P-log System manual

July 12, 2020

Contents

1 System installation

For the latest instructions on system installation, please refer to `https://github.com/iensen/plog2.0/wiki/Installation-Instructions`.

2 System usage

The system accepts ASCII files as inputs. The file can be of any extension (we recommend .plog).

A file acceptable by the system should consist of:

1. A P-log program, consisting of three sections:
   - sorts definitions,
   - attribute declarations, and
   - program rules
2. A query.

For example, consider the following program, where sort definitions start with keywords `sorts`, attribute declarations start with keyword `attributes`, program rules start with keyword `statements`, and query starts with symbol `?`:

```
sorts

#dice={d1,d2}.
#score={1,2,3,4,5,6}.
#person={mike,john}.
#bool = {true,false}.


attributes

roll:#dice->#score.
owns:#dice,#person->#bool.

statements

owns(d1,mike).
owns(d2,john).

random(roll(D)).
```

```
%probability information

pr(roll(D)=6|owns(D,mike))=1/4.

? roll(d1)=1.
```

The program, originally introduced in [3], describes a scenario with two dice being rolled, belonging to Mike and John respectively. Mike's die is more likely to produce *6* as an outcome, as defined by the pr-atom

$$pr(roll(D) = 6 \mid owns(D, mike)) = 1/4.$$

By the so called *principle of indifference* used by P-log, the probabilities of the remaining outcomes equal to (1 - 1/4)/5 = 3/20. Thus, the answer to the query $roll(d1) = 1$ is 3/20. Note that all outcomes of John's die are equally likely, so the answer to the query $roll(d2) = 1$ is equal to 1/6.

To compute the query probability using the system, we need to store it in a file and run the command:

$$plog2\ [path\_to\_file]$$

where $plog2$ is the name of the p-log executable. For example, assuming the file is stored in $plogapp/tests/paper/dice1.plog$ in our system, we get the following output:

