Translating Stan into Pyro.

Javier Burroni, Guillaume Baudart, Louis Mandel,
Martin Hirzel, Avraham Shinnar

IBM PL Day Dec. 10 2018

Introduction

- Probabilistic Programming Langauges are languages that allow to express probabilistic models: models of data and latent variables.
- Provided with data, inference can be done to learn properties of latent variables.

Families of PPL

- BLOG, Church, Anglican, WebPPL, Venture
    - Generative probabilistic programming languages
- Factory
    - based on factor graphs
- BUGS, Jags, Stan
    - declarative (?)
- Pyro, Edward
    - Deep Probabilistic Programming

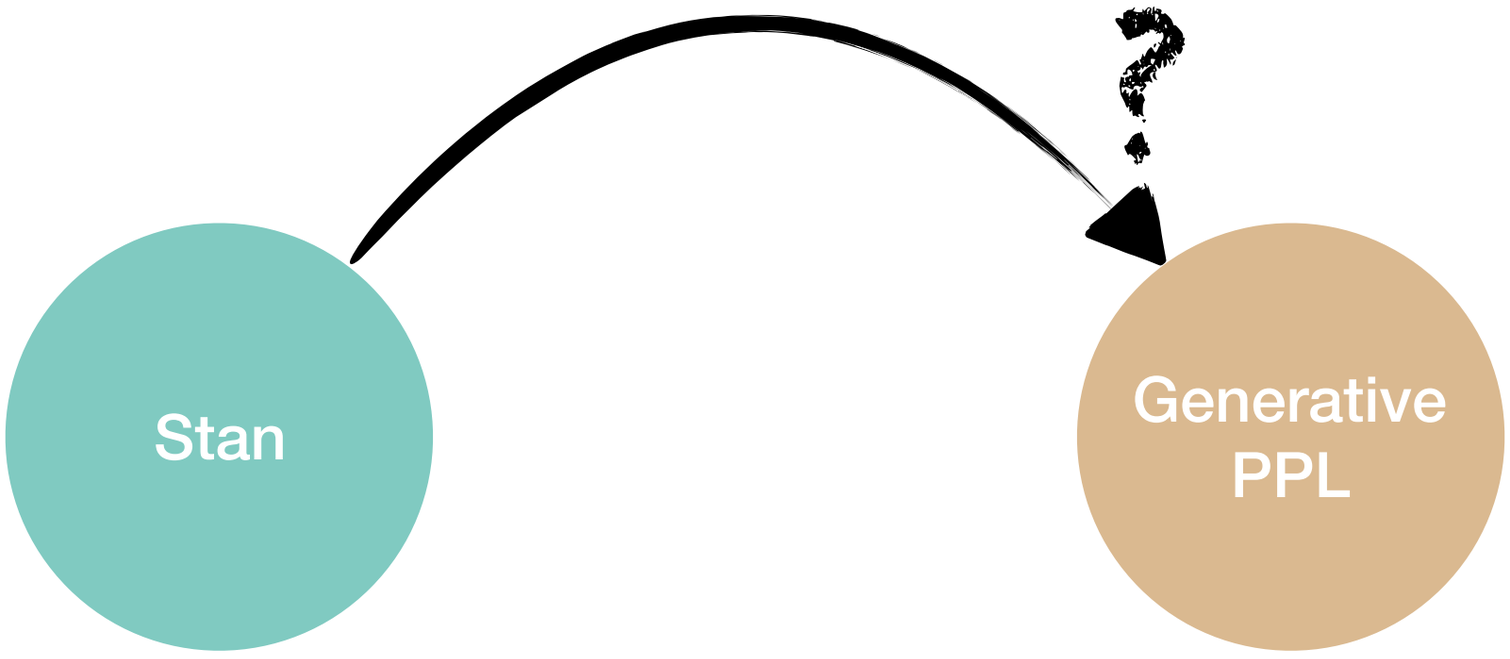
Translation

- Translating between different families allows us to understand the relative differences in expressive power.

- But it might not be possible.

Translation

- Translating between different families allows us to understand the relative differences in expressive power.
- But it might not be possible.
- We want to translate Stan code into Pyro



Stan

- One of the most used PPLs
- Feels like a regular language, with special elements:
  - blocks,
  - pseudo variable (target)
  - operator ~ (tilde)

B. Carpenter *et al*., “Stan: A probabilistic programming language,” *Journal of statistical software*, vol. 76, no. 1, 2017.

