Probabilistic Programming Deep Learning and Inference Compilation

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

Accelerating Data Science with HPC
Summer School, Lugano 2017

THE ALAN TURING INSTITUTE

Objectives For Today

- Get you to understand
  - probabilistic programming
    - Clojure / Anglican
  - deep learning and, in particular, automatic differentiation
    - PyTorch
  - amortized inference via inference compilation

Baydin

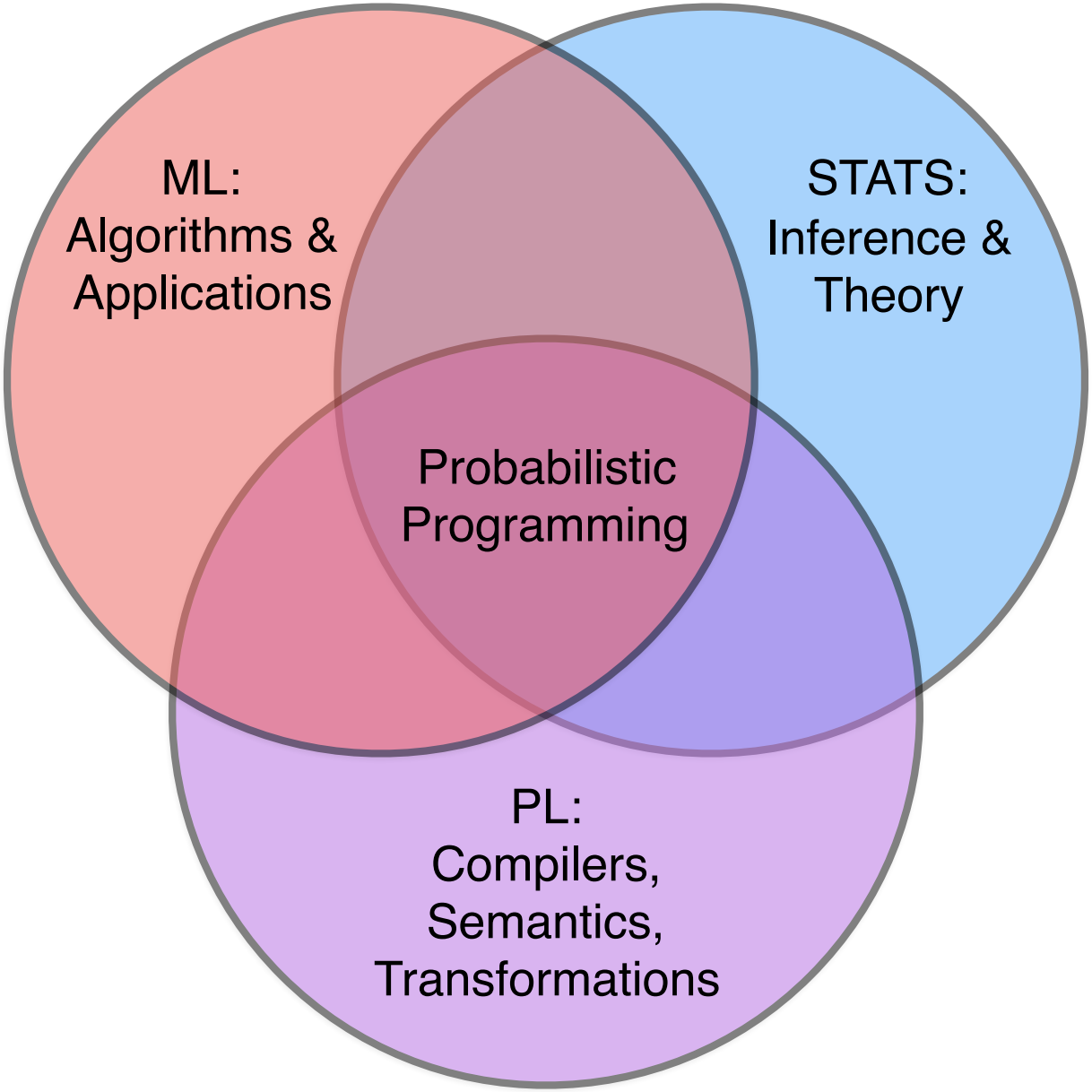
Le

Part 1 : Probabilistic Programming Objectives

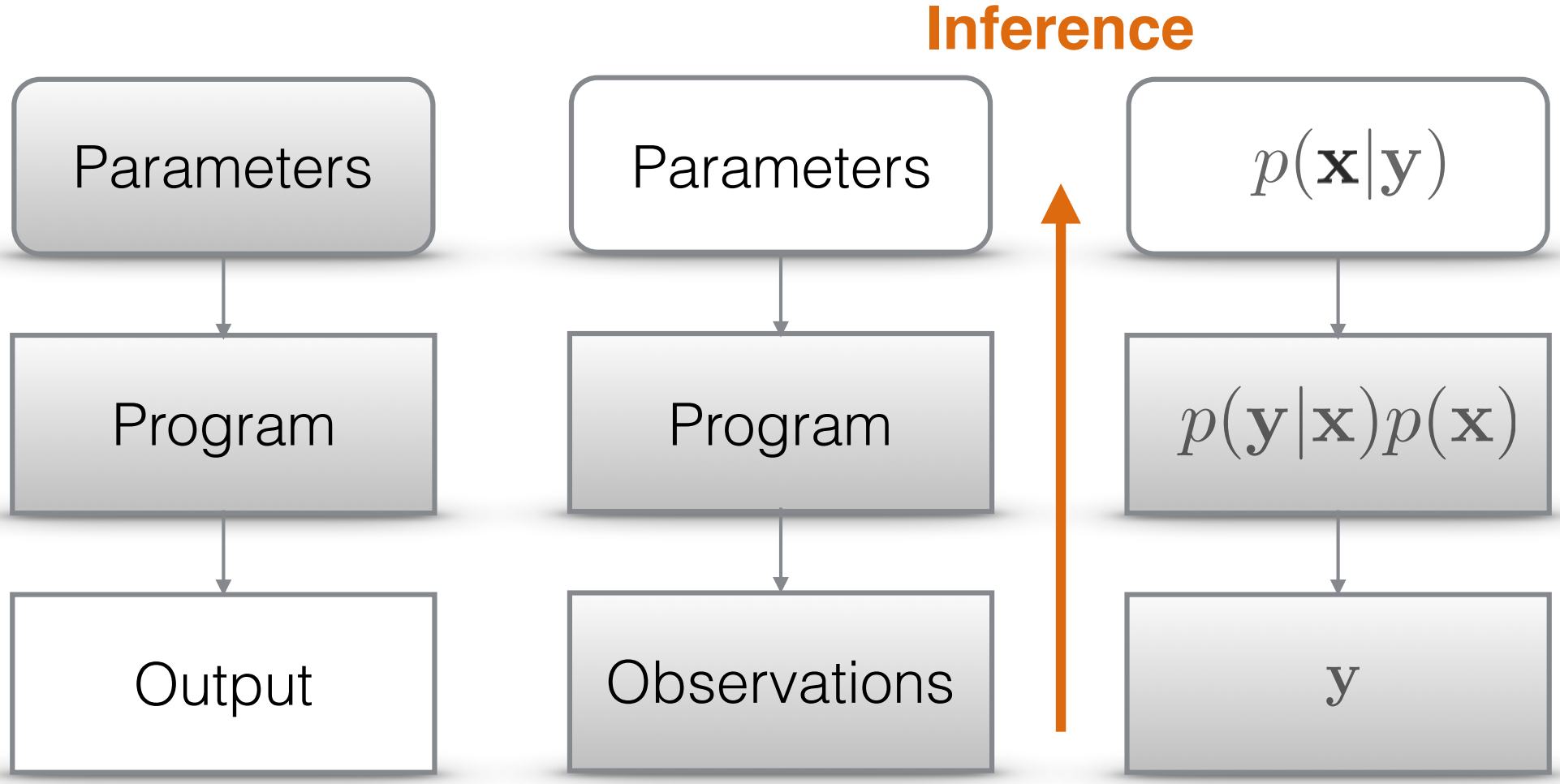
- Get you to understand
  - probabilistic programming in general
    - motivation
    - history
    - how it works
  - functional programming
    - Clojure/Anglican
  - probabilistic programming in practice
    - write your own probabilistic programs (in Anglican)
- Prerequisite understanding
  - generative modeling
  - inference and conditioning
  - approximate Bayesian computation
  - programming
  - stochastic simulation

What is probabilistic programming?

