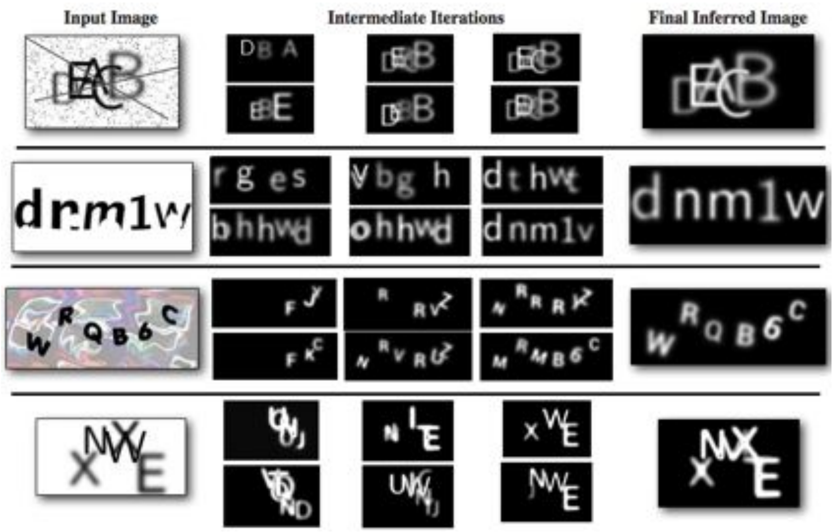
# Probabilistic Programming

Frank Wood
fwood@robots.ox.ac.uk

AGI 2015

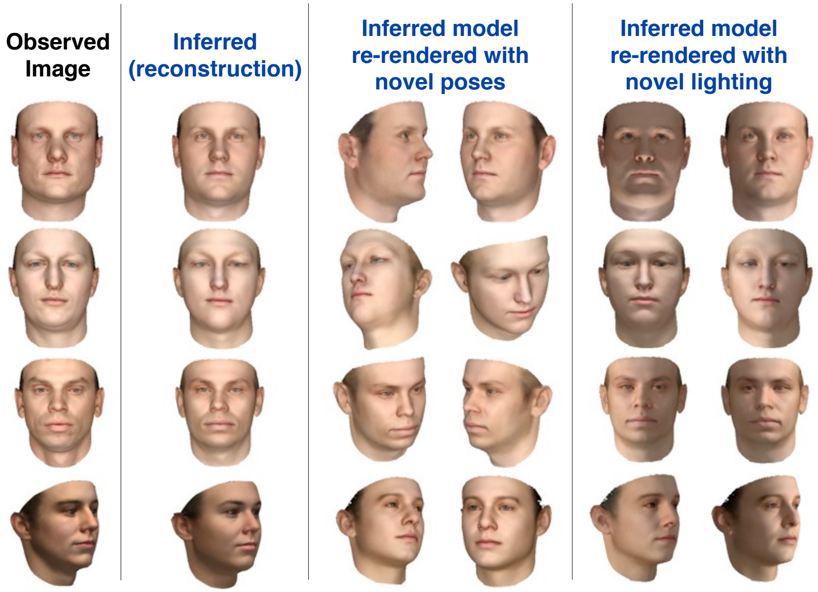
# Inverse Graphics

## Captcha Solving




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

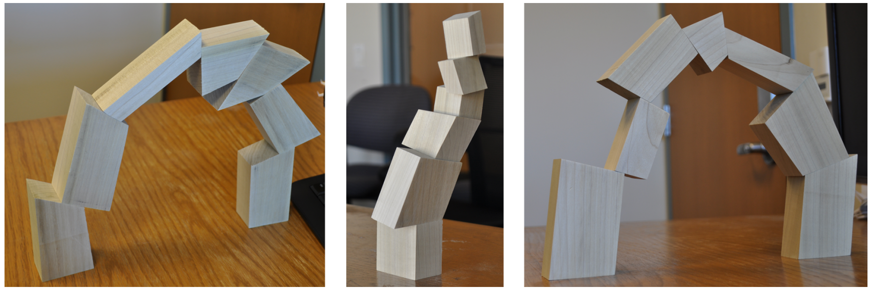
## Mesh Fitting




T. Kulkarni, et al "Picture: a probabilistic programming language for scene perception." CVPR (2015).

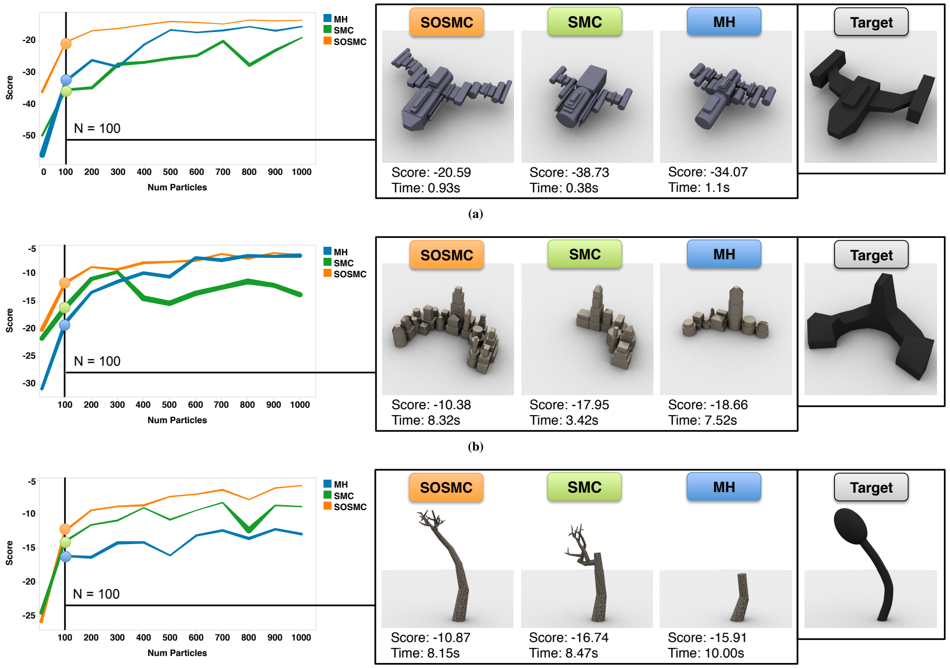
# Directed Design

## Stable Static Structures



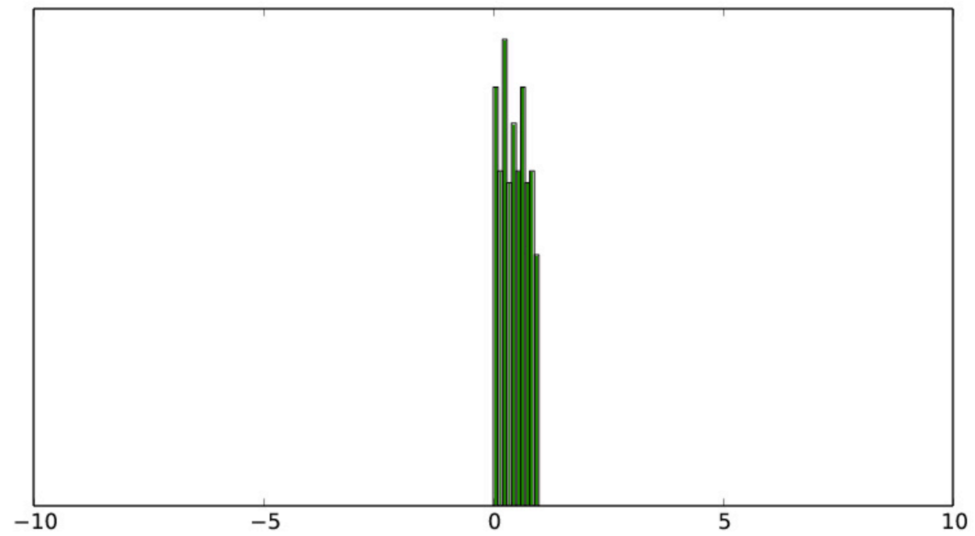
Ritchie, D., Lin, S., Goodman, N. D., & Hanrahan, P.
Generating Design Suggestions under Tight Constraints
with Gradient-based Probabilistic Programming.
In Computer Graphics Forum, (2015)

## Procedural Graphics

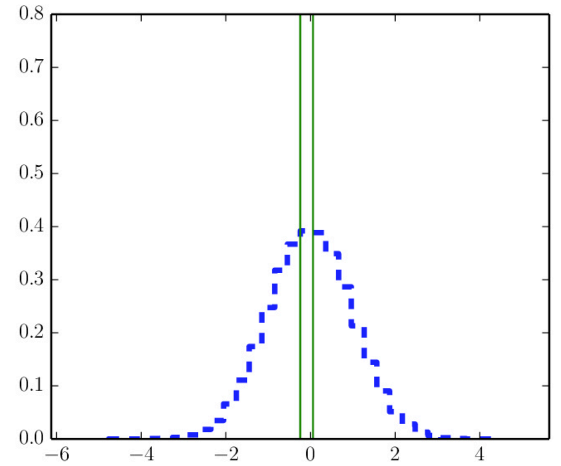


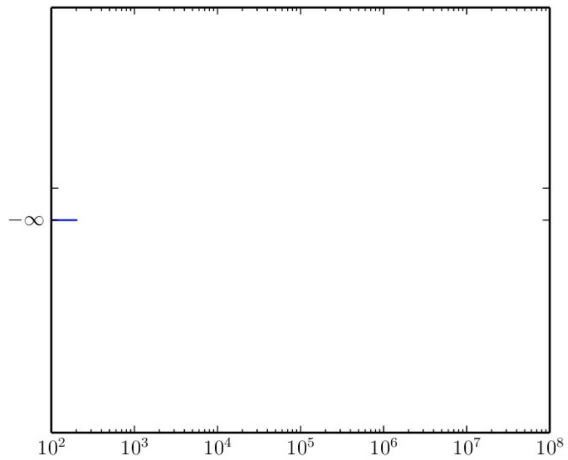

D. Ritchie, B. Mildenhall, N. D. Goodman, & P. Hanrahan.
"Controlling Procedural Modeling Programs with
Stochastically-Ordered Sequential Monte Carlo."
SIGGRAPH (2015)

