Probabilistic Programming

Hongseok Yang
University of Oxford

Joint work with Diane Gallois-Wong, Chris Heunen, Ohad Kammar,
Sam Staton, David Tolpin, Jan-Willem van de Meent, Frank Wood

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

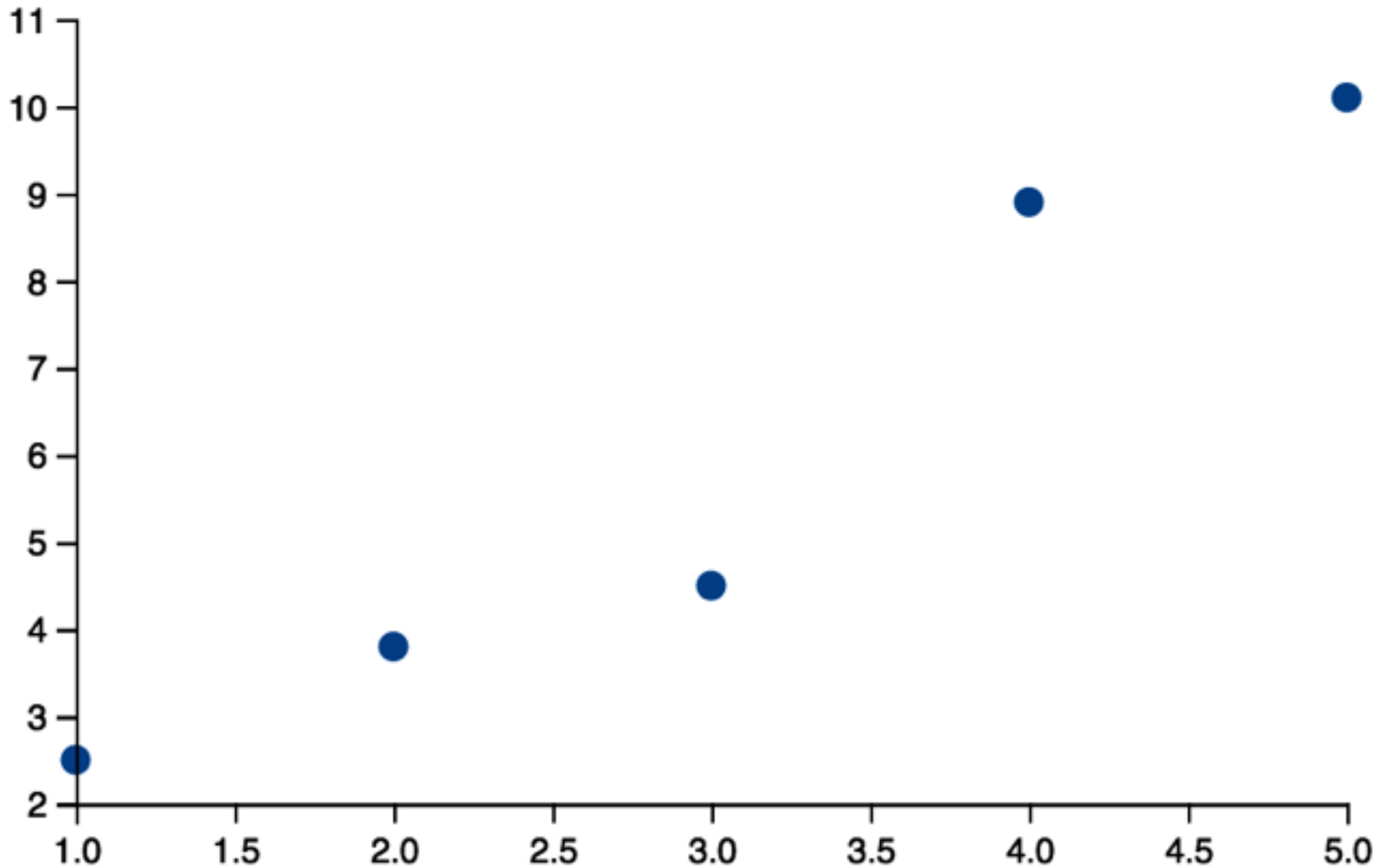
in a prob. prog. language

as a program

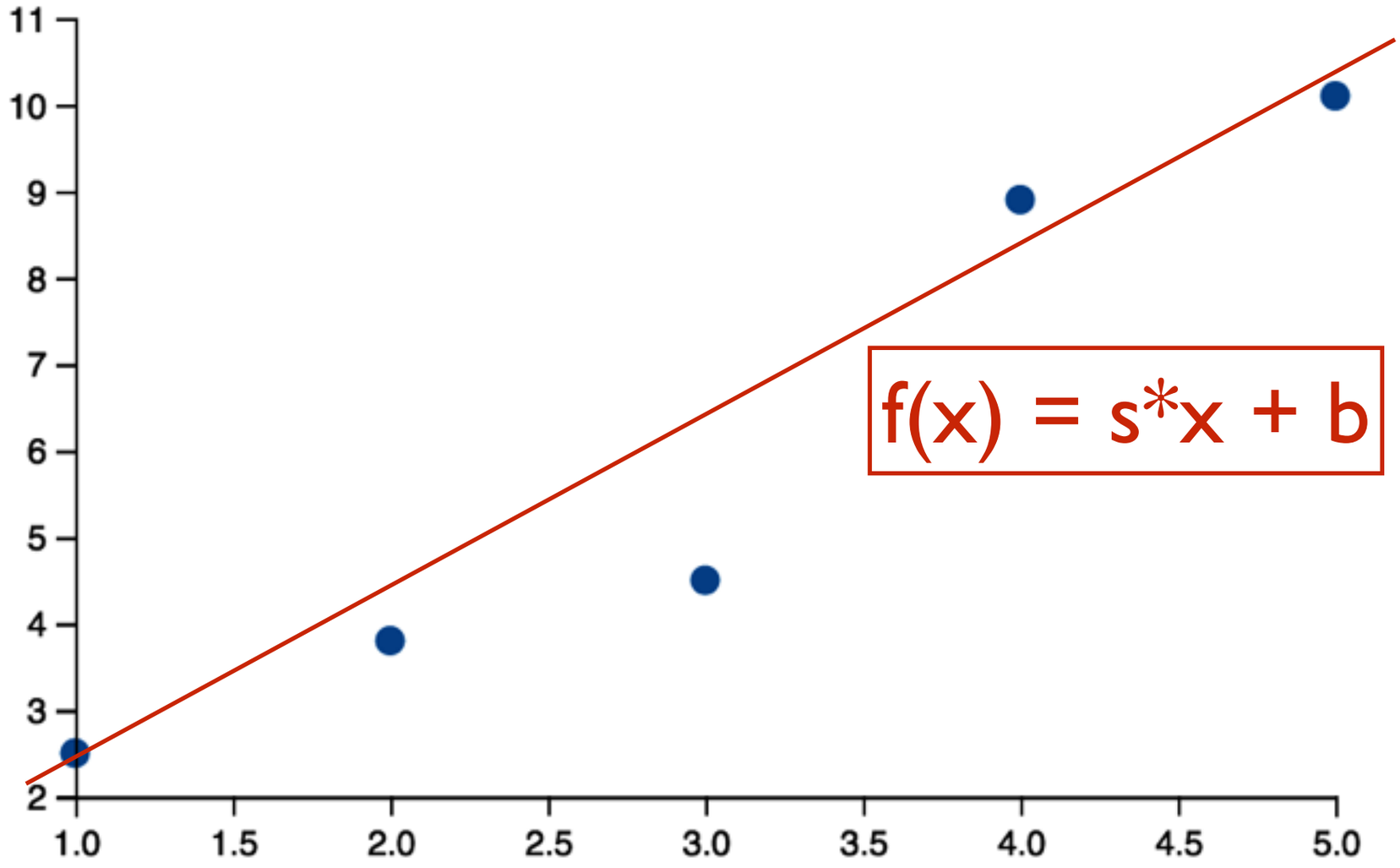
1. Develop a new probabilistic (generative) model.
2. ~~Design an inference algorithm for the model.~~
3. Using ~~the algo~~, fit the model to the data.

a generic inference algo.
of the language

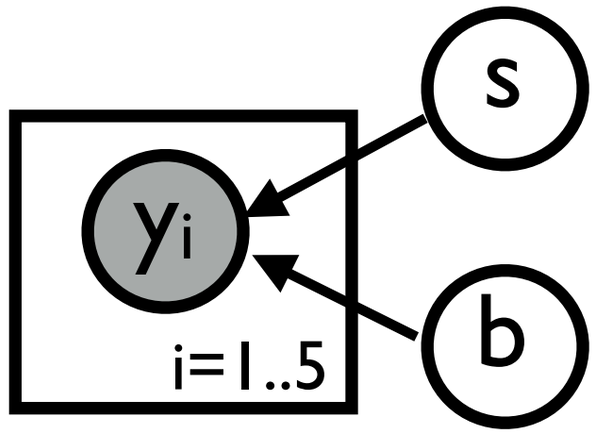
Line fitting



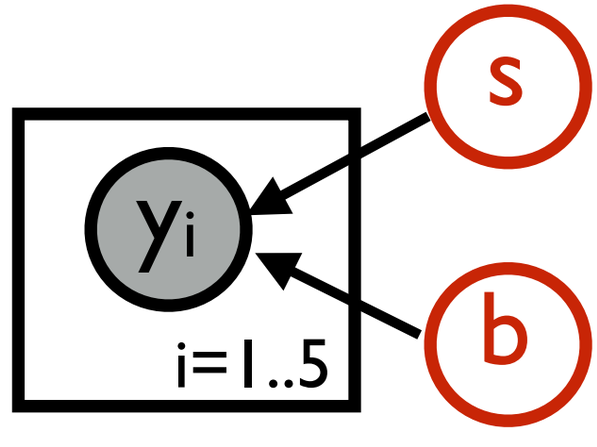
Line fitting



Bayesian generative model

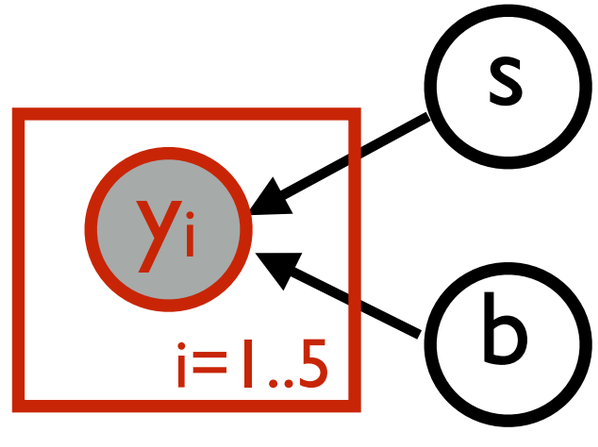


Bayesian generative model



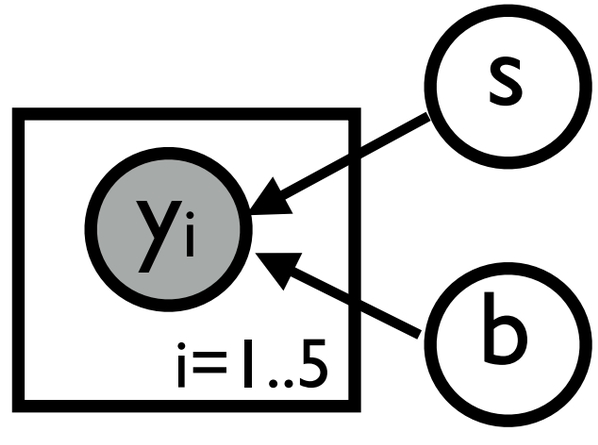

```
s     ~ normal(0, 10)
b     ~ normal(0, 10)
```

Bayesian generative model




```
s      ~ normal(0, 10)
b      ~ normal(0, 10)
f(x)   = s*x + b
y_i    ~ normal(f(i), 1)
           where i = 1 .. 5
```

Bayesian generative model




```
s      ~ normal(0, 10)
b      ~ normal(0, 10)
f(x)   = s*x + b
y_i    ~ normal(f(i), 1)
              where i = 1 .. 5
```

Q: posterior of (s,b) given y_1=2.5, …, y_5=10.1?

Anglican program

```
(let [s (sample (normal 0 10))
      b (sample (normal 0 10))
      f (fn [x] (+ (* s x) b))]
```

Anglican program

```
(let [s (sample (normal 0 10))
      b (sample (normal 0 10))
      f (fn [x] (+ (* s x) b))]

  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)
```

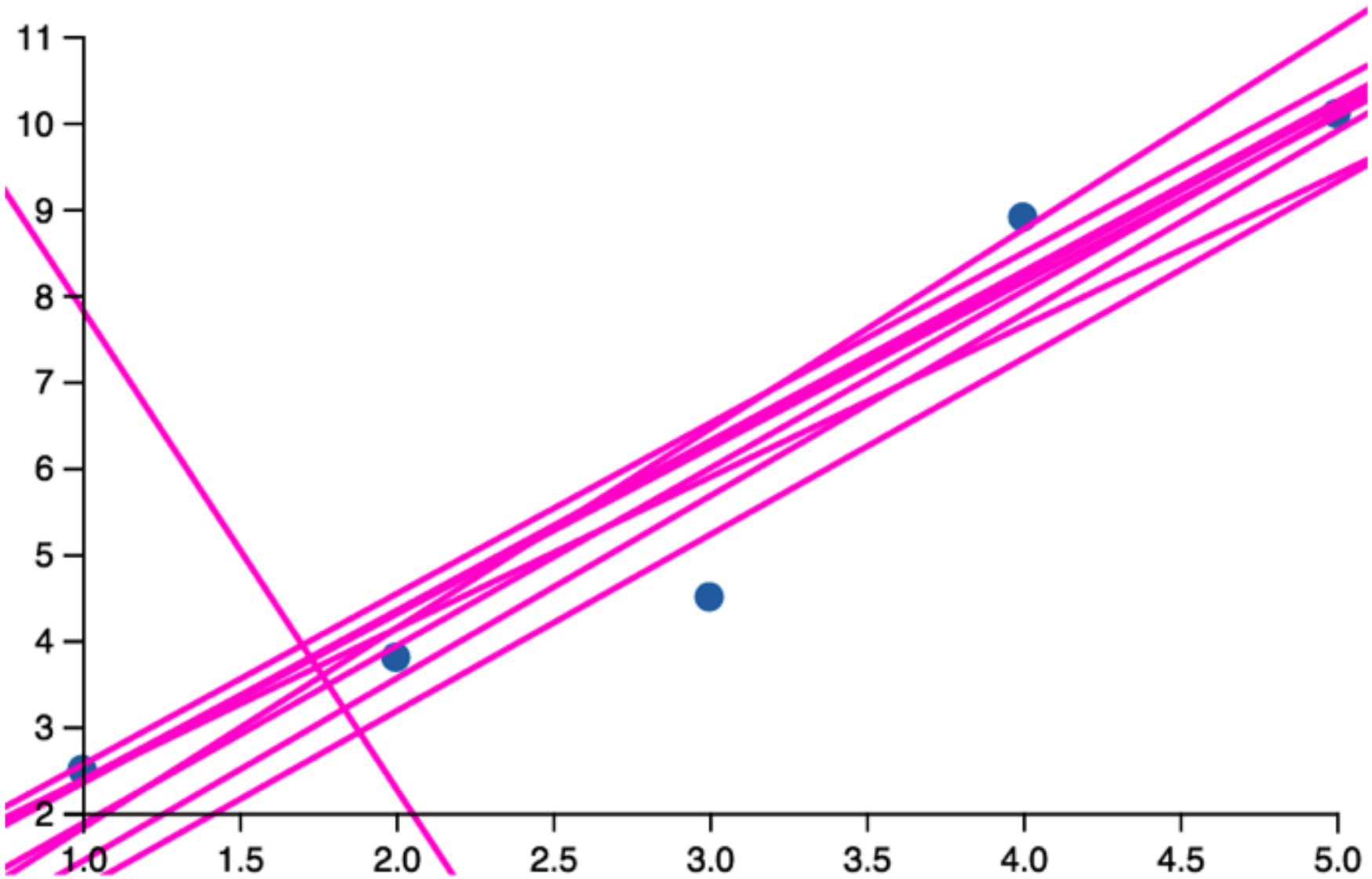
Anglican program

```
(let [s (sample (normal 0 10))
      b (sample (normal 0 10))
      f (fn [x] (+ (* s x) b))]

  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)

  (predict :sb [s b]))
```

Anglican program

```
(let [s (sample (normal 0 10))
      b (sample (normal 0 10))
      f (fn [x] (+ (* s x) b))]
  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)

  (predict :sb [s b]))
```

Samples from posterior



Why should one care about prob. programming?

My favourite answer

"Because probabilistic programming is a good way to build an AI." (My ML colleague)

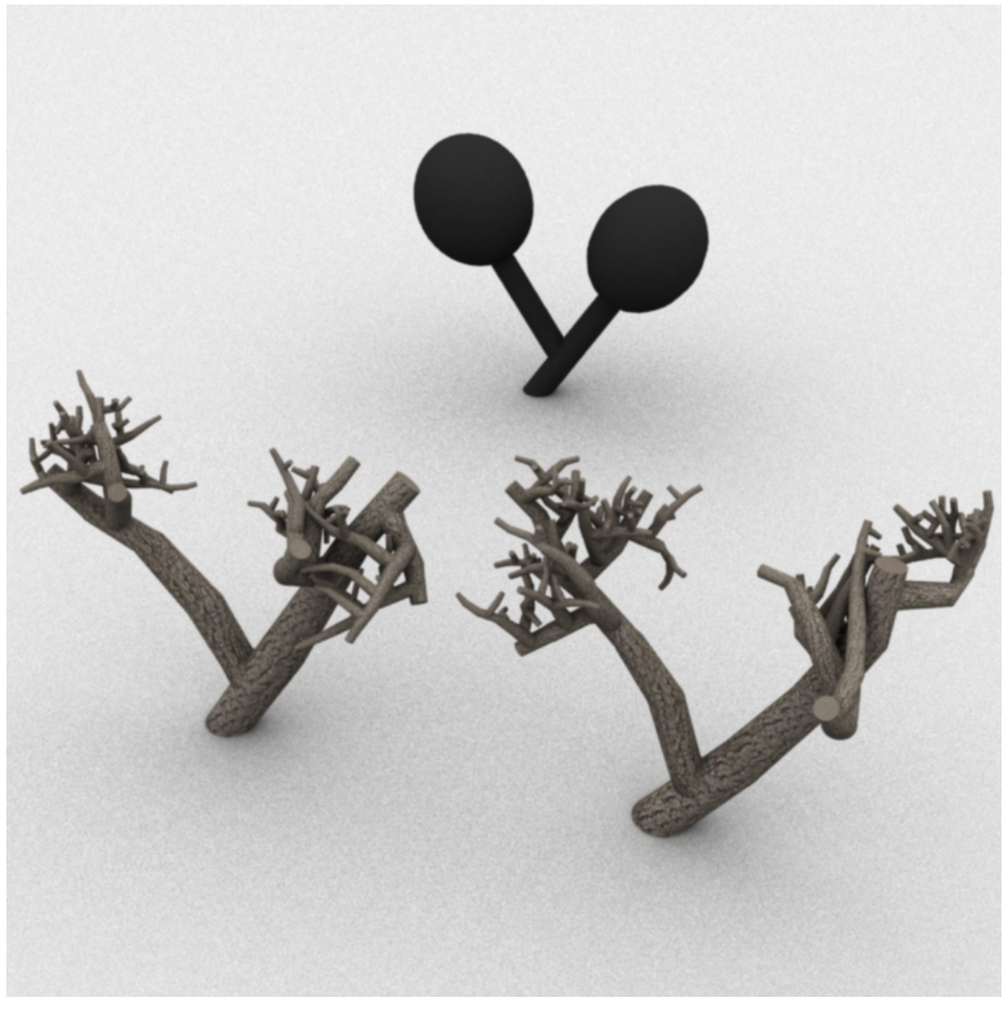
Lesson 1

- The expressiveness of the language matters.
  - Powerful language features: High-order fns, meta-programming features, etc.
  - New language concepts: Distribution objects, nested queries, stochastic future, etc.

