do(ND,x) ^ A(x) => D(x).
!Do(PD,x) ^ !Do(ND,x) ^ B(x) => D(x).
!Do(PA,x) ^ !Do(NA,x) ^ W => A(x).
```

```
Do(PC,x) => C(x).
Do(PA,x) => A(x).
Do(PB,x) => B(x).
Do(PD,x) => D(x).
//parsing complete!
A(_a) => (!Do(PA,_a) ^ !Do(NA,_a) ^ C(_a)) v (!Do(PA,_a) ^ !Do(NA,_a) ^
   W) v (Do(PA,_a)).
B(_a) => (!Do(PB,_a) ^ !Do(NB,_a) ^ C(_a)) v (Do(PB,_a)).
C(_a) => (!Do(PC,_a) ^ !Do(NC,_a) ^ U) v (Do(PC,_a)).
D(_a) => (!Do(PD,_a) ^ !Do(ND,_a) ^ A(_a)) v (!Do(PD,_a) ^ !Do(ND,_a) ^
   B(_a)) v (Do(PD,_a)).
!Do(_a,_b).
```

Output - MLN

We are trying to find the answer to the query: Given that the court has not ordered the execution, what is the probability that the prisoner is dead? We have already encoded the first part of the query by using the constraint ¡= *U*. in the encoding. We find out the second part by querying for variable D.

Pr[D] = Pr[U=F, W, D] = 0.1

The output that we get is

```
D(0) 0.10454
D(1) 0.10454
```

3.5.6 Inconsistent Database

Consider an instance where you have conflicting information from different data sources. One data source may mention that an entity Jo is a resident bird however some other data source may say that Jo is a migratory bird. Although we cannot say with certainty that Jo is a migratory bird or a resident bird but we can say with certainty that Jo is a Bird. Suppose

we want to find out what is the probability of Jo being a Bird given that one KB says it is ResidentBird and other says it is a MigratoryBird. Such a domain can be easily encoded in this language.

LP^{MLN} Input

```
//The values that would be contained in the sort Entity. Note that it
   starts with a capital letter

entuty(Jo, Bob)

//Declares constants. In Forst Order Logic terminology this is a
   function/predicate.
//There could be differnet types of constants as documented above.
//These constants below are all singleton i.e. they are all boolean
   valued. They ca n be either true or false.

Bird(entity)
MigratoryBird(entity)
ResidentBird(entity)

// Rules

//If an entity is a resident bird, it implies that it is also a bird.
Bird(x) <= ResidentBird(x).

//If an entity is a migratory bird, it implies that it is also a bird.
Bird(x) <= MigratoryBird(x).

//An entity cannot be both migratory bird and resident bird at the same
   time
```

```
<= ResidentBird(x) ^ MigratoryBird(x).

//We have a conflicting database.
//Datbase Source 1 says that Jo is a resident bird.
//Datbase Source 2 says that Jo is a migratory bird.
//We believe that Source 1 is more accurate.
//This information could be encoded by giving data from Source 1 a
   higher weight.

//We believe that an entity Jo could be a resident bird
//with a weight of 2 corresponding to Source 1.
2 ResidentBird(Jo)

//We believe that an entity Jo could be a migratory bird
//with a weight of 1 corresponding to Source 2.
//Note that this weight is lower than that of Source 1.
1 MigratoryBird(Jo)
```

Usage - MLN

Here we are using Alchemy Solver and querying for Bird without giving any evidence file.

```
lpmln2mln -i input.lpmln -q Bird,ResidentBird,MigratoryBird -r out.txt
```

Intermediate Program - MLN

```
//Declares a sort of type entity. Entity acts as a container.

//The values that would be contained in the sort Entity. Note that it
   starts with a capital letter
entity={Jo,Bob}
```

```
//Declares constants. In Forst Order Logic terminology this is a
   function/predicate.
//There could be differnet types of constants as documented above.
//These constants below are all singleton i.e. they are all boolean
   valued. They ca n be either true or false.
Bird(entity)
MigratoryBird(entity)
ResidentBird(entity)

// Rules

//If an entity is a resident bird, it implies that it is also a bird.
ResidentBird(x) => Bird(x).

