# Probabilistic Programming

Frank Wood

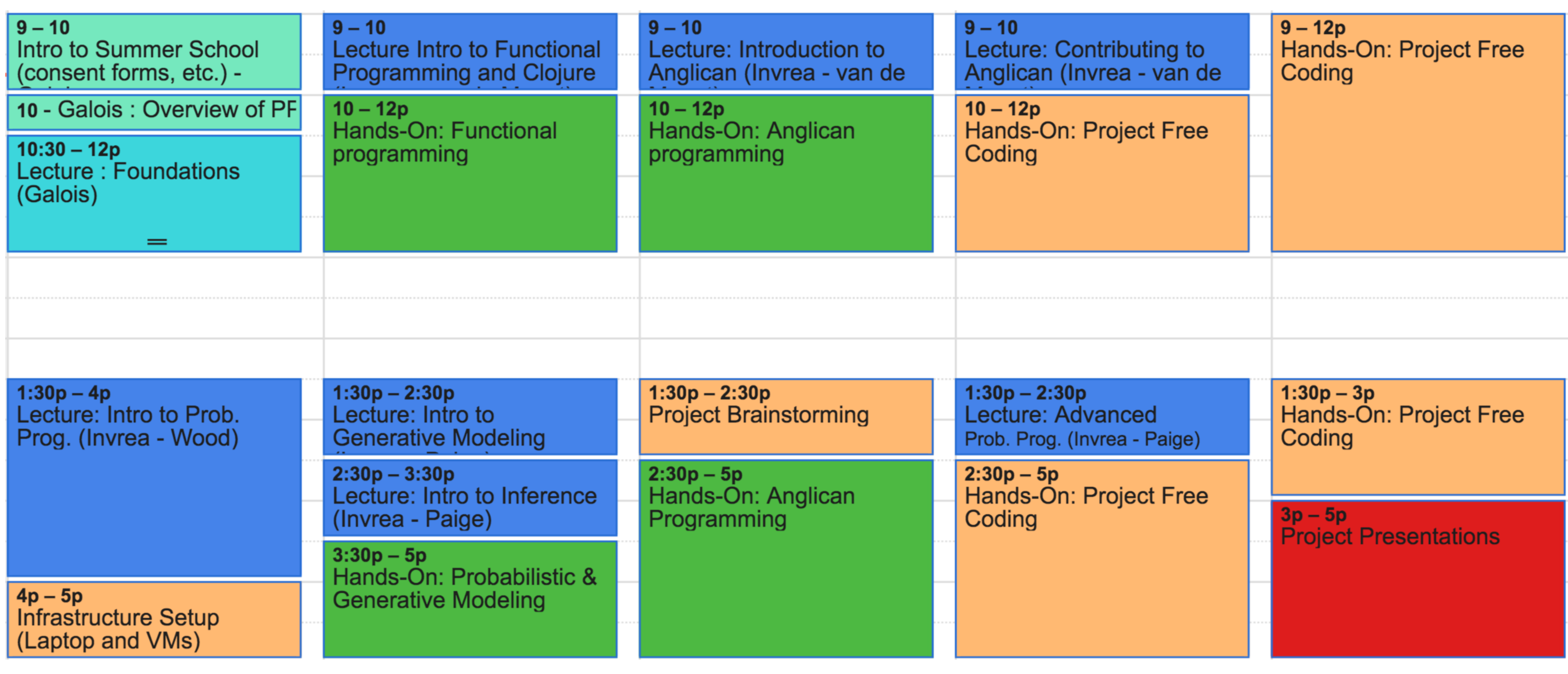
frank@invrea.com
http://www.invrea.com/

fwood@robots.ox.ac.uk
http://www.robots.ox.ac.uk/~fwood

# Objectives For This Week

- Get you to
  - understand and write functional programs (T)
    - know Clojure
  - understand generative modeling (T)
  - understand inference and conditioning (T)
  - understand and write probabilistic programs (W)
    - know Anglican
- Code up a project of your own and share it (Th/F)
  - https://bitbucket.org/probprog/anglican-examples

# Schedule




Public Google Calendar

https://goo.gl/SrNzPZ

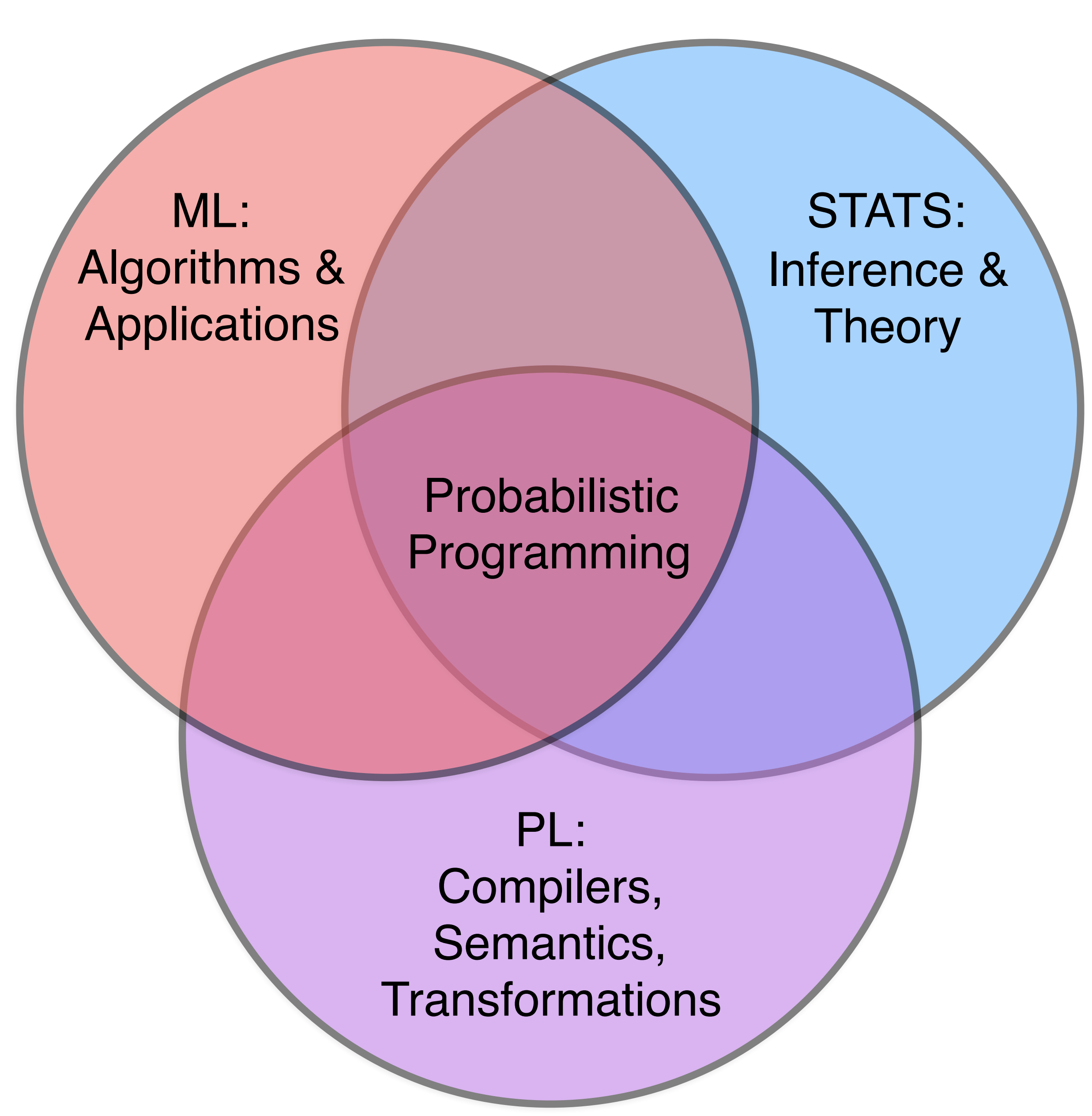
# Objectives For Today

Get you to

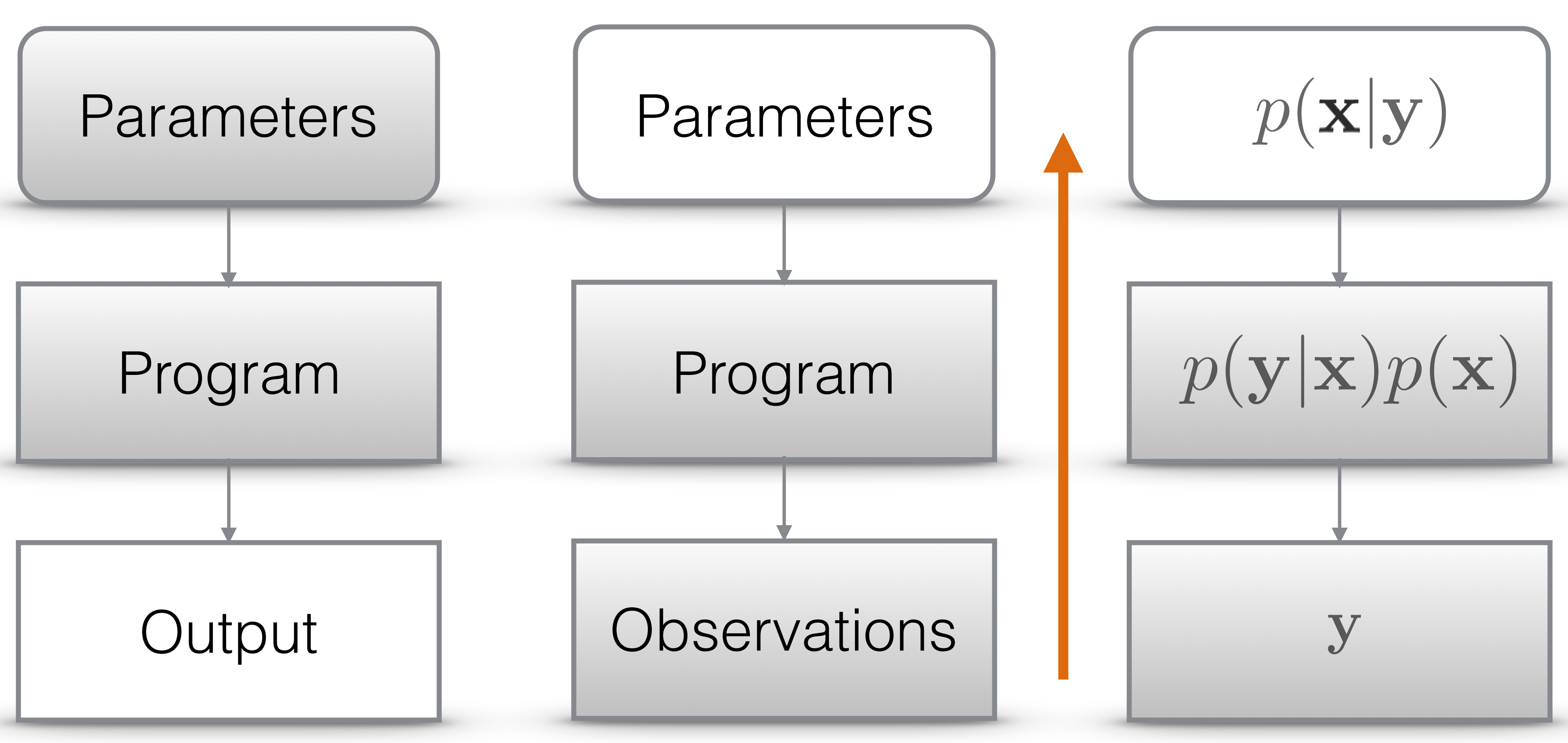
- Know what probabilistic program is and how it's different to a normal program.
- Understand how to write a probabilistic program and have the resources to get started if you want to.
- Understand the literature at a very high level.
- Know one way to roll your own state-of-the-art probabilistic programming system.

# What is probabilistic programming?

# The Field



# Intuition



# A Probabilistic *Program*

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

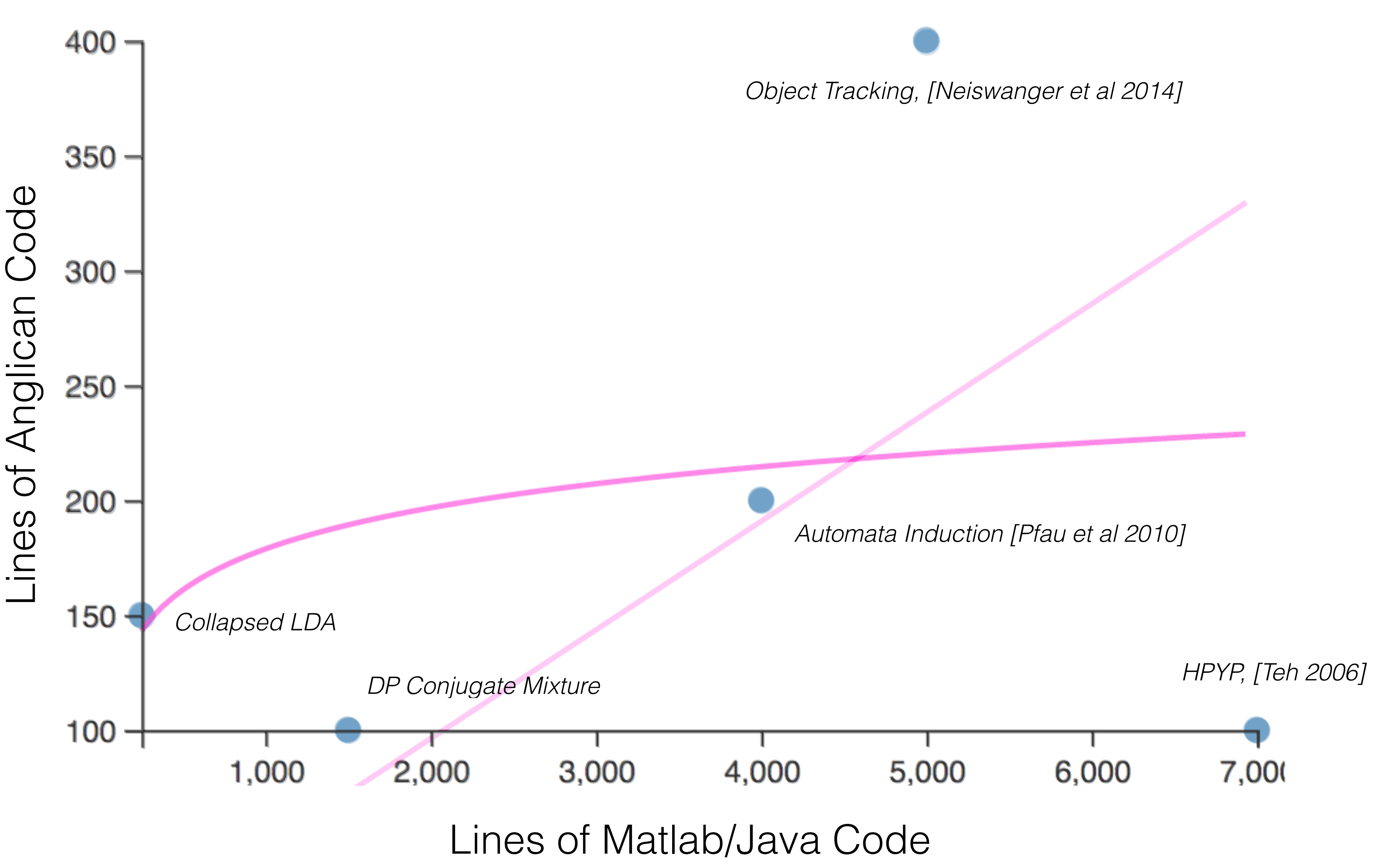
(1) the ability to draw values at random from distributions, and

(2) the ability to condition values of variables in a program via observations."

Gordon, Henzinger, Nori, and Rajamani
"Probabilistic programming." In Proceedings of On The Future of Software Engineering (2014).

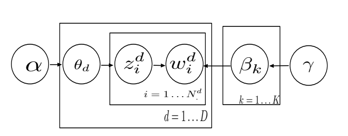
# Goals of the Field

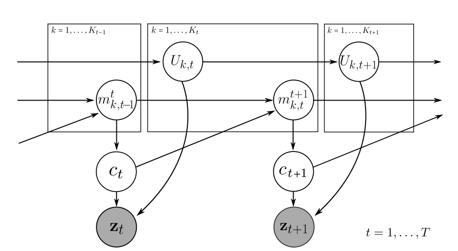
# Increase Productivity

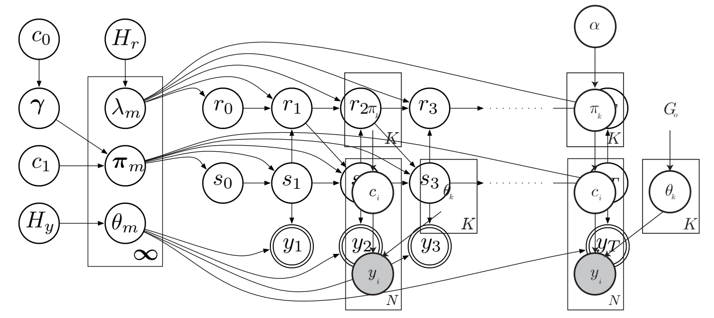


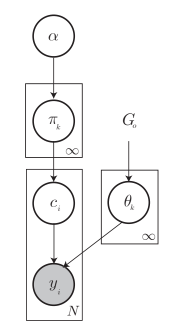
# Commodify Inference

## Models / Simulators

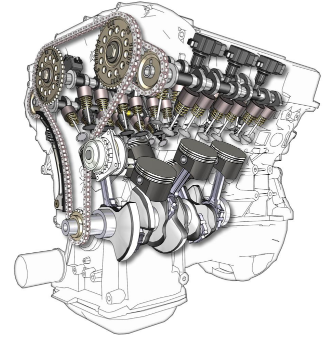








Language Representation / Abstraction Layer



## Inference engines

# History

# Long

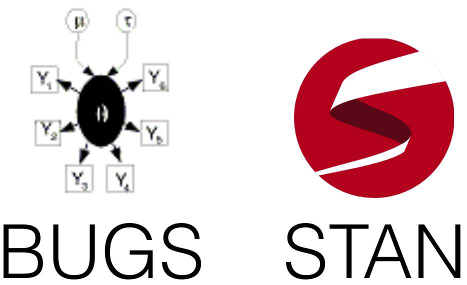
PL | AI | ML | STATS

- 2010
- 2000
- 1990

- PL: Figaro, HANSAI, IBAL, Simula
- AI: ProbLog, Blog, Prism, Prolog
- ML: Probabilistic-ML,Haskell,Scheme,…, webPPL, Probabilistic-C, Venture, Anglican, $\Lambda_o$, Church, Infer.NET, Factorie, KMP
- STATS: LibBi, STAN, JAGS, WinBUGS, BUGS

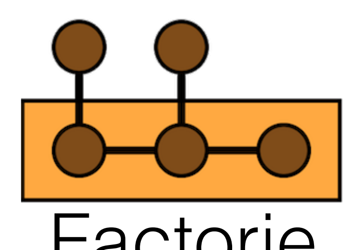
# Success Stories

## Graphical Models



BUGS STAN

## Factor Graphs

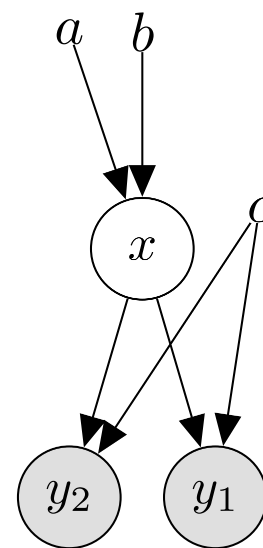


Factorie

Infer.NET

# BUGS

```
model {
    x ~ dnorm(a, 1/b)
    for (i in 1:N) {
        y[i] ~ dnorm(x, 1/c)
    }
}
```




- Language restrictions
    - Bounded loops
    - No branching
- Model class
    - Finite graphical models
- Inference - sampling
    - Gibbs



Spiegelhalter et al. "BUGS: Bayesian inference using Gibbs sampling, Version 0.50." Cambridge 1995.

# STAN : Finite Dimensional Differentiable Distributions

```
parameters {
    real xs[T];
}
model {
    xs[1] ~ normal(0.0, 1.0);
    for (t in 2:T)
        xs[t] ~ normal(a * xs[t - 1], q);
    for (t in 1:T)
        ys[t] ~ normal(xs[t], 1.0);
}
```

→ $\nabla_{\mathbf{x}} \log p(\mathbf{x}, \mathbf{y})$

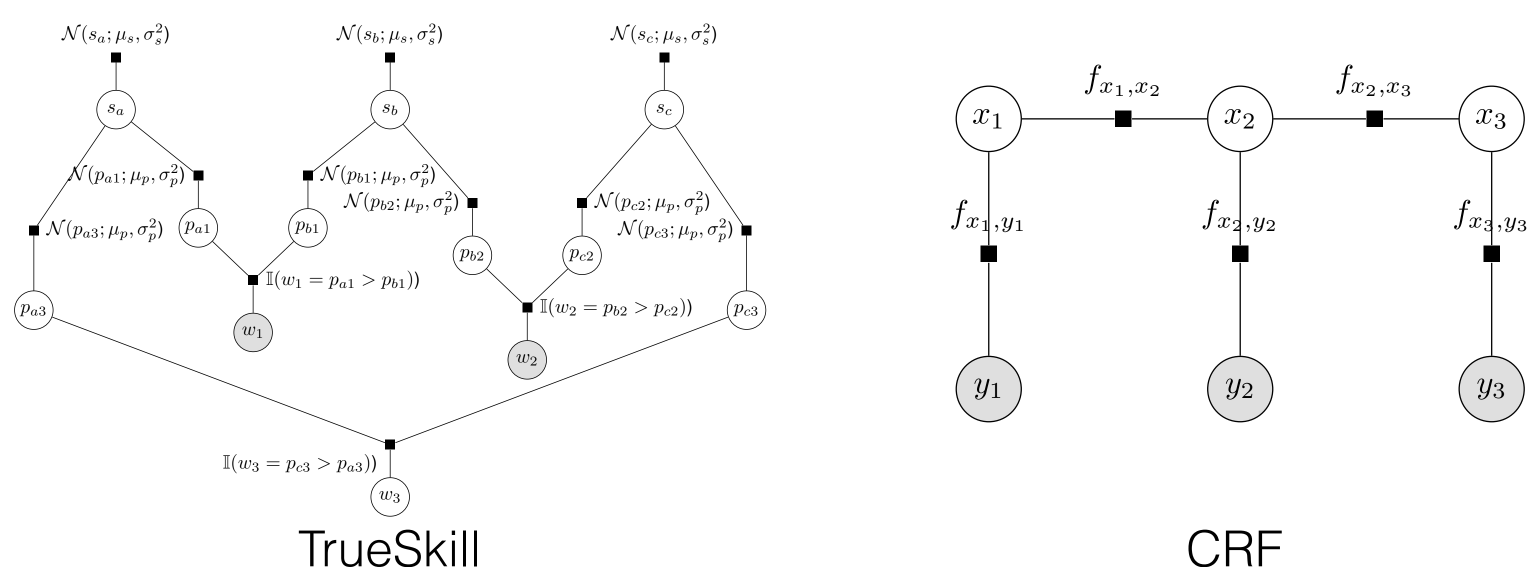
- Language restrictions
  - Bounded loops
  - No discrete random variables*
- Model class
  - Finite dimensional differentiable distributions
- Inference - sampling
  - Hamiltonian Monte Carlo
    - Reverse-mode automatic differentiation
  - Black box variational inference, etc.