# Probabilistic Program Induction



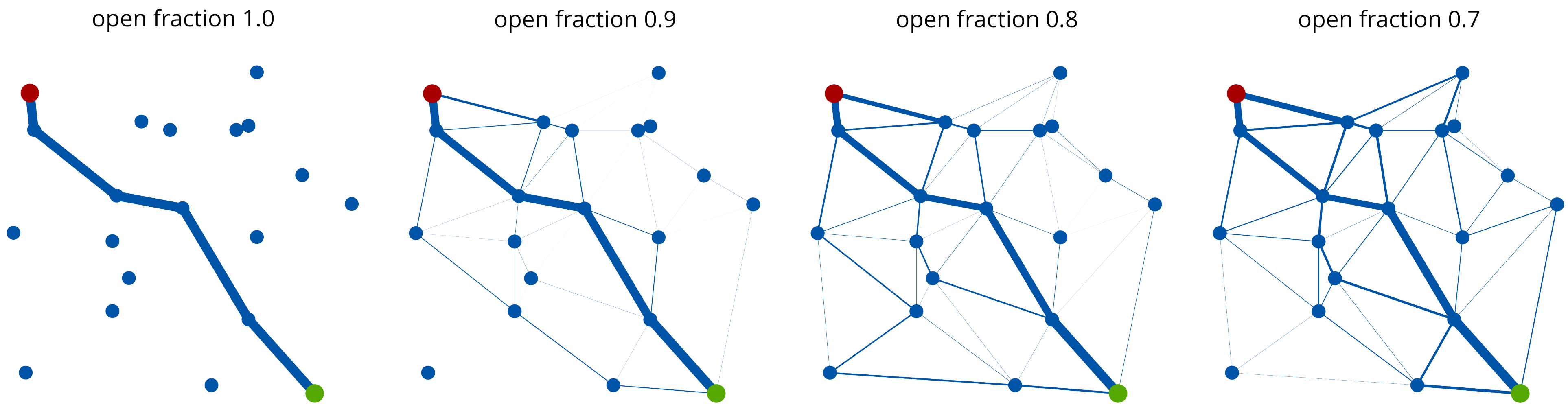

```
(lambda (stack-id) (safe-uc (* (if (< 0.0 (* (* (* -1.0 (begin (define
 G__1147 (safe-uc 1.0 1.0)) 0.0)) (* 0.0 (+ 0.0 (safe-uc (* (* (dec -2
.0) (safe-sqrt (begin (define G__1148 3.14159) (safe-log -1.0)))) 2.0)
 0.0)))) 1.0)) (+ (safe-div (begin (define G__1149 (* (+ 3.14159 -1.0)
 1.0)) 1.0) 0.0)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



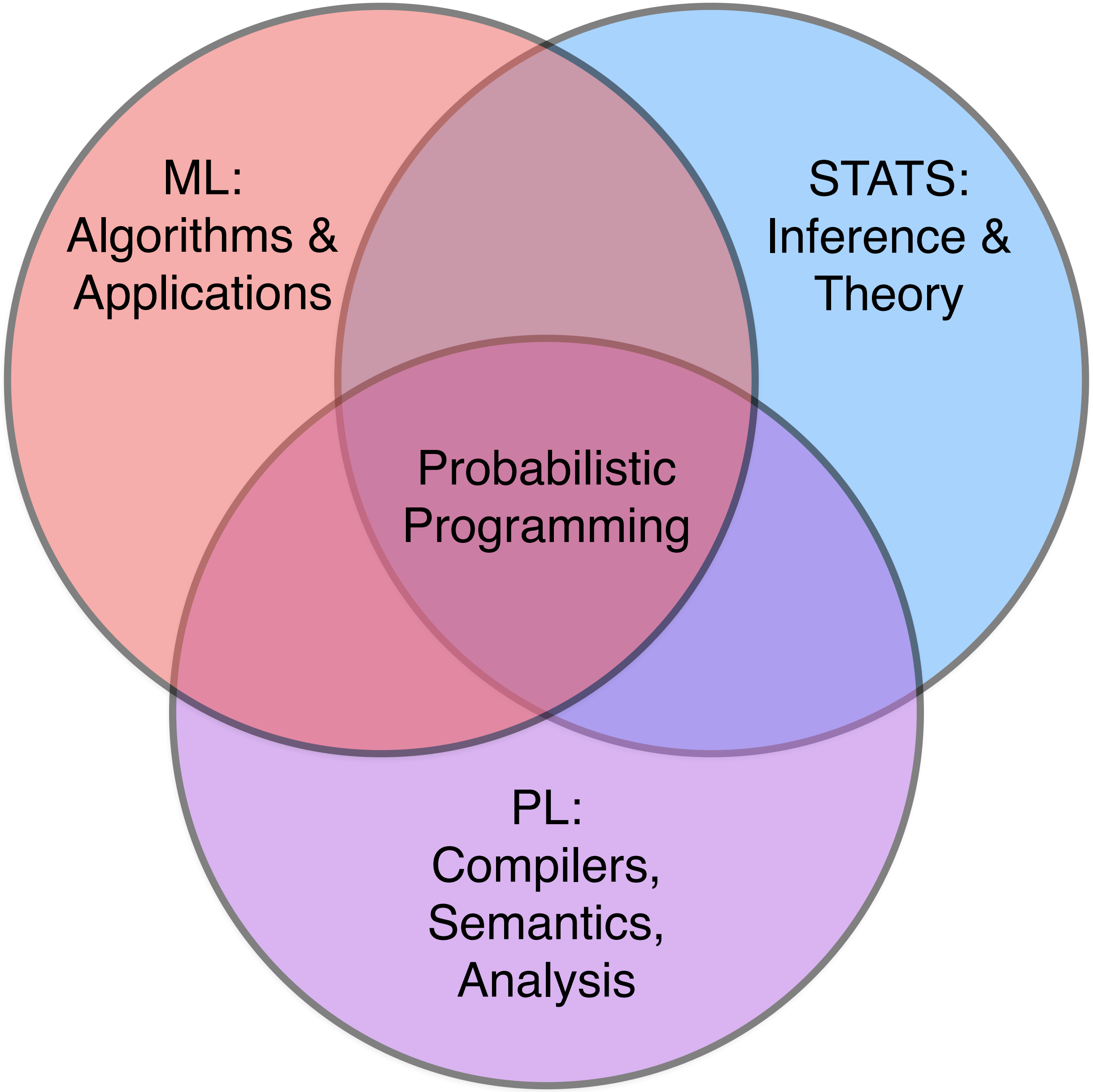

Yura Perov and Frank  Wood.
"Learning Probabilistic Programs."
arXiv preprint arXiv:1407.2646 (2014).

# Policy Learning in POMDPs

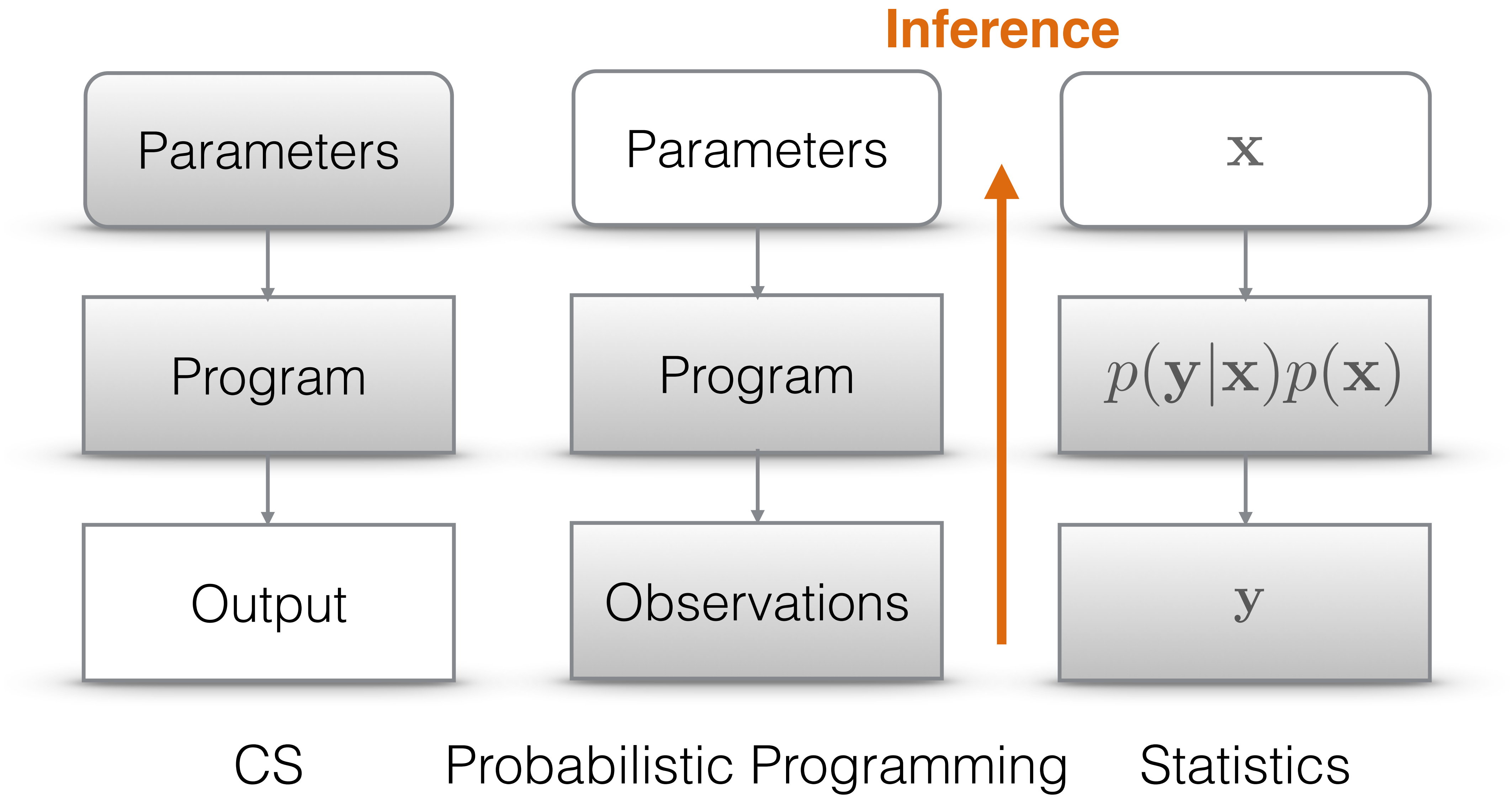



Jan-Willem van de Meent, David Tolpin, Brooks Paige, and Frank Wood.
"Black-Box Policy Search with Probabilistic Programs."
arXiv preprint arXiv:1507.04635 [under NIPS review] (2015).

# Landscape


ML:
Algorithms &
Applications
STATS:
Inference &
Theory
Probabilistic
Programming
PL:
Compilers,
Semantics,
Analysis

# Conceptualization



# Operative Definition

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

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

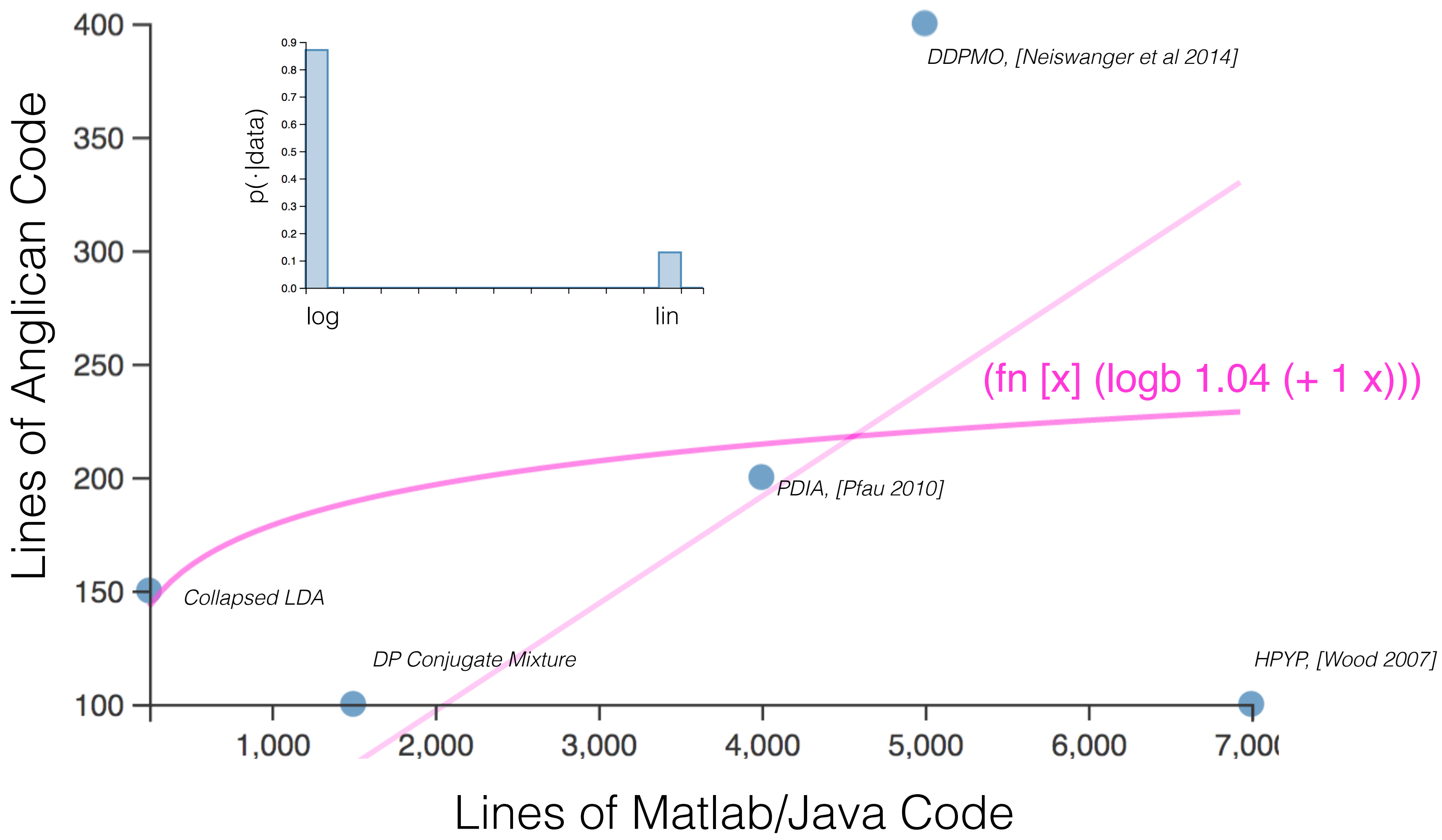
(2) the ability to condition values of variables in a program via observations.”

Gordon et al, 2014

# What are the goals of probabilistic programming?

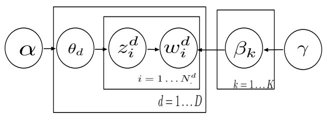
# Increase Programmer Productivity

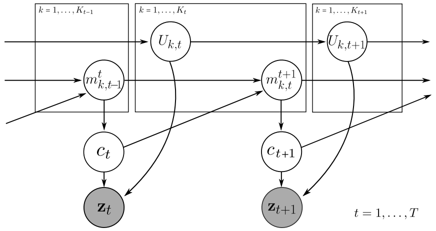
http://www.robots.ox.ac.uk/~fwood/anglican/examples/viewer/?worksheet=complexityreduction

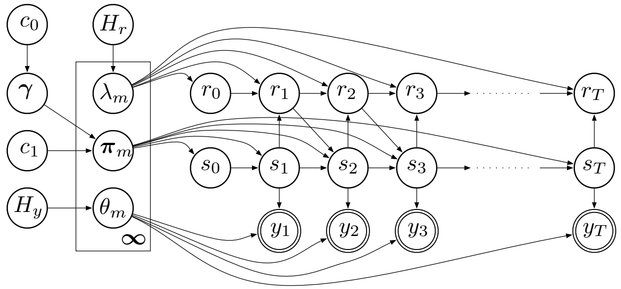


