Program transformation for probabilistic programs
(work in progress)

Hongseok Yang
University of Oxford

Joint work with David Tolpin, Jan-Willem van de Meent, Frank Wood

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?

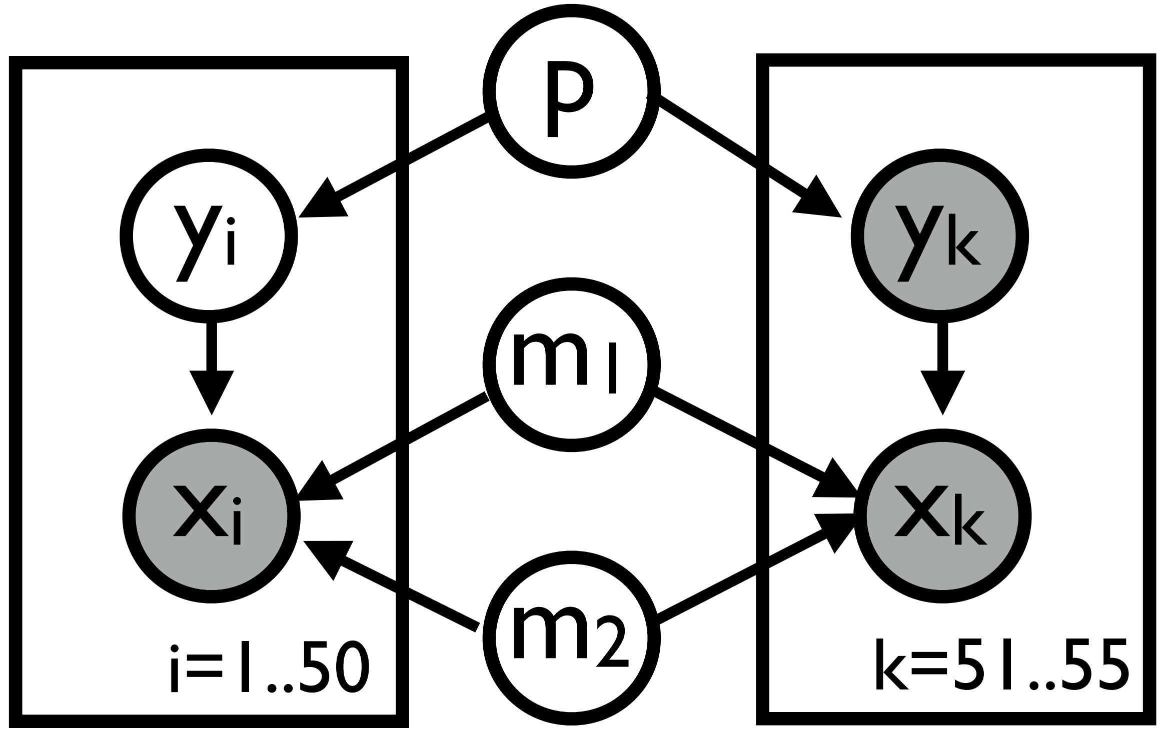
help MC inference algorithms for

Yes sometimes.

1. Marginalise <u>some</u> latent variables.
2. Compute posterior wrt. <u>some</u> data.

Take-home message

PL techniques might help develop symbolic techniques for optimising prob. programs locally.