The Field


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

Intuition



CS Probabilistic Programming Statistics

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

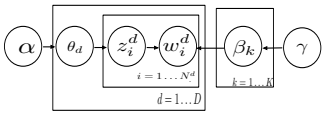
(2) the ability to condition values of variables in a program via observations."

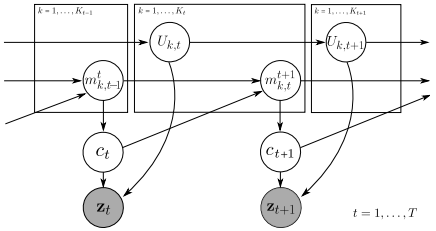
Gordon, Henzinger, Nori, and Rajamani
"Probabilistic programming." In Proceedings of On The Future of Software Engineering (2014).

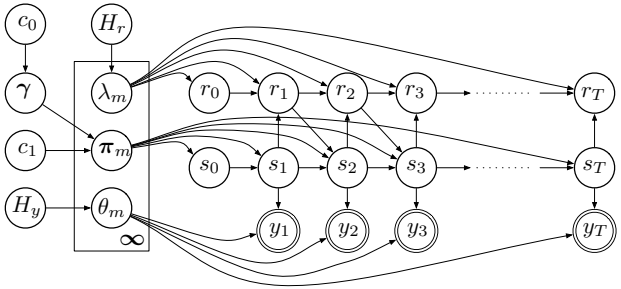
Goals of the Field

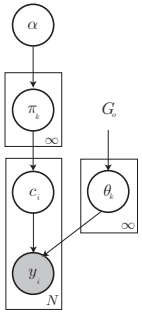
Commodify Inference

Models / Simulators

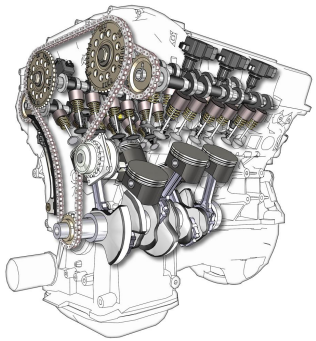








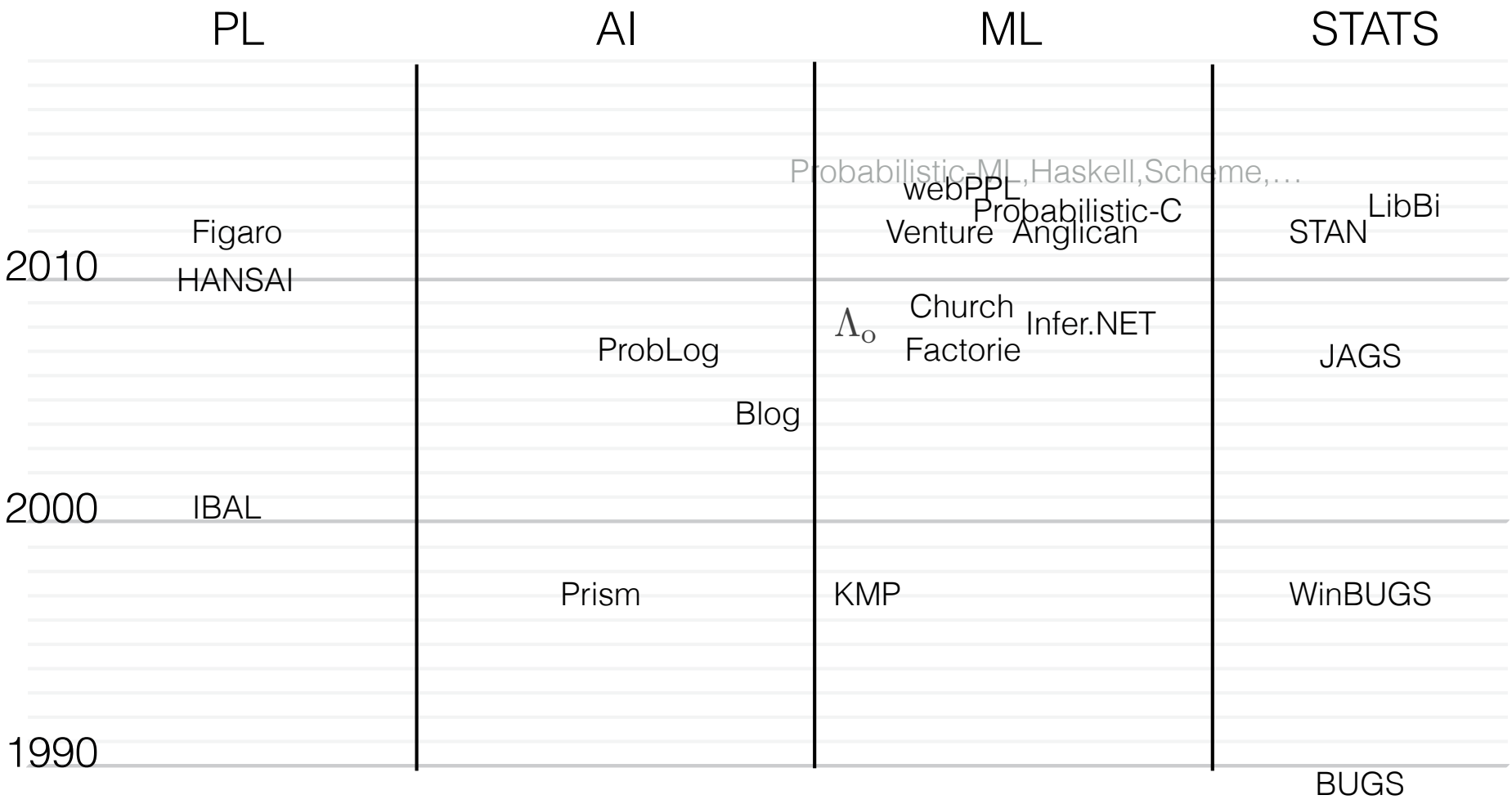
Language Representation / Abstraction Layer



Inference engines

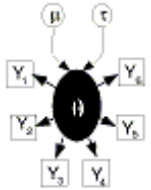
History

Long


PL
AI
ML
STATS
Probabilistic-ML,Haskell,Scheme,…
webPPL
Probabilistic-C
LibBi
Figaro
Venture
Anglican
STAN
2010
HANSAI
Church
Infer.NET
ProbLog
Factorie
JAGS
Blog
2000
IBAL
Prism
KMP
WinBUGS
1990
BUGS
Simula
Prolog