Lesson 2

- Look at advanced models.
- Procedural modelling.
- Nonparametric Bayesian models.
- Reinforcement learning.

Procedural modelling



Ritchie, Mildenhall, Goodman, Hanrahan [2015]

Procedural modelling

```
future.create(function(i, frame, prev)
        if flip(T.branchProb(depth, i)) then
                -- Theta mean/variance based on avg weighted by
                local theta_mu, theta_sigma = T.estimateThetaD:
                local theta = gaussian(theta_mu, theta_sigma)
                local maxbranchradius = 0.5*(nextframe.center
                local branchradius = math.min(uniform(0.9, 1)
                local bframe, prev = T.branchFrame(splitFrame,
                branch(bframe, depth+1, prev)
end
```

Ritchie, Mildenhall, Goodman, Hanrahan [2015]

Procedural

Asynchronous function call via future

```
future.create(function(i, frame, prev)
        if flip(T.branchProb(depth, i)) then
                -- Theta mean/variance based on avg weighted by
                local theta_mu, theta_sigma = T.estimateThetaD:
                local theta = gaussian(theta_mu, theta_sigma)
                local maxbranchradius = 0.5*(nextframe.center
                local branchradius = math.min(uniform(0.9, 1)
                local bframe, prev = T.branchFrame(splitFrame,
                branch(bframe, depth+1, prev)
end
```

Ritchie, Mildenhall, Goodman, Hanrahan [2015]

Nonparametric Bayesian: Indian buffer process

```
(define (ibp-stick-breaking-process concentration base-measure)
  (let ((sticks (mem (lambda j (random-beta 1.0 concentration))))
        (atoms  (mem (lambda j (base-measure)))))
    (lambda ()
      (let loop ((j 1) (dualstick (sticks 1)))
        (append (if (flip dualstick)                ;; with prob. dualstick
                    (atoms j)                       ;;   add feature j
                    '())                            ;; otherwise, next stick
                (loop (+ j 1) (* dualstick (sticks (+ j 1))))))))))
```

Roy et al. 2008

Nonparametric Bayesian: Indian buffer process

```
(define (ibp-stick-breaking-process concentration base-measure)
  (let ((sticks (mem (lambda j (random-beta 1.0 concentration))))
        (atoms  (mem (lambda j (base-measure)))))
    (lambda ()
      (let loop ((j   ) (dualstick (sticks 1))}
        (append (if   flip dualstick)                ;; with prob. dualstick
                   atoms j)                          ;;   add feature j
                   ))                                ;; otherwise, next stick
                (l    (+ j 1) (* dualstick (sticks (+ j 1)))) ))))))
```

Lazy infinite array

Roy et al. 2008

Nonparametric Bayesian: Indian buffer p

Higher-order parameter

```
(define (ibp-stick-breaking-process concentration base-measure)
  (let ((sticks (mem (lambda j (random-beta 1.0 concentration))))
        (atoms  (mem (lambda j (base-measure)))))
    (lambda ()
      (let loop ((j 1) (dualstick (sticks 1)))
        (append (if (flip dualstick)                ;; with prob. dualstick
                    (atoms j)                       ;;   add feature j
                    '())                            ;; otherwise, next stick
                (loop (+ j 1) (* dualstick (sticks (+ j 1))))))))))
```

Roy et al. 2008

My research 1:
Program optimisation

Can program language techniques be used to do efficient inference on Anglican programs?

Can program language techniques be used to do efficient inference on Anglican programs?

1. Small. No assignments.
2. Continuous distributions.
3. High-order functions.
4. Lisp-style quote/eval.

Can program language techniques be used to ~~do efficient inference on~~ Anglican programs? help MC inference algorithms for

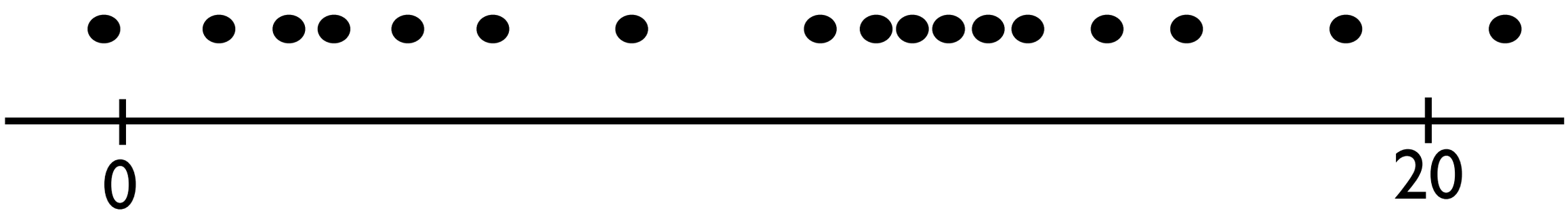
Can program language techniques be used to ~~do efficient inference on~~ Anglican programs?

help MC inference algorithms for

Yes. In compile time,

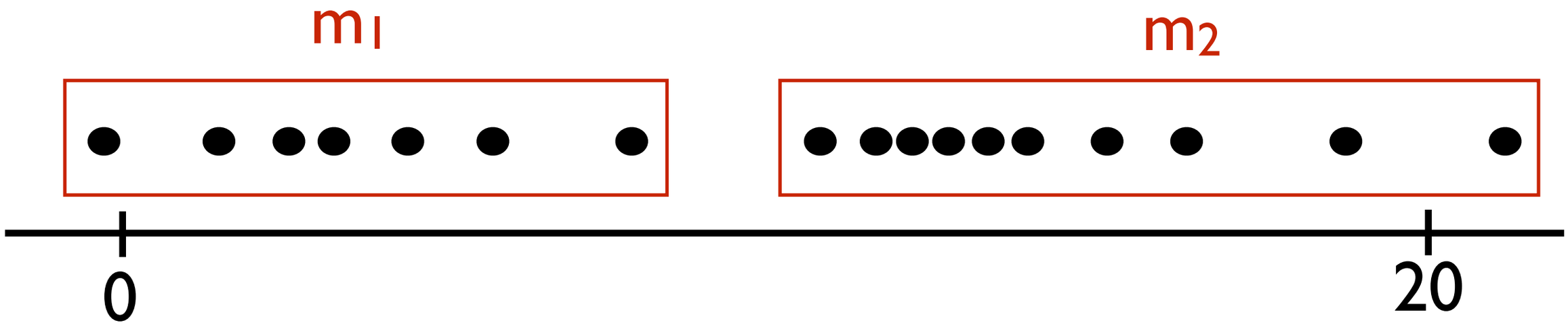
1. integrate some random variables;
2. compute posterior of some random variables wrt. some data.

Problem: find two clusters



Problem: find two clusters

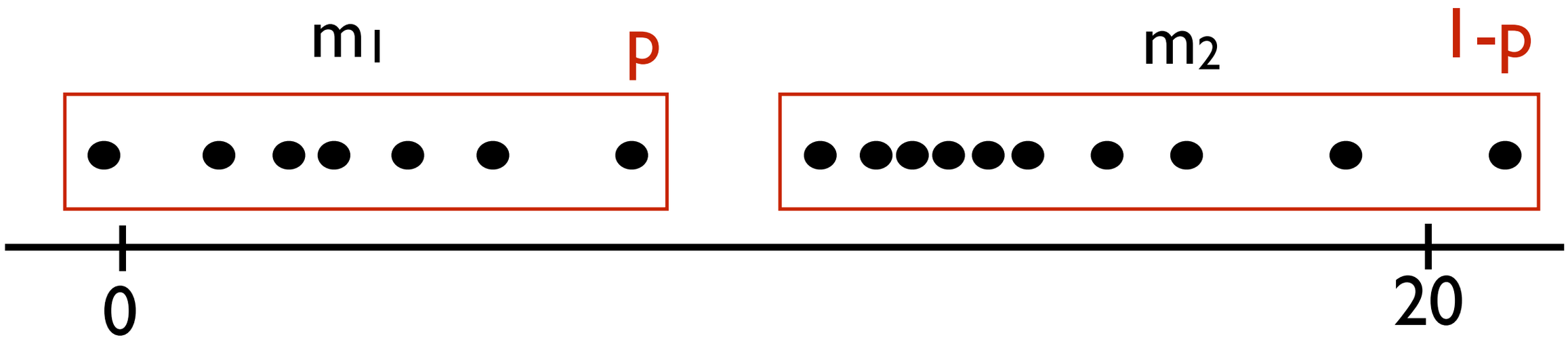
Find means of two clusters: m_1 and m_2



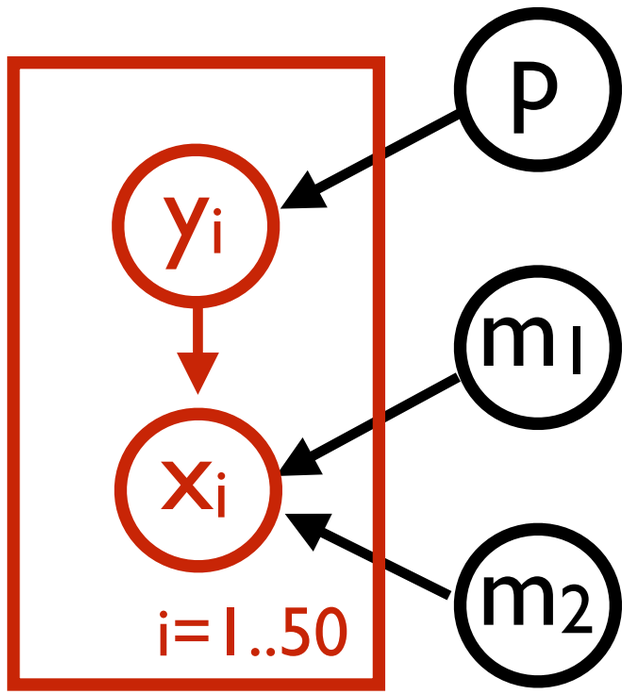
Problem: find two clusters

Find means of two clusters: m_1 and m_2

Find the probability p of choosing the first cluster.




```
p ~ beta(1,1)
m1 ~ normal(10,10) m2 ~ normal(10,10)
```

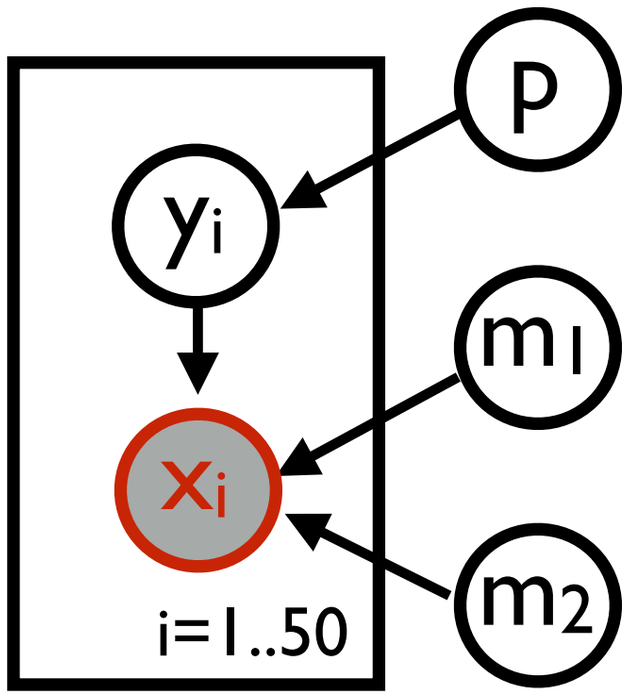


$p \sim beta(1,1)$

$m_1 \sim normal(10,10)$ $m_2 \sim normal(10,10)$

$y_i \sim flip(p)$ $x_i \sim normal(m_{(y_i+1)},10)$

for i = 1..50

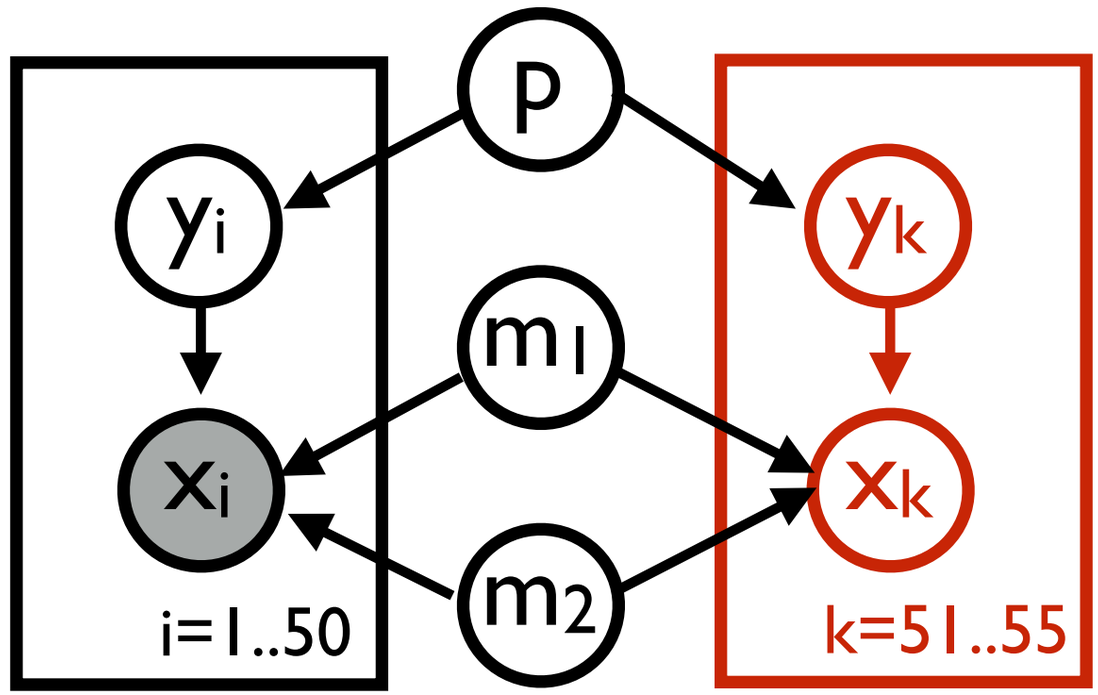


$p \sim \text{beta}(1,1)$

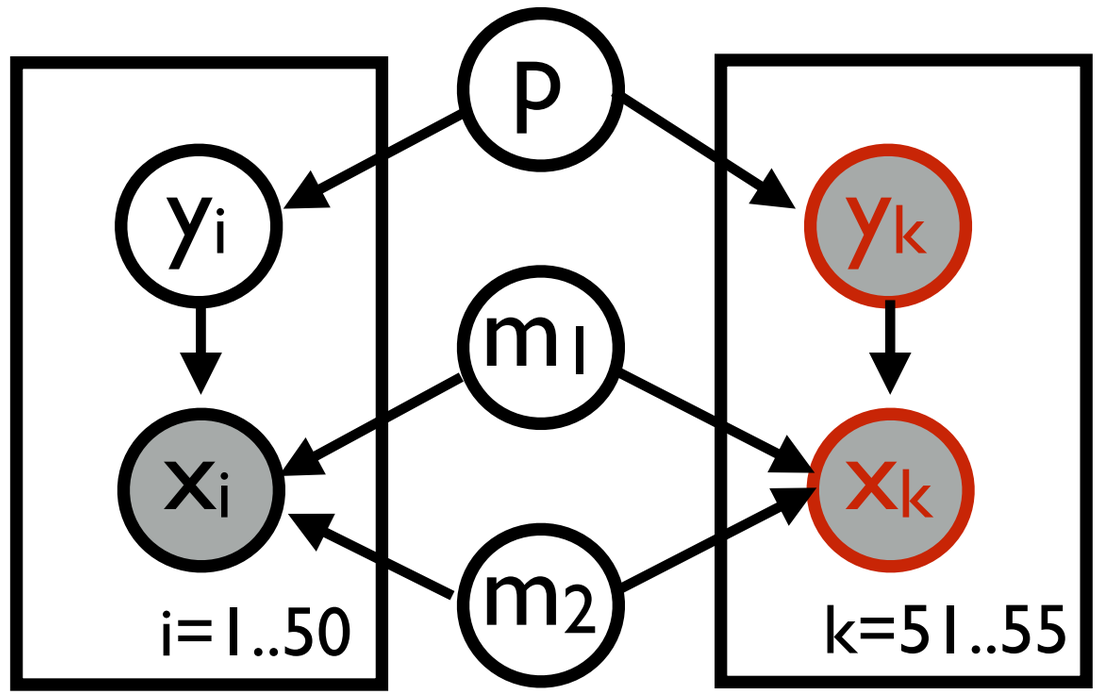
$m_1 \sim \text{normal}(10,10)$ $m_2 \sim \text{normal}(10,10)$