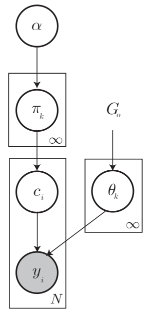
# Commodify Inference

## Models / Simulators

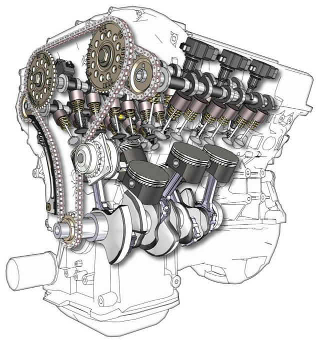


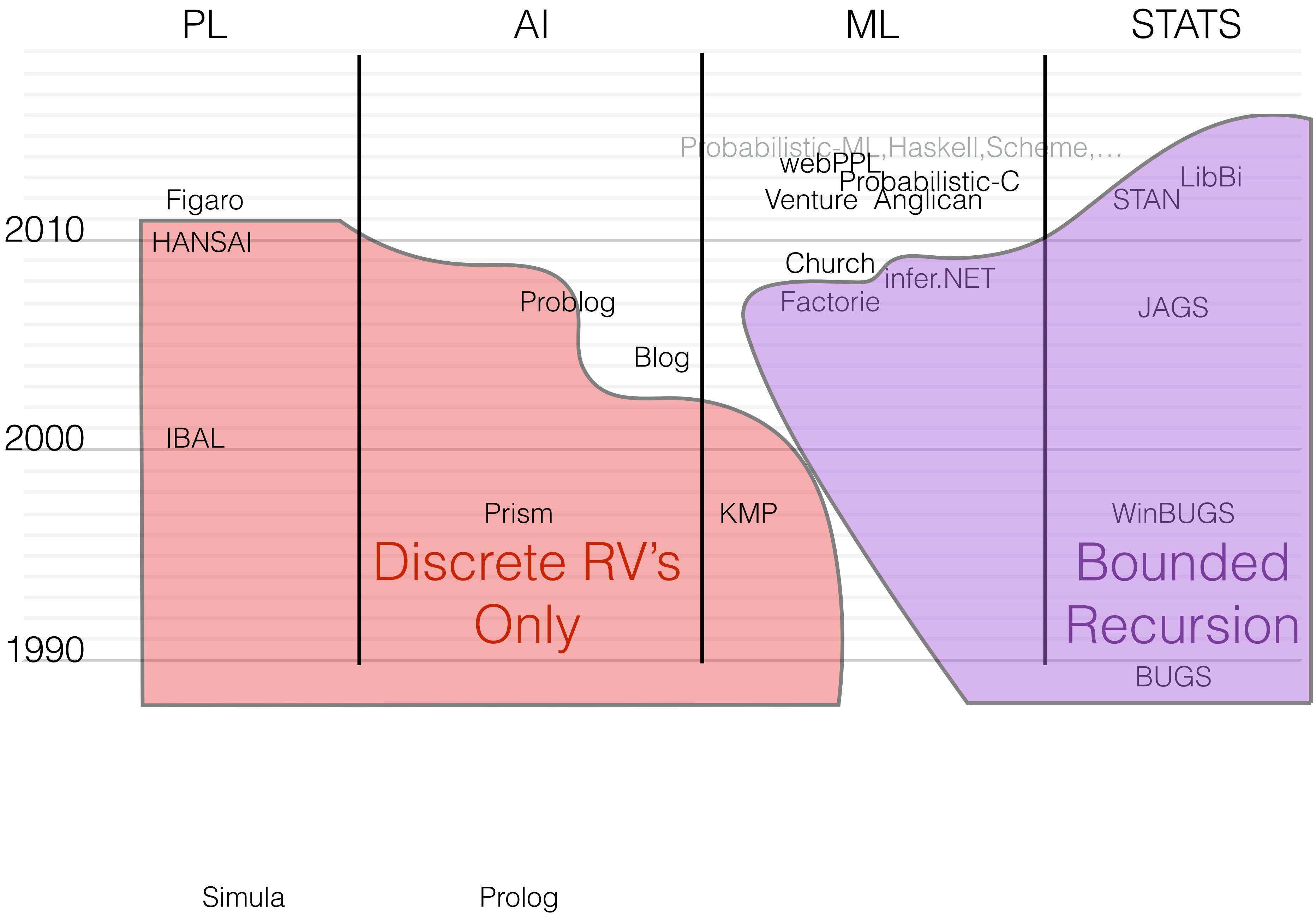




Programming Language Representation / Abstraction Layer



## Inference Engine(s)

# Systems



PL
AI
ML
STATS
2010
2000
1990
Probabilistic-ML,Haskell,Scheme,…
webPPL
Probabilistic-C
LibBi
Figaro
STAN
Venture
Anglican
HANSAI
Church
infer.NET
Problog
Factorie
JAGS
Blog
IBAL
Prism
KMP
WinBUGS
Discrete RV's Only
Bounded Recursion
BUGS
Simula
Prolog

# Anglican

- http://www.robots.ox.ac.uk/~fwood/anglican/

- Higher order, pure functional
- Compiled (CPS -> Clojure -> JVM bytecode)
  - Complete JVM language family interoperability
- First class distributions
- 15+ composable inference algorithms
  - SMC
  - CASCADE
  - PMCMC (PIMH, PGIBBS, PGAS)
  - (Adaptive) LMH
  - …

# Anglican

- http://www.robots.ox.ac.uk/~fwood/anglican/

- Open source (GPLv3)
  - core: https://bitbucket.org/probprog/anglican
  - user: https://bitbucket.org/probprog/anglican-user
  - tutorial: https://bitbucket.org/probprog/mlss2015

# Traditional Bayesian Statistics

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

$$\mu \sim \mathrm{Normal}(1, \sqrt{5})$$

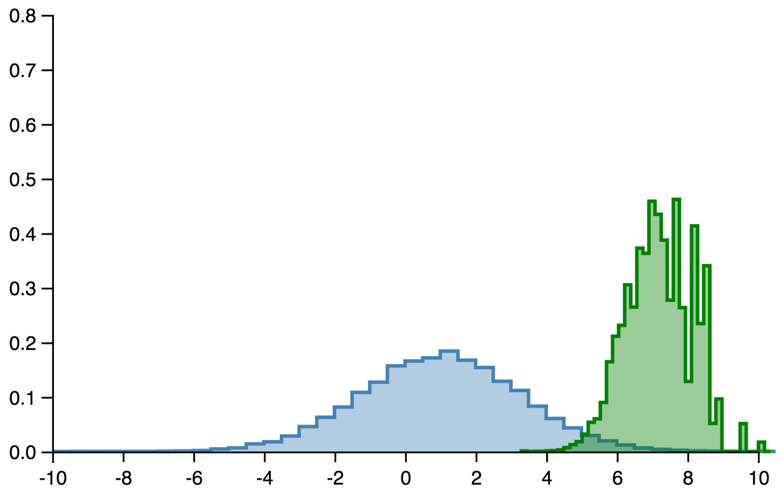
$$y_i|\mu \sim \mathrm{Normal}(\mu, \sqrt{2})$$

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

$$y_1 = 9, y_2 = 8$$

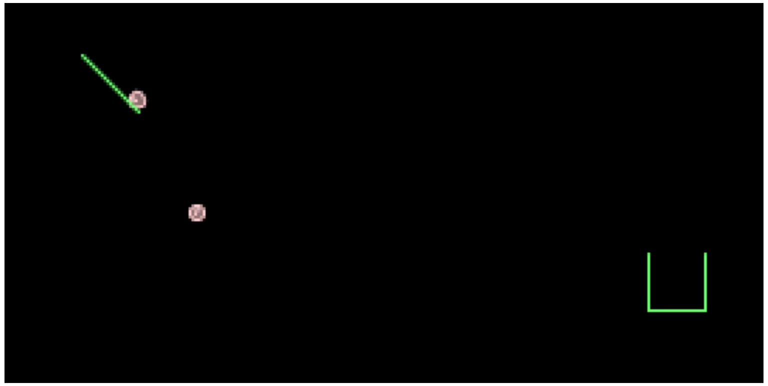
```
(def posterior
  ((conditional gaussian-model
                :pgibbs
                :number-of-particles 1000) dataset))