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

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

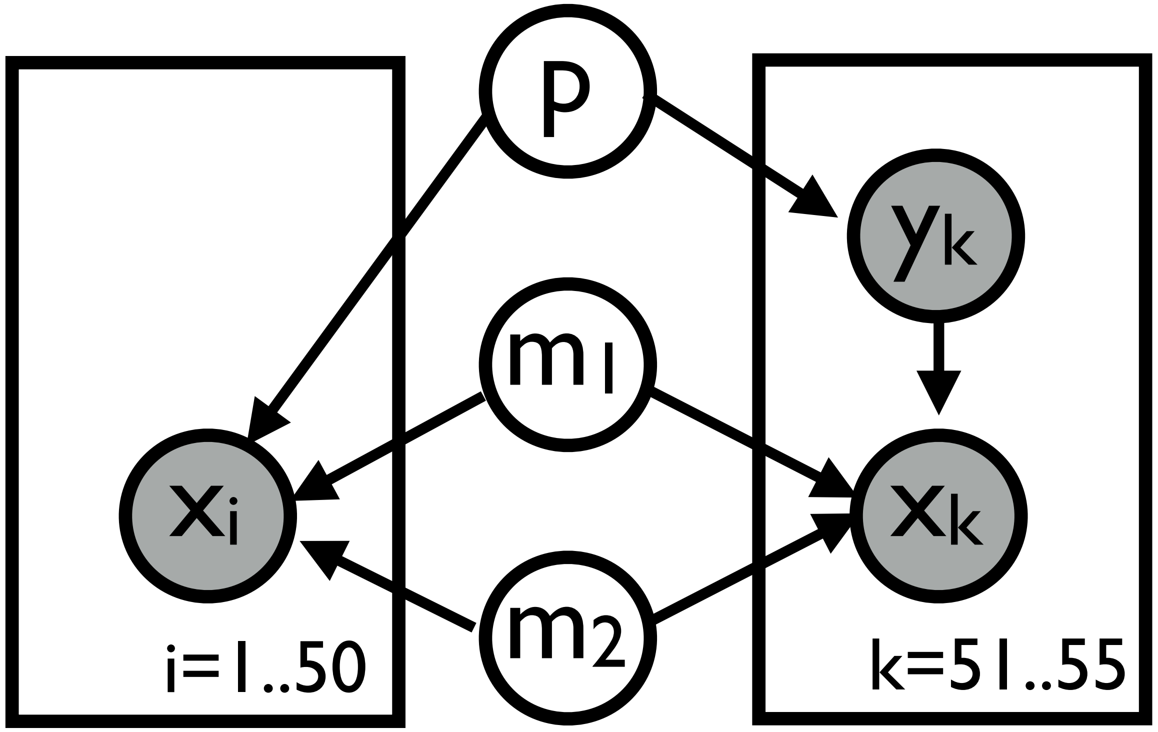
$y_i \sim \mathrm{flip}(p)$ $\underline{x_i} \sim \mathrm{normal}(m_{y_i},10)$

$\underline{y_k} \sim \mathrm{flip}(p)$ $\underline{x_k} \sim \mathrm{normal}(m_{\underline{y_k}},10)$

for $i = 1..50$, $k = 51..55$

Q: posterior of p?

Marginalise y_i.



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

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

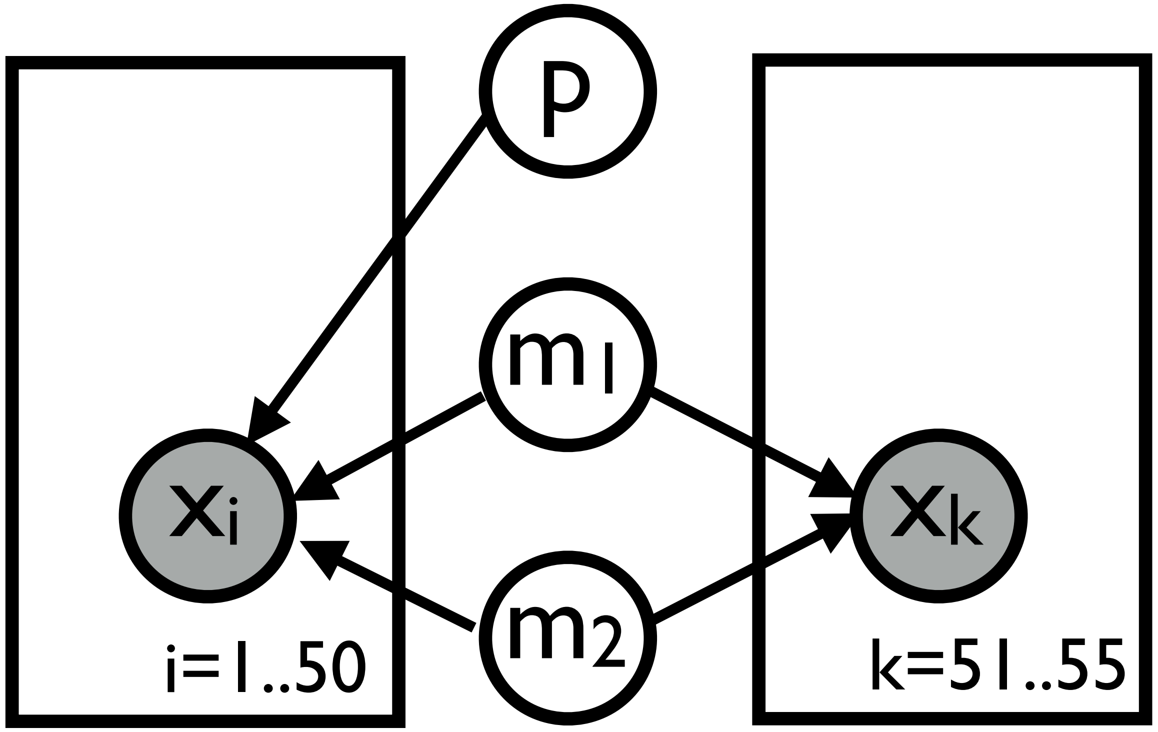
$\underline{x_i} \sim p*\mathrm{normal}(m_1,10) + (1-p)*\mathrm{normal}(m_2,10)$