//If an entity is a migratory bird, it implies that it is also a bird.
MigratoryBird(x) => Bird(x).

//An entity cannot be both migratory bird and resident bird at the same
   time
(!ResidentBird(x) v !MigratoryBird(x)).

//We have a conflicting database.
//Datbase Source 1 says that Jo is a resident bird.
//Datbase Source 2 says that Jo is a migratory bird.
//We believe that Source 1 is more accurate.
//This information could be encoded by giving data from Source 1 a
   higher weight.

//We believe that an entity Jo could be a resident bird
//with a weight of 2 corresponding to Source 1.
2 ResidentBird(Jo)
```

```
//We believe that an entity Jo could be a migratory bird
//with a weight of 1 corresponding to Source 2.
//Note that this weight is lower than that of Source 1.
1 MigratoryBird(Jo)

//parsing complete!
Bird(_a) => (_a=Jo) v (ResidentBird(_a)) v (MigratoryBird(_a)).
MigratoryBird(_a) => (_a=Jo).
ResidentBird(_a) => (_a=Jo).
```

Output - MLN

The result states that, there is a 68.5% chance for entity Jo to be a Resident Bird and 22.5% chance of it being a Migratory Bird and hence 68.5 + 22.5 = 91% of it being a Bird. However since we have no information about Bob from either of the databases we do not have any conclusion about Bob and hence 0% chance of Bob being a Bird, Resident Bird or Migratory Bird.

```
Bird(Jo) 0.910959
Bird(Bob) 4.9995e-05
ResidentBird(Jo) 0.685981
ResidentBird(Bob) 4.9995e-05
MigratoryBird(Jo) 0.225027
MigratoryBird(Bob) 4.9995e-05
```

Usage - ASP

3.5.7 Probabilistic States

Consider an example where there are two states S_1 and S_2 and action A_1. The initial state is S_1. A_1 can be true or false. A_1 switches state from S_1 to S_2 if true or stays in the

same state. A_1 does not affect S_2 and the state does not change thereafter. A_1 has certain probability of being true. The question is to find the probability of a model in which state is S_2.

LP^{MLN} Input

```
step(0,1)
boolean(T,F)



p(boolean,step)
a(boolean,step)
PF(step)
next(step,step)

//We define literal next(0,1) as hard rule
//It means that a state transition from Time 0 to 1 is defined.
next(0,1).

//Literal PF makes the state transitions probabilistic
//At time 0 PF has 50% chances of being true.
//50% chance is denoted by ln(0.5)=-0.6931
-0.6931 PF(0)
-0.6931 PF(1)

//Rules

//Since only next(0,1) is true, t can only take the value 0 and
//t1 can only take the value 1 in the below rules.
```

```
//At time t if PF is True and action is True then state is True at t1=t
   +1
p(T,t1) <= a(T,t) ^ next(t,t1) ^ PF(T,t) ^ p(F,t).

//At time t if state is True and action is False then state remains True
    at t1=t+1
//Once the stte is True it does not change.
p(T,t1) <= a(F,t) ^ next(t,t1) ^ p(T,t).

//At time t if state is False and action is False then state remains
   False at t1=t+1
//Action has to be True for state to change from False to True.
p(F,t1) <= a(F,t) ^ next(t,t1) ^ p(F,t).

//exogenity fluents
//There has to be a cause for state to be True or False at time 0.
p(b,0) <= p(b,0).

//exogenity actions
//There has to be a cause for action to execute at any time t.
a(b,t) <= a(b,t).

//Uniquenses and existential constraint for state p

//State has to be True or False at any time.
p(T,t) v p(F,t).
//State cannot be True and False at the same time.
<= p(T,t) ^ p(F,t).