Stan: blocks

- data block: inputs to the model

```
data
{
  int<lower=0> N;
  real x[N];
}
```

Stan: blocks

- data block: inputs to the model

```
data
{
  int<lower=0> N;
  real x[N];
}
```

- parameters block: declaration of latent random variables

```
parameters
{
  real mu;
  real<lower=0> sigma;
}
```

Stan: blocks

- data block: inputs to the model

```
data
{
  int<lower=0> N;
  real x[N];
}
```

- parameters block: declaration of latent random variables

```
parameters
{
  real mu;
  real<lower=0> sigma;
}
```

- model block: where data meets parameters

```
model
{
  // Jeffreys prior
  target += log(1/sigma);
  x ~ normal(mu, sigma);
}
```

- the calculation expressed in the model computes the value of a (differentiable) log density function (given data and parameters).

Stan: target

- The objective of the model block is to define a log density.
- target is the variable where the log density is partially accumulated

```
model
{
  // Normal(0,1)
  target += -0.5*x*x;
}
```

Stan: ~

- Instead of directly modifying the target, when using known distributions, Stan offers a syntactic sugar.

```
model
{
  x ~ normal(0, 1);
}
```

- This is easier to read: “x follows a normal zero one"

Stan: ~

- Instead of directly modifying the target, when using known distributions, Stan offers a syntactic sugar.

```
model
{
  x ~ normal(0, 1);
}
```

- This is easier to read: “x follows a normal zero one"

- sometimes this interpretation is **incorrect**:

```
model
{
  y ~ normal(0, 1);
  y ~ normal(0, 1);
}
```

Now y has a different distribution than x!

~ is not an assignment!

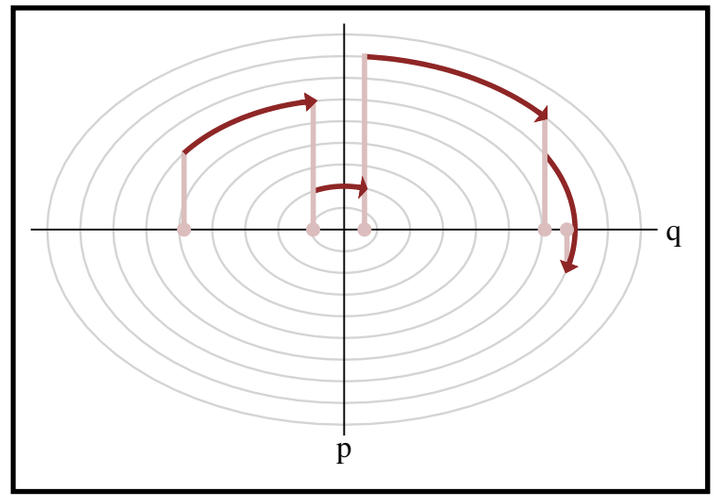
$$p(y) \propto e^{-\frac{1}{2}y^2} e^{-\frac{1}{2}y^2} = e^{-\frac{1}{2}\frac{2}{1}y^2}$$

Stan: inference

- With the representation of the model and data, Stan builds a log density function.

- This function will be proportional to the posterior

- Using Automatic differentiation, we can run Hamiltonian Monte Carlo to explore the posterior distribution



HMC trajectories of distribution q with auxiliary distribution p

M. Betancourt, "A Conceptual Introduction to Hamiltonian Monte Carlo," *arXiv:1701.02434 [stat]*, Jan. 2017.

Pyro

- Deep Probabilistic Programming
- built on top of Python and PyTorch
- supports Neural Networks and many other things outside of the scope of this talk

E. Bingham *et al*., “Pyro: Deep Universal Probabilistic Programming,” *Journal of Machine Learning Research*, 2018.

Pyro

- Deep Probabilistic Programming
- It’s generative
    - It has a way to generate values from a distribution

```
x = pyro.sample("x",
                distribution(...))
```

- and a way to incorporate observations

```
pyro.sample("x",
                distribution(...),
                obs=x)
```

Pyro: Inference

- Pyro supports many inference methods (Markov Chain Monte Carlo, Variational Inference, et cetera).

