Probabilistic Programming

Hongseok Yang
KAIST

Last part based on work with Chris Heunen, Ohad Kammar and Sam Staton

This Review … discusses some of the state-of-the-art advances in the field, namely, probabilistic programming, Bayesian optimization, data compression and automatic model discovery.

Zoubin Gharahmani
2015 Nature Review

What is probabilistic programming?

(Bayesian) probabilistic modelling of data

1. Develop a new probabilistic (generative) model.
2. Design an inference algorithm for the model.
3. Using the algo., fit the model to the data.

(Bayesian) probabilistic modelling of data

in a prob. prog. language

1. Develop a new probabilistic (generative) model.
2. Design an inference algorithm for the model.
3. Using the algo., fit the model to the data.

(Bayesian) probabilistic modelling of data

in a prob. prog. language

as a program

1. Develop a new probabilistic (generative) model.
2. Design an inference algorithm for the model.
3. Using the algo., fit the model to the data.

(Bayesian) probabilistic modelling of data

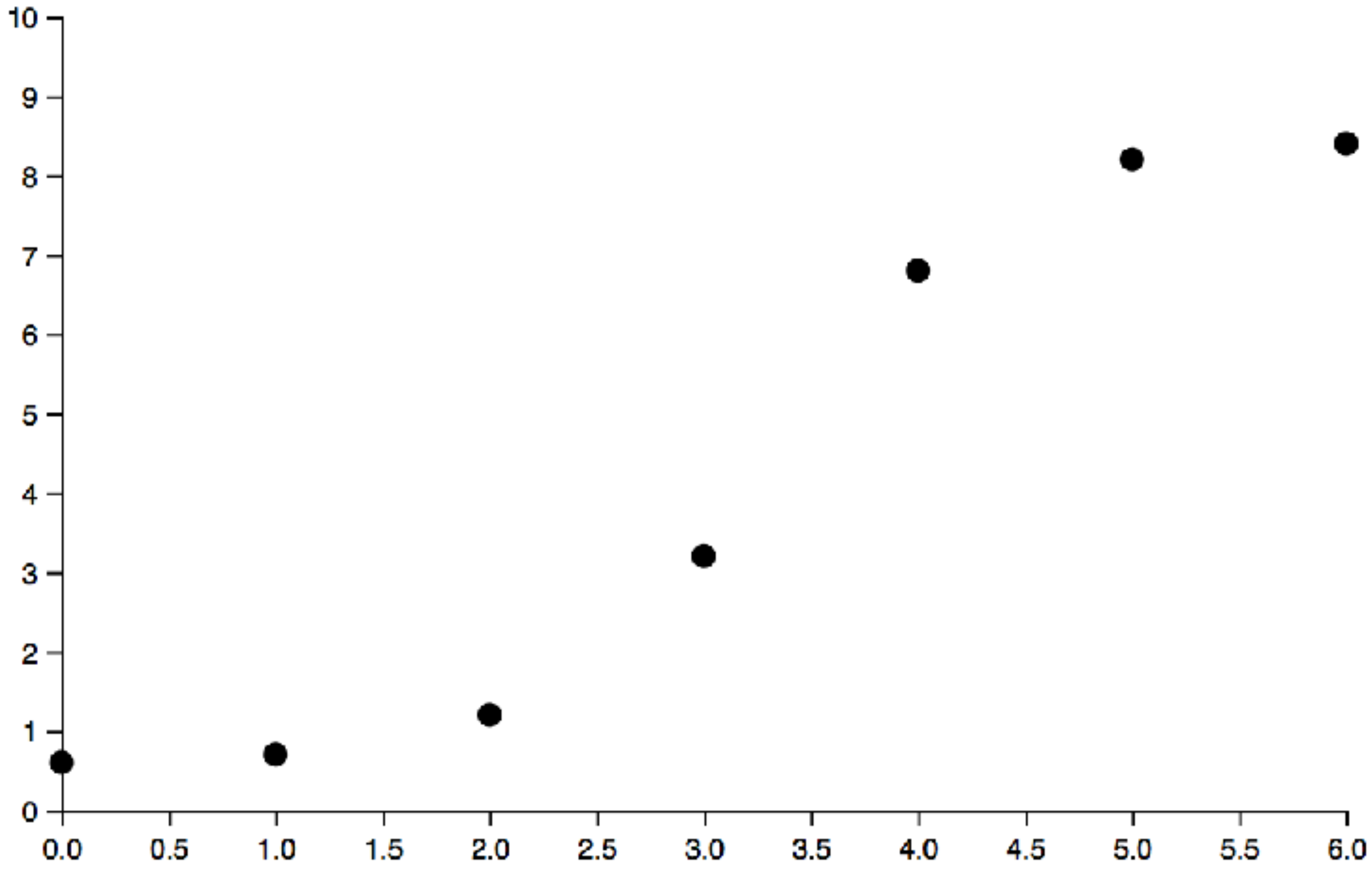
in a prob. prog. language

as a program

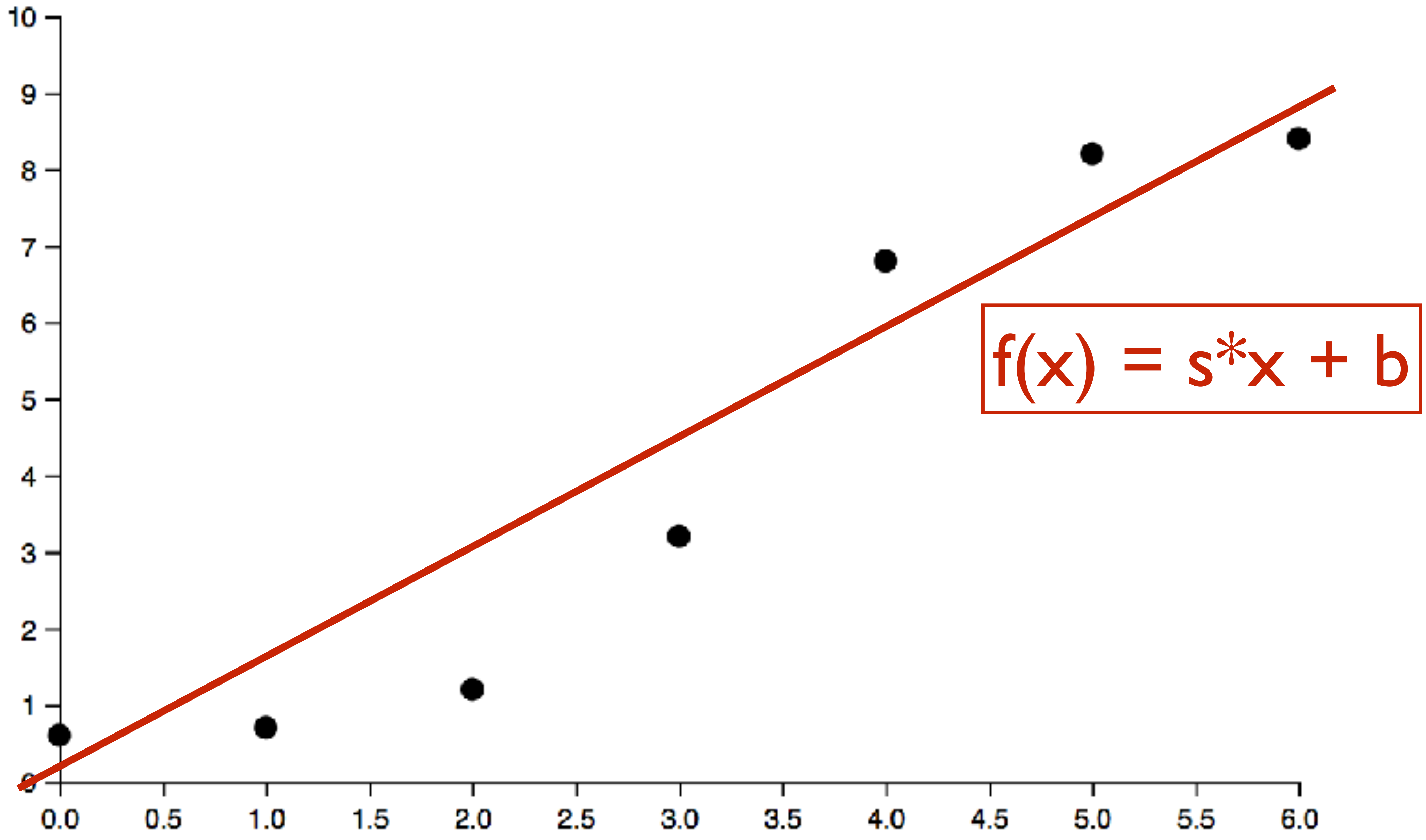
1. Develop a new probabilistic (generative) model.
2. ~~Design an inference algorithm for the model.~~
3. Using ~~the algo~~, fit the model to the data.

a generic inference algo.
of the language

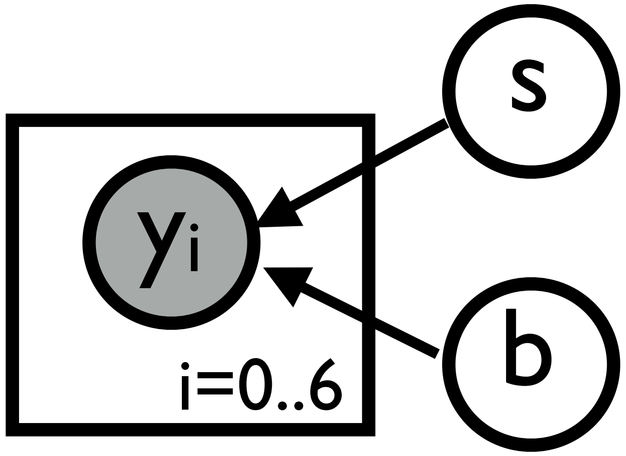
Line fitting



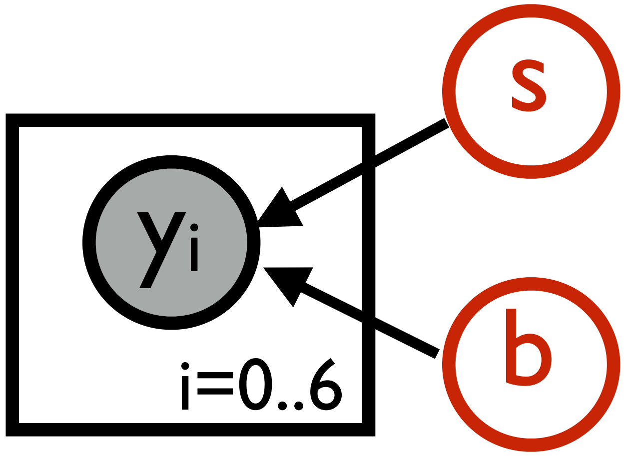
Line fitting



Bayesian generative model

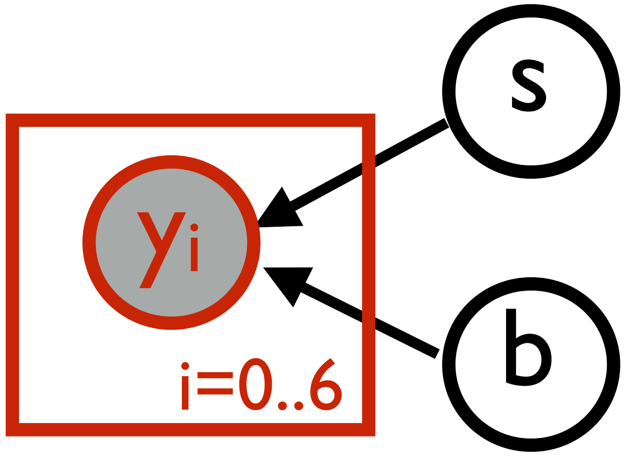


Bayesian generative model



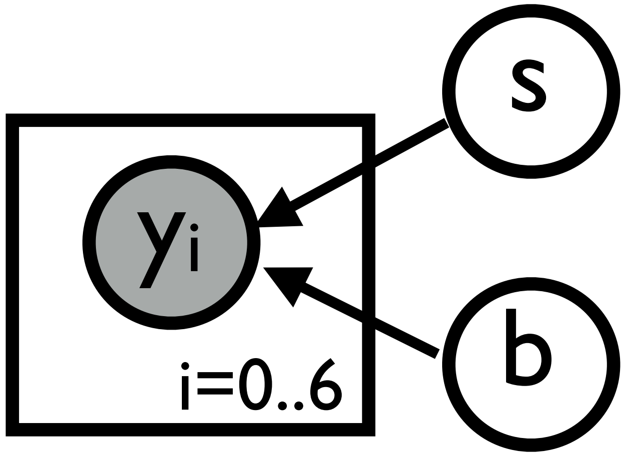
```
s     ~ normal(0, 2)
b     ~ normal(0, 6)
```

Bayesian generative model



```
s     ~ normal(0, 2)
b     ~ normal(0, 6)
f(x)  = s*x + b
$y_i$     ~ normal(f(i), 0.5)
               where i = 0 .. 6
```

Bayesian generative model



```
s     ~ normal(0, 2)
b     ~ normal(0, 6)
f(x)  = s*x + b
y_i    ~ normal(f(i), 0.5)
          where i = 0 .. 6
```

Q: posterior of (s,b) given y_0=0.6, …, y_6=8.4?

Posterior of s and b given y_i's

$$p(s, b \mid y_0, .., y_6) = \frac{p(y_0, .., y_6 \mid s,b) \times p(s,b)}{p(y_0, .., y_6)}$$

Posterior of s and b given y_i's

$$p(s, b \mid y_0, .., y_6) = \frac{p(y_0, .., y_6 \mid s,b) \times p(s,b)}{p(y_0, .., y_6)}$$

Posterior of s and b given y_i's

$$p(s, b \mid y_0, .., y_6) = \frac{p(y_0, .., y_6 \mid s,b) \times p(s,b)}{p(y_0, .., y_6)}$$

Posterior of s and b given y_i's

$$p(s, b \mid y_0, .., y_6) = \frac{p(y_0, .., y_6 \mid s,b) \times p(s,b)}{p(y_0, .., y_6)}$$

Anglican program

```
(let [s (sample (normal 0 2))
      b (sample (normal 0 6))
      f (fn [x] (+ (* s x) b))]
```

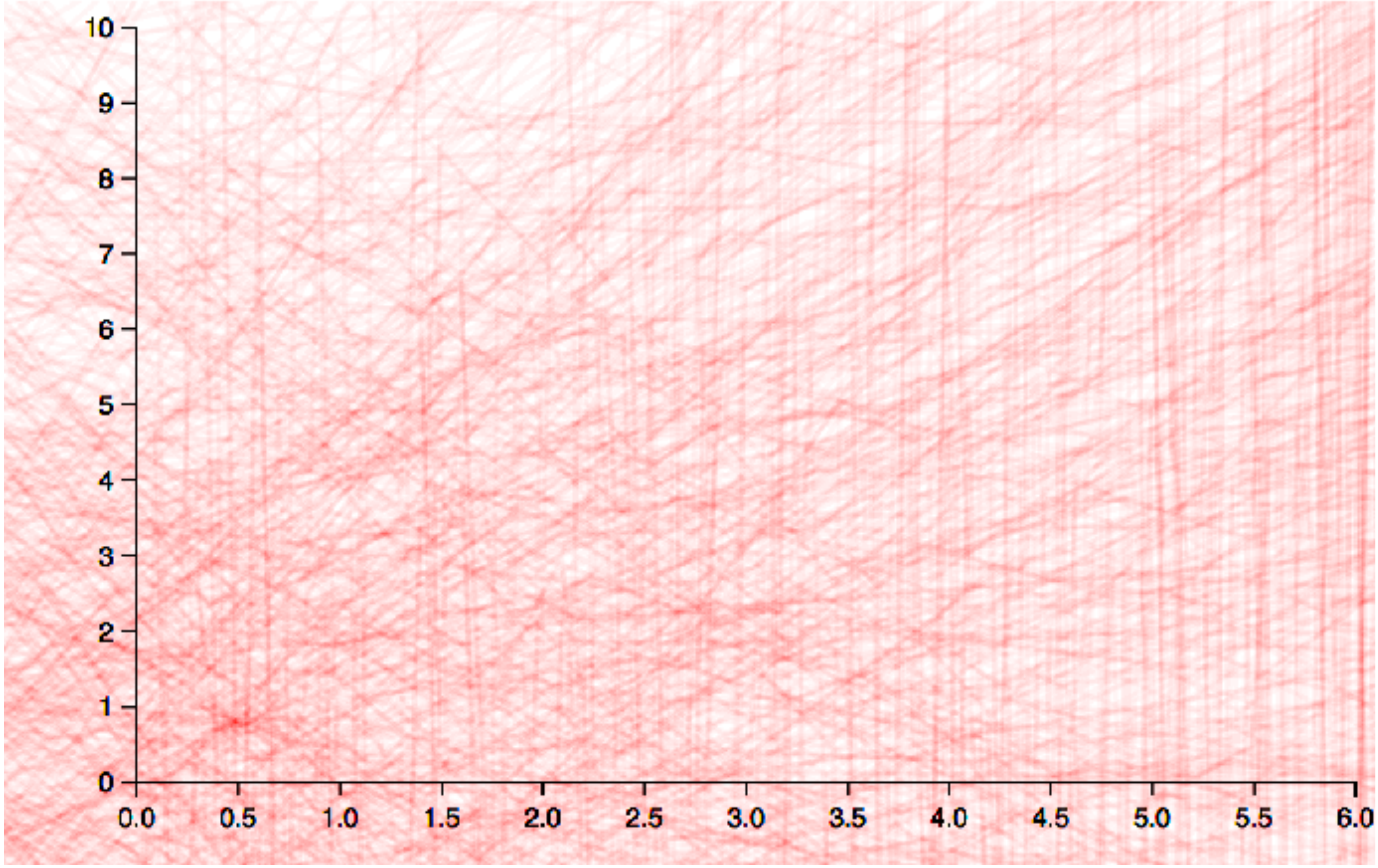
Anglican program

```
(let [s (sample (normal 0 2))
      b (sample (normal 0 6))
      f (fn [x] (+ (* s x) b))]

    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)
```

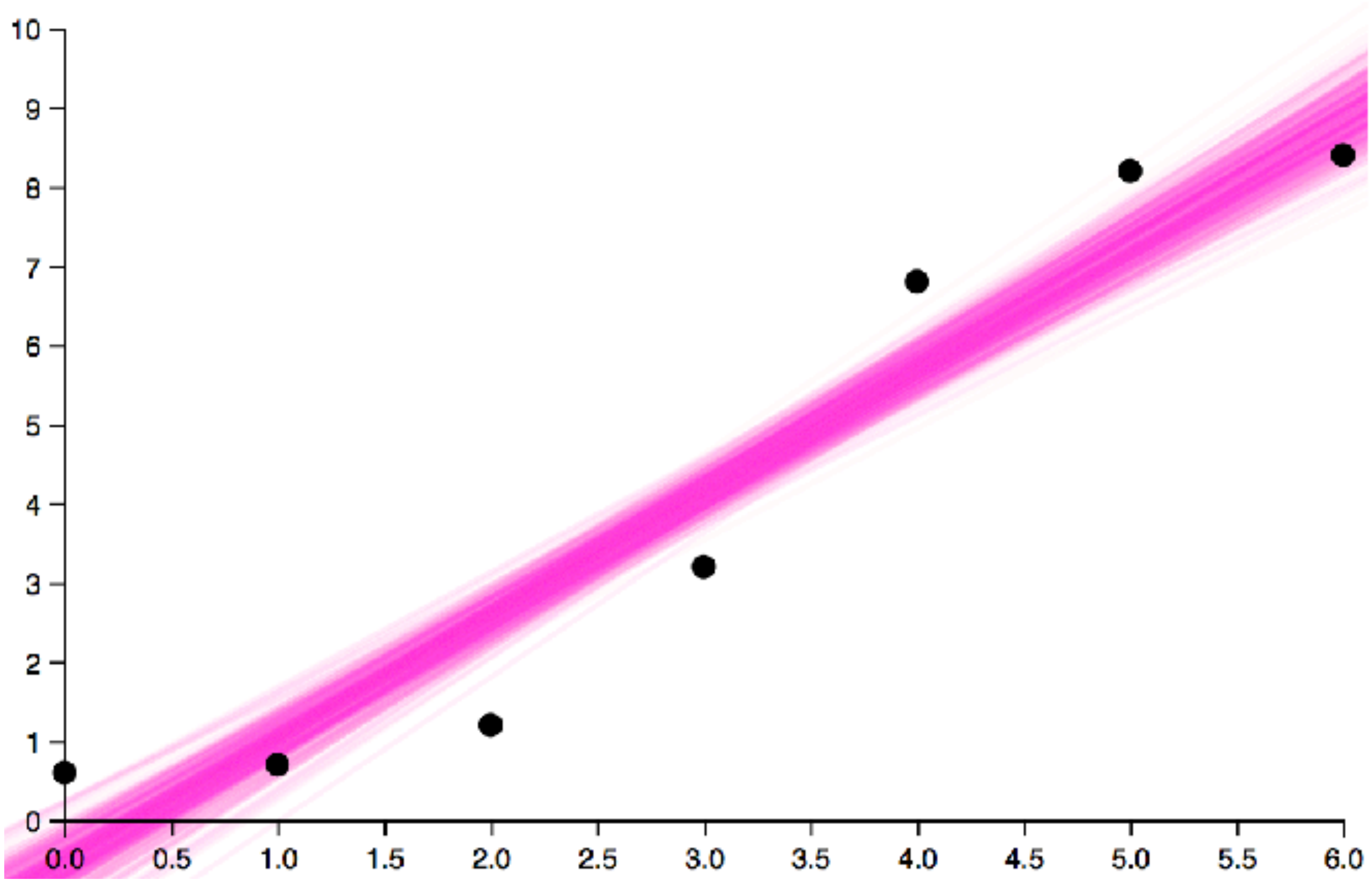
Anglican program

```
(let [s (sample (normal 0 2))
      b (sample (normal 0 6))
      f (fn [x] (+ (* s x) b))]

    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  [s b])
```

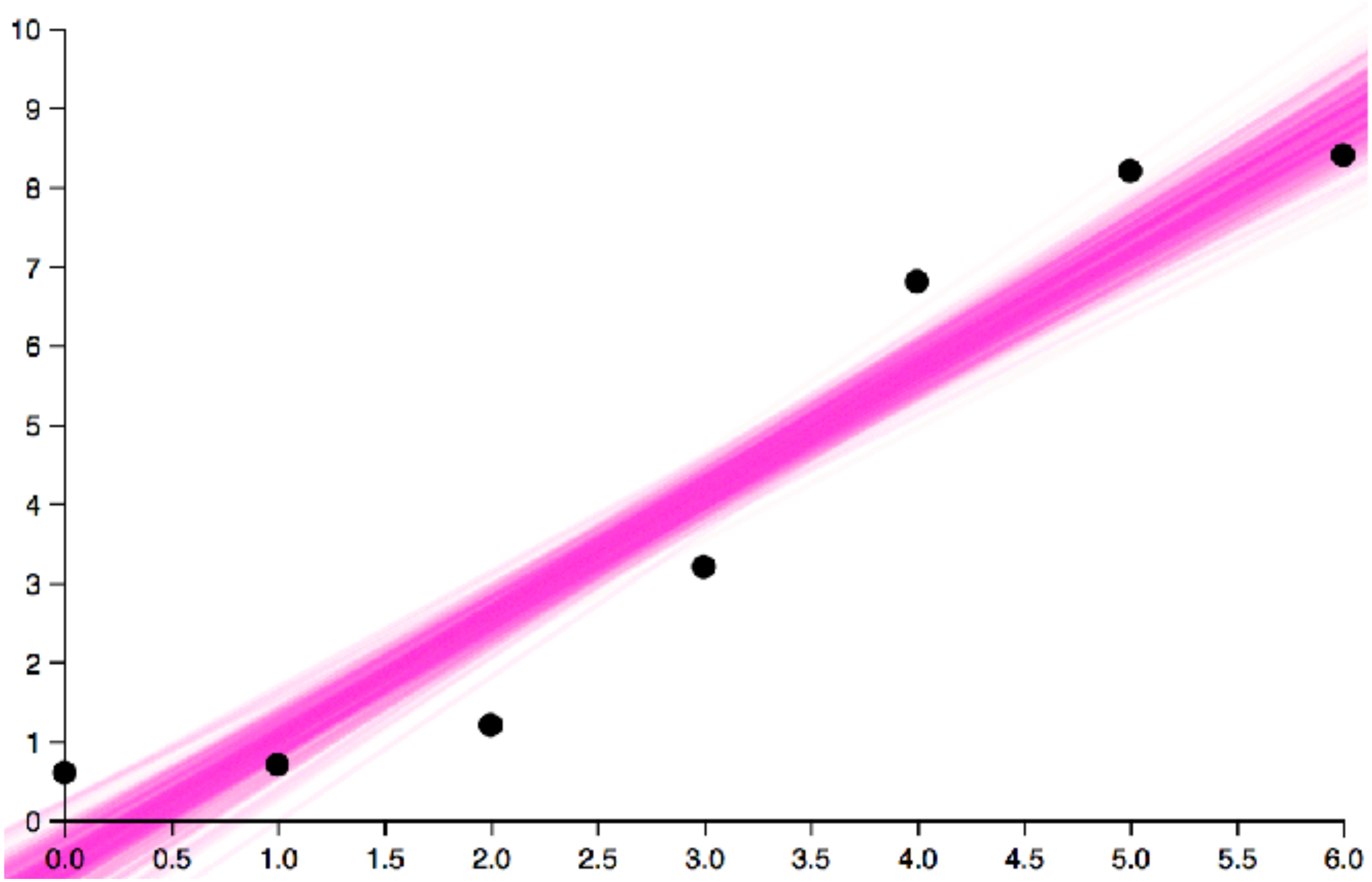
Samples from prior



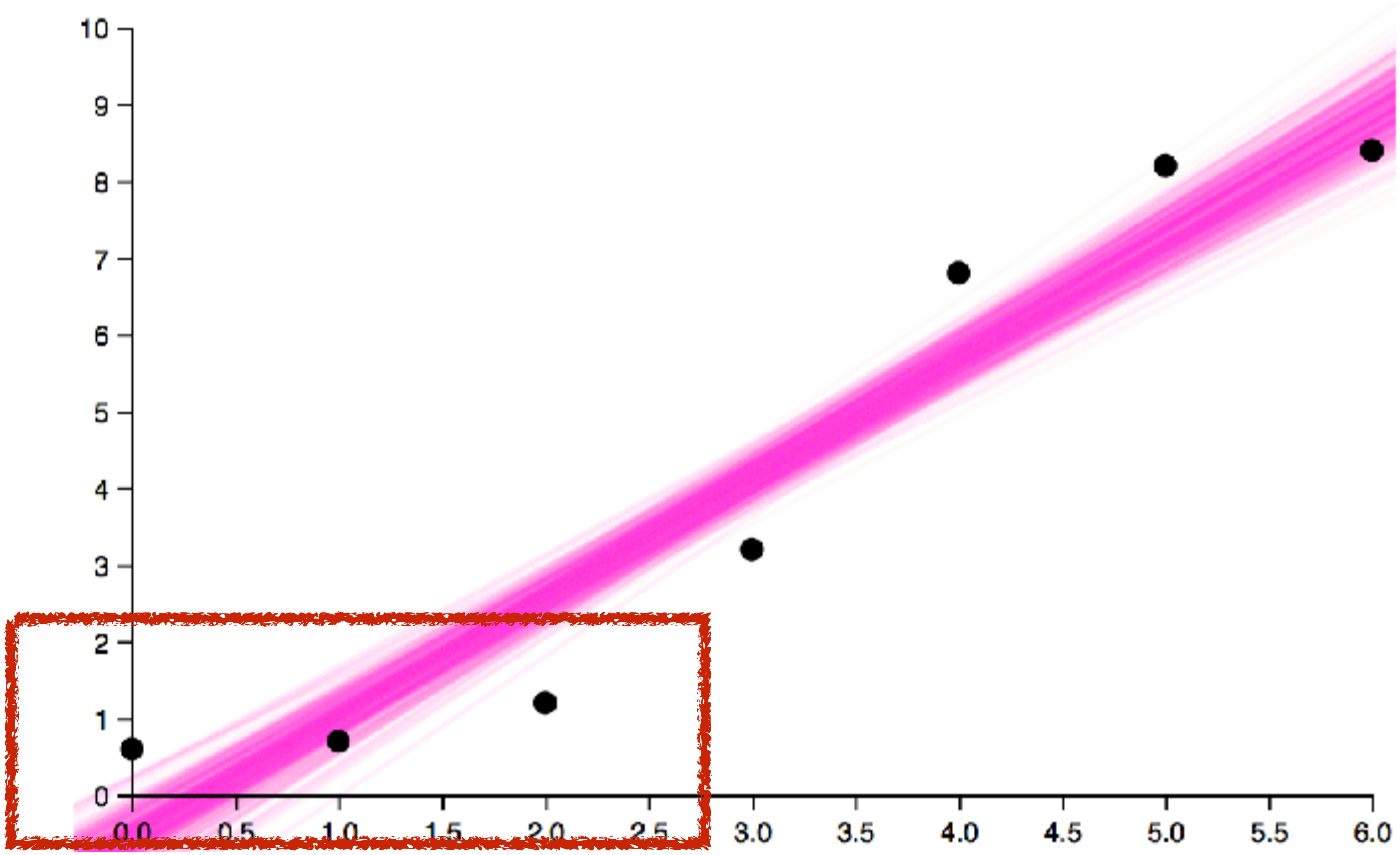
Samples from posterior



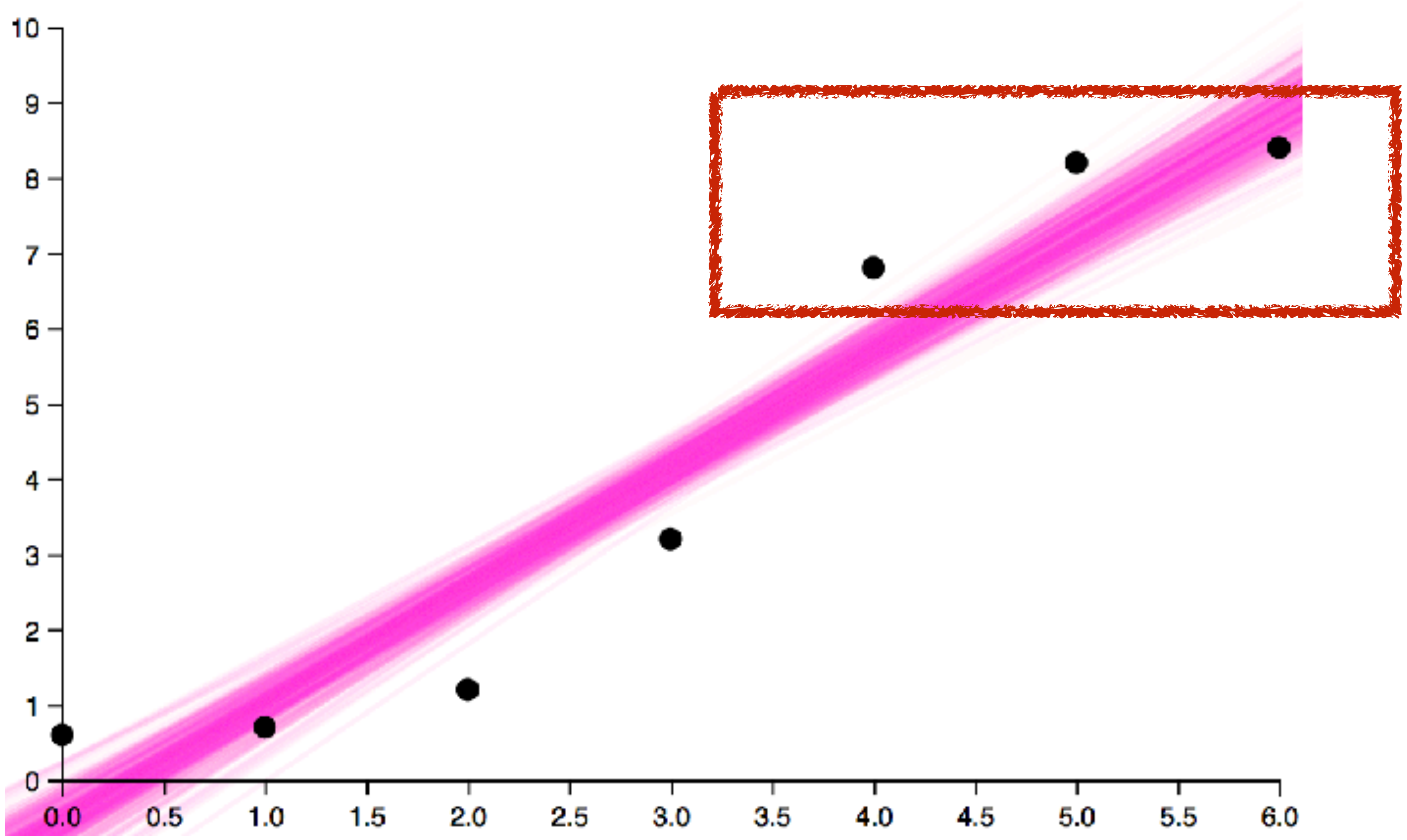
Underfit?