Success Stories

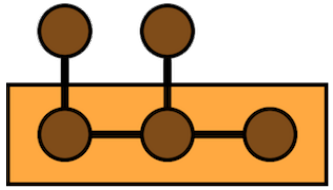
Graphical Models



BUGS

STAN

Factor Graphs

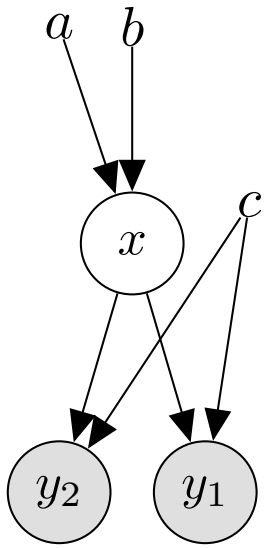


Factorie

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);
}
```

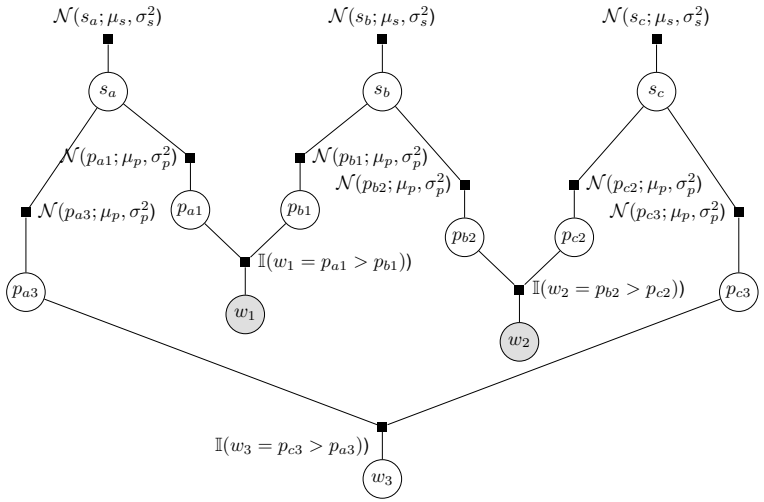
$$\nabla_{\mathbf{x}} \log p(\mathbf{x}, \mathbf{y})$$

- 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.

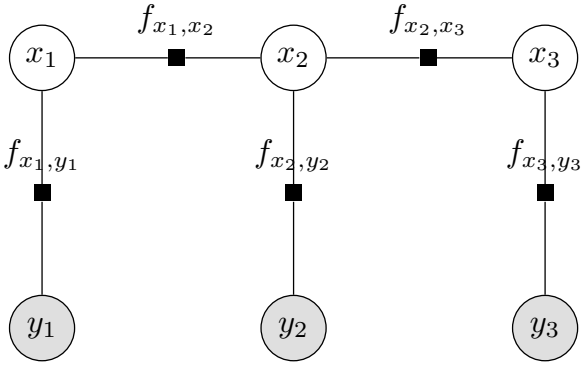
STAN Development Team "Stan: A C++ Library for Probability and Sampling." 2014.

Factorie and Infer.NET

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



TrueSkill

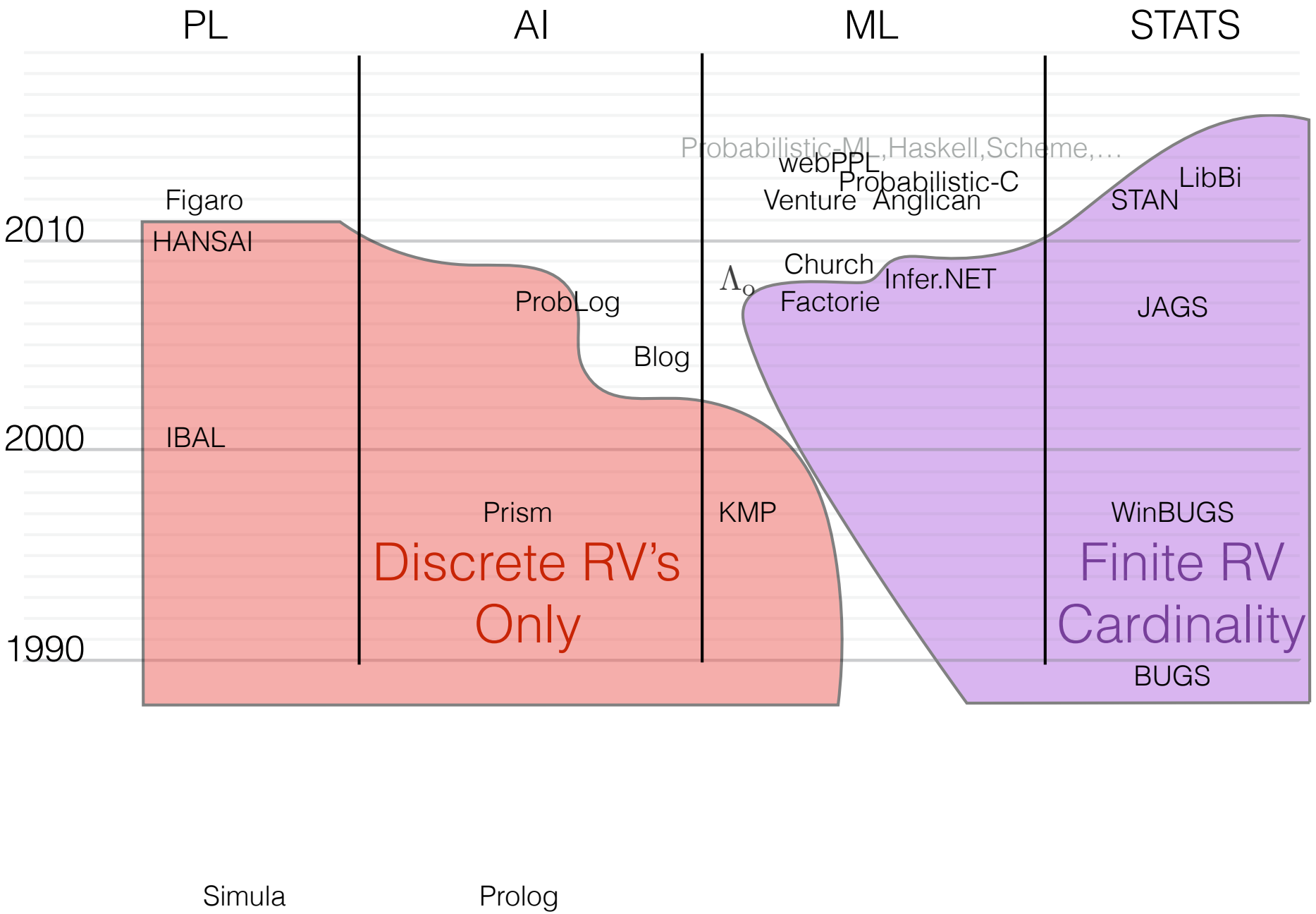


CRF

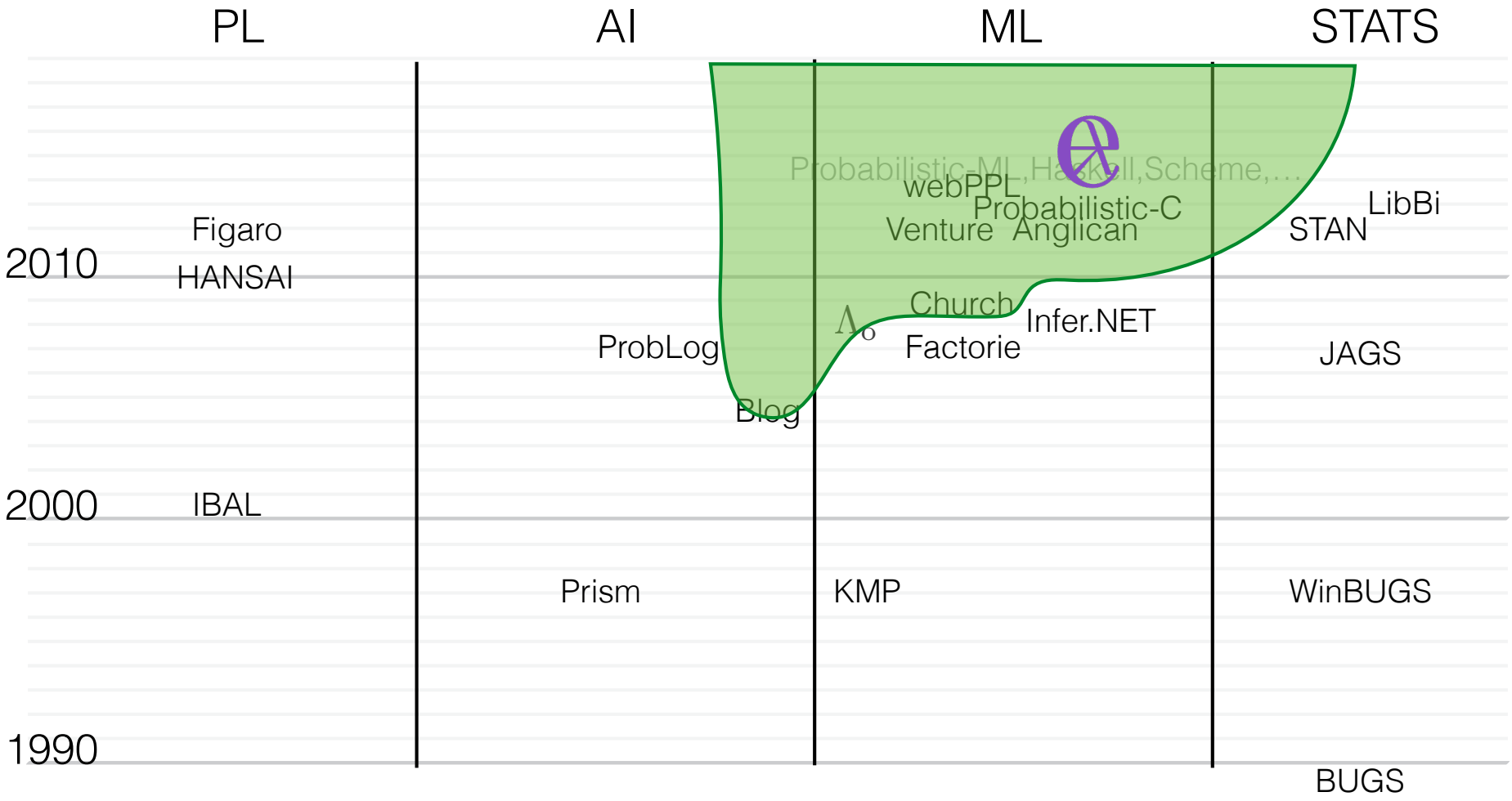
Minka, Winn, Guiver, and Knowles "Infer .NET 2.4, Microsoft Research Cambridge." 2010.

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

First-Order PPLs : (FOPPL)s

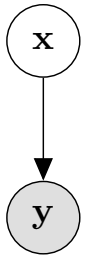


Higher-Order PPLs : (HOPPL)s



New Kinds of Models

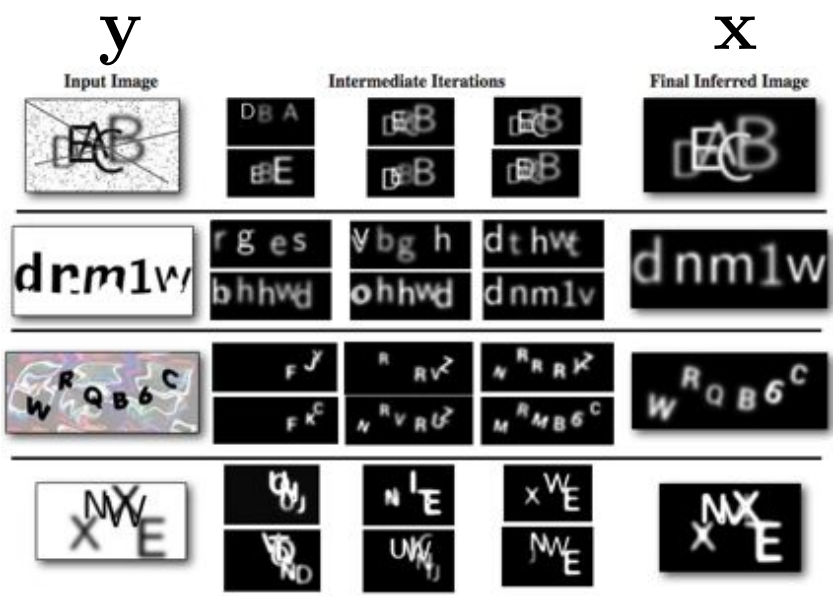
$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{y})}$$



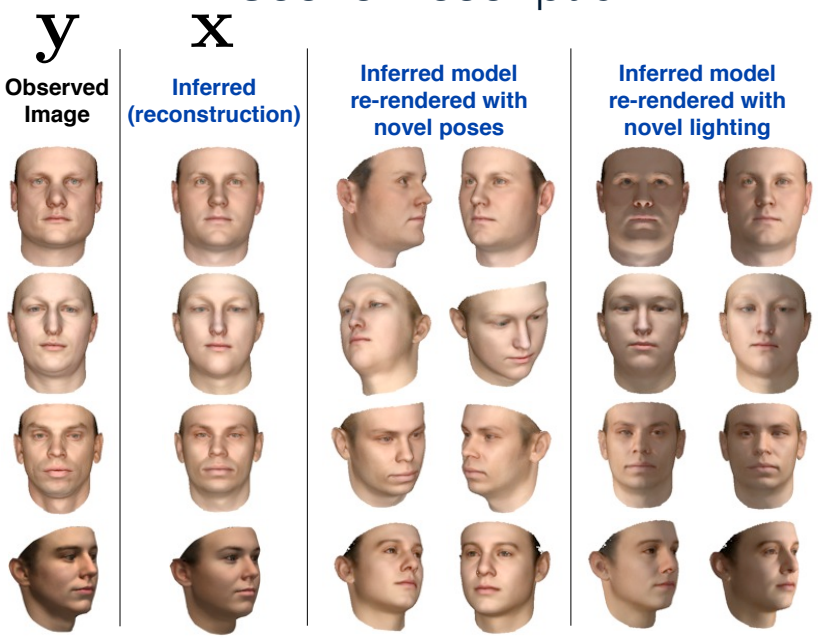
$\mathbf{x}$	$\mathbf{y}$
program source code	program output
scene description	image