$\underline{y_k} \sim \mathrm{flip}(p)$ $\underline{x_k} \sim \mathrm{normal}(m_{\underline{y_k}},10)$

for i = 1..50, k = 51..55

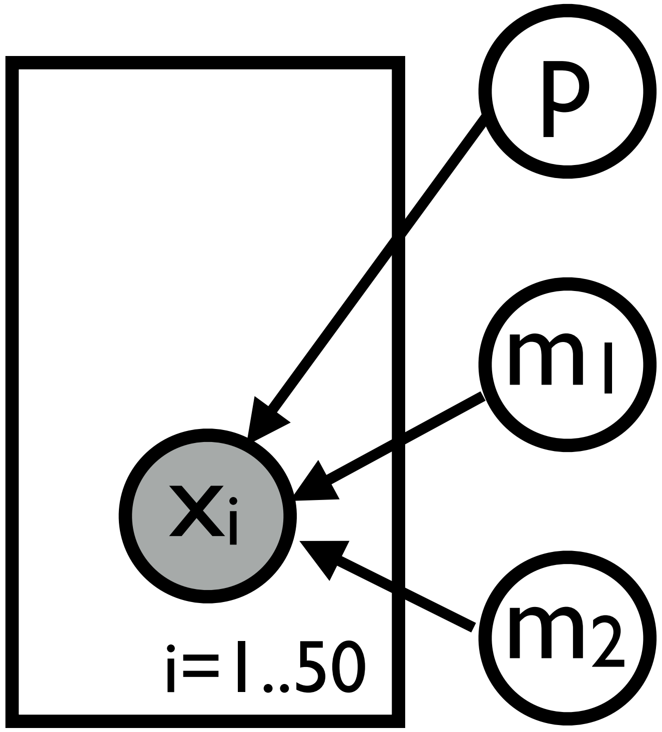
Q: posterior of p?

Marginalise y_i.
Post. wrt. y_k.