Underfit?



Underfit?



Expressive prob. PLs enable one to explore advanced models easily.

```
(let [s (sample (normal 0 2))
      b (sample (normal 0 6))
      f (fn [x] (+ (* s x) b))]


    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  [s b])
```

```
(let [s (sample (normal 0 2))
      b (sample (normal 0 6))
      f (fn [x] (+ (* s x) b))]


    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  [s b])
```

Anglican fully supports higher-order functions

```
(let [s (sample (normal 0 2))
      b (sample (normal 0 6))
      f (fn [x] (+ (* s x) b))]

    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  ~~[s b])~~
  f)
```

Anglican fully supports higher-order functions

```
(let [F (fn []
          (let [s (sample (normal 0 2))
                b (sample (normal 0 6))]
            (fn [x] (+ (* s x) b))))
      f (F)]
    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  ~~[s b])~~
  f)
```

Anglican fully supports
higher-order functions

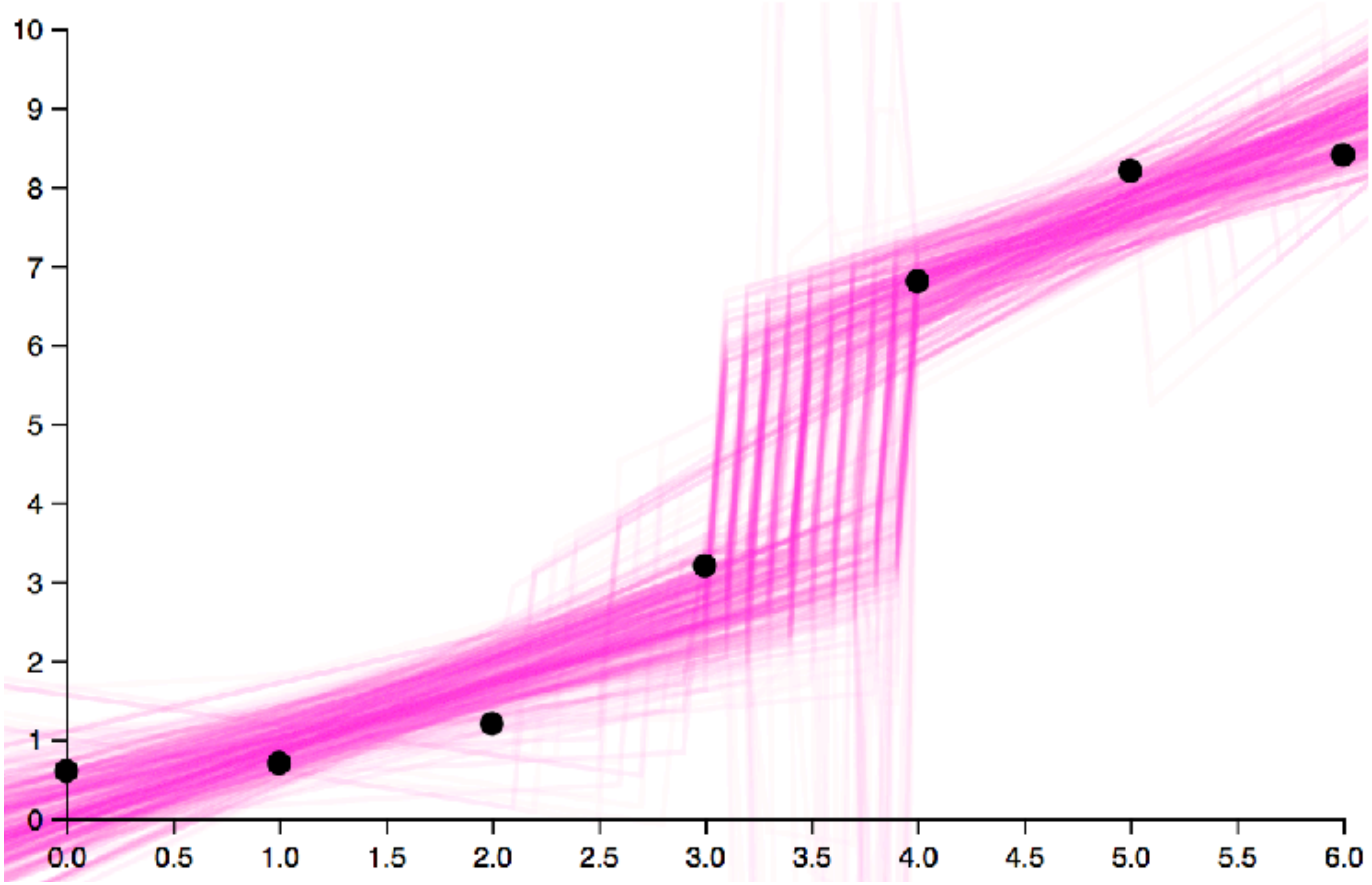
```
(let [F (fn []
          (let [s (sample (normal 0 2))
                b (sample (normal 0 6))]
            (fn [x] (+ (* s x) b))))
      f (add-change-points F 0 6)]
    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)
```

~~[s b])~~

```
  f)
```

Anglican fully supports higher-order functions

Samples from posterior



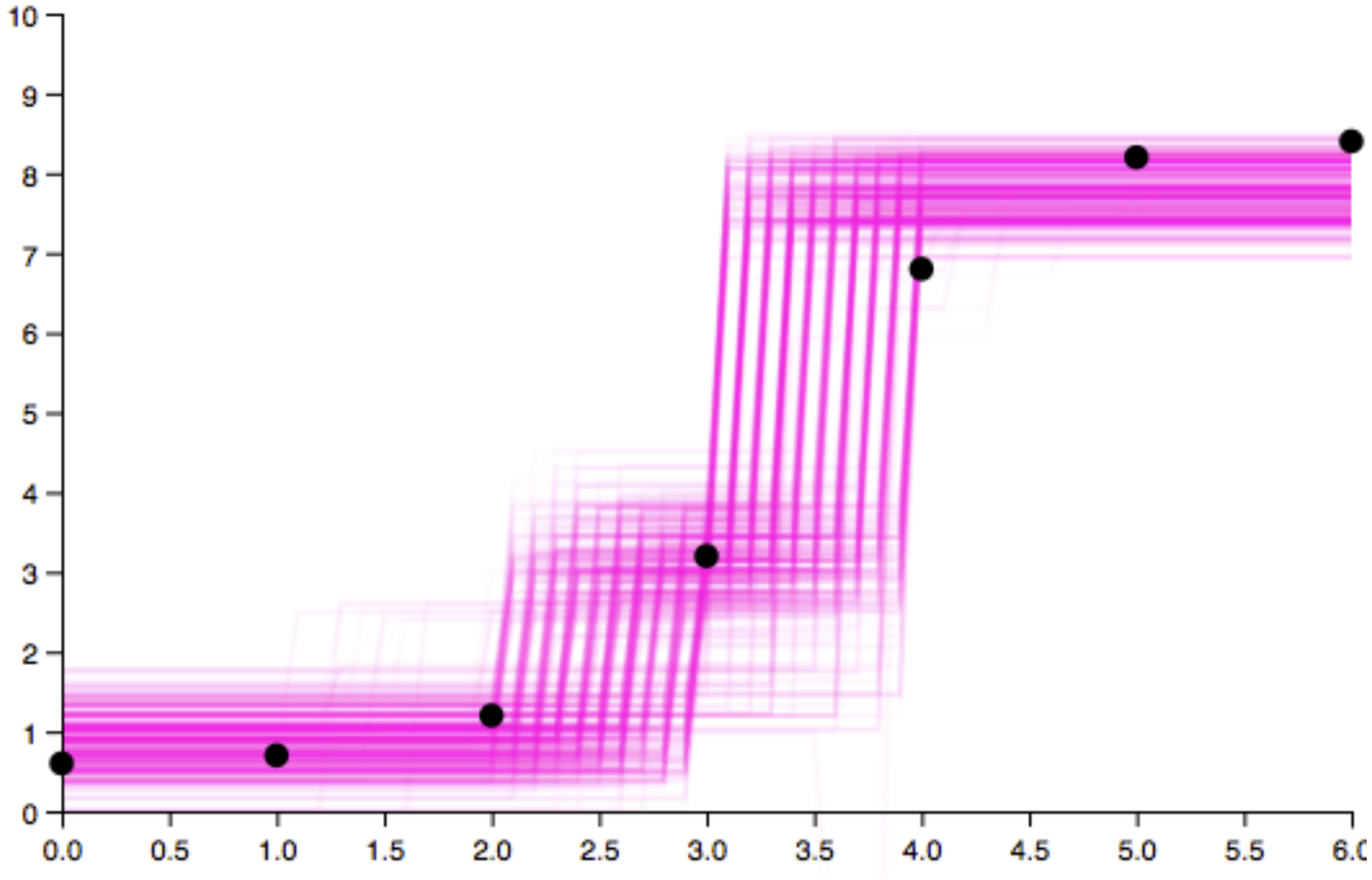
```
(let [F (fn []
          (let [s (sample (normal 0 2))
                b (sample (normal 0 6))]
            (fn [x] (+ (* s x) b))))
      f (add-change-points F 0 6)]
    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  ~~[s b])~~
  f)
```

Anglican fully supports higher-order functions

```
(let [F (fn []
          (let [~~s (sample (normal 0 2))~~
                b (sample (normal 0 6))]
            (fn [x] (+ ~~(* s x)~~ b))))
      f (add-change-points F 0 6)]
    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  ~~[s b])~~
  f)
```

Anglican fully supports higher-order functions

Samples from posterior



Captcha breaking

→ SMKBDF

Le, Baydin, Wood [2016]

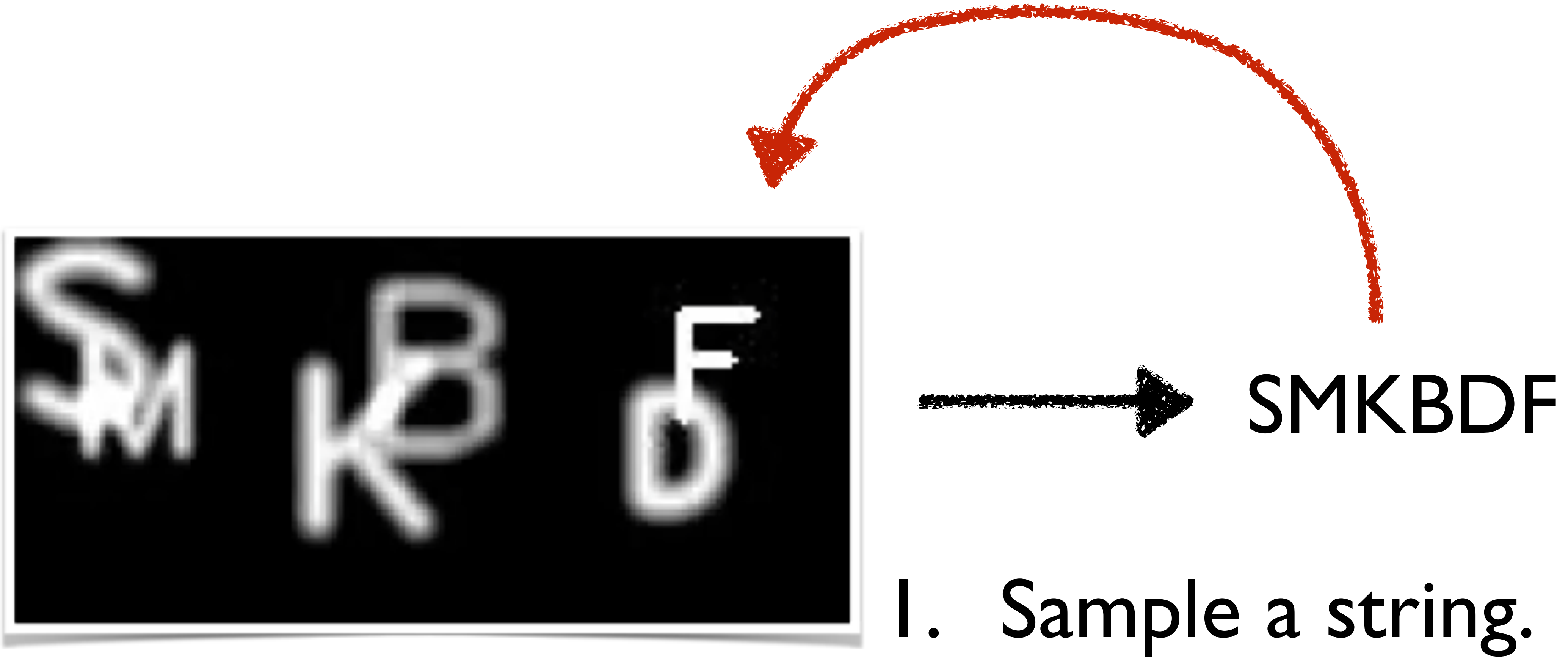
Captcha breaking

→ SMKBDF

1. Sample a string.

Le, Baydin, Wood [2016]

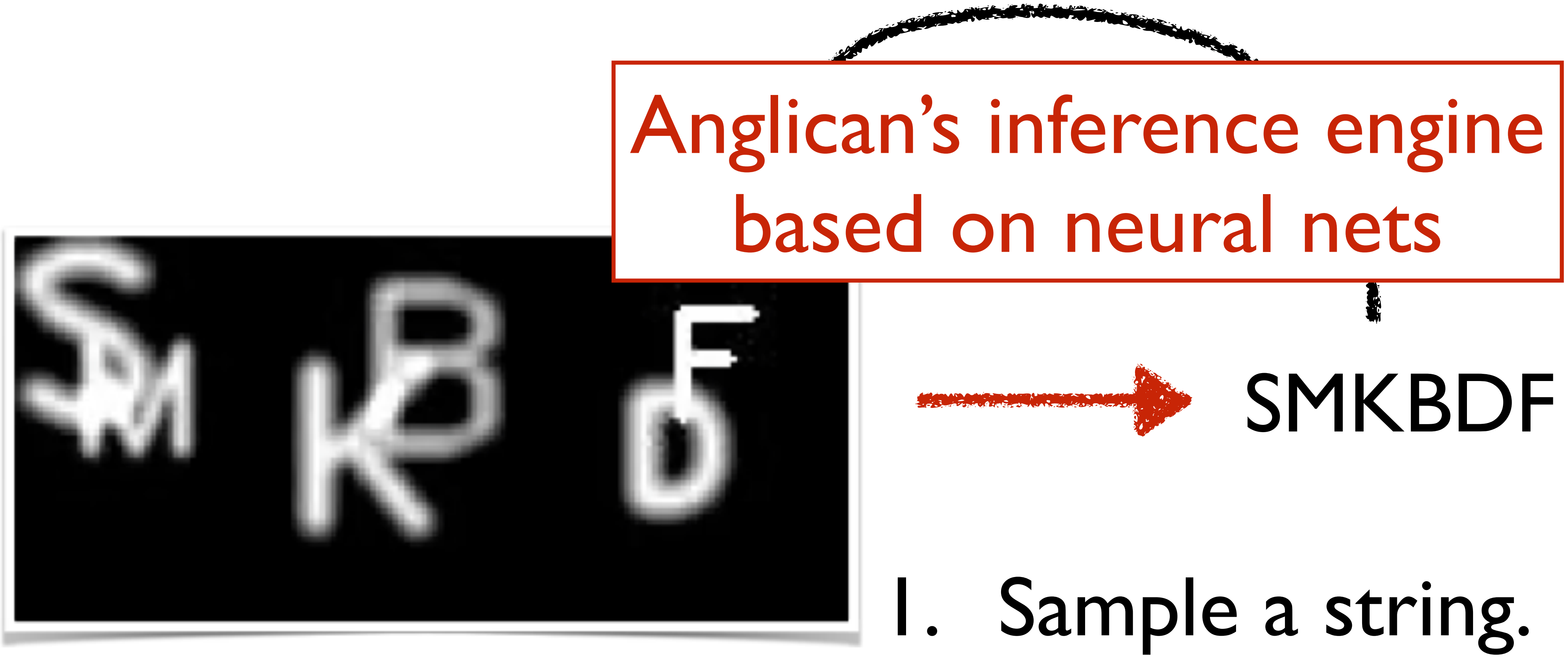
Captcha breaking



1. Sample a string.
2. Generate an image using complex JVM code.

Le, Baydin, Wood [2016]

Captcha breaking



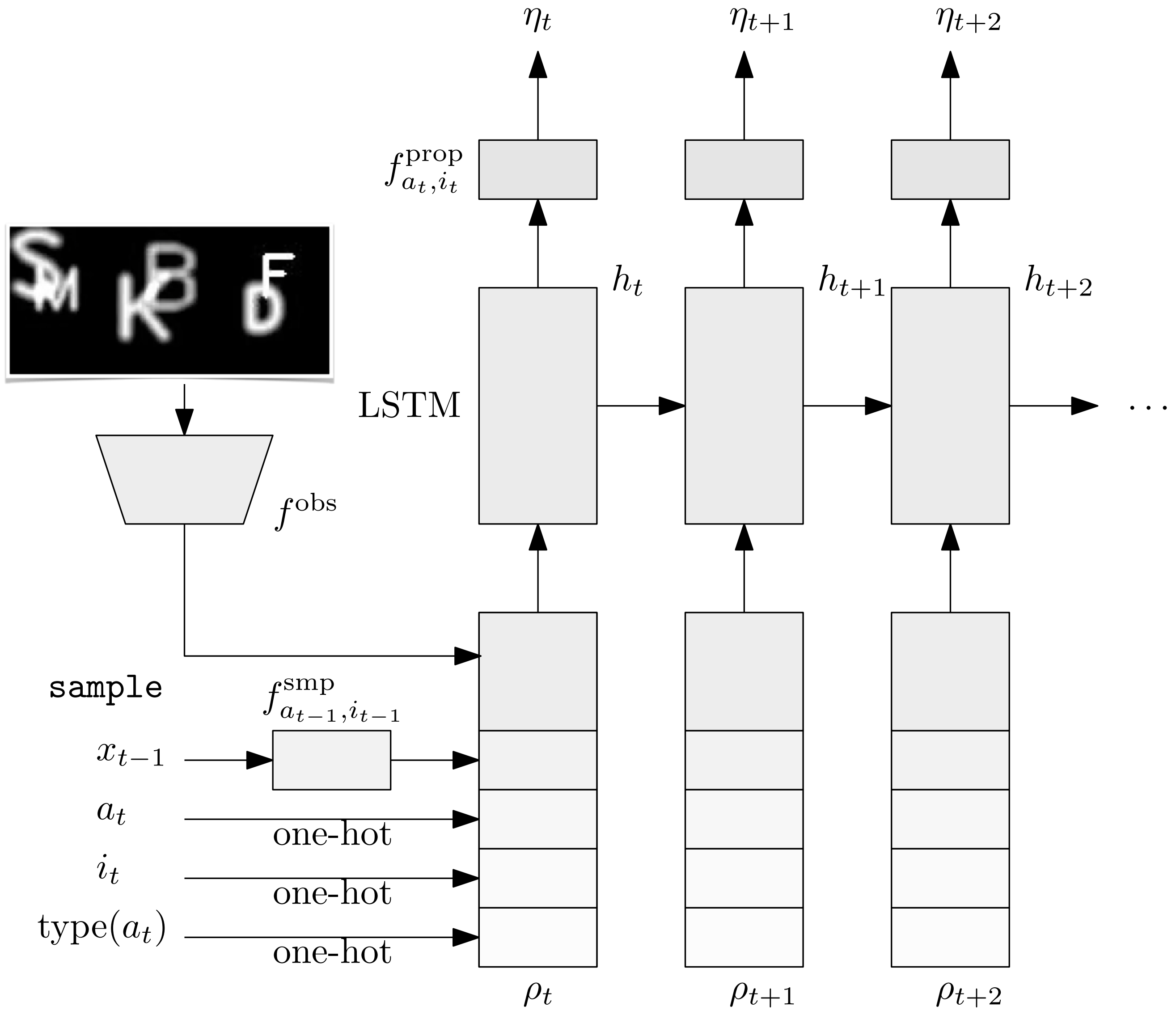
Anglican’s inference engine based on neural nets

SMKBDF

1. Sample a string.
2. Generate an image using complex JVM code.

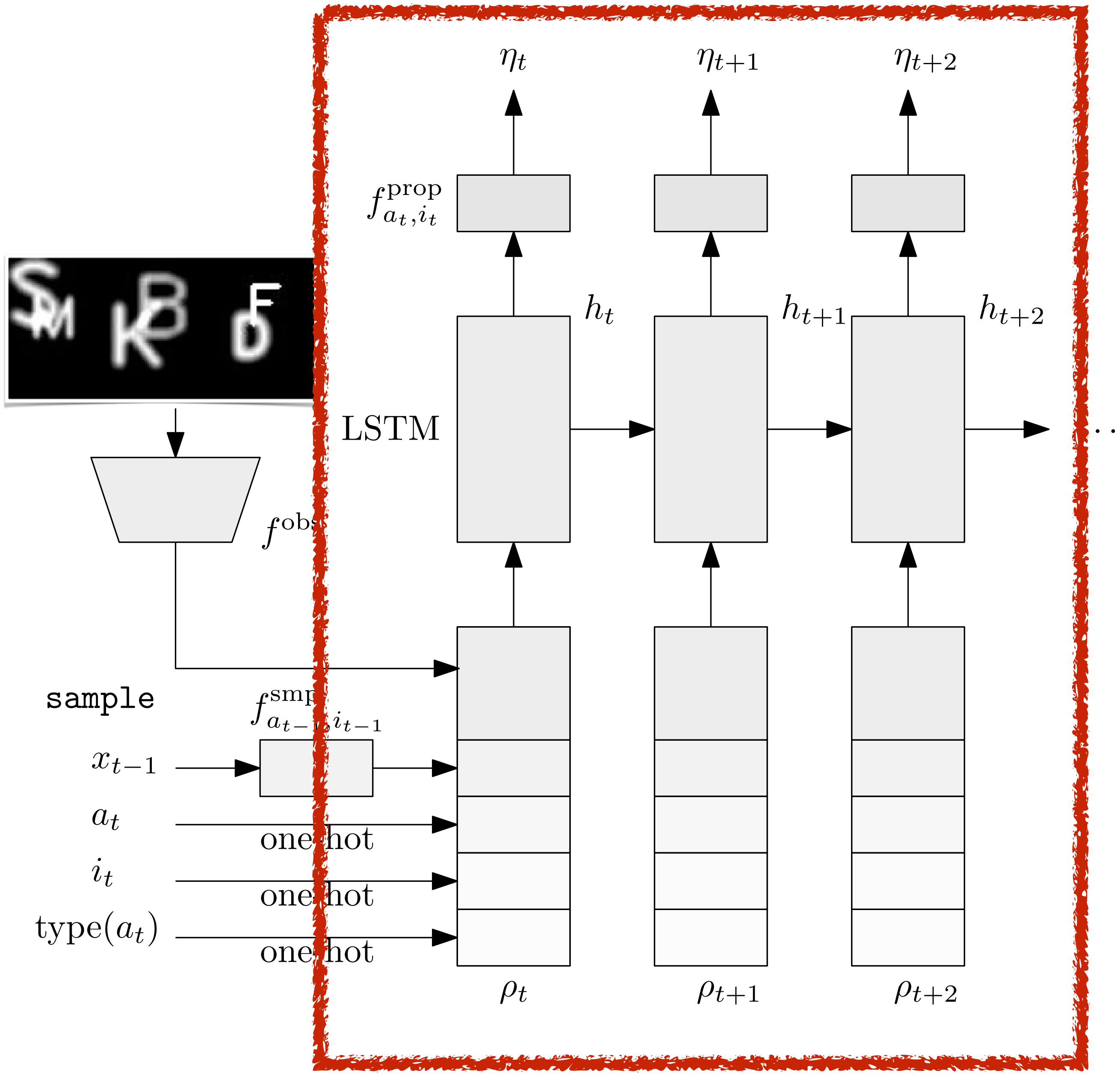
Le, Baydin, Wood [2016]

Captcha breaking



Le, Baydin, Wood [2016]

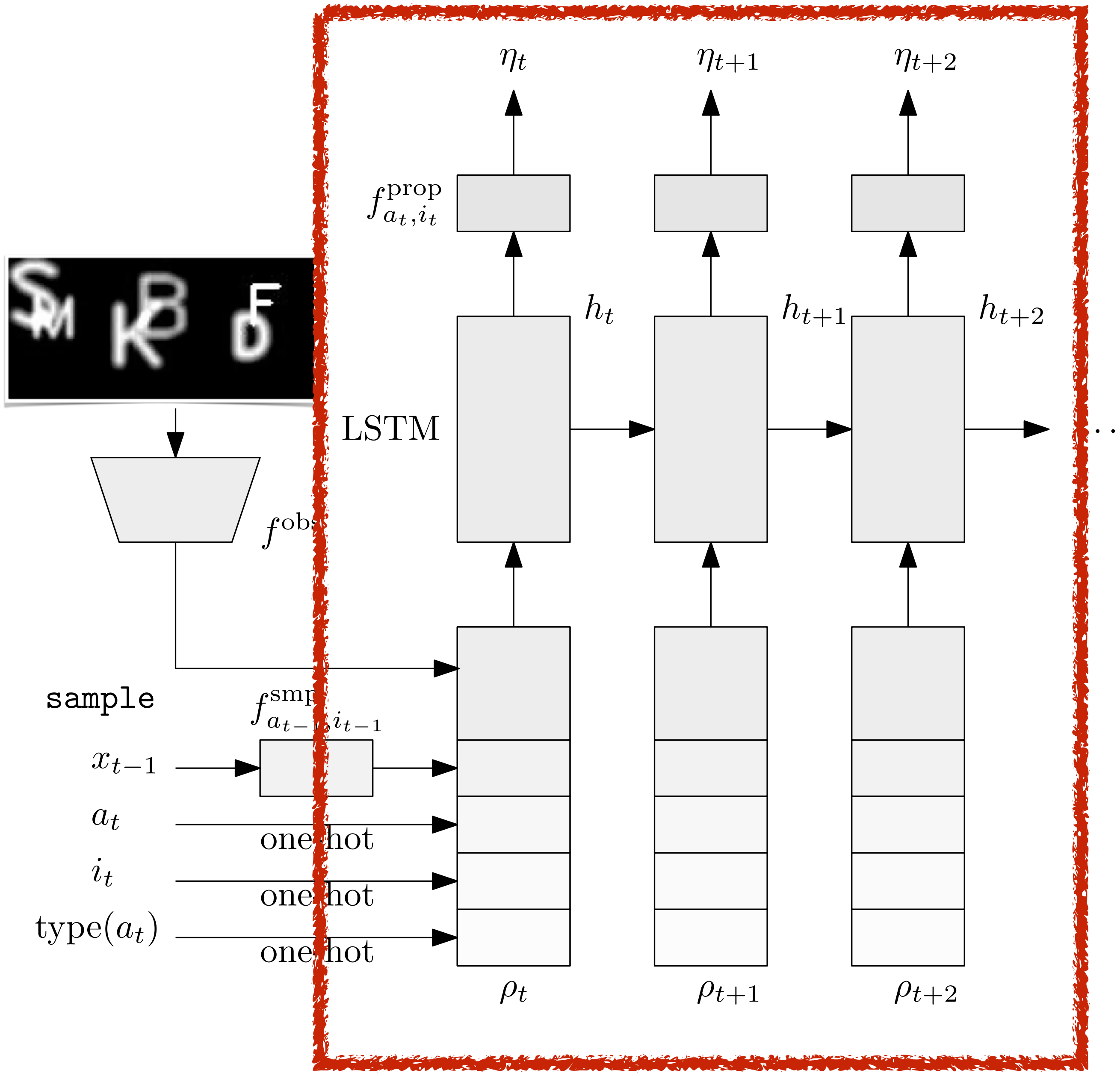
Captcha breaking



Neural net as a part of inference engine.