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

$$\mu|y_{1:2} \sim \mathrm{Normal}(7.25, 0.9129)$$



# Mechanism Design

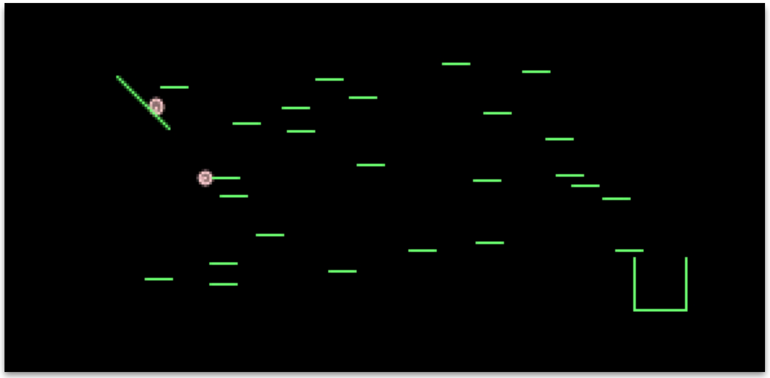
```
(defquery arrange-bumpers []
    (let [bumper-positions []

          ;; 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)]

      (predict :balls balls)
      (predict :num-balls-in-box num-balls-in-box)
      (predict :bumper-positions bumper-positions)))
```

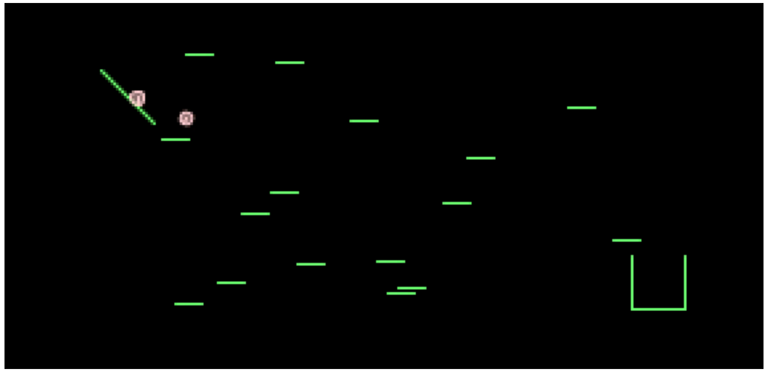


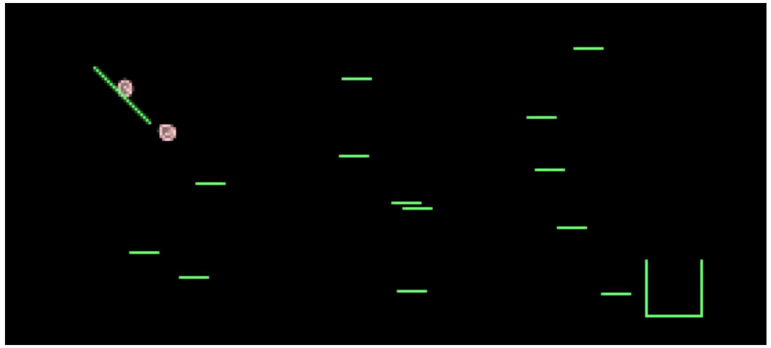
goal: ~20% of balls in box…

# Procedural Design

```
(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)]

      (predict :balls balls)
      (predict :num-balls-in-box num-balls-in-box)
      (predict :bumper-positions bumper-positions)))
```

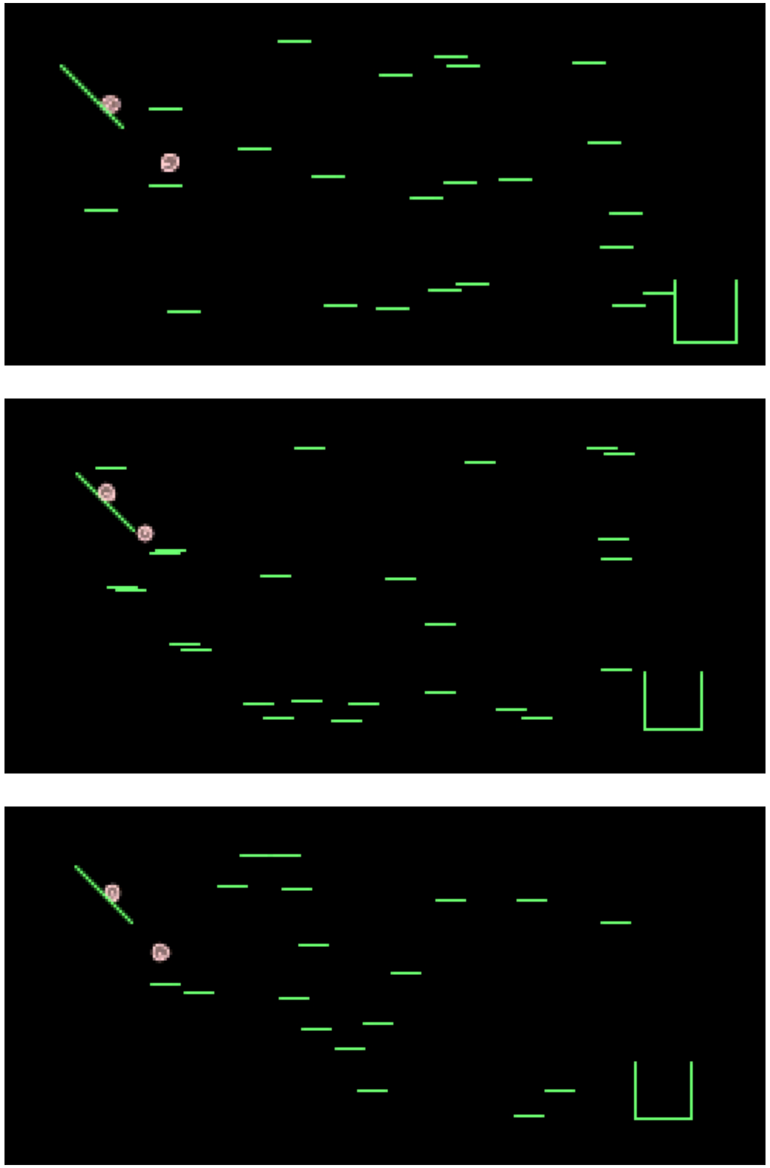






# Inference Over Conditioned Execution Traces

```
(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 2 0.1)]

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

    (predict :balls balls)
    (predict :num-balls-in-box num-balls-in-box)
    (predict :bumper-positions bumper-positions)))
```