- On way to understand inference in generative models is through "Likelihood-weighted sampling"

$$\underbrace{P(Z|X)}_{\text{posterior}} \propto \underbrace{P(X|Z)}_{\text{likelihood}} \underbrace{P(Z)}_{\text{prior}}$$

- Sample values from the prior and weight them using the likelihood

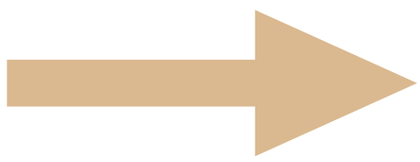
Stan to Pyro

- They look like two different species
- But for simple examples a direct translation can be done

```
data
{
  int<lower=0> N;
  real x[N];
}

parameters
{
  real mu;
}

model
{
  mu ~ normal(0, 1);
  x ~ normal(mu, 1);
}
```

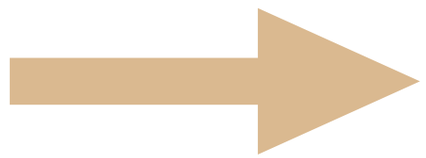


```
def model(x):
  mu = pyro.sample("mu",
                   dist.Normal(0, 1))
  x = pyro.sample("x",
                  dist.Normal(mu, 1),
                  obs=x)
```

Stan to Pyro

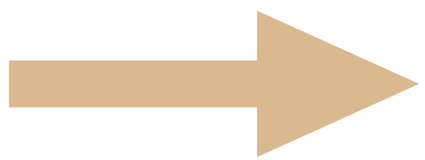
- Naïve translation rules:

```
parameters
{
  real mu;
}
model
{
  ...
  mu ~ distribution(...);
}
```



```
pyro.sample("mu", distribution(…))
```

```
data
{
  real x;
}
model
{
  ...
  x ~ distribution(...);
}
```



```
pyro.sample("x", distribution(…),
            obs=x)
```

Challenges of this translation

- Naïve translation is **incomplete**.

Challenges of this translation

- Naïve translation is **incomplete**.
- In Stan, random variables can be used without explicitly declaring a prior on them

```
data
{
  int<lower=0> N;
  real x[N];
}
parameters
{
  real mu;
}
model
{
  x ~ normal(mu, 1);
}
```

$$P(\mathbf{mu} \mid x) \propto e^{-\frac{1}{2}\frac{N}{1}(\mathbf{mu}-\bar{x})}$$

Challenges of this translation

- Naïve translation is **incomplete**.

- In Stan, random variables can be used without explicitly declaring a prior on them

- `exp ~ distribution();`, where `exp` is any arbitrary expression is a valid Stan statement

```
parameters {
  real<lower=0> beta;
}
// ...
model {
  log(beta) ~ normal(mu, sigma);
}
```

Stan Development Team. 2018. Stan Modeling Language Users Guide and Reference Manual, Version 2.18.0. http://mc-stan.org

Challenges of this translation

- Naïve translation is **incomplete**.

- In Stan, random variables can be used without explicitly declaring a prior on them

- `exp ~ distribution();`, where `exp` is any arbitrary expression is a valid Stan statement

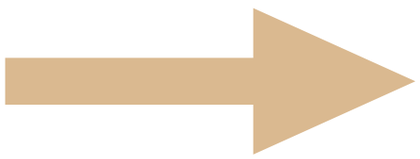
- Stan provides the ability to directly modify target

```
model
{
  // Jeffreys prior
  // for sigma
  target += log(1/sigma);
  x ~ normal(mu, sigma);
}
```

Compilation scheme

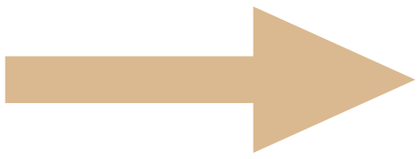
- We present the following **complete** compilation scheme:
- each parameter (random variable) is sampled “uniformly" from the reals

```
exp ~ distribution(...);
```



```
pyro.sample(uid,
            distribution(...),
            obs=exp)
```

```
target += exp;
```



```
pyro.sample(uid,
            dist.Exponential(1),
            obs=-exp)
```

Compilation scheme

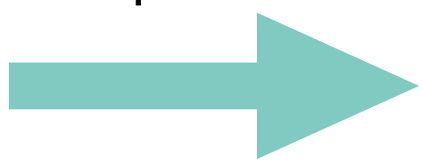
- This scheme solves the presented issues
- Uses an improper prior *by default*