policy and world	rewards
cognitive process	behavior
simulation	constraint

Perception / Inverse Graphics

Captcha Solving



Scene Description



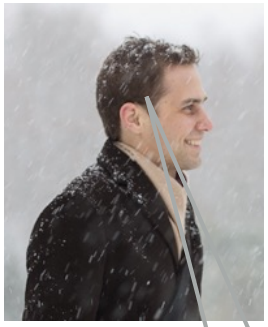
x	y
scene description	image

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…



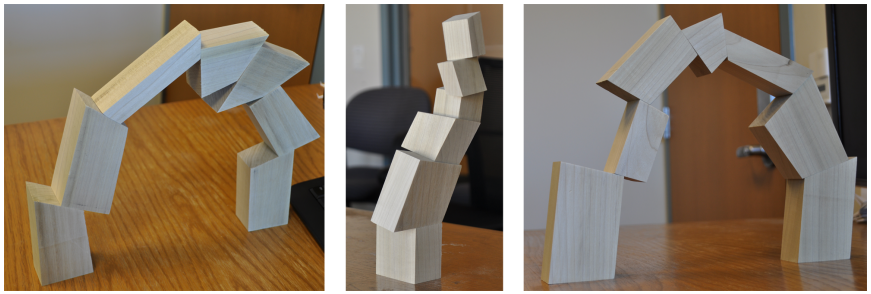
I prefer the pub.
Where will Noah go?
Simulate Noah:
Noah prefers pub
but will go wherever Andreas is
Simulate Noah simulating Andreas:
…
-> both go to pub

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

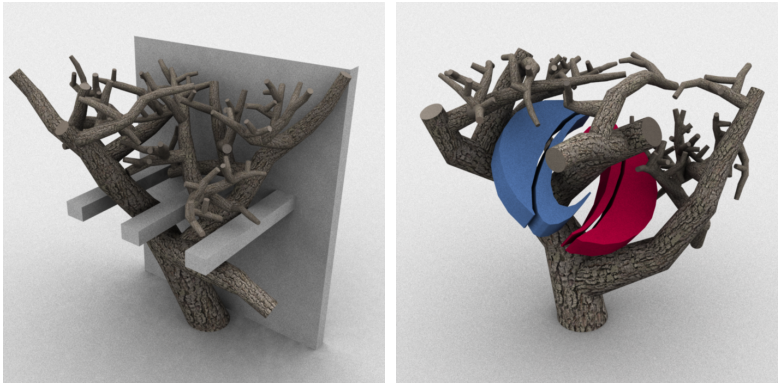
Stuhlmüller, and Goodman.
"Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs."
Cognitive Systems Research 28 (2014): 80-99.

Directed Procedural Graphics

Stable Static Structures



Procedural Graphics

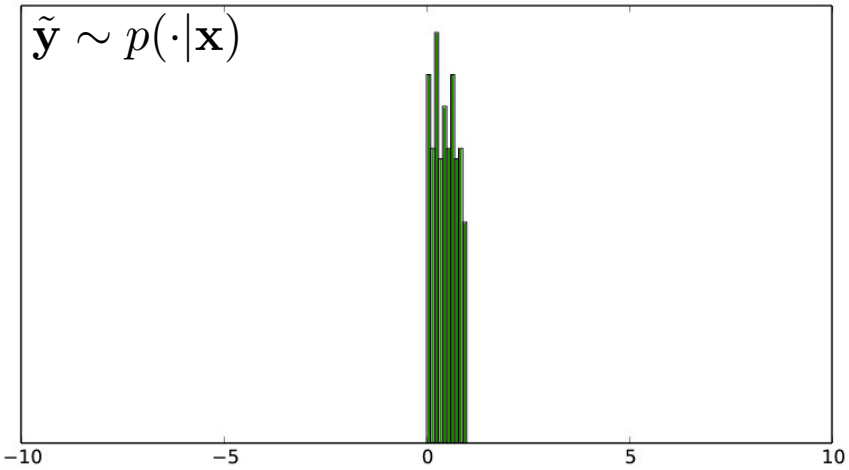


x	y
simulation	constraint

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

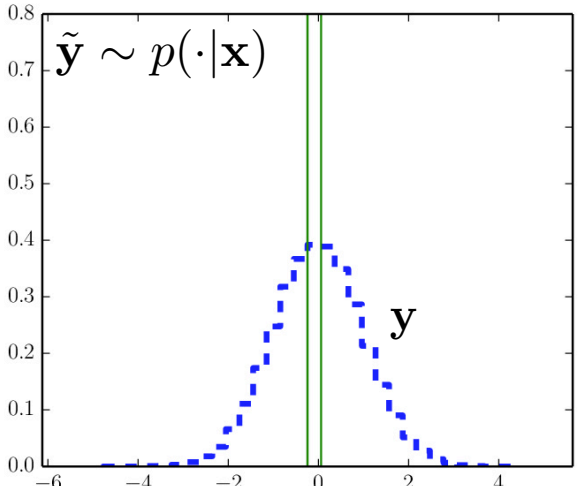
Ritchie, Mildenhall, Goodman, & Hanrahan.
"Controlling Procedural Modeling Programs with Stochastically-Ordered Sequential Monte Carlo."
SIGGRAPH (2015)

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
...
```