Le, Baydin, Wood [2016]

Captcha breaking

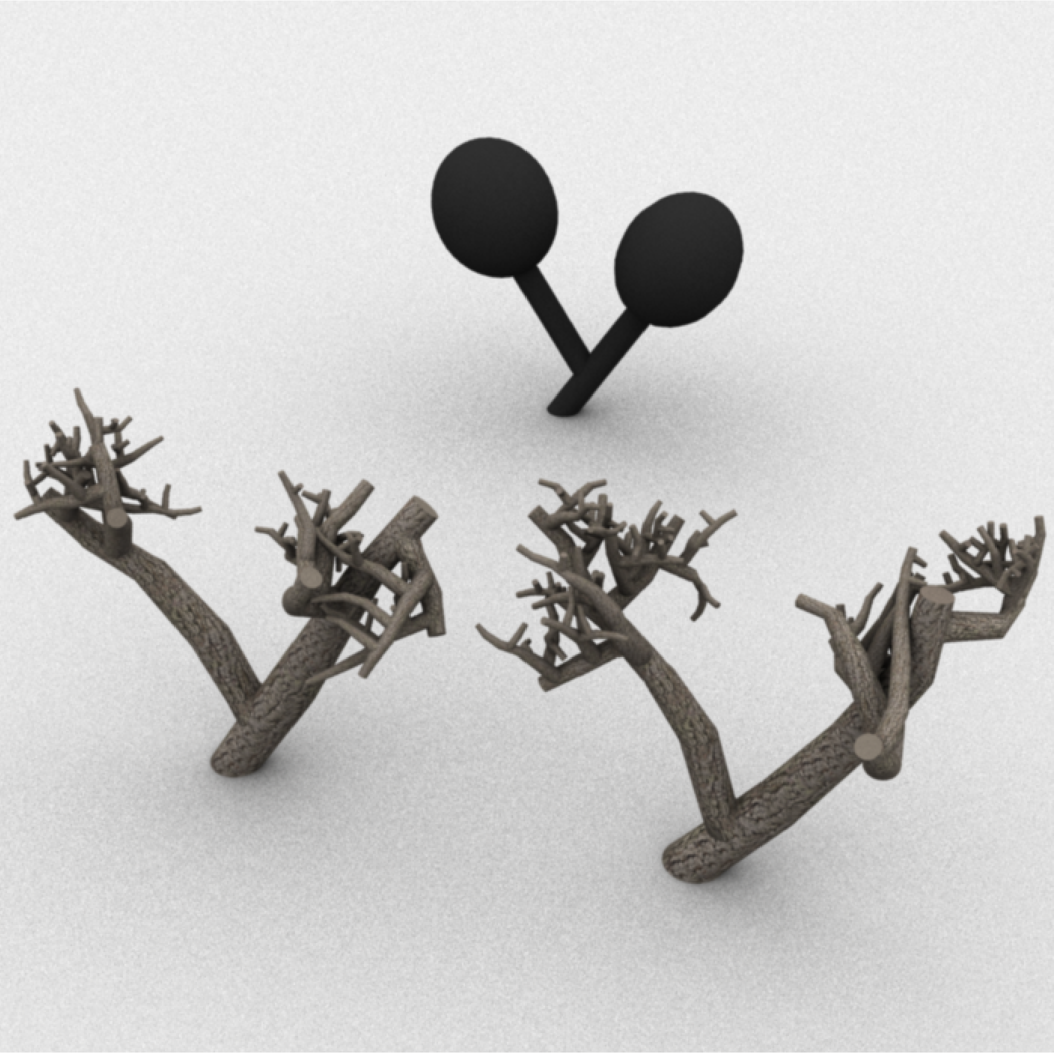


Neural net as a part of inference engine.

Approximates the inverse of the Captcha program.

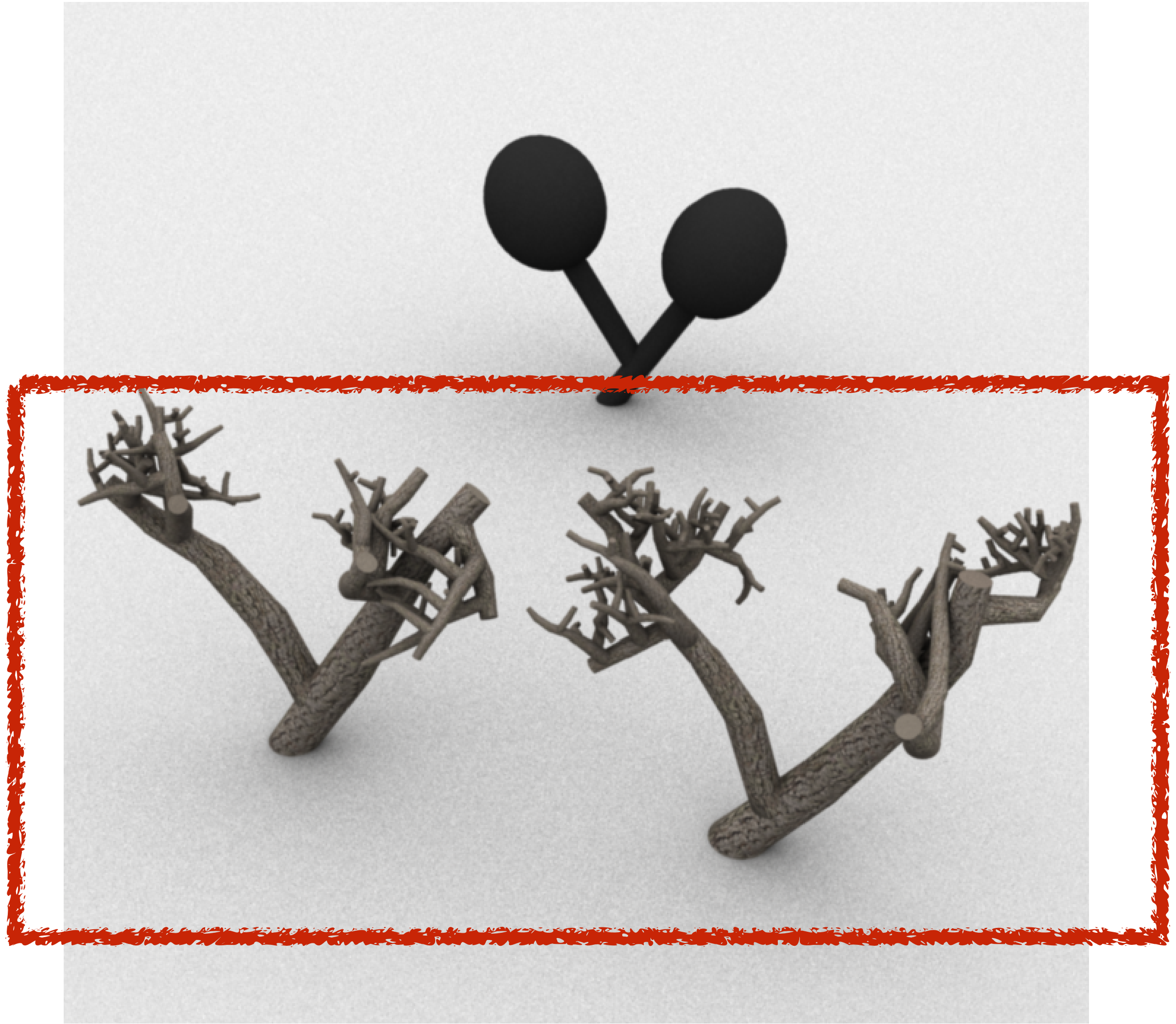
Le, Baydin, Wood [2016]

Procedural modelling



Ritchie, Mildenhall, Goodman,
Hanrahan [SIGGRAPH'15]

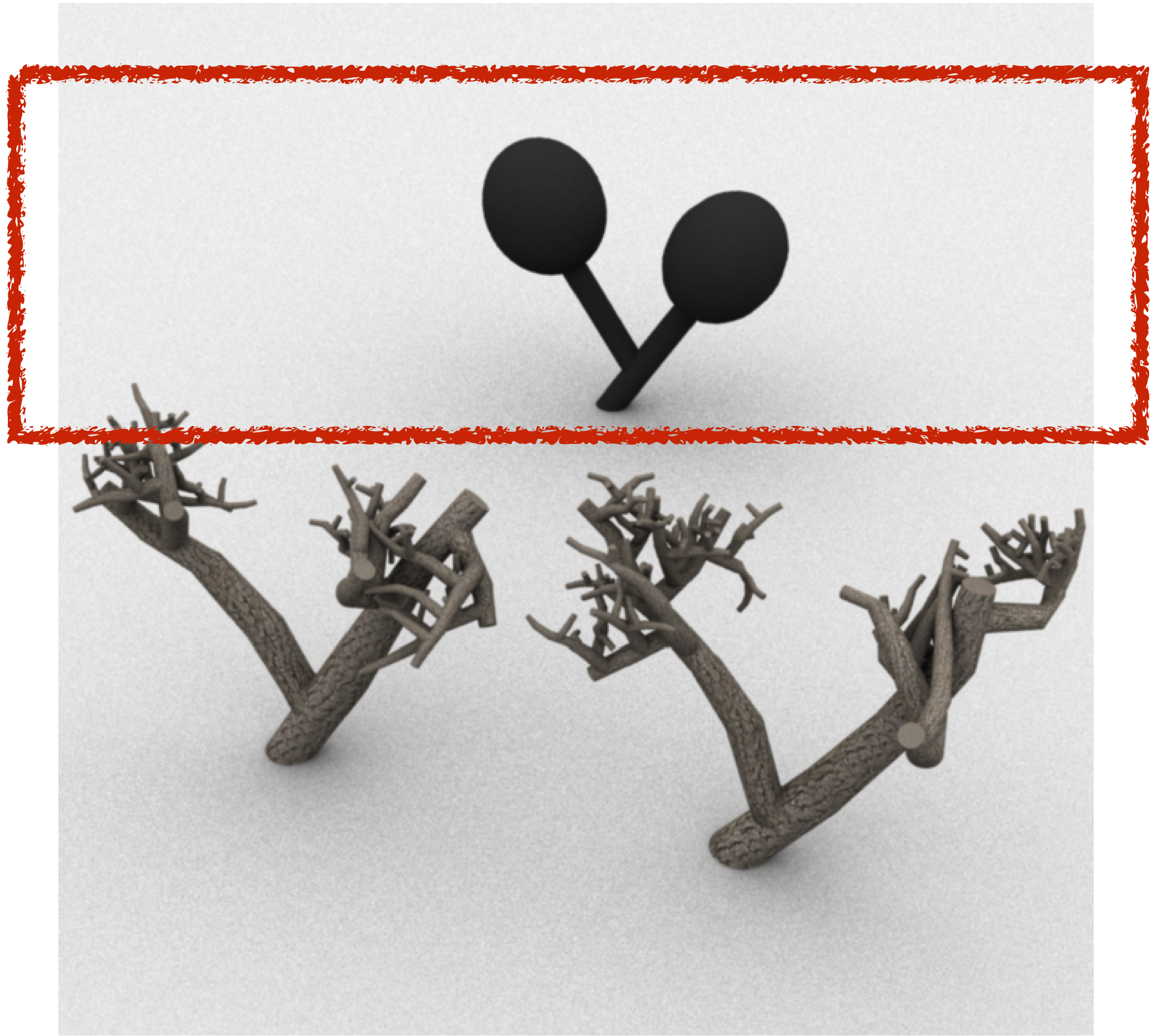
Procedural modelling



1. Sample a 3D object.

Ritchie, Mildenhall, Goodman, Hanrahan [SIGGRAPH'15]

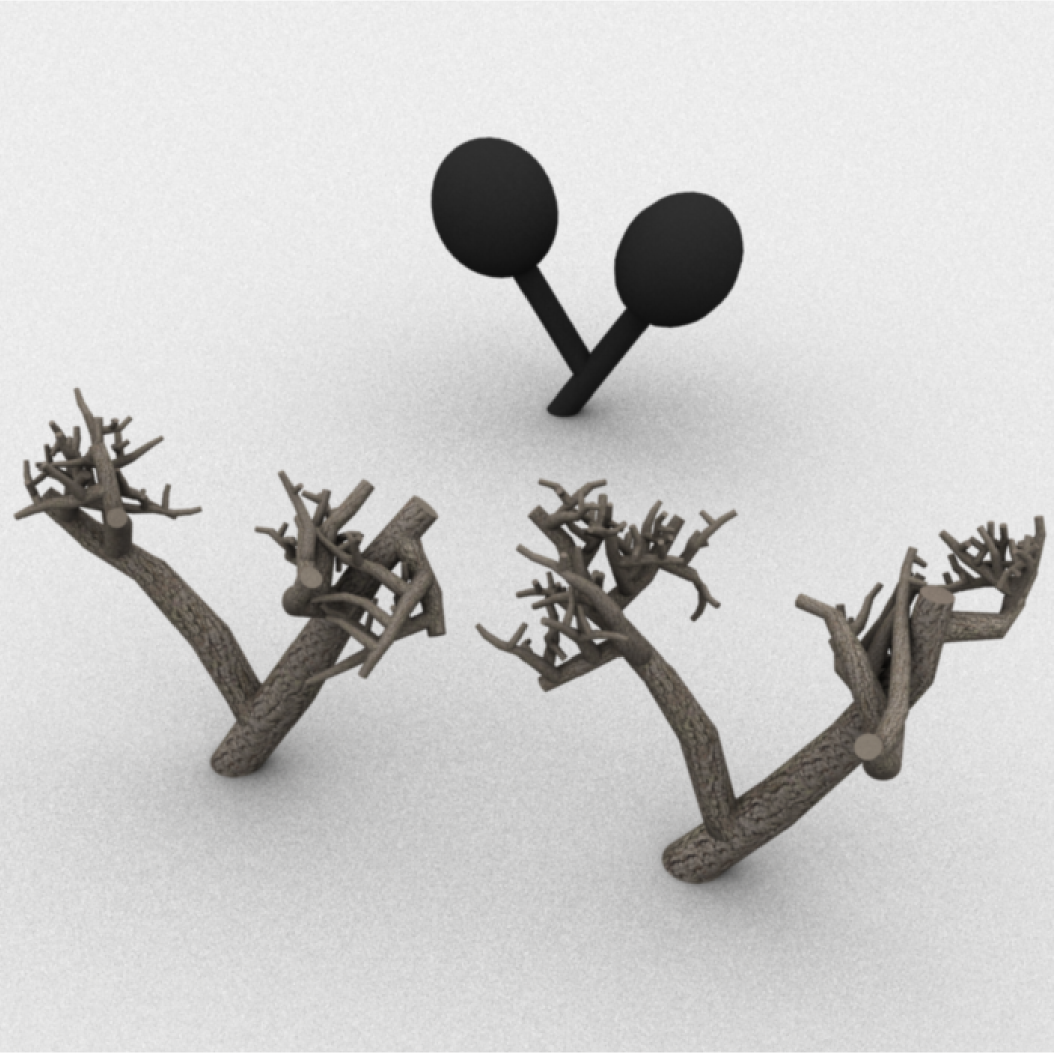
Procedural modelling



1. Sample a 3D object.
2. Score the object.

Ritchie, Mildenhall, Goodman, Hanrahan [SIGGRAPH'15]

Procedural modelling

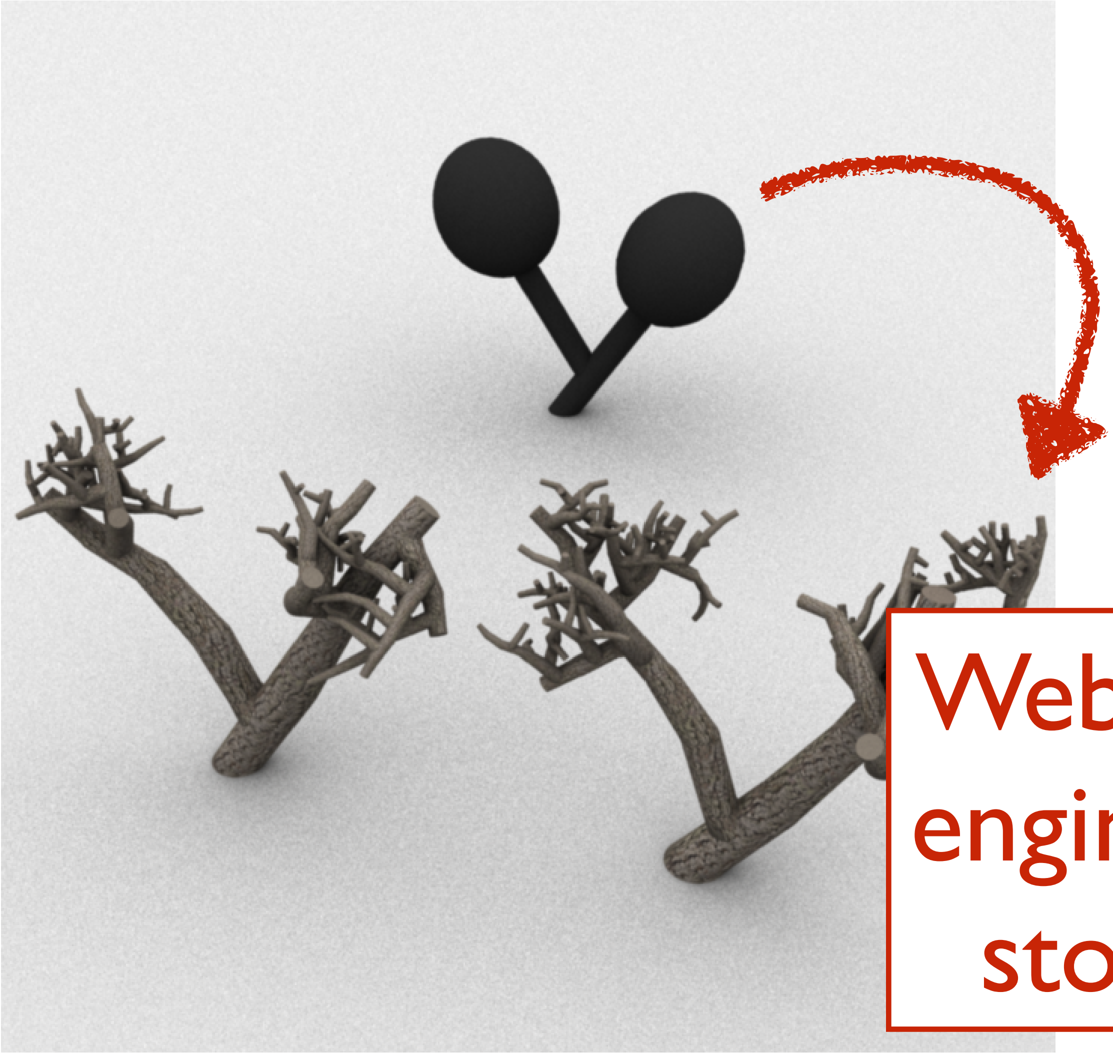


1. Sample a 3D object.
2. Score the object.

Used stochastic future.

Ritchie, Mildenhall, Goodman, Hanrahan [SIGGRAPH'15]

Procedural modelling



WebPPL's inference engine that exploits stochastic future

1. Sample a 3D object.
2. Score the object.

Used stochastic future.

Ritchie, Mildenhall, Goodman, Hanrahan [SIGGRAPH'15]

Why might QONFEST audience be interested?

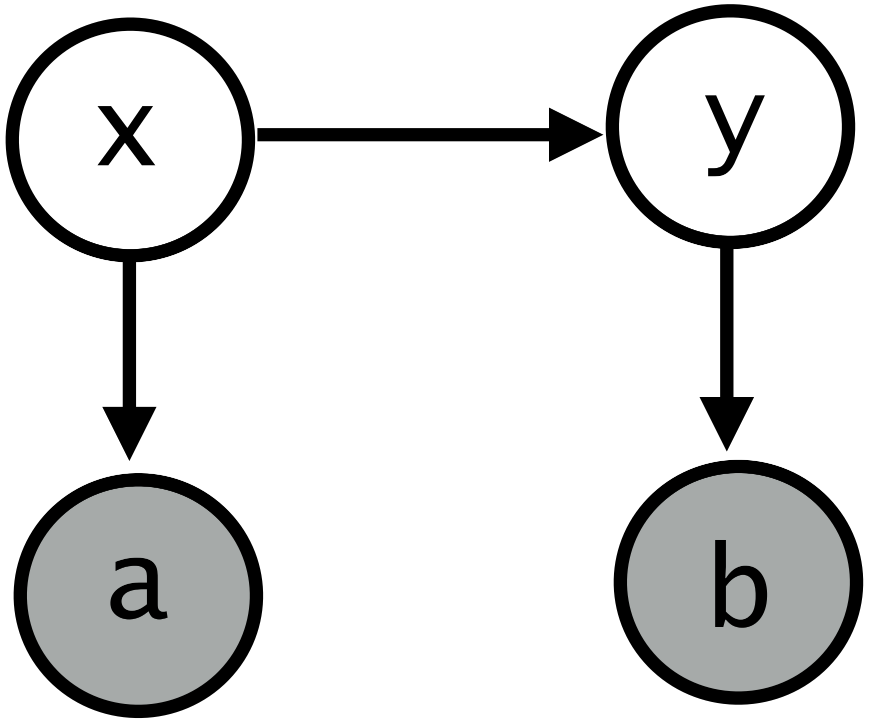
Reason 1:
Unusual use of concurrency.

An inference algo. for a prob. PL can be viewed as a non-standard approximate interpreter.

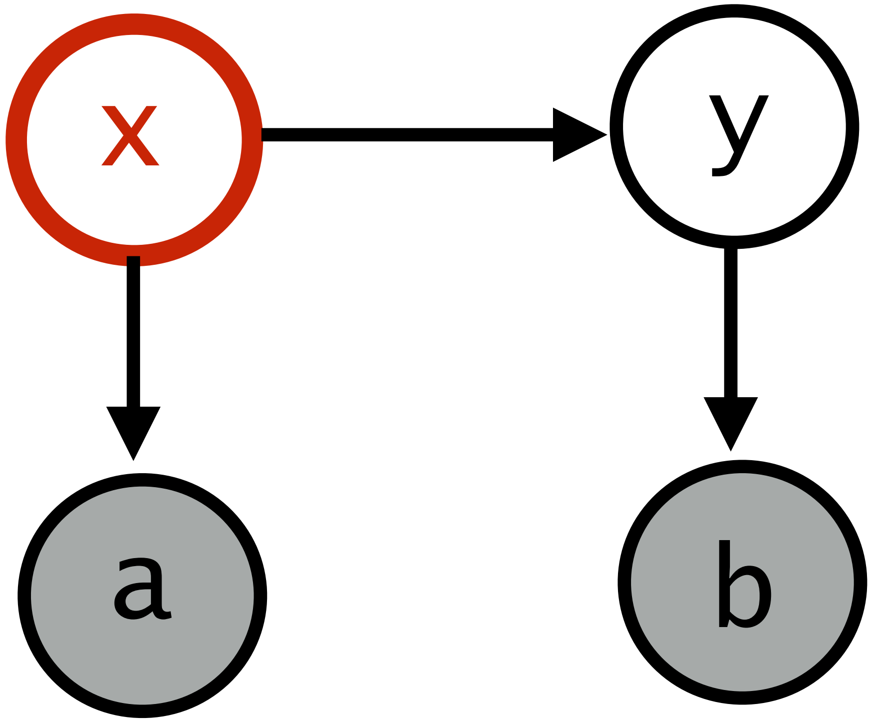
Sequential Monte Carlo (in short SMC) is a popular algo. with many variants.

SMC runs sequential probabilistic programs concurrently with added synchronisation.