STAN Development Team "Stan: A C++ Library for Probability and Sampling." 2014.

# Factorie and Infer.NET

- Language restrictions
  - Finite compositions of factors
- Model class
  - Finite factor graphs
- Inference - message passing, etc.

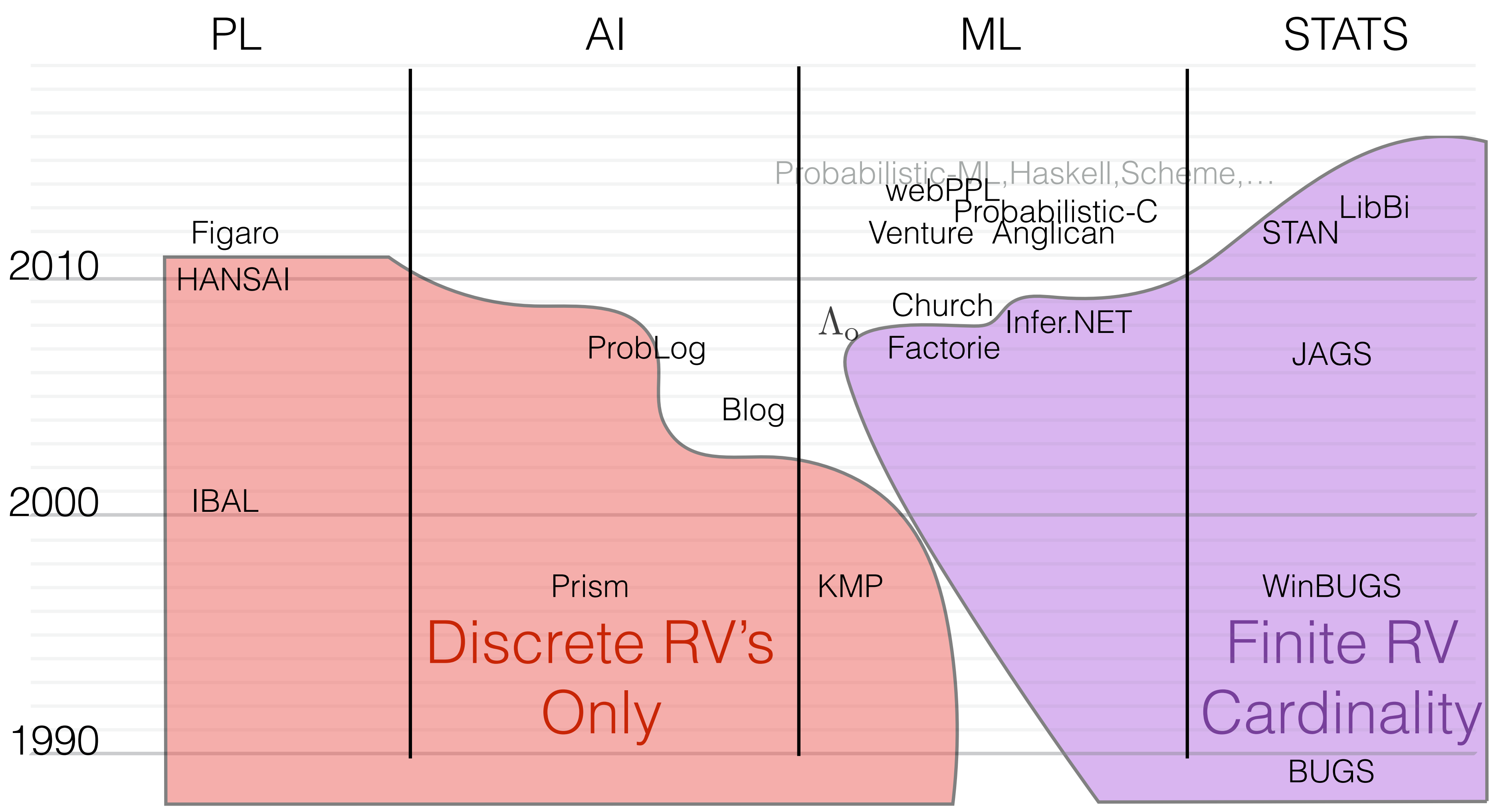




Minka, Winn, Guiver, and Knowles "Infer .NET 2.4, Microsoft Research Cambridge." 2010.

McCallum, Schultz, and Singh. “Factorie Probabilistic programming via imperatively defined factor graphs.“ NIPS 2009

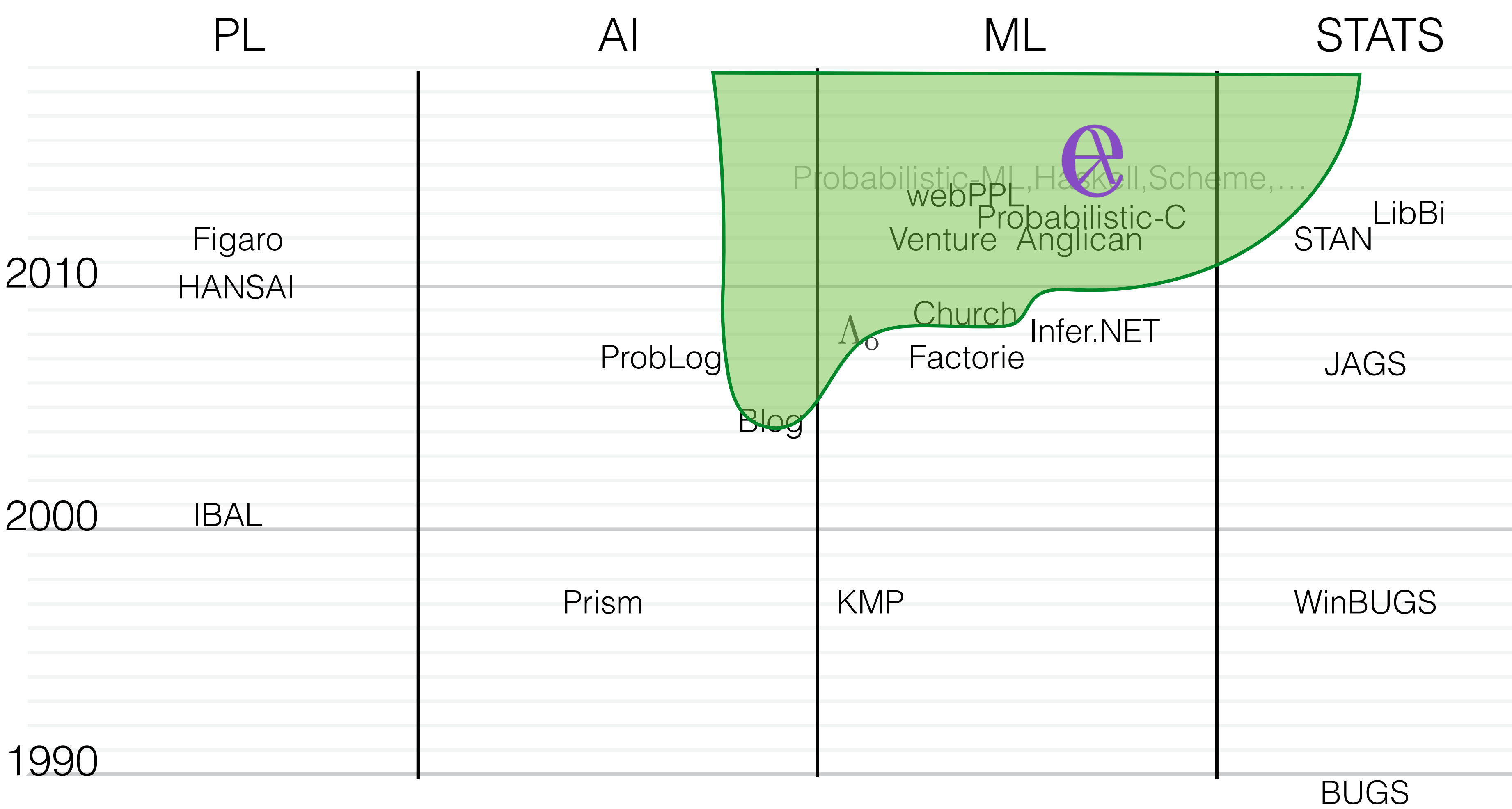
# First-Order PPLs : (FOPPL)s




Simula

Prolog

# Higher-Order PPLs : (HOPPL)s




Simula

Prolog

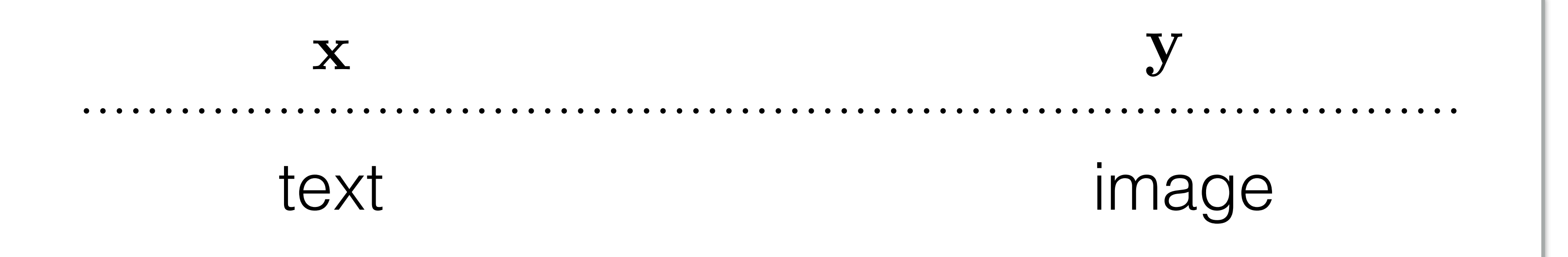
# CAPTCHA breaking

## Observation

## Posterior Samples

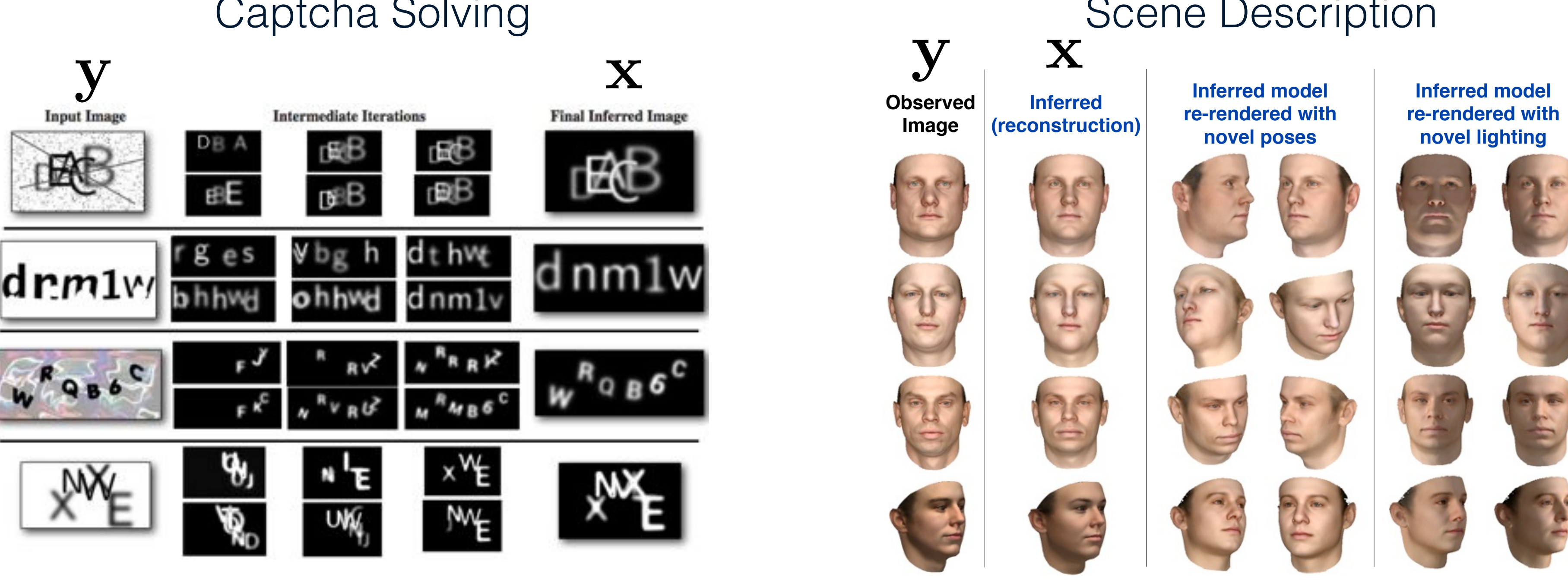
## Generative Model

```
(defquery captcha
 [image num-chars tol]
 (let [[w h] (size image)
       ;; sample random characters
       num-chars (sample
                   (poisson num-chars))
       chars (repeatedly
               num-chars sample-char)]
  ;; compare rendering to true image
  (map (fn [y z]
         (observe (normal z tol) y))
       (reduce-dim image)
       (reduce-dim (render chars w h)))
  ;; predict captcha text
  {:text
   (map :symbol (sort-by :x chars))}))
```




Mansinghka,, Kulkarni, Perov, and Tenenbaum
"Approximate Bayesian image interpretation using generative probabilistic graphics programs." NIPS (2013).

# Perception / Inverse Graphics




$\mathbf{x}$ ........ $\mathbf{y}$

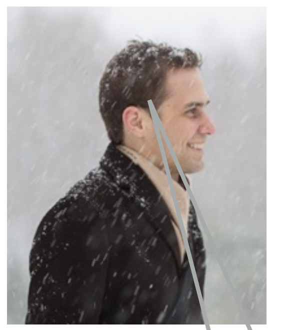
scene description ........ image

Mansinghka,, Kulkarni, Perov, and Tenenbaum. "Approximate Bayesian image interpretation using generative probabilistic graphics programs." NIPS (2013).

Kulkarni, Kohli, Tenenbaum, Mansinghka "Picture: a probabilistic programming language for scene perception." CVPR (2015).

# Reasoning about reasoning

Want to meet up but phones are dead…



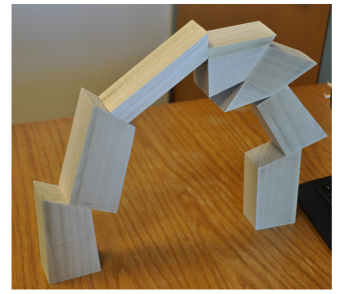
I prefer the pub.
Where will Noah go?
Simulate Noah:
Noah prefers pub
but will go wherever Andreas is
Simulate Noah simulating Andreas:
…
-> both go to pub

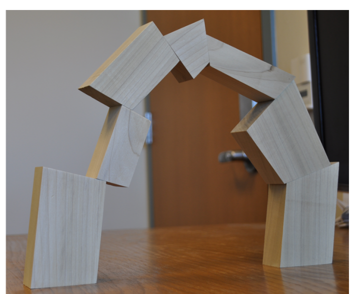
| $\mathbf{x}$ | $\mathbf{y}$ |
| --- | --- |
| cognitive process | behavior |

Stuhlmüller, and Goodman.
"Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs."
Cognitive Systems Research 28 (2014): 80-99.

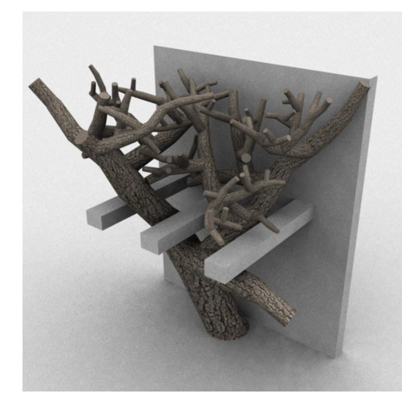
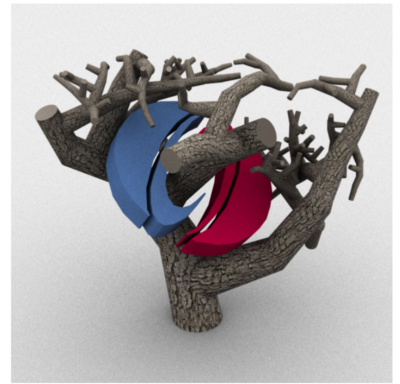
# Directed Procedural Graphics

Stable Static Structures





Procedural Graphics

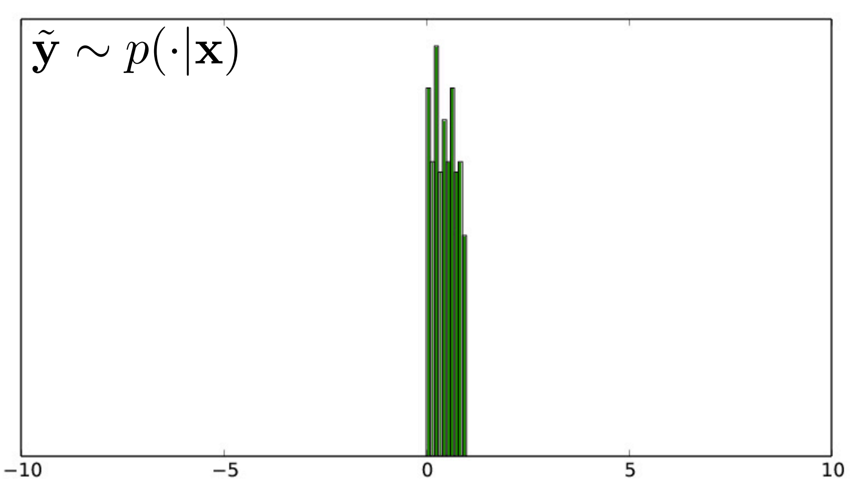



| $\mathbf{x}$ | $\mathbf{y}$ |
|---|---|
| simulation | constraint |


Ritchie, Lin, Goodman, & Hanrahan.
Generating Design Suggestions under Tight Constraints
with Gradient-based Probabilistic Programming.
In Computer Graphics Forum, (2015)

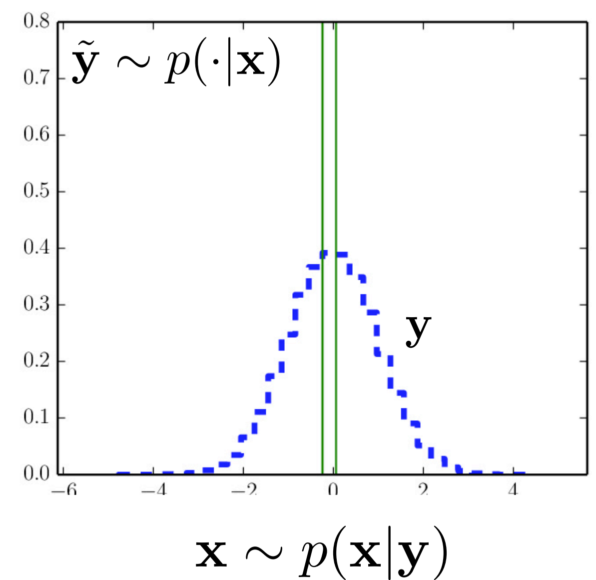
Ritchie, Mildenhall, Goodman, & Hanrahan.
"Controlling Procedural Modeling Programs with
Stochastically-Ordered Sequential Monte Carlo."
SIGGRAPH (2015)

# Program Induction




```
(lambda (stack-id) (safe-uc (* (if (< 0.0 (* (* (* -1.0 (begin (define
 G__1147 (safe-uc 1.0 1.0)) 0.0)) (* 0.0 (+ 0.0 (safe-uc (* (* (dec -2
.0) (safe-sqrt (begin (define G__1148 3.14159) (safe-log -1.0)))) 2.0)
 0.0)))) 1.0)) (+ (safe-div (begin (define G__1149 (* (+ 3.14159 -1.0)
 1.0)) 1.0) 0.0) (safe-log 1.0)) (safe-log -1.0)) (begin (define G__11
...
```

$\mathbf{x} \sim p(\mathbf{x})$

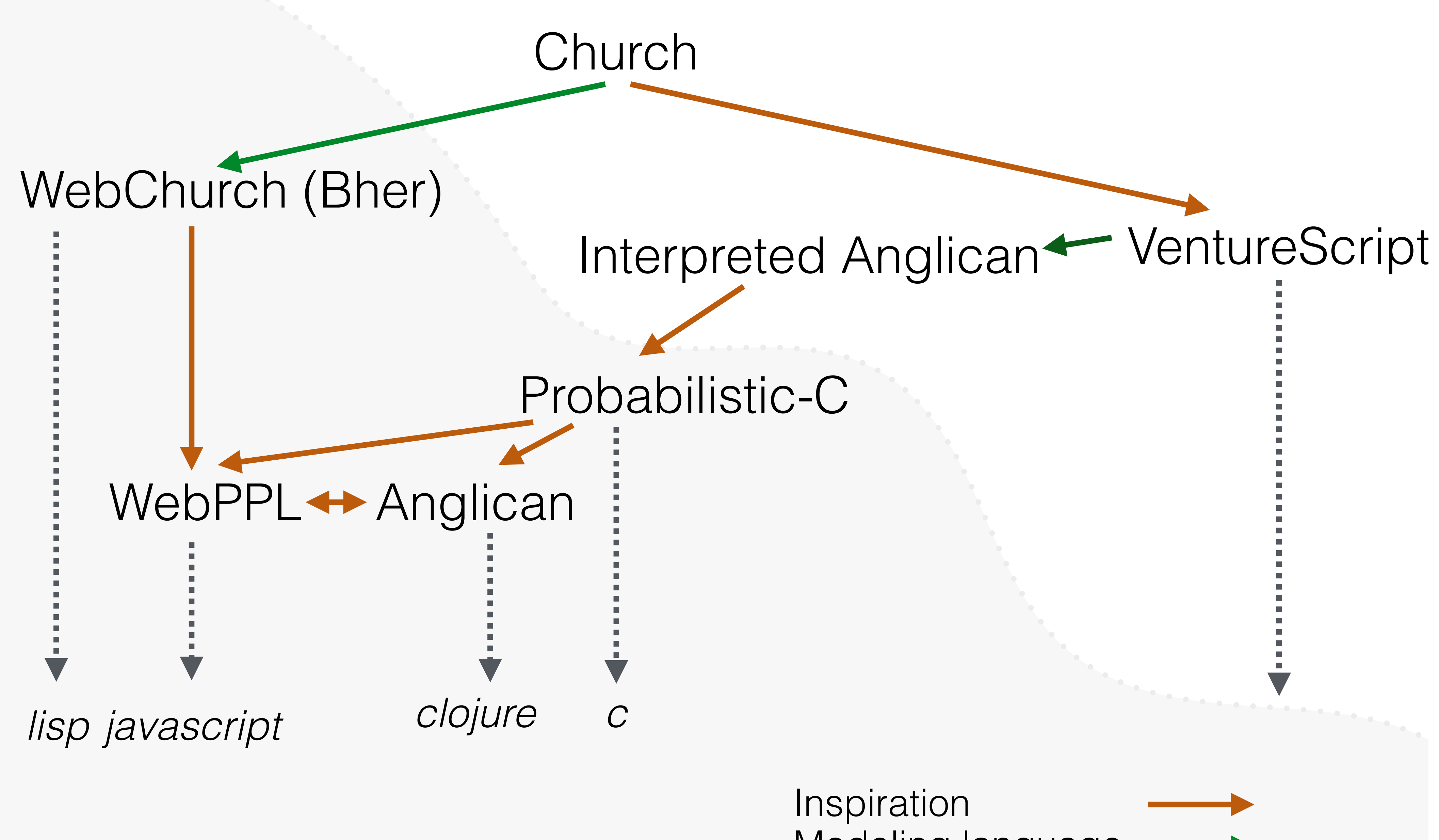


Perov and **Wood**.
"Automatic Sampler Discovery via Probabilistic Programming and Approximate Bayesian Computation"
AGI (2016).