$y_i \sim \text{flip}(p)$ $x_i \sim \text{normal}(m_{(1-y_i)},10)$

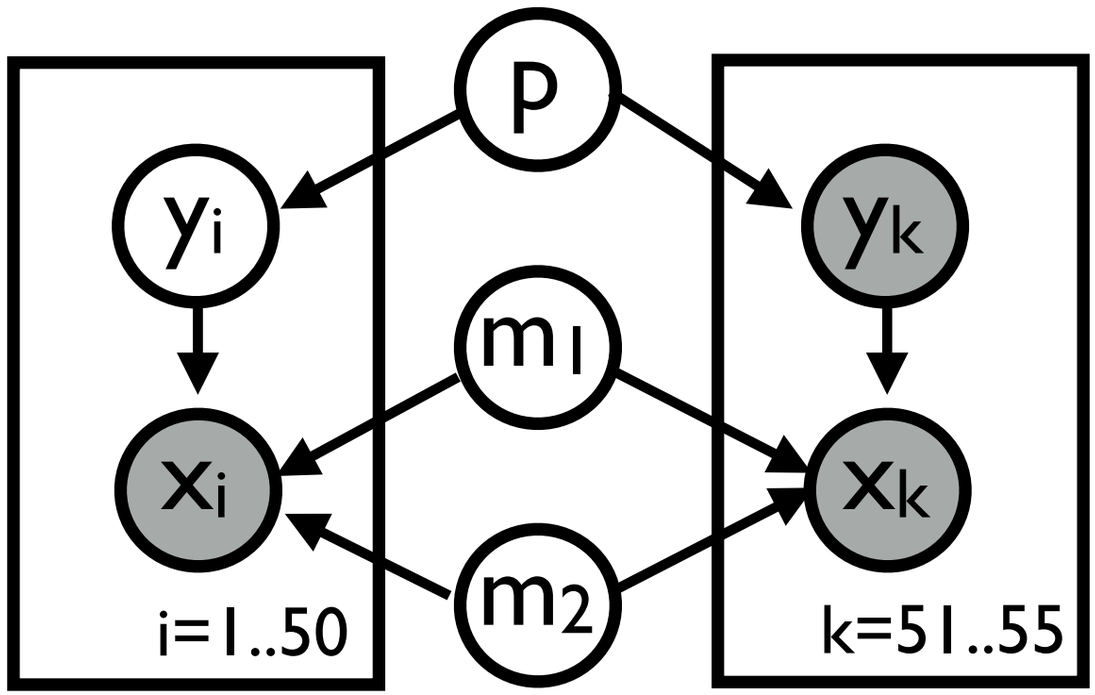
for i = 1..50




```
p ~ beta(1,1)
m1 ~ normal(10,10)  m2 ~ normal(10,10)
yi ~ flip(p)        xi ~ normal(m(1-yi),10)
yk ~ flip(p)        xk ~ normal(m(1-yk),10)
                  for i = 1..50, k = 51..55
```



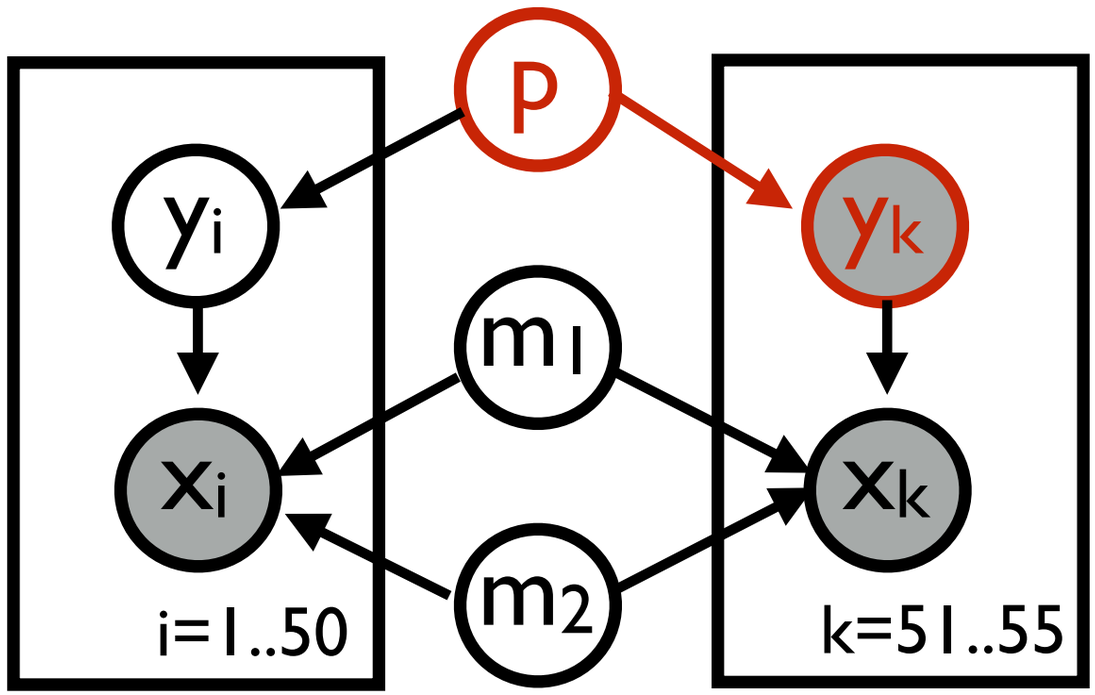

```
p ~ beta(1,1)
m1 ~ normal(10,10)  m2 ~ normal(10,10)
yi ~ flip(p)        xi ~ normal(m(1-yi),10)
yk ~ flip(p)        xk ~ normal(m(1-yk),10)
                  for i = 1..50, k = 51..55
```




```
p ~ beta(1,1)
m1 ~ normal(10,10)  m2 ~ normal(10,10)
yi ~ flip(p)        xi ~ normal(m(1-yi),10)
yk ~ flip(p)        xk ~ normal(m(1-yk),10)
                  for i = 1..50, k = 51..55
```

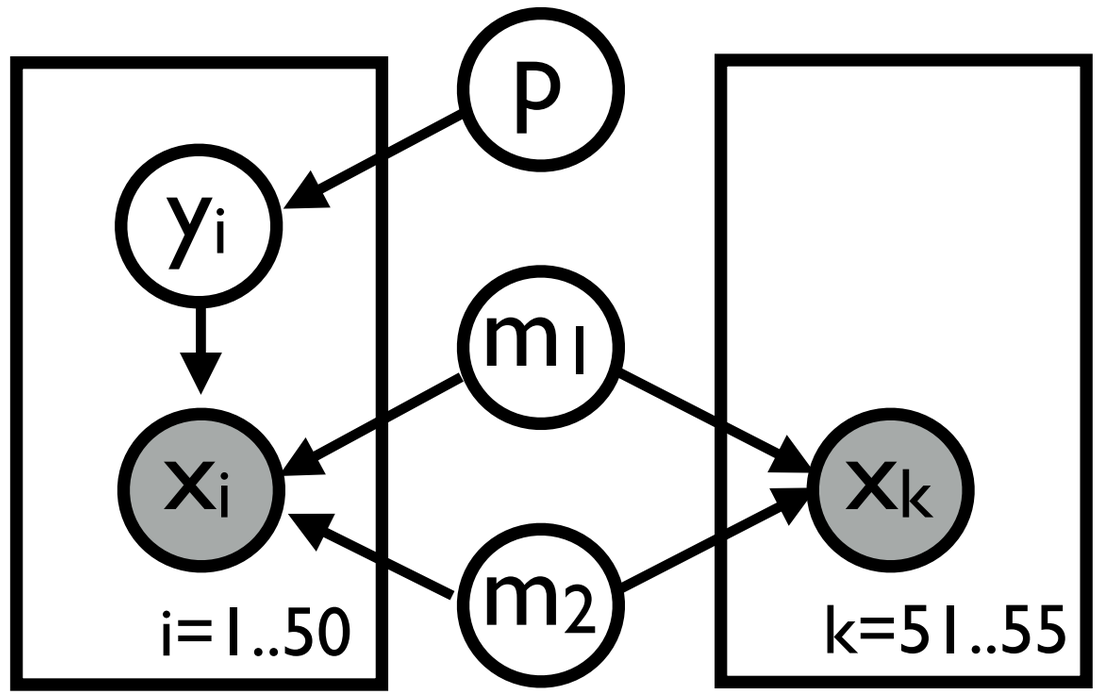
Q: posterior of p?

Post. wrt. y_k.




```
p ~ beta(1,1)
m1 ~ normal(10,10)  m2 ~ normal(10,10)
yi ~ flip(p)        xi ~ normal(m(1-yi),10)
yk ~ flip(p)        xk ~ normal(m(1-yk),10)
                  for i = 1..50, k = 51..55

Q: posterior of p?
```

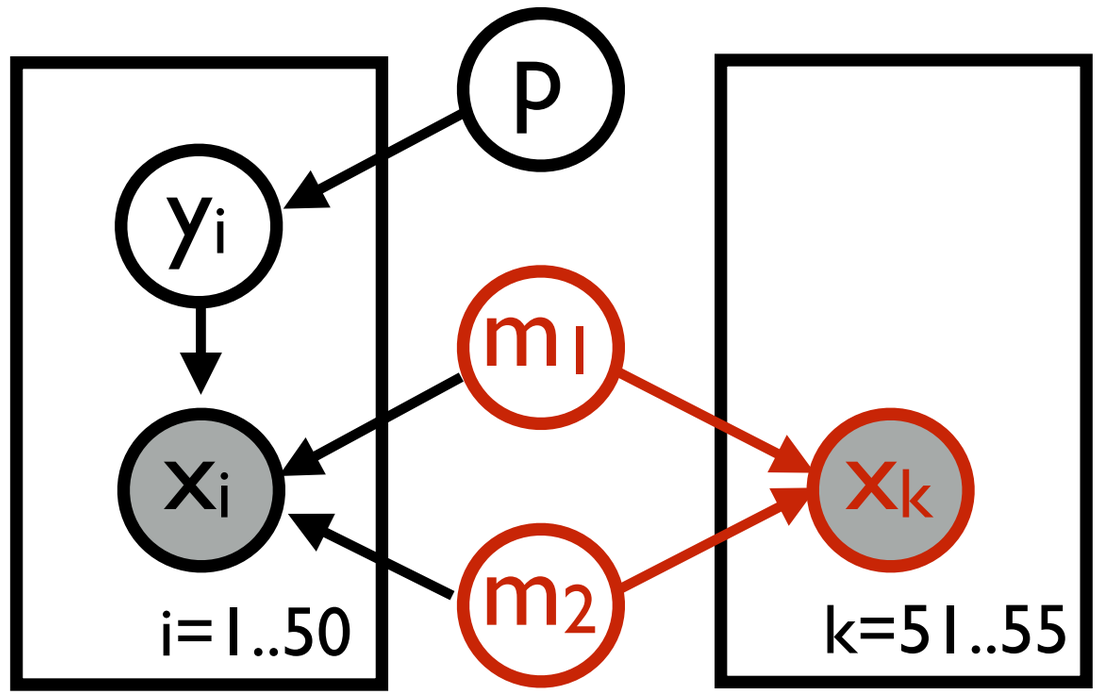
Post. wrt. y_k.




```
p ~ beta(…, …)
m1 ~ normal(10,10) m2 ~ normal(10,10)
yi ~ flip(p)    xi ~ normal(m(1-yi),10)
                xk ~ normal(m(1-yk),10)
              for i = 1..50, k = 51..55

Q: posterior of p?
```

Post. wrt. y_k.

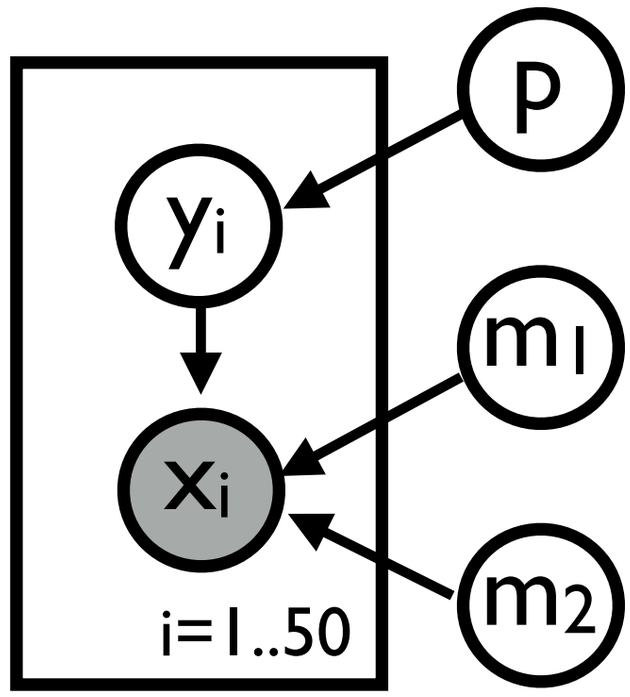
Post. wrt. x_k.




```
p ~ beta(…, …)
m1 ~ normal(10,10) m2 ~ normal(10,10)
yi ~ flip(p)    xi ~ normal(m(1-yi),10)
                xk ~ normal(m(1-yk),10)
              for i = 1..50, k = 51..55

Q: posterior of p?
```

Post. wrt. y_k.

Post. wrt. x_k.

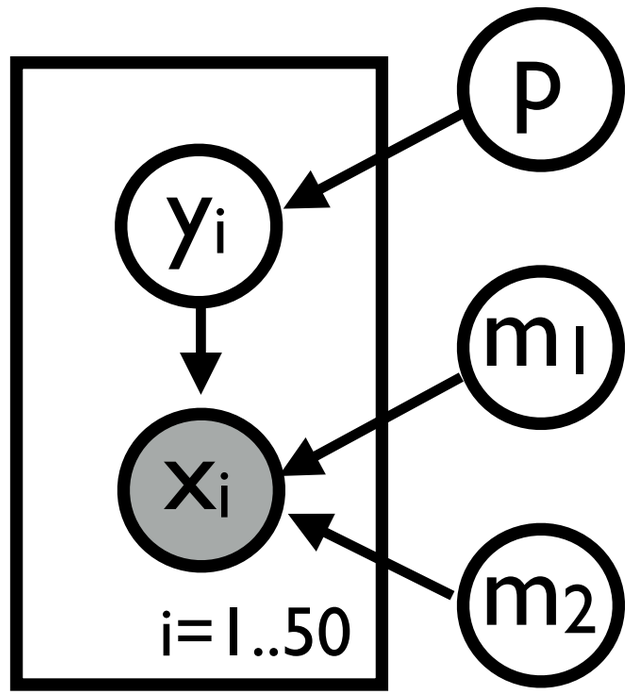



```
p ~ beta(…, …)
m1 ~ normal(…, …)   m2 ~ normal(…, …)
yi ~ flip(p)        xi ~ normal(m(1-yi),10)

                for i = 1..50

Q: posterior of p?
```

Post. wrt. y_k.

Post. wrt. x_k.

Integrate y_i.

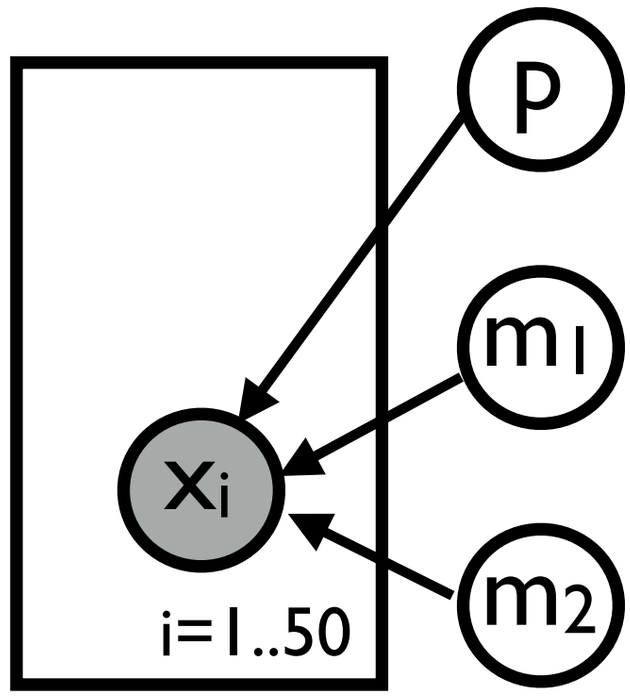



```
p ~ beta(…, …)
m1 ~ normal(…, …)   m2 ~ normal(…, …)
yi ~ flip(p)        xi ~ normal(m(1-yi),10)

                for i = 1..50

Q: posterior of p?
```

Post. wrt. y_k.

Post. wrt. x_k.

Integrate y_i.



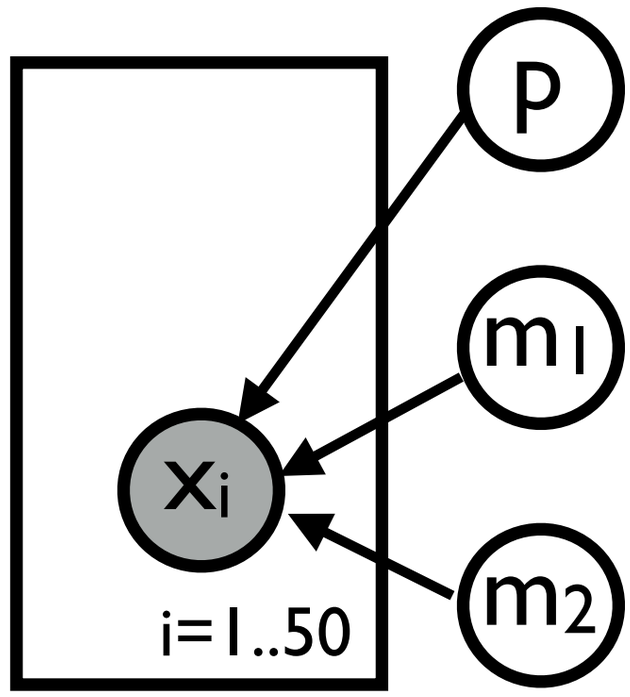

```
p ~ beta(…, …)
m1 ~ normal(…, …)   m2 ~ normal(…, …)
xi ~ p*normal(m1,10) + (1-p)*normal(m2,10)

                for i = 1..50

Q: posterior of p?
```

Post. wrt. y_k.

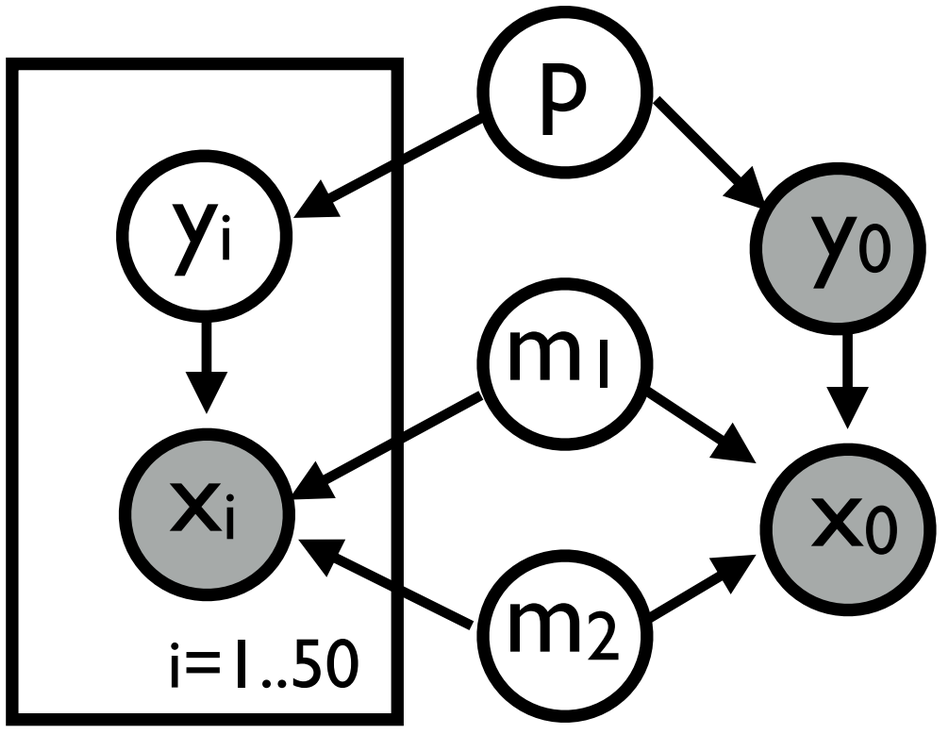
Post. wrt. x_k.