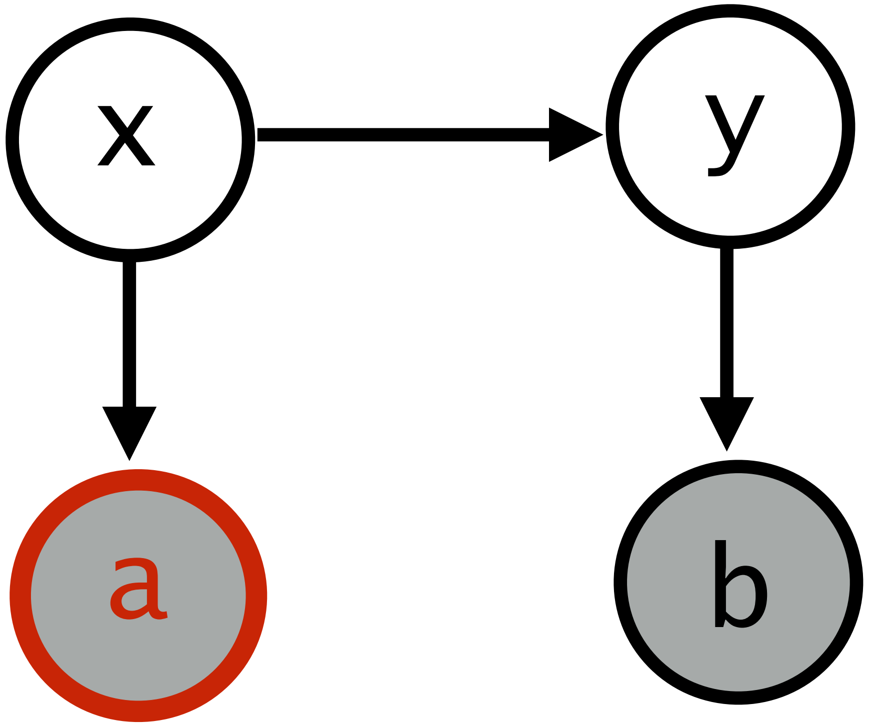
```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



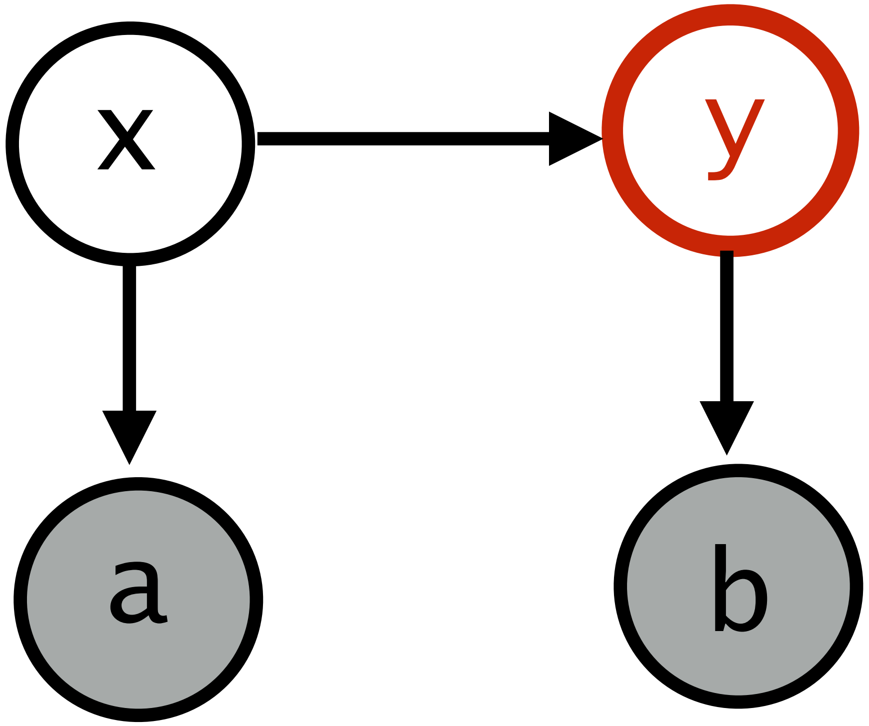
```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



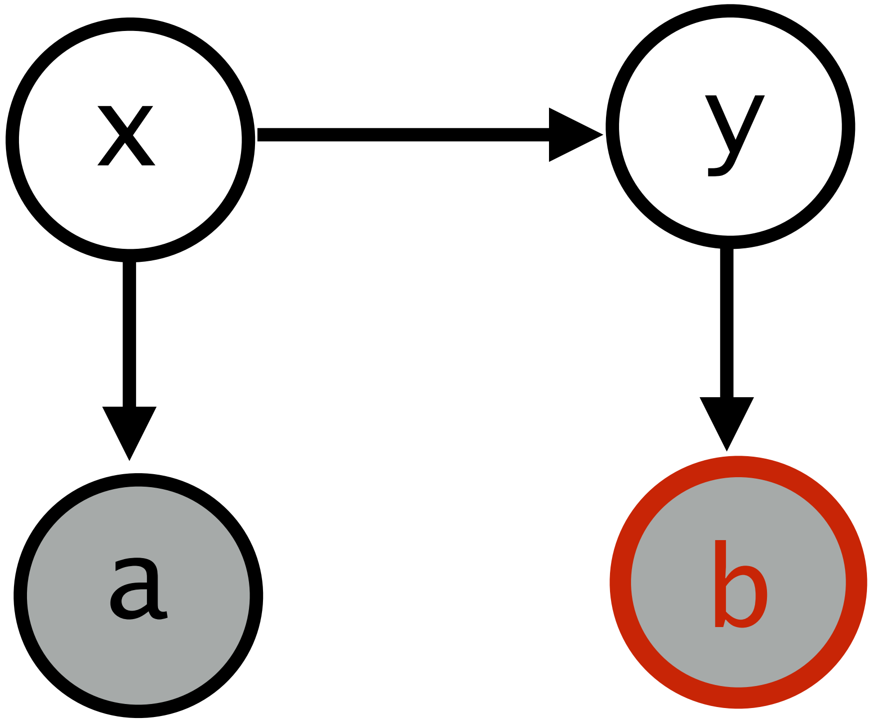
```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



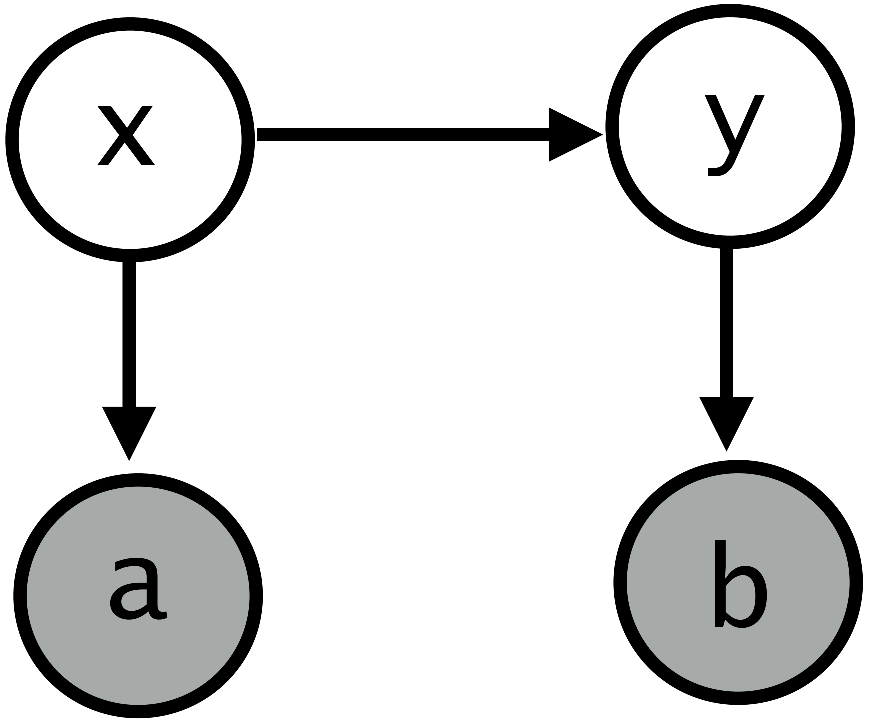
```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



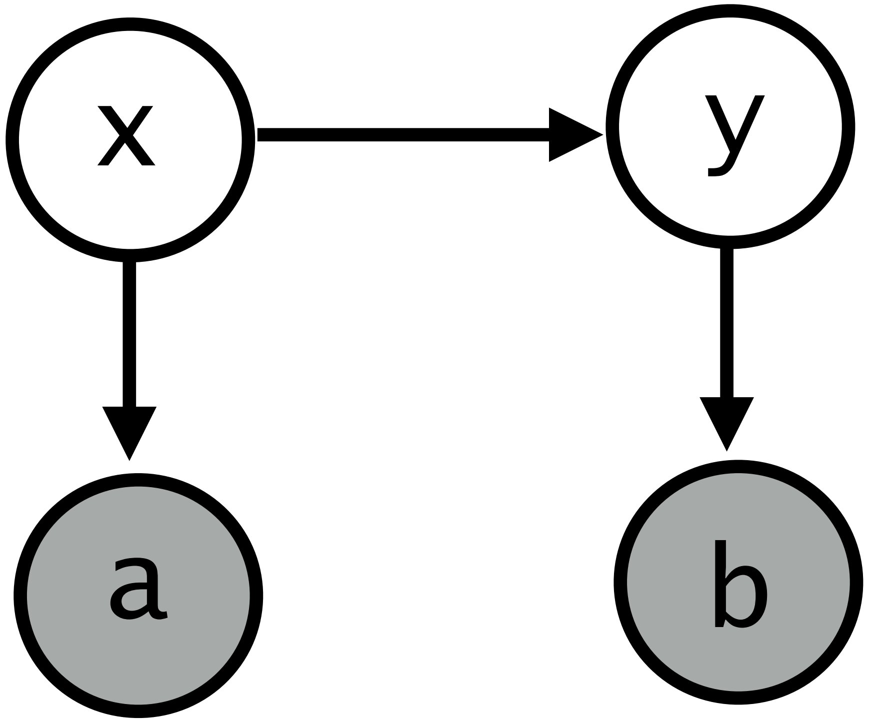
```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



Goal: Generate samples from posterior $p(x,y \mid a=2.1,b=1.8)$.

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Sequential Monte-Carlo algorithm

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

w:1

w:1

w:1

w:1

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

w:1

w:1

w:1

w:1

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

```
w:1
x:1.1
```

```
w:1
x:-0.3
```

```
w:1
x:2.3
```

```
w:1
x:0.2
```

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Sequential Monte-Carlo algorithm