# Higher Order Probabilistic Programming Modeling Language

# Introduction to Anglican/Church/Venture/WebPPL…

# A Language Family Tree



# Anglican By Example : Graphical Model

```
(defquery gaussian-model [data]
  (let [x (sample (normal 1 (sqrt 5)))
        sigma (sqrt 2)]
    (map (fn [y] (observe (normal x sigma) y)) data)
    x))
```

$$x \sim \text{Normal}(1, \sqrt{5})$$

$$y_i | x \sim \text{Normal}(x, \sqrt{2})$$

```
(def dataset [9 8])
```

$$y_1 = 9, y_2 = 8$$

```
(def posterior
  ((conditional gaussian-model
                :pgibbs
                :number-of-particles 1000) dataset))
```

$$x | \mathbf{y} \sim \text{Normal}(7.25, 0.91)$$

```
(def posterior-samples
  (repeatedly 20000 #(sample posterior)))
```

0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
-10 -8 -6 -4 -2 0 2 4 6 8 10

# Graphical Model

```
(defquery gaussian-model [data]
  (let [x (sample (normal 1 (sqrt 5)))
        sigma (sqrt 2)]
    (map (fn [y] (observe (normal x sigma) y)) data)
    x))
```

$$x \sim \text{Normal}(1, \sqrt{5})$$

$$y_i | x \sim \text{Normal}(x, \sqrt{2})$$

```
(def dataset [9 8])
```

$$y_1 = 9, y_2 = 8$$

```
(def posterior
  ((conditional gaussian-model
                :pgibbs
                :number-of-particles 1000) dataset))
```

$$x | \mathbf{y} \sim \text{Normal}(7.25, 0.91)$$

```
(def posterior-samples
  (repeatedly 20000 #(sample posterior)))
```

# Graphical Model

```
(defquery gaussian-model [data]
  (let [x (sample (normal 1 (sqrt 5)))
        sigma (sqrt 2)]
    (map (fn [y] (observe (normal x sigma) y)) data)
    x))
```

$$x \sim \text{Normal}(1, \sqrt{5})$$

$$y_i | x \sim \text{Normal}(x, \sqrt{2})$$

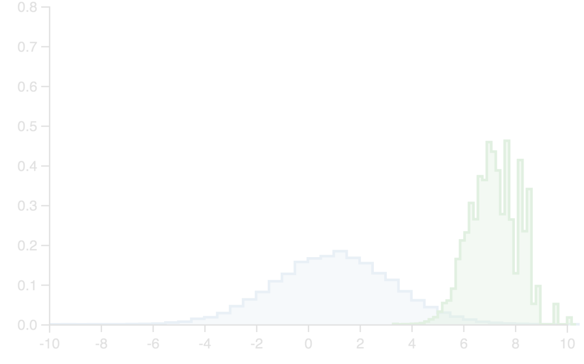
```
(def dataset [9 8])
```

$$y_1 = 9, y_2 = 8$$

```
(def posterior
  ((conditional gaussian-model
                :pgibbs
                :number-of-particles 1000) dataset))
```

$$x | \mathbf{y} \sim \text{Normal}(7.25, 0.91)$$

```
(def posterior-samples
  (repeatedly 20000 #(sample posterior)))
```



# Graphical Model

```
(defquery gaussian-model [data]
  (let [x (sample (normal 1 (sqrt 5)))
        sigma (sqrt 2)]
    (map (fn [y] (observe (normal x sigma) y)) data)
    x))
```

$$x \sim \text{Normal}(1, \sqrt{5})$$

$$y_i | x \sim \text{Normal}(x, \sqrt{2})$$

```
(def dataset [9 8])
```

$$y_1 = 9, y_2 = 8$$

```
(def posterior
  ((conditional gaussian-model
                :pgibbs
                :number-of-particles 1000) dataset))
```

$$x | \mathbf{y} \sim \text{Normal}(7.25, 0.91)$$

```
(def posterior-samples
  (repeatedly 20000 #(sample posterior)))
```

# Anglican : Syntax ≈ Clojure, Semantics ≠ Clojure

```
(defquery gaussian-model [data]
  (let [x (sample (normal 1 (sqrt 5)))
        sigma (sqrt 2)]
    (map (fn [y] (observe (normal x sigma) y)) data)
    x))
```

$x \sim \text{Normal}(1, \sqrt{5})$

$y_i | x \sim \text{Normal}(x, \sqrt{2})$

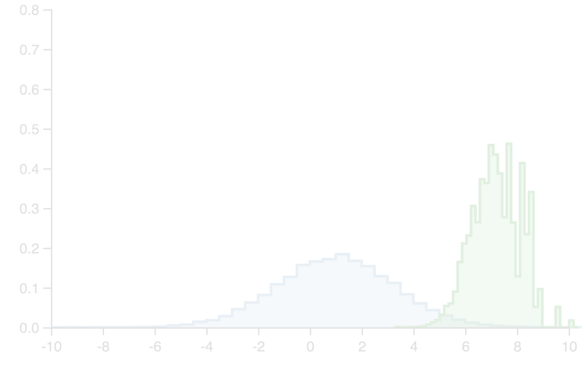
```
(def dataset [9 8])
```

$y_1 = 9, y_2 = 8$

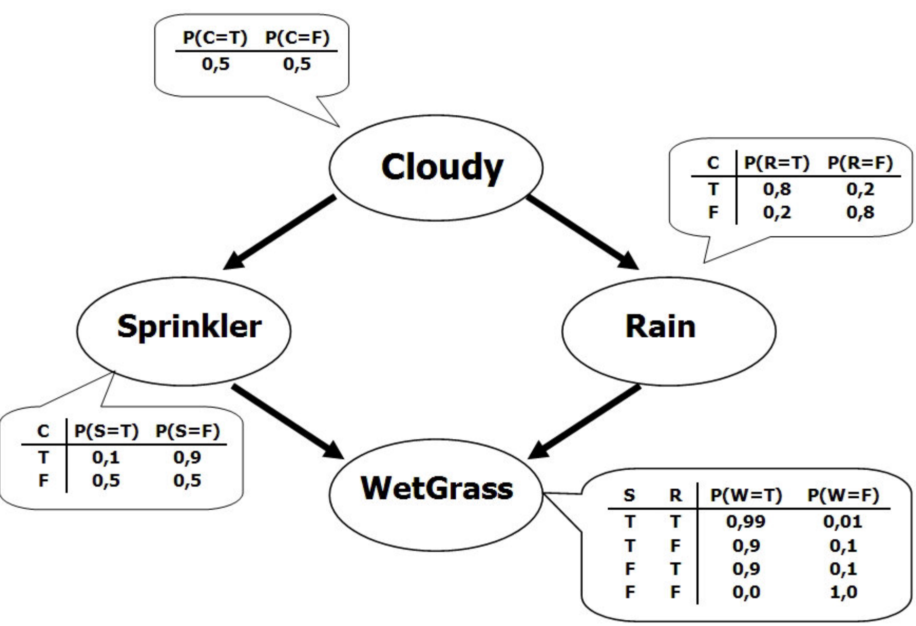
```
(def posterior
  ((conditional gaus                  l
                :pgi
                :num              rticles 1000) dataset))
```

$x | \mathbf{y} \sim \text{Normal}(7.25, 0.91)$

```
(def posterior-samples
  (repeatedly 20000 #(sample posterior)))
```



# Bayes Net




```
(defquery sprinkler-bayes-net [sprinkler wet-grass]
  (let [is-cloudy (sample (flip 0.5))

        is-raining (cond (= is-cloudy true )
                         (sample (flip 0.8))
                         (= is-cloudy false)
                         (sample (flip 0.2)))
        sprinkler-dist (cond (= is-cloudy true)
                             (flip 0.1)
                             (= is-cloudy false)
                             (flip 0.5))
        wet-grass-dist (cond
                         (and (= sprinkler true)
                              (= is-raining true))
                         (flip 0.99)
                         (and (= sprinkler false)
                              (= is-raining false))
                         (flip 0.0)
                         (or  (= sprinkler true)
                              (= is-raining true))
                         (flip 0.9))]
    (observe sprinkler-dist sprinkler)
    (observe wet-grass-dist wet-grass)

    is-raining))
```

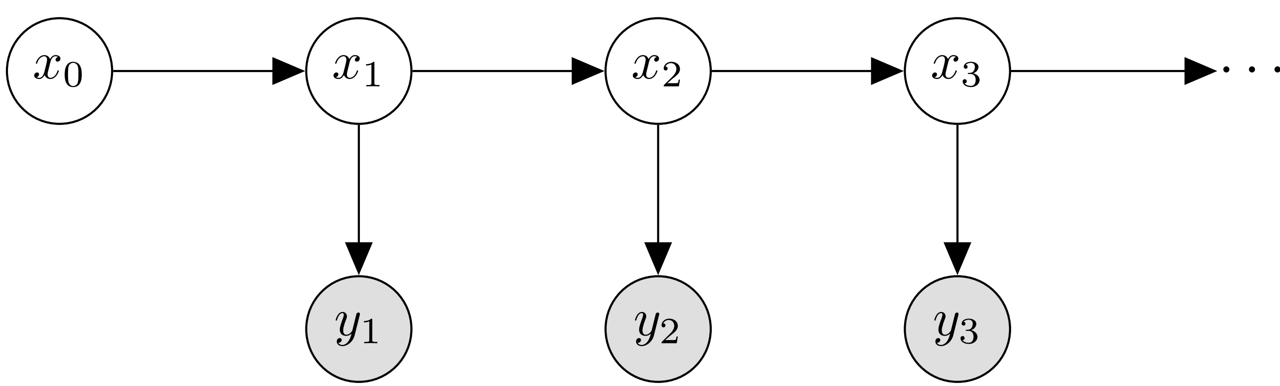
# One Hidden Markov Model

$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots$, with $x_1 \rightarrow y_1$, $x_2 \rightarrow y_2$, $x_3 \rightarrow y_3$

```
(defquery hmm
  (let [init-dist (discrete [1 1 1])
        trans-dist (fn [s]
                     (cond
                       (= s 0) (discrete [0 1 1])
                       (= s 1) (discrete [0 0 1])
                       (= s 2) (dirac 2)))
        obs-dist (fn [s] (normal s 1))
        y-1 1
        y-2 1
        x-0 (sample init-dist)
        x-1 (sample (trans-dist x-0))
        x-2 (sample (trans-dist x-1))]

    (observe (obs-dist x-1) y-1)
    (observe (obs-dist x-2) y-2)
    [x-0 x-1 x-2]))
```

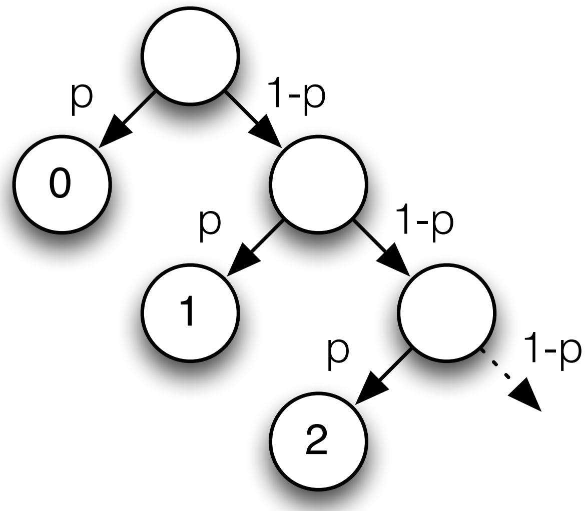
# All Hidden Markov Models

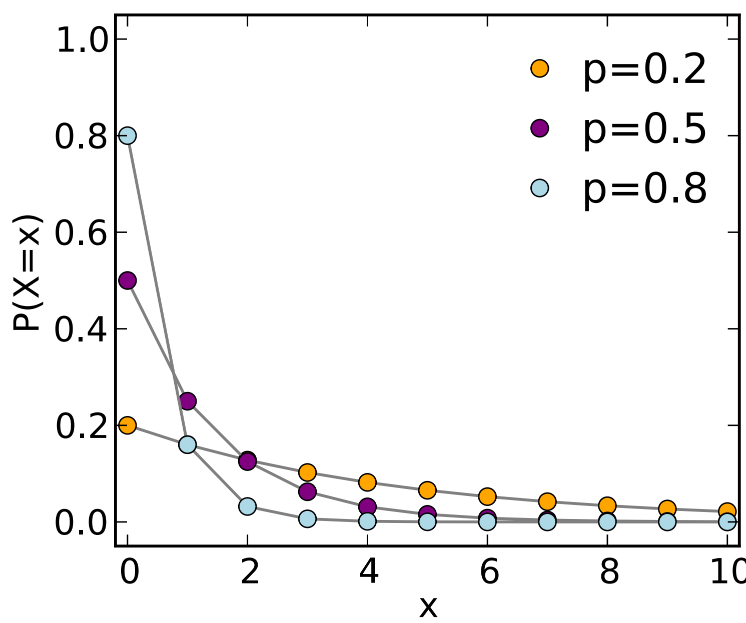
```
(defquery hmm
  [ys init-dist trans-dists obs-dists]
  (reduce
    (fn [xs y]
      (let [x (sample (get trans-dists (peek xs)))]
        (observe (get obs-dists x) y)
        (conj xs x)))
    [(sample init-dist)]
    ys))
```



# New Primitives

```
(defquery geometric [p]
  "geometric distribution"
  (let [dist (flip p)
        samp (loop [n 0]
               (if (sample dist)
                 n
                 (recur (+ n 1))))]
    samp))
```



# A Hard Inference Problem

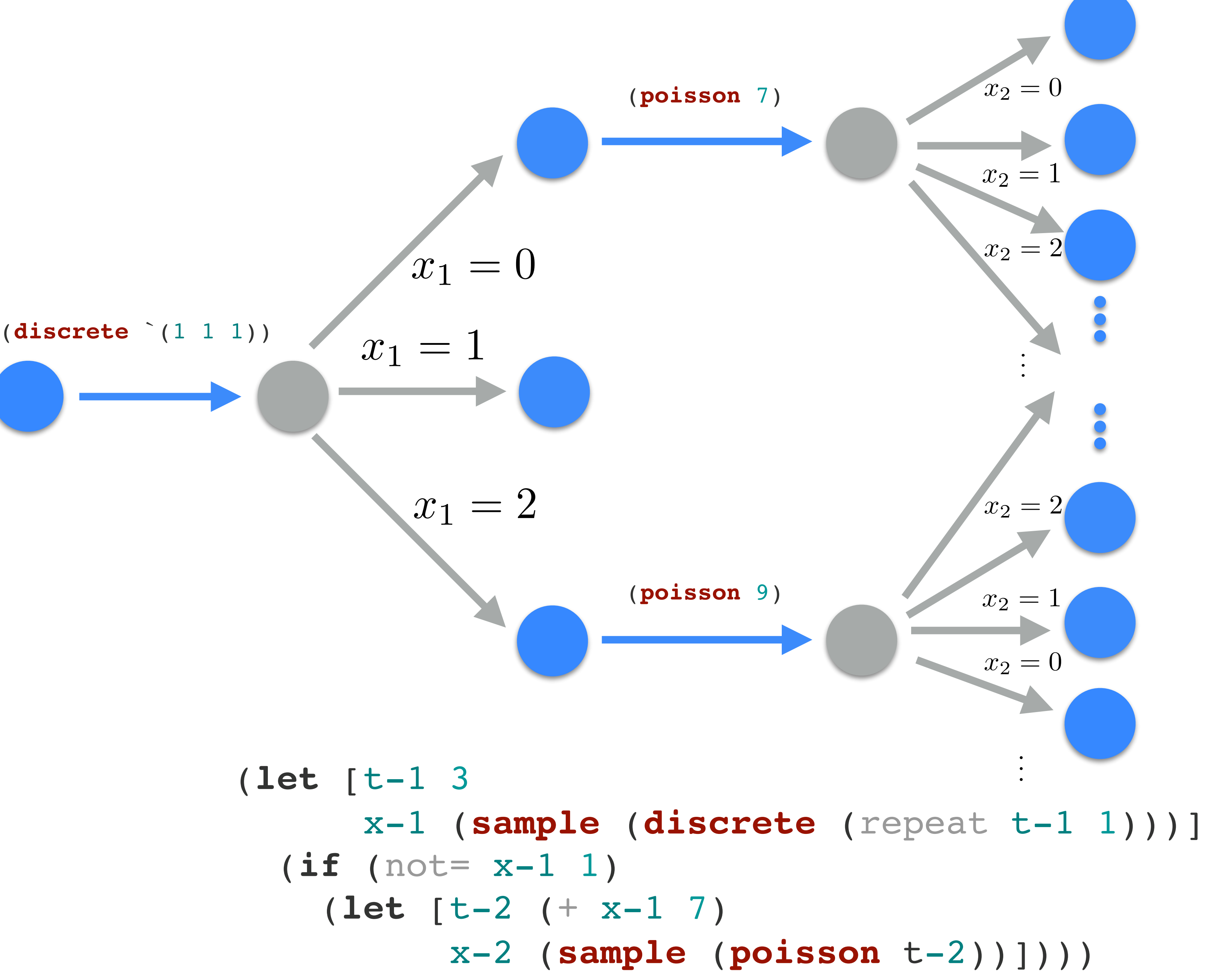
```
(defquery md5-inverse [L md5str]
    "conditional distribution of strings
     that map to the same MD5 hashed string"
    (let [mesg (sample (string-generative-model L))]
      (observe (dirac md5str) (md5 mesg))
      mesg)))
```

# Evaluation-Based Inference for Higher-Order PPLs

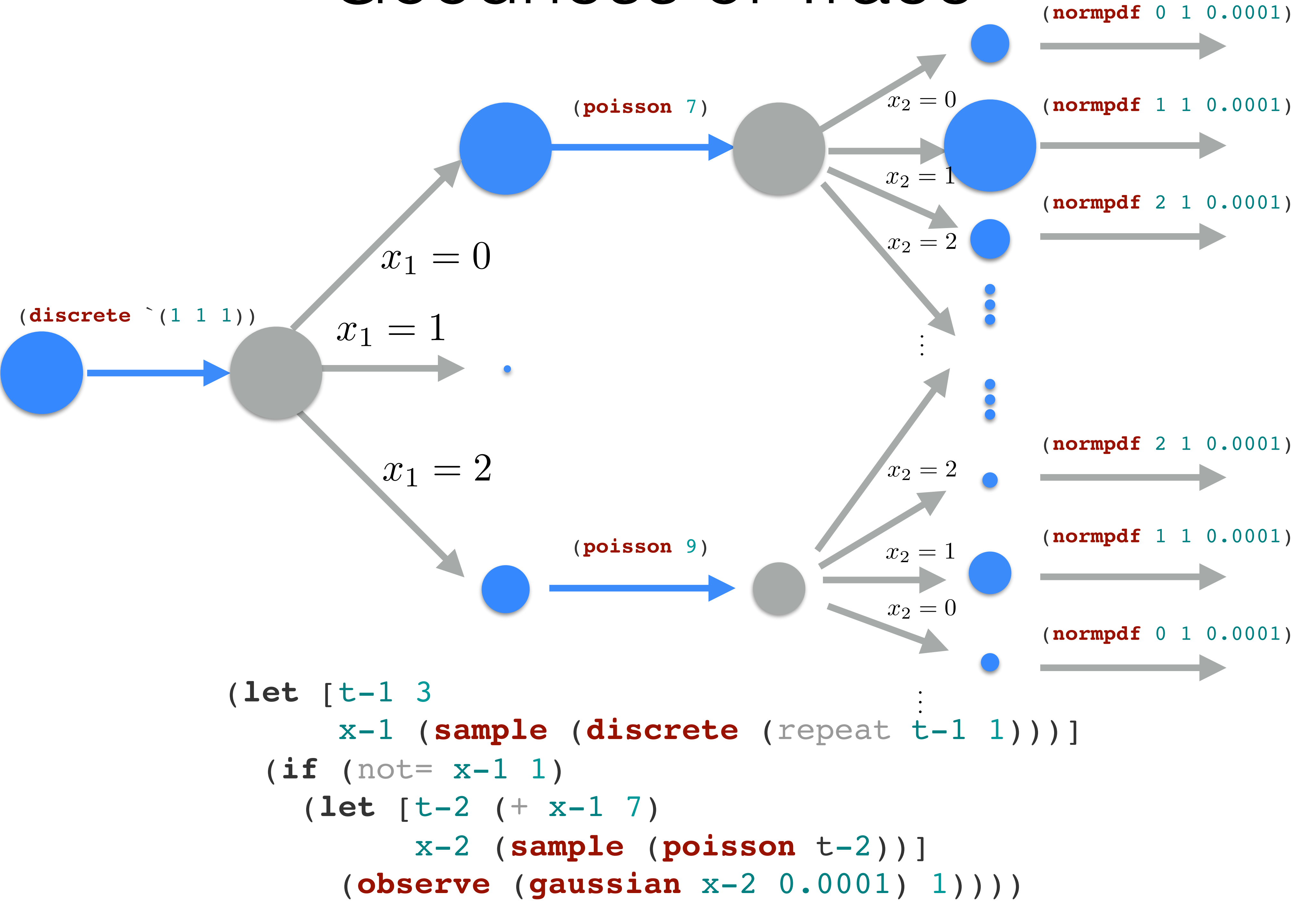
# The Gist

- Explore as many "traces" as possible, intelligently
  - Each trace contains all random choices made during the execution of a generative model
- Compute trace "goodness" (probability) as side-effect
- Combine weighted traces probabilistically coherently
- Report projection of posterior over traces

# Traces



# Goodness of Trace



# Trace

- Sequence of $N$ **observe**'s

$$\{(g_i, \phi_i, y_i)\}_{i=1}^N$$

- Sequence of $M$ **sample**'s

$$\{(f_j, \theta_j)\}_{j=1}^M$$

- Sequence of $M$ sampled values

$$\{x_j\}_{j=1}^M$$

- Conditioned on these sampled values the entire computation is *deterministic*

# Trace Probability

- Defined as (up to a normalization constant)

$$\gamma(\mathbf{x}) \triangleq p(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{N} g_i(y_i|\phi_i) \prod_{j=1}^{M} f_j(x_j|\theta_j)$$