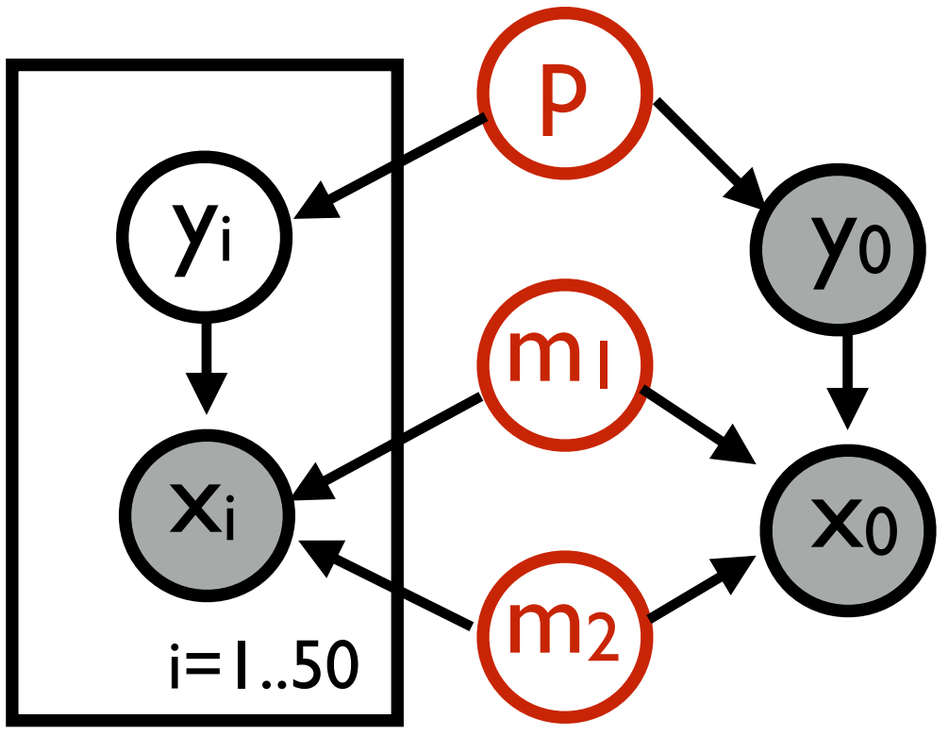
Integrate y_i.

Run MC.

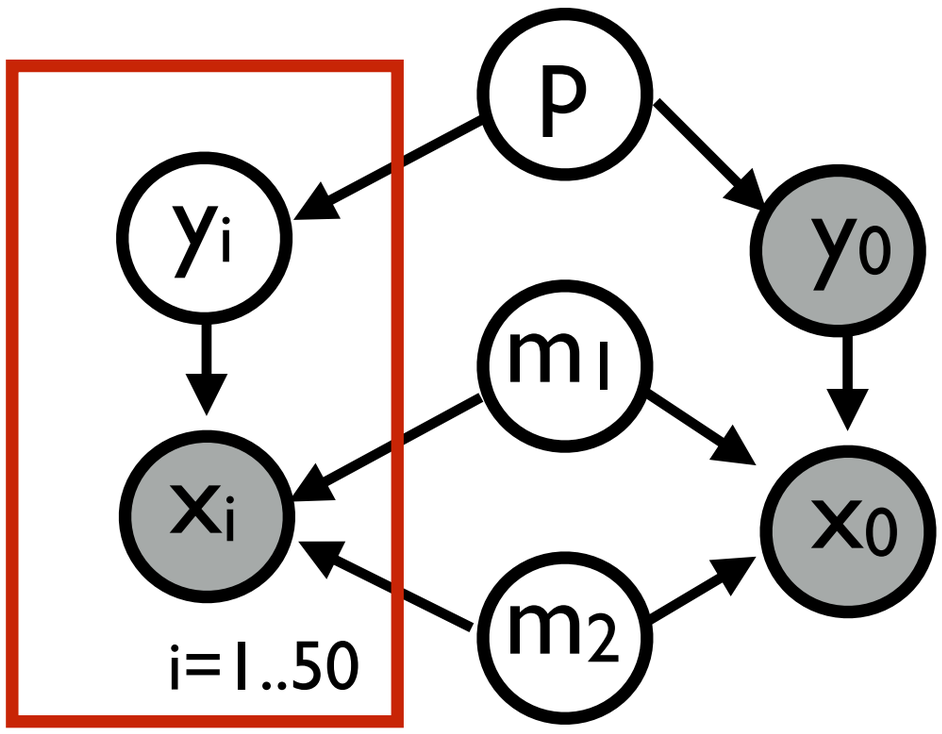



```
p ~ beta(…, …)
m1 ~ normal(…, …)   m2 ~ normal(…, …)
xi ~ p*normal(m1,10) + (1-p)*normal(m2,10)

                for i = 1..50

Q: posterior of p?
```

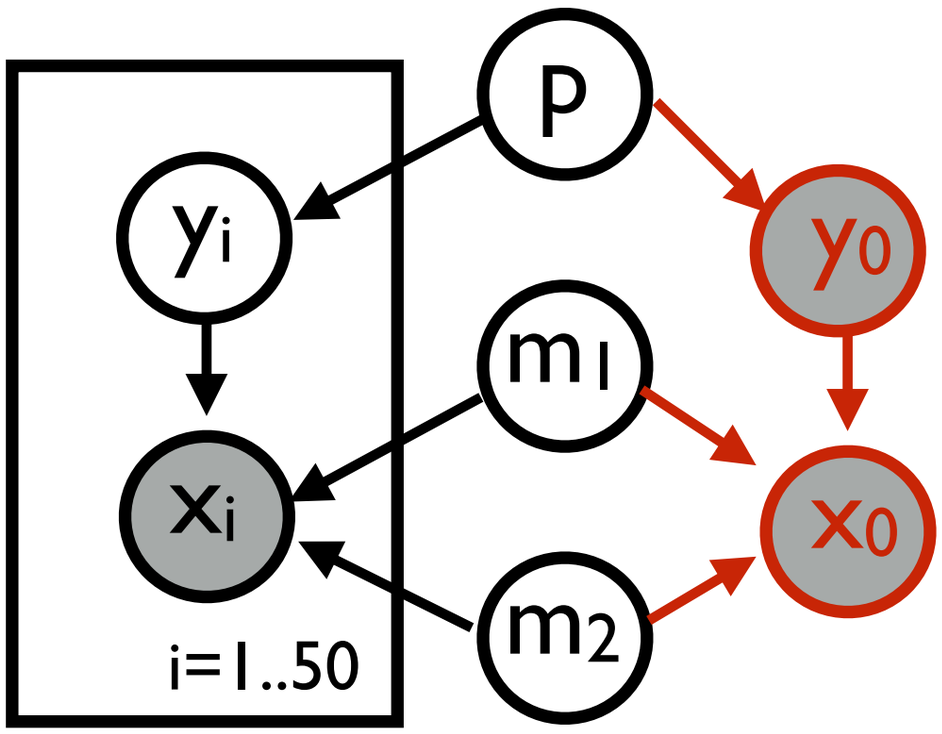

P
yi
y0
ml
xi
x0
m2
i=1..50



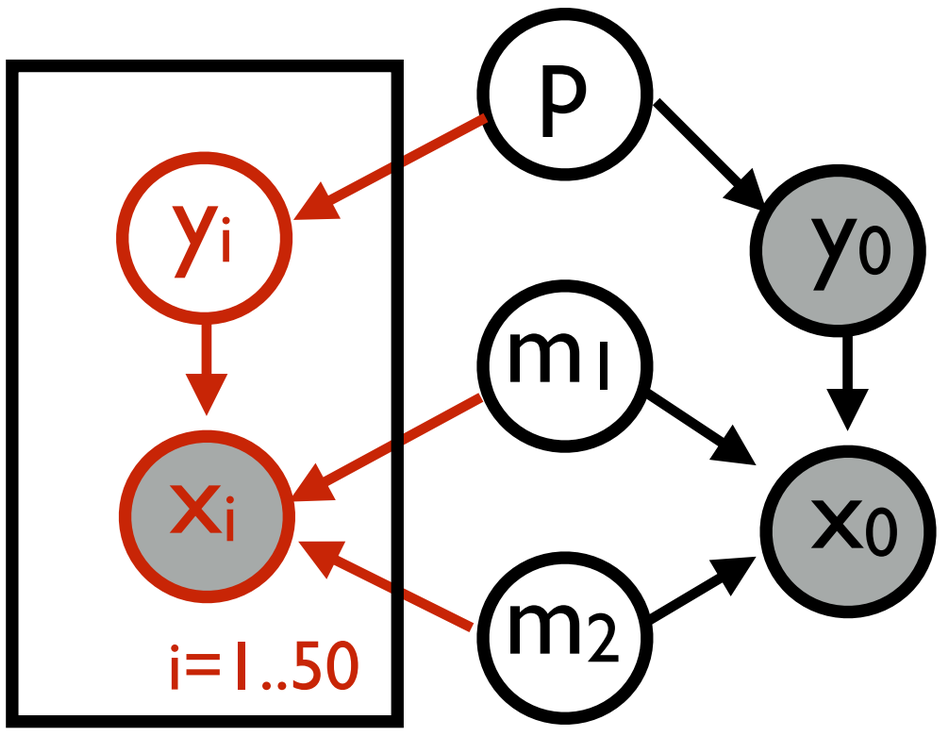

```
(let [p  (sample (beta 1. 1.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))
```



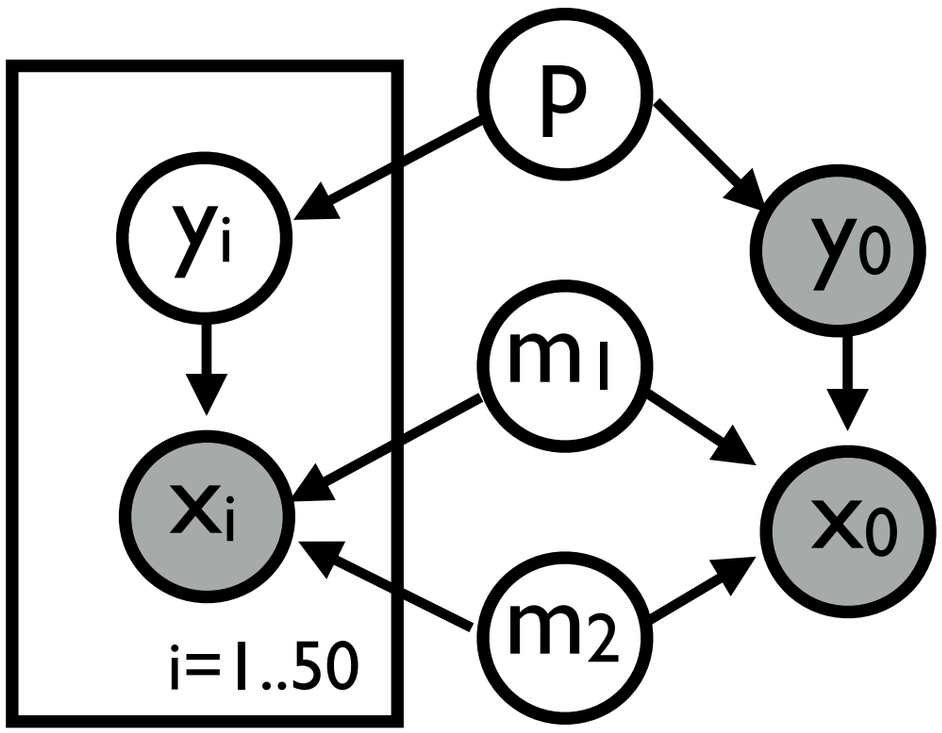

```
(let [p  (sample (beta 1. 1.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))) ]
```




```
(let [p  (sample (beta 1. 1.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))) ]

(observe (flip p) false)
(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
```



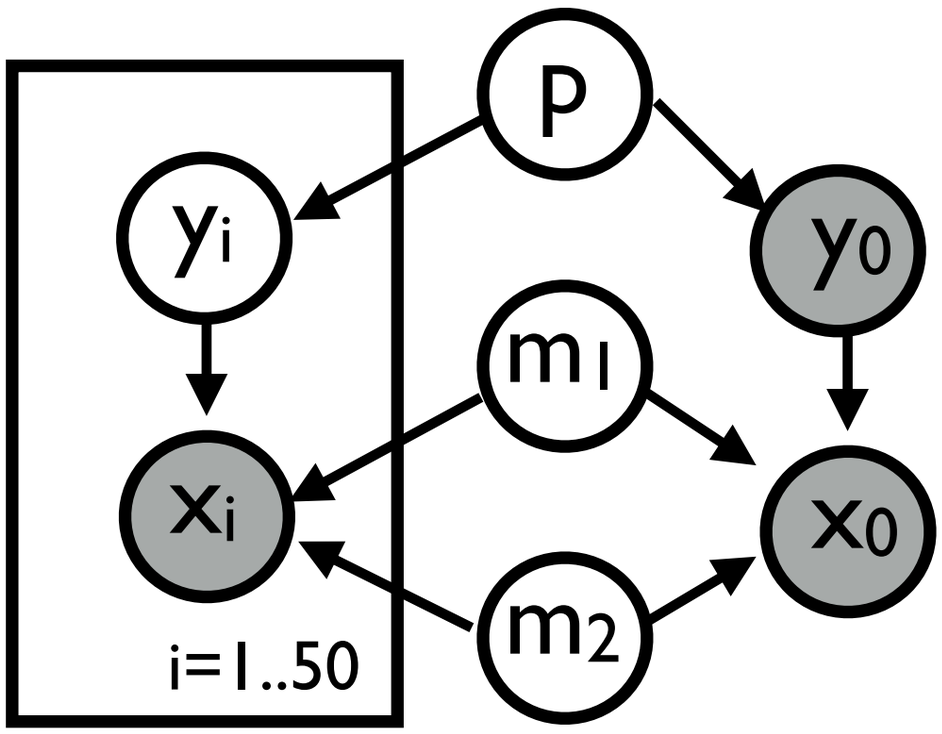

```
(let [p  (sample (beta 1. 1.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))) ]

(observe (flip p) false)
(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.15)
…
(observe (f p m1 m2) 1.88)
```




```
(let [p  (sample (beta 1. 1.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))) ]

(observe (flip p) false)
(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

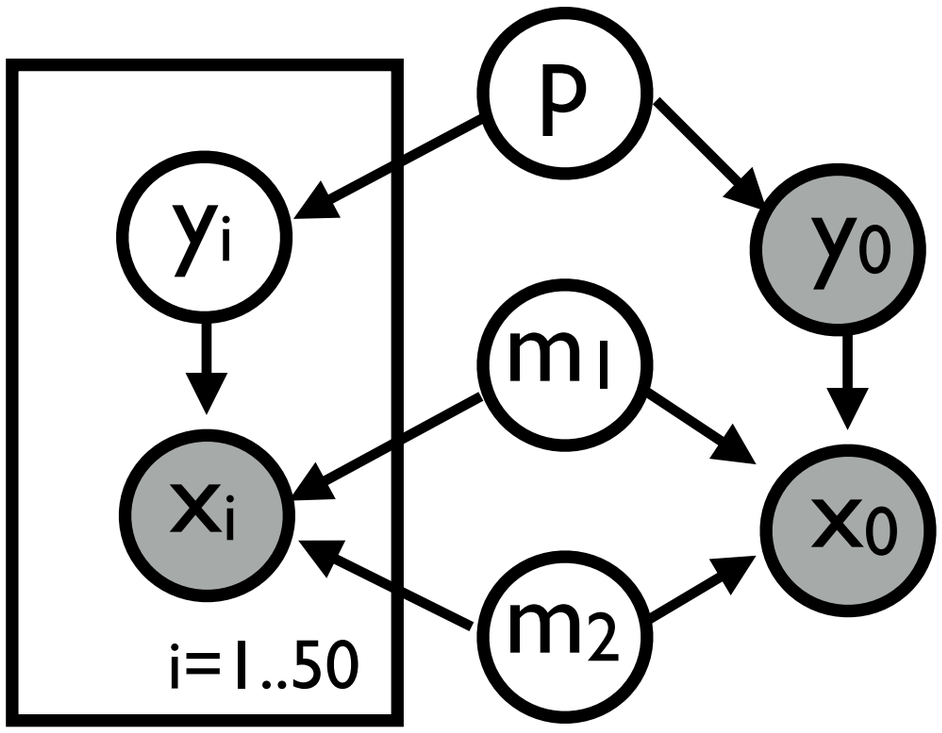
Optimising a model via program transformation

- Compute posterior locally by moving observe backwards.
- Integrate random variables (p in our case) by ERPification.



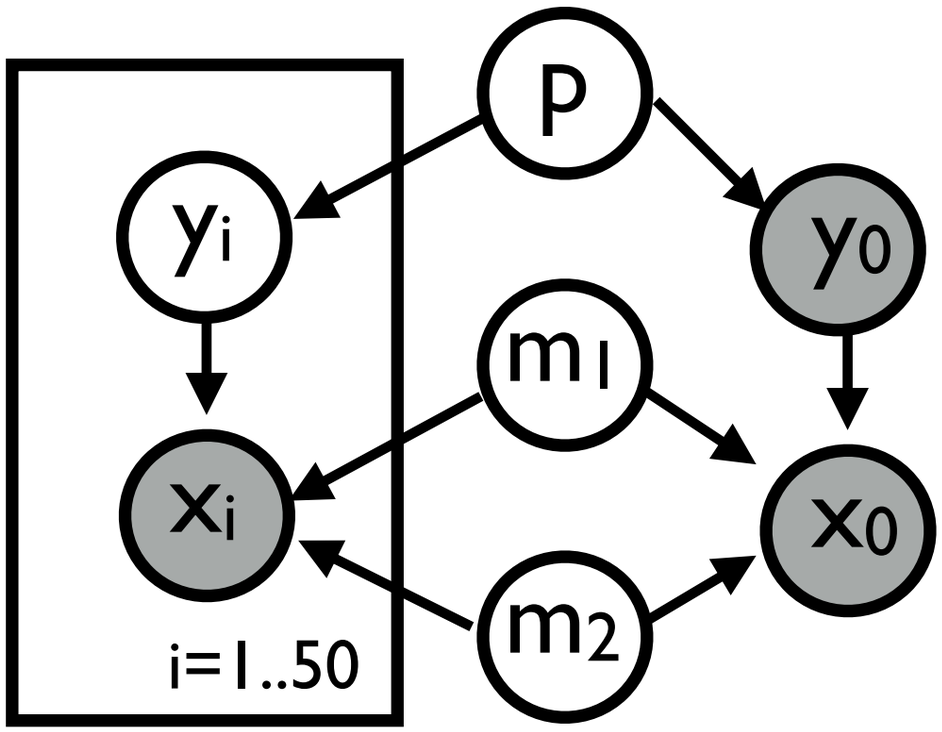