Compilation scheme

- This scheme solves the presented issues
- Uses an improper prior *by default*

```
pyro.sample(uid,
            distribution(...),
            obs=exp)
```

exp can be any arbitrary expression



```
exp ~ distribution(...);
```

Compilation scheme

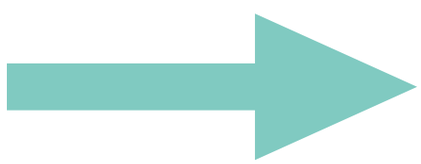
- This scheme solves the presented issues
- Uses an improper prior *by default*

```
pyro.sample(uid,
            distribution(...),
            obs=exp)
```

exp can be any arbitrary expression

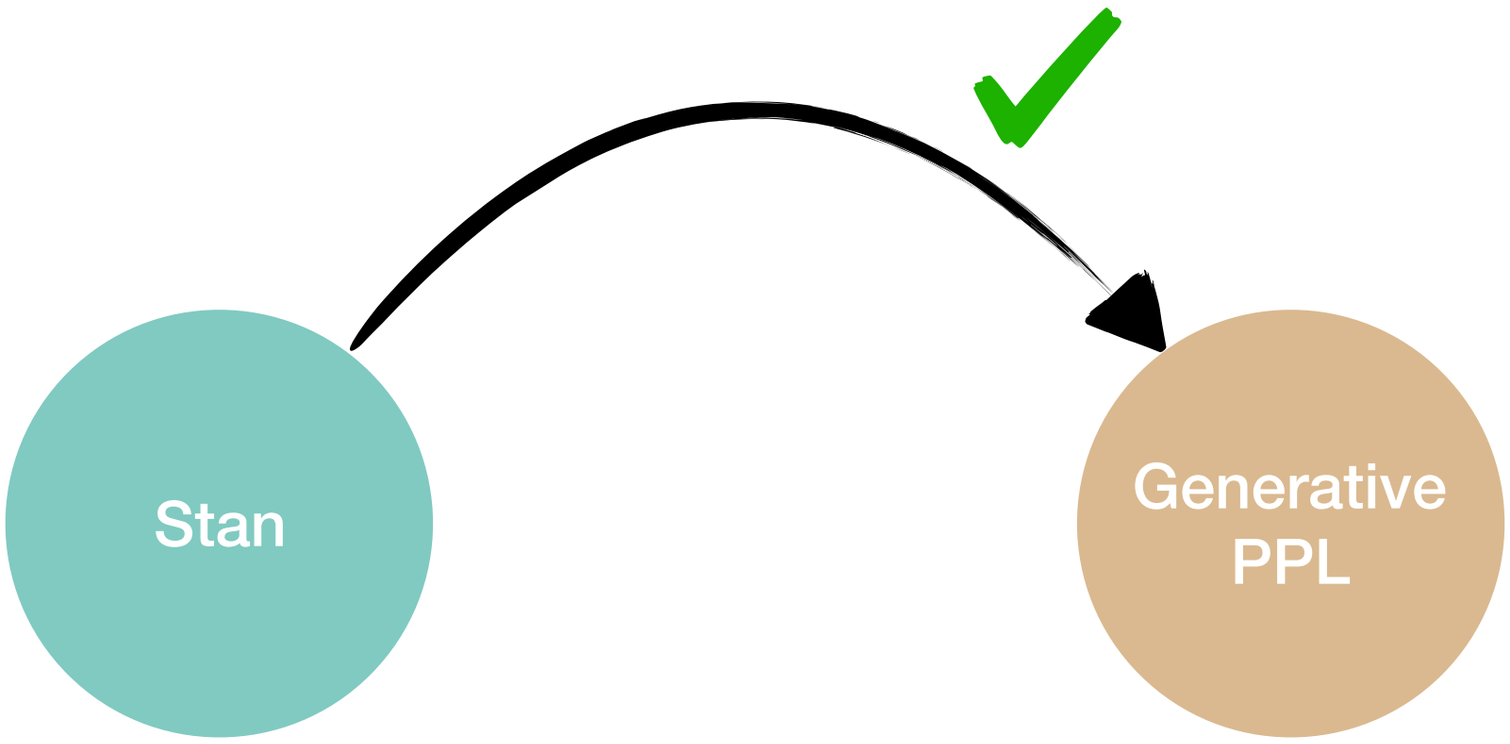
```
exp ~ distribution(...);
```

```
pyro.sample(uid,
            dist.Exponential(1),
            obs=-exp)
```