```
w:1
x:1.1
```

```
w:1
x:-0.3
```

```
w:1
x:2.3
```

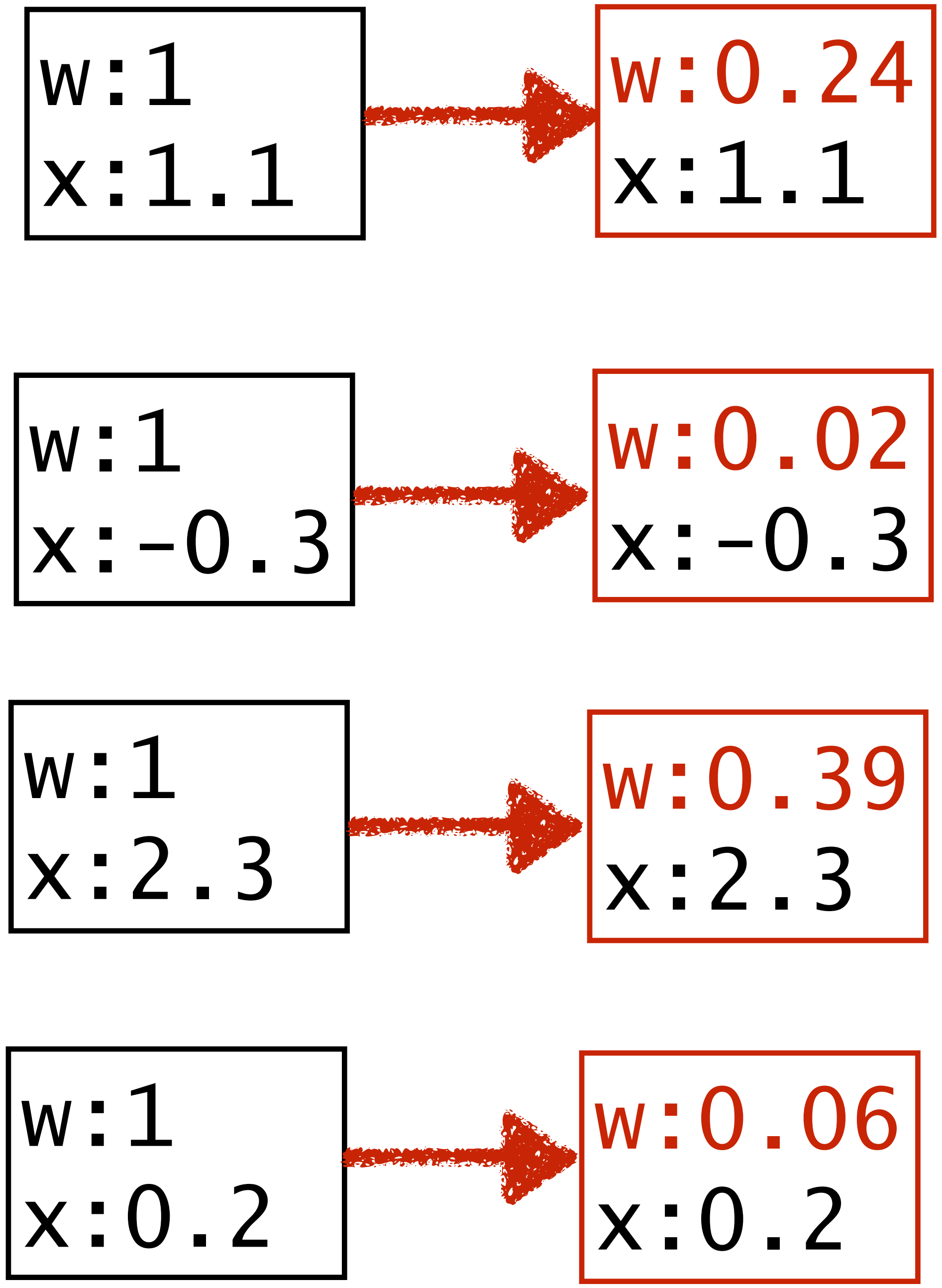
```
w:1
x:0.2
```

1. Update weights.

$w \longleftarrow p_{\texttt{normal}}(2.1; x, 1)$

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1 x:1.1 → w:0.24 x:1.1

w:1 x:-0.3 → w:0.02 x:-0.3

w:1 x:2.3 → w:0.39 x:2.3

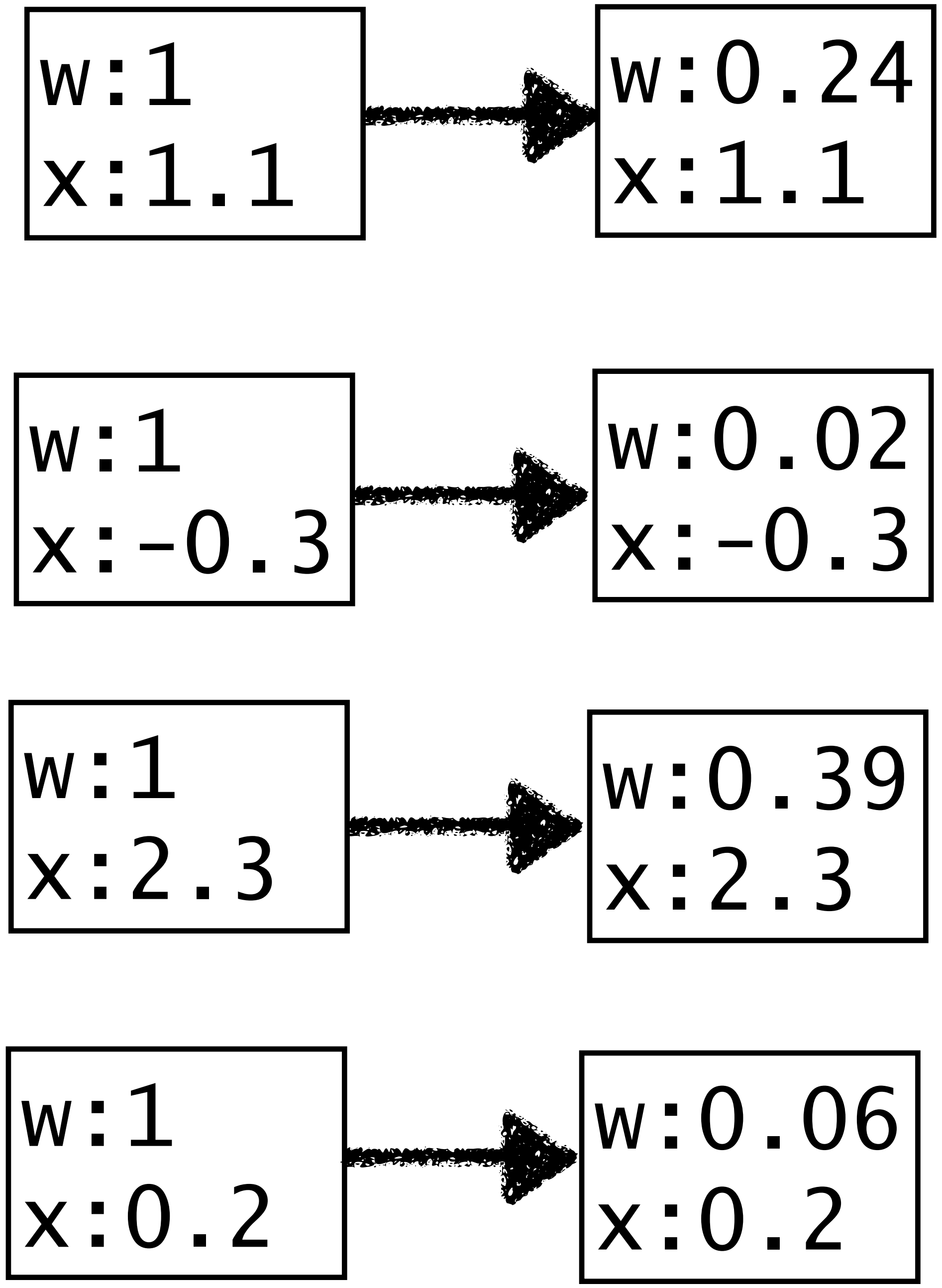
w:1 x:0.2 → w:0.06 x:0.2

1. Update weights.

$w \leftarrow p_{\texttt{normal}}(2.1;x,1)$

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1 x:1.1 → w:0.24 x:1.1

w:1 x:-0.3 → w:0.02 x:-0.3

w:1 x:2.3 → w:0.39 x:2.3

w:1 x:0.2 → w:0.06 x:0.2

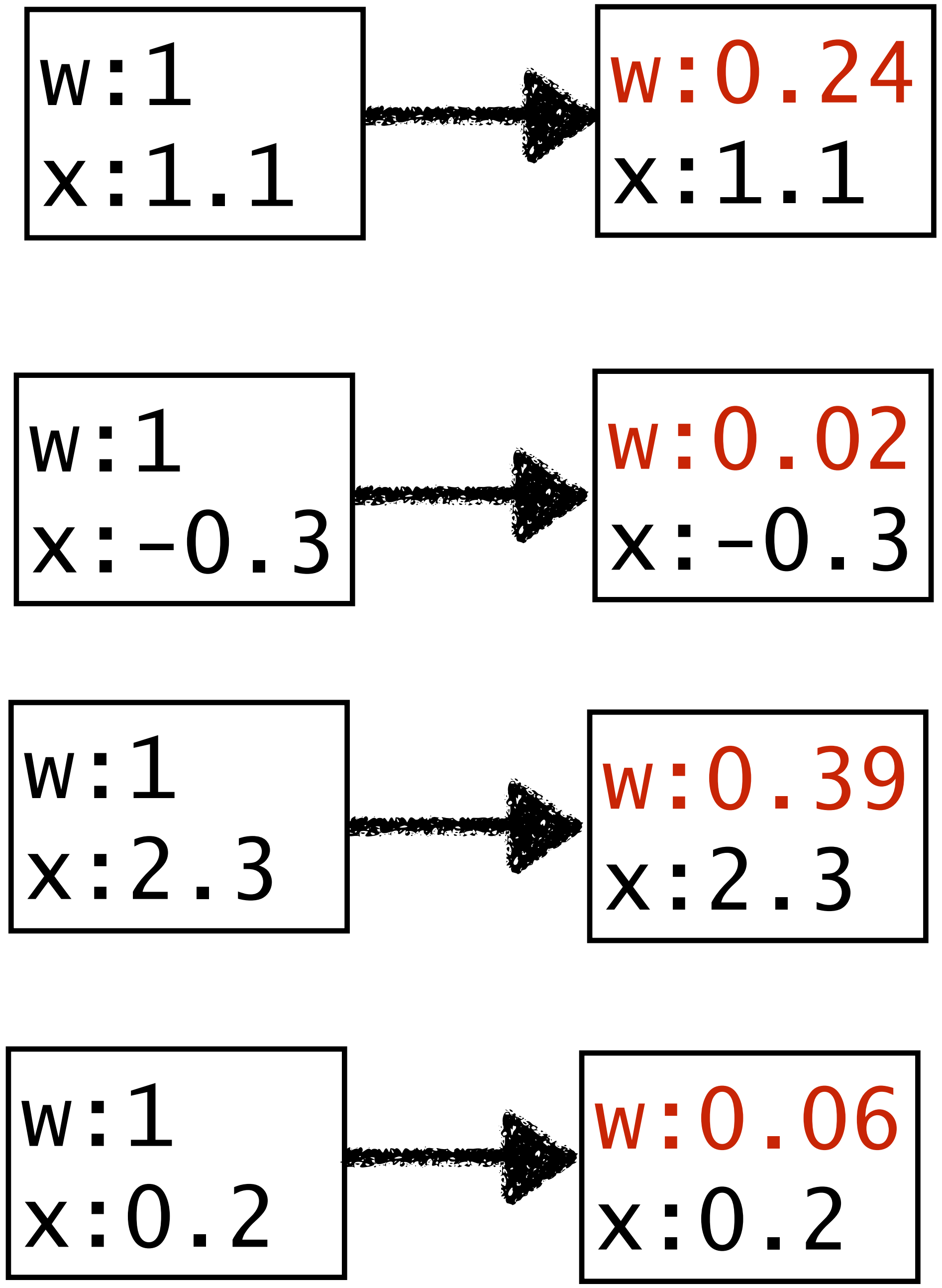
1. Update weights.

$w \leftarrow p_{\texttt{normal}}(2.1;x,1)$

2. Resample and reset weights.

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Sequential Monte-Carlo algorithm



w:1
x:1.1 → w:0.24
x:1.1

w:1
x:-0.3 → w:0.02
x:-0.3

w:1
x:2.3 → w:0.39
x:2.3

w:1
x:0.2 → w:0.06
x:0.2

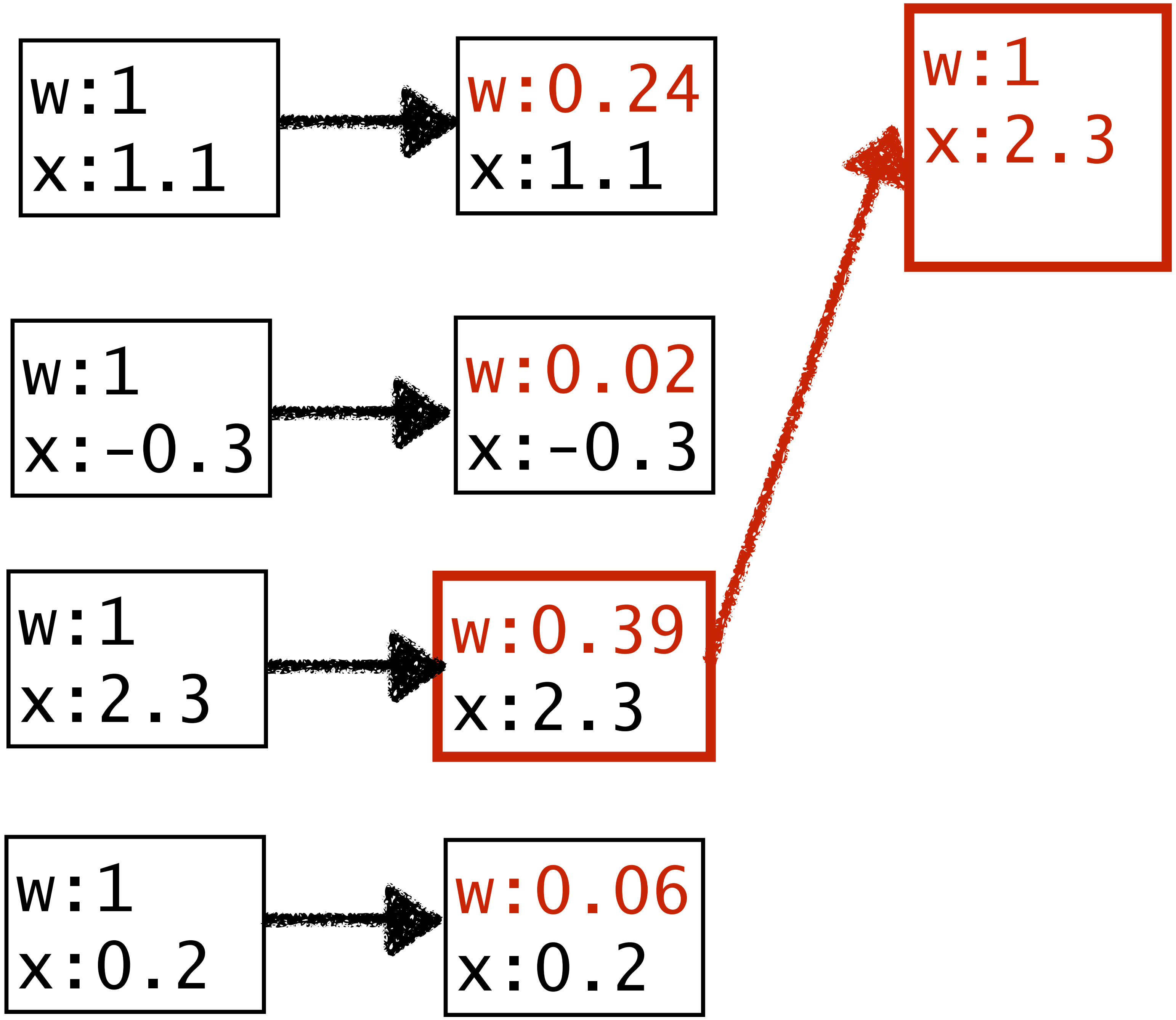
1. Update weights.

$w \longleftarrow p_{\mathtt{normal}}(2.1;x,1)$

2. Resample and reset weights.

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:1.1

w:0.24
x:1.1

w:1
x:2.3

w:1
x:-0.3

w:0.02
x:-0.3

w:1
x:2.3

w:0.39
x:2.3

w:1
x:0.2

w:0.06
x:0.2

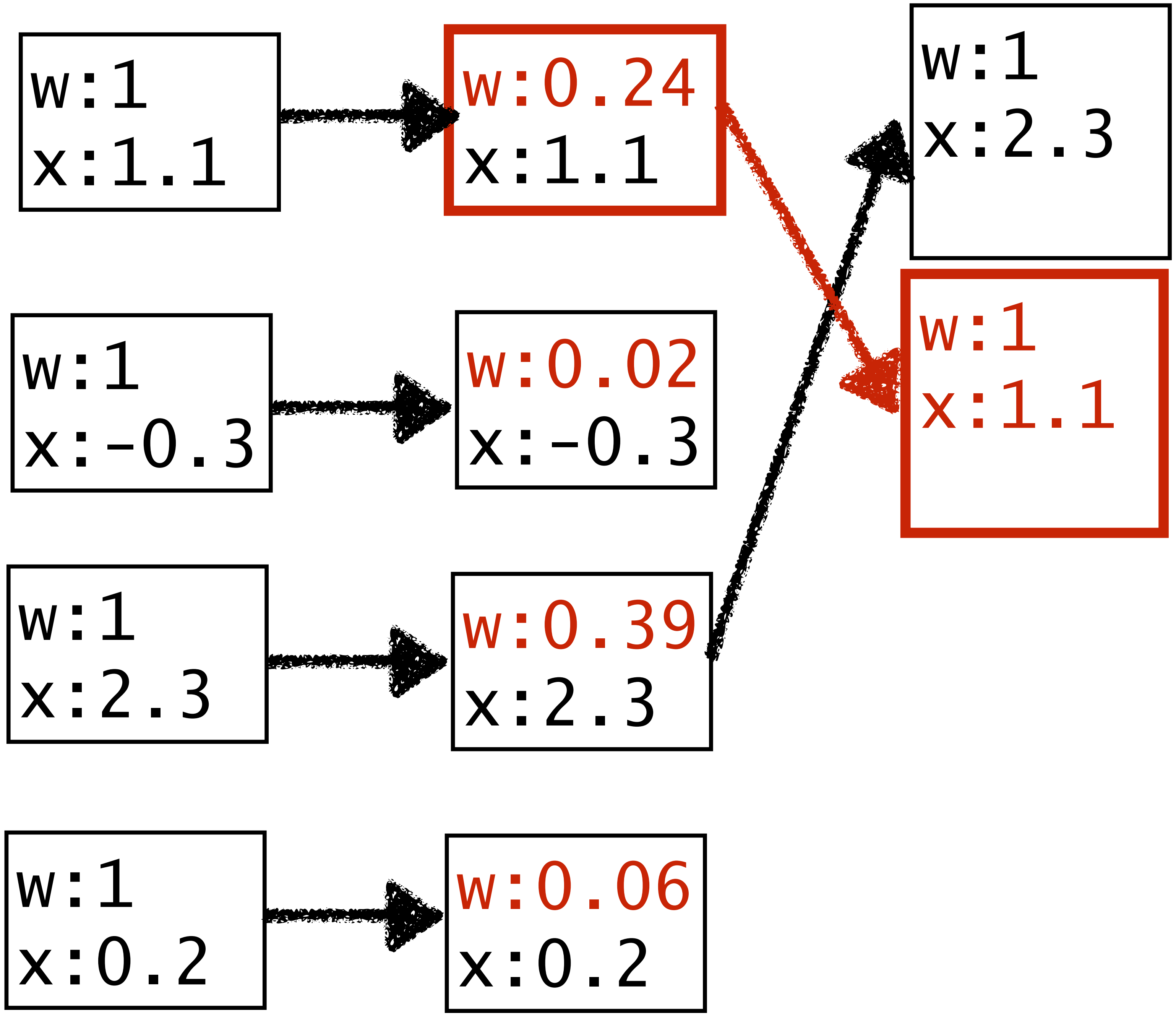
1. Update weights.

$w \leftarrow p_{\mathtt{normal}}(2.1;x,1)$

2. Resample and reset weights.

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Sequential Monte-Carlo algorithm



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3

w:1
x:1.1

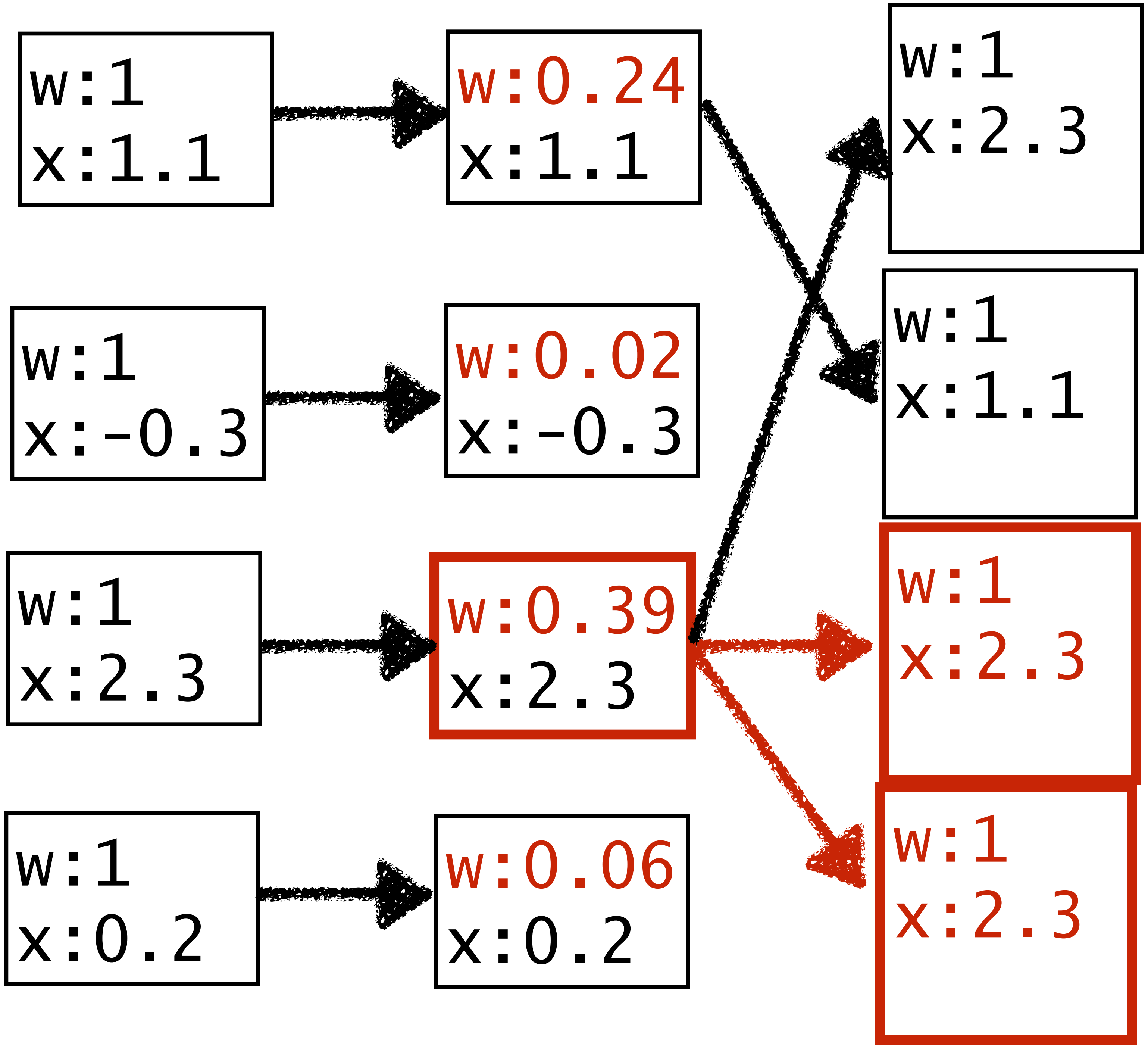
1. Update weights.

`w ⟵ p`$_{\texttt{normal}}$`(2.1;x,1)`

2. Resample and reset weights.

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3

w:1
x:1.1

w:1
x:2.3

w:1
x:2.3

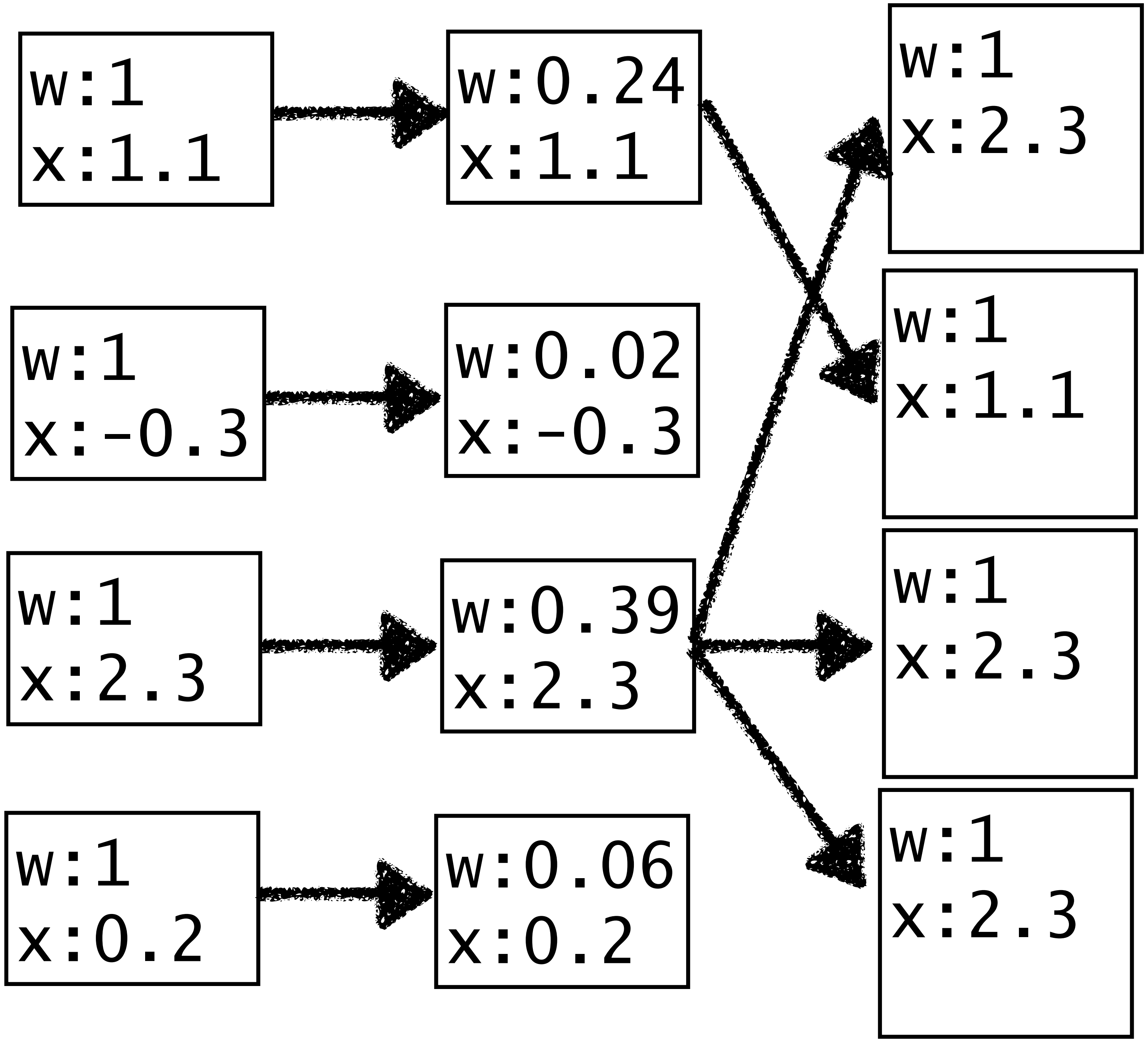
1. Update weights.

$w \longleftarrow p_{\texttt{normal}}(2.1; x, 1)$

2. Resample and reset weights.

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3

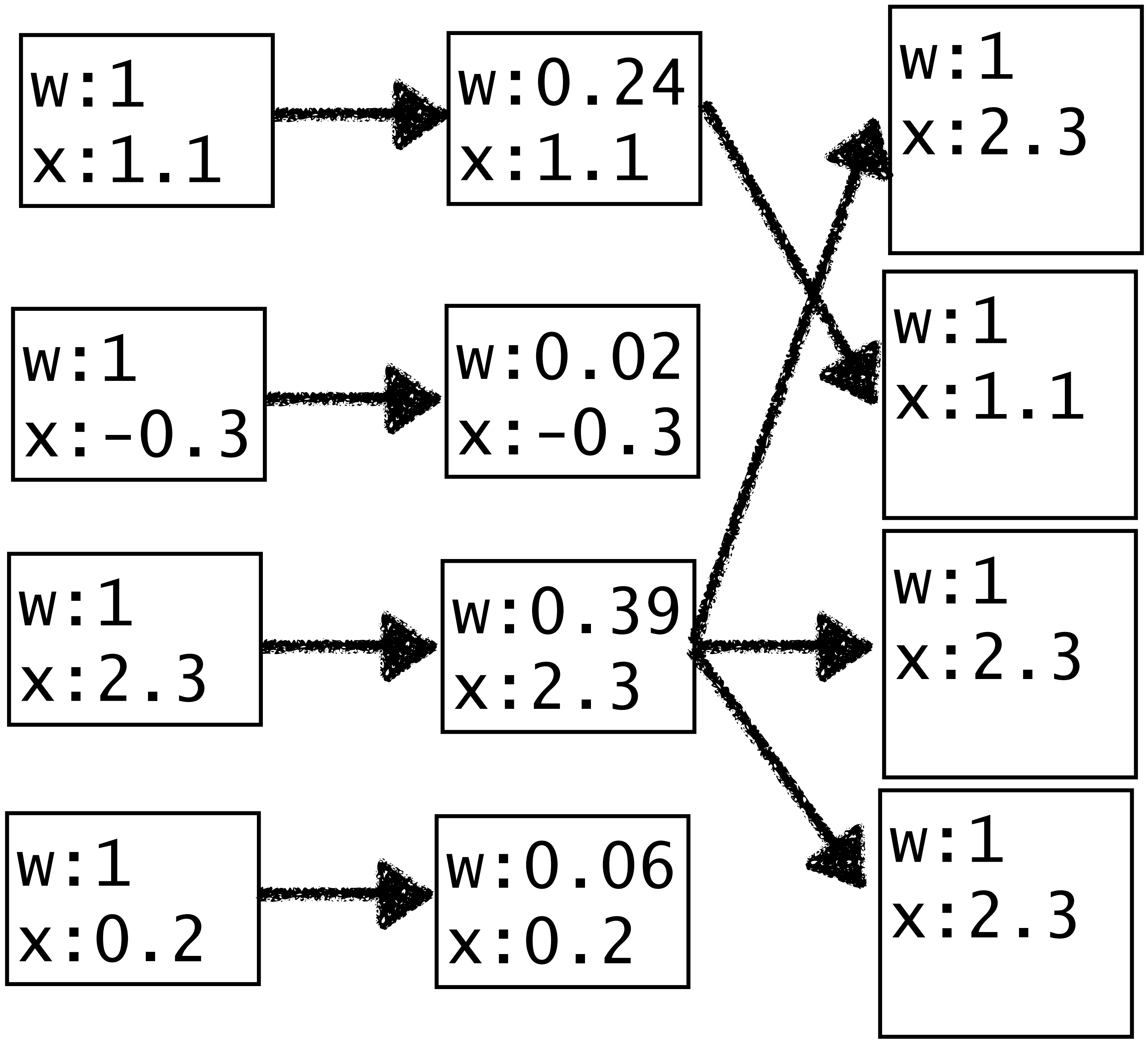
w:1
x:1.1

w:1
x:2.3

w:1
x:2.3

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Sequential Monte-Carlo algorithm



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3

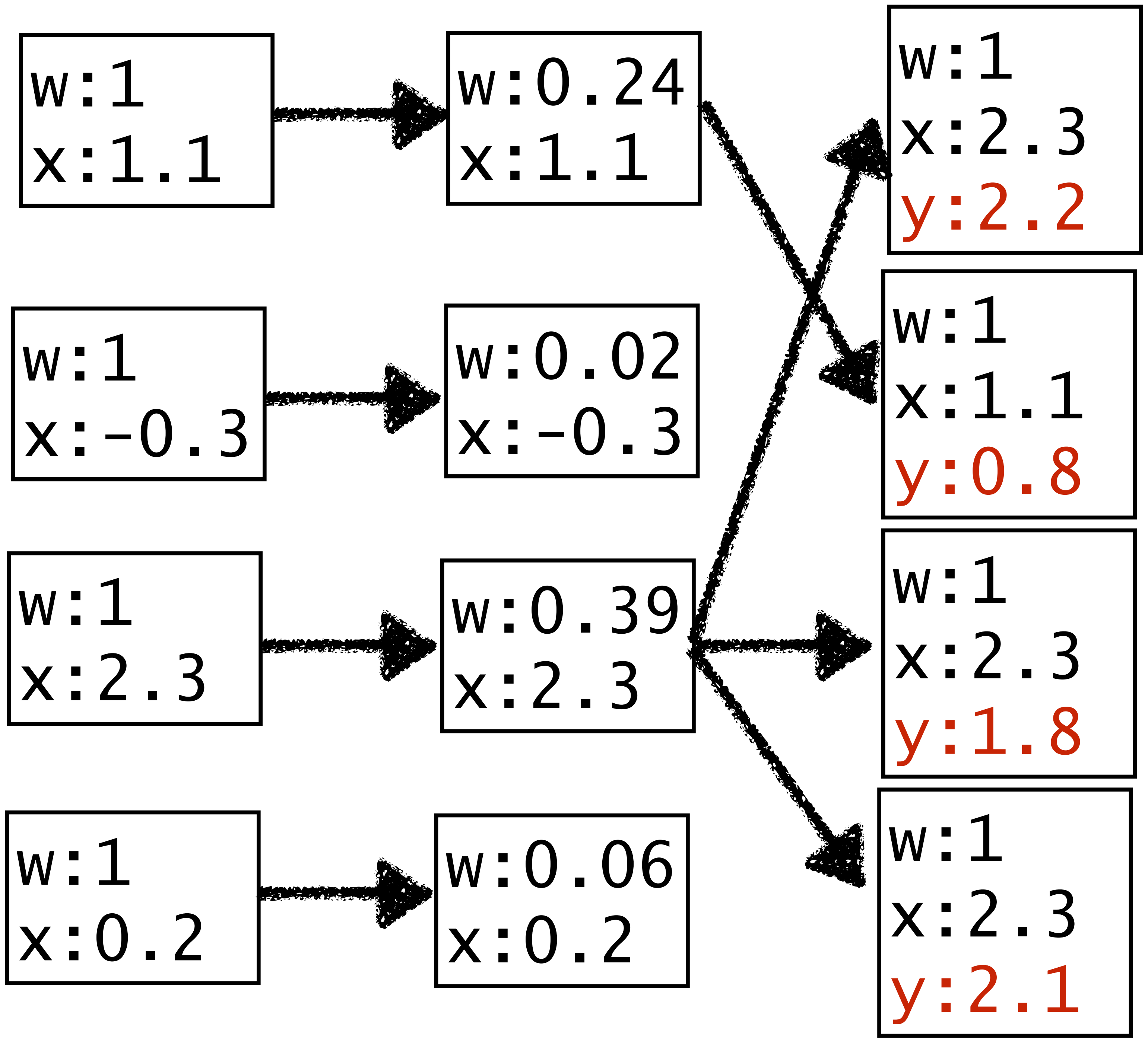
w:1
x:1.1

w:1
x:2.3

w:1
x:2.3

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3
y:2.2

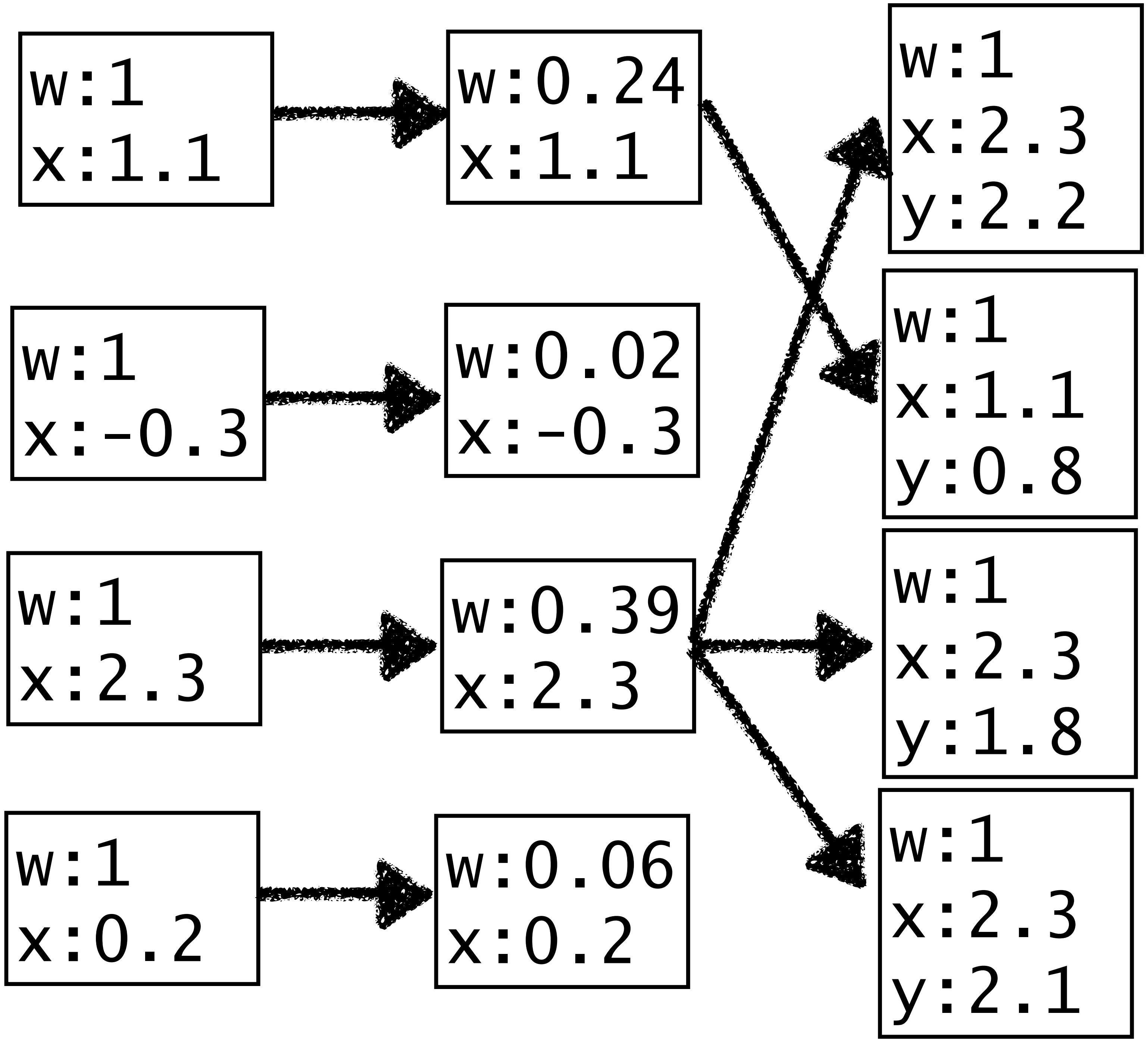
w:1
x:1.1
y:0.8

w:1
x:2.3
y:1.8

w:1
x:2.3
y:2.1

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3
y:2.2

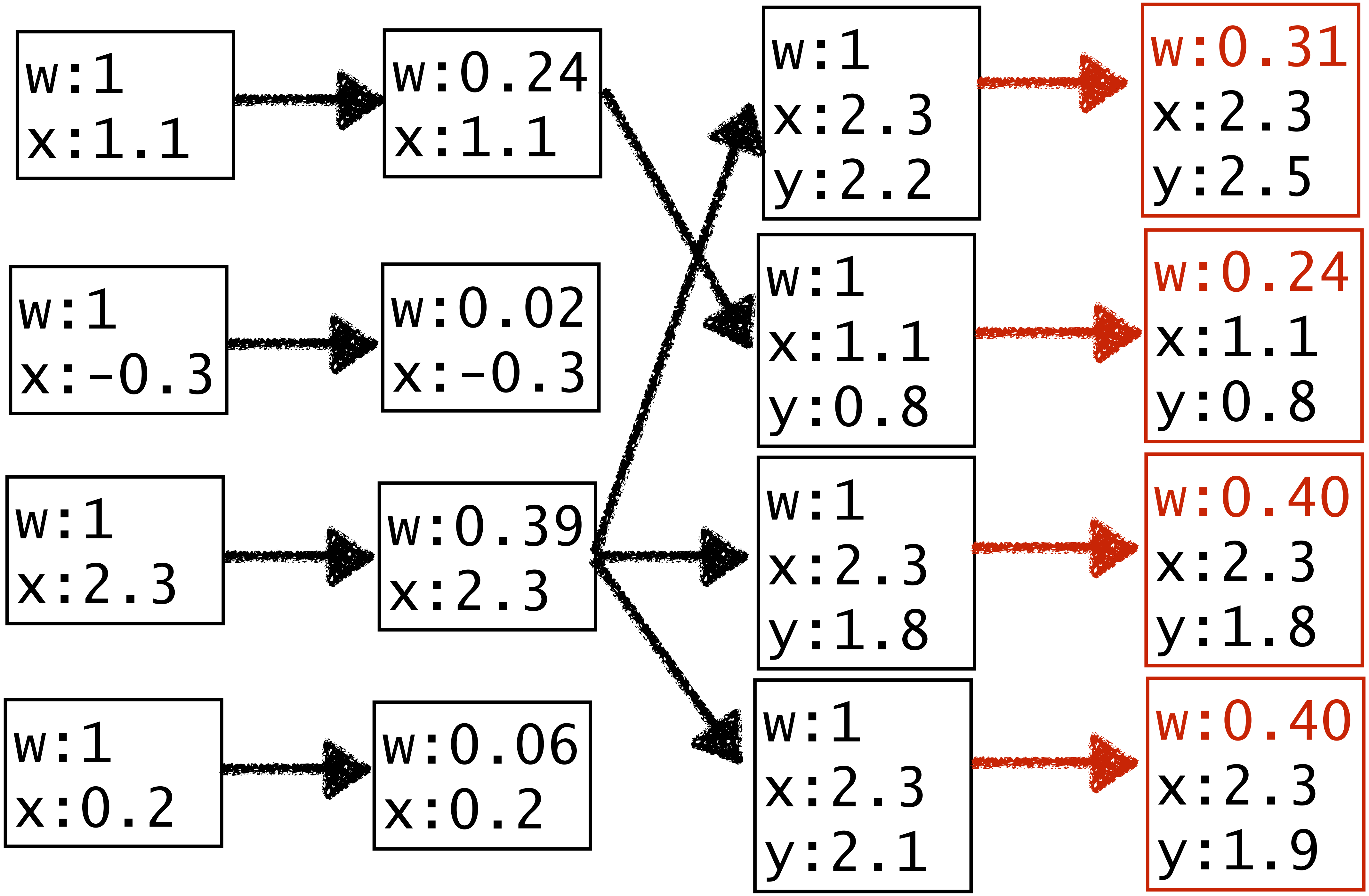
w:1
x:1.1
y:0.8

w:1
x:2.3
y:1.8

w:1
x:2.3
y:2.1

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3
y:2.2

w:1
x:1.1
y:0.8

w:1
x:2.3
y:1.8

w:1
x:2.3
y:2.1

w:0.31
x:2.3
y:2.5

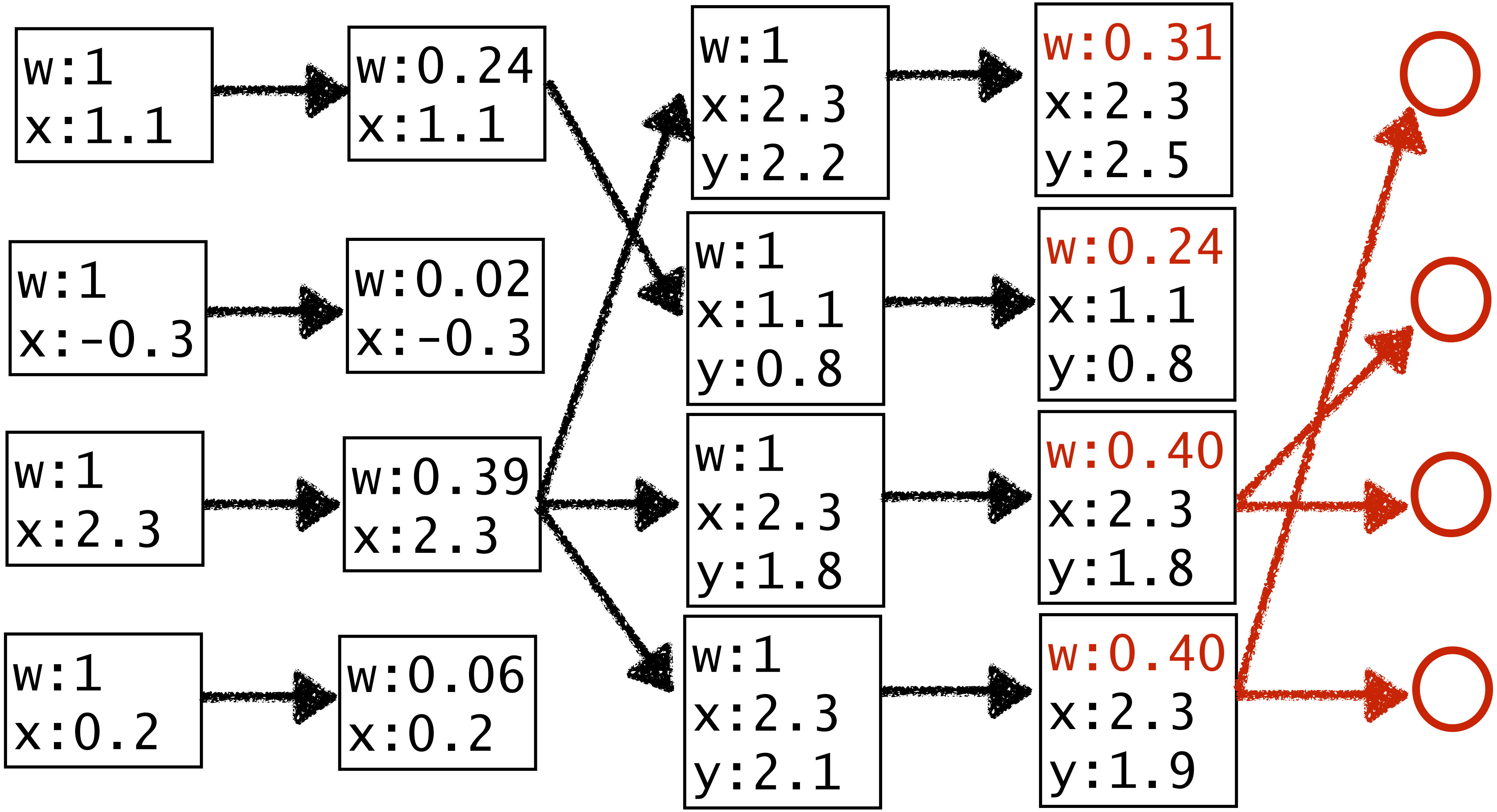
w:0.24
x:1.1
y:0.8

w:0.40
x:2.3
y:1.8

w:0.40
x:2.3
y:1.9

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3
y:2.2

w:1
x:1.1
y:0.8

w:1
x:2.3
y:1.8

w:1
x:2.3
y:2.1

w:0.31
x:2.3
y:2.5

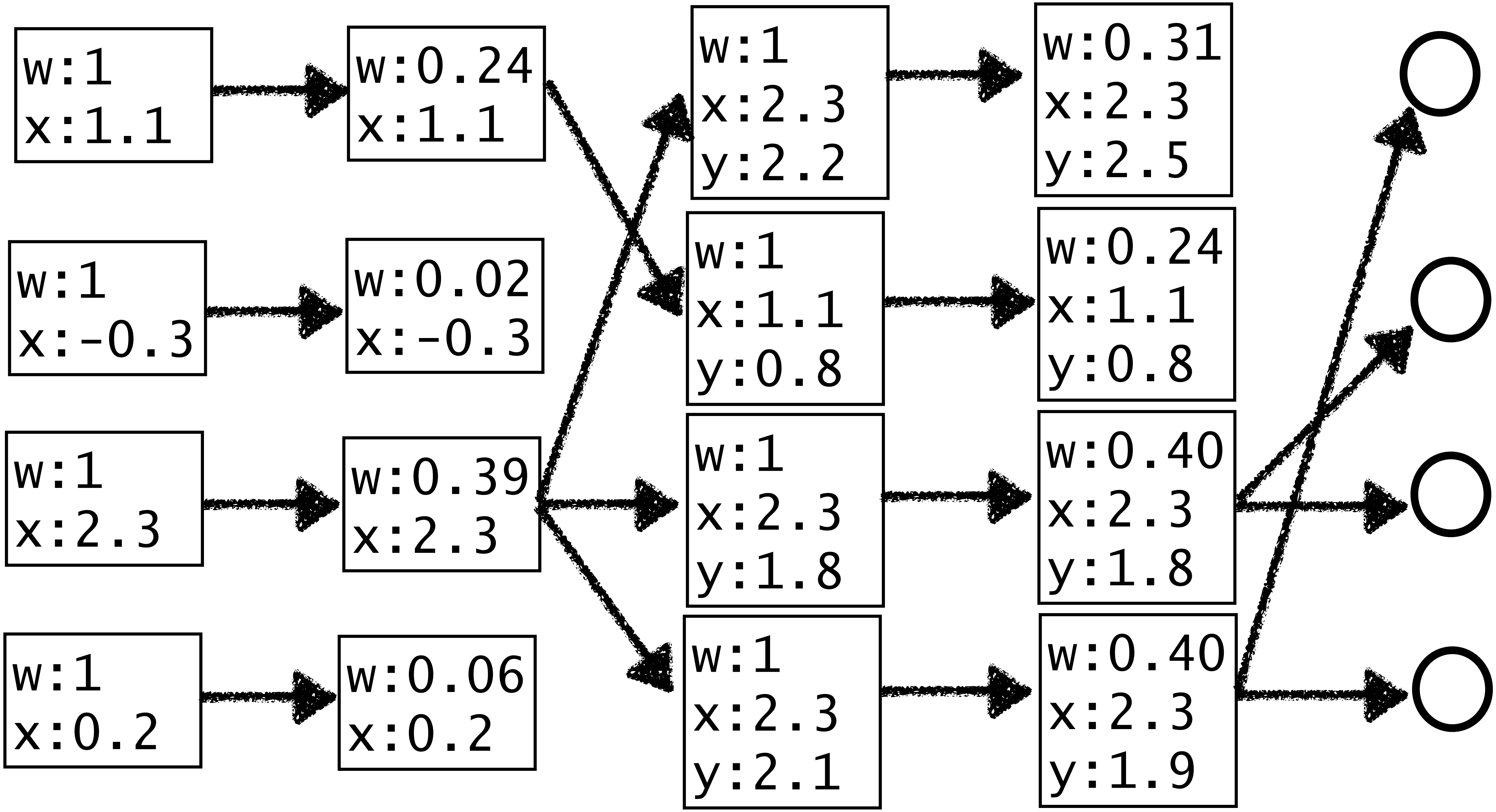
w:0.24
x:1.1
y:0.8

w:0.40
x:2.3
y:1.8

w:0.40
x:2.3
y:1.9

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Sequential Monte-Carlo algorithm



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3
y:2.2

w:1
x:1.1
y:0.8

w:1
x:2.3
y:1.8

w:1
x:2.3
y:2.1

w:0.31
x:2.3
y:2.5

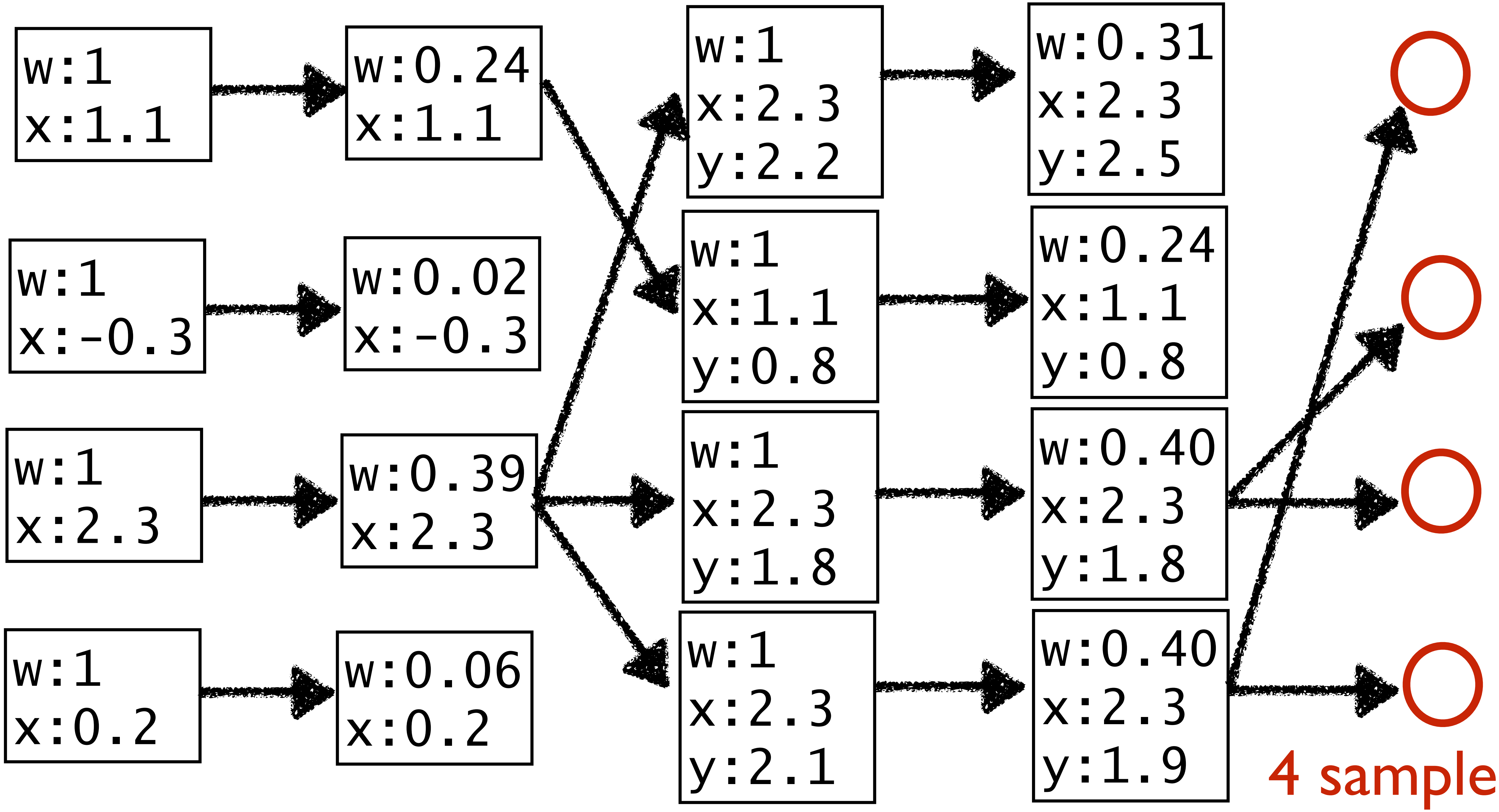
w:0.24
x:1.1
y:0.8

w:0.40
x:2.3
y:1.8

w:0.40
x:2.3
y:1.9

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3
y:2.2

w:1
x:1.1
y:0.8

w:1
x:2.3
y:1.8

w:1
x:2.3
y:2.1

w:0.31
x:2.3
y:2.5

w:0.24
x:1.1
y:0.8

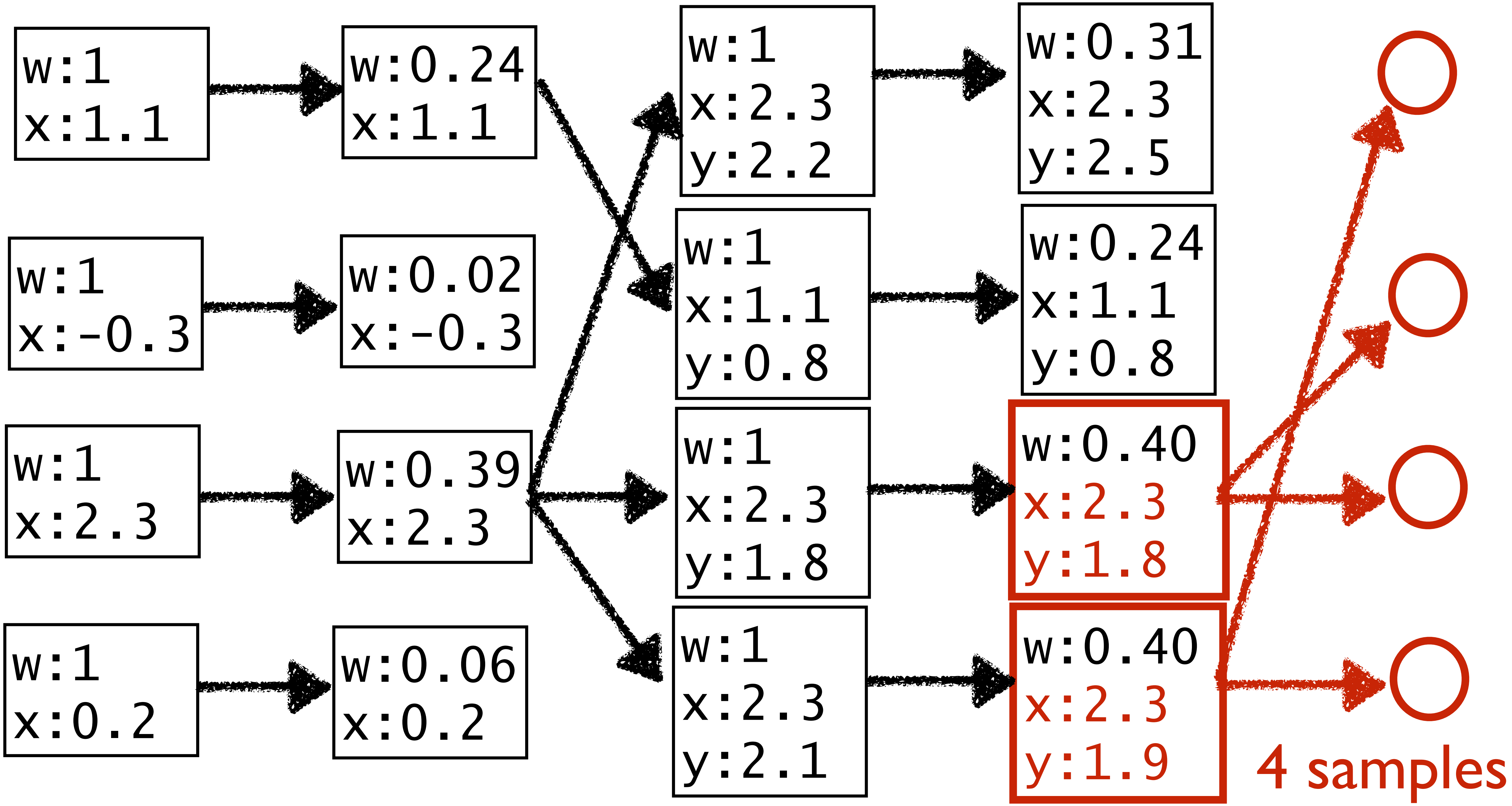