```
(let [p  (sample (beta 1. 1.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]

(observe (flip p) false)
(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.




```
(let [p  (sample (beta 1. 1.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]

(observe (flip p) false)
(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.

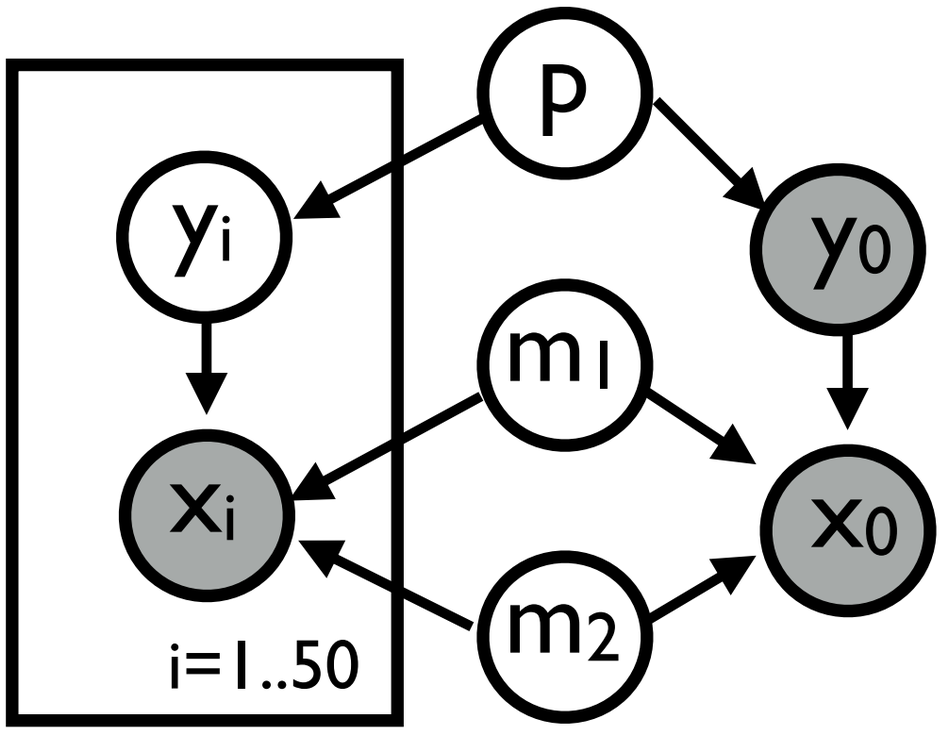


p not defined.

Hence, independent.

```
(let [p  (sample (beta 1. 1.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]

(observe (flip p) false)
(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

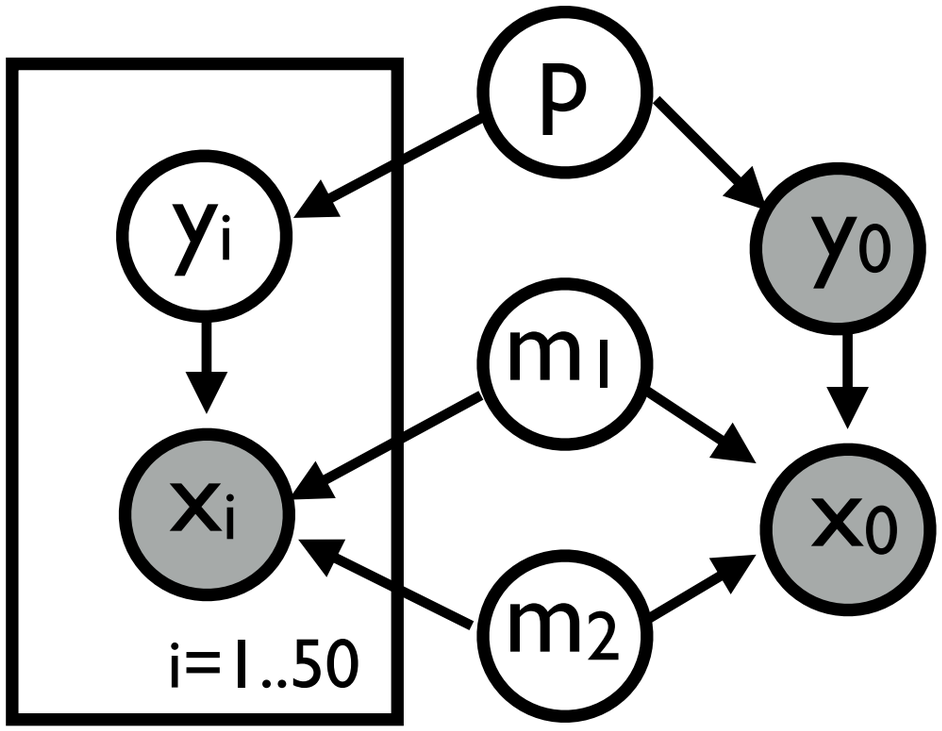
1. Compute posterior.



p not defined.
Hence, independent.

```
(let [p  (sample (beta 1. 1.))
      _  (observe (flip p) false)
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.



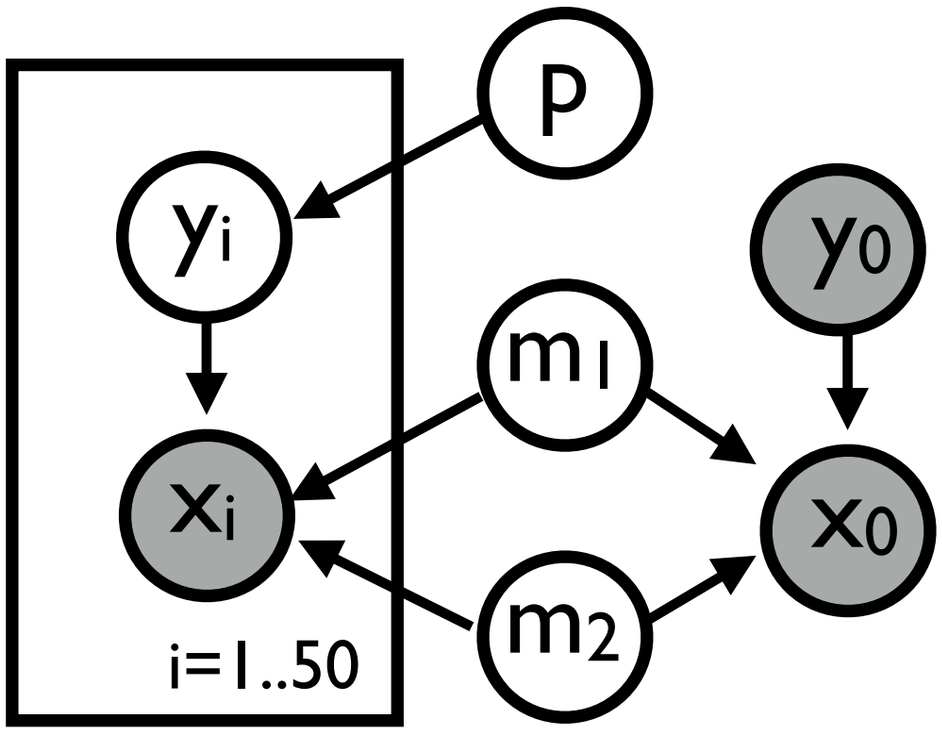
p not defined.

Hence, independent.

Beta-Flip conjugacy.

```
(let [p  (sample (beta 1. 1.))
      _  (observe (flip p) false)
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.



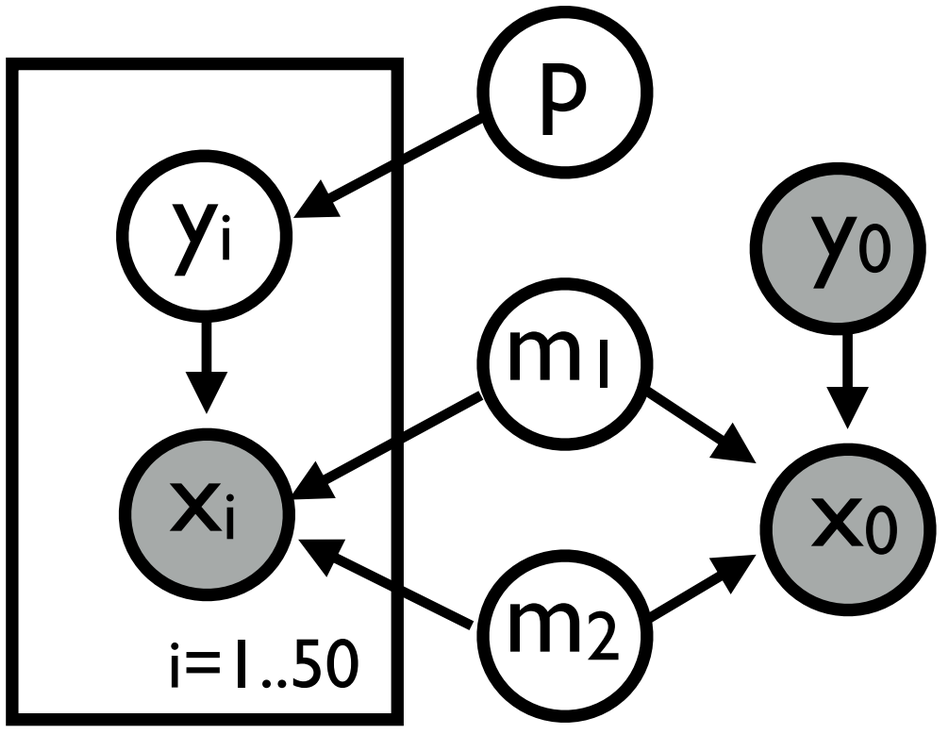
p not defined.

Hence, independent.

Beta-Flip conjugacy.

```
(let [_  (observe (flip 0.5) false)
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.




```
(let [_  (observe (flip 0.5) false)
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

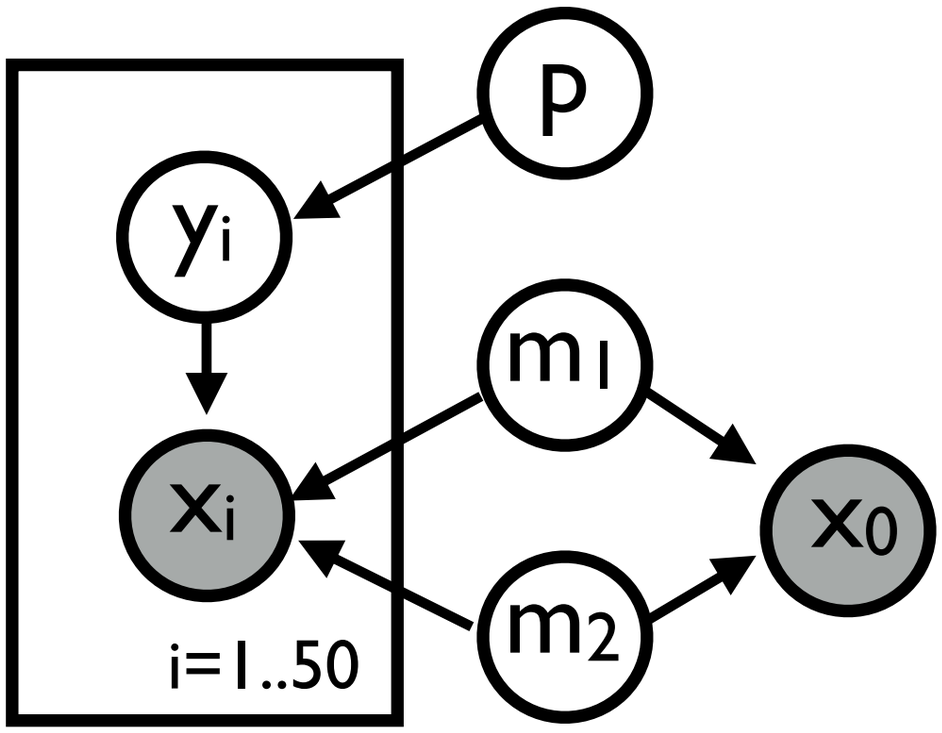
p not defined.

Hence, independent.

Beta-Flip conjugacy.

Observe w/o free vars.

1. Compute posterior.




```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

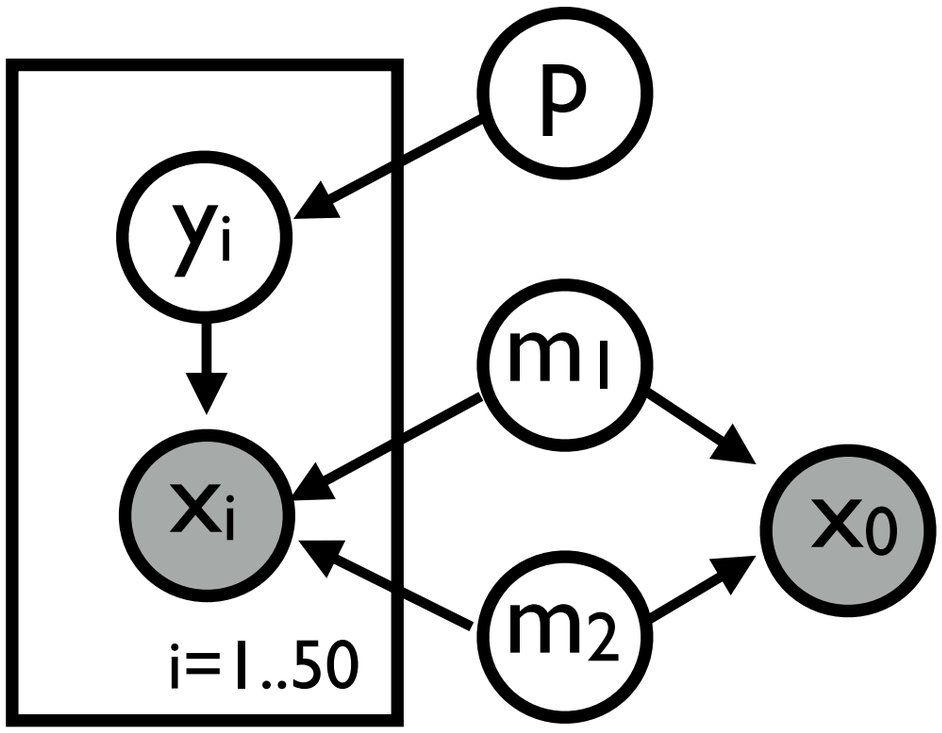
p not defined.

Hence, independent.

Beta-Flip conjugacy.

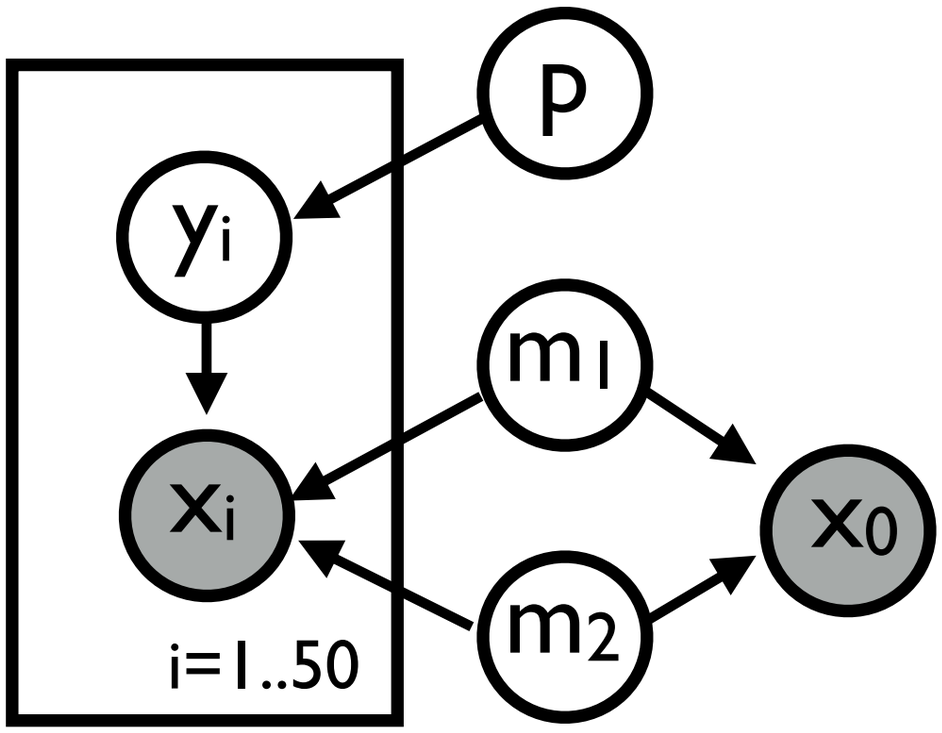
Observe w/o free vars.

1. Compute posterior.




```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

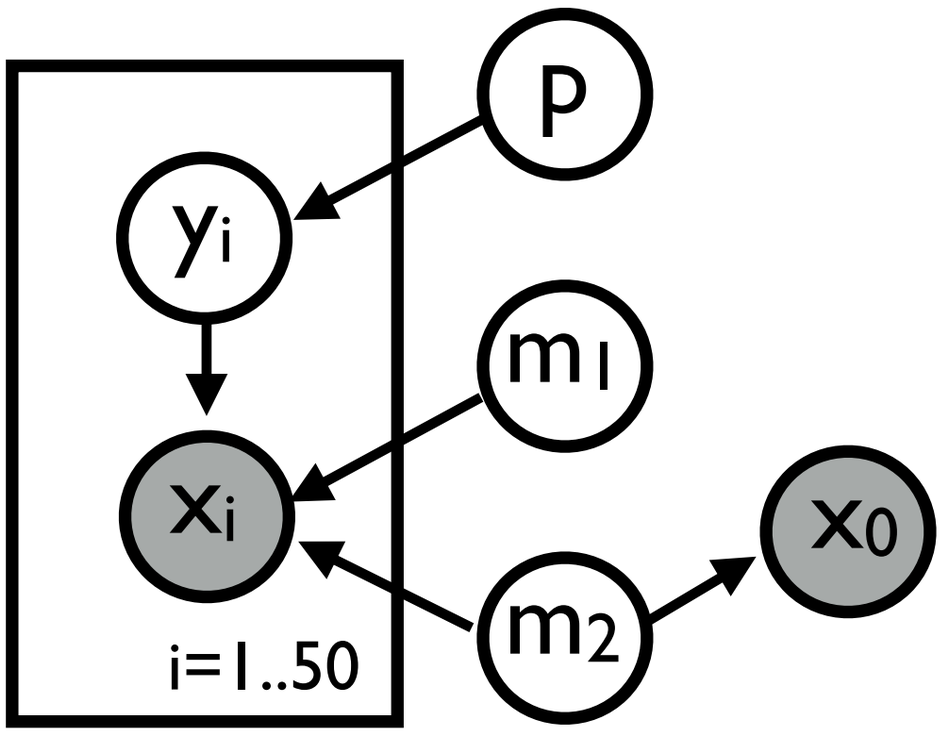
1. Compute posterior.