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



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

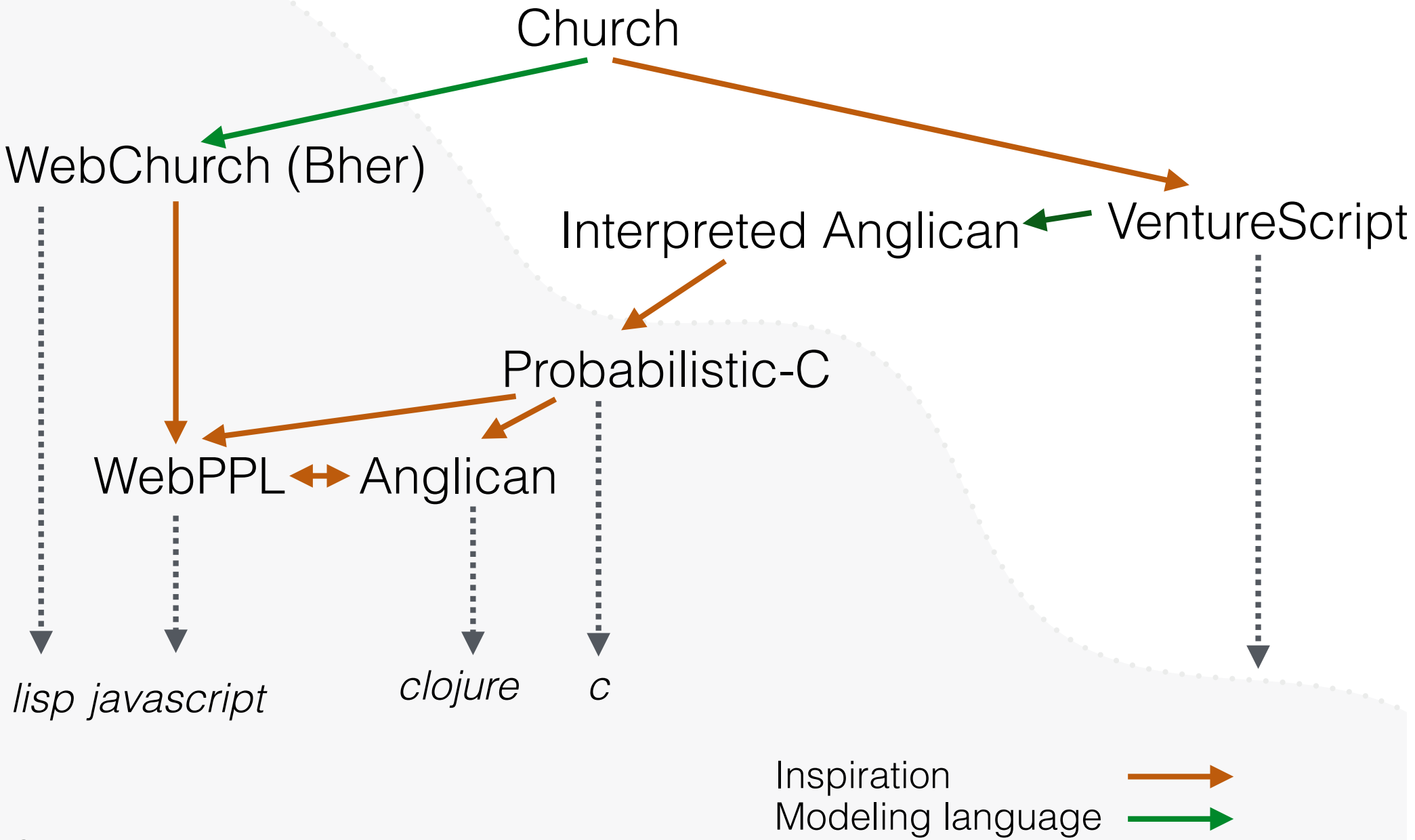
$\mathbf{x}$	$\mathbf{y}$
program source code	program output

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



Syntax : Anglican ≈ Clojure ≈ Church ≈ Lisp

- Notation : *Prefix* vs. infix

```
;; Add two numbers
(+ 1 1)

;; Subtract: "10 - 3"
(- 10 3)

;; (10 * (2.1 + 4.3) / 2)
(/ (* 10 (+ 2.1 4.3)) 2)
```

Syntax

- Notation : *Prefix* vs. infix

```
;; Add two numbers
(+ 1 1)

;; Subtract: "10 - 3"
(- 10 3)

;; (10 * (2.1 + 4.3) / 2)
(/ (* 10 (+ 2.1 4.3)) 2)
```

- Branching

```
;; outputs 4
(+ (if (< 4 5) 1 2) 3)
```

Functions

- Functions are first class

```
;; evaluates to 32
((fn [x y] (+ (* x 3) y))
  10
  2)
```

Functions

- Functions are first class

```
;; evaluates to 32
((fn [x y] (+ (* x 3) y))
  10
  2)
```

- Local bindings

```
;; let is syntactic "sugar" for the same
(let [x 10
      y 2]
  (+ (* x 3) y))
```

Higher-Order

- map

```
;; Apply the function f(x,y) = x + 2y to the
;; x values [1 2 3] and the y values [10 9 8]
;; Produces [21 20 19]
(map (fn [x y] (+ x (* 2 y)))
     [1 2 3]    ; these are values x1, x2, x3
     [10 9 8]) ; these are values y1, y2, y3
```

- reduce

```
;; Reduce recursively applies function,
;; to result and next element, i.e.
(reduce + 0 [1 2 3 4])
;; does (+ (+ (+ 0 1) 2) ...
;; and evaluates to 10
```

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])

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

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

$$y_1 = 9, y_2 = 8$$

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

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))
```

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

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

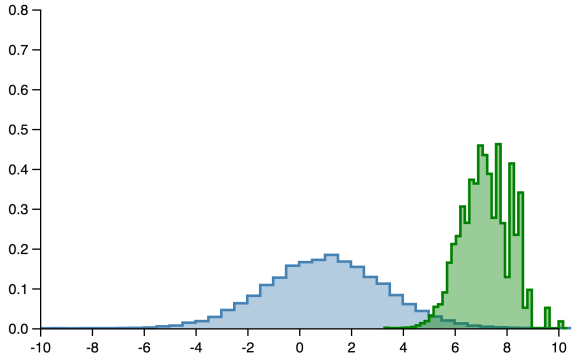
```
(def dataset [9 8])
```

$$y_1 = 9, y_2 = 8$$

```
(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))
```

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

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

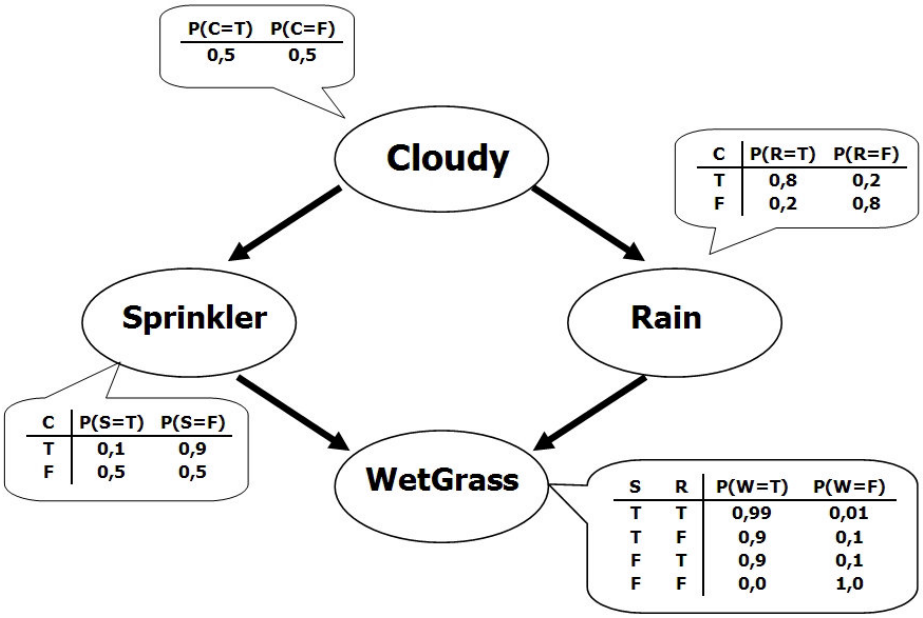
```
(def dataset [9 8])