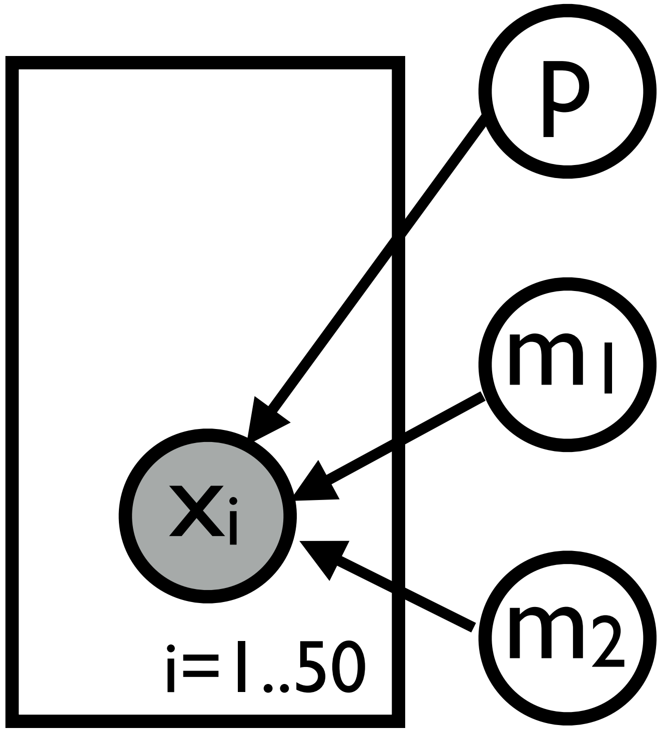

```
p ~ beta(..., ...)
m1 ~ normal(10,10) m2 ~ normal(10,10)
xi ~ p*normal(m1,10) + (1-p)*normal(m2,10)
                 xk ~ normal(myk,10)
               for i = 1..50, k = 51..55

Q: posterior of p?
```

Marginalise y_i.
Post. wrt. y_k.
Post. wrt. x_k.