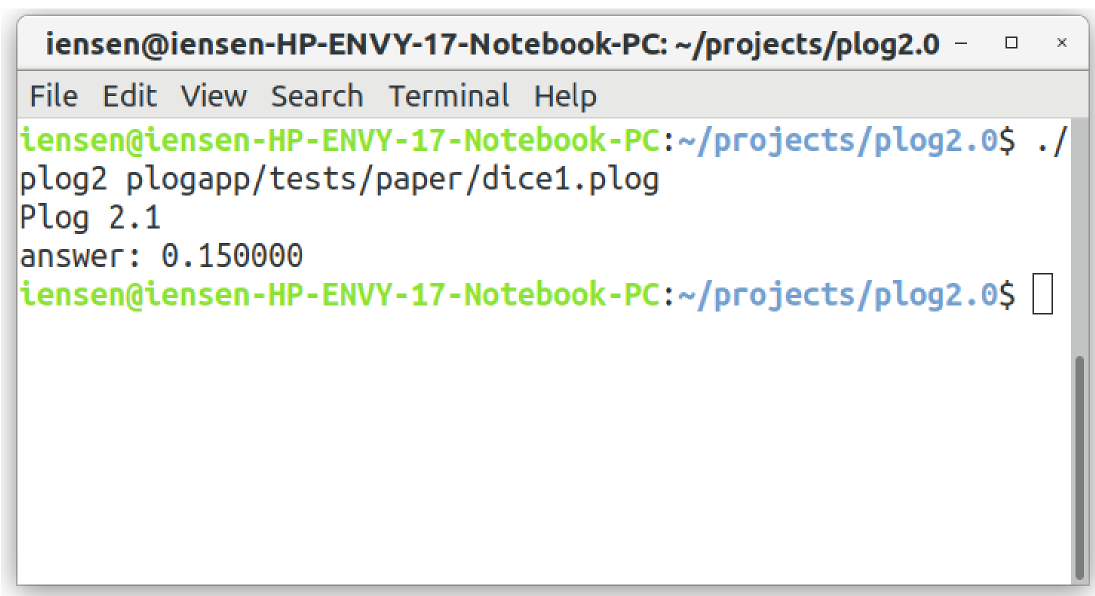

Figure 1: P-log output

The solver also allows to compute possible worlds of a given program. For that, pass an option `--mode=pw` to the solver. For example, in order to compute possible worlds of a program stored in `plogapp/tests/causal/courtorder.plog`, run the command `./plog2 --mode=pw plogapp/tests/causal/courtorder.plog`, where plog2 is the system executable. Below is an example output:

```
 ./plog2 --mode=pw plogapp/tests/causality/courtorder.plog
Plog 2.1
Possible Worlds:
1: {court_order(0), court_order(1) = false, court_order(2) = false,
 court_order(3) = false, captain_order(1), captain_order(0) = false,
 captain_order(2) = false, captain_order(3) = false,
shoot(a, 2), shoot(b, 2), shoot(a, 0) = false,
shoot(a, 1) = false, shoot(a, 3) = false, shoot(b, 0) = false,
 shoot(b, 1) = false, shoot(b, 3) = false, dead(3),
dead(0) = false, dead(1) = false, dead(2) = false}

Probabilities:
1: 1
```

The details of the syntax of the language can be found in [2] and in the following sections.

3 Command Line Options

In this section we will describe the meanings of command line options supported by P-log. As of right now, the system requires a single argument which is a path to the p-log file.

In addition, it supports the following named arguments:

- `--mode=X`, where `X` is one of `pw` or `query`. When `X` is `pw`, the solver computes possible worlds of a given program. Otherwise, it computes the probability of the query. The default value is `query`.

- `--algo=X`, where `X` is one of `naive` or `dco`. When `X` is `naive`, the solver uses naive algorithm to compute the query by computing possible worlds and iterating over all of them. The algorithm is similar to the translate algorithm from [4], but, unlike the implementation by its author, our uses a more modern ASP solver, Clingo. When `X` is `dco`, the solver uses special algorithm for dynamically causally ordered programs, described in [1], which can be more efficient. Note that when the input program is not dynamically causally ordered, and `X` is `dco`, the solver behavior is undefined.

See Section 2 for an example of using this argument.

4 Syntax Description

4.1 Sort definitions

This section starts with a keyword $sorts$ followed by a collection of sort definitions of the form:

$$sort\_name = sort\_expression.$$

sort_name is an identifier preceeded by the pound sign (#). *sort_expression* on the right hand side denotes a collection of strings called $a\ sort$. As of right now, the system only supports a basic sort definition of the form $sort\_name = \{t_1, \ldots, t_n\}$, where $t_1, \ldots, t_n$ are ground terms..

4.2 Attribute Declarations

The second part of a P-log program starts with the keyword `attributes` and is followed by statements of the form

$$attr\_symbol : \#sortName_1, \ldots, \#sortName_n \rightarrow \#sortName$$

where $attr\_symbol$ is an identifier (in what follows referred to as an attribute symbol) and $\#sortName$, $\#sortName_1$,...,$\#sortName_n$ are sorts defined in sort definitions section of the program.

Multiple declarations containing the same attribute symbol are not allowed. Attributes with no arguments must be declared as $attr\_symbol : sortName_1$ (example: `a:#bool`).

4.3 Program Statements

The third part of a P-log program starts with the keyword *statements* followed by a collection *rules* and *pr-atoms* (in any order). Here we give a brief overview of the system assuming that a reader is familiar with notions of a term and a literal. For the details, please refer to [2].

4.3.1 Program Rules

P-log rules are of the following form:

$$a(t) = y \leftarrow l_1, \ldots, l_k, not\ l_{k+1} \ldots not\ l_n. \tag{1}$$

where $a(t) = y$ is a program atom and $l_1, \ldots, l_k$ are program literals. The atom $a(t) = y$ is referred to as the *head* of the rule, and $l_1, \ldots, l_k$ is referred to as the *body* of the rule. A rule with $k = 0$ is referred to as a *fact*. The system allows for a shorthand $a(t)$ for $a(t) = true$, and standard arithmetic relations of the form $t_1\ op\ t_2$, where $op \in$

{>=, <=, >, <, =, !=} in the body of the rule . Three special kinds of rules are *random selection rules*, *observations* and *actions*. As defined in [2], random selection rule is a rule whose head is of one of the forms $random(a, p)$ or $random(a)$. Action is a fact whose head is $do(a, y)$[1]. Observation is a fact whose head is of the form $obs(a, y)$ Right now, the system only supports observations and actions whose arguments are positive atoms.