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

$$y_1 = 9, y_2 = 8$$

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

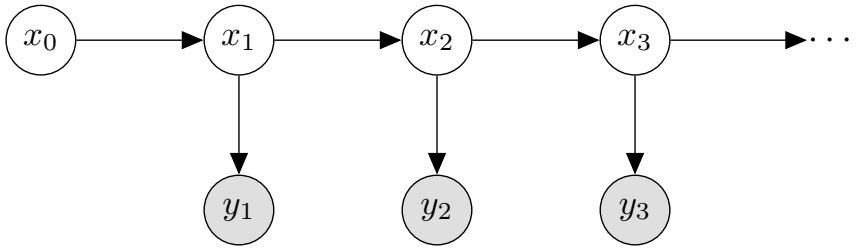
```
(def posterior-samples
  (repeatedly 20000 #(sample posterior)))
```

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))
```

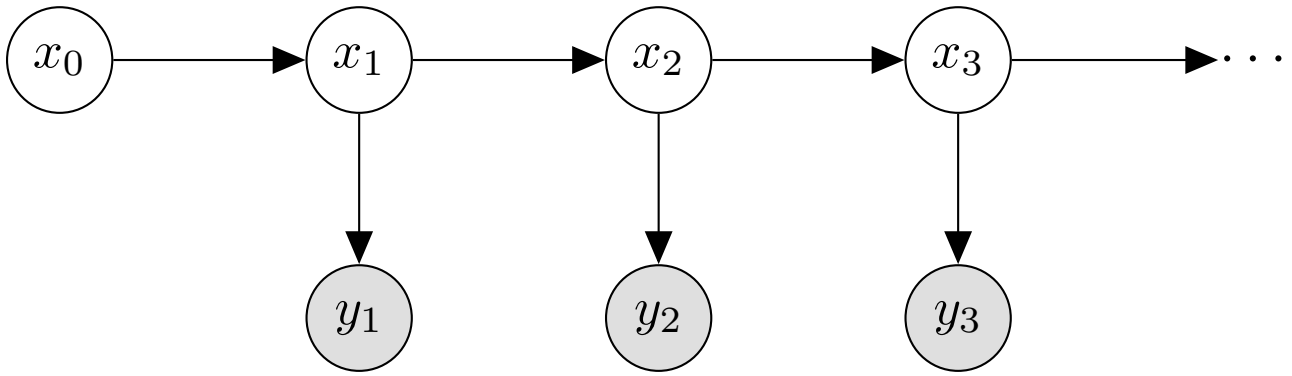
One Hidden Markov Model




```
(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))
```



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).

Evaluation-Based Inference for Higher-Order PPLs

Semantics : Anglican ≠ Clojure

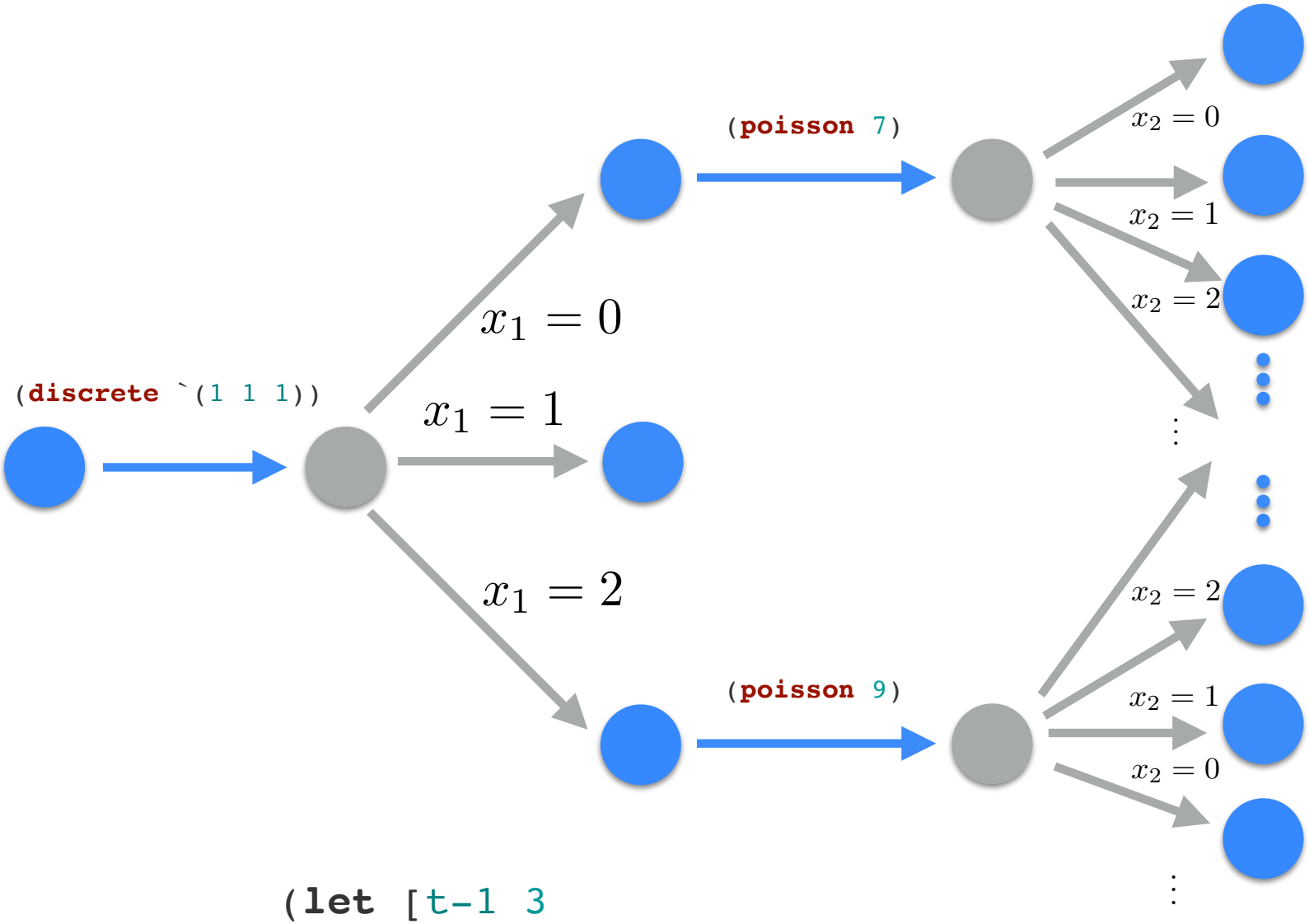
```
(defquery example [param y]
  (let [x (sample (f (theta param)))  ; f(x)
        z (Q x)]
    (observe (g (phi param x)) y)     ; g(y|x)
    (predict :x x)
    (predict :z z)))