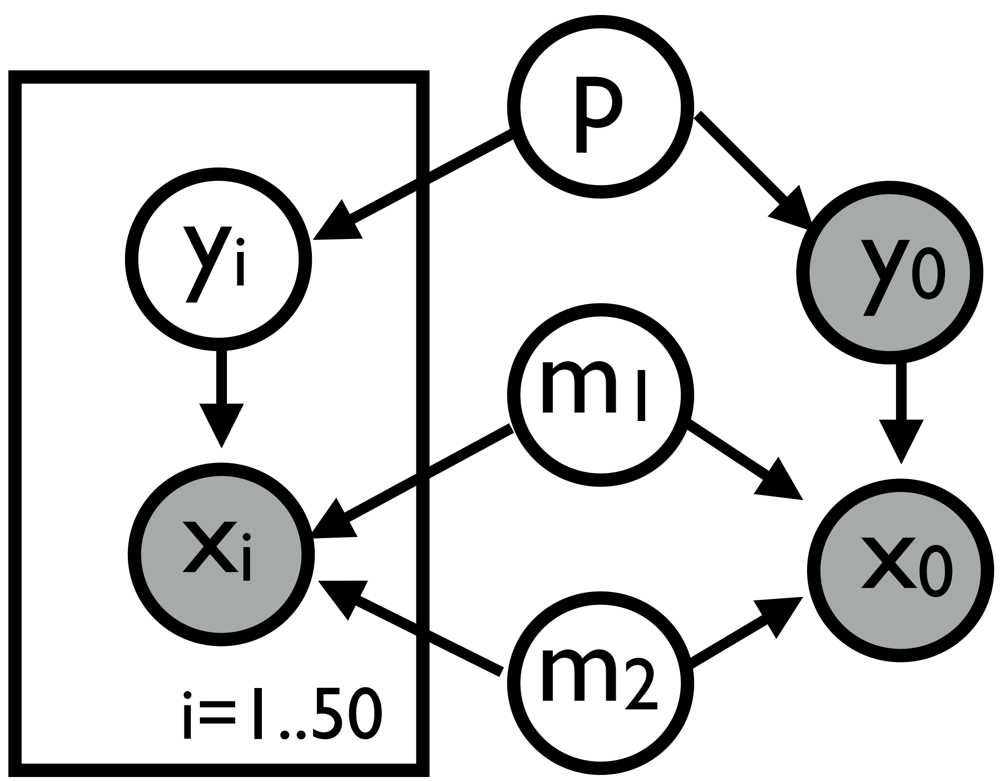


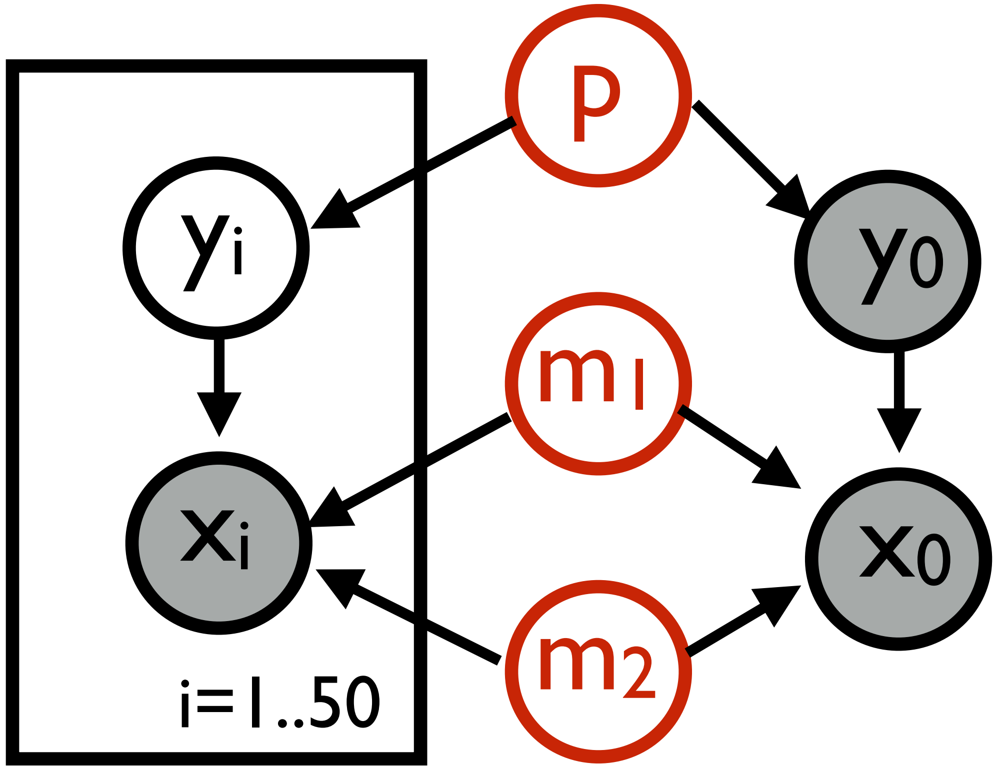

```
p ~ beta(…, …)
m₁ ~ normal(…, …)   m₂ ~ normal(…, …)
xᵢ ~ p*normal(m₁,10) + (1-p)*normal(m₂,10)

              for i = 1..50

Q: posterior of p?
```

Marginalise y_i.