# Inference

likehood

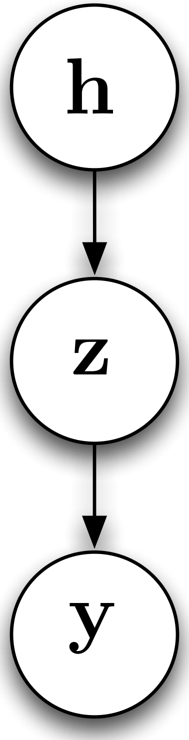
posterior

prior

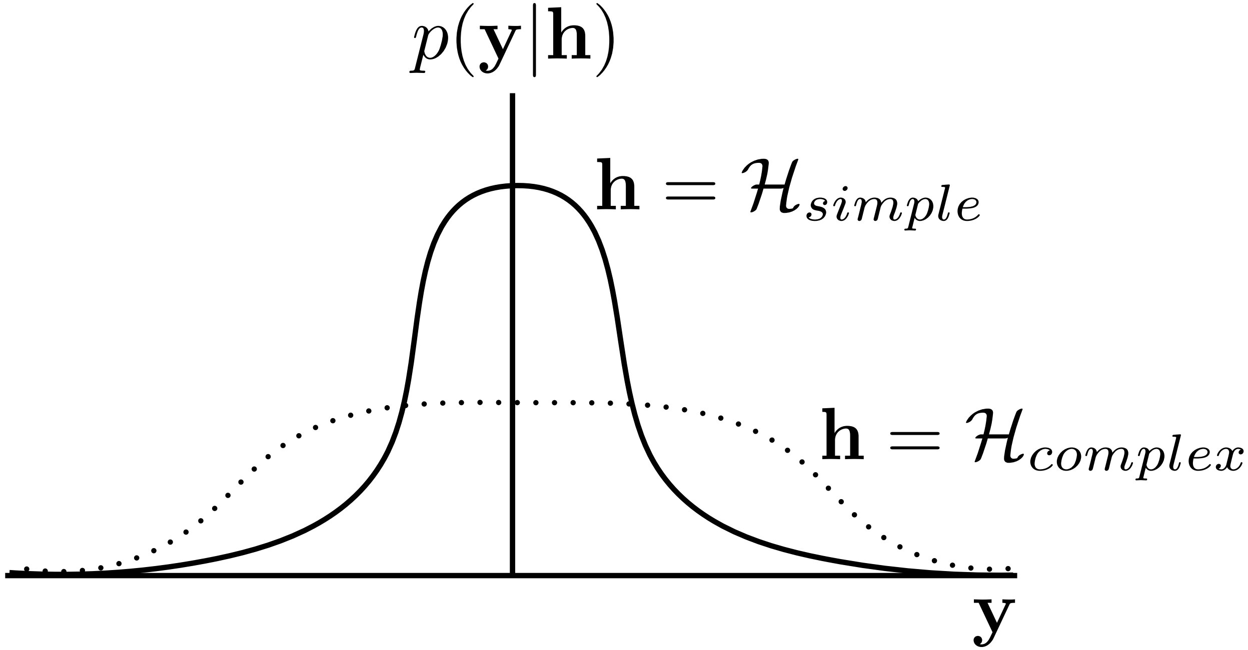
$$p(\mathbf{z}|\mathbf{y},\mathbf{h}) = \frac{p(\mathbf{y}|\mathbf{z},\mathbf{h})p(\mathbf{z}|\mathbf{h})}{p(\mathbf{y}|\mathbf{h})}$$

evidence

$$p(\mathbf{y}|\mathbf{h}) = \int p(\mathbf{y}|\mathbf{z},\mathbf{h})p(\mathbf{z}|\mathbf{h})d\mathbf{z}$$



# Automatic Complexity Regularization




$$p(\mathbf{h}|\mathbf{y}) \propto p(\mathbf{y}|\mathbf{h})p(\mathbf{h})$$

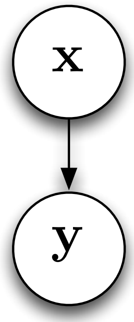
Bayesian Occam’s Razor

D. J. C. MacKay. Information Theory, Inference, and Learning Algorithms. 2003.
I. Murray, Z. Ghahramani. "A note on the evidence and Bayesian Occam’s razor." 2005.

# Probabilistic Programming Is Fully Generative

$$\mathbf{x} = \mathbf{z} \cup \mathbf{h}$$

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




| $\mathbf{x}$ | $\mathbf{y}$ |
|---|---|
| program source code | program output |
| scene description | image |
| policy | reward |
| world | simulator output |
| automata | sequence |

# How Does it Work?

# 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

# Trace Probability

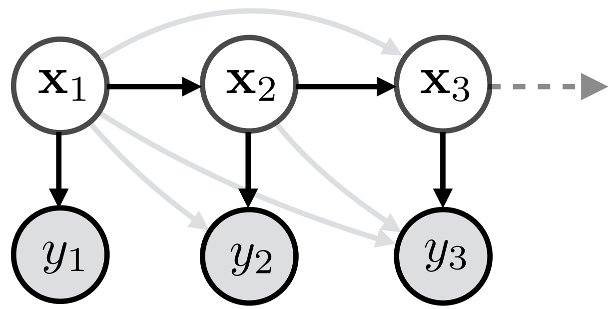
- **`observe`** data points $y_n$
- internal random choices $\mathbf{x}_n$
- simulate from