w:0.40
x:2.3
y:1.8

w:0.40
x:2.3
y:1.9

4 samples

Sequential Monte-Carlo algorithm

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3
y:2.2

w:1
x:1.1
y:0.8

w:1
x:2.3
y:1.8

w:1
x:2.3
y:2.1

w:0.31
x:2.3
y:2.5

w:0.24
x:1.1
y:0.8

w:0.40
x:2.3
y:1.8

w:0.40
x:2.3
y:1.9

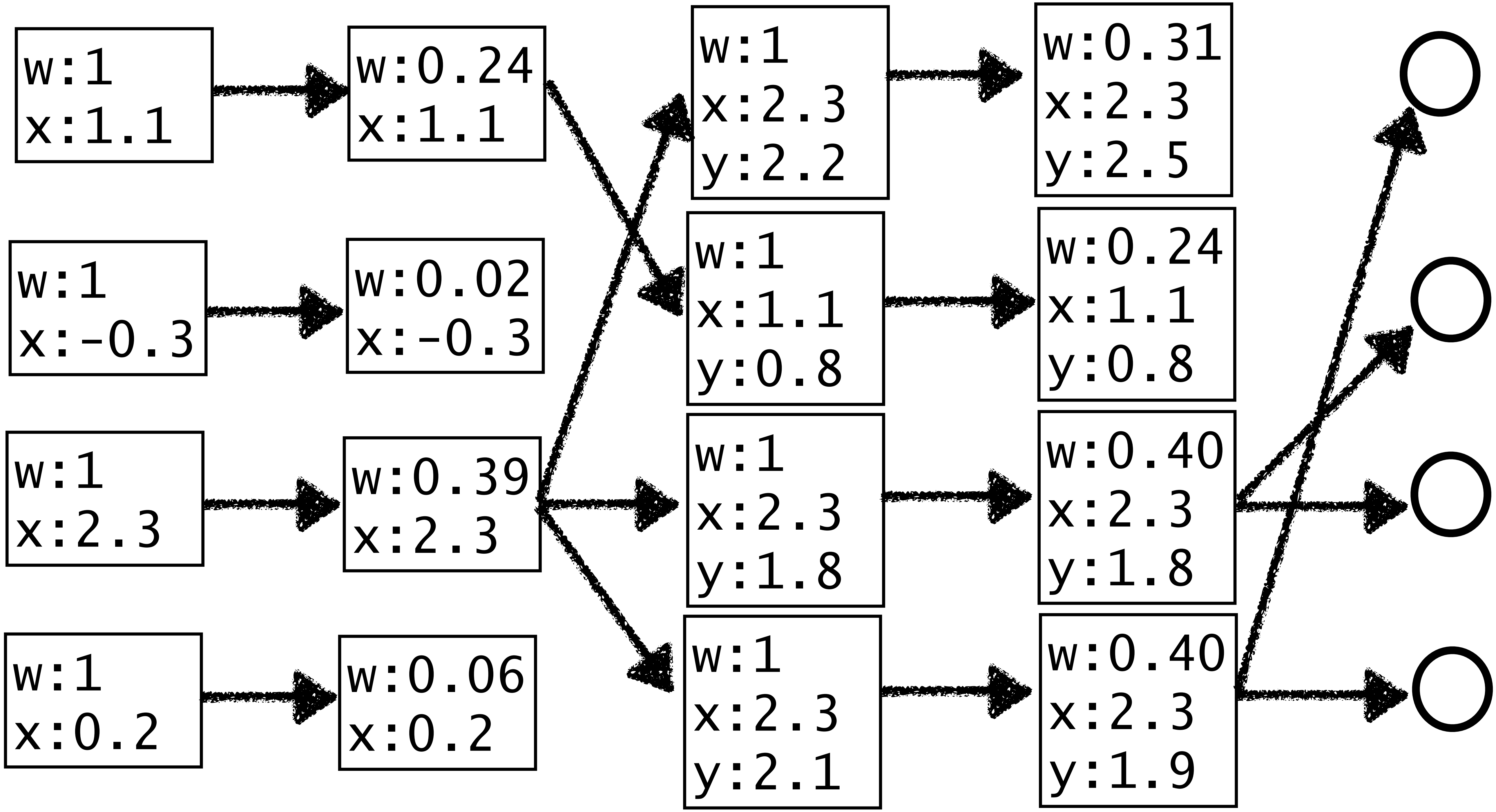
4 samples

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Particle Gibbs

Particle Gibbs

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3
y:2.2

w:1
x:1.1
y:0.8

w:1
x:2.3
y:1.8

w:1
x:2.3
y:2.1

w:0.31
x:2.3
y:2.5

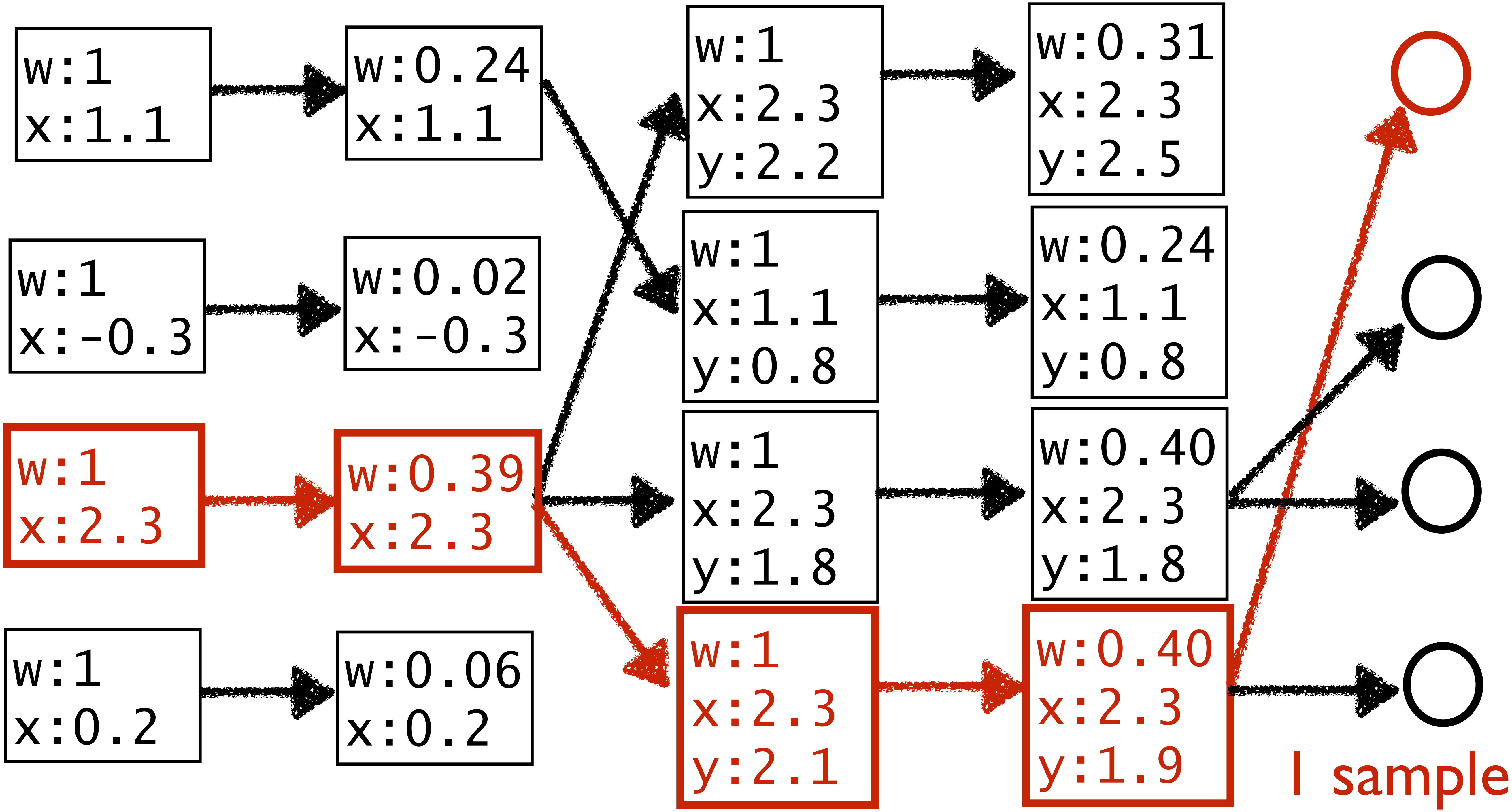
w:0.24
x:1.1
y:0.8

w:0.40
x:2.3
y:1.8

w:0.40
x:2.3
y:1.9

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Particle Gibbs



w:1
x:1.1

w:1
x:-0.3

w:1
x:2.3

w:1
x:0.2

w:0.24
x:1.1

w:0.02
x:-0.3

w:0.39
x:2.3

w:0.06
x:0.2

w:1
x:2.3
y:2.2

w:1
x:1.1
y:0.8

w:1
x:2.3
y:1.8

w:1
x:2.3
y:2.1

w:0.31
x:2.3
y:2.5

w:0.24
x:1.1
y:0.8

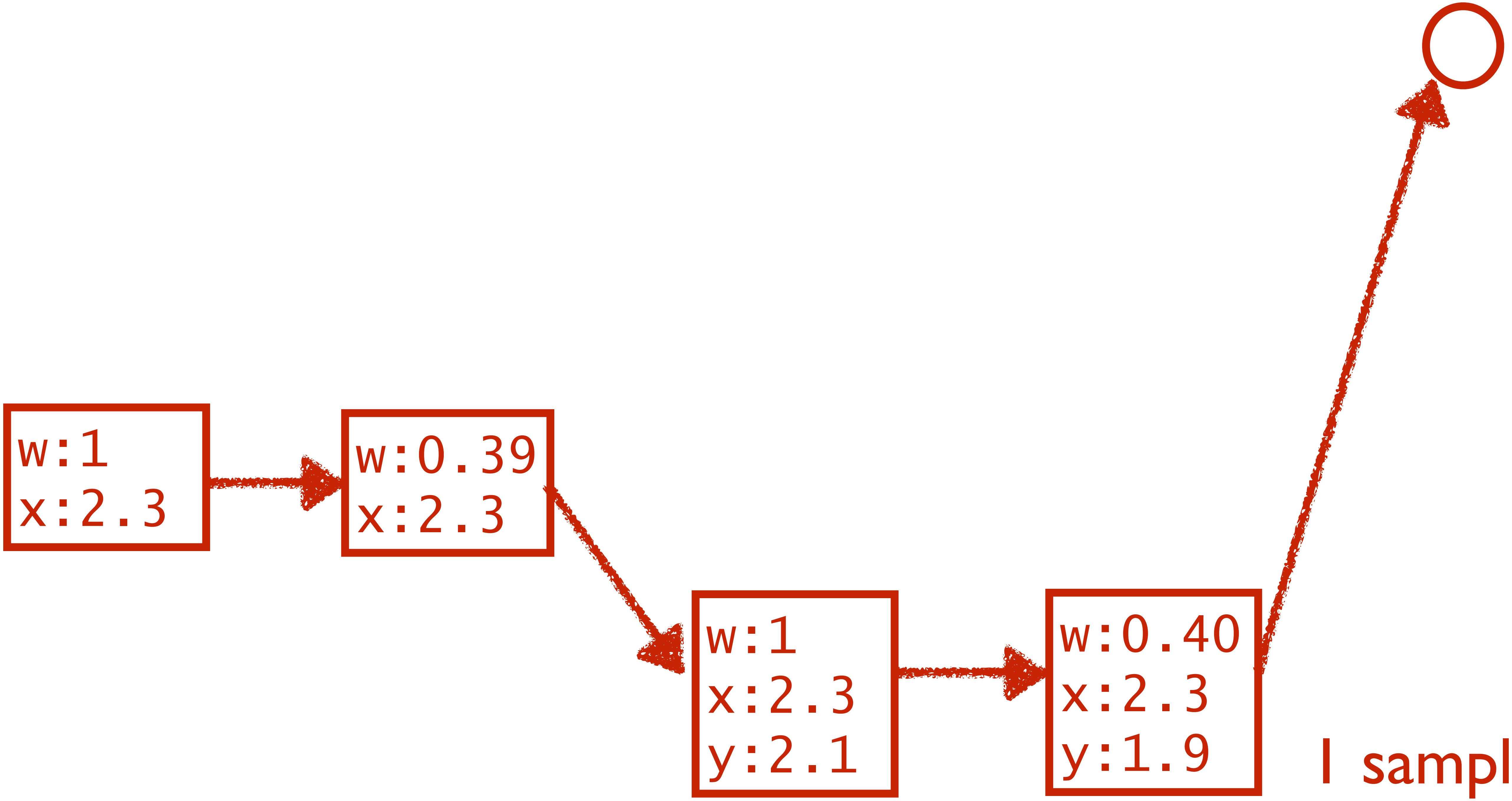
w:0.40
x:2.3
y:1.8

w:0.40
x:2.3
y:1.9

1 sample

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Particle Gibbs



w:1
x:2.3

w:0.39
x:2.3

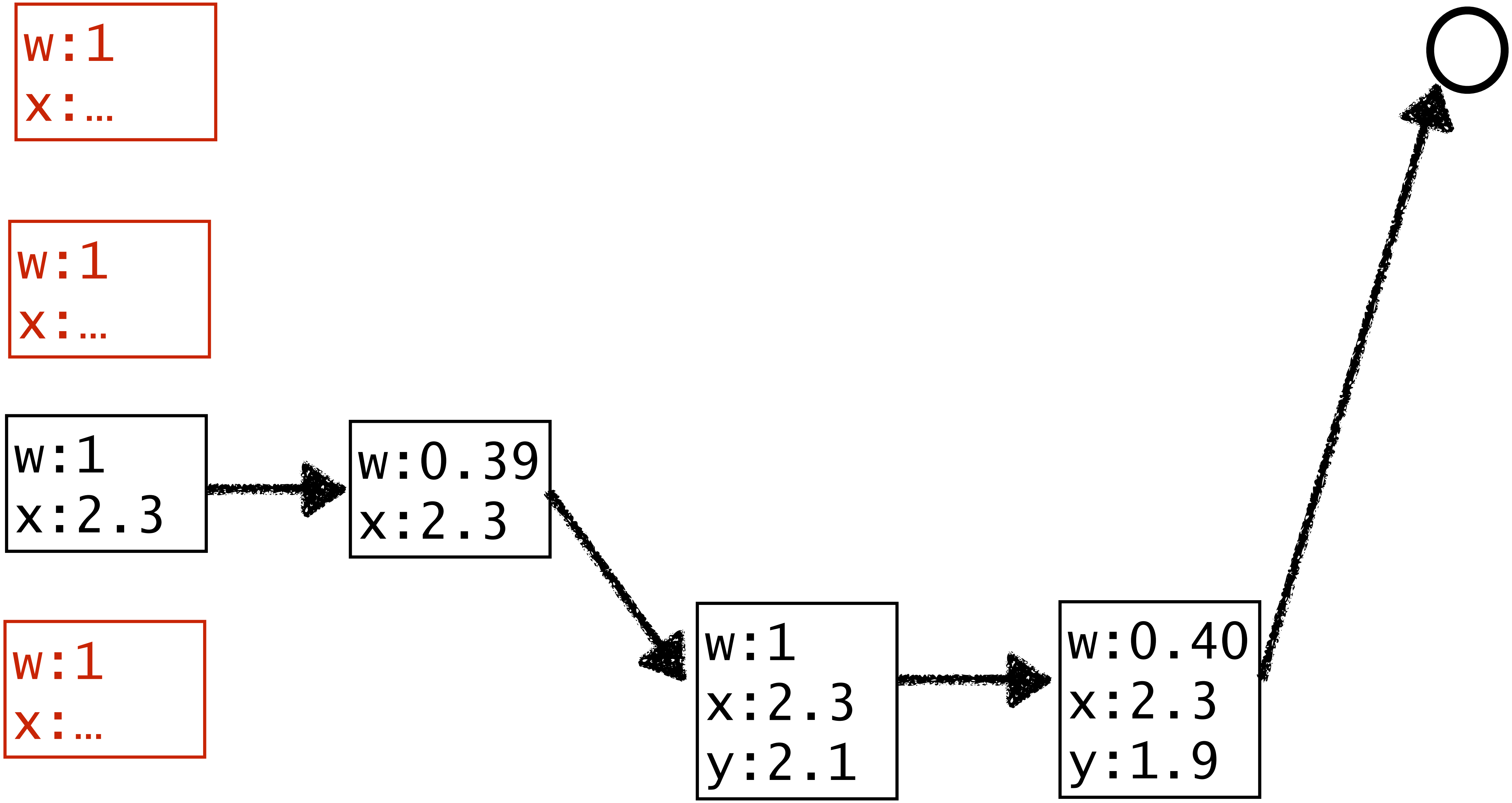
w:1
x:2.3
y:2.1

w:0.40
x:2.3
y:1.9

1 sample

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Particle Gibbs



w:1
x:…

w:1
x:…

w:1
x:2.3

w:0.39
x:2.3

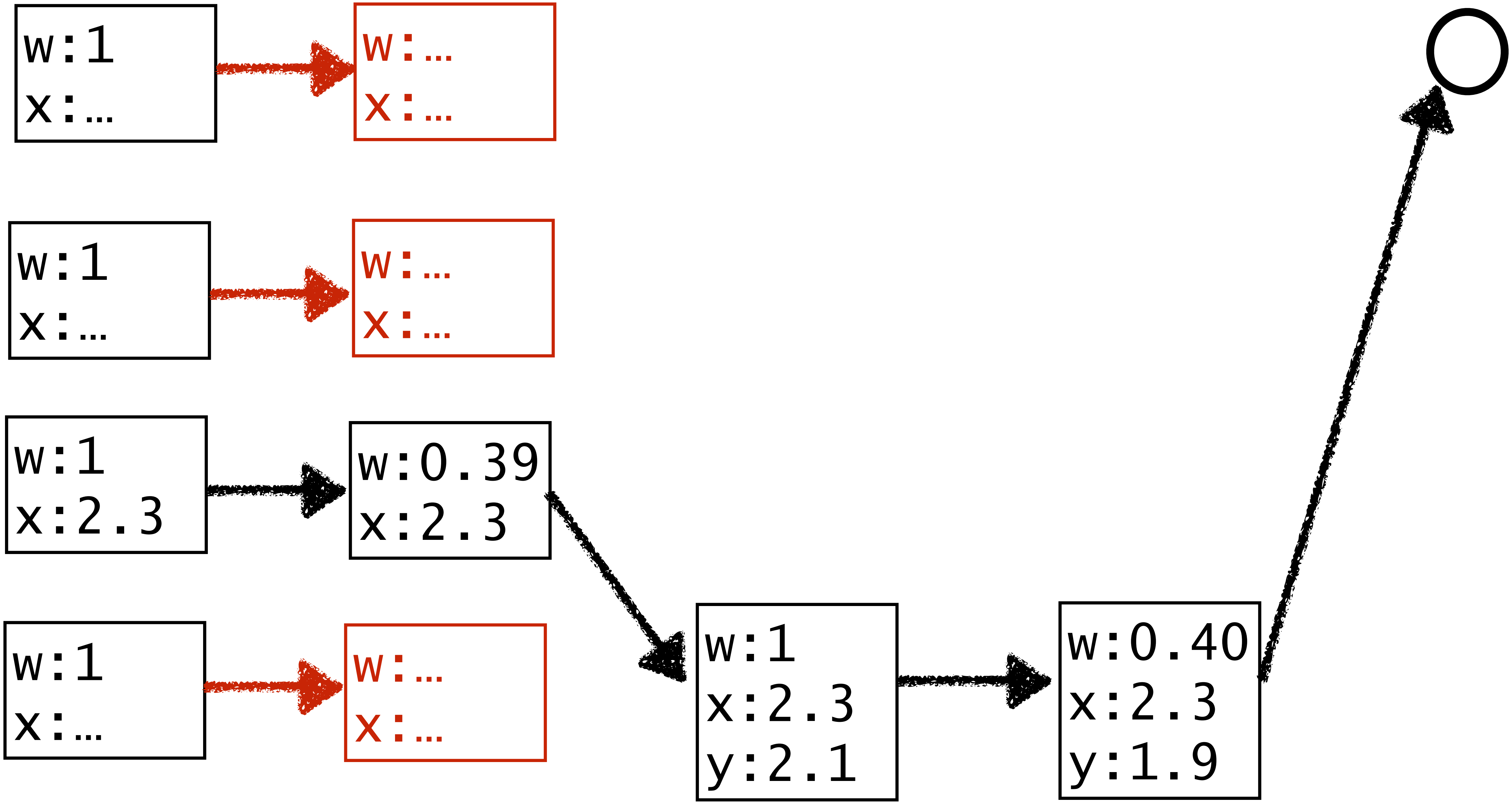
w:1
x:2.3
y:2.1

w:0.40
x:2.3
y:1.9

w:1
x:…

Particle Gibbs

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:…

w:…
x:…

w:1
x:…

w:…
x:…

w:1
x:2.3

w:0.39
x:2.3

w:1
x:…

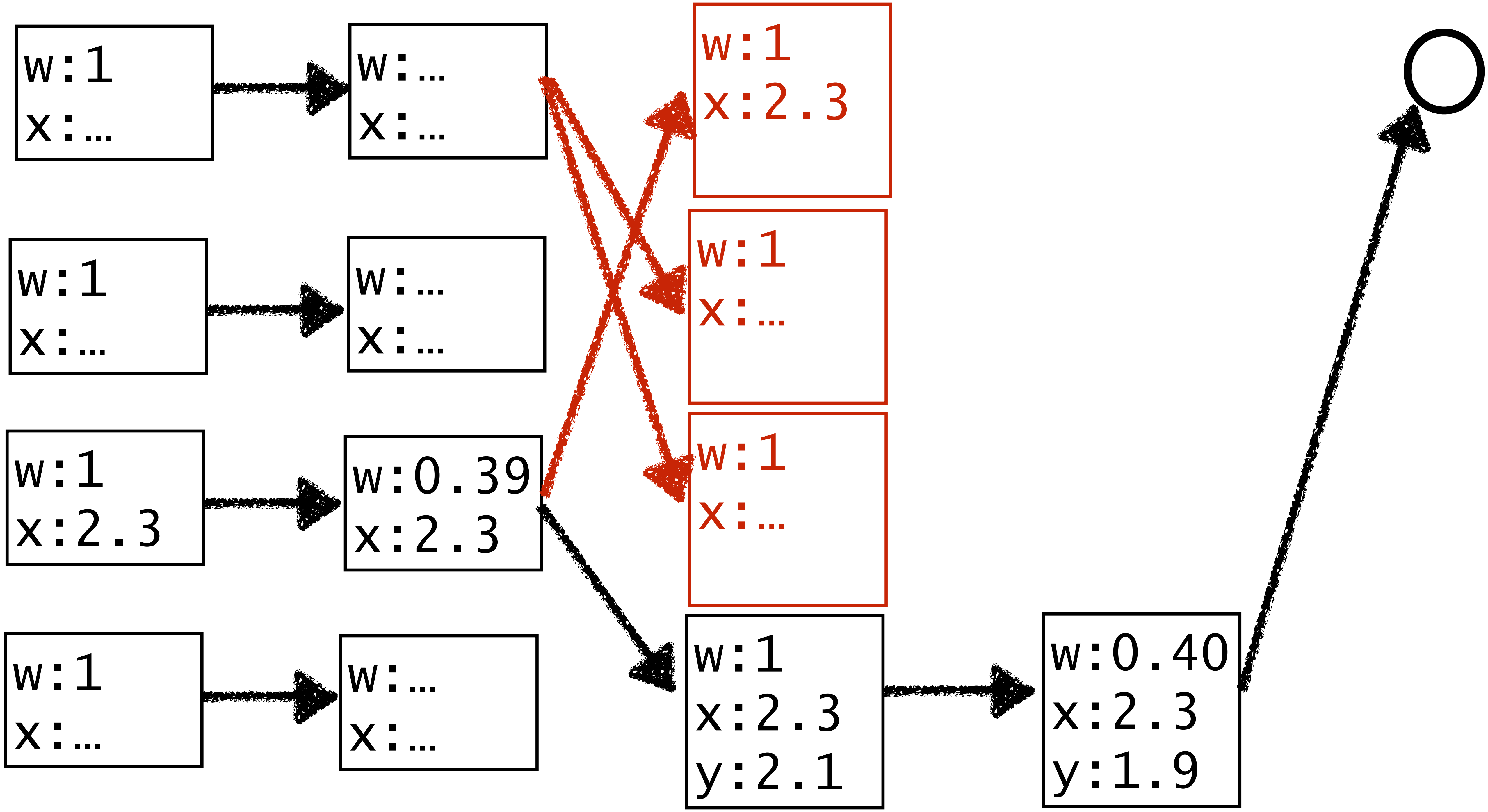
w:…
x:…

w:1
x:2.3
y:2.1

w:0.40
x:2.3
y:1.9

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Particle Gibbs



w:1
x:…

w:1
x:…

w:1
x:2.3

w:1
x:…

w:…
x:…

w:…
x:…

w:0.39
x:2.3

w:…
x:…

w:1
x:2.3

w:1
x:…

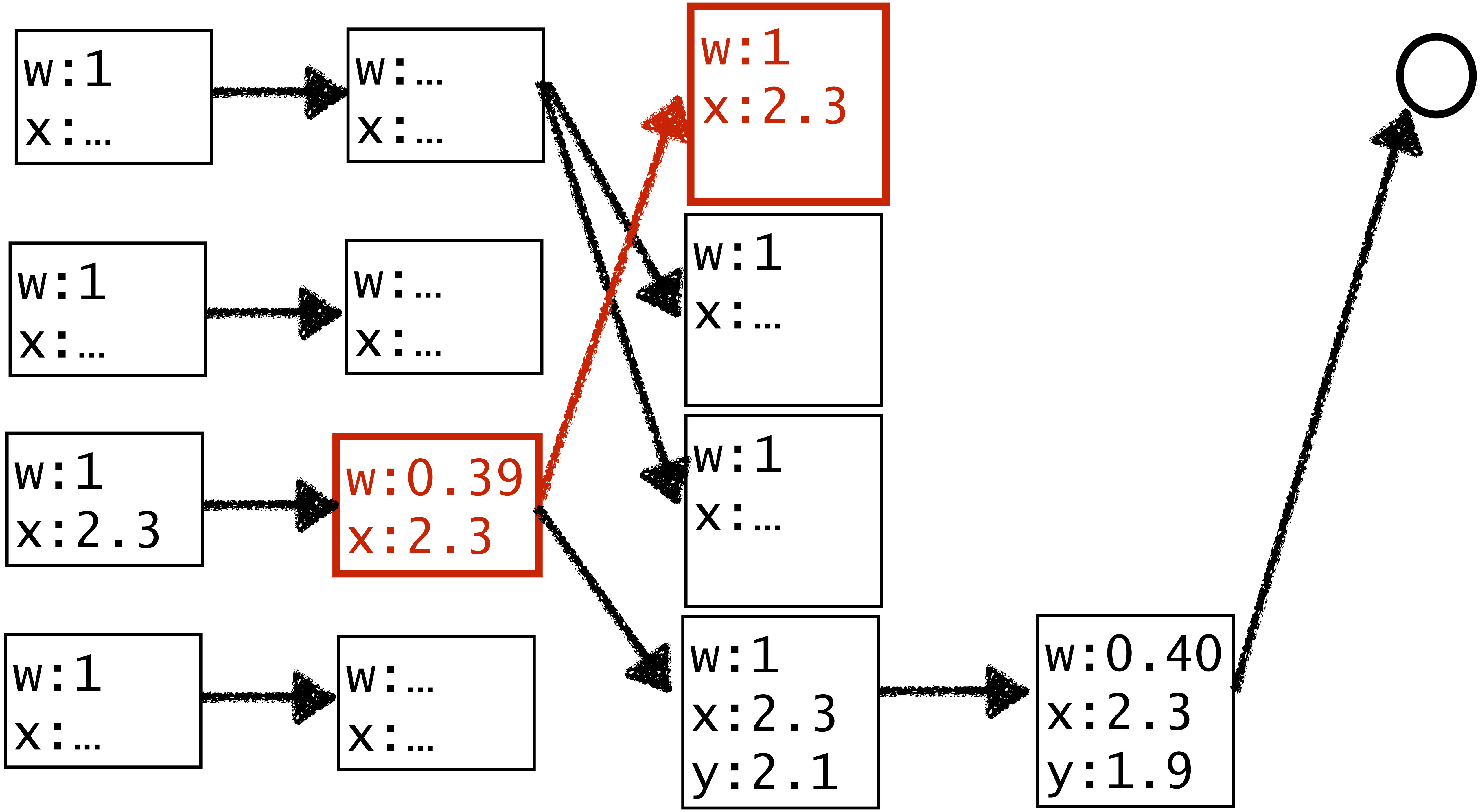
w:1
x:…

w:1
x:2.3
y:2.1

w:0.40
x:2.3
y:1.9

Particle Gibbs

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:…

w:1
x:…

w:1
x:2.3

w:1
x:…

w:…
x:…

w:…
x:…

w:0.39
x:2.3

w:…
x:…

w:1
x:2.3

w:1
x:…

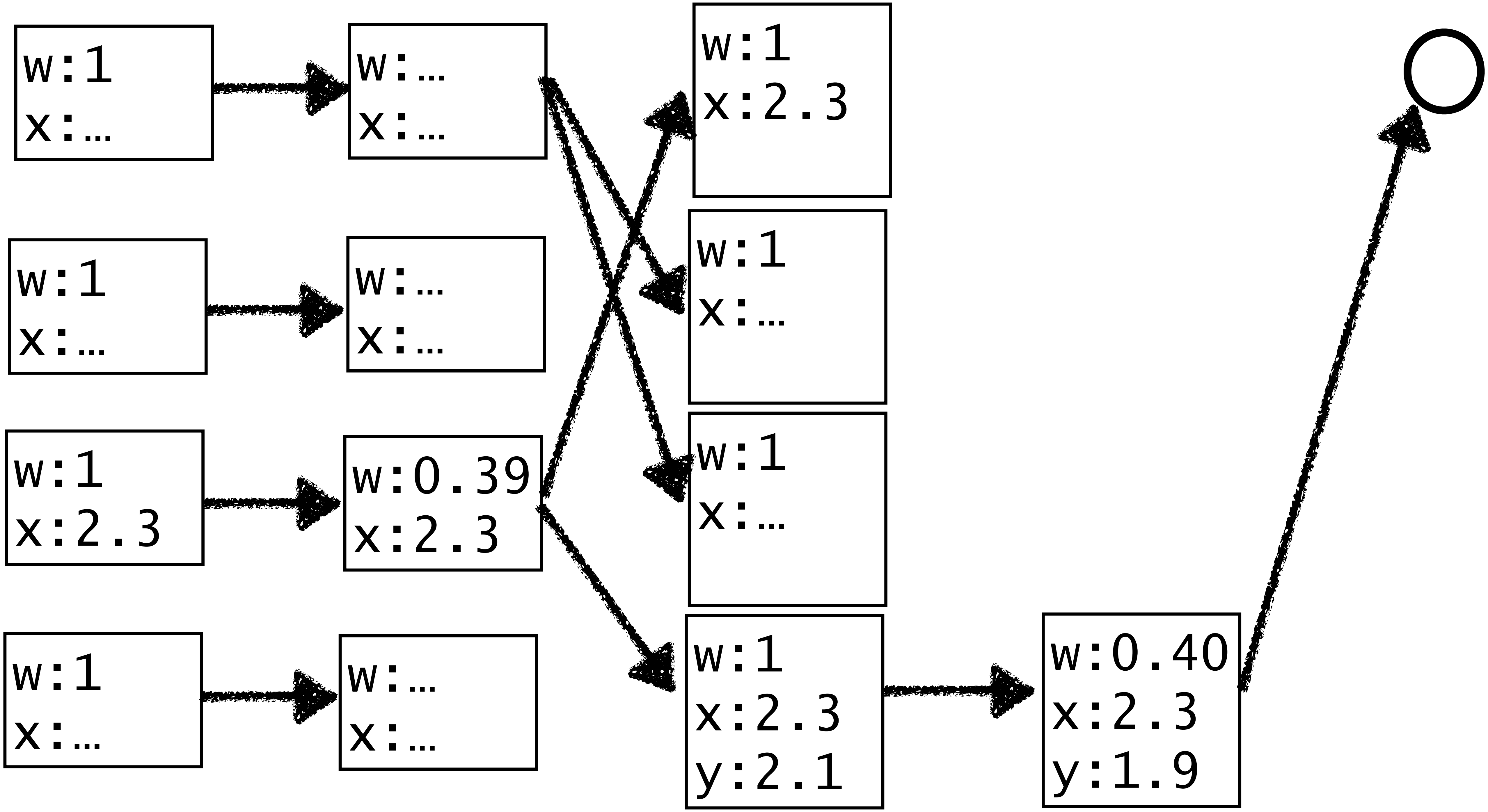
w:1
x:…

w:1
x:2.3
y:2.1

w:0.40
x:2.3
y:1.9

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Particle Gibbs



w:1
x:…

w:1
x:…

w:1
x:2.3

w:1
x:…

w:…
x:…

w:…
x:…

w:0.39
x:2.3

w:…
x:…

w:1
x:2.3

w:1
x:…

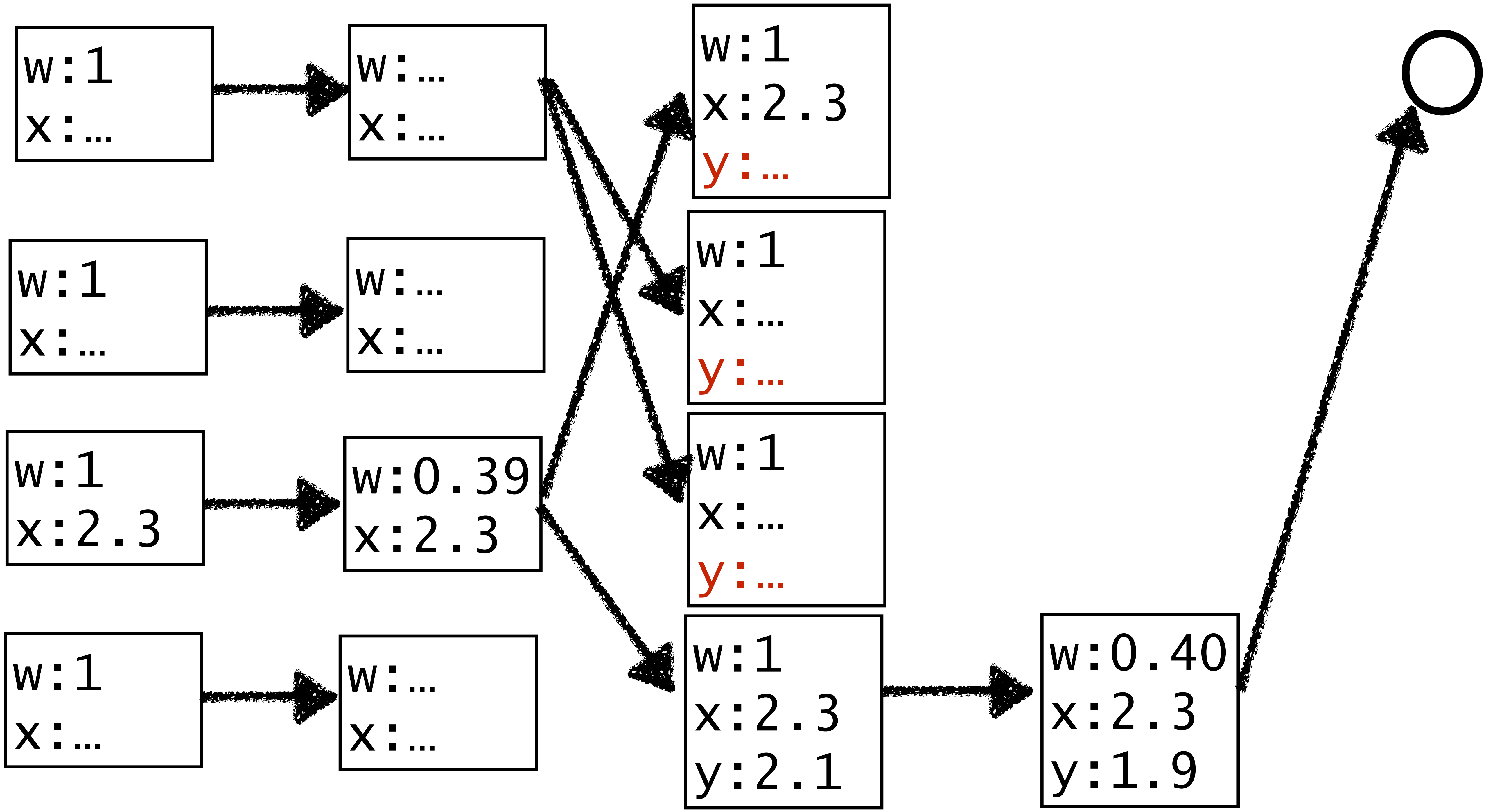
w:1
x:…

w:1
x:2.3
y:2.1

w:0.40
x:2.3
y:1.9

Particle Gibbs

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:…

w:1
x:…

w:1
x:2.3

w:1
x:…

w:…
x:…

w:…
x:…

w:0.39
x:2.3

w:…
x:…

w:1
x:2.3
y:…

w:1
x:…
y:…

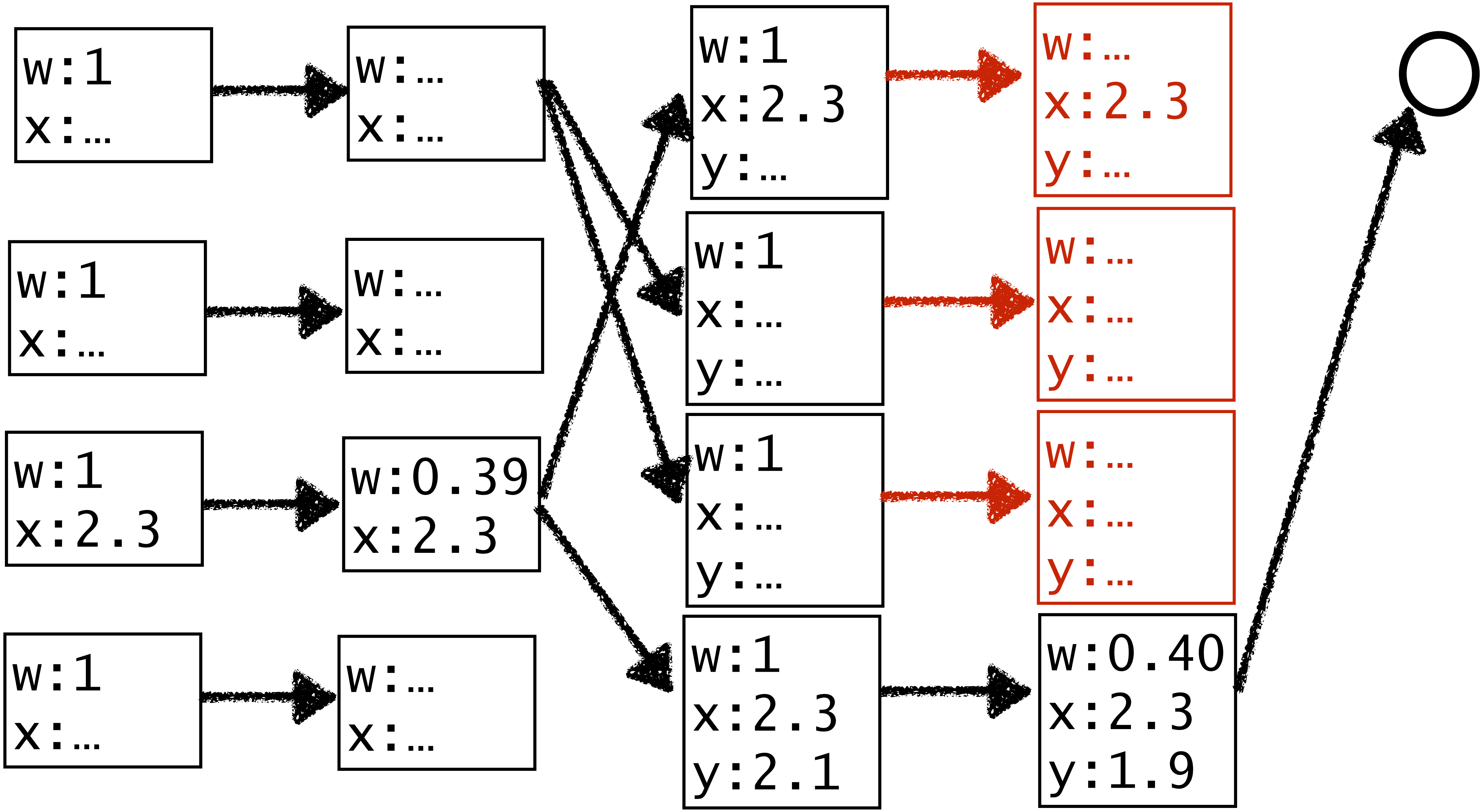
w:1
x:…
y:…

w:1
x:2.3
y:2.1

w:0.40
x:2.3
y:1.9

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Particle Gibbs



w:1
x:…

w:1
x:…

w:1
x:2.3

w:1
x:…

w:…
x:…

w:…
x:…

w:0.39
x:2.3

w:…
x:…

w:1
x:2.3
y:…

w:1
x:…
y:…

w:1
x:…
y:…

w:1
x:2.3
y:2.1

w:…
x:2.3
y:…

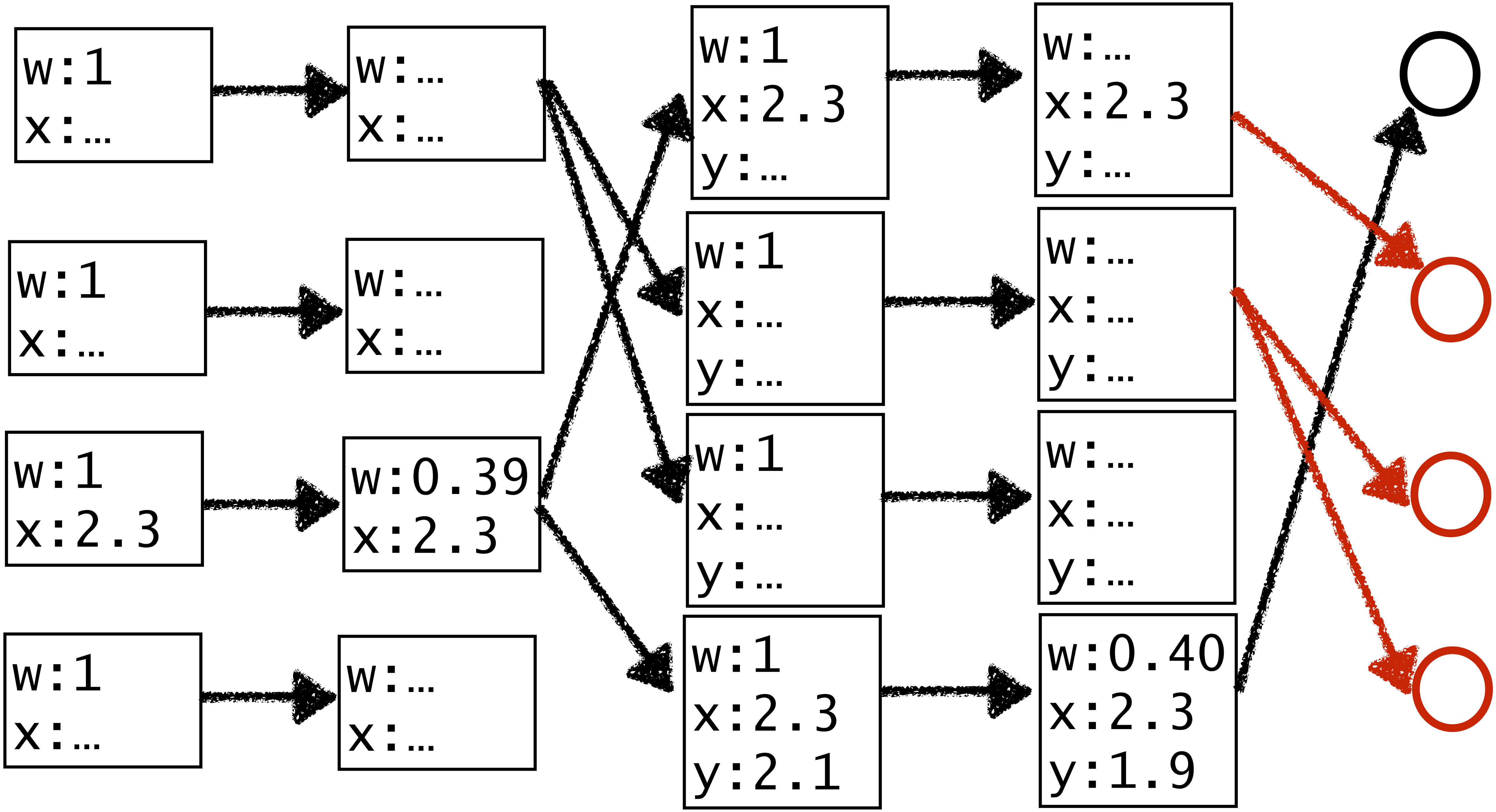
w:…
x:…
y:…

w:…
x:…
y:…

w:0.40
x:2.3
y:1.9

Particle Gibbs

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



w:1
x:…

w:1
x:…

w:1
x:2.3

w:1
x:…

w:…
x:…

w:…
x:…

w:0.39
x:2.3

w:…
x:…

w:1
x:2.3
y:…

w:1
x:…
y:…

w:1
x:…
y:…

w:1
x:2.3
y:2.1

w:…
x:2.3
y:…

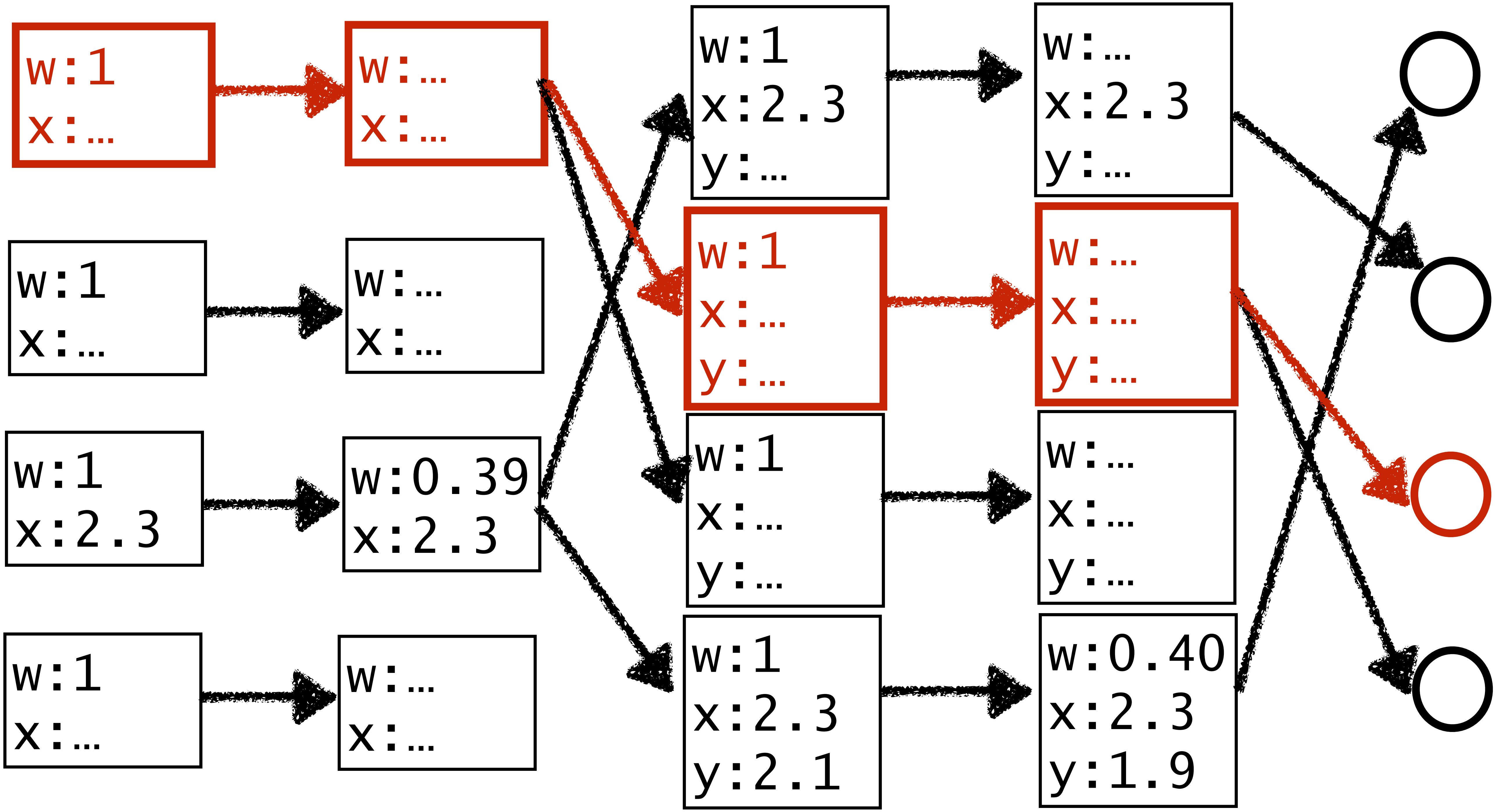
w:…
x:…
y:…

w:…
x:…
y:…

w:0.40
x:2.3
y:1.9

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Particle Gibbs



w:1
x:…

w:1
x:…

w:1
x:2.3

w:1
x:…

w:…
x:…

w:…
x:…

w:0.39
x:2.3

w:…
x:…

w:1
x:2.3
y:…

w:1
x:…
y:…

w:1
x:…
y:…

w:1
x:2.3
y:2.1

w:…
x:2.3
y:…

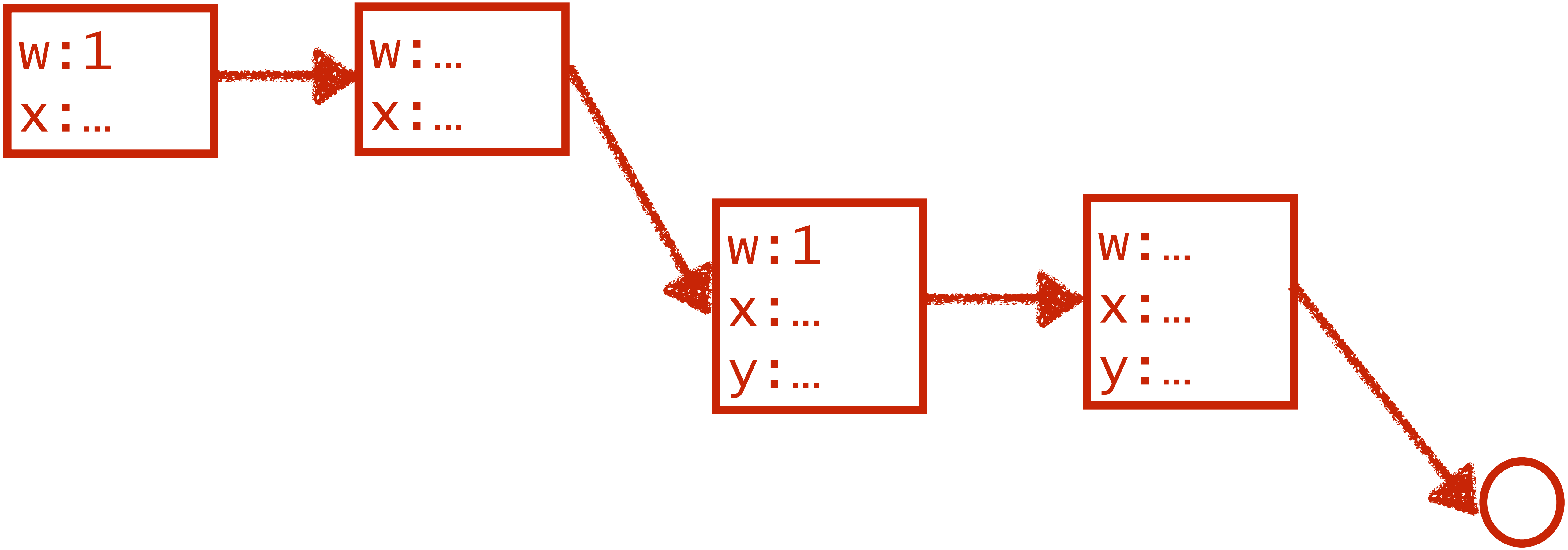
w:…
x:…