Partial evaluation.

```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.



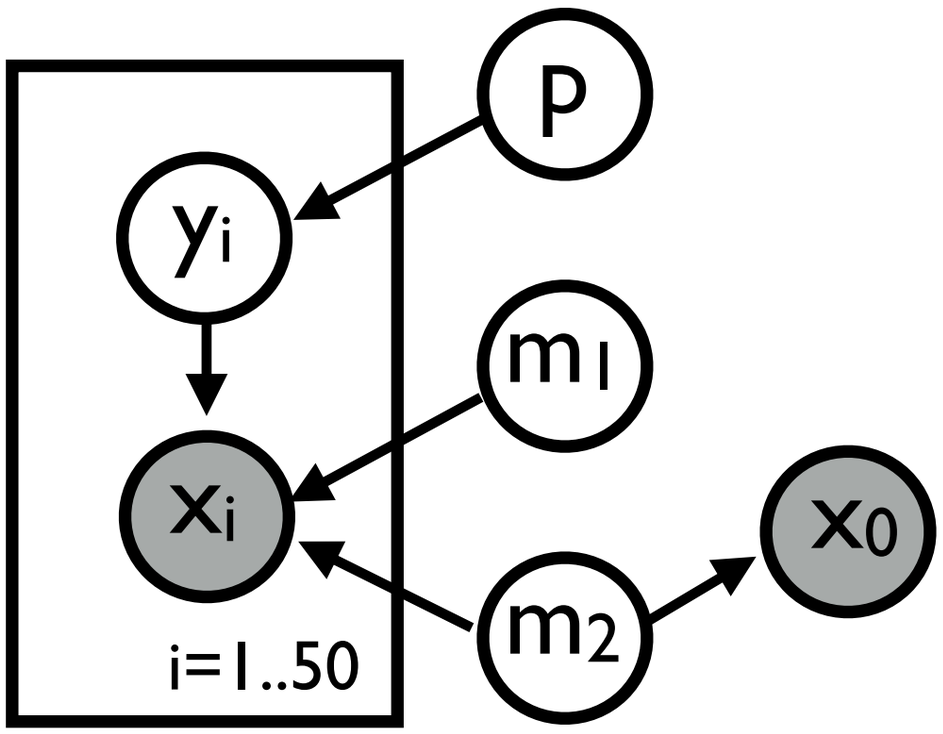

```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]
```

```
(observe (normal m2 10.) 102.3)
```

Partial evaluation.

```
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.




```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]
```

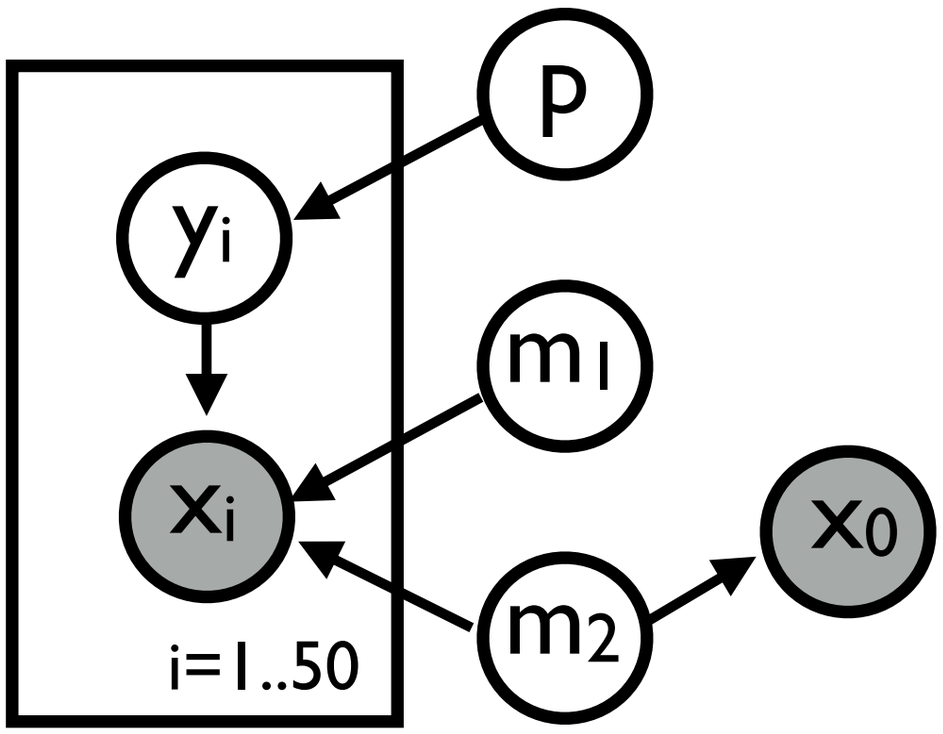
```
(observe (normal m2 10.) 102.3)
```

Partial evaluation.

m2 and f are different.

```
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.



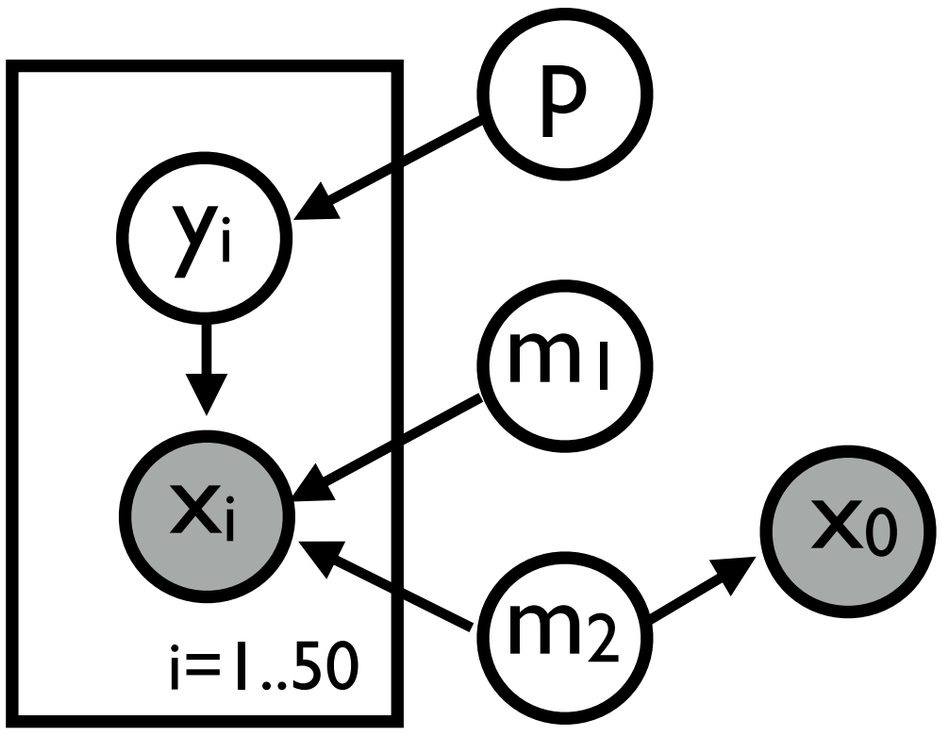

```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))
      _  (observe (normal m2 10.) 102.3)

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]
```

Partial evaluation.

m2 and f are different.

```
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.




```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      m2 (sample (normal 10. 10.))
      _  (observe (normal m2 10.) 102.3)

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]
```

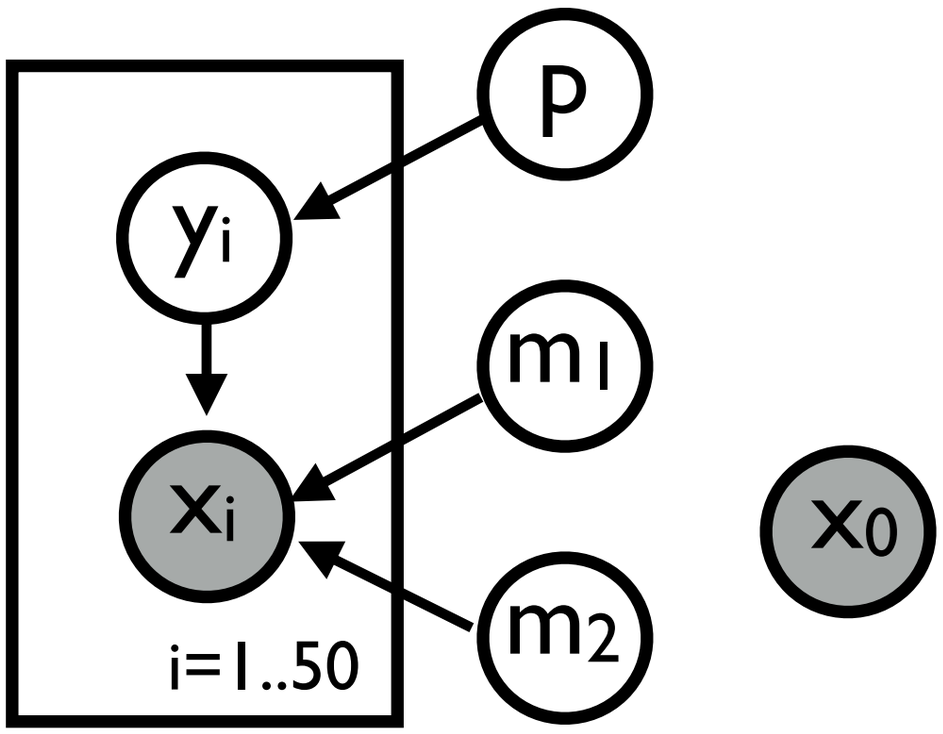
Partial evaluation.

m2 and f are different.

Normal-Normal conjugacy.

```
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.




```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      _  (observe (normal …) 102.3)
      m2 (sample (normal …))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]
```

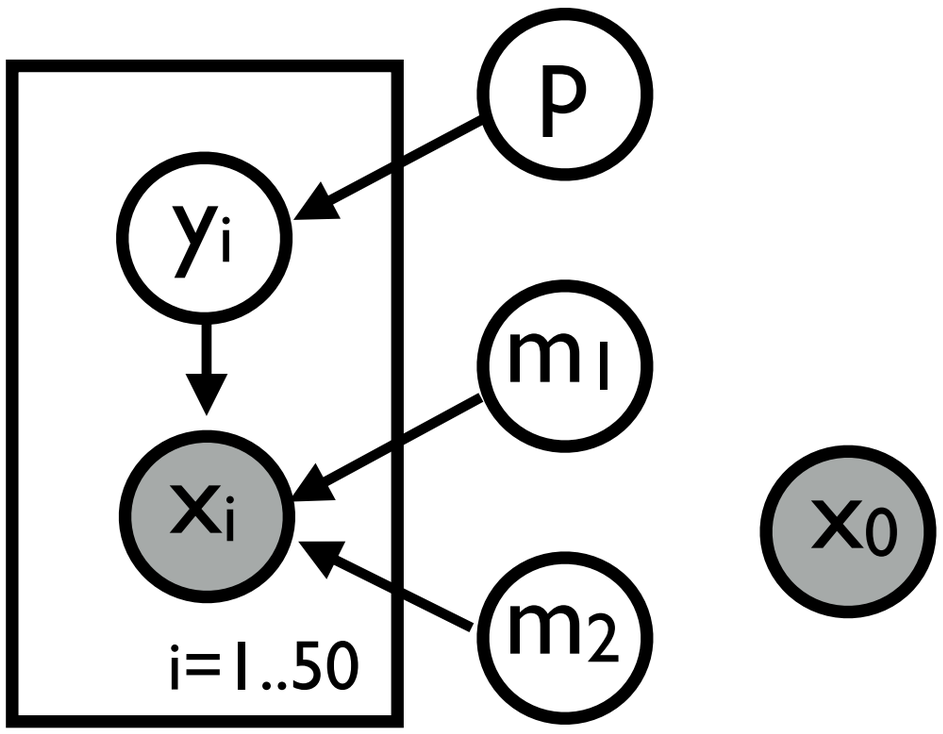
Partial evaluation.

m2 and f are different.

Normal-Normal conjugacy.

```
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.




```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))
      _  (observe (normal …) 102.3)
      m2 (sample (normal …))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]
```

Partial evaluation.

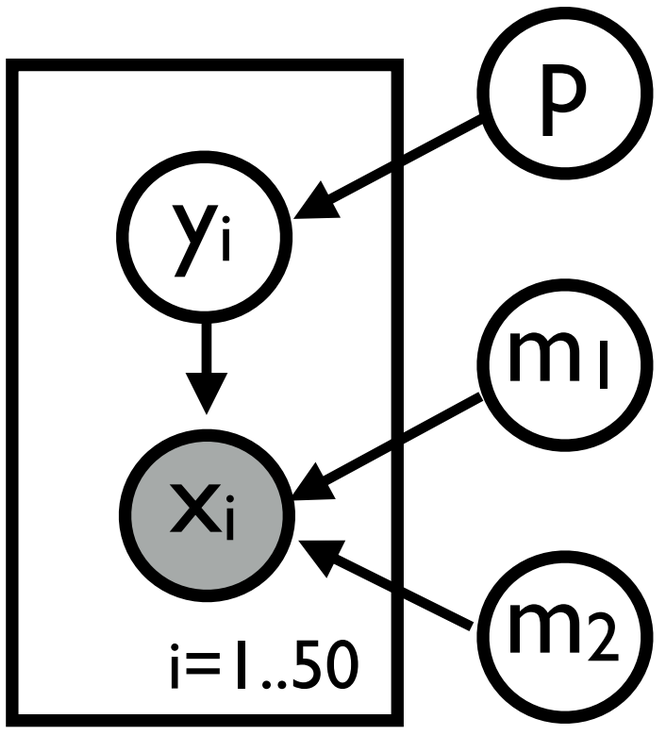
m2 and f are different.

Normal-Normal conjugacy.

Observe w/o free vars.

```
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.




```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))

      m2 (sample (normal …))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]
```

Partial evaluation.

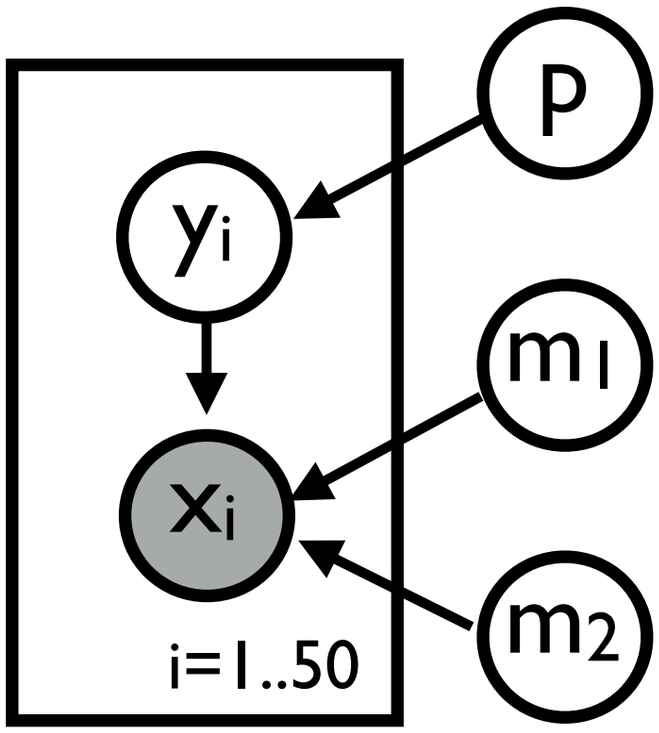
m2 and f are different.

Normal-Normal conjugacy.

Observe w/o free vars.

```
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

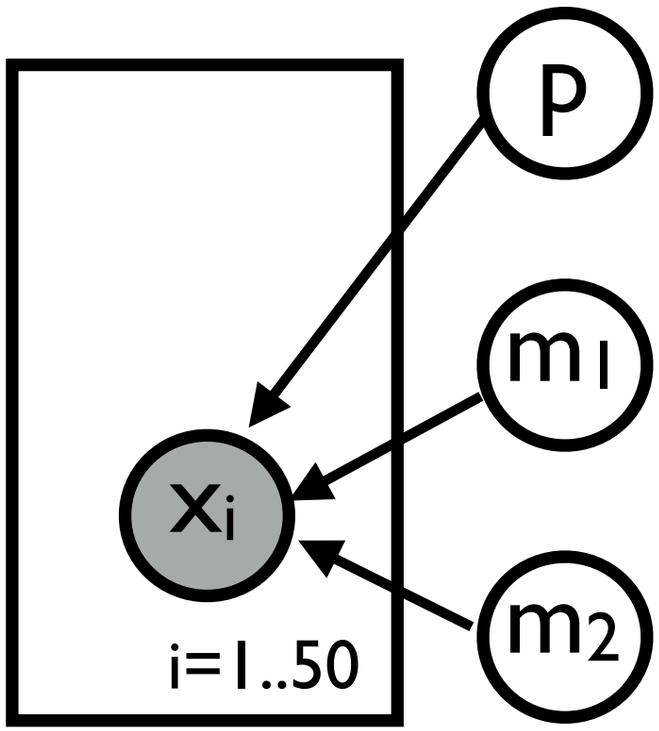
1. Compute posterior.
2. Integrate.




```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))

      m2 (sample (normal …))

      f  (erpify-dist
      (fn [P M1 M2]
        (let [y (sample (flip P))]
          (if y (normal M1 10.)
                (normal M2 10.)))))]
```

```
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

1. Compute posterior.
2. Integrate.




```
(let [
      p  (sample (beta 1. 2.))
      m1 (sample (normal 10. 10.))

      m2 (sample (normal …))

      f
      (fn [P M1 M2]
        (mixture-dist
          [P         (normal M1 10.)]
          [(- 1. P) (normal M2 10.)]))]
```

```
(observe (f p m1 m2) 11.55)
…
(observe (f p m1 m2) 1.88)
(predict :p p))
```

Distribution object ds

- Represents a probability distribution that has density.
- Has two methods:

`(sample ds)` for sampling a value

`(observe ds v)` for computing the prob. of v

Elementary random procedure (ERP)

- Procedure that creates a dist. object.
- Examples:

```
flip, normal, gamma, …
```

- Non-example:

```
(fn [P] (not (sample (flip P))))
```

Erpification

- Generates an ERP from a procedure.
- Finds a formula for probability.
- Results expressed in terms of existing ERPs.

