P-log System manual

November 30, 2021

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 [4], 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:

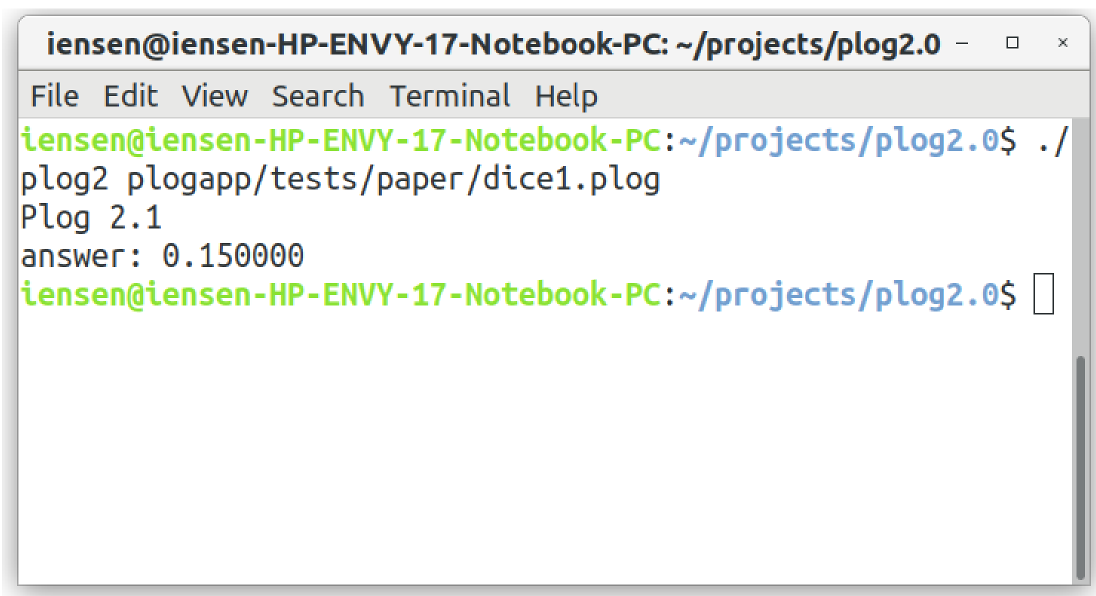

Figure 1: P-log output

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 [5], 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 preceded by the pound sign (#). *sort_expression* on the right hand side denotes a collection of strings called $a\ sort$.

Basic sorts are defined as named collections of numbers and *identifiers*, i.e, strings consisting of

- letters: $\{a, b, c, d, ..., z, A, B, C, D, ..., Z\}$
- digits: $\{0, 1, 2, ..., 9\}$
- underscore: _

  and starting with a lowercase letter.

A *non-basic sort* also contains at least one *record* of the form $id(\alpha_1, \ldots, \alpha_n)$ where id is an identifier and
$\alpha_1, \ldots, \alpha_n$ are either identifiers, numbers or records.

We define sorts by means of expressions (in what follows sometimes referred to as statements) of six types:

1. **numeric range** is of the form:

   $$number_1..number_2$$

   where $number_1$ and $number_2$ are non-negative numbers such that $number_1 \leq number_2$. The expression defines the set

   of sequential numbers
   $\{number_1, number_1 + 1, \ldots, number_2\}$.

   Example:

   ```
   #sort1=1..3.
   ```

   `#sort1` consists of numbers $\{1, 2, 3\}$.

2. **set of ground terms** is of the form:

$$\{t_1, .., , t_n\}$$

The expression denotes a set of *ground terms* $\{t_1, ..., t_n\}$, defined as follows:

- numbers and identifiers are ground terms;
- If f is an identifier and $\alpha_1, \ldots, \alpha_n$ are ground terms, then $f(\alpha_1, \ldots, \alpha_n)$ is a ground term.

Example:

```
#sort1={f(a),a,b,2}.
```

3. **set of records** is of the form:

$$f(sort\_name_1, ..., sort\_name_n)$$

where f is an identifier, for $1 \leq i \leq n$, $sort\_name_i$ occurs in one of the preceding sort definitions.