- Hides true dependency structure

$$\gamma(\mathbf{x}) = p(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{N} \tilde{g}_i(\mathbf{x}_{n_i}) \left( y_i \middle| \tilde{\phi}_i(\mathbf{x}_{n_i}) \right) \prod_{j=1}^{M} \tilde{f}_j(\mathbf{x}_{j-1}) \left( x_j \middle| \tilde{\theta}_j(\mathbf{x}_{j-1}) \right)$$

$\mathbf{x}_6$

$\mathbf{x}_4$

$\mathbf{x}_j = x_1 \times \cdots \times x_j$

$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $etc$

$y_1$ $y_2$

# Inference Goal

- Posterior over traces

$$\pi(\mathbf{x}) \triangleq p(\mathbf{x}|\mathbf{y}) = \frac{\gamma(\mathbf{x})}{Z}$$

$$Z = p(\mathbf{y}) = \int \gamma(\mathbf{x}) d\mathbf{x}$$

- Output

$$\mathbb{E}[z] = \mathbb{E}[Q(\mathbf{x})] = \int Q(\mathbf{x})\pi(\mathbf{x}) d\mathbf{x} = \frac{1}{Z} \int Q(\mathbf{x}) \frac{\gamma(\mathbf{x})}{q(\mathbf{x})} q(\mathbf{x}) d\mathbf{x}$$

# Three Base Algorithms

- Likelihood Weighting
- Sequential Monte Carlo
- Metropolis Hastings

# Likelihood Weighting

- Run $K$ independent copies of program simulating from the prior

$$q(\mathbf{x}^k) = \prod_{j=1}^{M^k} f_j(x_j^k | \theta_j^k)$$

- Accumulate *unnormalized* weights (likelihoods)

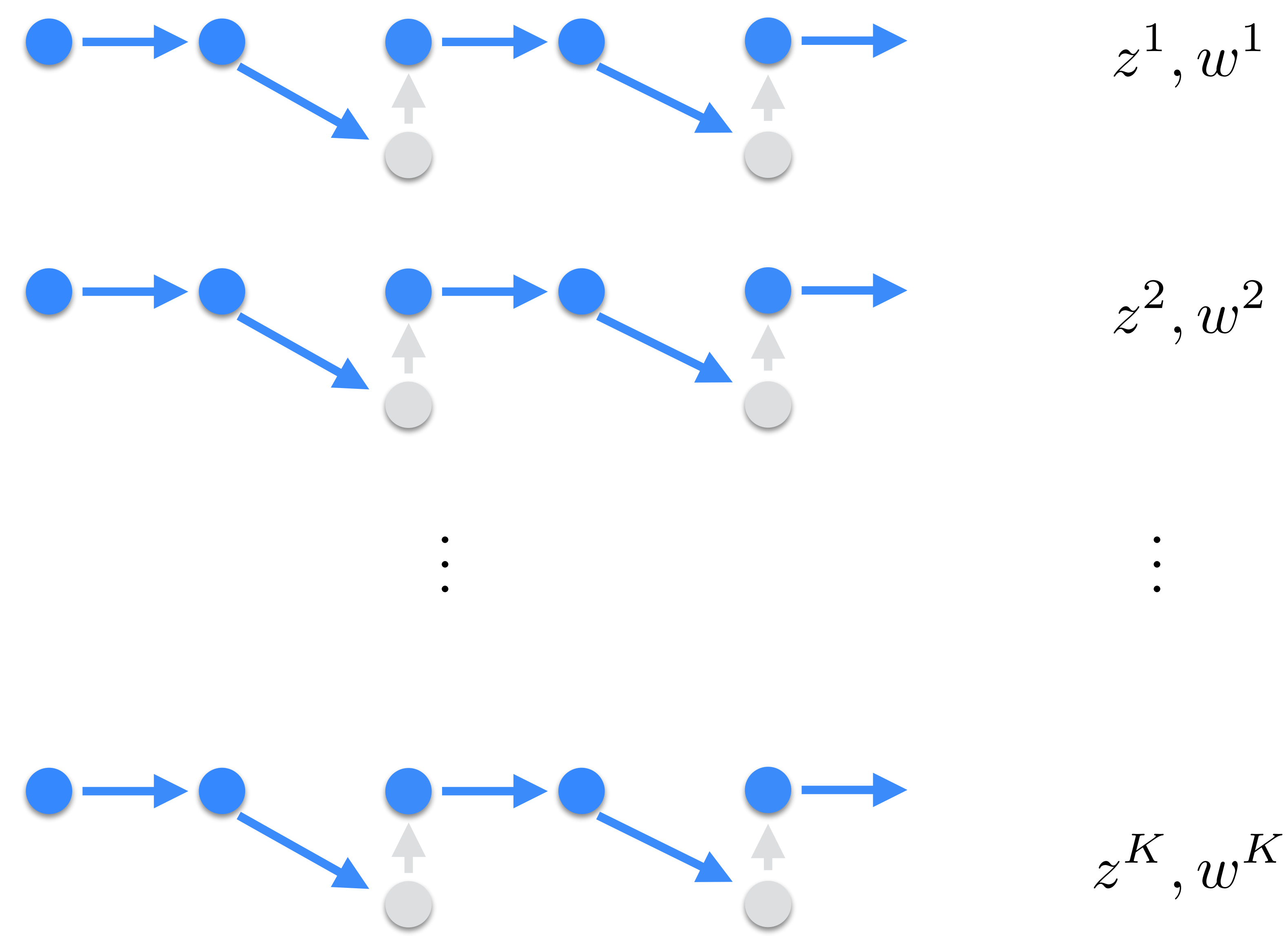
$$w(\mathbf{x}^k) = \frac{\gamma(\mathbf{x}^k)}{q(\mathbf{x}^k)} = \prod_{i=1}^{N^k} g_i^k(y_i^k | \phi_i^k)$$

- Use in approximate (Monte Carlo) integration

$$W^k = \frac{w(\mathbf{x}^k)}{\sum_{\ell=1}^{K} w(\mathbf{x}^\ell)} \qquad \widehat{\mathbb{E}}_\pi[Q(\mathbf{x})] = \sum_{k=1}^{K} W^k Q(\mathbf{x}^k)$$

BLOG default inference engine:
http://bayesianlogic.github.io/pages/users-manual.html

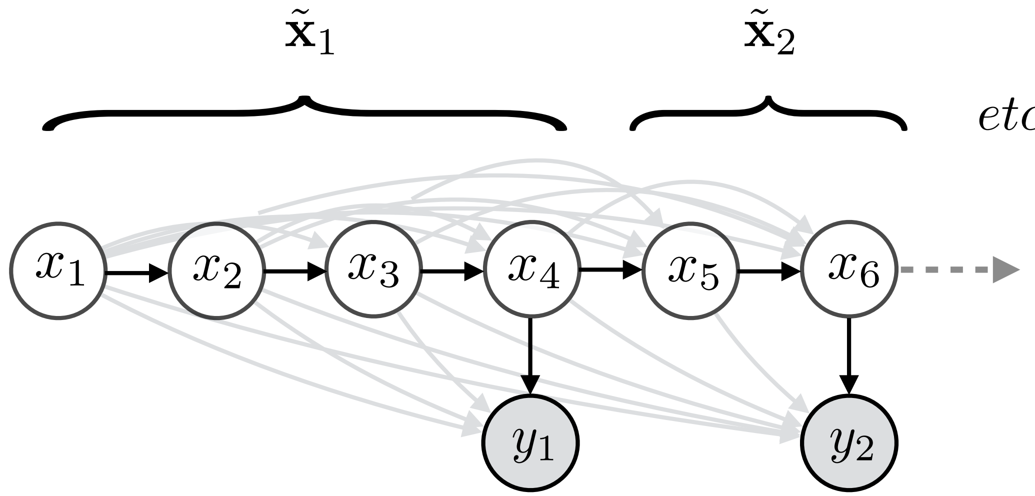
# Likelihood Weighting Schematic



# Sequential Monte Carlo

- Notation

$$\tilde{\mathbf{x}}_{1:n} = \tilde{\mathbf{x}}_1 \times \cdots \times \tilde{\mathbf{x}}_n$$




- Incrementalized joint

$$\gamma_n(\tilde{\mathbf{x}}_{1:n}) = \prod_{n=1}^{N} g(y_n|\tilde{\mathbf{x}}_{1:n})p(\tilde{\mathbf{x}}_n|\tilde{\mathbf{x}}_{1:n-1})$$

- Incrementalized target

$$\pi_n(\tilde{\mathbf{x}}_{1:n}) = \frac{1}{Z_n}\gamma_n(\tilde{\mathbf{x}}_{1:n})$$

# SMC for Probabilistic Programming

Want samples from

$$\pi_n(\tilde{\mathbf{x}}_{1:n}) \propto p(y_n|\tilde{\mathbf{x}}_{1:n})p(\tilde{\mathbf{x}}_n|\tilde{\mathbf{x}}_{1:n-1})\pi_{n-1}(\tilde{\mathbf{x}}_{1:n-1})$$

Have a sample-based approximation to

$$\hat{\pi}_{n-1}(\tilde{\mathbf{x}}_{1:n-1}) \triangleq \sum_{k=1}^{K} W_{n-1}^k \delta_{\tilde{\mathbf{x}}_{1:n-1}^k}(\tilde{\mathbf{x}}_{1:n-1})$$

Sample from

$$\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \sim \hat{\pi}_{n-1}(\tilde{\mathbf{x}}_{1:n-1}) \qquad \tilde{\mathbf{x}}_n^k|\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \sim p(\tilde{\mathbf{x}}_n|\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k})$$

$$\tilde{\mathbf{x}}_{1:n}^k = \tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \times \tilde{\mathbf{x}}_n^k$$

Importance weight by

$$w(\tilde{\mathbf{x}}_{1:n}^k) = p(y_n|\tilde{\mathbf{x}}_{1:n}^k) = g_n^k(y_n|\tilde{\mathbf{x}}_{1:n}^k) \qquad W_n^k \triangleq \frac{w(\tilde{\mathbf{x}}_{1:n}^k)}{\sum_{k'=1}^{K} w(\tilde{\mathbf{x}}_{1:n}^{k'})}$$


Wood, van de Meent, and Mansinghka “A New Approach to Probabilistic Programming Inference” AISTATS 2014
Paige and Wood “A Compilation Target for Probabilistic Programming Languages” ICML 2014

# SMC for Probabilistic Programming

Want samples from

$$\pi_n(\tilde{\mathbf{x}}_{1:n}) \propto p(y_n|\tilde{\mathbf{x}}_{1:n})p(\tilde{\mathbf{x}}_n|\tilde{\mathbf{x}}_{1:n-1})\pi_{n-1}(\tilde{\mathbf{x}}_{1:n-1})$$

Have a sample-based approximation to

$$\hat{\pi}_{n-1}(\tilde{\mathbf{x}}_{1:n-1}) \triangleq \sum_{k=1}^{K} W_{n-1}^k \delta_{\tilde{\mathbf{x}}_{1:n-1}^k}(\tilde{\mathbf{x}}_{1:n-1})$$

Sample from

$$\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \sim \hat{\pi}_{n-1}(\tilde{\mathbf{x}}_{1:n-1}) \qquad \tilde{\mathbf{x}}_n^k|\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \sim p(\tilde{\mathbf{x}}_n|\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k})$$

$$\tilde{\mathbf{x}}_{1:n}^k = \tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \times \tilde{\mathbf{x}}_n^k$$

Importance weight by

$$w(\tilde{\mathbf{x}}_{1:n}^k) = p(y_n|\tilde{\mathbf{x}}_{1:n}^k) = g_n^k(y_n|\tilde{\mathbf{x}}_{1:n}^k) \qquad W_n^k \triangleq \frac{w(\tilde{\mathbf{x}}_{1:n}^k)}{\sum_{k'=1}^K w(\tilde{\mathbf{x}}_{1:n}^{k'})}$$

Wood, van de Meent, and Mansinghka “A New Approach to Probabilistic Programming Inference” AISTATS 2014
Paige and Wood “A Compilation Target for Probabilistic Programming Languages” ICML 2014

# SMC for Probabilistic Programming

Want samples from

$$\pi_n(\tilde{\mathbf{x}}_{1:n}) \propto p(y_n|\tilde{\mathbf{x}}_{1:n})p(\tilde{\mathbf{x}}_n|\tilde{\mathbf{x}}_{1:n-1})\pi_{n-1}(\tilde{\mathbf{x}}_{1:n-1})$$

Have a sample-based approximation to

$$\hat{\pi}_{n-1}(\tilde{\mathbf{x}}_{1:n-1}) \triangleq \sum_{k=1}^{K} W_{n-1}^k \delta_{\tilde{\mathbf{x}}_{1:n-1}^k}(\tilde{\mathbf{x}}_{1:n-1})$$

Sample from

$$\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \sim \hat{\pi}_{n-1}(\tilde{\mathbf{x}}_{1:n-1}) \qquad \tilde{\mathbf{x}}_n^k|\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \sim p(\tilde{\mathbf{x}}_n|\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k})$$

$$\tilde{\mathbf{x}}_{1:n}^k = \tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \times \tilde{\mathbf{x}}_n^k$$

Importance weight by

$$w(\tilde{\mathbf{x}}_{1:n}^k) = p(y_n|\tilde{\mathbf{x}}_{1:n}^k) = g_n^k(y_n|\tilde{\mathbf{x}}_{1:n}^k) \qquad W_n^k \triangleq \frac{w(\tilde{\mathbf{x}}_{1:n}^k)}{\sum_{k'=1}^K w(\tilde{\mathbf{x}}_{1:n}^{k'})}$$

Wood, van de Meent, and Mansinghka “A New Approach to Probabilistic Programming Inference” AISTATS 2014
Paige and Wood “A Compilation Target for Probabilistic Programming Languages” ICML 2014

# SMC for Probabilistic Programming

Want samples from

$$\pi_n(\tilde{\mathbf{x}}_{1:n}) \propto p(y_n|\tilde{\mathbf{x}}_{1:n})p(\tilde{\mathbf{x}}_n|\tilde{\mathbf{x}}_{1:n-1})\pi_{n-1}(\tilde{\mathbf{x}}_{1:n-1})$$

Have a sample-based approximation to

$$\hat{\pi}_{n-1}(\tilde{\mathbf{x}}_{1:n-1}) \triangleq \sum_{k=1}^{K} W_{n-1}^k \delta_{\tilde{\mathbf{x}}_{1:n-1}^k}(\tilde{\mathbf{x}}_{1:n-1})$$

Sample from

$$\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \sim \hat{\pi}_{n-1}(\tilde{\mathbf{x}}_{1:n-1})$$

$$\tilde{\mathbf{x}}_n^k|\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \sim p(\tilde{\mathbf{x}}_n|\tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k})$$

$$\tilde{\mathbf{x}}_{1:n}^k = \tilde{\mathbf{x}}_{1:n-1}^{a_{n-1}^k} \times \tilde{\mathbf{x}}_n^k$$

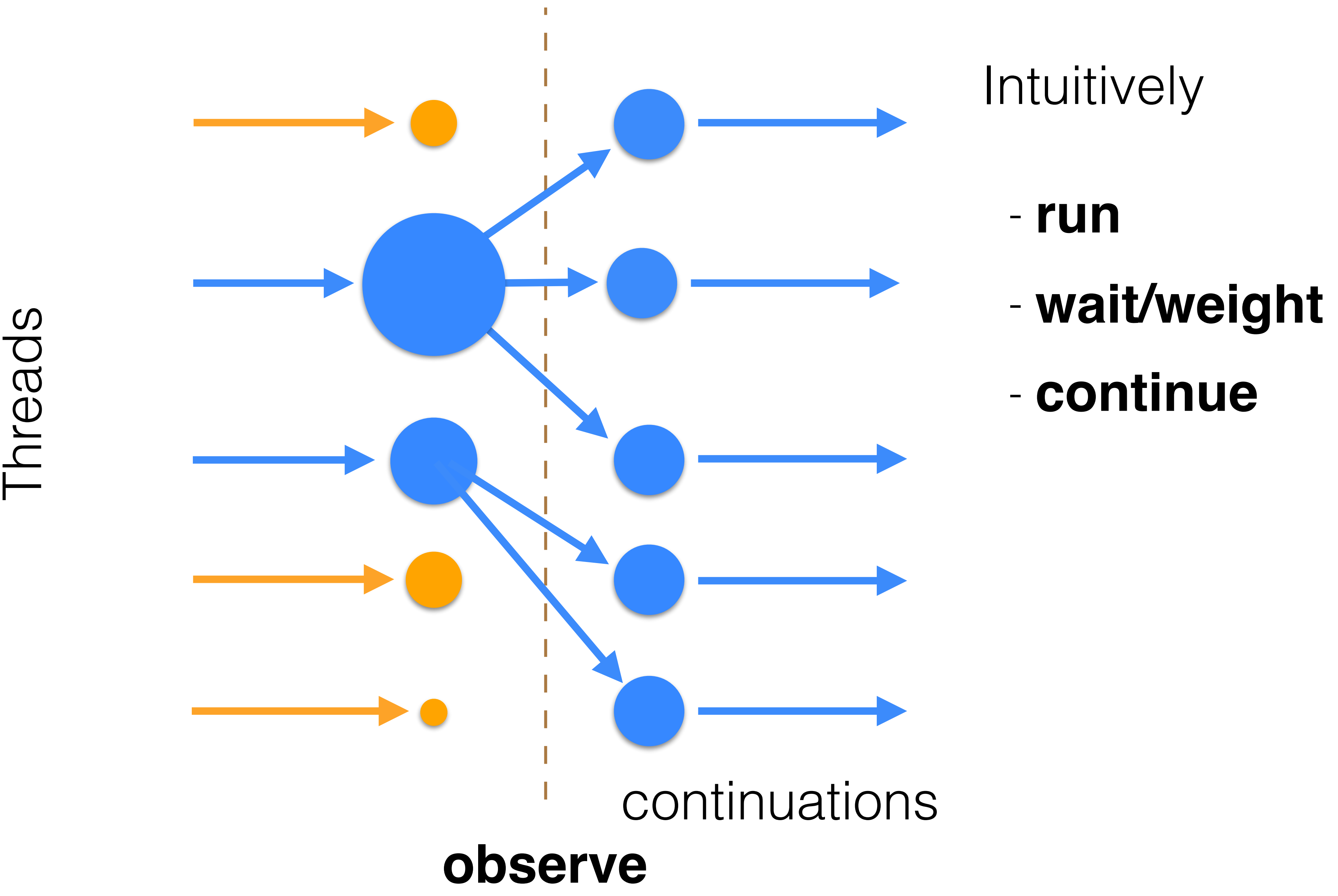
Importance weight by

$$w(\tilde{\mathbf{x}}_{1:n}^k) = p(y_n|\tilde{\mathbf{x}}_{1:n}^k) = g_n^k(y_n|\tilde{\mathbf{x}}_{1:n}^k)$$

$$W_n^k \triangleq \frac{w(\tilde{\mathbf{x}}_{1:n}^k)}{\sum_{k'=1}^K w(\tilde{\mathbf{x}}_{1:n}^{k'})}$$

Wood, van de Meent, and Mansinghka “A New Approach to Probabilistic Programming Inference” AISTATS 2014
Paige and Wood “A Compilation Target for Probabilistic Programming Languages” ICML 2014

# SMC Schematic




Intuitively

- **run**
- **wait/weight**
- **continue**

# Metropolis Hastings = “Single Site” MCMC = LMH

Posterior distribution of execution traces is proportional to trace score with observed values plugged in

$$\gamma(\mathbf{x}) \triangleq p(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{N} g_i(y_i|\phi_i) \prod_{j=1}^{M} f_j(x_j|\theta_j) \qquad \pi(\mathbf{x}) \triangleq p(\mathbf{x}|\mathbf{y}) = \frac{\gamma(\mathbf{x})}{Z}$$

Metropolis-Hastings acceptance rule

$$\alpha = \min\left(1, \frac{\pi(\mathbf{x}')q(\mathbf{x}|\mathbf{x}')}{\pi(\mathbf{x})q(\mathbf{x}'|\mathbf{x})}\right)$$

Need proposal

Milch and Russell “General-Purpose MCMC Inference over Relational Structures.” UAI 2006.
Goodman, Mansinghka, Roy, Bonawitz, and Tenenbaum “Church: a language for generative models.” UAI 2008.
Wingate, Stuhlmüller, Goodman “Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation” AISTATS 2011

# LMH Proposal

Probability of new part of proposed execution trace

$$q(\mathbf{x}'|\mathbf{x}^s) = \frac{1}{M^s}\kappa(x'_\ell|x^s_\ell)\prod_{j=\ell+1}^{M'} f'_j(x'_j|\theta'_j)$$

Number of samples in original trace

# LMH Acceptance Ratio

"Single site update" = sample from the prior = run program forward

$$\kappa(x'_m | x_m) = f_m(x'_m | \theta_m), \theta_m = \theta'_m$$

MH acceptance ratio

Number of sample statements in original trace

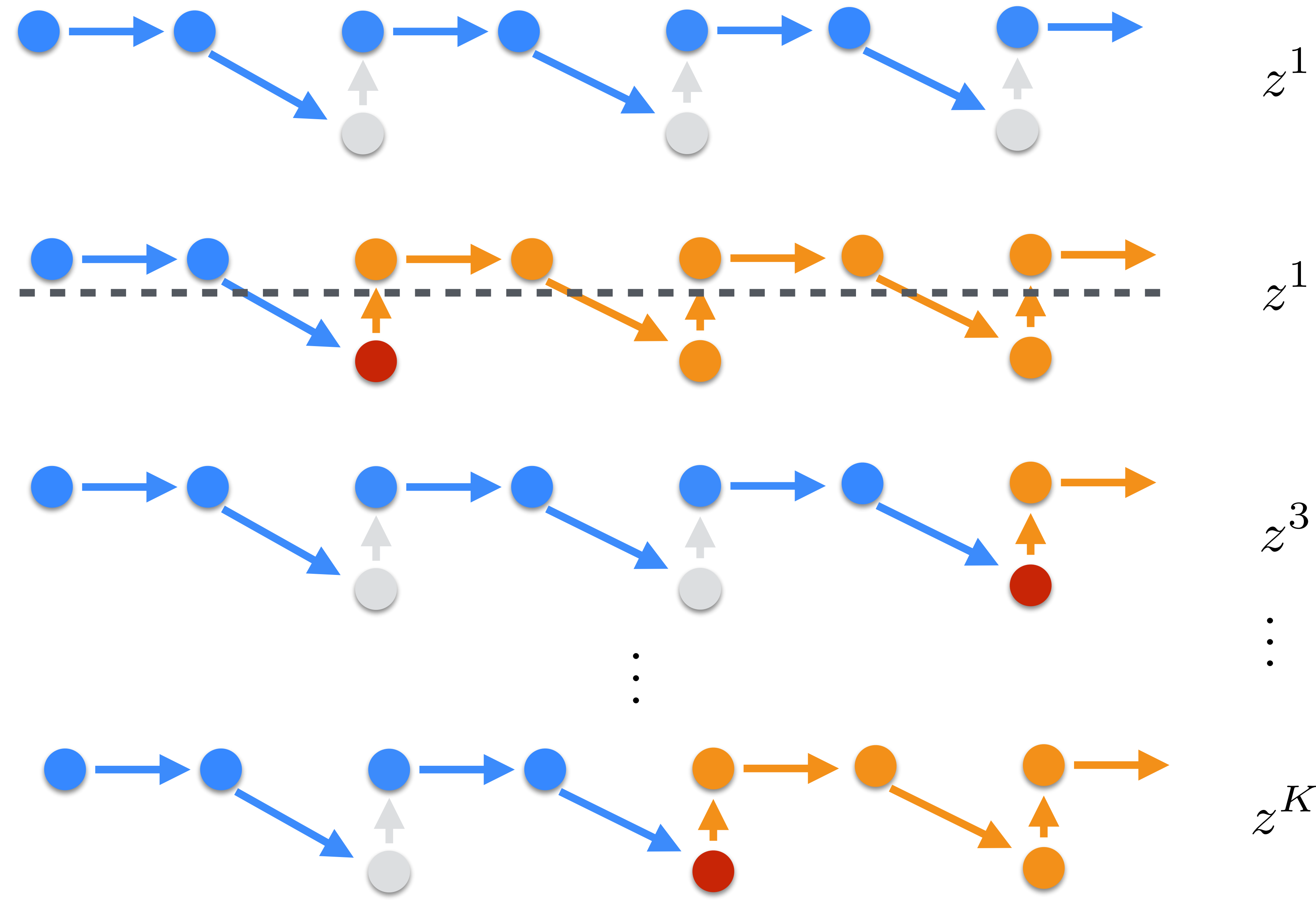
Probability of original trace continuation restarting proposal trace at $m^{th}$ sample

$$\alpha = \min\left(1, \frac{\gamma(\mathbf{x}')M \prod_{j=m}^{M} f_j(x_j|\theta_j)}{\gamma(\mathbf{x})M' \prod_{j=m}^{M'} f'_j(x'_j|\theta'_j)}\right)$$