Post. wrt. y_k.
Post. wrt. x_k.
Run MC algo.

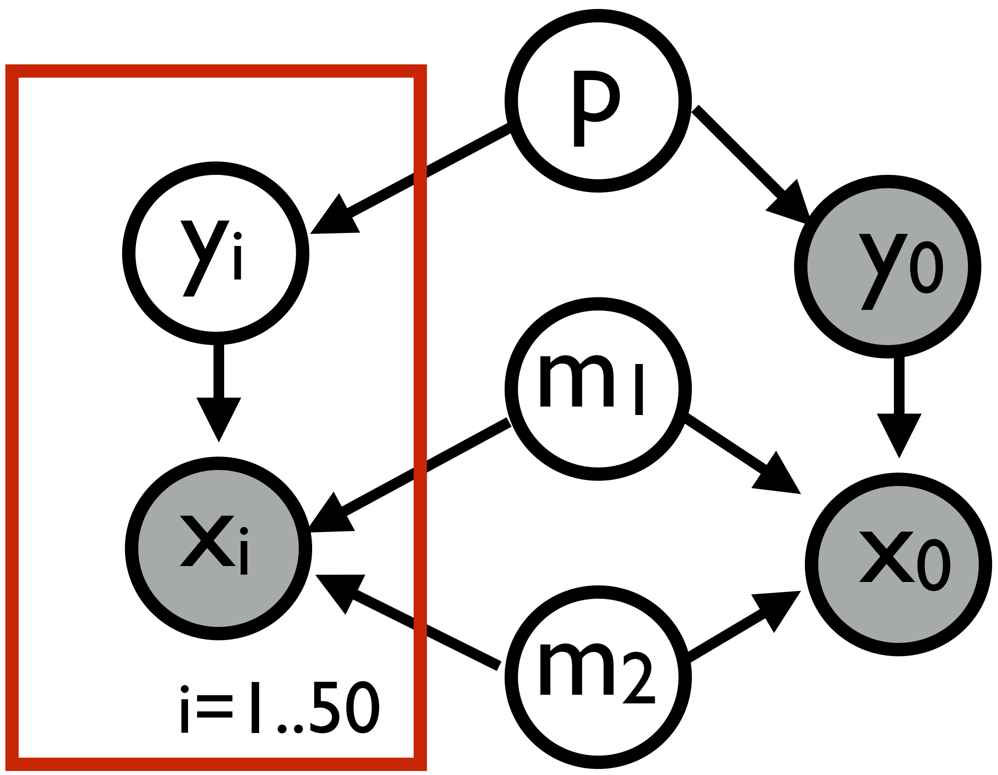



```
p ~ beta(…, …)
m₁ ~ normal(…, …)   m₂ ~ normal(…, …)
xᵢ ~ p*normal(m₁,10) + (1-p)*normal(m₂,10)

            for i = 1..50

Q: posterior of p?
```