The expression defines a collection of ground terms $\{f(t_1, \ldots, t_n) : t_1 \in s_i \wedge \cdots \wedge t_n \in s_n\}$

Example

```
#s1=1..2.
#s2 = {a,b}.
#sf=f(#s1, #s2)).
```

The sort `#sf` consists of records $\{f(1, 1, 2), f(1, 1, 1), f(2, 1, 1)\}$

4. **set-theoretic expression** is in one of the following forms:

- $\#sort\_name$
- an expression of the form 1, 2, or 3, denoting a set of ground terms
- $(S_1 \bigtriangledown S_2)$, where $\bigtriangledown \in \{+, -, *\}$ and both S_1 and S_2 are set theoretic expressions

$\#sort\_name$ must be a name of a sort occurring in one of the preceding sort definitions. The operations $+$ $*$ and $-$ stand for union, intersection and difference correspondingly.

Example :

```
#sort1={a,b,2}.
#sort2={1,2,3} + {a,b,f(c)} + f(#sort1).
```

`#sort2` consists of ground terms $\{1, 2, 3, a, b, f(c), f(a), f(b), f(2)\}$.

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:

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

where $\Leftarrow \in \{\leftarrow, \stackrel{+}{\leftarrow}\}$, $a(\vec{t}) = y$ is a program atom and $l_1, \ldots, l_n$ are program literals.

The atom $a(\vec{t}\,) = y$ 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 .

The rules with $\stackrel{+}{\leftarrow}$ are called consistency restoring rules (cr-rules) [3]. The system implements cr-rules semantics with preference relation based on minimal cardinality of the selected rules. Cr-rules are only supported for naive algorithm.

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 $obs(a, y)$ The system only supports observations and actions whose arguments are positive atoms. Note that an observation of a negative atom, say $b \neq z$,

[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 for each attribute term. This limitation will be removed in future.

can be easily modeled by introducing an extra rule $a = y \leftarrow b \neq z$, where a is a fresh attribute term, and adding an observation $obs(a, y)$.

4.3.2 Pr-Atoms

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

$$pr(a(\vec{t}) = y|B) = v.$$

where $a(\vec{t}) = y$ 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.
```

4.3.3 Query

The last (optional) part of a P-log program is a query of the form $?l$, where l is a program literal.

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.

In this section, we describe additional errors which are detected for a program satisfying P-log grammar[2]. They are necessary to ensure the syntax and semantic requirements of P-log not captured by the grammar. We expect this section to be more useful for developers of the system , but we hope that users will find it helpful as well.

For example, consider the following program:

```
sorts
#s=1..5.
attributes
a : #s.
statements
a = 6.
```

[2]https://github.com/iensen/plog2.0/blob/ecd3e243383fa75b4a88baf254f4702baaef838f/libplog/plog/ploggrammar.yy#L191

The language grammar accepts this program, but its only statement `a = 6` contains a type error, since 6 doesn't belong to sort `#s`. This is not a correct program per our language specification. Therefore, the solver will produce an error about it.

In what follows we describe the kind of errors the system can produce. The first three subsections correspond to three sections of the program: sorts, attributes and statements. The last section describes errors for program's query.

5.1 Sort Definition Errors

The following are possible causes of a sort definition error that will result in a type error message.

1. 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.
   ```

2. A set-theoretic expression (statement 4 in section 4.1) containing a sort name that has not been defined.

   Example:

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

3. Declaring a sort more than once.

   Example:

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

4. A record definition (statement 3 in section 4.1) that contains an undefined sort.

   Example:

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

   Sort `#s2` is not defined by the program.

