Interactive Bayesian Probabilistic Programming and Debugging

Javier Burroni, Arjun Guha, David Jensen

College of Information and Computer Sciences • University of Massachusetts Amherst

1. Motivation

Current Bayesian Probabilistic programming languages presents limitations to interactivity.
New queries require re-execution of the entire program.
✦ Not efficient as neither data nor generative model were changed.

2. Approach

Perform backward inference only once.

The result of inference is a posterior distribution over **traces** — FOL structures.

New keyword: `inspect(expr) -> value`. Evaluate the expression in a **trace**.

`query(expr)` implemented as application of inspect over a sample of the posterior distribution of **traces**.

3. Dynamically querying

No need to know the queries before running inference.
✦ Allows interactively querying of the posterior distribution.

```
 0: type City;
 1: type PrepLevel;
 2: type DamageLevel;

 3: random City First ~ UniformChoice({c for City c});
 4: random City NotFirst ~ UniformChoice({c for City c: c !=
First});
 5: random PrepLevel Prep(City c) ~
 6:   if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
 7:   else case Damage(First) in
 8:           {Severe -> Categorical({High -> 0.9, Low -> 0.1}),
 9:             Mild -> Categorical({High -> 0.1, Low -> 0.9})};
10: random DamageLevel Damage(City c) ~
11:    case Prep(c) in {
12:             High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
13:             Low -> Categorical({Severe -> 0.8, Mild -> 0.2})};

14: distinct City A, B;
15: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;

16: obs Damage(First) = Severe;

17: query Damage(NotFirst);
```

```
query Damage(A)

Severe    0.641092
Mild      0.358908
```

```
query if Damage(A) == Severe then Prep(A) else Prep(B);
Low     0.741546
High    0.258454
```

4. Inspect one world

inspect(expr) accepts any valid BLOG expression.

```
inspect Damage(A)
Svalue: Severe
```

```
inspect if Damage(A) == Severe then Prep(A) else Prep(B);
value: Low
```

5. Step-by-step debugging

`inspect(expr)` accepts any valid BLOG expression.
✦ The generative model is made of BLOG expressions.
✦ step-by-step debugging can be implemented by recursively inspecting the generative process.

```
debugger.step()
Entering: obs Damage(First) = Severe

debugger.step()
Entering: Damage(First)

debugger.step()
Entering: First

debugger.step()
Entering: UniformChoice({c for City c})

debugger.runToLine(11)
Entering: case Prep(c) in
{
    High -> Categorical({
                         Severe -> 0.2,
                         Mild -> 0.8}),
    Low -> Categorical({
                         Severe -> 0.8,
                         Mild -> 0.2})}
```

```
14: distinct City A, B;
15: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;

16: obs Damage(First) = Severe;      <- Entry point

17: query Damage(NotFirst);
```