y:…

w:…
x:…
y:…

w:0.40
x:2.3
y:1.9

Particle Gibbs

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```



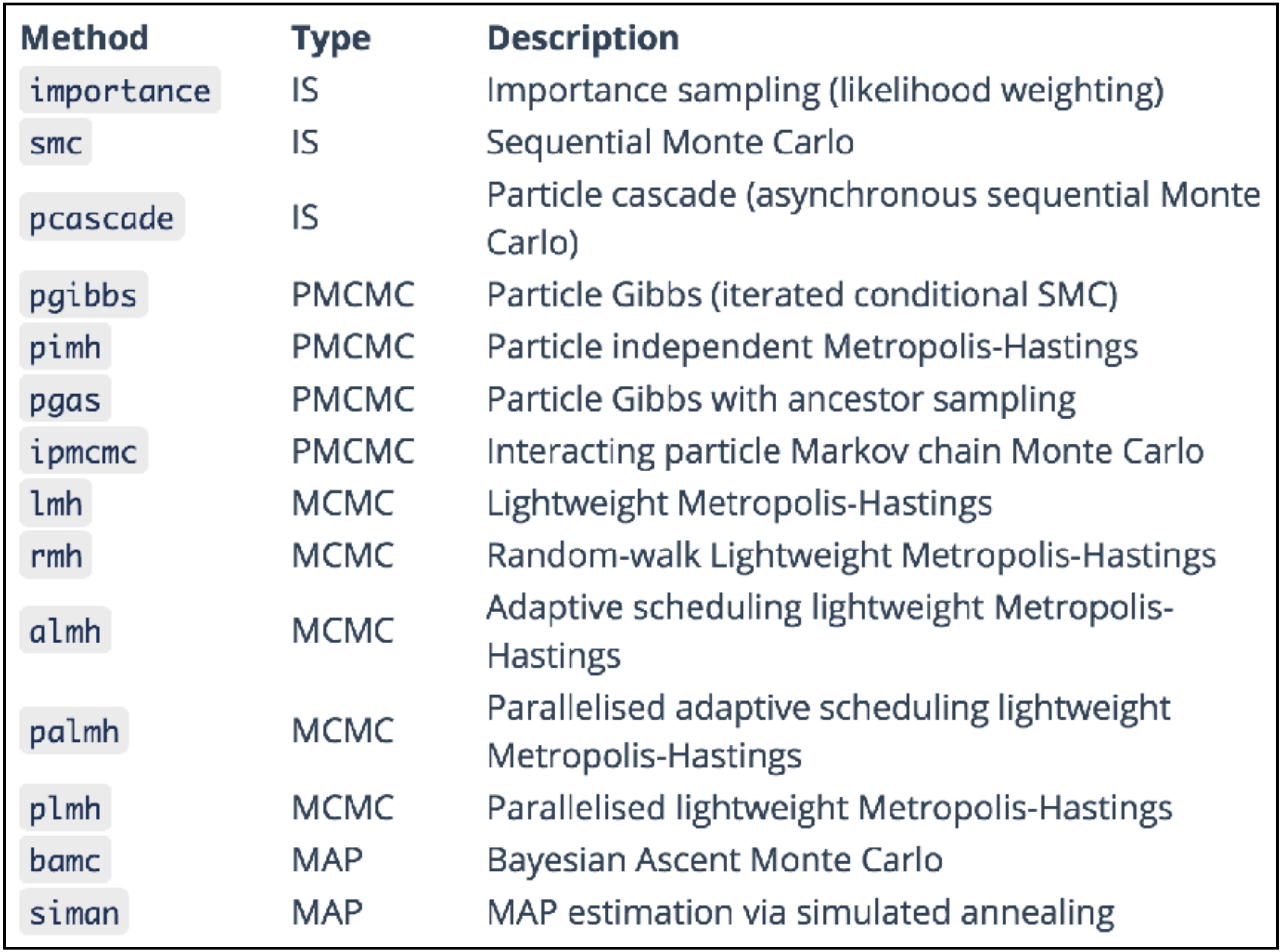
w:1
x:…

w:…
x:…

w:1
x:…
y:…

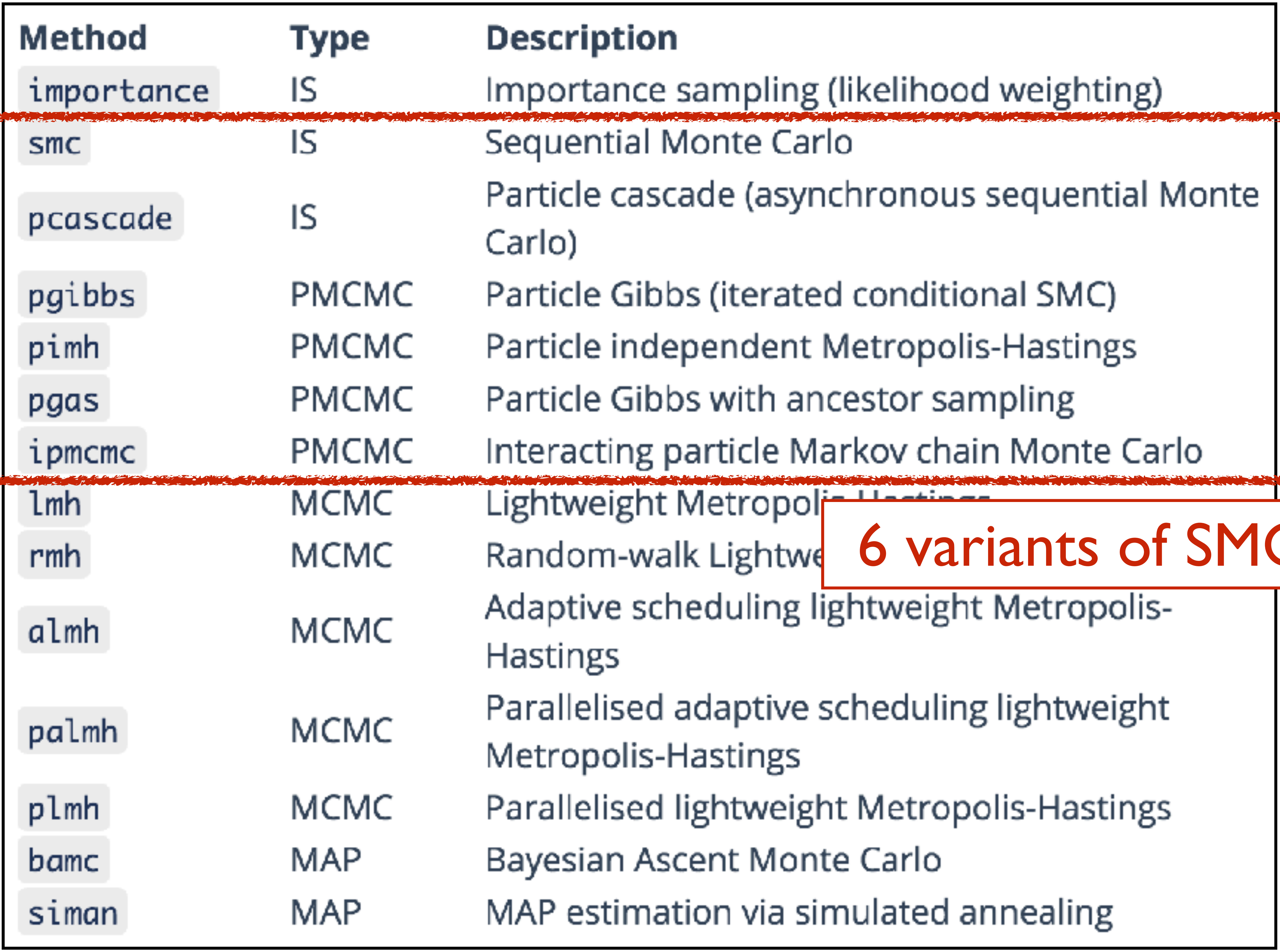
w:…
x:…
y:…

Inference algorithms for Anglican



Method	Type	Description
`importance`	IS	Importance sampling (likelihood weighting)
`smc`	IS	Sequential Monte Carlo
`pcascade`	IS	Particle cascade (asynchronous sequential Monte Carlo)
`pgibbs`	PMCMC	Particle Gibbs (iterated conditional SMC)
`pimh`	PMCMC	Particle independent Metropolis-Hastings
`pgas`	PMCMC	Particle Gibbs with ancestor sampling
`ipmcmc`	PMCMC	Interacting particle Markov chain Monte Carlo
`lmh`	MCMC	Lightweight Metropolis-Hastings
`rmh`	MCMC	Random-walk Lightweight Metropolis-Hastings
`almh`	MCMC	Adaptive scheduling lightweight Metropolis-Hastings
`palmh`	MCMC	Parallelised adaptive scheduling lightweight Metropolis-Hastings
`plmh`	MCMC	Parallelised lightweight Metropolis-Hastings
`bamc`	MAP	Bayesian Ascent Monte Carlo
`siman`	MAP	MAP estimation via simulated annealing

Inference algorithms for Anglican



Method	Type	Description
`importance`	IS	Importance sampling (likelihood weighting)
`smc`	IS	Sequential Monte Carlo
`pcascade`	IS	Particle cascade (asynchronous sequential Monte Carlo)
`pgibbs`	PMCMC	Particle Gibbs (iterated conditional SMC)
`pimh`	PMCMC	Particle independent Metropolis-Hastings
`pgas`	PMCMC	Particle Gibbs with ancestor sampling
`ipmcmc`	PMCMC	Interacting particle Markov chain Monte Carlo
`lmh`	MCMC	Lightweight Metropoli
`rmh`	MCMC	Random-walk Lightwe
`almh`	MCMC	Adaptive scheduling lightweight Metropolis-Hastings
`palmh`	MCMC	Parallelised adaptive scheduling lightweight Metropolis-Hastings
`plmh`	MCMC	Parallelised lightweight Metropolis-Hastings
`bamc`	MAP	Bayesian Ascent Monte Carlo
`siman`	MAP	MAP estimation via simulated annealing

6 variants of SMC

Key question

How to reason about prob. programs under such concurrent non-standard approximate semantics?

Concrete question 1: Correctness

[Q] Are these algo. correct for prob. programs?

Usually proved for $\mathbb{R}^n$ or simple cases.

Challenge 1: Expressiveness of prob. PLs.

Challenge 2: Subtle notion of correctness.

Concrete question 1: Correctness

[Q] Are these algo. correct for prob. programs?

Usually proved for $\mathbb{R}^n$ or simple cases.

Challenge 1: Expressiveness of prob. PLs.

Challenge 2: Subtle notion of correctness.

Seq. Monte Carlo gives a right answer (weak convergence) as the # of threads goes to ∞.

Concrete question 2: Refinement

Programs may be semantically equivalent but some are easy for these algo., and some hard.

[Q] Capture this difference by an algo.-specific refinement $\sqsubseteq$. Develop proof rules for $\sqsubseteq$.

Concrete question 2: Refinement

Programs may be semantically equivalent but some are easy for these algo., and some hard.

[Q] Capture this difference by an algo.-specific refinement ⊑. Develop proof rules for ⊑.

Prog1

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Concrete question 2: Refinement

Programs may be semantically equivalent but some are easy for these algo., and some hard.

[Q] Capture this difference by an algo.-specific refinement ⊑. Develop proof rules for ⊑.

Prog1

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Prog2

Concrete question 2: Refinement

Programs may be semantically equivalent but some are easy for these algo., and some hard.

[Q] Capture this difference by an algo.-specific refinement ⊑. Develop proof rules for ⊑.

Prog1