5.2 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.
```

3. An attribute is named `do`, `obs`, or `random`. *Example:*

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

4. An attribute name coincides with the name of a record belonging to one of the sorts. *Example:*

```
sorts
#s1={f(a)}.
#s2 = {1,2,3}.
attributes
f:#s1.
```

   Here there is an attribute named f and a record $f(a)$ belonging to `#s1`.

5.3 Errors in Program Statements

To describe errors in program statements, we need the following definitions.

Definition 1 (Term) *Terms* are divided into three kinds:

(1) numbers, identifiers and variables are *terms*;

(2) if f is an identifier and $\alpha_1, \ldots, \alpha_n$ are *terms*, then $f(\alpha_1, \ldots, \alpha_n)$ is a *term*.

(3) standard arithmetic expressions constructed from numbers, variables and operators $+$, $-$, $*$, $/$ are terms.

We will refer to terms of the form (3) as *arithmetic terms*. We will also say that the term of the form (2) is *formed by* f. Terms formed by attributes are called *attribute terms*. $\square$

Consider program rule r:

$$a(\vec{t}) = y \Leftarrow l_1, \ldots, l_k, not\ l_{k+1} \ldots not\ l_n. \tag{2}$$

where $a(\vec{t}) = y$, is an atom, and $l_1, \ldots, l_n$ are literals of the program of the form $\tau_1 \odot \tau_2$, where τ_1 may be formed by an attribute, and τ_2 is a term from the program's signature (not containing attribute terms).

To ensure correctness of the program, we require that r satisfies the following constraints:

(1) a is an attribute name;

(2) if $\Leftarrow$ is $\stackrel{+}{\leftarrow}$, then $a \notin \{obs, do\}$;

(3) if $a \in \{obs, do\}$, then $n = 0$ (that is, the body of the rule is empty);

(4) if a is obs or do, then $\vec{t} = \langle a(\vec{q}), y \rangle$, where $a(\vec{q})$ is an attribute term of program's signature.

Additionally, for each literal $\tau_1 \odot \tau_2$ occurring in a program rule, where $\odot \in \{>, \leq, \geq, =, \neq, <\}$ and τ_1, τ_2 are terms, the system will produce an error in the following cases:

(1) τ_2 contains a term of the form $f(t_1, \ldots, t_n)$, where f is an attribute name;

(2) τ_1 is of the form $f(t_1, \ldots, t_n) = y$, f is an attribute name, and $\odot \notin \{=, \neq\}$;

(3) τ_1 is of the form $f(t_1, \ldots, t_n) = y$, f is an attribute name, and one of $t_1, \ldots t_n$ contains a term $g(q_1, \ldots, q_n)$, where g is an attribute name;

(4) τ_1 is of the form $f(t_1, \ldots, t_n) = y$, f is an attribute declared as:

$$f : s_1, \ldots, s_m \rightarrow s_{m+1}$$

where $n \neq m$;

(5) τ_1 is of the form $f(t_1, \ldots, t_n) = y$, f is an attribute declared as:

$$f : s_1, \ldots, s_n \rightarrow s_{n+1}$$

and for some $i \in 1...n$, t_i is a ground term not belonging to s_i;

(6) τ_1 is of the form $f(t_1, \ldots, t_n)$, f is an attribute declared as:

$$f : s_1, \ldots, s_n \rightarrow s_{n+1}$$

and τ_2 is a ground term such that $\tau_2 \notin s_{n+1}$;

(7) at least one of terms τ_1 or τ_2 contains a term $f(t_1, \ldots, t_n)$, where f is neither a record name nor an attribute term of the program;

(8) for each atom $do(a, y)$ occurring in the program, if y doesn't contain variables, then $y \in range(a)$; similarly for $obs(a, y)$.

Pr-atom $pr(a(\vec{t}) = y \mid B) = v$ has to satisfy all the constraints for the rule $a(\vec{t}) = y \leftarrow B$, and, additionally, a cannot be formed by obs or do.

5.4 Errors in Query

The literal in program's query has to satisfy the same requirements as literal in the body of program rules, as defined in Section 5.3.

5.5 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. In addition, we fill flag empty sorts. There should be a user option to disable both warnings.

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] Marcello Balduccini and Michael Gelfond. Logic programs with consistency-restoring rules. *International Symposium on Logical Formalization of Commonsense Reasoning*, 02 2003.

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

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