Number of sample statements in new trace

Probability of proposal trace continuation restarting original trace at $m^{th}$ sample

# LMH Schematic



# Implementation Strategy

- Interpreted
  - Interpreter tracks side effects and directs control flow for inference
- Compiled
  - Leverages existing compiler infrastructure
  - Can only exert control over flow from *within* function calls
    - e.g. sample, observe, predict

Wingate, Stuhlmüller, Goodman "Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation" AISTATS 2011
Paige and Wood "A Compilation Target for Probabilistic Programming Languages" ICML 2014

# Probabilistic C

Standard C plus new directives: observe and predict

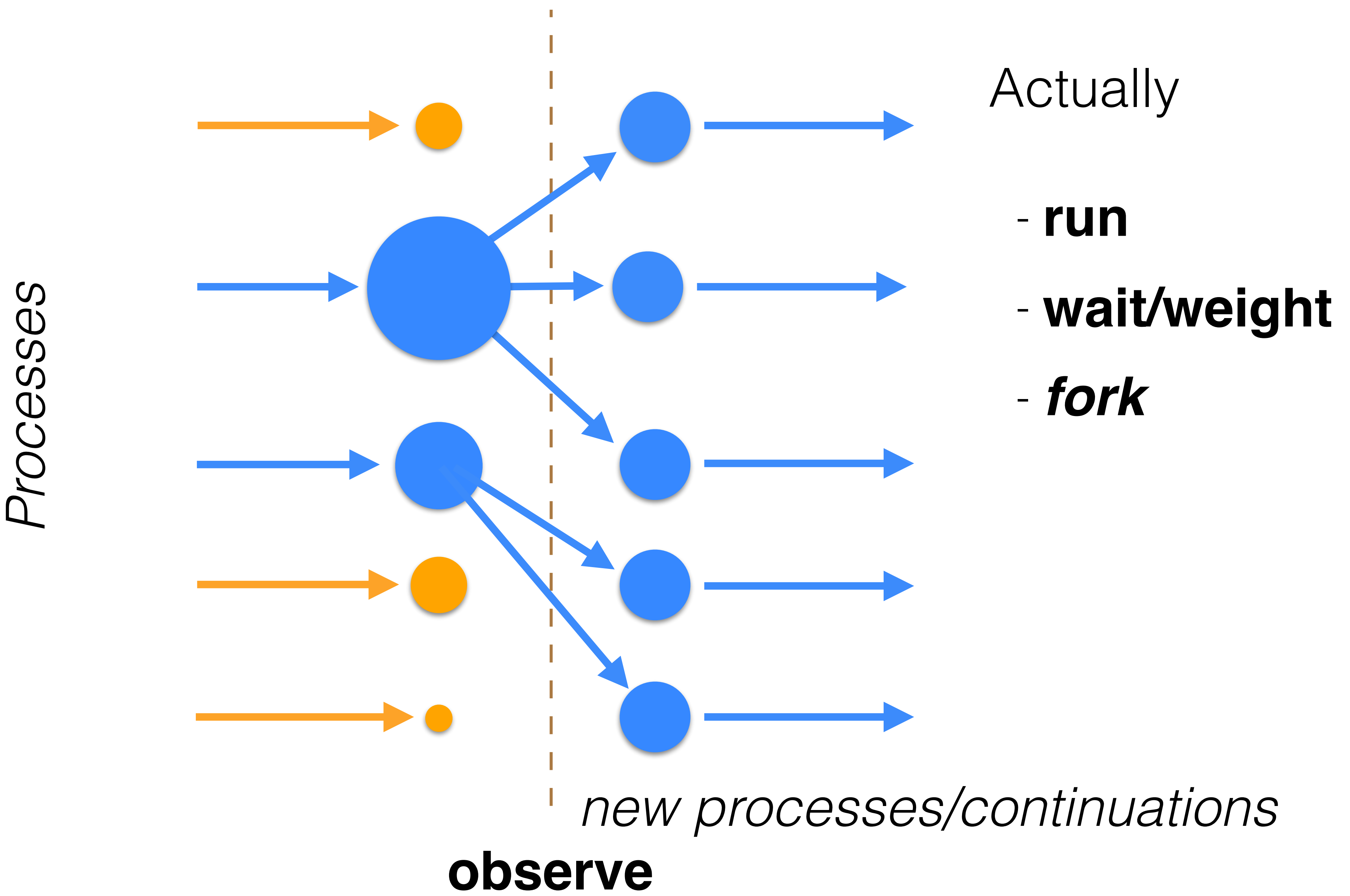
**`observe`** constrains program execution

**`predict`** emits sampled values

```
mean,8.013323
mean,8.013323
mean,6.132654
mean,7.229289
mean,7.027069
mean,7.194609
mean,7.194609
mean,5.218672
mean,6.184513
```

```
#include "probabilistic.h"

int main(int argc, char **argv) {

    double var = 2;
    double mu = normal_rng(1, 5);

    observe(normal_lnp(9, mu, var));
    observe(normal_lnp(8, mu, var));

    predict("mu,%f\n", mu);

    return 0;
}
```

# Probabilistic C Implementation




Paige and Wood "A Compilation Target for Probabilistic Programming Languages" ICML 2014

# Continuations

- A *continuation* is a function that encapsulates the "rest of the computation"
- A Continuation Passing Style (CPS) transformation rewrites programs so
    - no function ever returns
    - every function takes an extra argument, a function called the *continuation*
- Standard programming language technique
- No limitations

Friedman and Wand. "Essentials of programming languages." MIT press, 2008.
Fischer, Kiselyov, and Shan "Purely functional lazy non-deterministic programming" ACM Sigplan 2009
Goodman and Stuhlmüller http://dippl.org/ 2014
Tolpin https://bitbucket.org/probprog/anglican/ 2014

# Example CPS Transformation

```
;; Standard Clojure:
(println (+ (* 2 3) 4))

;; CPS transformed:
(*& 2 3 (fn [x] (+& x 4 println)))
```

Second cont.

First continuation

```
;; CPS-transformed "primitives"
(defn +& [a b k] (k (+ a b)))
(defn *& [a b k] (k (* a b)))
```

# CPS Explicitly Linearizes Execution

```
(defn pythag&
  "compute sqrt(x^2 + y^2)"
  [x y k]
  (square& x
           (fn [xx]
             (square& y
                      (fn [yy]
                        (+& xx yy
                            (fn [xxyy]
                              (sqrt& xxyy k))))))))
```

$$xx = x^2$$

$$yy = y^2$$

$$xxyy = xx + yy$$

$$\cdot = \sqrt{xxyy}$$

- Compiling to a pure language with lexical scoping ensures
  - A. variables needed in subsequent computation are bound in the environment
  - B. can't be modified by multiple calls to the continuation function

# Anglican Programs

```
(defquery flip-example [outcome]
  (let [p (sample (uniform-continuous 0 1))]
    (observe (flip p) outcome)
    (predict :p p)))
```

```
(let [u (uniform-continuous 0 1)

      p (sample u)

      dist (flip p)]

      (observe dist outcome)

      (predict :p p))
```

# Are “Compiled” to Native CPS-Clojure

```
(defn flip-query& [outcome k1]
  (uniform-continuous& 0 1
    (fn [dist1]
      (sample& dist1
        (fn [p] ((fn [p k2]
                   (flip& p
                     (fn [dist2]
                       (observe& dist2 outcome
                         (fn []
                           (predict& :p p k2))))))
                 p k1))))))
```

```
(let [u (uniform-continuous 0 1)
      p (sample u)
      dist (flip p)]
  (observe dist outcome)
  (predict :p p))
```

```
;; CPS-ed distribution constructors
(defn uniform-continuous& [a b k]
  (k (uniform-continuous a b)))

(defn flip& [p k]
  (k (flip p)))
```

Clojure

Anglican “linearized”

# Are “Compiled” to Native CPS-Clojure

```
(defn flip-query& [outcome k1]
  (uniform-continuous& 0 1          <---->  (let [u (uniform-continuous 0 1)
    (fn [dist1]
      (sample& dist1                <---->        p (sample u)
        (fn [p] ((fn [p k2]
                   (flip& p         <---->        dist (flip p)]
                     (fn [dist2]
                       (observe& dist2 outcome <----> (observe dist outcome)
                         (fn []
                           (predict& :p p k2)))))) <----> (predict :p p))
                 p k1))))))

;; CPS-ed distribution constructors
(defn uniform-continuous& [a b k]
  (k (uniform-continuous a b)))

(defn flip& [p k]
  (k (flip p)))
```

Clojure ↑ Anglican “linearized” ↑

# Explicit Functional Form for "Rest of Program"

```
(defn flip-query& [outcome k1]
  (uniform-continuous& 0 1
    (fn [dist1]
      (sample& dist1
        (fn [p] ((fn [p k2]
                   (flip& p
                     (fn [dist2]
                       (observe& dist2 outcome
                         (fn []
                           (predict& :p p k2))))))
                 p k1))))))
```

continuation functions

# Interruptible

```
(defn flip-query& [outcome k1]
  (uniform-continuous& 0 1
    (fn [dist1]
      (sample& dist1
        (fn [p] ((fn [p k2]
                   (flip& p
                     (fn [dist2]
                       (observe& dist2 outcome
                         (fn []
                           (predict& :p p k2))))))
                 p k1))))))
```

continuation functions

Anglican primitives

# Controllable

```
(defn flip-query& [outcome k1]
  (uniform-continuous& 0 1
    (fn [dist1]
      (sample& dist1
        (fn [p] ((fn [p k2]
                   (flip& p
                     (fn [dist2]
                       (observe& dist2 outcome
                         (fn []
                           (predict& :p p k2))))))
                 p k1))))))
```

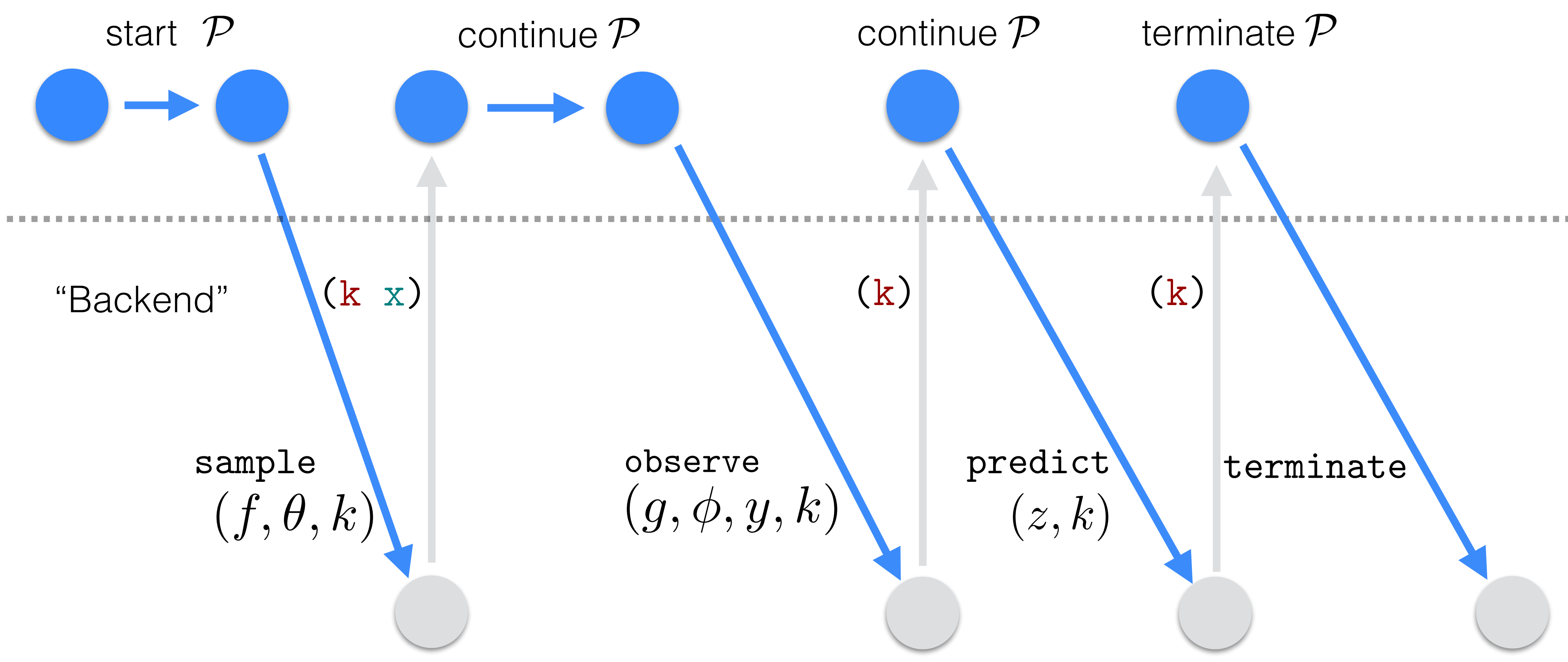


webPPL CPS compiles to pure functional Javascript

# Inference “Backend”

```
(defn sample& [dist k]
  ;; [ ALGORITHM-SPECIFIC IMPLEMENTATION HERE ]
  ;; Pass the sampled value to the continuation
  (k (sample dist)))

(defn observe& [dist value k]
  (println "log-weight =" (observe dist value))
  ;; [ ALGORITHM-SPECIFIC IMPLEMENTATION HERE ]
  ;; Call continuation with no arguments
  (k))

(defn predict& [label value k]
  ;; [ ALGORITHM-SPECIFIC IMPLEMENTATION HERE ]
  (k label value))
```

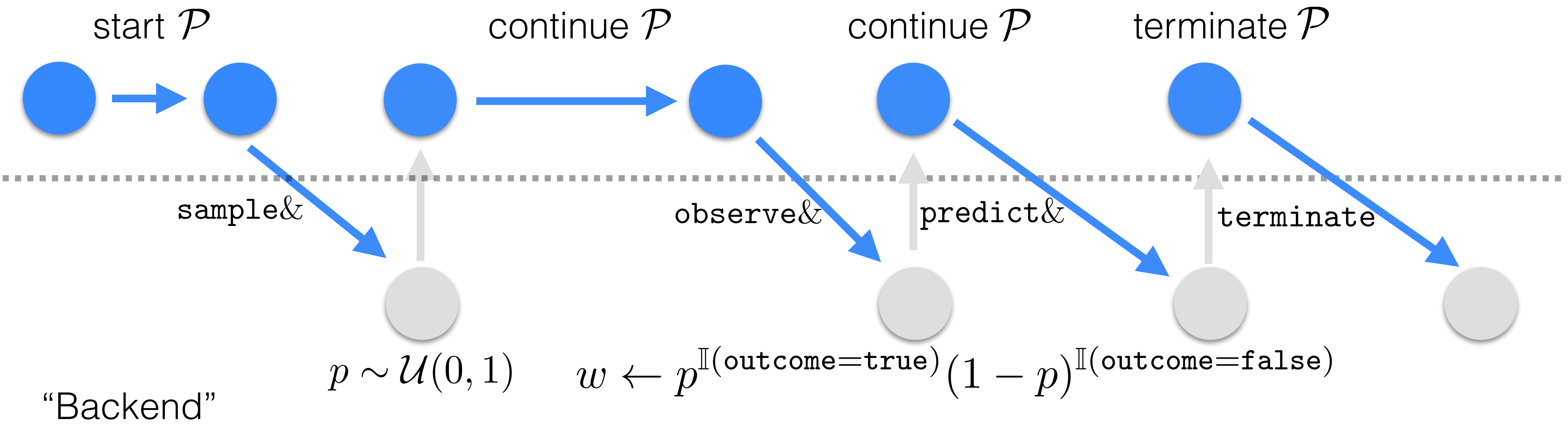
# Common Framework



# Likelihood Weighting "Backend"

```
(defn sample& [dist k]
  ;; Call the continuation with a sampled value
  (k (sample dist)))

(defn observe& [dist value k]
  ;; Compute and record the log weight
  (add-log-weight! (observe dist value))
  ;; Call the continuation with no arguments
  (k))

(defn predict& [label value k]
  ;; Store predict, and call continuation
  (store! label value)
  (k))
```

# Likelihood Weighting Example




```
(defquery flip-example [outcome]
  (let [p (sample (uniform-continuous 0 1))]
    (observe (flip p) outcome)
    (predict :p p))
```

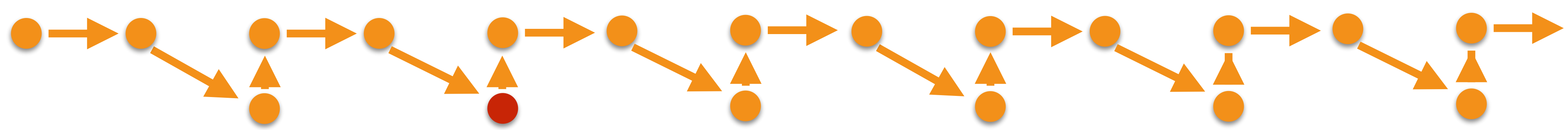
# SMC Backend

```
(defn sample& [dist k]
  ;; Call the continuation with a sampled value
  (k (sample dist)))

(defn observe& [dist value k]
  ;; Block and wait for K calls to reach observe&
  ;; Compute weights
  ;; Use weights to subselect continuations to call
  ;; Call K sampled continuations (often multiple times)
  )

(defn predict& [label value k]
  ;; Store predict, and call continuation
  (store! label value)
  (k))
```

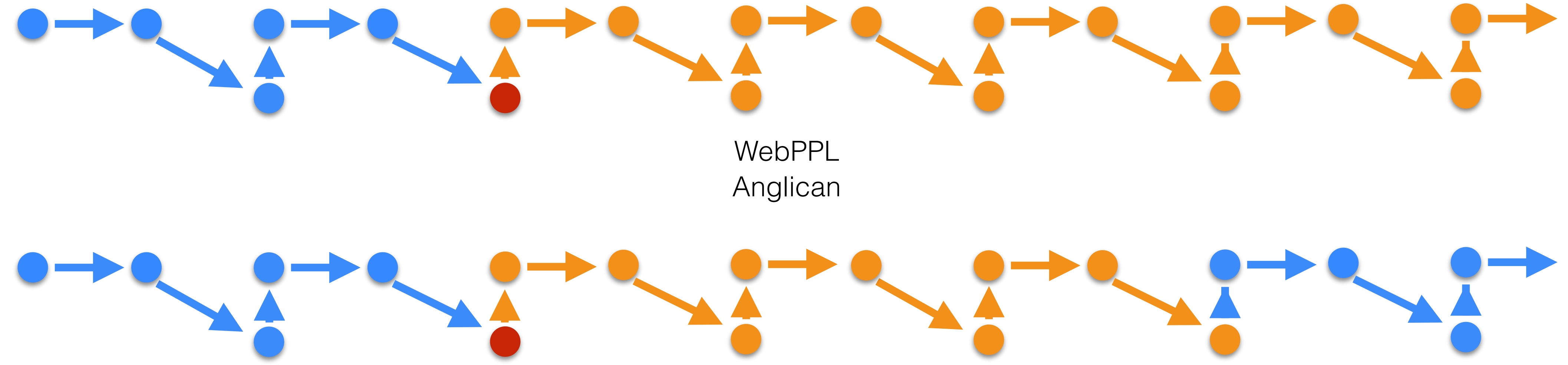
# LMH Backend

```
(defn sample& [a dist k]
  (let [;; reuse previous value,
        ;; or sample from prior
        x (or (get-cache a)
              (sample dist))]
    ;; add to log-weight when reused
    (when (get-cache a)
      (add-log-weight! (observe dist x)))
    ;; store value and its log prob in trace
    (store-in-trace! a x dist)
    ;; continue with value x
    (k x)))

(defn observe& [dist value k]
  ;; Compute and record the log weight
  (add-log-weight! (observe dist value))
  ;; Call the continuation with no arguments
  (k))
```

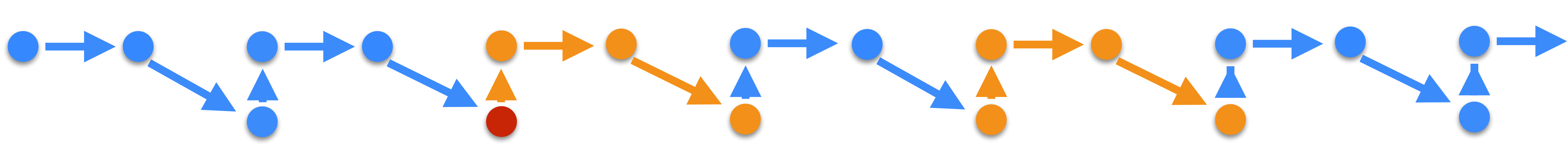
# LMH Variants



D. Wingate, A. Stuhlmueller, and N. D. Goodman.
"Lightweight implementations of probabilistic programming languages via transformational compilation." AISTATS (2011).




"C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching."
D. Ritchie, A. Stuhlmuller, and N. D. Goodman. arXiv:1509.02151 (2015).