//Uniquenses and existential constraint for action a
```

```
//Action has to be True or False at any time.
a(T,t) v a(F,t).
//Action cannot be True and False at the same time.
<= a(T,t) ^ a(F,t).

//State is inertial
//State remains the same if no action occurs.
p(b,t) <= p(b,t1) ^ p(b,t) ^ next(t1,t).

//Initially state is False.
p(F,0).

//Action is True at time 0.
a(T,0).
```

Usage - MLN

```
lpmln input.lpmln –mln –q p,a
```

Intermediate Output - MLN

```
step={0,1,2}

boolean={True,False}



p(boolean,step)

a(boolean,step)

next(step,step)
```

```
0.8473 a(True,t)

a(True,t) ^ next(t,t1) => p(True,t1).

a(False,t) ^ next(t,t1) ^ p(True,t) => p(True,t1).

a(False,t) ^ next(t,t1) ^ p(False,t) => p(False,t1).

//exogenity fluents

p(b,0) => p(b,0).

//exogenity actions

a(b,t) => a(b,t).

//uec - p

p(True,t) v p(False,t).

(!p(True,t) v !p(False,t)).

a(True,t) v a(False,t).

(!a(True,t) v !a(False,t)).

// p is inertial

p(b,t1) ^ p(b,t) ^ next(t1,t) => p(b,t).
```

```
//parsing complete!
aux_p_0(step,boolean,step)
aux_p_0(_b,_a,t) <=> _a=True ^ a(True,t) ^ next(t,_b).
aux_p_1(step,boolean,step)
aux_p_1(_b,_a,t) <=> _a=True ^ a(False,t) ^ next(t,_b) ^ p(True,t).
aux_p_2(step,boolean,step)
aux_p_2(_b,_a,t) <=> _a=False ^ a(False,t) ^ next(t,_b) ^ p(False,t).
aux_p_3(step,boolean,step)
aux_p_3(_b,_a,t1) <=> p(_a,t1) ^ p(_a,_b) ^ next(t1,_b).
a(_a,_b) => (a(_a,_b)) v (_a=True ^ !a(False,_b)) v (_a=False ^ !a(True,
   _b)).
next(_a,_b) => (_a=0 ^ _b=1) v (_a=1 ^ _b=2).
p(_a,_b) => (_a=False ^ _b=0) v (EXIST t (aux_p_0(_b,_a,t))) v (EXIST t
   (aux_p_1(_b,_a,t))) v (EXIST t (aux_p_2(_b,_a,t))) v (_b=0 ^ p(_a,0)
   ) v (_a=True ^ !p(False,_b)) v (_a=False ^ !p(True,_b)) v (EXIST t1
   (aux_p_3(_b,_a,t1))).
```

Output - MLN

The output shows that Action a is True at time 0(which is what we had encoded) and at time 1 it has a 50% chance (since Action a is equally likely to be False or True at time 1). Note that these values are not exactly 50% since alchemy does not do an exact inference. The value of p(T,1) is the value of probability P(p(T,1) — a(T,0), PF(0), p(F,0)) = 5/8.

```
p(T,0) 4.9995e-05
p(T,1) 0.606989
p(F,0) 0.99995
p(F,1) 0.393011
a(T,0) 0.99995
a(T,1) 0.478002
a(F,0) 4.9995e-05
a(F,1) 0.521998
```

3.5.8 *Monty Hall Problem*

This is an example of a complicated bigger domain that this framework can solve. A player is given an opportunity to select one of the three closed doors, behind one of which there is a prize. Behind the other two doors are empty rooms. Once the player has chosen Monty opens one of the remaining closed doors which does not contain the prize. The player is then asked to switch his selection to other unopened door. We query the probability of the player winning if he switches.

LP^{MLN} Input

```
doors = {1,2,3}
numbers = {2,3}
boolean = {T,F}
attributes = {Attropen ,Attrselected ,Attrprize}

Open(doors)

Selected(doors)

Prize(doors)

CanOpen(doors,boolean)

Obs(attributes ,doors)

UnObs(attributes ,doors)

// new predicate

Intervene(attributes)
```

```
NumDefProb(attributes ,numbers)

PossWithDefProb(attributes ,doors)

PossWithAssProb(attributes ,doors)

// Regular part

CanOpen(d,F) <= Selected(d).
CanOpen(d,F) <= Prize(d).