```
(let [x (sample (normal 0 2))
      a (observe (normal x 1) 2.1)
      y (sample (normal (* 0.9 x) 2))
      b (observe (normal y 1) 1.8)]
  [x y])
```

Prog2

Prog2 ⊑ Prog1

Concrete question 3: Good sublanguages

[Q1] Find a sublanguage that drops the time complexity of an inference algorithm.

[Q2] Find a sublanguage that allows a GPU-based implementation of an inference algo.

Reason 2:
Probabilistic PLs raise new semantic issues.

```
(let [F (fn []
          (let [s (sample (normal 0 2))
                b (sample (normal 0 6))]
            (fn [x] (+ (* s x) b))))
      f (add-change-points F 0 6)]
    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  f)
```

```
(let [F (fn []
          (let [s (sample (normal 0 2))
                b (sample (normal 0 6))]
            (fn [x] (+ (* s x) b))))
      f (add-change-points F 0 6)]
    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  f)
```

1. Higher-order functions.

Issue 1:
Higher-order functions

Measure theory provides a standard foundation of probability theory.

But it doesn't support HO fns well.

$$\mathrm{ev} : (\mathbb{R} \to_{m} \mathbb{R}) \times \mathbb{R} \to \mathbb{R}, \qquad \mathrm{ev}(f,x) = f(x).$$

[Aumann 61] ev is not measurable no matter which σ-algebra is used for $\mathbb{R} \to_{m} \mathbb{R}$.

```
(let [F (fn []
          (let [s (sample (normal 0 2))
                b (sample (normal 0 6))]
            (fn [x] (+ (* s x) b))))
      f (add-change-points F 0 6)]
    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  f)
```

1. Higher-order functions.

```
(let [F (fn []
          (let [s (sample (normal 0 2))
                b (sample (normal 0 6))]
            (fn [x] (+ (* s x) b))))
      f (add-change-points F 0 6)]
    (observe (normal (f 0) .5) .6)
    (observe (normal (f 1) .5) .7)
    (observe (normal (f 2) .5) 1.2)
    (observe (normal (f 3) .5) 3.2)
    (observe (normal (f 4) .5) 6.8)
    (observe (normal (f 5) .5) 8.2)
    (observe (normal (f 6) .5) 8.4)

  f)
```

1. Higher-order functions.
2. Conditioning and prog. eqs.

Issue 2:
Conditioning and prog. eqs

$$[\![\texttt{e:real}]\!] \in \mathsf{M}(\mathbb{R})$$

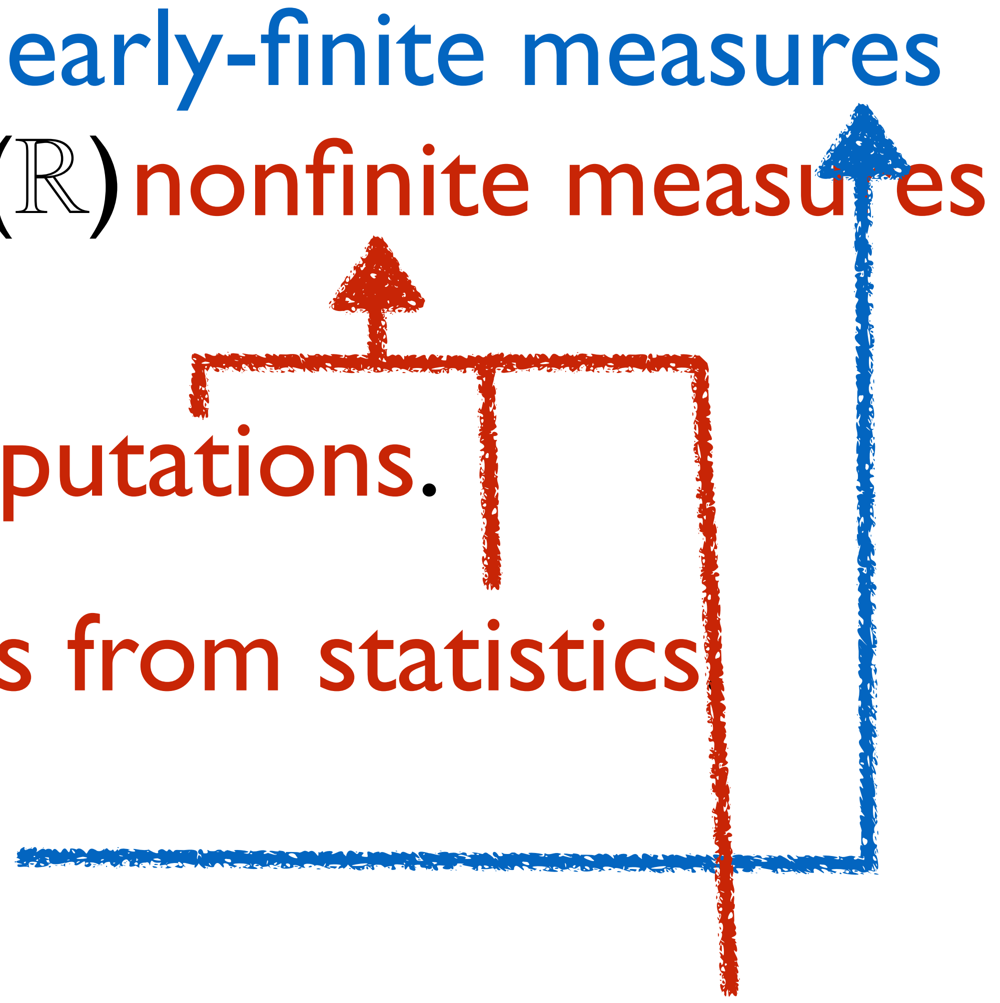
- M should model prob. computations.
- M should validate equations from statistics.
- M should be commutative.
- Difficult to find such M due to conditioning.

Issue 2:
Conditioning and prog. eqs

$[\![\ \texttt{e:real} \ ]\!] \in M(\mathbb{R})$ nonfinite measures

- M should model prob. computations.
- M should validate equations from statistics.
- M should be commutative.
- Difficult to find such M due to conditioning.

Issue 2:
Conditioning and prog. eqs



nearly-finite measures

$[\![\texttt{e:real}]\!] \in M(\mathbb{R})$ nonfinite measures

- M should model prob. computations.
- M should validate equations from statistics.
- M should be commutative.
- Difficult to find such M due to conditioning.

Issue 3:
Meta-programming features

My ML colleagues are very much interested in probabilistic models for programs.

They express such models using quote & eval.

[Q] How to interpret meta-programming features in Anglican/Church/Venture?

My research*:
Address issues 1&2 with Quasi-Borel spaces.

* based on Heunen et al.'s LICS'17

Big picture 1:
Extend measure theory
using category theory.

1. Higher-order fns.
2. Conditioning, prog. eqs.

1. Higher-order fns.
2. Conditioning, prog. eqs.

Meas_B

~~1. Higher-order fns.~~
2. Conditioning, prog. eqs.

$$\begin{tikzcd}
\mathrm{Meas}_B \arrow[dr, swap, "\text{Yoneda embedding}"] & \\
& {[\mathrm{Meas}_B{}^{\mathrm{op}}, \mathrm{Set}]_\Pi}
\end{tikzcd}$$

~~1. Higher-order fns.~~
2. Conditioning, prog. eqs.

$$\begin{tikzcd}
\mathrm{Meas}_B \arrow[dr, "\text{Yoneda embedding}"'] & \\
& {[\mathrm{Meas}_B{}^{op}, \mathrm{Set}]_\Pi}
\end{tikzcd}$$

Preserves nearly all the structures

~~1. Higher-order fns.~~

2. Conditioning, prog. eqs.

$$\begin{tikzcd}
\mathrm{Meas}_B \arrow[dr, "\text{Yoneda embedding}"'] & \\
& {[\mathrm{Meas}_B{}^{op}, \mathrm{Set}]_\Pi}
\end{tikzcd}$$

Enough structure for function types

~~1. Higher-order fns.~~
2. Conditioning, prog. eqs.

$$\begin{tikzcd}
\mathrm{Meas}_B \arrow[dr, "\text{Yoneda embedding}"'] & \\
& {[\mathrm{Meas}_B{}^{op}, \mathrm{Set}]_\Pi}
\end{tikzcd}$$

~~1. Higher-order fns.~~
2. Conditioning, prog. eqs.

$$\begin{tikzcd}
\mathrm{Meas}_B \arrow[rr, dashed] \arrow[dr, "\text{Yoneda embedding}"'] & & \mathrm{QBS} \arrow[dl, "\text{Full subcat. of separated functors}"] \\
& {[\mathrm{Meas}_B{}^{op}, \mathrm{Set}]_\Pi} &
\end{tikzcd}$$

~~1. Higher-order fns.~~
2. Conditioning, prog. eqs.

Function spaces (CCC).
Concrete (extensional).

$$\begin{tikzcd}
\mathrm{Meas}_B \arrow[rr, dashed] \arrow[dr, "\text{Yoneda embedding}"'] & & \mathrm{QBS} \arrow[dl, "\text{Full subcat. of separated functors}"] \\
& {[\mathrm{Meas}_B{}^{op}, \mathrm{Set}]_\Pi} &
\end{tikzcd}$$

~~1. Higher-order fns.~~
~~2. Conditioning, prog. eqs.~~

Strong monad of s-finite kernels

$$\begin{tikzcd}
\mathrm{Meas}_B \arrow[rr, dashed] \arrow[dr, "\text{Yoneda embedding}"'] & & \mathrm{QBS} \arrow[loop right, "\mathrm{SFinKer}"] \arrow[dl, "\text{Full subcat. of separated functors}"] \\
& {[\mathrm{Meas}_B{}^{op}, \mathrm{Set}]_{\Pi}} &
\end{tikzcd}$$

~~1. Higher-order fns.~~

~~2. Conditioning, prog. eqs.~~

$$\begin{tikzcd}
\mathrm{Meas}_B \arrow[rr, dashed] \arrow[dr, swap, "\text{Yoneda embedding}"] & & \mathrm{QBS} \arrow[loop right, "\mathrm{SFinKer}"] \arrow[dl, "\text{Full subcat. of separated functors}"] \\
& {[\mathrm{Meas}_B{}^{\mathrm{op}}, \mathrm{Set}]_{\Pi}} &
\end{tikzcd}$$

Big picture 2:
Random element first.

Random element α in X

Random element α in X

$$\alpha : \Omega \to X$$

- X - set of values.
- Ω - set of random seeds.
- Random seed generator.

Random element α in X
in measure theory

$$\alpha : \Omega \to X$$

- X - set of values.
- Ω - set of random seeds.
- Random seed generator.

Random element α in X
in measure theory

$$\alpha : \Omega \rightarrow X$$

- X - set of values.
- Ω - set of random seeds.
- Random seed generator.

1. $\Sigma \subseteq 2^{\Omega}$, $\Theta \subseteq 2^{X}$

Random element α in X
in measure theory

$$\alpha : \Omega \to X$$

- X - set of values.
- Ω - set of random seeds.
- Random seed generator.

1. $\Sigma \subseteq 2^{\Omega}, \Theta \subseteq 2^{X}$
2. $\mu : \Sigma \to [0,1]$

Random element α in X in measure theory

$\alpha : \Omega \to X$ is a random element if $\alpha^{-1}(A) \in \Sigma$ for all $A \in \Theta$

- X - set of values.
- Ω - set of random seeds.
- Random seed generator.

1. $\Sigma \subseteq 2^{\Omega}$, $\Theta \subseteq 2^{X}$
2. $\mu : \Sigma \to [0,1]$

Random element α in X

$$\alpha : \Omega \to X$$

- X - set of values.
- Ω - set of random seeds.
- Random seed generator.

Random element α in X in quasi-Borel spaces

$$\alpha : \Omega \to X$$

- X - set of values.
- Ω - set of random seeds.
- Random seed generator.

Random element α in X
in quasi-Borel spaces

$$\alpha : \mathbb{R} \to X$$

- X - set of values.
- $\mathbb{R}$ - set of random seeds.
- Random seed generator.

1. $\mathbb{R}$ as random source
2. Borel subsets $\mathfrak{B} \subseteq 2^{\mathbb{R}}$

Random element α in X in quasi-Borel spaces

$$\alpha : \mathbb{R} \to X$$

- X - set of values.
- $\mathbb{R}$ - set of random seeds.
- Random seed generator.

1. $\mathbb{R}$ as random source
2. Borel subsets $\mathfrak{B} \subseteq 2^{\mathbb{R}}$

Random element α in X
in quasi-Borel spaces

$$\alpha : \mathbb{R} \to X$$

- X - set of values.
- $\mathbb{R}$ - set of random seeds.
- Random seed generator.

1. $\mathbb{R}$ as random source
2. Borel subsets $\mathfrak{B} \subseteq 2^{\mathbb{R}}$
3. $M \subseteq [\mathbb{R} \to X]$

Random element α in X in quasi-Borel spaces

$\alpha : \mathbb{R} \to X$ is a random variable if $\alpha \in M$

- X - set of values.
- $\mathbb{R}$ - set of random seeds.
- Random seed generator.

1. $\mathbb{R}$ as random source
2. Borel subsets $\mathfrak{B} \subseteq 2^{\mathbb{R}}$
3. $M \subseteq [\mathbb{R} \to X]$

- Measure theory:
  - Measurable space $(X, \Theta \subseteq 2^X)$.
  - Random element is an induced concept.
- QBS:
  - Quasi-Borel space $(X, M \subseteq [\mathbb{R} \to X])$.
  - M is the set of random elements.

Quasi-Borel spaces

- New axiomatisation of probability theory.
- Enabled us to generalise classical results in probability theory such as de Finetti thm.

Try probabilistic PLs

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

WebPPL:
http://webppl.org/