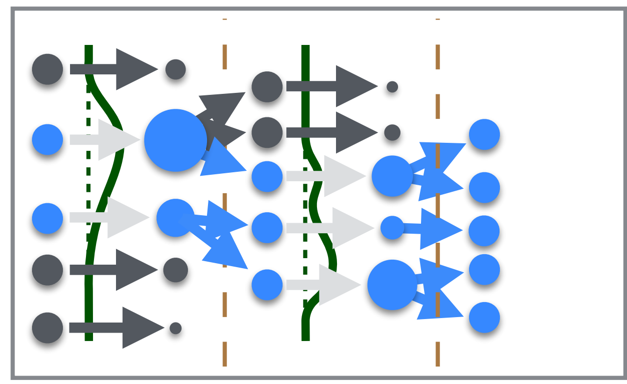


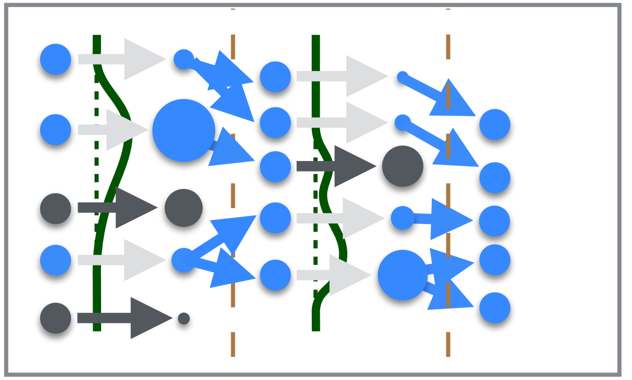
"Venture: a higher-order probabilistic programming platform with programmable inference."
V. Mansinghka, D. Selsam, and Y. Perov. arXiv:1404.0099 (2014).

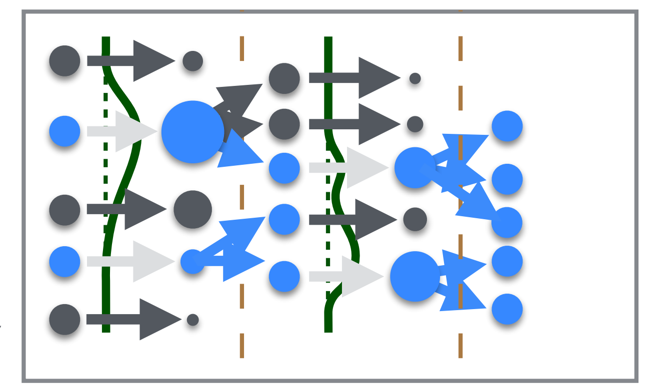
# Inference Improvements Relevant to in Higher-Order PPLs

# Add Hill Climbing

- PMCMC = MH with SMC proposals, e.g.
  - PIMH : “particle independent Metropolis-Hastings”
  - PGIBBS : “iterated conditional SMC”





Andrieu, Doucet, Holenstein “Particle Markov chain Monte Carlo methods.“ JRSSB 2010

# Blockwise Anytime Algorithm

- PIMH is MH that accepts entire new particle sets w.p.

$$\alpha^s_{PIMH} = \min\left(1, \frac{\hat{Z}^\star}{\hat{Z}^{s-1}}\right)$$

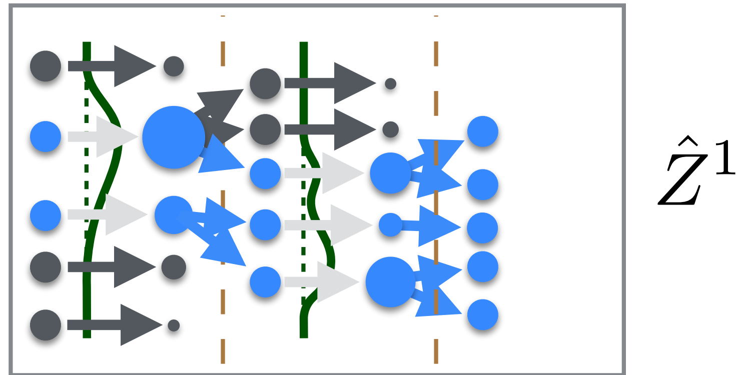
- Each SMC sweep computes marginal likelihood estimate

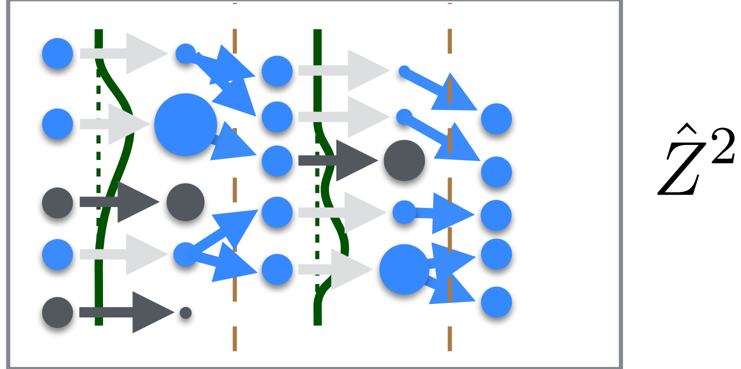
$$\hat{Z} = \prod_{n=1}^{N} \hat{Z}_n = \prod_{n=1}^{N} \frac{1}{K} \sum_{k=1}^{K} w(\tilde{\mathbf{x}}^k_{1:n})$$

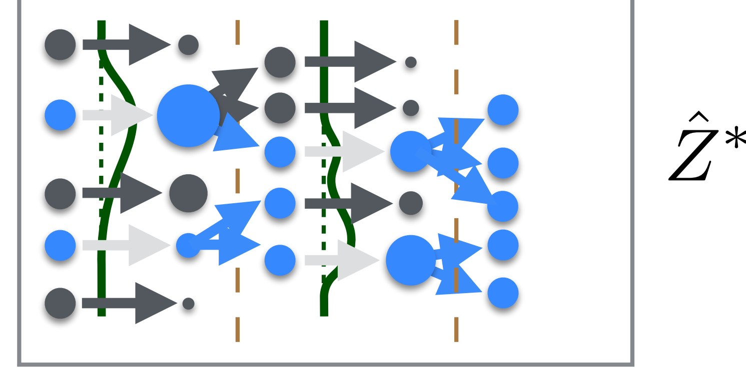
- And all particles can be used

$$\hat{\mathbb{E}}_{PIMH}[Q(\mathbf{x})] = \frac{1}{S} \sum_{s=1}^{S} \sum_{k=1}^{K} W^{s,k} Q(\mathbf{x}^{s,k})$$

Sweep

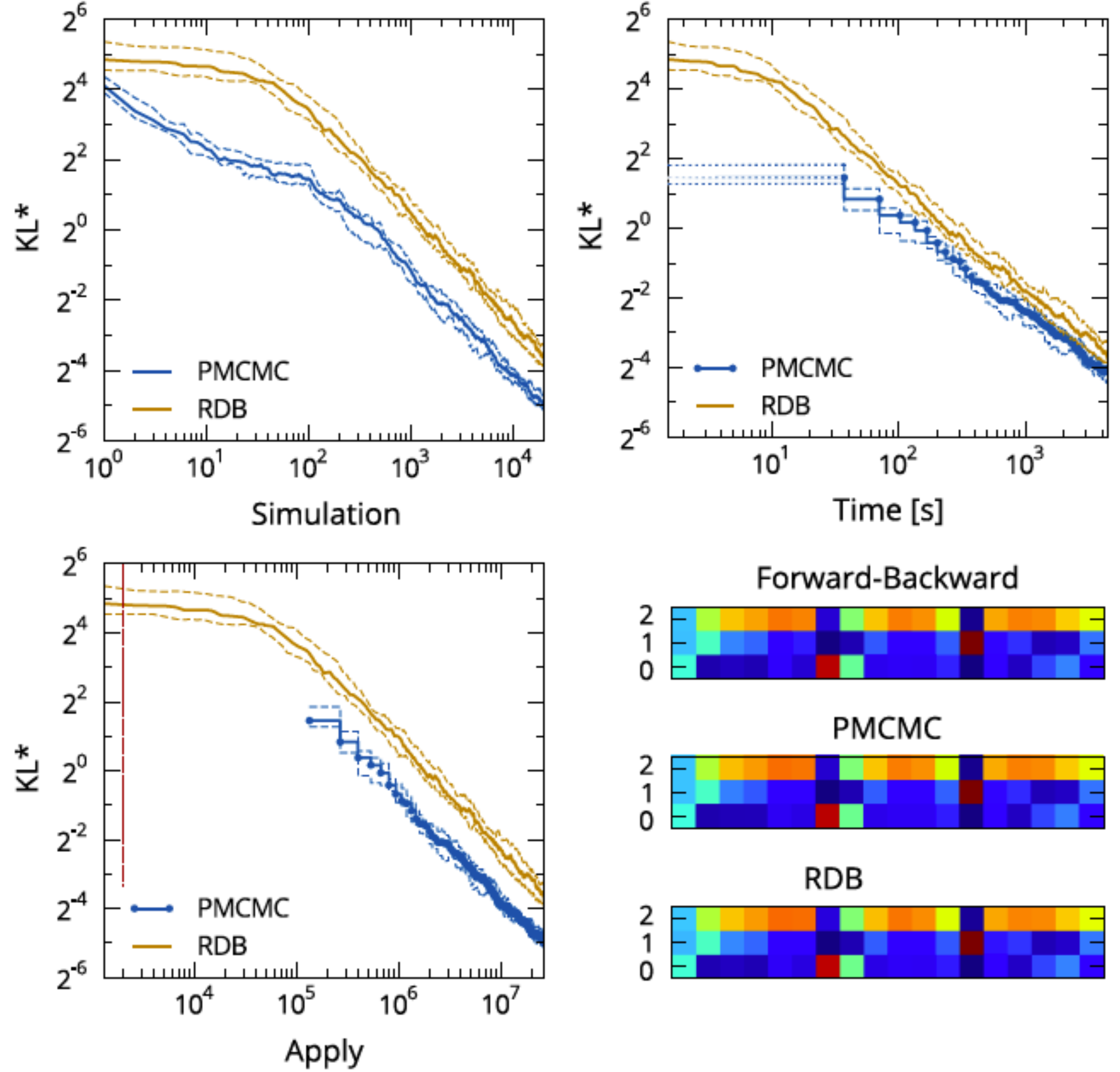



Paige and **Wood** "A Compilation Target for Probabilistic Programming Languages" ICML 2014

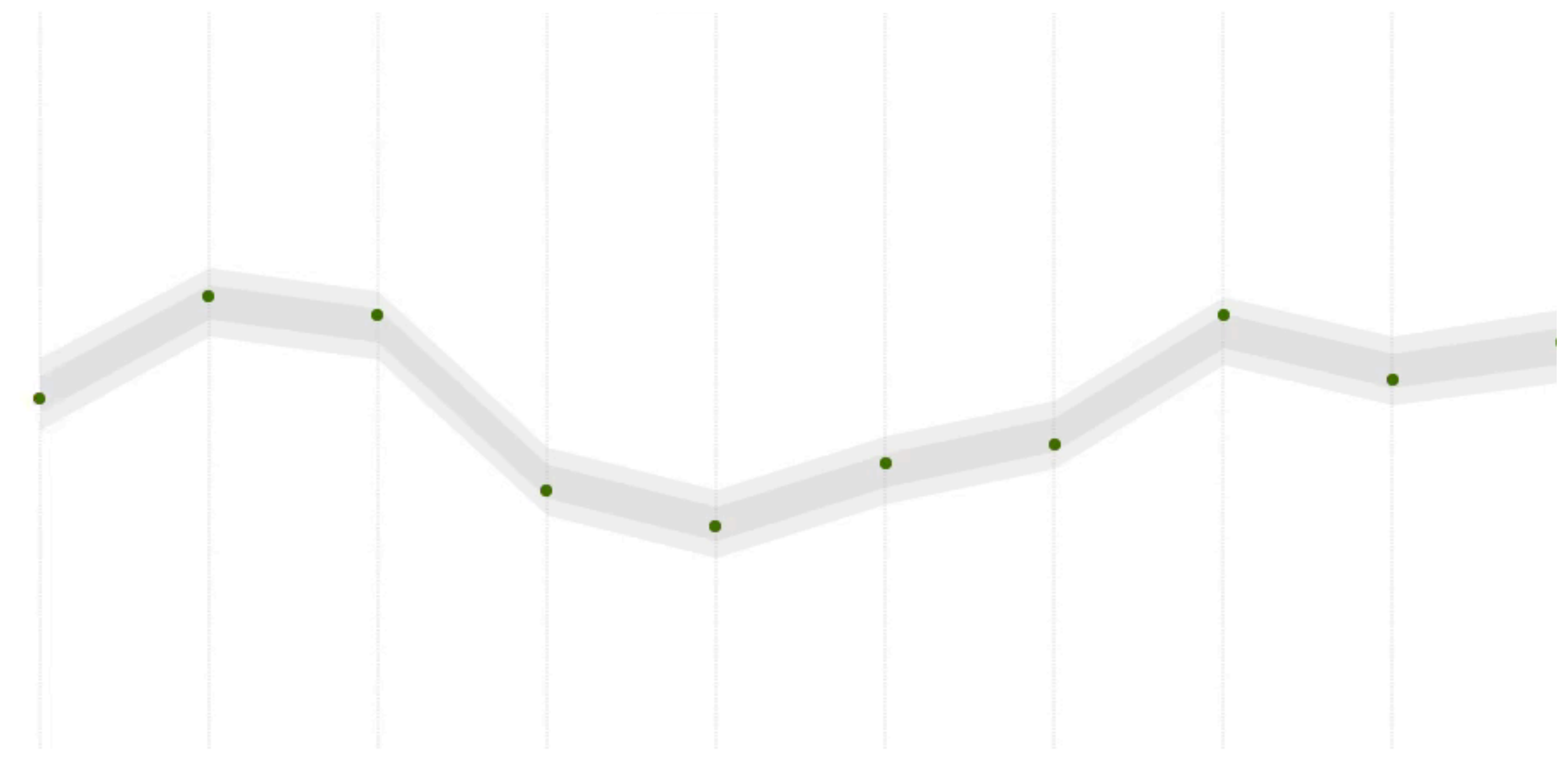
# PMCMC For Probabilistic Programming Inference





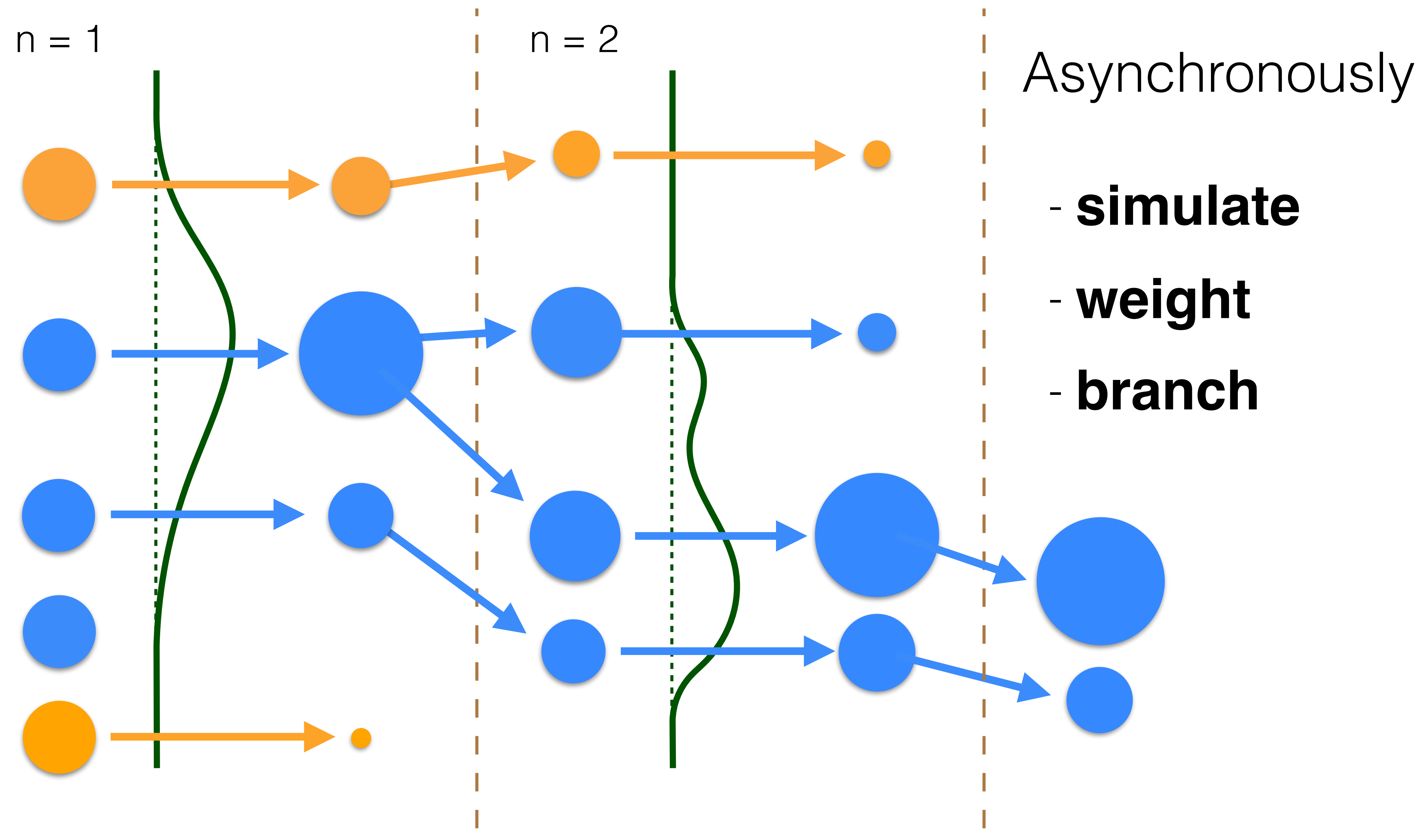

**Wood**, van de Meent, Mansinghka "A new approach to probabilistic programming inference" AISTATS 2014

# Remove Synchronization



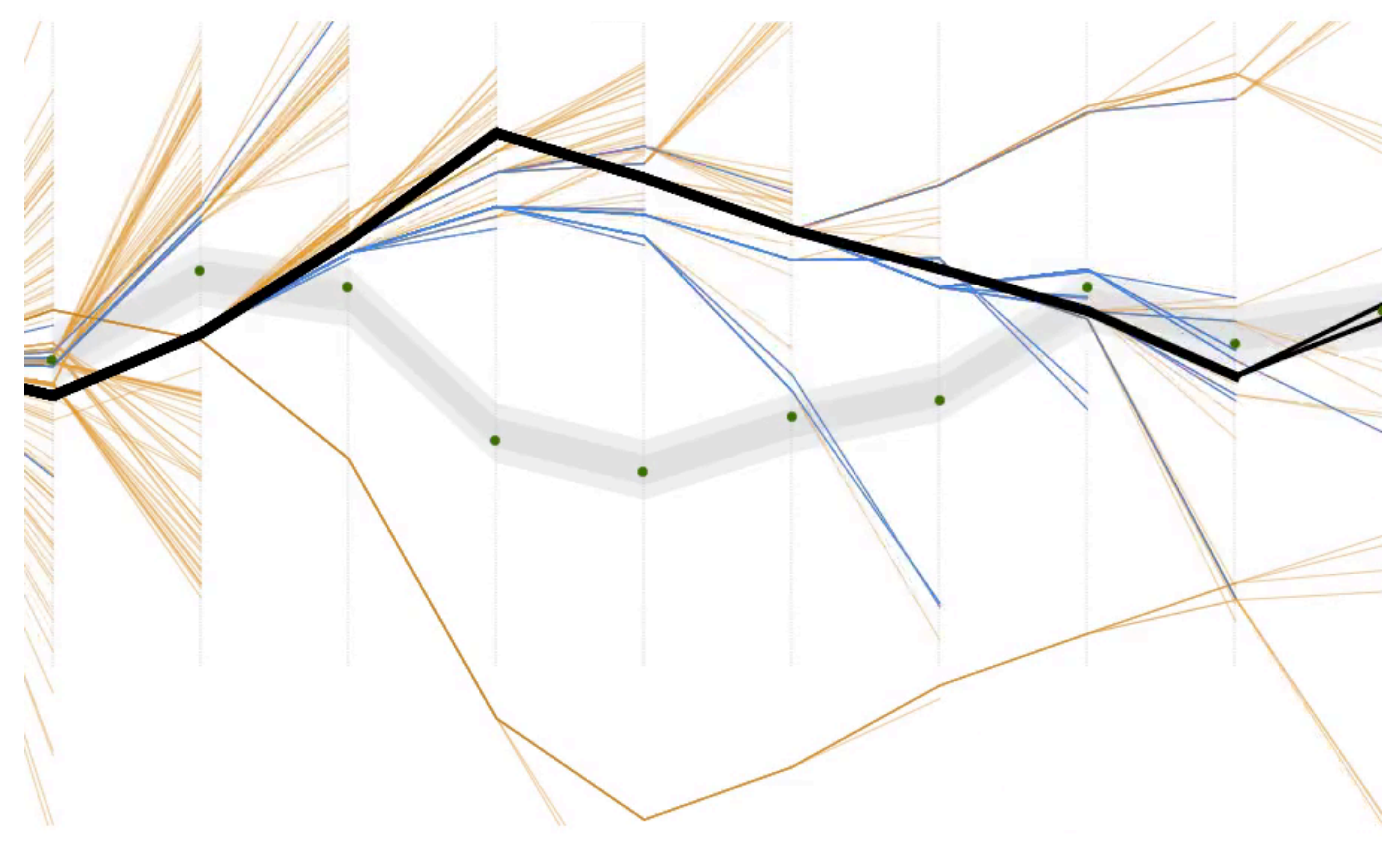
SMC in LDS slowed down for clarity

# Particle Cascade

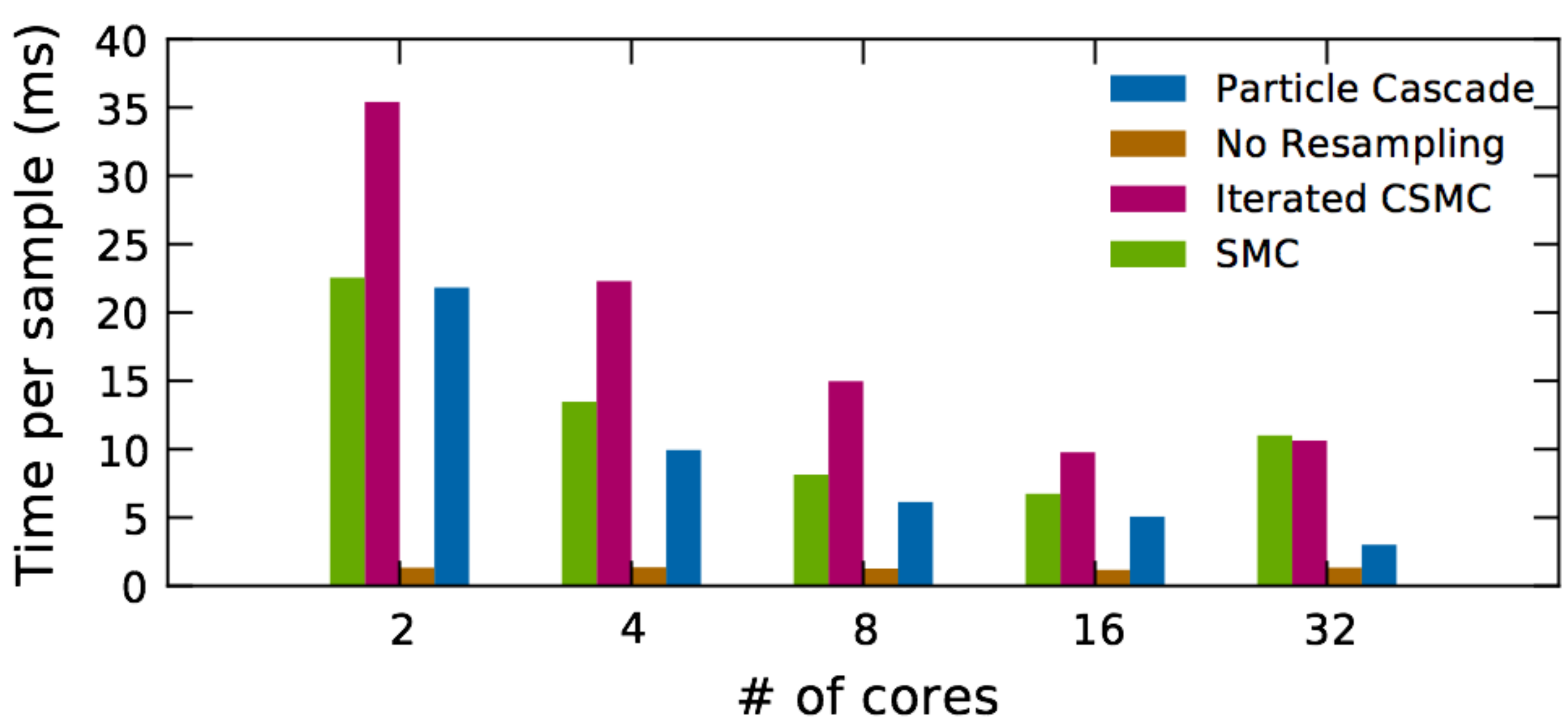



Paige, **Wood**, Doucet, Teh “Asynchronous Anytime Sequential Monte Carlo” NIPS 2014