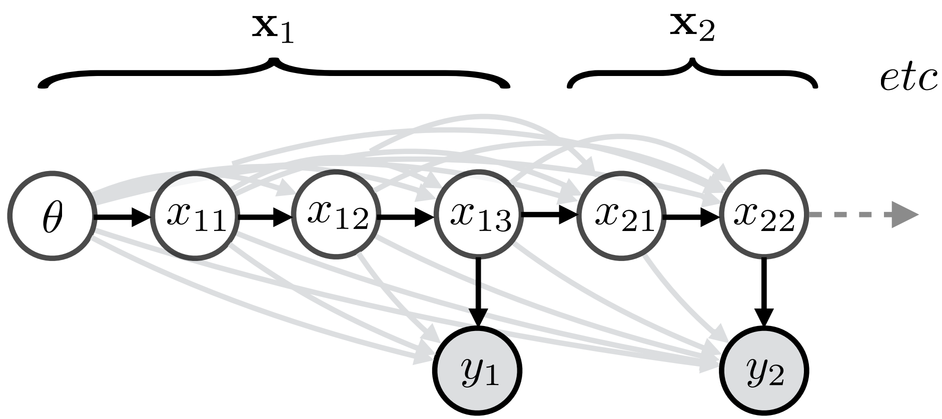
$$f(\mathbf{x}_n|\mathbf{x}_{1:n-1})$$

- by running the program forward
- weight execution traces by

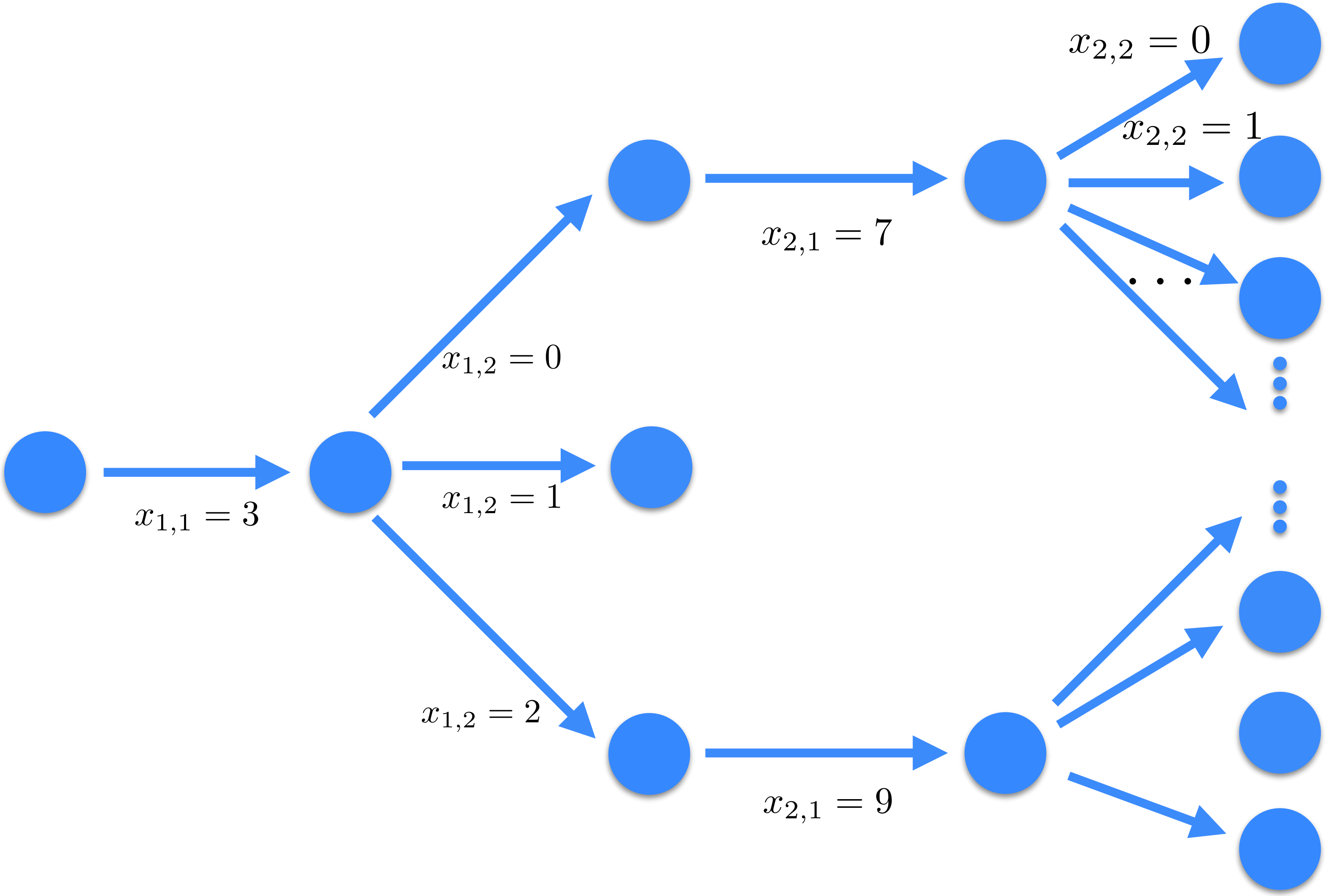
$$g(y_n|\mathbf{x}_{1:n})$$

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



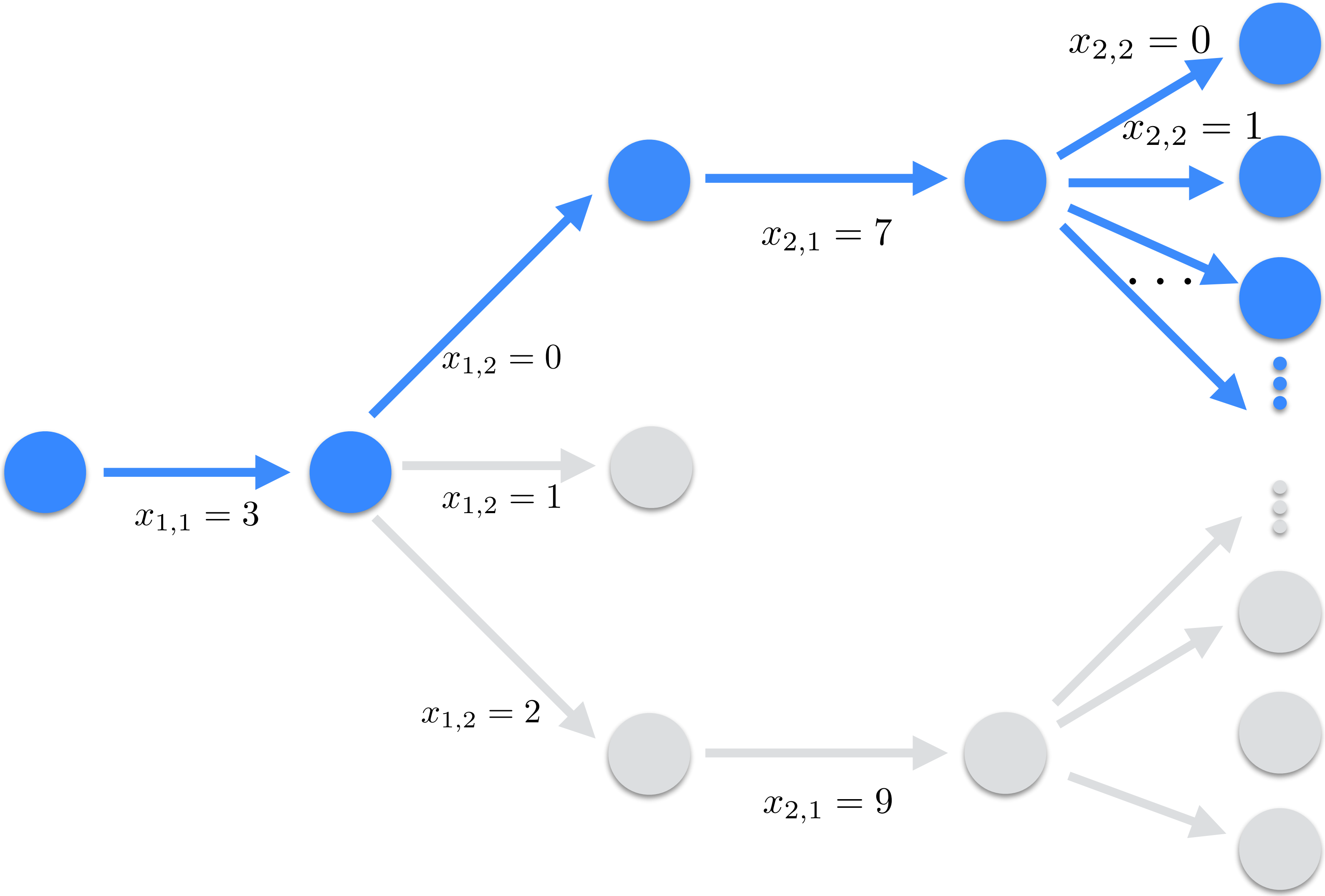



# Traces



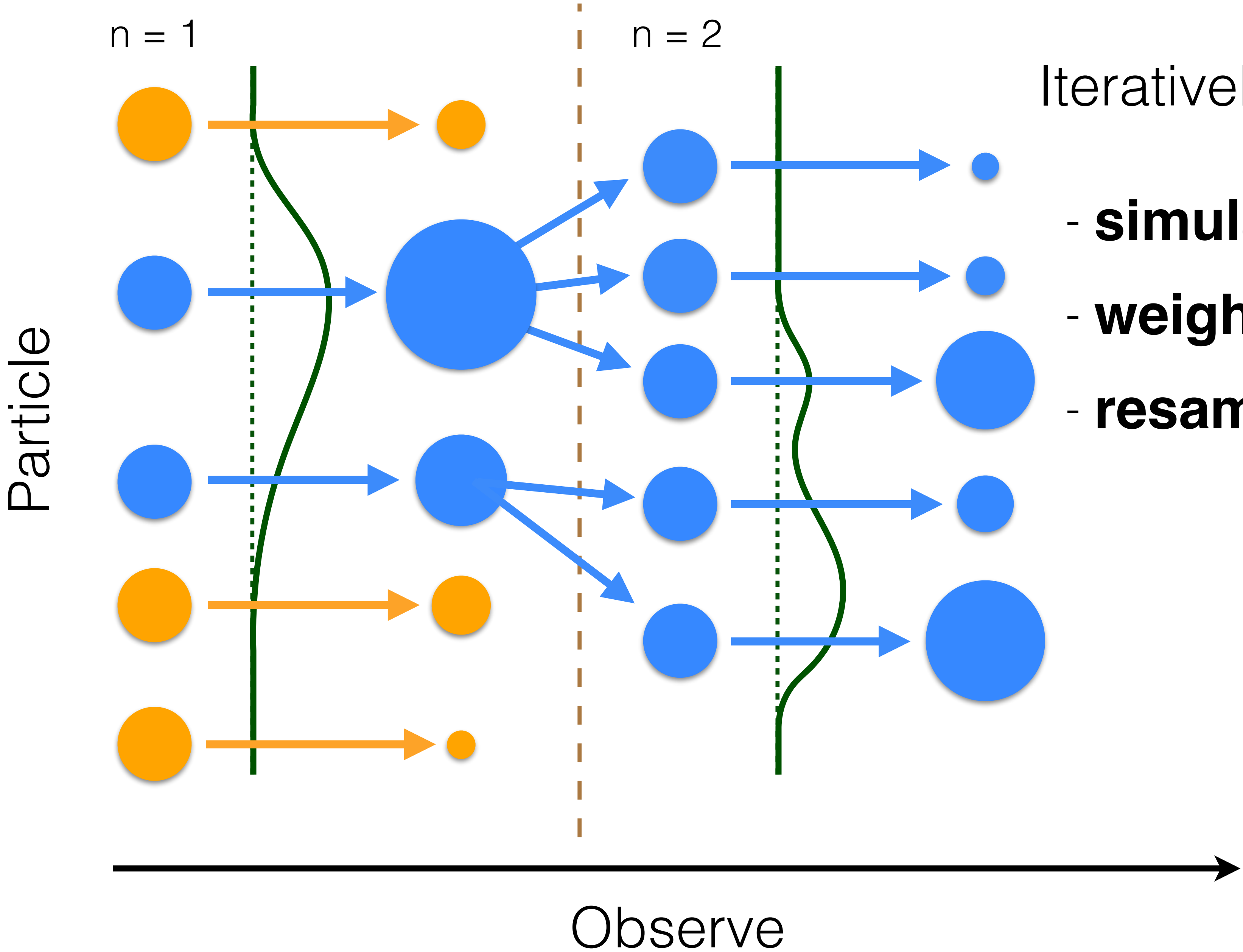

```
(let [x-1-1 3
      x-1-2 (sample (discrete (range x-1-1)))]
  (if (not= x-1-2 1)
    (let [x-2-1 (+ x-1-2 7)]
      (sample (poisson x-2-1)))))
```

# Observe




```
(let [x-1-1 3
      x-1-2 (sample (discrete (range x-1-1)))]
  (if (not= x-1-2 1)
    (let [x-2-1 (+ x-1-2 7)]
      (sample (poisson x-2-1)))))
(observe (gaussian x-2-1 0.0001) 7))
```

# SMC




Iteratively,

- **simulate**
- **weight**
- **resample**

# SMC for Probabilistic Programming

Parallel executions

$$p(\mathbf{x}_{1:n-1}|\mathbf{y}_{1:n-1}) \approx \sum_{\ell=1}^{L} w_{n-1}^{\ell} \delta_{\mathbf{x}_{1:n-1}^{\ell}}(\mathbf{x}_{1:n-1})$$

Sequence of environments

$$p(\mathbf{x}_{1:n}|y_{1:n}) \propto g(y_n|\mathbf{x}_{1:n}) f(\mathbf{x}_n|\mathbf{x}_{1:n-1}) p(\mathbf{x}_{1:n-1}|y_{1:n-1})$$

$$q(\mathbf{x}_{1:n}|y_{1:n}) = f(\mathbf{x}_n|\mathbf{x}_{1:n-1}) p(\mathbf{x}_{1:n-1}|y_{1:n-1})$$

Proposal

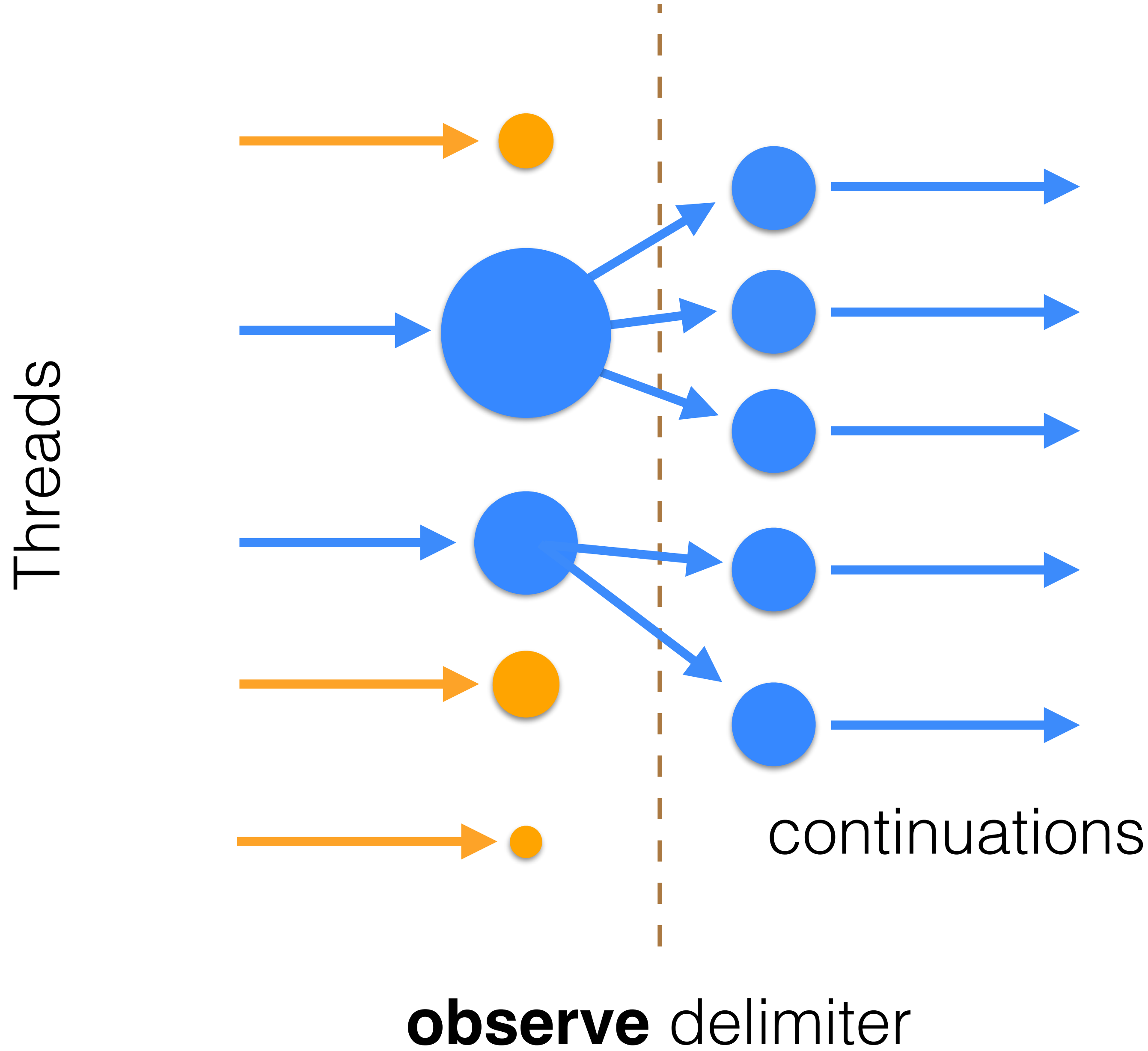
$$p(\mathbf{x}_{1:n}|y_{1:n}) \approx \sum_{\ell=1}^{L} g(y_n|\mathbf{x}_{1:n}^{\ell}) \delta_{\mathbf{x}_{1:n}^{\ell}}(\mathbf{x}_{1:n}), \qquad \mathbf{x}_{1:n}^{\ell} = \mathbf{x}_n^{\ell} \mathbf{x}_{1:n-1}^{a_{n-1}^{\ell}} \sim f$$

