RankPL: A Qualitative Probabilistic Programming Language

Tjitze Rienstra

University of Luxembourg
Esch-sur-Alzette, Luxembourg
tjitze@gmail.com

July 13, 2017

Overview

Probabilistic Programming

"Probabilistic programs are usual functional or imperative programming languages with two added constructs:

1. the ability to draw values at random from distributions, and
2. the ability to condition values of variables in a program via observations."[1]

Probabilistic programming ...

- provides a universal modelling language for Bayesian inference.
- untangles the modelling task (writing the program) and inference task (executing the program).
- simplifies Bayesian inference from a knowledge engineering perspective.

[1] Andrew D. Gordon et al. "Probabilistic programming". In: *Proceedings of FOSE 2014*. 2014, pp. 167–181.

Probabilistic Programming

Instead of a deterministic outcome, a probabilistic program generates a probability distribution over outcomes. The observe statement is used to express conditional inference.

Example

Program:

```
bool c1, c2;
c1 = Bernoulli(0.5);
c2 = Bernoulli(0.5);
observe(c1 || c2);
return(c1, c2);
```

Output:

```
(true,false)    (0.33)
(false,true)    (0.33)
(true,true)     (0.33)
```

Alternatives to the Bayesian approach

Although the Bayesian approach seems to be the most successful approach to modelling uncertainty, there are many alternatives.

- Dempster Shafer
- Imprecise Probability
- Possibility Theory
- Ranking Theory

Ranking Theory

- A ranking function measures the degree of surprise that some event occurs. Formally, a ranking function K is defined as

$$\kappa : \Omega \rightarrow \mathbb{N} \cup \{\infty\},$$

  such that $\kappa(w) = 0$ for at least one $w \in \Omega$.

- Extended to propositions:

$$\text{for all } A \subseteq \Omega, \kappa(A) = min(\{\kappa(w) \mid w \in A\}).$$

- A is *believed with firmness* x (for $x > 0$) iff $\kappa(\overline{A}) = x$.
- Conditioning: the rank of A conditional on B (if $B \neq \emptyset$) is defined as

$$\kappa(A \mid B) = \kappa(A \cap B) - \kappa(B).$$

Ranking Theory vs. Probability Theory

Ranking Theory . . .

- models everyday, categorical notion of belief:

$$Bel(\kappa) = \{w \in \Omega \mid \kappa(w) = 0\}.$$

- permits reasoning about events that "normally" or "surprisingly" (to some degree) occur, without having to specify probabilities.
- still supports many powerful features of the Bayesian approach (such as revision through conditioning).
- is computationally easier to handle than probability theory.

Ranked Programming?

Questions:

- Can we develop a *ranked programming language*?
- What should such a language look like?
- What can we do with it?

Goals:

- Design a simple imperative programming language (variable assignment, if-else, while-do) with statements for ranked choice and ranking-theoretic observation.
- Formally specify the language with a denotational semantics (see paper).
- Develop an efficient implementation faithful to the semantics (see `github.com/tjitze/RankPL`).

Overview

Syntax

Definition

```
e: (numerical expressions)
    n | x | e1 + e2 | e1 - e2 | e1 * e2 | e1 / e2;
b: (boolean expressions)
    !b | b1 or b2 | b1 and b2 | e1 = e2 | e1 < e2;
s: (statements)
    {s1; s1} |
    x := e |
    if b s1 else s2 |
    while b do s |
    skip |
    normally (e) s1 exceptionally s2 |
    observe b;
```

Syntax

Definition

```
e: (numerical expressions)
    n | x | e1 + e2 | e1 - e2 | e1 * e2 | e1 / e2;
b: (boolean expressions)
    !b | b1 or b2 | b1 and b2 | e1 = e2 | e1 < e2;
s: (statements)
    {s1; s1} |
    x := e |
    if b s1 else s2 |
    while b do s |
    skip |
    normally (e) s1 exceptionally s2 |
    observe b;
```

Ranked Choice

Expresses a choice between alternatives. Basic form is as follows.

```
normally (e) A exceptionally B;
```

This statement states that:

- Normally, `A` is executed.
- If `A` is not executed (surprising to degree `e`) then `B` is executed.