CanOpen(d,T) <= !CanOpen(d,F).

<= CanOpen(d,T) ^ CanOpen(d,F).

<= Prize(d1) ^ Prize(d2) ^ d1 != d2.

<= Selected(d1) ^ Selected(d2) ^ d1 != d2.

<= Open(d1) ^ Open(d2) ^ d1 != d2.

// Random Selection rules

Prize(1) v Prize(2) v Prize(3) <= !Intervene(Attrprize).

Selected(1) v Selected(2) v Selected(3) <= !Intervene(Attrselected).

Open(1) v Open(2) v Open(3) <= !Intervene(Attropen).

<= Open(d) ^ !CanOpen(d,T) ^ !Intervene(Attropen).
```

```
PossWithDefProb(Attrprize ,d) <= !PossWithAssProb(Attrprize ,d) ^
!Intervene(Attrprize).

NumDefProb(Attrprize ,2) <= Prize(d1) ^ PossWithDefProb(Attrprize ,d1) ^
PossWithDefProb(Attrprize ,d2) ^ d1!=d2.

NumDefProb(Attrprize ,3) <= Prize(d1) ^ PossWithDefProb(Attrprize ,d1) ^
PossWithDefProb(Attrprize ,d2) ^ PossWithDefProb(Attrprize ,d3) ^d1!=d2
   ^
d1!=d3 ^ d2!=d3.

-0.6931 not not NumDefProb(Attrprize ,2)

-0.4055 not not NumDefProb(Attrprize ,3)

PossWithDefProb(Attrselected ,d) <= !PossWithAssProb(Attrselected ,d) ^
!Intervene(Attrselected).

NumDefProb(Attrselected ,2) <= Selected(d1) ^PossWithDefProb(
   Attrselected ,d1) ^
PossWithDefProb(Attrprize ,d2) ^ d1!=d2.

NumDefProb(Attrselected ,3) <= Selected(d1) ^PossWithDefProb(
   Attrselected ,d1) ^
PossWithDefProb(Attrselected ,d2) ^PossWithDefProb(Attrselected ,d3) ^
d1!=d2 ^ d1!=d3 ^ d2!=d3.

-0.6931 not not NumDefProb(Attrselected ,2)

-0.4055 not not NumDefProb(Attrselected ,3)
```

```
PossWithDefProb(Attropen ,d) <= !PossWithAssProb(Attropen ,d) ^
!Intervene(Attropen) ^ CanOpen(d,T).

NumDefProb(Attropen ,2) <= Open(d1) ^ PossWithDefProb(Attropen ,d1) ^
PossWithDefProb(Attropen ,d2) ^ d1!=d2.

NumDefProb(Attropen ,3) <= Open(d1) ^ PossWithDefProb(Attropen ,d1) ^
PossWithDefProb(Attropen ,d2) ^ PossWithDefProb(Attropen ,d3) ^ d1!=d2 ^
d1!=d3 ^ d2!=d3.

-0.6931 not not NumDefProb(Attropen ,2)

-0.4055 not not NumDefProb(Attropen ,3)

// **** Obs ****

Obs(Attrselected ,1).

<= Obs(Attrselected ,1) ^ !Selected(1).

Obs(Attropen ,2).

<= Obs(Attropen ,2) ^ !Open(2).

UnObs(Attrprize ,2).