4.3.2 Pr-Atoms

Probability atoms (in short, pr-atoms) are of the form

$$pr(a(t) = y|B) = v.$$

where $a(t) = y$ is a program atom, B is a collection of e-literals and v written as a fraction of the form n/m, where n and m are natural numbers. Example:

```
pr(roll(D)=6 | not owns(D,mike))=1/4.
```

4.4 Program Examples

For a collection of working P-log programs used for system testing and development, please refer to

`https://github.com/iensen/plog2.0/tree/master/plogapp/tests`.

5 Error Checking

Please be aware that some of the errors are not implemented yet.

Additional error checking is performed when no syntax error is found. Majority of the errors are related to types.

5.1 Type errors

Type errors are considered as serious issues which make it impossible to compile and execute the program. Type errors can occur in all four sections of a P-log program.

5.1.1 Sort definition errors

The following are possible causes of a sort definition error that will result in a type error message from $\mathcal{SPARC}$:

[1]The system doesn't support rule labels and actions of the form $do(r, a, y)$ defined in [2]. Therefore there has to be at most one action of each attribute term. This is a limitation we plan to overcome in future.

1. A set-theoretic expression (statement 5 in section 4.1) containing a sort name that has not been defined.

   Example:

   ```
   sorts
   #s={a}.
   #s2=#s1-#s.
   ```

2. Declaring a sort more than once.

   Example:

   ```
   sorts
   #s={a}.
   #s={b}.
   ```

3. An identifier range $id_1..id_2$ (statement 2 in section 4.1) where id_1 is greater than id_2.

   Example:

   ```
   sorts
   #s=zbc..cbz.
   ```

4. A numeric range $n_1..n_2$ (statement 1 in section 4.1) where n_1 is greater than n_2.

   Example:

   ```
   sorts
   #s=100500..1.
   ```

5. A numeric range (statement 2 in section 4.1) $n_1..n_2$ that contains an undefined constant.

   Example:

   ```
   #const n1=5.
   sorts
   #s=n1..n2.
   ```

6. An identifier range $id_1..id_2$ (statement 3 in section 4.1) where the length of id_1 is greater than the length of id_2.

   Example:

   ```
   sorts
   #s=abc..a.
   ```

7. A concatenation (statement 4 in section 4.1) that contains a non-basic sort.

   Example:

   ```
   sorts
   #s={f(a)}.
   #sc=[a][#s].
   ```

8. A record definition (statement 5 in section 4.1) that contains an undefined sort.

   Example:

   ```
   sorts
   #s=1..2.
   #fs=f(s,s2).
   ```

9. A record definition (statement 5 in section 4.1) that contains a condition with relation $>, <, \geq, \leq$ such that the corresponding sorts are not basic.

   Example:

   ```
   #s={a,b}.
   #s1=f(#s).
   #s2=g(s1(X),s2(Y)):X>Y.
   ```

10. A variable that is used more than once in a record definition (statement 5 in section 4.1).

    Example:

    ```
    sorts
    #s1={a}.
    #s=f(#s1(X),#s1(X)):(X!=X).
    ```

11. A sort that contains an empty collection of ground terms.

    Example

    ```
    sorts
    #s1={a,b,c}
    #s=#s1-{a,b,c}.
    ```

5.1.2 Attribute declarations errors

1. An attribute with the same name is declared more than once. *Example:*

```
sorts
#s={a}.
attributes
a: #s -> #s.
a: #s,#s ->#s.
```

2. An attribute declaration contains an undefined sort. *Example:*

```
sorts
#s={a}.
attributes
p:#ss.
```

5.1.3 Program statements errors

In program rules and pr-atoms we first check each atom of the form $a(t_1, \ldots, t_n) = y$ and each term occurring in the program Π for satisfying the definitions of program atom and program term correspondingly (see Section 3 of [2]).In addition, we check that no sort occurs in a head of a rule of Π.

5.2 Type warnings

This section is to be completed and implemented. At the very least, we plan to support finite-domain constraint-based typechecking which will flag rules with no ground instances.

References

[1] Evgenii Balai et al. *Investigating and Extending P-log*. PhD thesis, 2017.

[2] Evgenii Balai, Michael Gelfond, and Yuanlin Zhang. P-log: refinement and a new coherency condition. *Annals of Mathematics and Artificial Intelligence*, 86(1):149–192, Jul 2019.

[3] Michael Gelfond and Yulia Kahl. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press, 2014.

[4] Weijun Zhu. *Plog: Its algorithms and applications*. PhD thesis, 2012.