(def posterior ((conditional example :method) parameter-value y-value))
(repeatedly K (fn [] sample posterior))
; samples x^k ~ p(x|y) and z^k = Q(x^k)
```

$$\mathbb{E}[z] = \mathbb{E}[Q(\mathbf{x})] = \int Q(\mathbf{x})p(\mathbf{x}|\mathbf{y})d\mathbf{x} = \int Q(\mathbf{x})\pi(\mathbf{x})d\mathbf{x}$$

The Gist

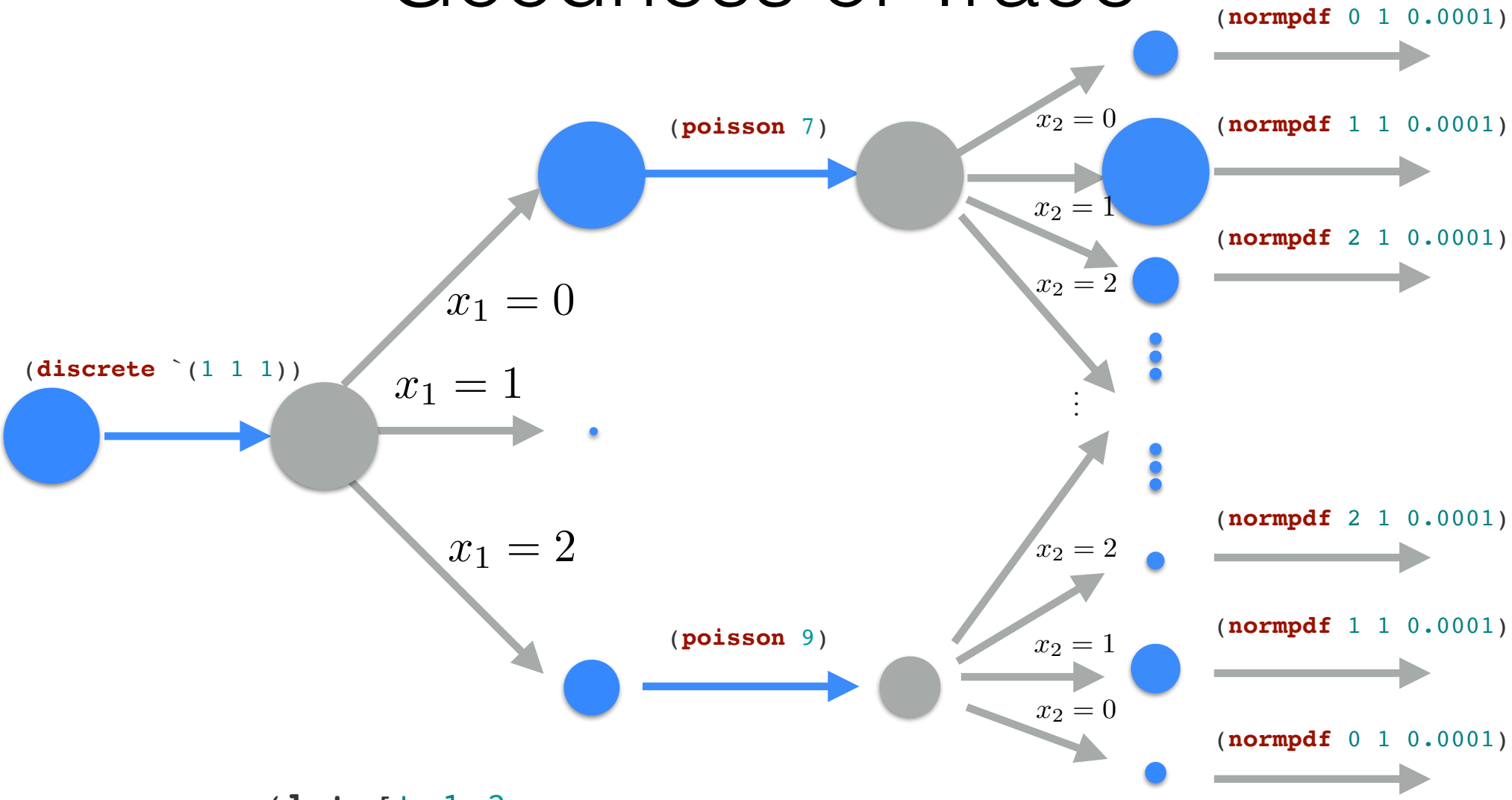
- Explore as many “traces” as possible, intelligently
  - Each trace contains all random choices made during the execution of a generative model
- Compute trace “goodness” (probability) as side-effect
- Combine weighted traces probabilistically coherently
- Report projection of posterior over traces

Traces




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

Goodness of Trace




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

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*

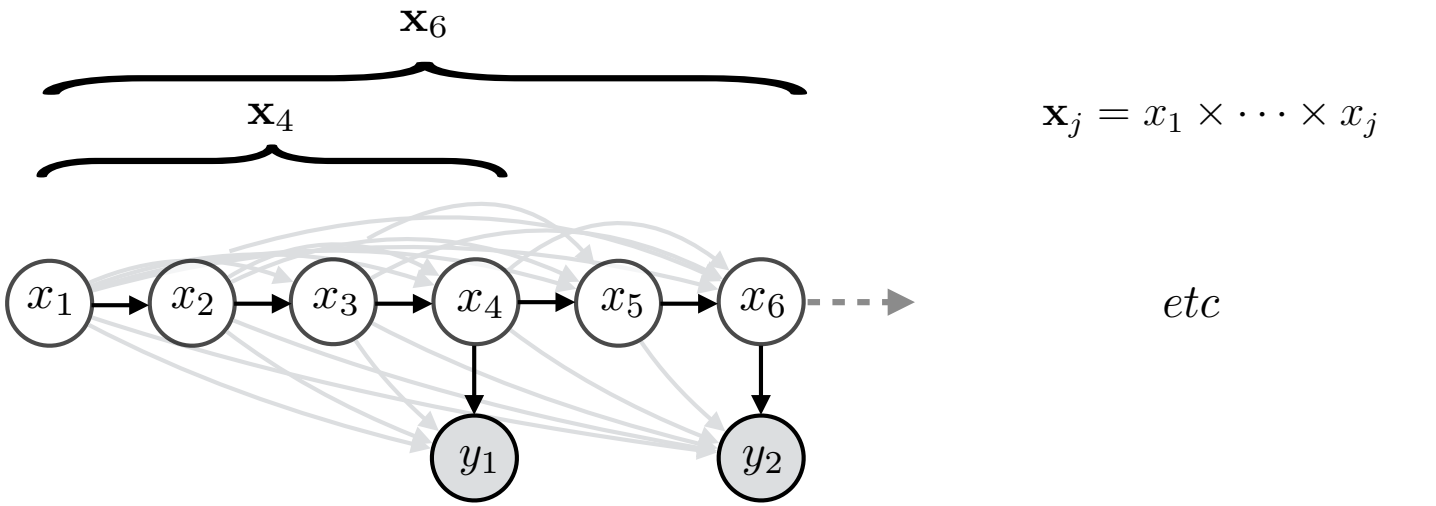
Trace Probability

- 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

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



Inference Goal

- Posterior over traces

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

- 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)

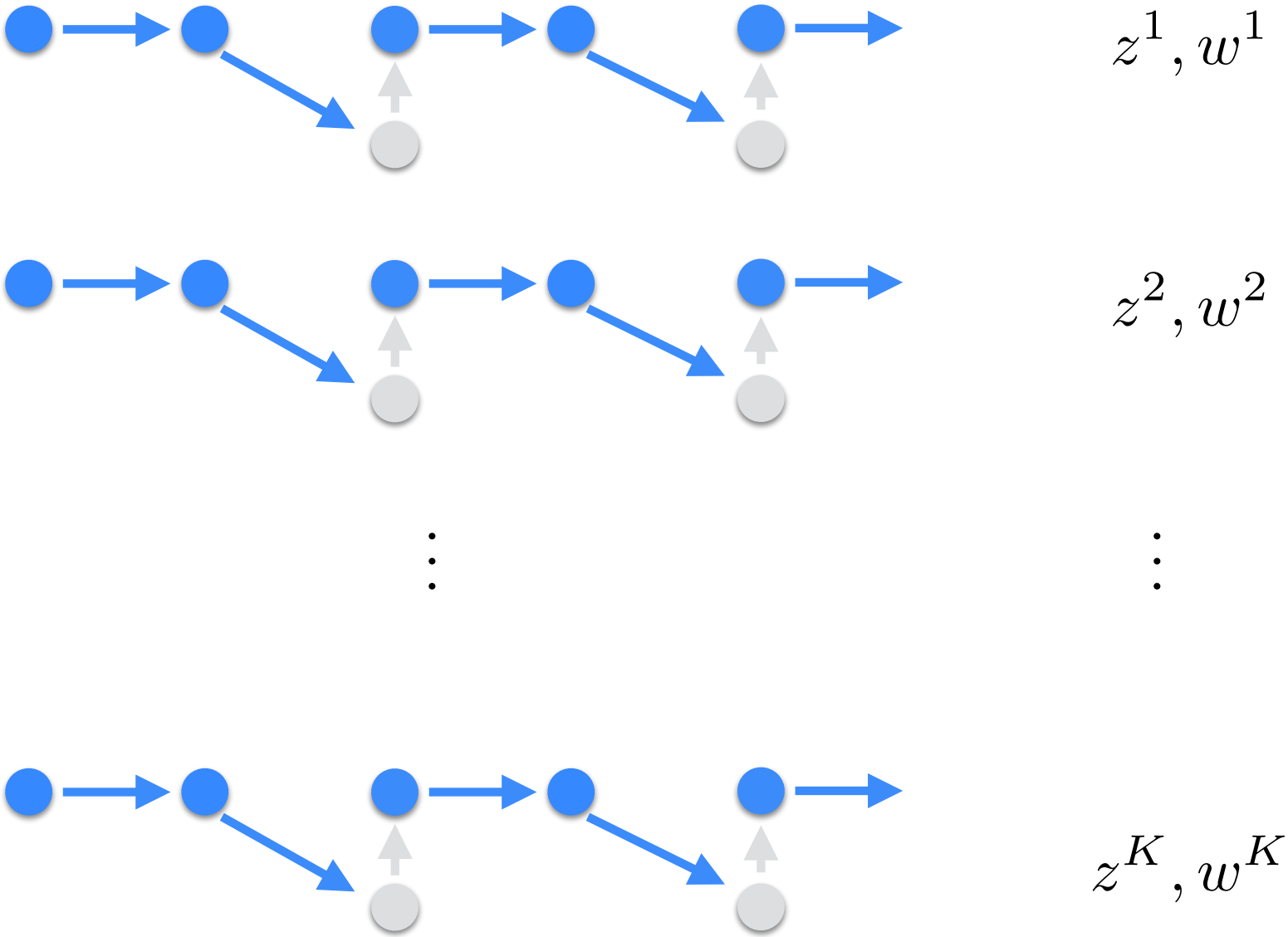
$$w(\mathbf{x}^k) = \frac{\gamma(\mathbf{x}^k)}{q(\mathbf{x}^k)} = \prod_{i=1}^{N^k} g_i^k(y_i^k | \phi_i^k)$$

- 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



Wrap Up

Where We Stand

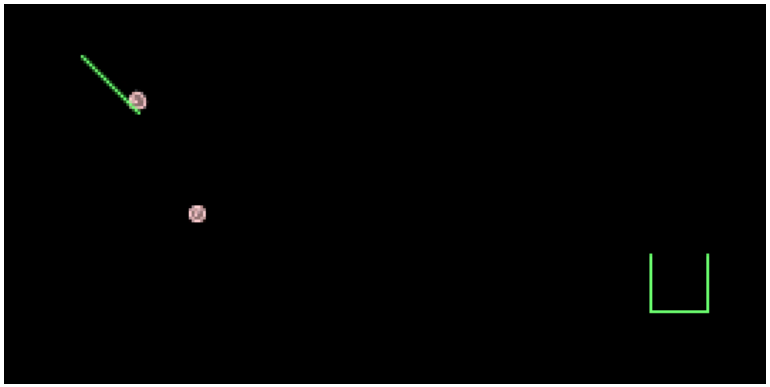
- Probabilistic programming concept
  - Long well established

- 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

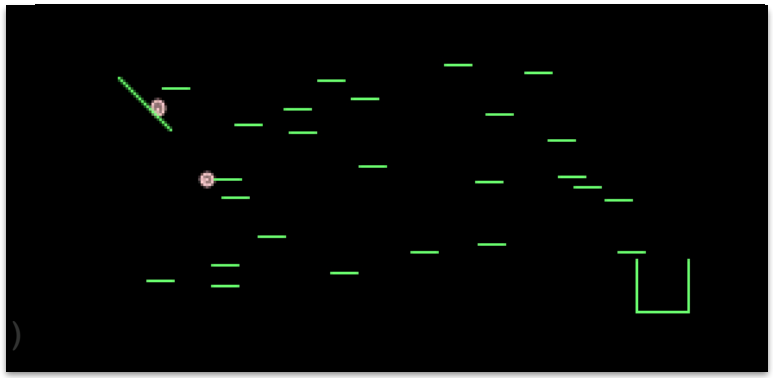
```
(defquery arrange-bumpers []
    (let [bumper-positions []