<= UnObs(Attrprize ,2) ^ Prize(2).
```

Usage - MLN

```
lpmln2al test -q Prize -maxSteps 10000
```

Intermediate Program - MLN

```
doors={1,2,3}

numbers={2,3}

boolean={T,F}

attributes={Attropen,Attrselected,Attrprize}



Open(doors)

Selected(doors)

Prize(doors)

CanOpen(doors,boolean)

Obs(attributes,doors)

UnObs(attributes,doors)

// new predicates

Intervene(attributes)

NumDefProb(attributes,numbers)

PossWithDefProb(attributes,doors)
```

```
PossWithAssProb(attributes,doors)

// Regular part

Selected(d) => CanOpen(d,F).
Prize(d) => CanOpen(d,F).

!CanOpen(d,F) => CanOpen(d,T).

(!CanOpen(d,T) v !CanOpen(d,F)).

(!Prize(d1) v !Prize(d2) v (d1=d2)).

(!Selected(d1) v !Selected(d2) v (d1=d2)).

(!Open(d1) v !Open(d2) v (d1=d2)).

// Random Selection rules

!Intervene(Attrprize) => Prize(1) v Prize(2) v Prize(3).

!Intervene(Attrselected) => Selected(1) v Selected(2) v Selected(3).

!Intervene(Attropen) => Open(1) v Open(2) v Open(3).

(!Open(d) v CanOpen(d,T) v Intervene(Attropen)).

!PossWithAssProb(Attrprize,d) ^ !Intervene(Attrprize) =>
PossWithDefProb(Attrprize,d).

Prize(d1) ^ PossWithDefProb(Attrprize,d1) ^ PossWithDefProb(Attrprize,d2
```

```
   ) ^
d1!=d2 => NumDefProb(Attrprize,2).

Prize(d1) ^ PossWithDefProb(Attrprize,d1) ^ PossWithDefProb(Attrprize,d2
   ) ^
PossWithDefProb(Attrprize,d3) ^ d1!=d2 ^ d1!=d3 ^ d2!=d3 =>
NumDefProb(Attrprize,3).

-0.6931 NumDefProb(Attrprize,2)

-0.4055 NumDefProb(Attrprize,3)

!PossWithAssProb(Attrselected,d) ^ !Intervene(Attrselected) =>
PossWithDefProb(Attrselected,d).

Selected(d1) ^ PossWithDefProb(Attrselected,d1) ^
PossWithDefProb(Attrprize,d2) ^ d1!=d2 => NumDefProb(Attrselected,2).

Selected(d1) ^ PossWithDefProb(Attrselected,d1) ^
PossWithDefProb(Attrselected,d2) ^ PossWithDefProb(Attrselected,d3) ^
d1!=d2 ^ d1!=d3 ^ d2!=d3 => NumDefProb(Attrselected,3).

-0.6931 NumDefProb(Attrselected,2)

-0.4055 NumDefProb(Attrselected,3)

!PossWithAssProb(Attropen,d) ^ !Intervene(Attropen) ^ CanOpen(d,T) =>
PossWithDefProb(Attropen,d).

Open(d1) ^ PossWithDefProb(Attropen,d1) ^ PossWithDefProb(Attropen,d2) ^
d1!=d2 => NumDefProb(Attropen,2).
```

```
Open(d1) ^ PossWithDefProb(Attropen,d1) ^ PossWithDefProb(Attropen,d2) ^
PossWithDefProb(Attropen,d3) ^ d1!=d2 ^ d1!=d3 ^ d2!=d3 =>
NumDefProb(Attropen,3).

-0.6931 NumDefProb(Attropen,2)

-0.4055 NumDefProb(Attropen,3)

// **** Obs ****

Obs(Attrselected,1).

(!Obs(Attrselected,1) v Selected(1)).

Obs(Attropen,2).

(!Obs(Attropen,2) v Open(2)).

UnObs(Attrprize,2).

(!UnObs(Attrprize,2) v !Prize(2)).