Weight of particle
Is observation likelihood

Run program forward
until next observe

**Fischer, Kiselyov, and Shan** "Purely functional lazy non-deterministic programming" ACM Sigplan 2009
**W., van de Meent, and Mansinghka** "A New Approach to Probabilistic Programming Inference" AISTATS 2014
**Paige and W.** "A Compilation Target for Probabilistic Programming Languages" ICML 2014

# SMC for Probabilistic Programming




Intuitively

- **run**
- **wait**
- **fork**

# Issues

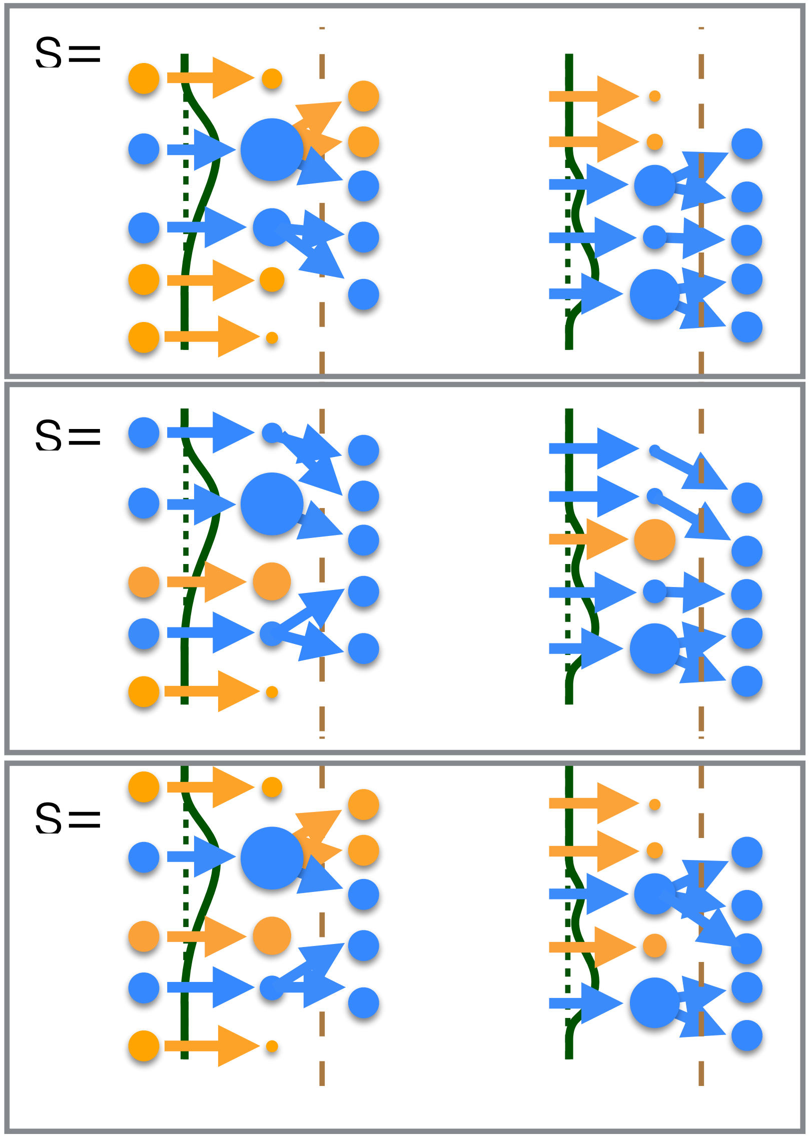
- Degeneracy
- Not iterable (naively)

# PMCMC for Probabilistic Programming

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

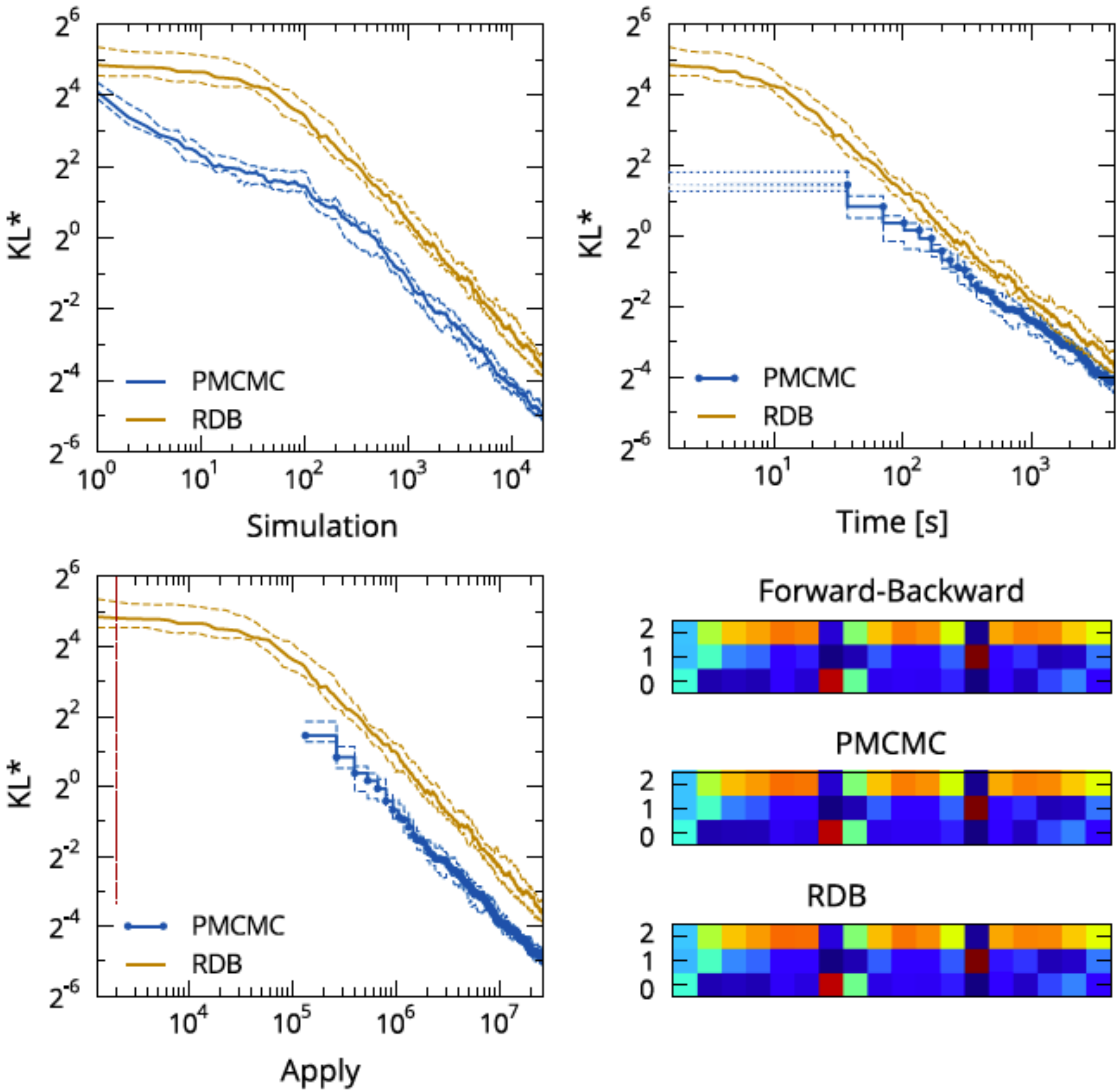
- Sequential Monte Carlo is now a building block for other inference techniques

- Iterable SMC
  - PIMH : "particle independent Metropolis-Hastings"
  - PGIBBS : "iterated conditional SMC"
  -




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

# Better Inference Per Unit Energy



# PMCMC (and SMC) Methods Only Require

- Initialization (sample)

$$p(\mathbf{x}_1)$$

- Forward simulation (sample)

$$f(\mathbf{x}_n|\mathbf{x}_{1:n-1})$$

- Observation likelihood computation
  - pointwise evaluation up to normalization

$$g(y_n|\mathbf{x}_{1:n})$$

# Stop Making New Probabilistic Programming Languages

sort-of

# Probabilistic C

- Standard C with two new directives: `observe` and `predict`

- Is compiled to parallel machine code by standard compilers

- Relies on standard operating system functionality: processes, forking, mutexes, shared memory

- Compiled programs automatically do **inference**

- Emits posterior samples of predicted quantities

Paige, W.; ICML 2014

# Simple example program

Posterior mean of a Gaussian, given i.i.d. draws

**`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;
}
```

# A Markov model

$$z_0 \sim \text{Discrete}([1/K, \ldots, 1/K]) \quad z_n | z_{n-1} \sim \text{Discrete}(T_{z_{n-1}})$$

hmm.c

```c
#include "probabilistic.h"
#define K 3
#define N 17

/* Markov transition matrix */
static double T[K][K] = { { 0.1,  0.5,  0.4 },
                          { 0.2,  0.2,  0.6 },
                          { 0.15, 0.15, 0.7 } };

/* Prior distribution on initial state */
static double initial_state[K] = { 1.0/3, 1.0/3, 1.0/3 };

/* Generative program for a Markov model */
int main(int argc, char **argv) {

    int states[N];
    for (int n=0; n<N; n++) {
        states[n] = (n==0) ? discrete_rng(initial_state, K)
                           : discrete_rng(T[states[n-1]], K);
        predict("state[%d],%d\n", n, states[n]);
    }

    return 0;
}
```

Line 1 Col 1 | ANSI C | Unicode (UTF-8) | Unix (LF) | Last saved: 6/22/14, 1:59:35 | 658 / 85 / 25

# Conditioning on observed data

$$z_0 \sim \text{Discrete}([1/K, \ldots, 1/K]) \quad z_n | z_{n-1} \sim \text{Discrete}(T_{z_{n-1}}) \quad y_n | z_n \sim \text{Normal}(\mu_{z_n}, \sigma^2)$$

hmm.c