          ;; 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}))
```

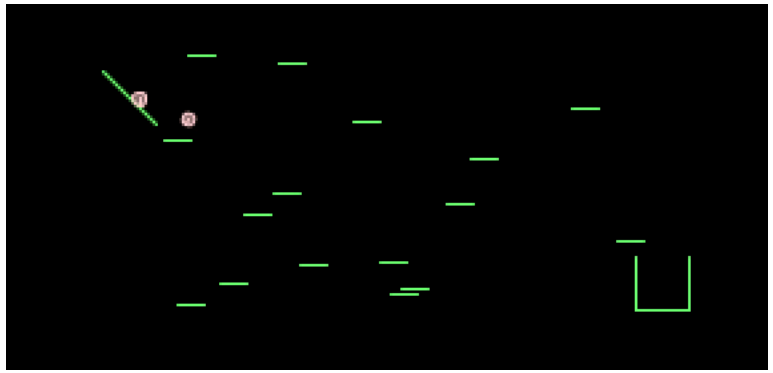


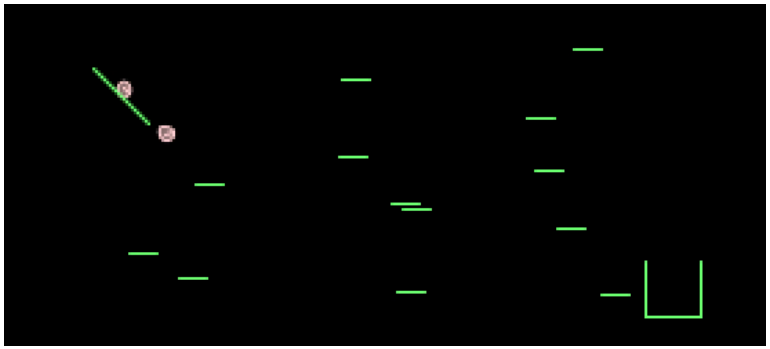
goal: “world” that puts ~20% of balls in box…

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}))
```

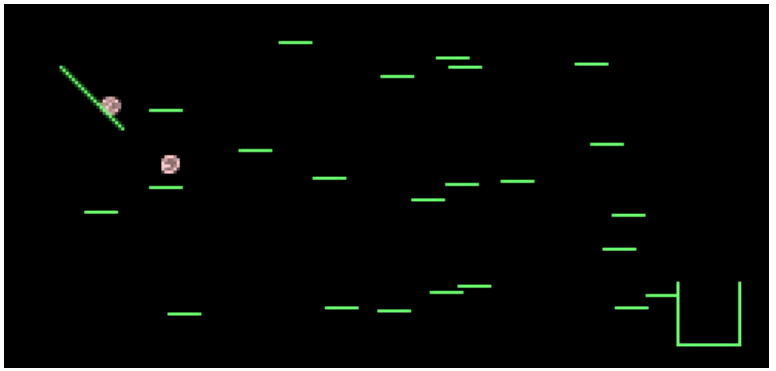


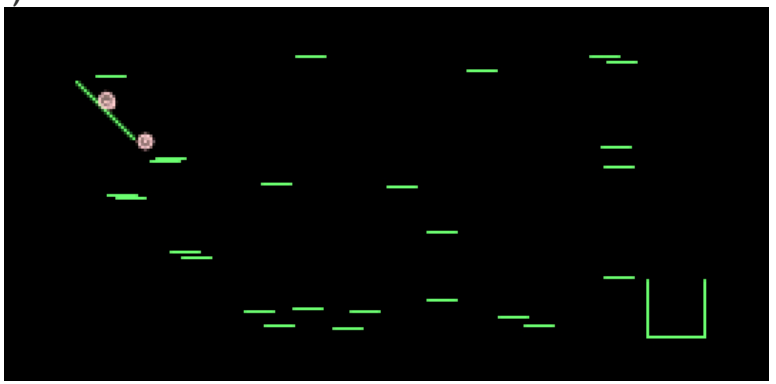


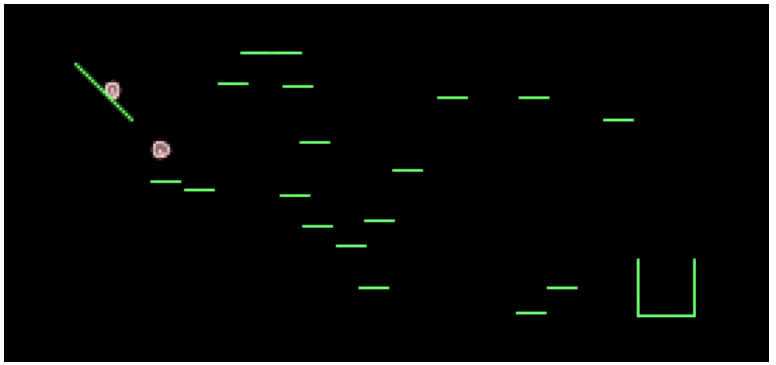


Approximate Bayesian Computation

```
(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

van de Meent

Paige

Tolpin

Yang

Baydin

Le

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

Postdoc Openings

- 2 probabilistic programming postdoc openings

Let's Get Started!

https://github.com/probprog/CSCS-summer-school-2017

Anglican Resources

- General
  - http://www.robots.ox.ac.uk/~fwood/anglican/
- Learning probabilistic programming and Anglican
  - https://bitbucket.org/probprog/mlss2015
- Writing applications
  - https://bitbucket.org/probprog/anglican-user
- The core / looking at inference algorithms
  - https://bitbucket.org/probprog/anglican
- Trying it out (5 min. install)
  - https://bitbucket.org/probprog/anglican-examples