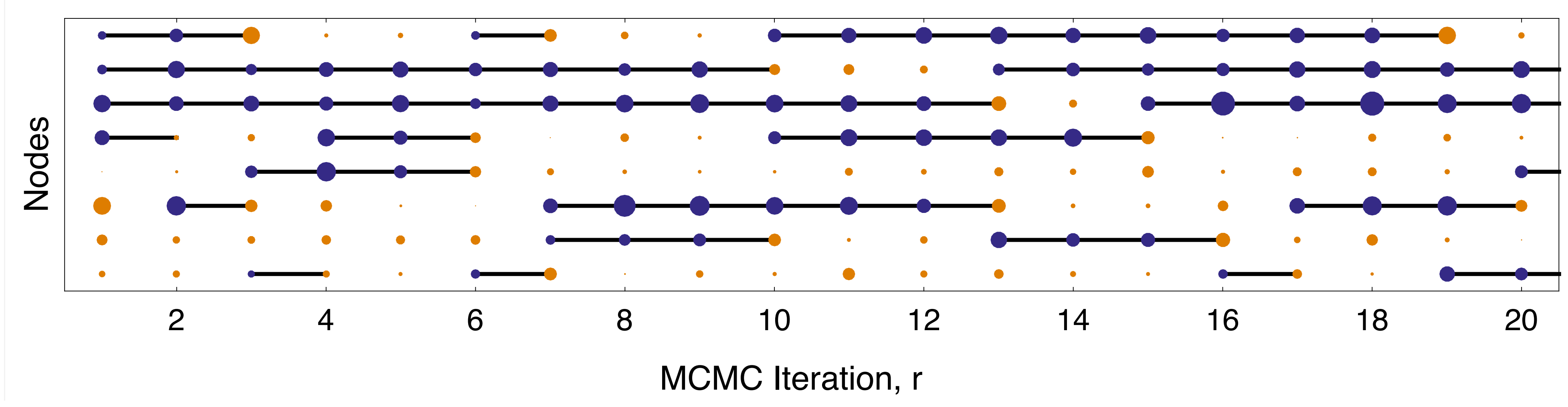
# Particle Cascade



# Shared Memory Scalability: Multiple Cores



# Distributed SMC




## iPMCMC

For each MCMC iteration $r = 1, 2, \ldots$

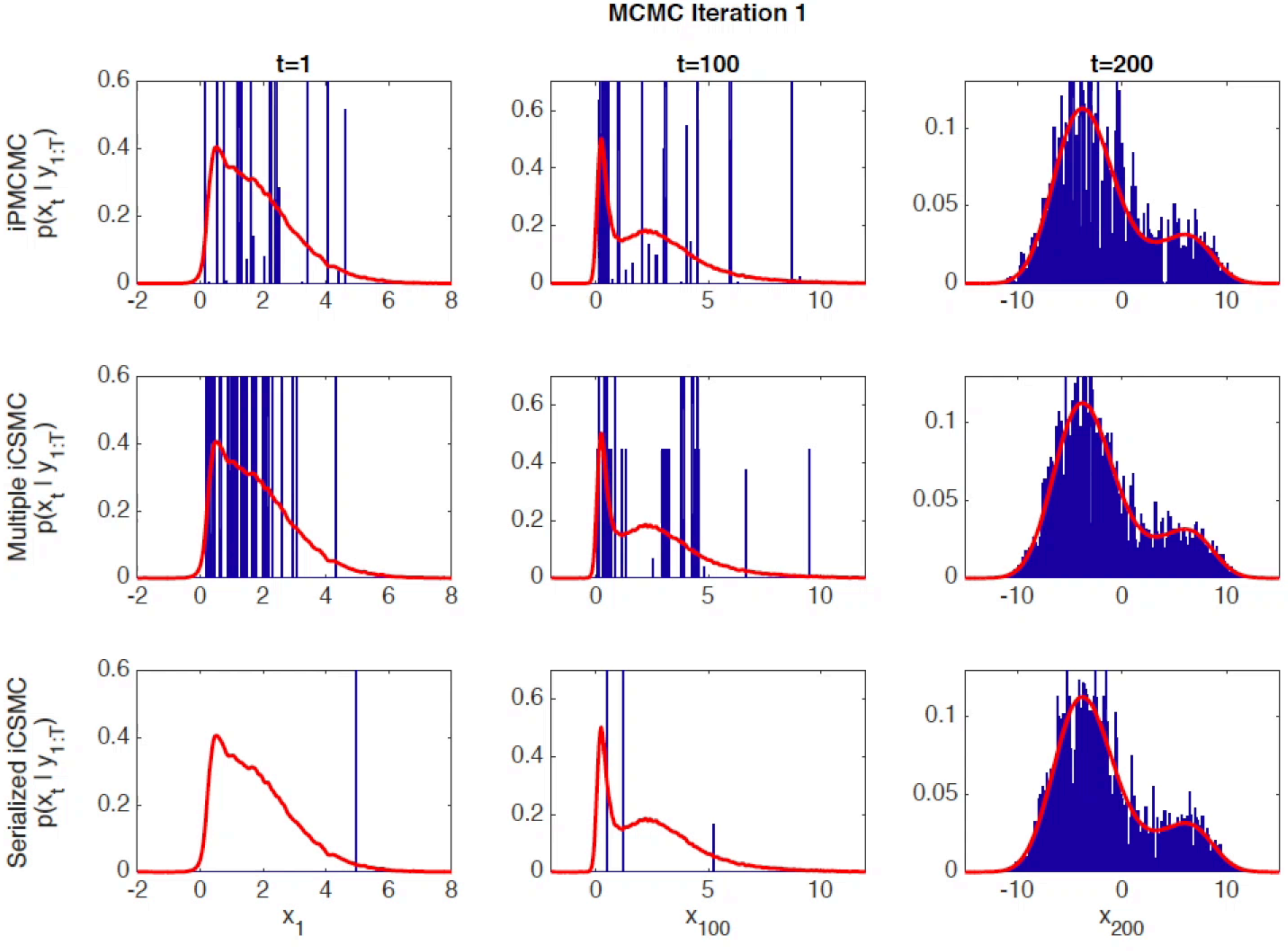
1. Nodes $c_j \in \{1, \ldots, M\}, \; j = 1, \ldots, P$ run CSMC, the rest run SMC
2. Each node $m$ returns a marginal likelihood estimate $\hat{Z}_m$ and candidate retained particle $x'_{1:T,m}$
3. A loop of Gibbs updates is applied to the retained particle indices

$$\mathbb{P}(c_j = m | c_{1:P \setminus j}) = \frac{\hat{Z}_m \mathbb{1}_{m \notin c_{1:P \setminus j}}}{\sum_{n=1}^{M} \hat{Z}_n \mathbb{1}_{n \notin c_{1:P \setminus j}}}$$

4. The retained particles for the next iteration are set $\mathbf{x}'_{1:T,j}[r] = x'_{1:T,c_j}$

Rainforth, Naesseth, Lindsten, Paige, van de Meent, Doucet, **Wood**, “Interacting Particle Markov Chain Monte Carlo” ICML 2016

# CSMC Exploitation / SMC Exploration



# Inference Backends in Anglican

- 14+ algorithms
- Average 165 lines of code per!
- Can implement and use without touching core code base.

| Algorithm | Type | Lines of Code | Citation | Description |
|---|---|---|---|---|
| smc | IS | 127 | Wood et al. AISTATS, 2014 | Sequential Monte Carlo |
| importance | IS | 21 | | Likelihood weighting |
| pcascade | IS | 176 | Paige et al., NIPS, 2014 | Particle cascade: Anytime asynchronous sequential Monte Carlo |
| pgibbs | PMCMC | 121 | Wood et al. AISTATS, 2014 | Particle Gibbs (iterated conditional SMC) |
| pimh | PMCMC | 68 | Wood et al. AISTATS, 2014 | Particle independent Metropolis-Hastings |
| pgas | PMCMC | 179 | van de Meent et al., AISTATS, 2015 | Particle Gibbs with ancestor sampling |
| lmh | MCMC | 177 | Wingate et al., AISTATS, 2011 | Lightweight Metropolis-Hastings |
| ipmcmc | MCMC | 193 | Rain forth et al., ICML, 2016 | Interacting PMCMC |
| almh | MCMC | 320 | Tolpin et al., ECML PKDD, 2015 | Adaptive scheduling lightweight Metropolis-Hastings |
| rmh* | MCMC | 319 | - | Random-walk Metropolis-Hastings |
| palmh | MCMC | 66 | - | Parallelised adaptive scheduling lightweight Metropolis-Hastings |
| plmh | MCMC | 62 | - | Parallelised lightweight Metropolis-Hastings |
| bamc | MAP | 318 | Tolpin et al., SoCS, 2015 | Bayesian Ascent Monte Carlo |
| siman | MAP | 193 | Tolpin et al., SoCS, 2015 | MAP estimation via simulated annealing |

# What Next?

# Commercial Impact

Assumptions

| Growth Type | 0 | 0 | 1 |
|---|---|---|---|
| *Linear* | | 0.5 | 0.5 |

Model

| | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 | 2016 | 2017 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Scenario | 0.4 | 1.5 | 3.2 | 6.1 | 10.6 | 16.6 | 21.8 | 23.7 | 29.3 | 37.9 | 50.18 | 55.5 | 65.7 | 76.9 | 76.0 | 85.0 |
| Actual | 0.4 | 1.5 | 3.2 | 6.1 | 10.6 | 16.6 | 21.8 | 23.7 | 29.3 | 37.9 | 50.2 | 55.5 | 65.7 | 74.5 | | |
| Analyst Concensus | | | | | | | | | | | | | | | 87.5 | 100.0 |

Google Revenue

Revenue $B's

Year

Scenario
Actual
Analyst Concensus

https://invrea.com/plugin/excel/v1/download/

# Symbolic Inference via Program Transformations

- Automated program transformations that simplify or eliminate inference (moving observes up and out)

```
(defquery beta-bernoulli [observation]
  (let [dist (beta 1 1)
        theta (sample dist)
        like (flip theta)]
    (observe like observation)
    (predict :theta theta)))
```

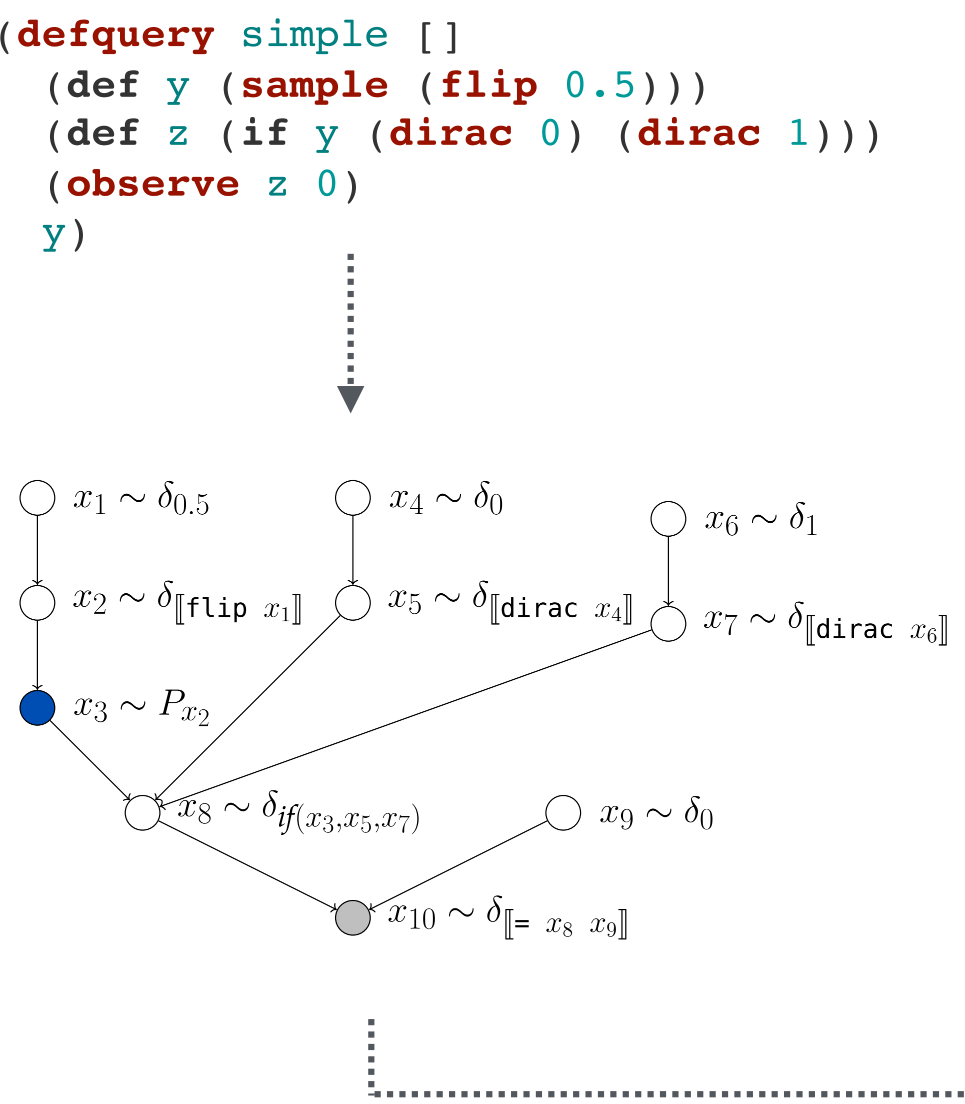
```
(defquery beta-bernoulli [observation]
  (let [dist (beta
               (if observation 2 1)
               (if observation 1 2))
        theta (sample dist)]
    (predict :theta theta)))
```

“Automatic Rao-Blackwellization”

Carette and Shan. “Simplifying Probabilistic Programs Using Computer Algebra⋆.” T.R. 719, Indiana University (2015)

# Exact Inference via Compilation

## Anglican




Figaro, etc.

variable elimination to compute

$$p(\mathbf{y})$$

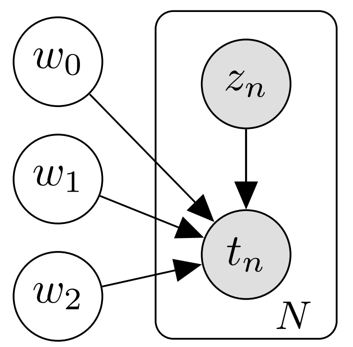
and

$$p(\mathbf{x}|\mathbf{y})$$

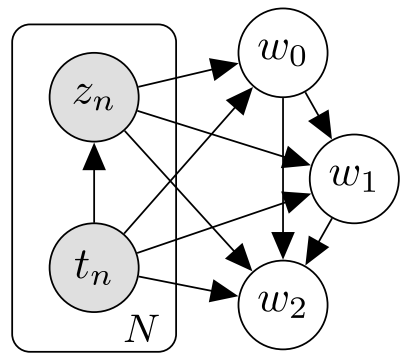
exactly

R Cornish, F **Wood**, and H Yang “Efficient exact inference in discrete Anglican programs” in prep. 2016

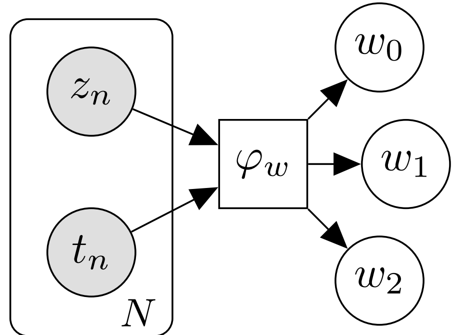
# Inference Compilation - FOPPLs




A probabilistic model




An inverse model generates latents




Can we learn how to sample from the inverse model?

Target density $\pi(\mathbf{x}) = p(\mathbf{x}|\mathbf{y})$, approximating family $q(\mathbf{x}|\lambda)$

Single dataset $\mathbf{y}$: $\underset{\lambda}{\operatorname{argmin}} D_{KL}(\pi||q_\lambda)$ ← fit λ to learn an importance sampling proposal

Averaging over all possible datasets:

$$\lambda = \varphi(\eta, \mathbf{y})$$

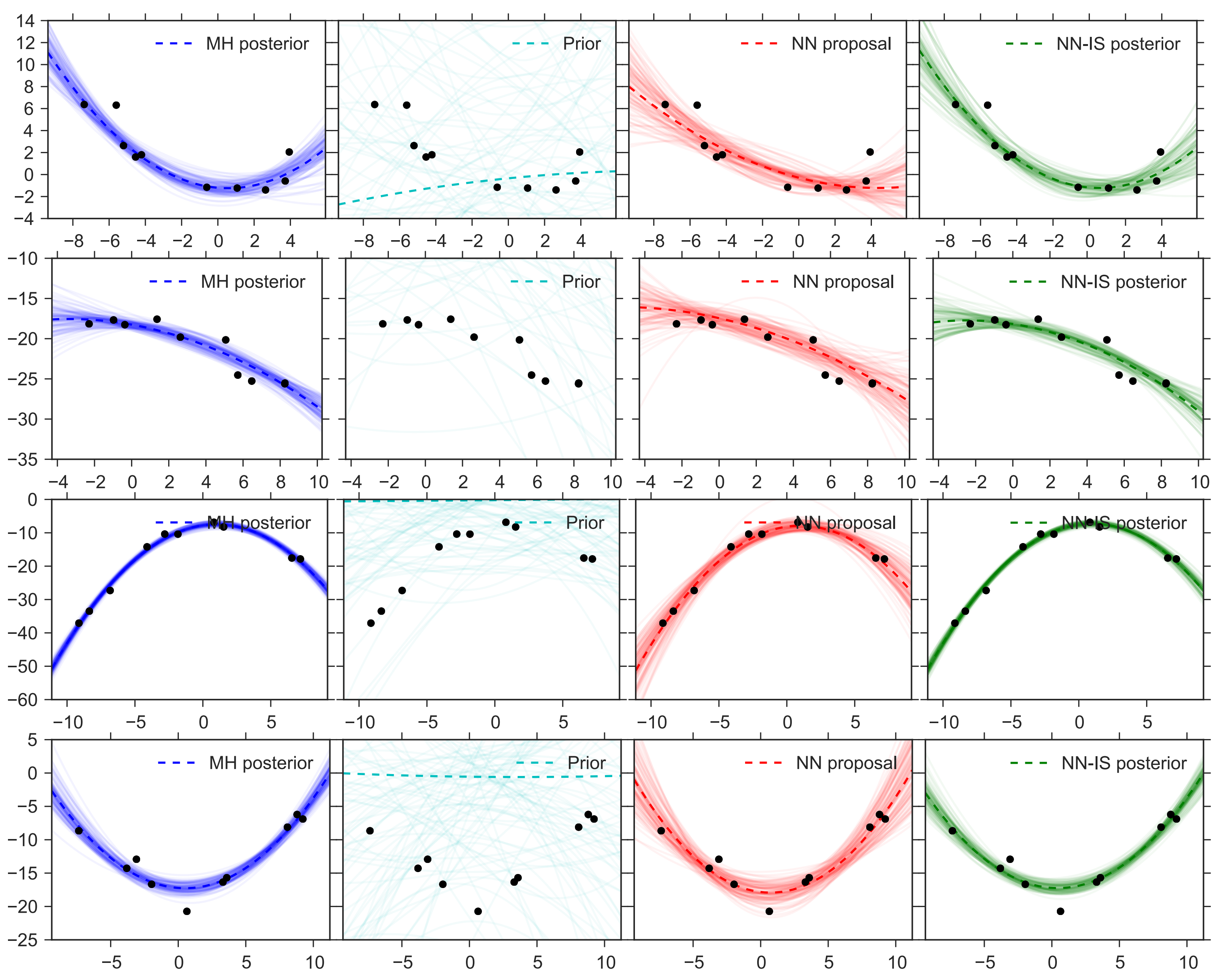
← learn a mapping from arbitrary datasets to λ

$$\underset{\eta}{\operatorname{argmin}} \mathbb{E}_{p(\mathbf{y})} \left[ D_{KL}(\pi||q_{\varphi(\eta,\mathbf{y})}) \right]$$

…compiles away runtime costs of inference!

Paige, **Wood** “Inference Networks for Sequential Monte Carlo in Graphical Models” ICML (2016).

# Compiled Inference Results




Paige, **Wood** “Inference Networks for Sequential Monte Carlo in Graphical Models” ICML (2016).