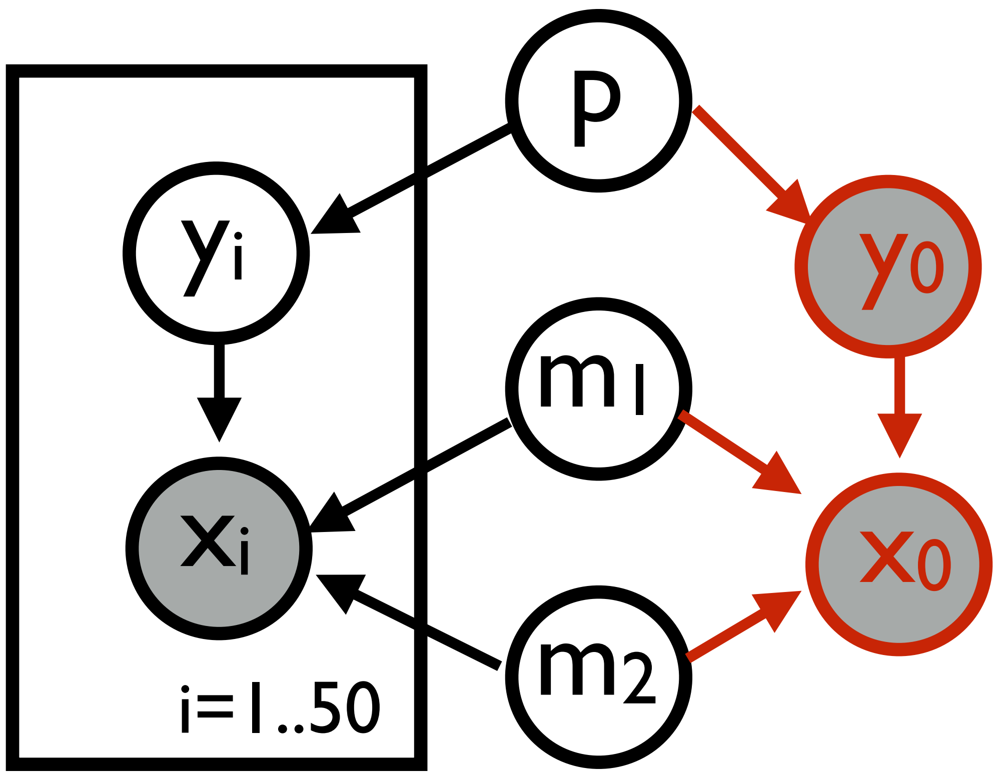

p
yi
y0
m1
xi
x0
m2
i=1..50



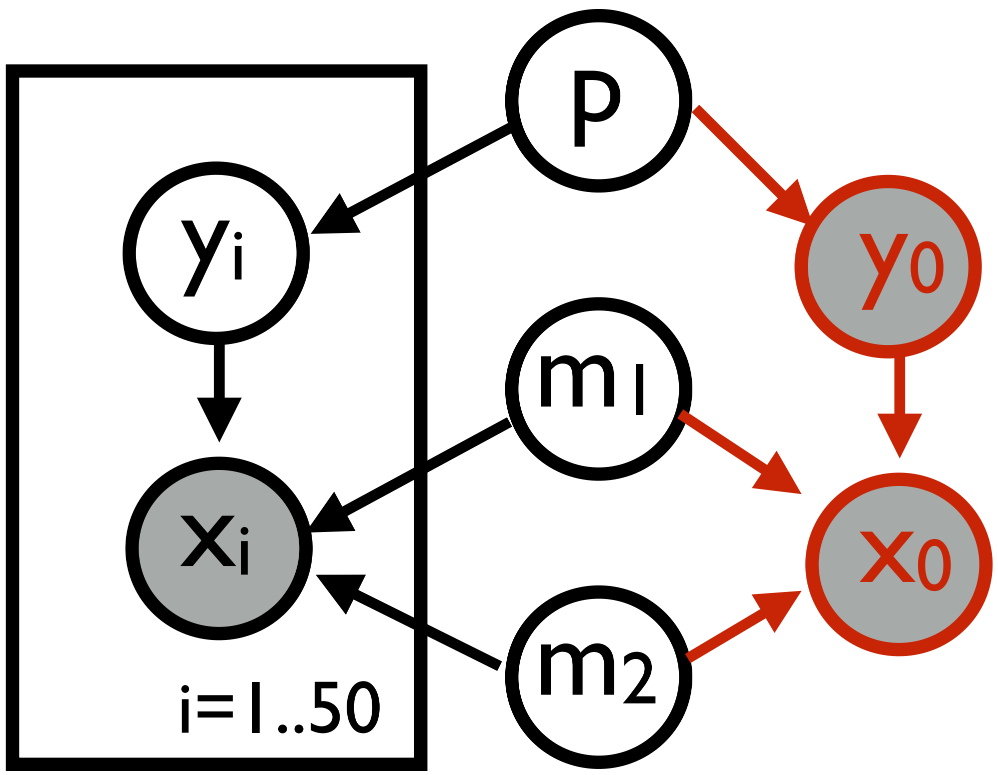

```
(assume p (sample (beta 1. 1.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))
```



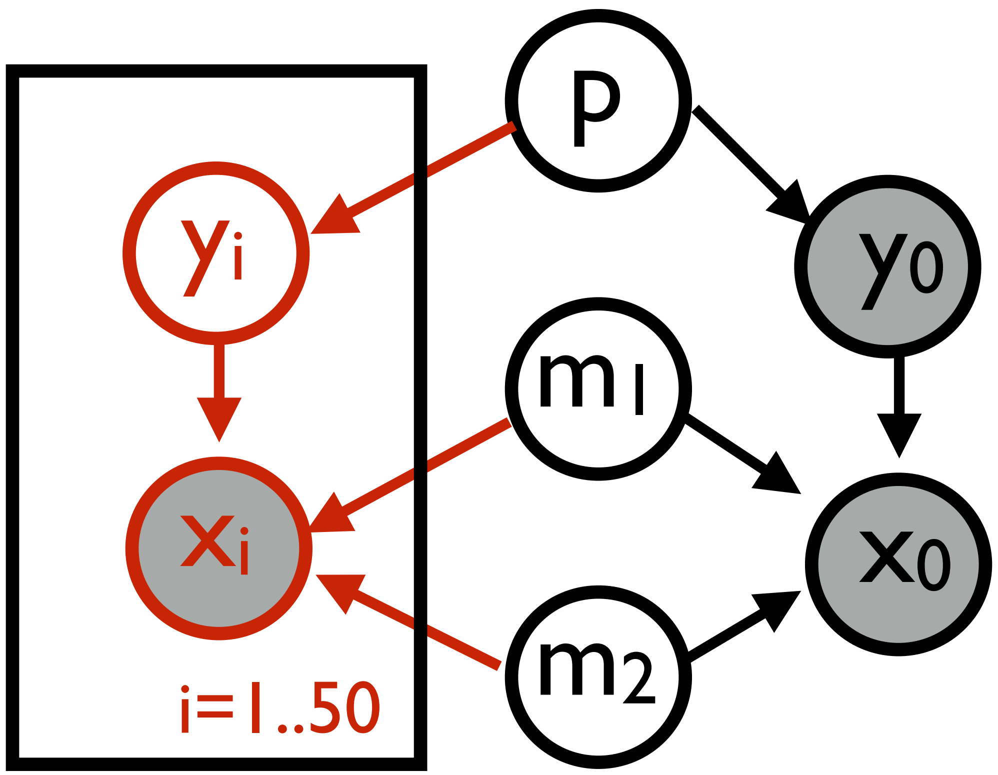

```
(assume p (sample (beta 1. 1.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))

(assume f
  (lambda (P M1 M2)
    (let ((y (sample (flip P))))
      (if y (normal M1 10.)
            (normal M2 10.)))))
```