```
(fn [P]
  (not (sample (flip P))))
```

```
(fn [P]
  (flip (- 1 P)))
```

Erpification

Suppose `erpify(f) = f-erp`.

- Then, `(f p) = (sample (f-erp p))`.
- Can: `(observe (f-erp p) false)`.
- Cannot: `(observe (f p) false)`.

```
(fn [P]
  (not (sample (flip P))))
```

```
(fn [P]
  (flip (- 1 P)))
```

1. Symbolically enumerate all possible outputs.
2. Express the outputs using existing ERPs.

1. Symbolically enumerate all possible outputs.
2. Express the outputs using existing ERPs.

```
(fn [P]
  (not
    (sample (flip P))))
```

1. Symbolically enumerate all possible outputs.
2. Express the outputs using existing ERPs.

```
(fn [P]
  (not
    (sample (flip P))))
```

```
{ (false, P,   []),
  (true,  1-P, [])
}
```

1. Symbolically enumerate all possible outputs.
2. Express the outputs using existing ERPs.

```
(fn [P]
  (not
    (sample (flip P))))
```

```
{ (false, P,   []),
  (true,  1-P, [])
}
```

```
(flip (- 1 P))
```

1. Symbolically enumerate all possible outputs.
2. Express the outputs using existing ERPs.

```
(fn [P]
  (not
    (sample (flip P))))
```

```
{ (false, P,   []),
  (true,  1-P, [])
}
```

```
(flip (- 1 P))
```

```
(fn [P M1 M2]
  (if (sample (flip P))
    (sample (normal M1 1))
    (let [R2 (sample (normal M2 1))]
      (sample (normal (+ R2 2) 1)))))
```

1. Symbolically enumerate all possible outputs.

2. Express the outputs using existing ERPs.

```
(fn [P]
  (not
    (sample (flip P))))
```

```
{ (false, P,   []),
  (true,  1-P, [])
}
```

```
(flip (- 1 P))
```

```
(fn [P M1 M2]
  (if (sample (flip P))
    (sample (normal M1 1))
    (let [R2 (sample (normal M2 1))]
      (sample (normal (+ R2 2) 1)))))
```

```
{ (R1, P,   [R1:(normal M1 1)]),
  (R3, 1-P, [R2:(normal M2 1),
             R3:(normal (* R2 2) 1)])
}
```

1. Symbolically enumerate all possible outputs.
2. Express the outputs using existing ERPs.

```
(fn [P]
  (not
    (sample (flip P))))
```

```
{ (false, P,   []),
  (true,  1-P, [])
}
```

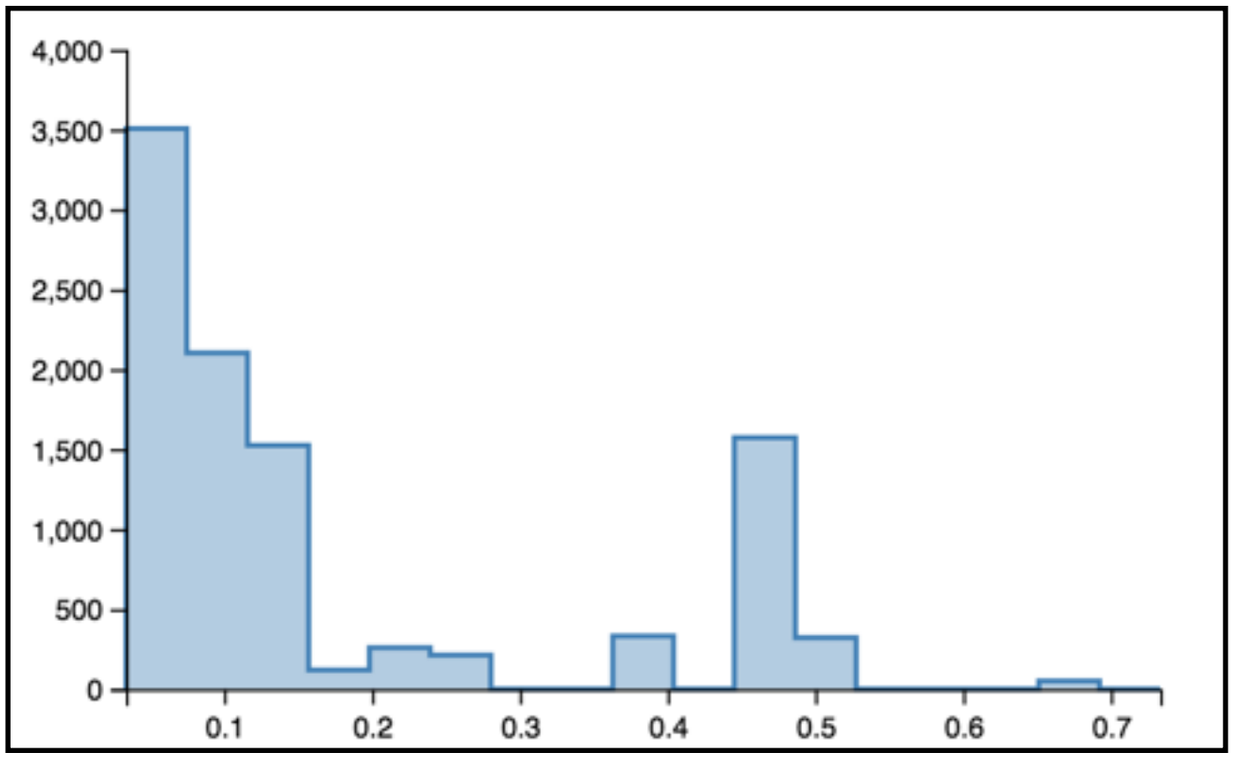
```
(flip (- 1 P))
```

```
(fn [P M1 M2]
  (if (sample (flip P))
    (sample (normal M1 1))
    (let [R2 (sample (normal M2 1))]
      (sample (normal (+ R2 2) 1)))))
```

```
{ (R1, P,   [R1:(normal M1 1)]),
  (R3, 1-P, [R2:(normal M2 1),
             R3:(normal (* R2 2) 1)])
}
```

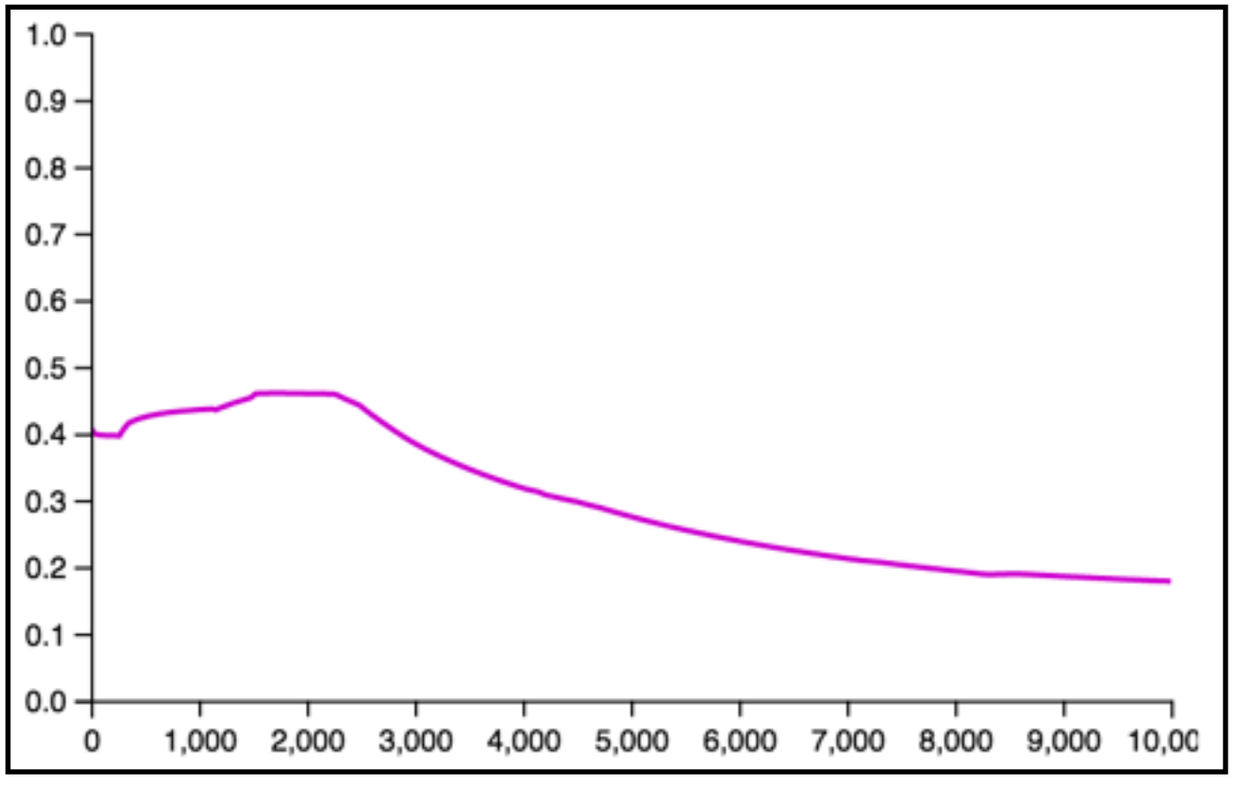
```
(mixture-dist
  [[P       (normal M1 1)]
   [(- 1 P) (normal (* M2 2)
                    (sqrt 5))]])
```

Gibbs-like algorithm (LMH)
applied to original program



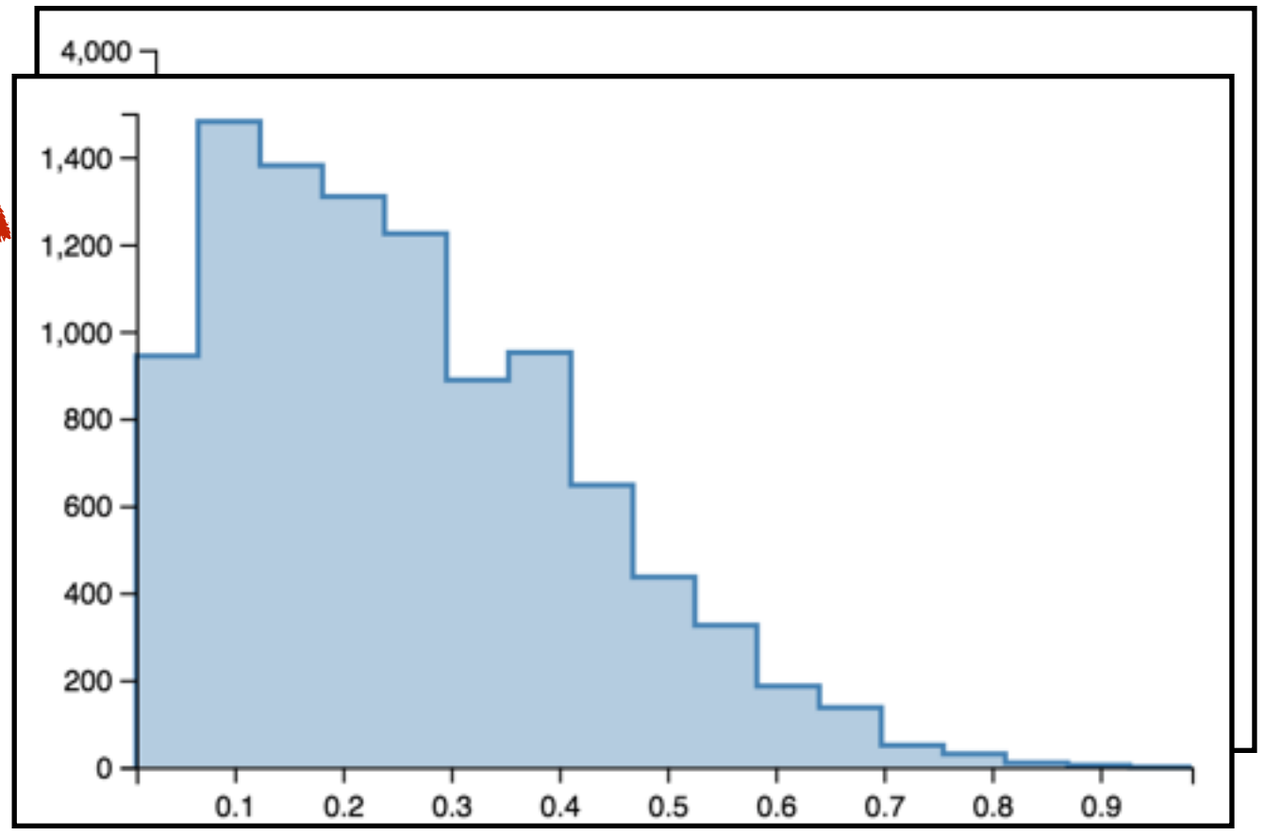
10K samples.
26 different samples.

posterior mean
of mixing prob. p

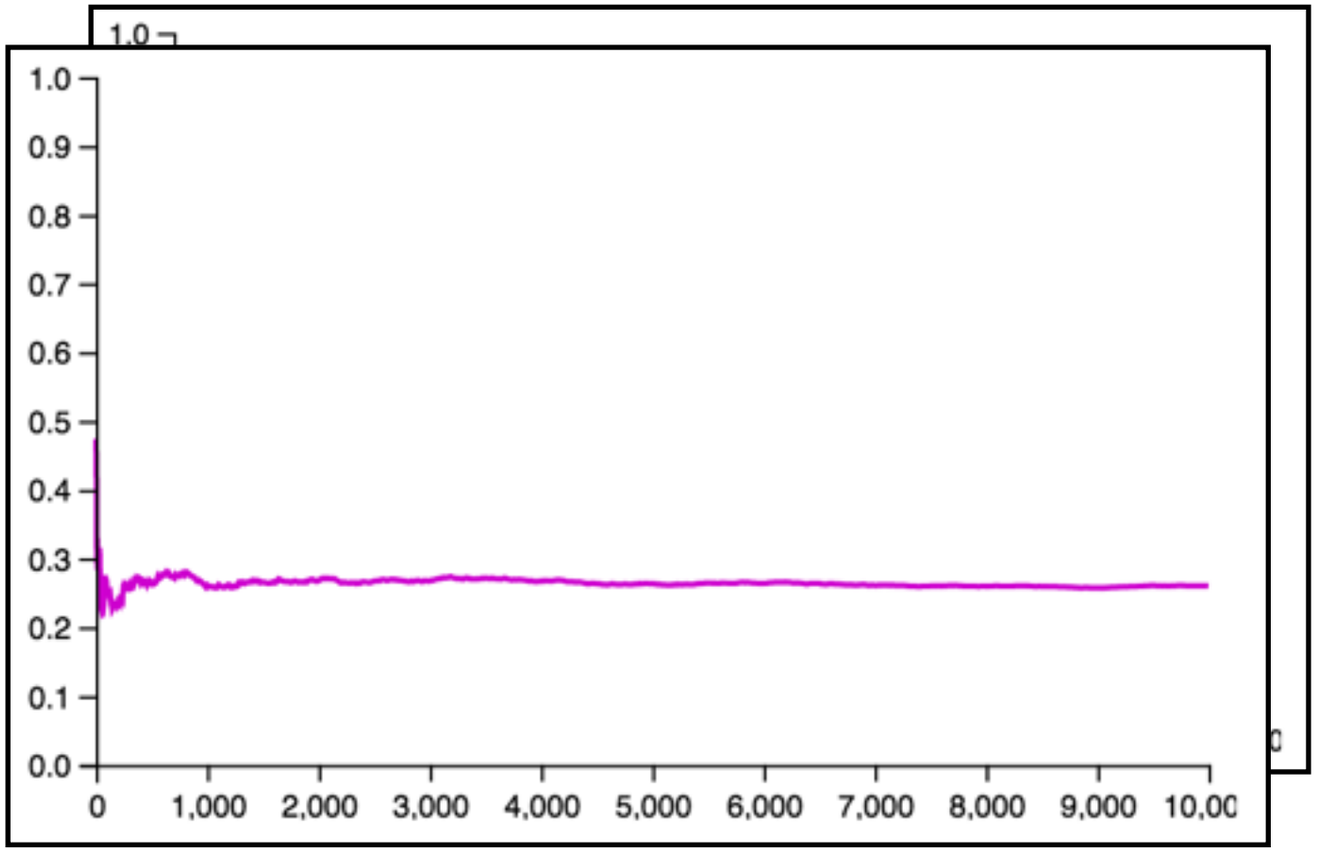


Gibbs-like algorithm (LMH)
applied to ~~original program~~
transformed program

10K samples.
~~26~~ different samples.
2721



posterior mean
of mixing prob. p



My research 2:
Semantics

```
(let [s (sample (normal 0 10))
      b (sample (normal 0 10))
      f (fn [x] (+ (* s x) b))]

  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)

  (predict :sb [s b]))
```

```
(let [s (sample (normal 0 10))
      b (sample (normal 0 10))
      f (fn [x] (+ (* s x) b))]

  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)

  (predict :sb [s b]))
```

(predict :f f)

```
(let [s (sample (normal 0 10))
      b (sample (normal 0 10))
      f (fn [x] (+ (* s x) b))]

  (observe (normal (f 1) 1) 2.5)
  (observe (normal (f 2) 1) 3.8)
  (observe (normal (f 3) 1) 4.5)
  (observe (normal (f 4) 1) 8.9)
  (observe (normal (f 5) 1) 10.1)