//parsing complete!
aux_NumDefProb_0(numbers,attributes,doors,doors)
aux_NumDefProb_0(_b,_a,d1,d2) <=> _a=Attrprize ^ _b=2 ^ Prize(d1) ^
PossWithDefProb(Attrprize,d1) ^ PossWithDefProb(Attrprize,d2) ^ d1!=d2.
aux_NumDefProb_1(numbers,attributes,doors,doors,doors)
aux_NumDefProb_1(_b,_a,d1,d2,d3) <=> _a=Attrprize ^ _b=3 ^ Prize(d1) ^
PossWithDefProb(Attrprize,d1) ^ PossWithDefProb(Attrprize,d2) ^
PossWithDefProb(Attrprize,d3) ^ d1!=d2 ^ d1!=d3 ^ d2!=d3.
```

```
aux_NumDefProb_2(numbers,attributes,doors,doors)
aux_NumDefProb_2(_b,_a,d1,d2) <=> _a=Attrselected ^ _b=2 ^ Selected(d1)
   ^
PossWithDefProb(Attrselected,d1) ^ PossWithDefProb(Attrprize,d2) ^ d1!=
   d2.
aux_NumDefProb_3(numbers,attributes,doors,doors,doors)
aux_NumDefProb_3(_b,_a,d1,d2,d3) <=> _a=Attrselected ^ _b=3 ^
Selected(d1) ^ PossWithDefProb(Attrselected,d1) ^ PossWithDefProb(
   Attrselected,d2)
^ PossWithDefProb(Attrselected,d3) ^ d1!=d2 ^ d1!=d3 ^ d2!=d3.
aux_NumDefProb_4(numbers,attributes,doors,doors)
aux_NumDefProb_4(_b,_a,d1,d2) <=> _a=Attropen ^ _b=2 ^ Open(d1) ^
PossWithDefProb(Attropen,d1) ^ PossWithDefProb(Attropen,d2) ^ d1!=d2.
aux_NumDefProb_5(numbers,attributes,doors,doors,doors)
aux_NumDefProb_5(_b,_a,d1,d2,d3) <=> _a=Attropen ^ _b=3 ^ Open(d1) ^
PossWithDefProb(Attropen,d1) ^ PossWithDefProb(Attropen,d2) ^
PossWithDefProb(Attropen,d3) ^ d1!=d2 ^ d1!=d3 ^ d2!=d3.
CanOpen(_a,_b) => (_b=F ^ Selected(_a)) v (_b=F ^ Prize(_a)) v (_b=T ^
!CanOpen(_a,F)).
NumDefProb(_a,_b) => (EXIST d1,d2 (aux_NumDefProb_0(_b,_a,d1,d2))) v
(EXIST d1,d2,d3 (aux_NumDefProb_1(_b,_a,d1,d2,d3))) v (EXIST d1,d2
(aux_NumDefProb_2(_b,_a,d1,d2))) v (EXIST d1,d2,d3
(aux_NumDefProb_3(_b,_a,d1,d2,d3))) v (EXIST d1,d2
(aux_NumDefProb_4(_b,_a,d1,d2))) v (EXIST d1,d2,d3
(aux_NumDefProb_5(_b,_a,d1,d2,d3))).
Obs(_a,_b) => (_a=Attrselected ^ _b=1) v (_a=Attropen ^ _b=2).
Open(_a) => (_a=1 ^ !Open(2) ^ !Open(3) ^ !Intervene(Attropen)) v (_a=2
   ^
!Open(1) ^ !Open(3) ^ !Intervene(Attropen)) v (_a=3 ^ !Open(1) ^ !Open
   (2)
^ !Intervene(Attropen)).
```