Syntactic shortcuts:

```
     normally (e) A  =  normally (e) A exceptionally skip
exceptionally (e) A  =  normally (e) skip exceptionally A
        either A or B  =  normally (0) A exceptionally B
       x := a <<e>> b  =  normally (e) x := a exceptionally x := b
```

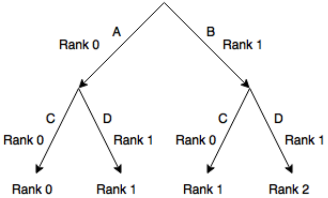
Ranked Choice

Combining ranked choice statements:

```
normally (1) A exceptionally B;
normally (1) C exceptionally D;
```

Four alternative program flows:

- A-C (ranked 0)
- A-D (ranked 1)
- B-C (ranked 1)
- B-D (ranked 2)

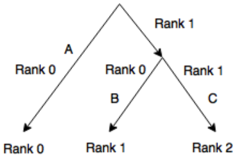


Always executed least-surprising-first!

Ranked Choice

Combining ranked choice statements:

```
normally (1) {
        A;
} exceptionally {
        normally (1) {
                B;
        } exceptionally {
                C;
        }
}
```



This statement states that:

- Normally, A executed.
- If A is not executed (surprising to degree 1) then, normally, B is executed.
- If neither A nor B is executed (surprising to degree 2) then C is executed.

An example: coin tossing

Alice is tossing an *extremely* biased coin. It normally lands heads, and only surprisingly (to degree 1) lands tails. She tosses the coin three times. How many times will she throw tails?

```
flip1 := 0 <<1>> 1;
flip2 := 0 <<1>> 1;
flip3 := 0 <<1>> 1;
return flip1 + flip2 + flip3;
```

Result:

Rank	Outcome
0	0
1	1
2	2
3	3

Observation

The statement

```
observe b
```

revises the ranking over alternatives due to observing or learning that the condition b is true. It does two things:

- Block execution of alternatives not satisfying ‘b‘.
- Uniformly shift down the ranks of the remaining alternatives to zero.

An example

Suppose we observe that Alice throws tails at least once. How often does Alice throw tails?

```
flip1 := 0 <<1>> 1;
flip2 := 0 <<1>> 1;
flip3 := 0 <<1>> 1;
observe flip1 + flip2 + flip3 >= 1;
return flip1 + flip2 + flip3;
```

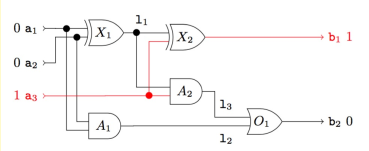
Result:

Rank	Outcome
0	1
1	2
2	3

Circuit Diagnosis

Program:

```
# Set input variables
a1 := FALSE;
a2 := FALSE;
a3 := TRUE;

# Set state of gates (TRUE is functioning, FALSE is broken)
x1_broken := FALSE <<1>> TRUE;
x2_broken := FALSE <<1>> TRUE;
a1_broken := FALSE <<1>> TRUE;
a2_broken := FALSE <<1>> TRUE;
o1_broken := FALSE <<1>> TRUE;

# Circuit logic
if (x1_broken) then l1 := FALSE <<0>> TRUE else l1 := (a1 ^ a2);
if (a1_broken) then l2 := FALSE <<0>> TRUE else l2 := (a1 & a2);
if (a2_broken) then l3 := FALSE <<0>> TRUE else l3 := (l1 & a3);
if (x2_broken) then b1 := FALSE <<0>> TRUE else b1 := (l1 ^ a3);
if (o1_broken) then b2 := FALSE <<0>> TRUE else b2 := (l3 | l2);

# Observe output
observe !b1 & b2;

# Return state of gates
return "X1: " + x1_broken + " X2: " + x2_broken + " A1: " + a1_broken +  " A2: " + a2_broken + " O1: " + o1_broken;
```



Output:

```
Rank    Outcome
  0     X1: true   X2: false  A1: false  A2: false  O1: false
  1     X1: false  X2: true   A1: false  A2: true   O1: false
  1     X1: false  X2: true   A1: false  A2: false  O1: true
 ...
  2     X1: true   X2: true   A1: true   A2: false  O1: false
  2     X1: true   X2: false  A1: true   A2: true   O1: false
 ...
```

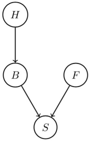
Ranking Networks

Program:

```
h := FALSE <<15>> TRUE;

if (h) {
        b := FALSE <<4>> TRUE;
} else {
        b := TRUE <<8>> FALSE;
};

f := TRUE <<10>> FALSE;

if (b && f) {
        s := TRUE <<3>> FALSE;
} else if (b && !f) {
        s := FALSE <<13>> TRUE;
} else if (!b && f) {
        s := FALSE <<11>> TRUE;
} else if (!b && !f) {
        s := FALSE <<27>> TRUE;
};

return "h: " + h + " b: " + b + " f: " + f + " s: " + s;
```