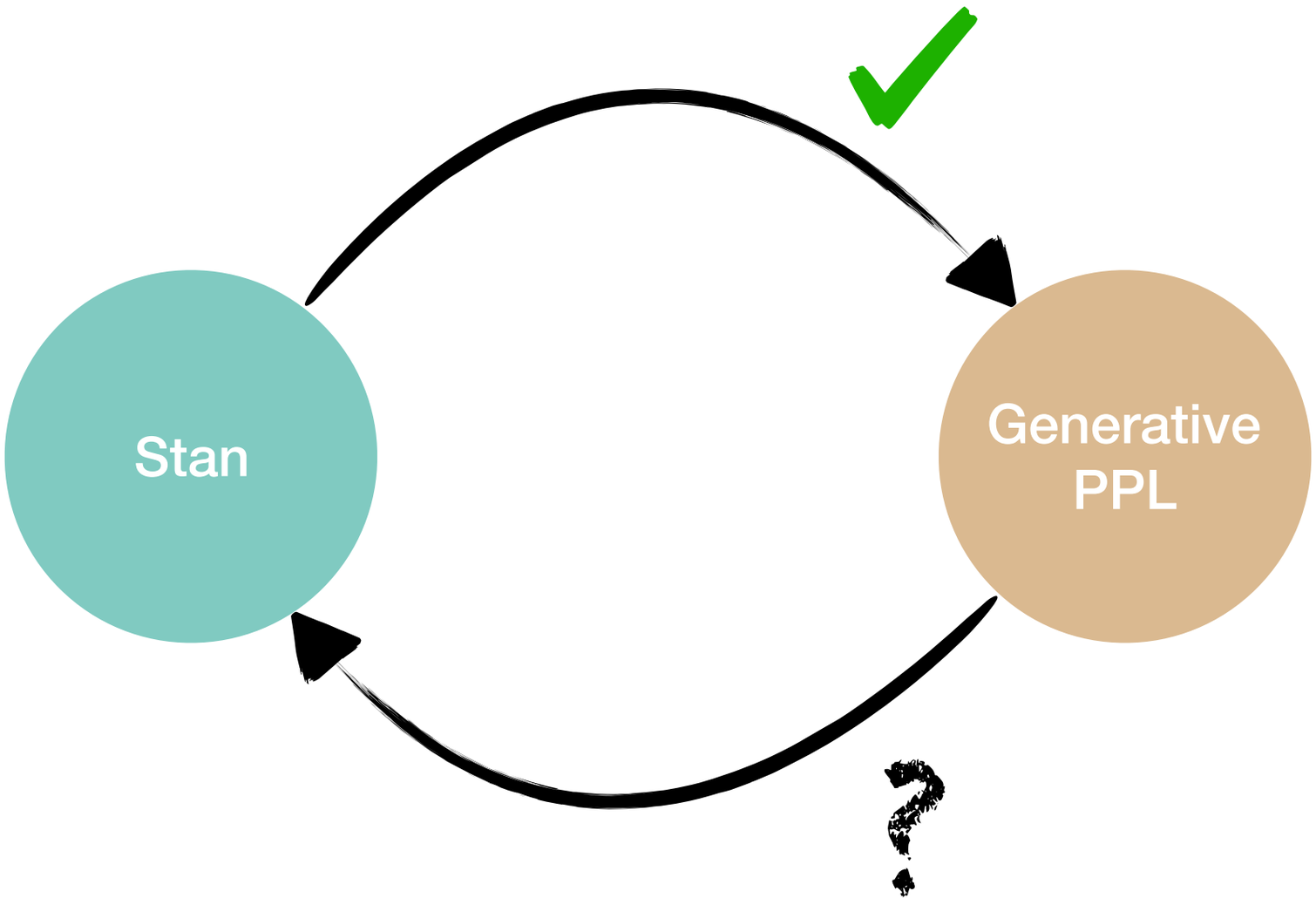


target += $\log(1e^{-1(\textbf{-exp})}) = \textbf{exp}$

Conclusion



Conclusion



Thank you

questions?