# Wrap Up

# Learning Dichotomy

## Supervised

$\mathbf{x}$

$\mathbf{y}$

+ data =

$\mathbf{x}$

$\mathbf{y}$

- Needs lots of labeled data
- Training is slow
- Uninterpretable model
- Fast at test time

## Unsupervised

$\mathbf{x}$

$\mathbf{y}$

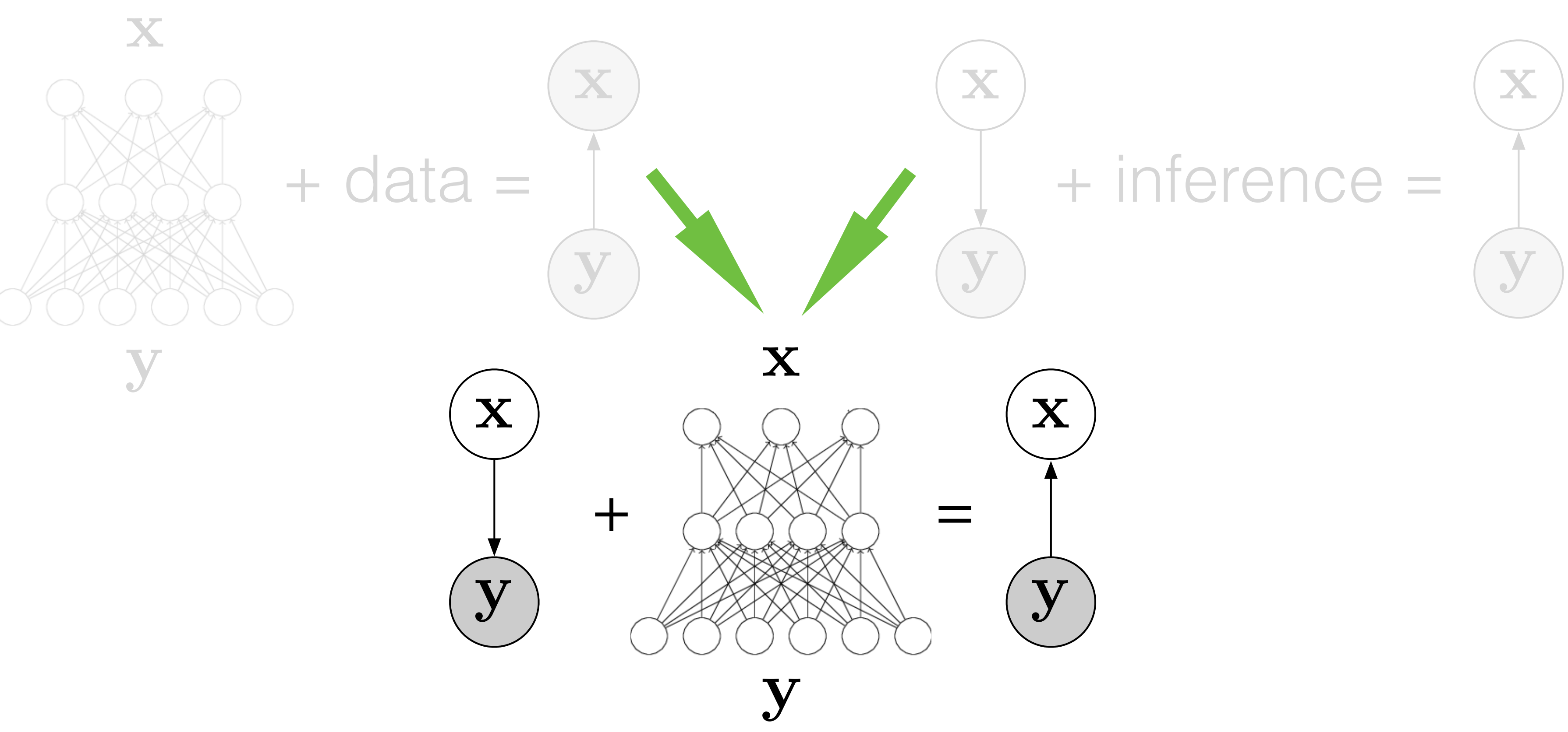
+ inference =

$\mathbf{x}$

$\mathbf{y}$

- Needs only unlabeled data
- No training
- Interpretable Model
- Slow at test time

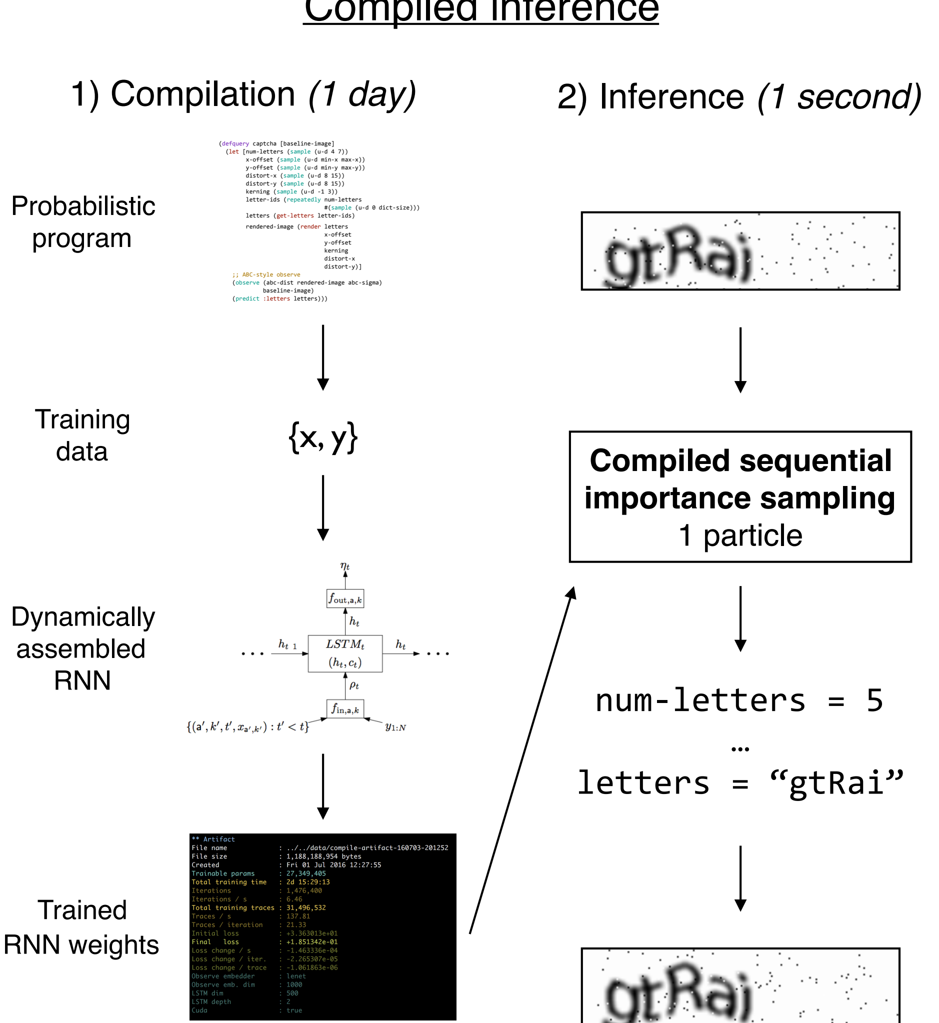
# Unified Learning

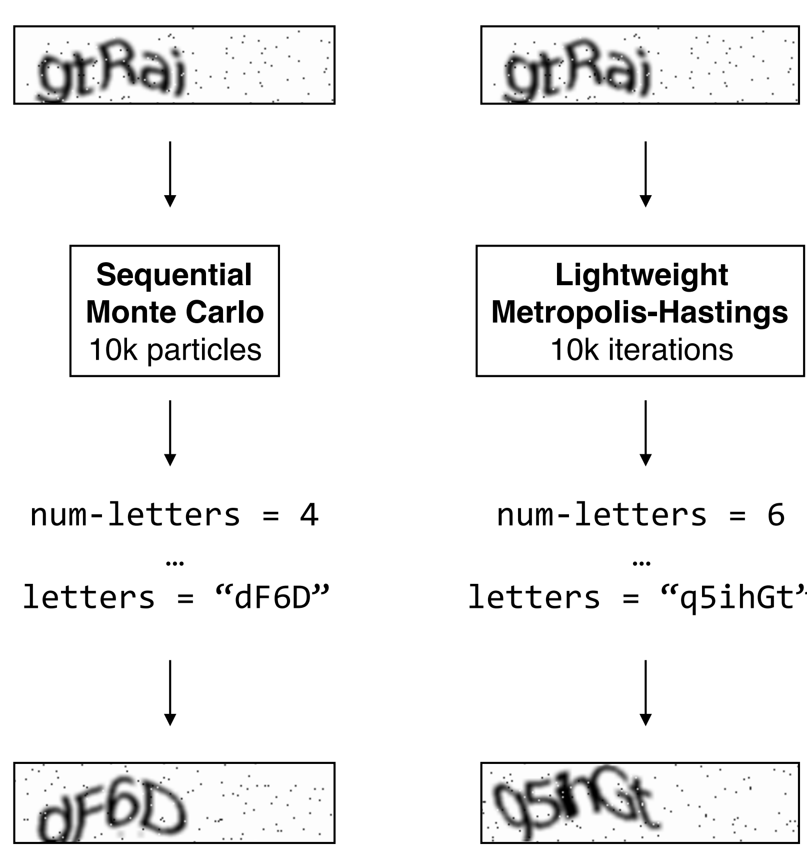



- Needs only unlabeled data
- Slow training
- Interpretable model
- Fast at test time

# HOPPL Compiled Inference

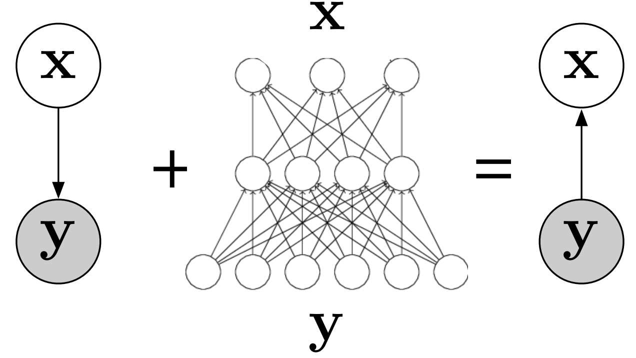
p(letters | captcha)




Le, Baydin, **Wood** "Inference Compilation and Universal Probabilistic Programming" in prep 2016

# Compiled HOPPL Models




| $\mathbf{x}$ | $\mathbf{y}$ |
|---|---|
| program source code | program output |
| scene description | image |
| policy and world | observations and rewards |
| neural net structures | input/output pairs |
| simulator | constraints |

# Wrap Up

# Where We Stand

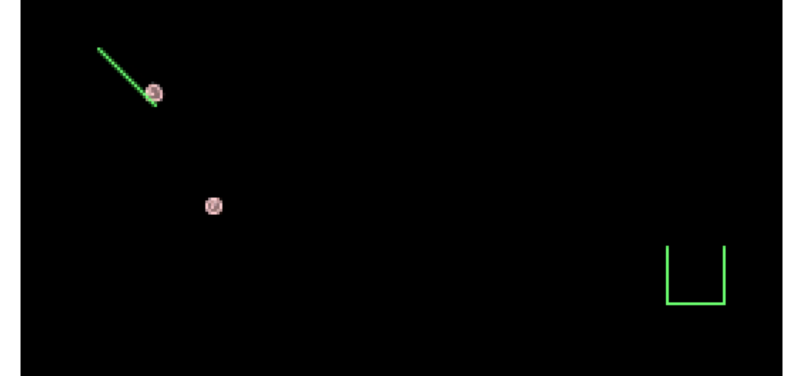
- Probabilistic programming concept
  - Long well established

- Tool maturity
  - Homework
  - Prototyping
  - Research
  - Advanced research
  - Small real-world applications

- Put-offs
  - Some highly optimized models that you know to scale well don’t necessarily scale well in current probabilistic programming systems.

# Deterministic Simulation and Other Libraries

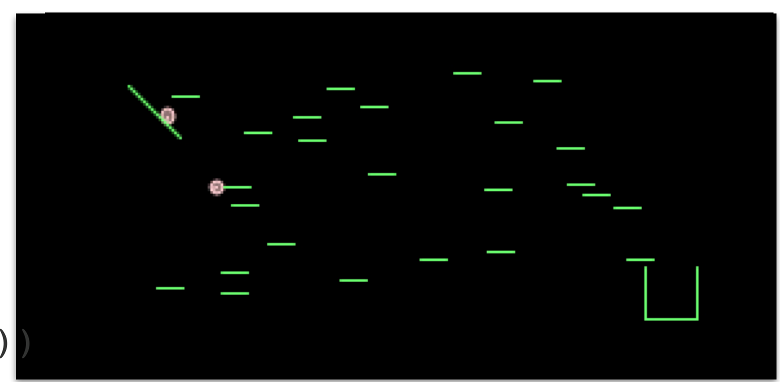
```
(defquery arrange-bumpers []
    (let [bumper-positions []

          ;; code to simulate the world
          world (create-world bumper-positions)
          end-world (simulate-world world)
          balls (:balls end-world)

          ;; how many balls entered the box?
          num-balls-in-box (balls-in-box end-world)]

      {:balls balls
       :num-balls-in-box num-balls-in-box
       :bumper-positions bumper-positions}))
```

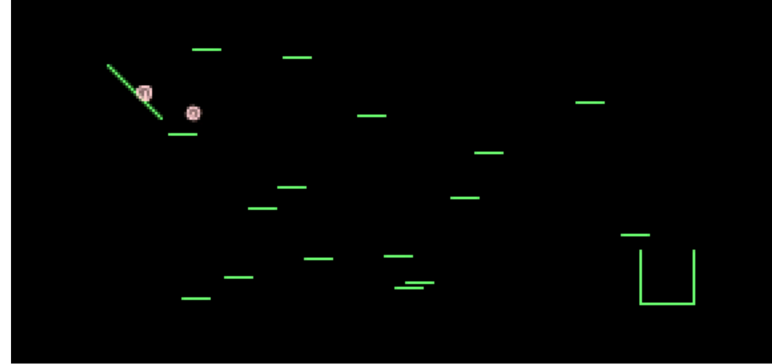


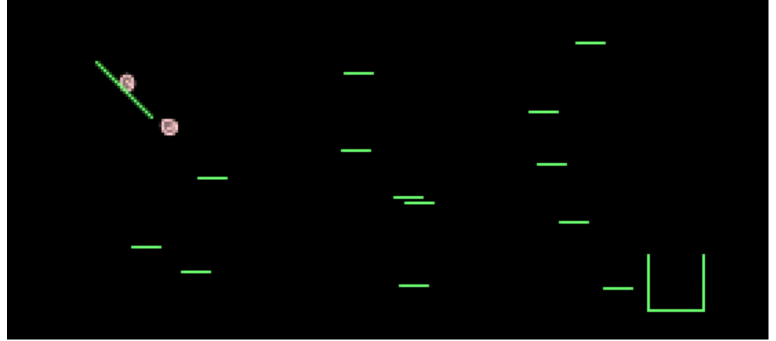
goal: “world” that puts ~20% of balls in box…

# Open Universe Models and Nonparametrics

```
(defquery arrange-bumpers []
    (let [number-of-bumpers (sample (poisson 20))
          bumpydist (uniform-continuous 0 10)
          bumpxdist (uniform-continuous -5 14)
          bumper-positions (repeatedly
                             number-of-bumpers
                             #(vector (sample bumpxdist)
                                      (sample bumpydist)))

          ;; code to simulate the world
          world (create-world bumper-positions)
          end-world (simulate-world world)
          balls (:balls end-world)

          ;; how many balls entered the box?
          num-balls-in-box (balls-in-box end-world)]

      {:balls balls
       :num-balls-in-box num-balls-in-box
       :bumper-positions bumper-positions}))
```

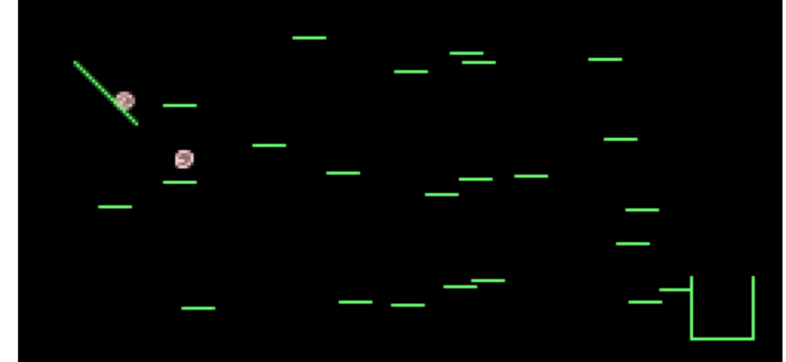


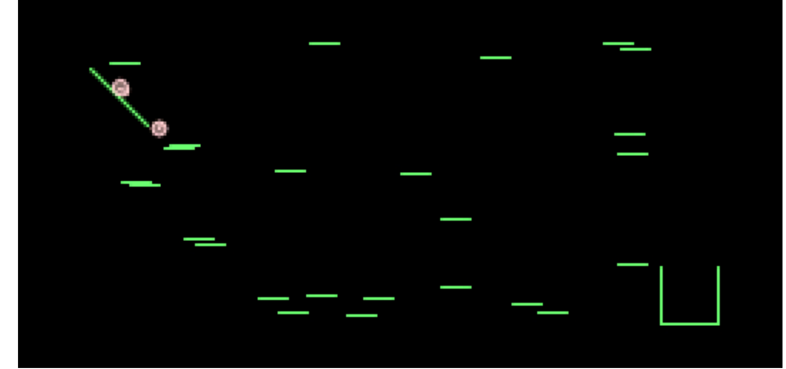


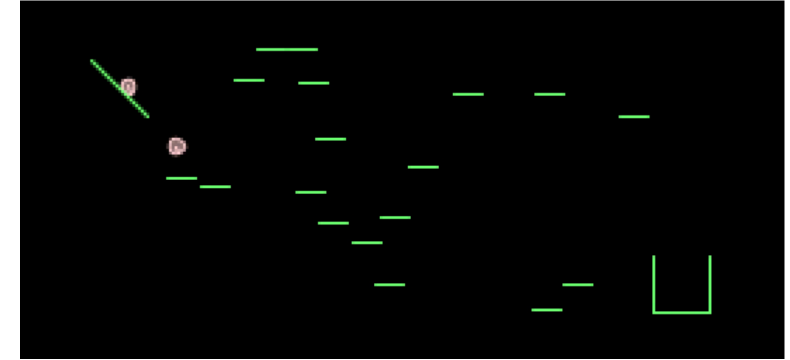


# Conditional (Stochastic) Simulation

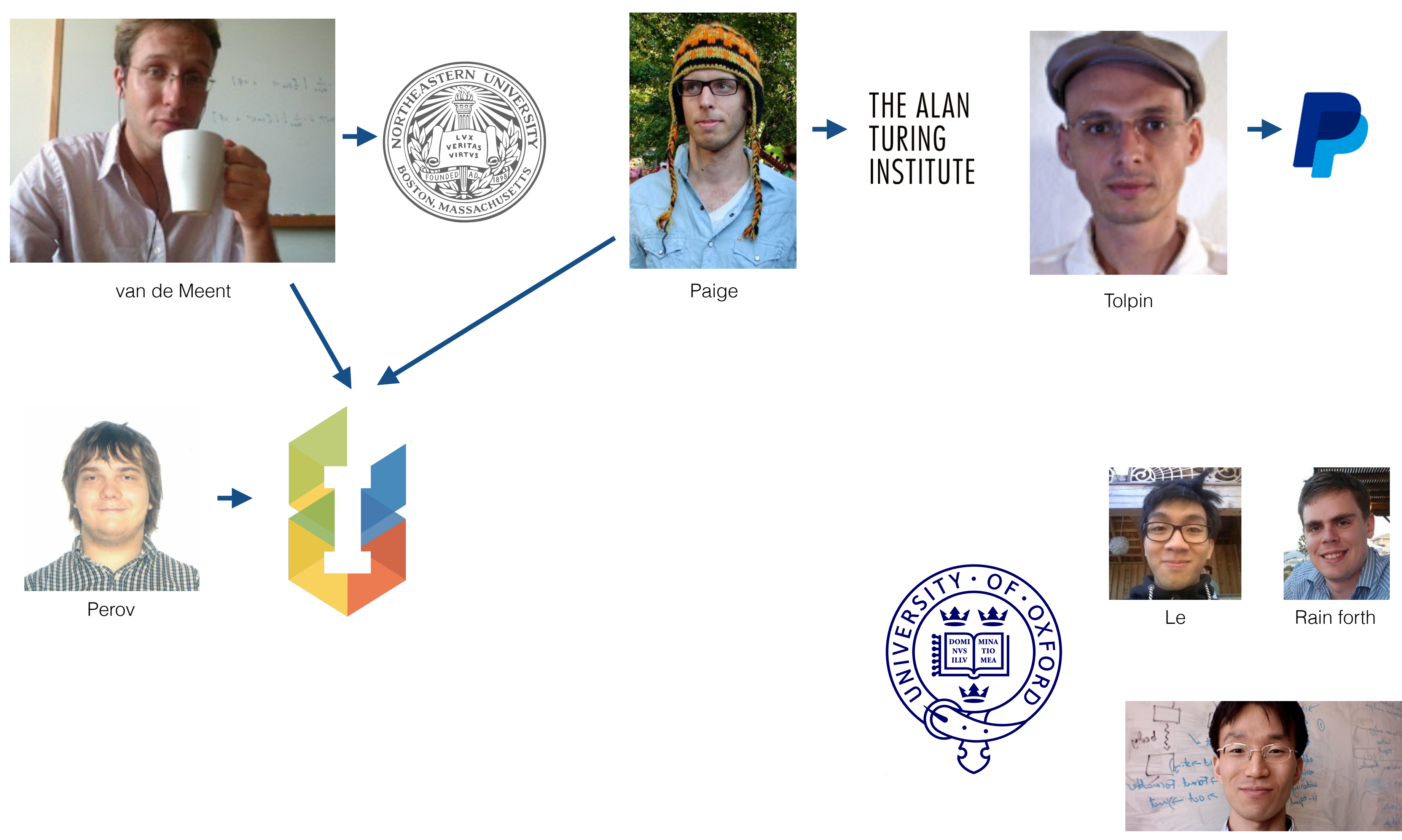
```
(defquery arrange-bumpers []
    (let [number-of-bumpers (sample (poisson 20))
          bumpydist (uniform-continuous 0 10)
          bumpxdist (uniform-continuous -5 14)
          bumper-positions (repeatedly
                             number-of-bumpers
                             #(vector (sample bumpxdist)
                                      (sample bumpydist)))

          ;; code to simulate the world
          world (create-world bumper-positions)
          end-world (simulate-world world)
          balls (:balls end-world)

          ;; how many balls entered the box?
          num-balls-in-box (balls-in-box end-world)

          obs-dist (normal 4 0.1)]

      (observe obs-dist num-balls-in-box)

      {:balls balls
       :num-balls-in-box num-balls-in-box
       :bumper-positions bumper-positions}))
```







# Thank You




- Funding : **DARPA**, BP, Amazon, Microsoft, Google

# Postdoc Openings

- 2 probabilistic programming postdoc openings

# Let's Go!  : Anglican Installation

https://goo.gl/US3b42