H	$\kappa_H(H)$
h	15
$\overline{h}$	0

$B\|H$	$\kappa_B(B\|H)$
$b\|h$	4
$\overline{b}\|h$	0
$b\|\overline{h}$	0
$\overline{b}\|\overline{h}$	8



F	$\kappa_F(F)$
f	0
$\overline{f}$	10

$S\|BF$	$\kappa_S(S\|BF)$	BFS	$\kappa_S(S\|BF)$
$s\|b\,f$	0	$\overline{s}\|b\,f$	3
$s\|b\,\overline{f}$	13	$\overline{s}\|b\,\overline{f}$	0
$s\|\overline{b}\,f$	11	$\overline{s}\|\overline{b}\,f$	0
$s\|\overline{b}\,\overline{f}$	27	$\overline{s}\|\overline{b}\,\overline{f}$	0

Output:

```
Rank     Outcome
   0     h: false   b: true    f: true    s: true
   3     h: false   b: true    f: true    s: false
   8     h: false   b: false   f: true    s: false
  10     h: false   b: true    f: false   s: false
  15     h: true    b: false   f: true    s: false
  18     h: false   b: false   f: false   s: false
 ...     ...
```

Overview

Iterated Revision in RankPL

Normal conditioning (and thus the `observe` statement) leads to irreversible belief with absolute certainty. *L-conditioning* (also called *evidence-oriented* conditioning) generalizes normal conditioning.

Definition

Let $A \subseteq \Omega$ and let $x \in \mathbb{N}$. The *L-conditioning* of κ on A with parameter x is denoted by $\kappa_{A\uparrow x}$ and is defined as

$$\kappa_{A\uparrow x}(w) = \begin{cases} \kappa(w) - y & \text{if } w \in A \text{, or} \\ \kappa(w) + x - y & \text{if } w \notin A \end{cases}$$

where $y = min(\kappa(A), x)$.

In RankPL implemented by the `observe-l (x) b` statement.

Iterated Revision in RankPL

We receive evidence that Alice threw tails at least once. This evidence strengthens our belief in this fact by five units of rank.

```
flip1 := 0 <<1>> 1;
flip2 := 0 <<1>> 1;
flip3 := 0 <<1>> 1;
observe-l (5) flip1 + flip2 + flip3 >= 1;
return flip1 + flip2 + flip3;
```

Result:

Rank	Outcome
0	1
1	2
2	3
4	0

Iterated Revision in RankPL

We receive *two* (independent) pieces of information strengthening our belief that Alice threw tails at least once:

```
flip1 := 0 <<1>> 1;
flip2 := 0 <<1>> 1;
flip3 := 0 <<1>> 1;
observe-l (5) flip1 + flip2 + flip3 >= 1;
observe-l (5) flip1 + flip2 + flip3 >= 1;
return flip1 + flip2 + flip3;
```

Result:

Rank	Outcome
0	1
1	2
2	3
9	0

Iterated Revision in RankPL

The second observation reverses the effect of the first one:

```
flip1 := 0 <<1>> 1;
flip2 := 0 <<1>> 1;
flip3 := 0 <<1>> 1;
observe-l (5) flip1 + flip2 + flip3 >= 1;
observe-l (5) flip1 + flip2 + flip3 < 1;
return flip1 + flip2 + flip3;
```

Result:

Rank	Outcome
0	0
1	1
2	2
3	3

An example: spelling correction

- Rank words in a dictionary according to how close they are to the input.
- Interpret each input character c_i (at index $i = 1, 2, ...$) as evidence that strengthens our belief that the character at i is *actually* c_i.
- Use L-observation for this:

```
observe-l (1) input_word[i] == potential_match[k];
```

- If mismatch: consider three possibilities (missing, superfluous, incorrect).
- This algorithm only takes about 20 lines of code.

Overview

Open issues:

- More applications (e.g. planning for minimal risk, game strategies, agent models...).
- Can we capture default rules that have ranking-based semantics (System Z)?

More information:

- See the paper for the denotational semantics of RankPL.
- Download RankPL at `github.com/tjitze/RankPL`.

Thanks for your attention