  (predict :sb [s b]))
```

```
(predict :f f)
```

Generates a random function of type R→R.
But its mathematical meaning is not clear.

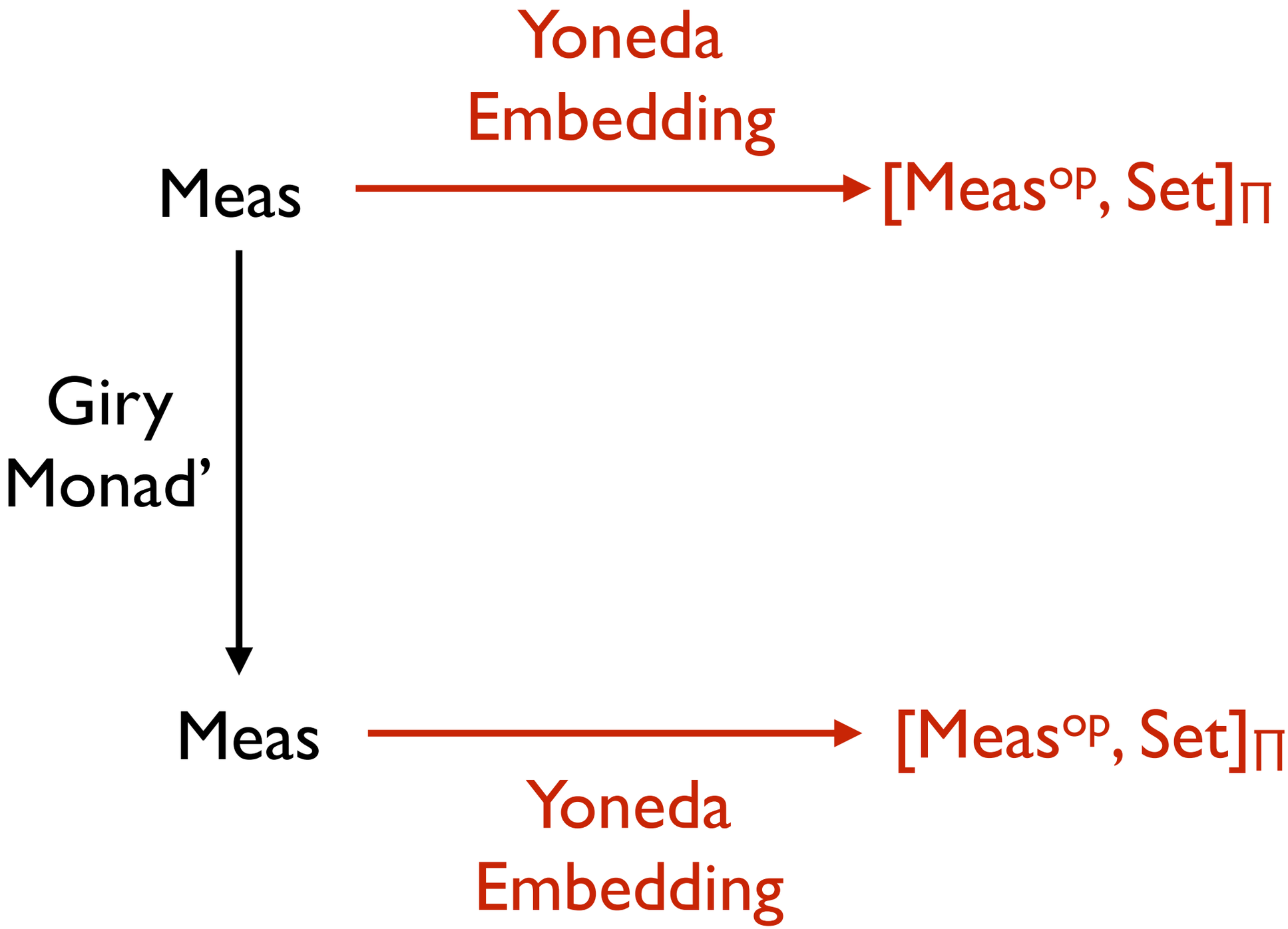
Measurability issue

- Measure theory is the foundation of probability theory that avoids paradoxes.
- Silent about high-order functions.
  - [Halmos] ev(f,a) = f(a) is not measurable.
  - The category of measurable sets is not CCC.
- But Anglican supports high-order functions.

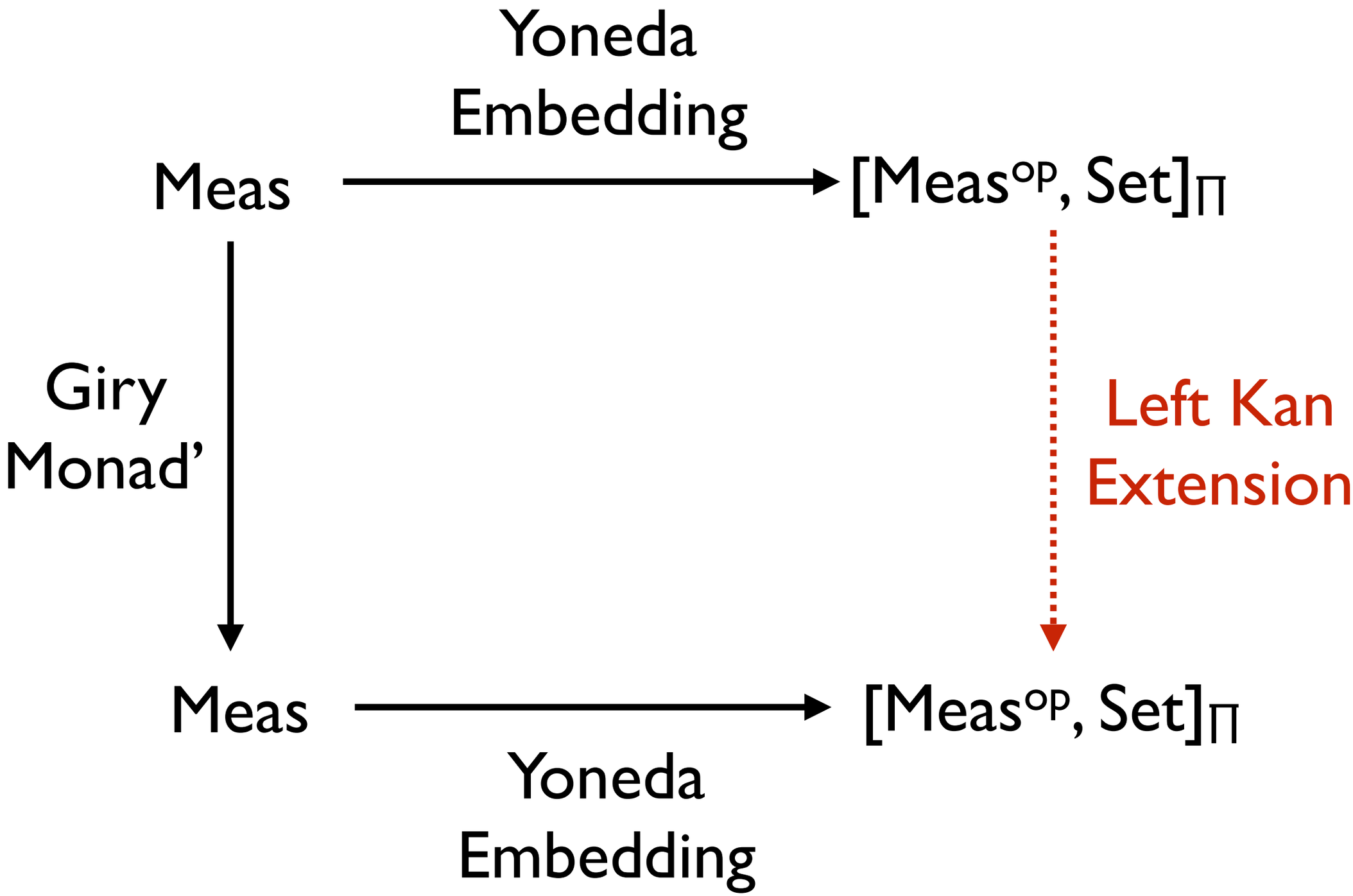
Use category theory to extend measure theory.

Use category theory to extend measure theory.

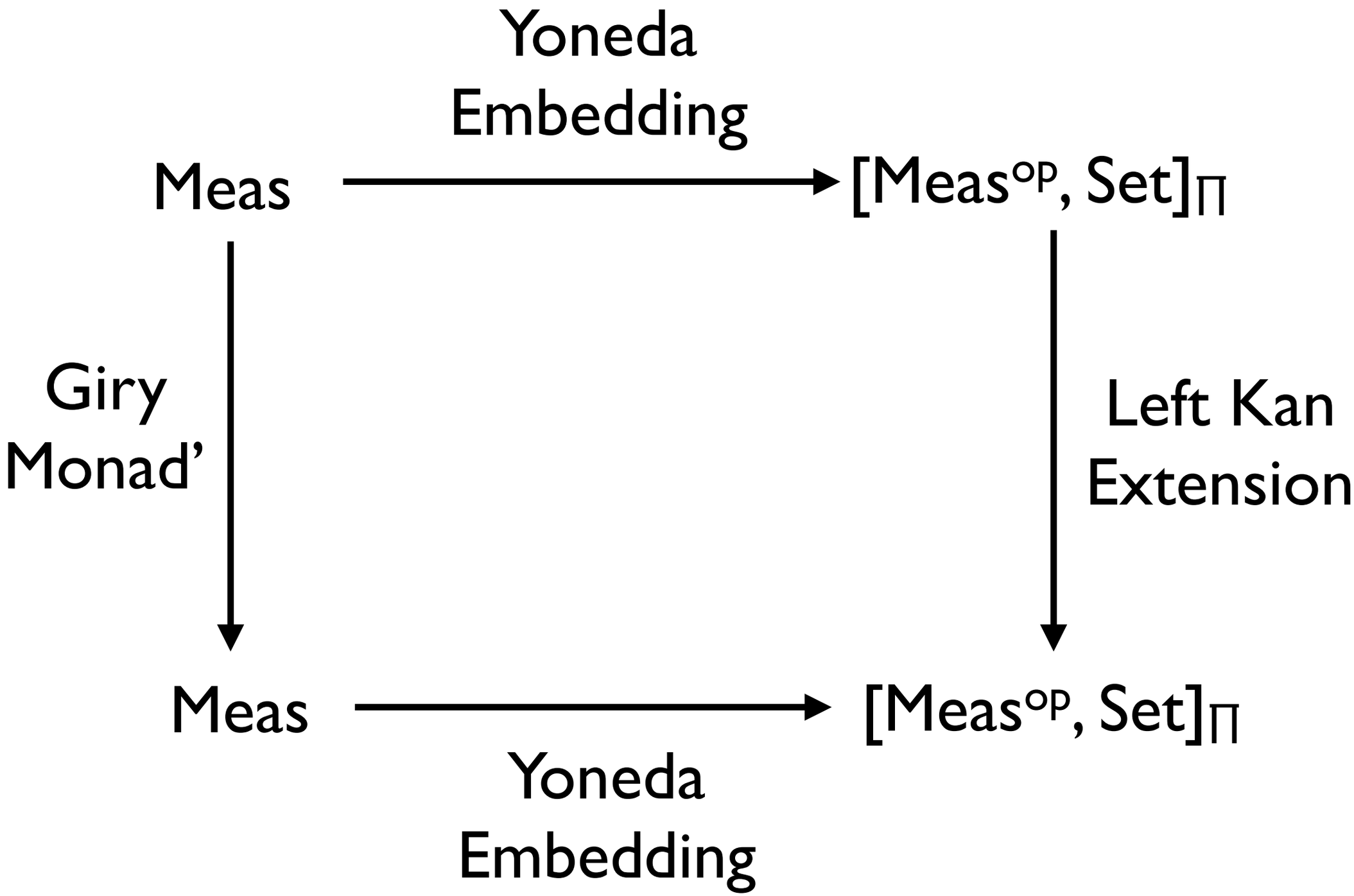
Use category theory to extend measure theory.



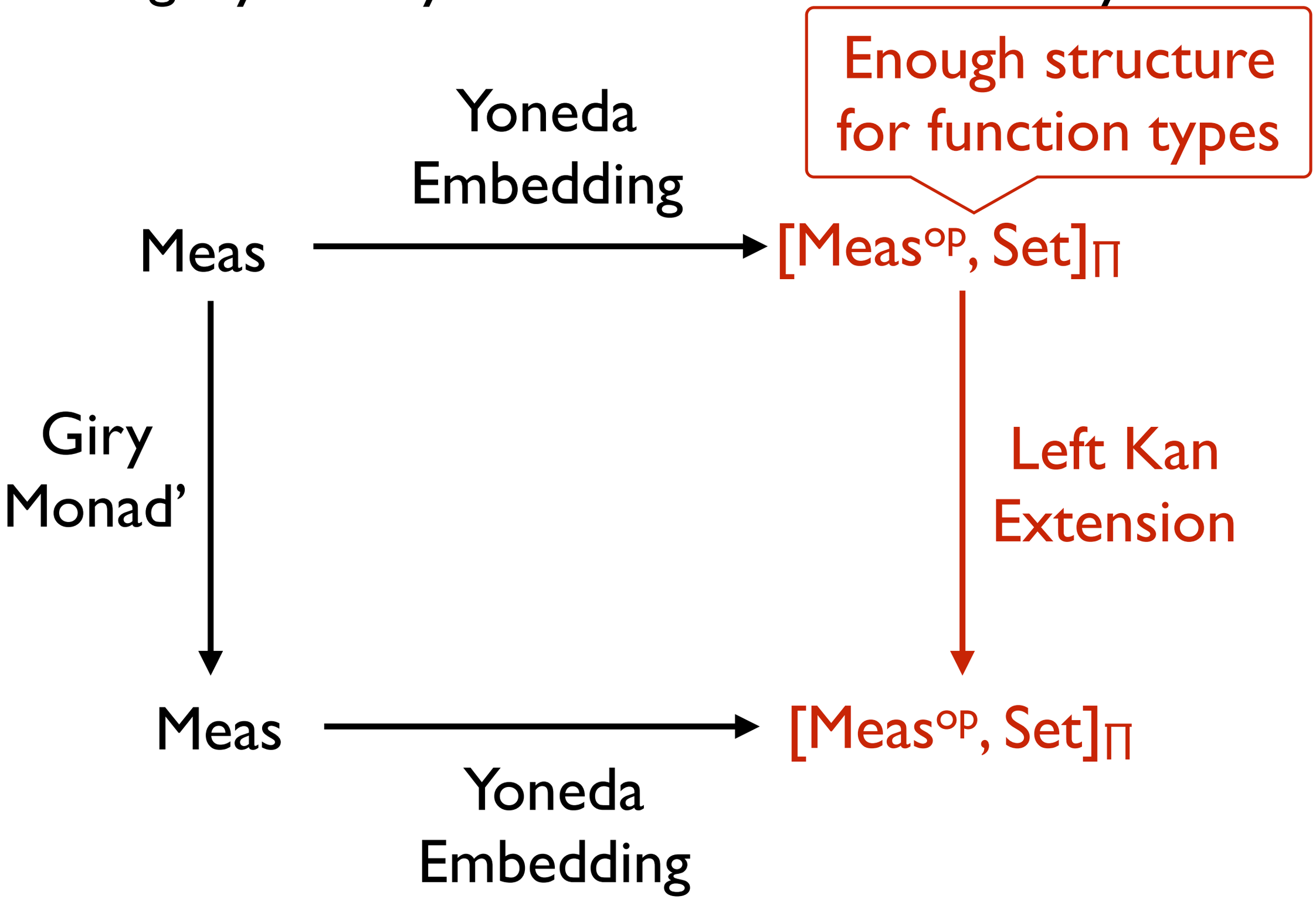
Use category theory to extend measure theory.



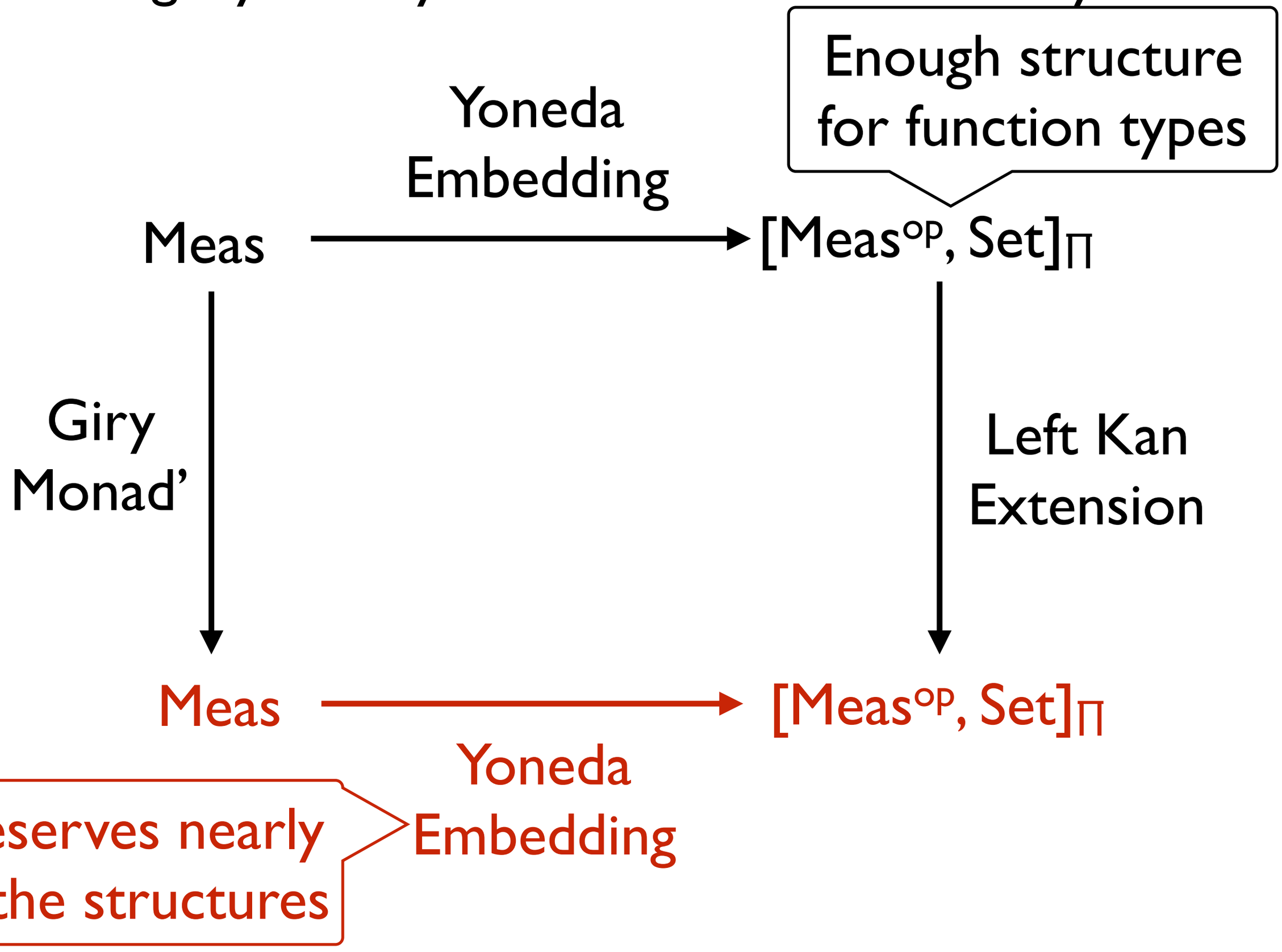
Use category theory to extend measure theory.



Use category theory to extend measure theory.



Use category theory to extend measure theory.



[Question] Are all definable functions from R to R in a high-order probabilistic PL measurable?

Our semantics says that the answer is yes for a core call-by-value language.

$[\![\mathbf{1} \to (R \to R)]\!]$ consists of:

equivalence classes of measurable functions $f : \Omega \times R \to R$ for probability spaces Ω.

The function f is what probabilists call a measurable stochastic process.