6. Local variables inspection

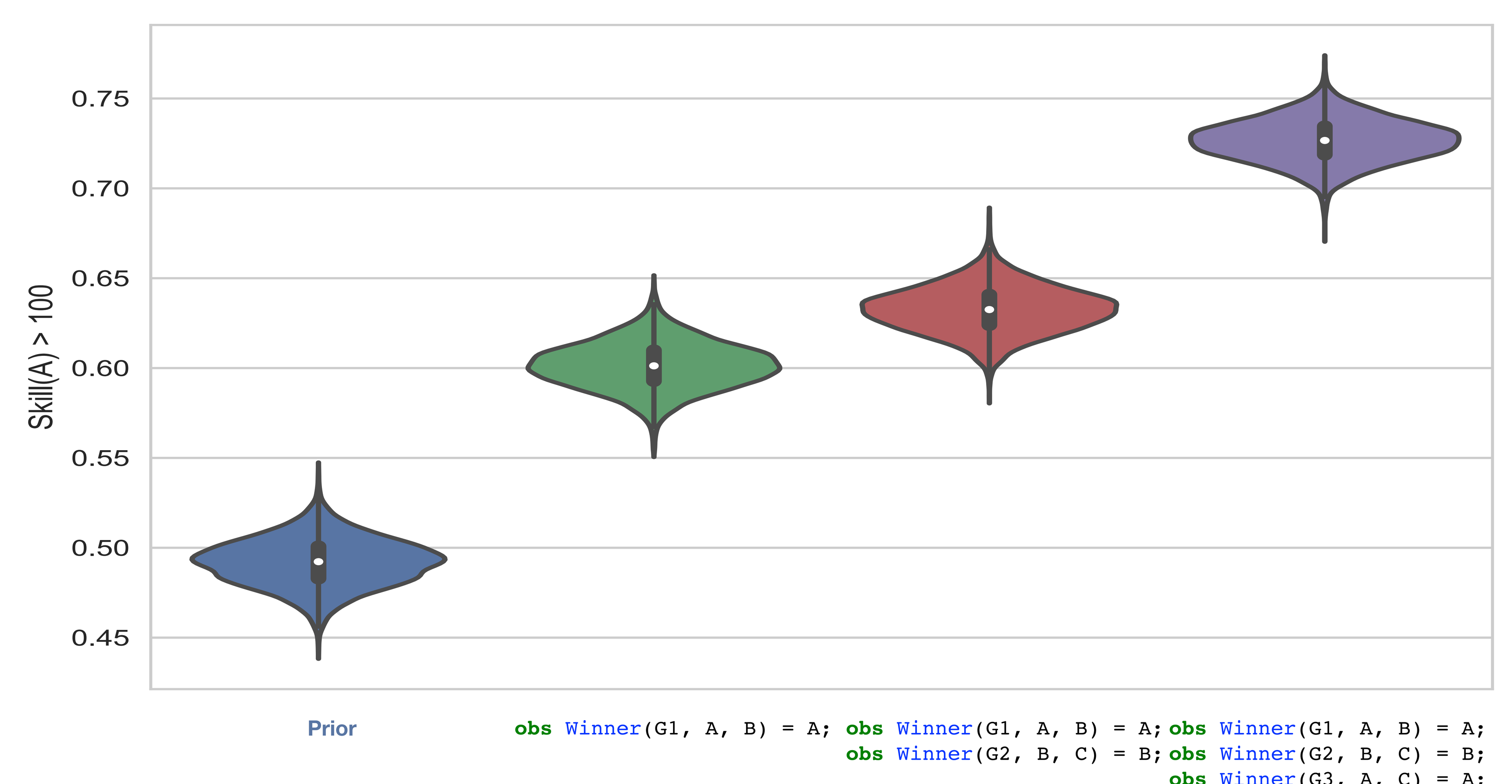
Arguments to function can be inspected.
It is possible to switch to a **trace** where a given predicate holds.

```
debugger.inspect('c');
Inspect: c
value: B

debugger.inspect('Prep(c)');
Inspect: Prep(c)
value: Low

debugger.switchTo('Prep(B) == High')
switching to trace compatible: #175
Entering: obs Damage(First) = Severe
```

7. Impact of Data

Evaluate impact of information for any `query.`
✦ Compute the posterior with different subsets of observations, and evaluate the expression in each sample of the posterior.
✦ TrueSkill [2] example.

```
random Real Skill(Player p) ~ Gaussian(100.0, 10.0);
random Real Performance(Player p, Game g) ~
         Gaussian(Skill(p), 15.0);
random Player Winner(Game g, Player p1, Player p2) ~
        if (Performance(p1, g) > Performance(p2, g))
         then p1
        else p2;
```



8. Future work

Implement this debugger in other PPLs using **addresses** and **traces.**

[1] Milch, Brian Christopher, and Stuart J. Russell. Probabilistic models with unknown objects. Diss. University of California, Berkeley, 2006.
[2] Gordon, A. D., Henzinger, T. A., Nori, A. V. & Rajamani, S. K. Probabilistic programming. in Proceedings of the on Future of Software Engineering 167–181 (ACM, 2014).