Appendix A
Denotational Semantics of (Featherweight) Stan

$$[\![\texttt{x = } e]\!]_M(t) = (t, M[\texttt{x} \leftarrow [\![e]\!]_M])$$

$$[\![\texttt{skip}]\!]_M(t) = (t, M)$$

$$[\![s_1 \texttt{;} s_2]\!]_M(t) = \texttt{let } (t', M') = [\![s_1]\!]_M(t) \texttt{ in } [\![s_2]\!]_{M'}(t')$$

$$[\![\texttt{if (}e\texttt{) } s_1 \texttt{ else } s_2]\!]_M(t) = \begin{cases} [\![\texttt{s}_1]\!]_M(t) & \text{if } [\![\texttt{e}]\!]_M = \texttt{tt} \\ [\![\texttt{s}_2]\!]_M(t) & \text{if } [\![\texttt{e}]\!]_M = \texttt{ff} \end{cases}$$

$$[\![\texttt{while (}e\texttt{) } s]\!]_M(t) = \begin{cases} [\![\texttt{s}; while(\texttt{e})s]\!]_M(t) & \text{if } [\![\texttt{e}]\!]_M = \texttt{tt} \\ [\![\texttt{skip}]\!]_M(t) & \text{if } [\![\texttt{e}]\!]_M = \texttt{ff} \end{cases}$$

$$[\![\texttt{target += } e]\!]_M(t) = (t + [\![e]\!]_M, M)$$

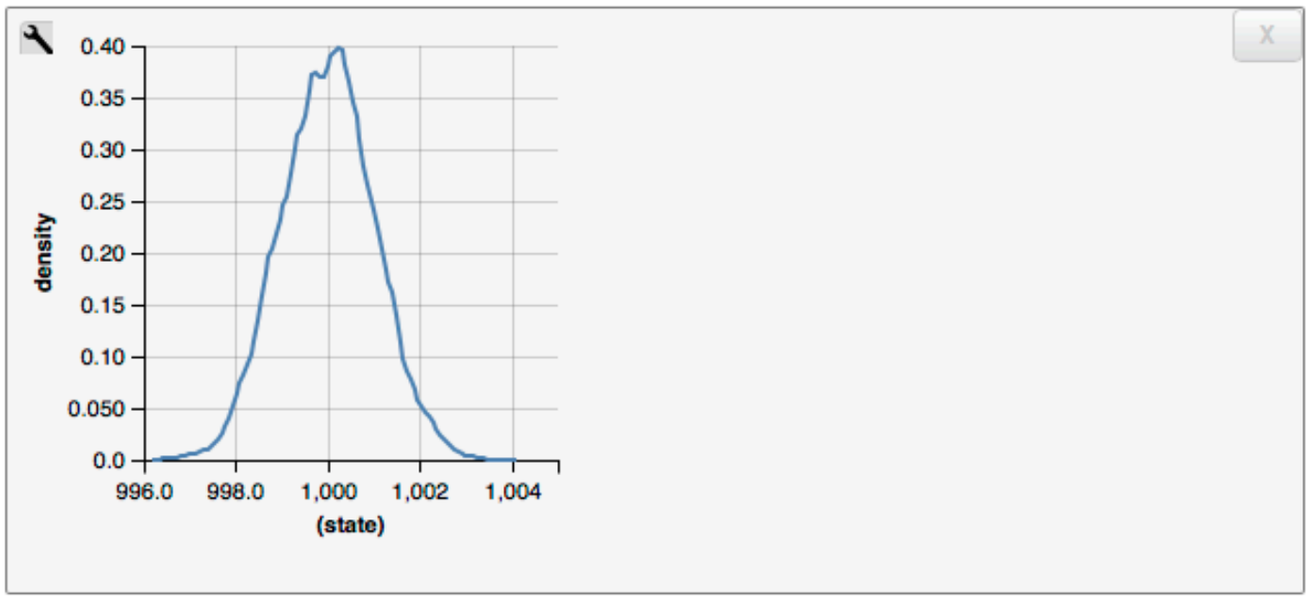
$$[\![e_1 \sim \texttt{f(}e \ldots\texttt{)}]\!]_M(t) = (t + [\![\texttt{f\_lpdf(}e_1 \texttt{ | } e \ldots\texttt{)}]\!]_M, M)$$

Appendix B
Improper Distribution in WebPPL

```
var model = function() {
  var generator = Gaussian({mu:0, sigma:1});
  var x = sample(generator);
  factor(-generator.score(x));   //<- Adjust for the sampled value
  factor(-0.5*(x-1000)*(x-1000));      // <- You can factor any expression of x
  return x;
}

viz.auto(Infer({method: 'MCMC',
                kernel: {HMC: {steps: 10, stepSize: 1}},
                samples: 10000,
                burn: 1000}, model));
```

run



http://webppl.org