```c
#include "probabilistic.h"
#define K 3
#define N 17

/* Markov transition matrix */
static double T[K][K] = { { 0.1,  0.5,  0.4 },
                          { 0.2,  0.2,  0.6 },
                          { 0.15, 0.15, 0.7 } };

/* Prior distribution on initial state */
static double initial_state[K] = { 1.0/3, 1.0/3, 1.0/3 };

/* Generative program for a Markov model */
int main(int argc, char **argv) {

    int states[N];
    for (int n=0; n<N; n++) {
        states[n] = (n==0) ? discrete_rng(initial_state, K)
                           : discrete_rng(T[states[n-1]], K);
        predict("state[%d],%d\n", n, states[n]);
    }

    return 0;
}
```

Line 13 Col 1 | ANSI C | Unicode (UTF-8) | Unix (LF) | Last saved: 6/22/14, 2:06:04 | 658 / 85 / 25

# Changing the generative model is easy

Suppose the transition matrix were unknown: $T_k \sim \mathrm{Dirichlet}(\alpha_k)$

hmm.c

```c
#define N 17

/* Markov transition matrix */
static double T[K][K] = { { 0.1,  0.5,  0.4 },
                          { 0.2,  0.2,  0.6 },
                          { 0.15, 0.15, 0.7 } };

/* Observed data */
static double data[N] = { NAN, .9, .8, .7, 0, -.025,
                          -5, -2, -.1, 0, 0.13, 0.45,
                           6, 0.2, 0.3, -1, -1 };

/* Prior distribution on initial state */
static double initial_state[K] = { 1.0/3, 1.0/3, 1.0/3 };

/* Per-state mean of Gaussian emission distribution */
static double state_mean[K] = { -1, 1, 0 };

/* Generative program for a HMM */
int main(int argc, char **argv) {

    int states[N];
    for (int n=0; n<N; n++) {
        states[n] = (n==0) ? discrete_rng(initial_state, K)
                           : discrete_rng(T[states[n-1]], K);
```

Line 6 Col 22 | ANSI C | Unicode (UTF-8) | Unix (LF) | Last saved: 6/22/14, 2:38:02 | 122 / 9 / 2

# Implementation

- Inference: forward simulation (SMC, particle MCMC, particle cascade, …)
- POSIX **fork:**
    - operating-system level call to clone a running process: branch on program execution state, explore many downstream paths
    - duplicates *entire* memory address space
    - efficient: lazy copy-on-write behaviour
    - parallel: each downstream path is explored by an independent OS process

# The Next 700 Probabilistic Programming Languages?

W., Jeffrey Mark Siskind and Brooks Paige
(in prep. 2015)

# Probabilistic Scheme

Gaussian example, in probabilistic scheme

```
;;; Define a (random) mean, mu
(define mu (normal 1 (sqrt 5)))
(define sigma (sqrt 2))

;;; Define a likelihood funciton
(define (likelihood x) (normal-lnp x mu sigma))

;;; Condition on the data
(define data (list 8 9))
(map observe-lnp (map likelihood data))

;;; Emit samples of the mean
(predict-float "mean" mu)
```

# All we need for probabilistic scheme

- existing scheme compiler (i.e. STALIN)

- existing C compiler (i.e. GCC, clang)

```
;;; ERPs
(define poisson-rng
 (foreign-procedure (double) long "poisson_rng" "probabilistic"))

(define normal-lnp
 (foreign-procedure (double double double) double "normal_lnp" "probabilistic"))
;;; plus more -rng and -lnp

;;; directives
(define observe (foreign-procedure (double) void "observe" "probabilistic"))

(define predict-value
 (foreign-procedure (char* double) void "predict_value" "probabilistic"))

;;; necessary boilerplate
(vector-ref argv 0)
```

# Probabilistic C: sum-equals

```
#include <probabilistic.h>

int main(int argc, char *argv[]) {
  long a = poisson_rng(100.0)-100;
  long b = poisson_rng(100.0)-100;
  observe(normal_lnp(7.0, (double)(a+b), 0.00001));
  predict_value("a", (double)a);
  predict_value("b", (double)b);
}
```

# C (GCC, CLANG)

```c
int main(int argc, char *argv[]) {
  long a = poisson_rng(100.0)-100;
  long b = poisson_rng(100.0)-100;
  observe(normal_lnp(7.0,
            (double)(a+b), 0.00001));
  predict_value("a", (double)a);
  predict_value("b", (double)b);
}
```

# Scheme (STALIN)

```scheme
(define a (- (poisson-rng 100.0) 100))
(define b (- (poisson-rng 100.0) 100))
(observe (normal-lnp 7.0
    (exact->inexact (+ a b)) .00001))
(predict-value "a" (exact->inexact a))
(predict-value "b" (exact->inexact b))
```

# Standard ML (MLTON)

```sml
val a = (poisson_rng 100.0)-100
val b = (poisson_rng 100.0)-100
val _ = observe (normal_lnp (7.0,
          (int64ToReal (a+b)), 0.00001))
val _ = predict_value ("a", (int64ToReal a))
val _ = predict_value ("b", (int64ToReal b))
val _ = return_from_main 0
```

# Haskell (GHC)

```haskell
model = do
  a <- (+(-100)) <$> poisson_rng 100.0
  b <- (+(-100)) <$> poisson_rng 100.0
  observe $ normal_lnp 7
        (realToFrac (a+b)) 0.00001
  predict_value "a" (realToFrac a)
  predict_value "b" (realToFrac b)
  return ()
```

# Bubble Up

AI

↑

Models

↑

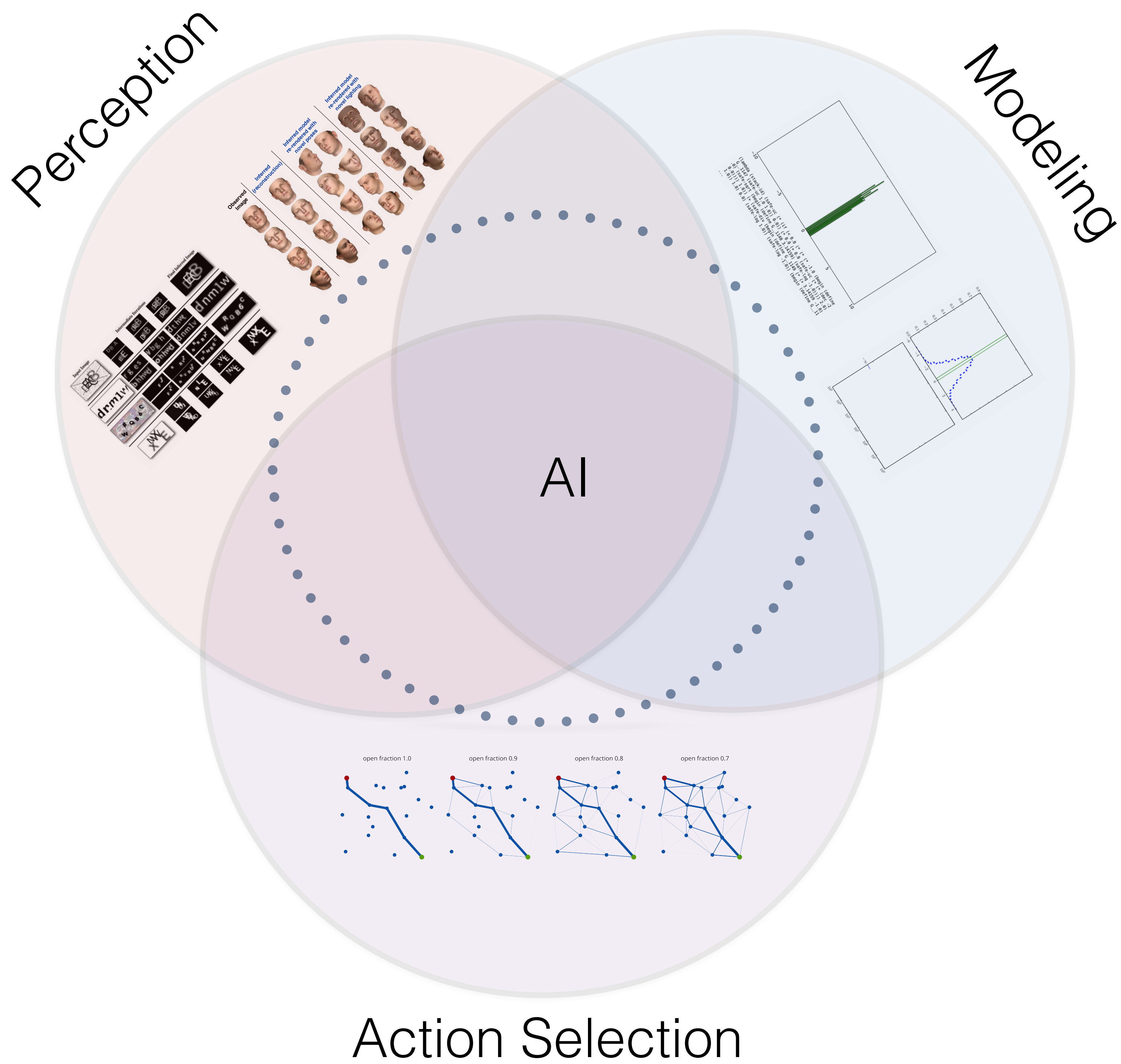
Probabilistic Programming Language

↑

Probabilistic Programming System

↑

Inference


Perception
Modeling
AI
Action Selection
open fraction 1.0
open fraction 0.9
open fraction 0.8
open fraction 0.7

# It’s All About Inference

- Parallelism

  “Asynchronous Anytime Sequential Monte Carlo” [Paige, W., Doucet, Teh NIPS 2014]

- Backwards passing

  “Particle Gibbs with Ancestor Sampling for Probabilistic Programs” [van de Meent, Yang, Mansinghka, W. AISTATS 2015]

- Search

  “Maximum a Posteriori Estimation by Search in Probabilistic Models” [Tolpin, W., SOCS, 2015]

- Adaptation

  “Output-Sensitive Adaptive Metropolis-Hastings for Probabilistic Programs” [Tolpin, van de Meent, Paige, W ; ECML, 2015]

- Novel proposals

  “Neural Adaptive Inference for Probabilistic Programming” [Paige, W.; in submission]

# Thank You

- Questions?

van de Meent

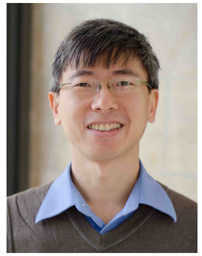
Paige

Yang

Tolpin

Perov

Le



Teh

Doucet

Siskind

- Funding: DARPA, Amazon, Microsoft