```
PossWithDefProb(_a,_b) => (_a=Attrprize ^ !PossWithAssProb(Attrprize,_b)
    ^
!Intervene(Attrprize)) v (_a=Attrselected ^
!PossWithAssProb(Attrselected,_b) ^ !Intervene(Attrselected)) v
(_a=Attropen ^ !PossWithAssProb(Attropen,_b) ^ !Intervene(Attropen) ^
CanOpen(_b,T)).
Prize(_a) => (_a=1 ^ !Prize(2) ^ !Prize(3) ^ !Intervene(Attrprize)) v
(_a=2 ^ !Prize(1) ^ !Prize(3) ^ !Intervene(Attrprize)) v (_a=3 ^
!Prize(1) ^ !Prize(2) ^ !Intervene(Attrprize)).
Selected(_a) => (_a=1 ^ !Selected(2) ^ !Selected(3) ^
!Intervene(Attrselected)) v (_a=2 ^ !Selected(1) ^ !Selected(3) ^
!Intervene(Attrselected)) v (_a=3 ^ !Selected(1) ^ !Selected(2) ^
!Intervene(Attrselected)).
UnObs(_a,_b) => (_a=Attrprize ^ _b=2).
!Intervene(_a).
!PossWithAssProb(_a,_b).
```

Output - MLN

```
Prize(1) 0.312019
Prize(2) 4.9995e-05
Prize(3) 0.687981
```

The output clearly shows that initially if Door 1 was selected and Door 2 was opened, the chances of winning if you switch to Door 3 is 2/3.

3.6 BNF for input Language

```
start ::= prog
prog ::= prog NEWLINE predicate
prog ::= predicate
prog ::= prog NEWLINE rule
prog ::= rule
prog ::= prog NEWLINE domain
prog ::= domain
prog ::= prog NEWLINE decl
prog ::= decl
prog ::= prog NEWLINE
prog ::=
rule ::= REVERSE_IMPLICATION body DOT
rule ::= number REVERSE_IMPLICATION body
rule ::= head DISJUNCTION bodydef DOT
rule ::= number head DISJUNCTION bodydef
rule ::= head REVERSE_IMPLICATION body DOT
rule ::= number head REVERSE_IMPLICATION body
rule ::= number NEGATION NEGATION LBRACKET head REVERSE_IMPLICATION
   body RBRACKET
rule ::= LPAREN head RPAREN REVERSE_IMPLICATION body DOT
body ::= body CONJUNCTION bodydef
head ::= head DISJUNCTION bodydef
head ::= bodydef
body ::= bodydef
bodydef ::= literal
bodydef ::= NEGATION literal
bodydef ::= NEGATION NEGATION literal
bodydef ::= LBRACKET NEGATION NEGATION literal RBRACKET
bodydef ::= string EQUAL string
```

```
bodydef ::= NEGATION string EQUAL string
bodydef ::= string NEGATION EQUAL string
literal ::= string LBRACKET variables RBRACKET EQUAL variable
literal ::= string LBRACKET variables RBRACKET
literal ::= variable
literal ::= string EQUAL COUNT LPAREN aggregateCum RPAREN
literal ::= string EQUAL SUM LPAREN aggregateCum RPAREN
predicate ::= literal DOT
predicate ::= number literal
predicate ::= number NEGATION NEGATION literal
predicate ::= number NEGATION literal
predicate ::= NEGATION NEGATION literal DOT
variables ::= variable
variables ::= variables COMMA variable
variable ::= string
variable ::= number
string ::= STRING
number ::= NUMBER
domain ::= string EQUAL domains
domains ::= LPAREN variables RPAREN
decl ::= string LBRACKET variables RBRACKET
decl ::= string
```

3.6.1 BNF Terminals

WS	[\t \v \f]
NL	[\n]
”.”	DOT
[-]?([0-9]+[.]1}[0-9]+[0-9]+) —	NUMBER
”not”|”NOT”	NEGATION
[a-zA-Z]+[a-zA-Z0-9]*	STRING
"=>"	IMPLICATION
"<="	REVERSE_IMPLICATION
”}”	RPAREN
””	LPAREN
”=”	EQUAL
”(”	LBRACKET
”)”	RBRACKET
”,”	COMMA
”∧”	CONJUNCTION
WS+ ”v” WS+	DISJUNCTION
”!”	NEGATION
”;”	SEMI_COLON
”:”	COLON
NL	NEWLINE