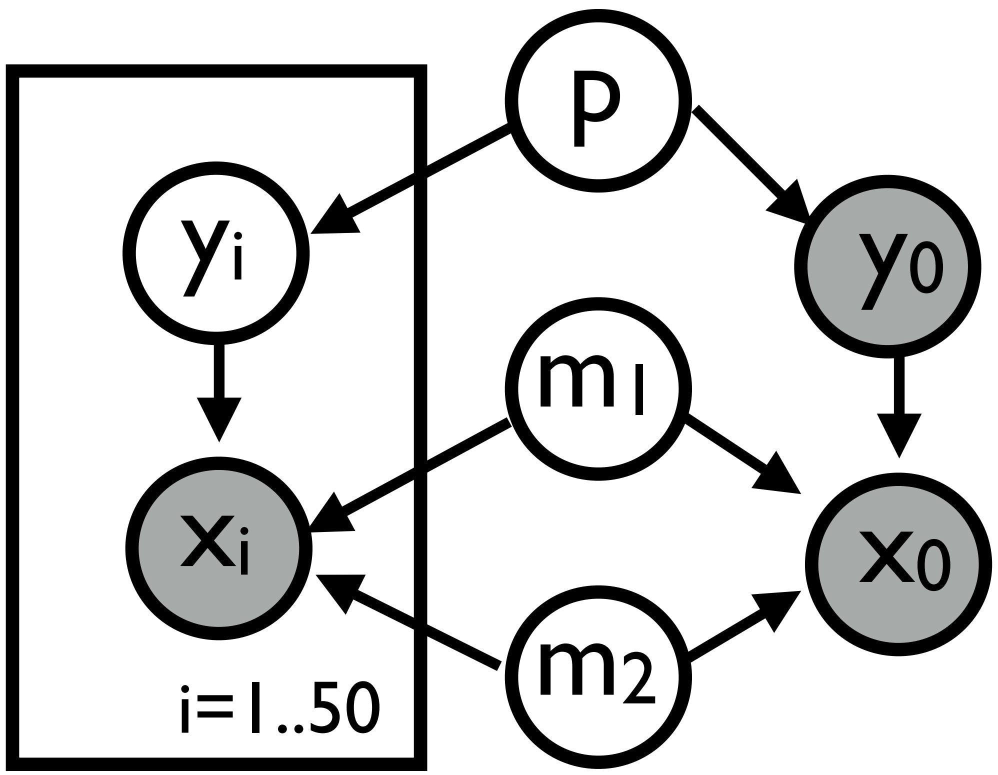

```
(assume p (sample (beta 1. 1.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))

(assume f
  (lambda (P M1 M2)
    (let ((y (sample (flip P))))
      (if y (normal M1 10.)
            (normal M2 10.)))))

(observe (flip p) y0)
(observe (if y0   (normal m1 10.)
                  (normal m2 10.))
         x0)
```



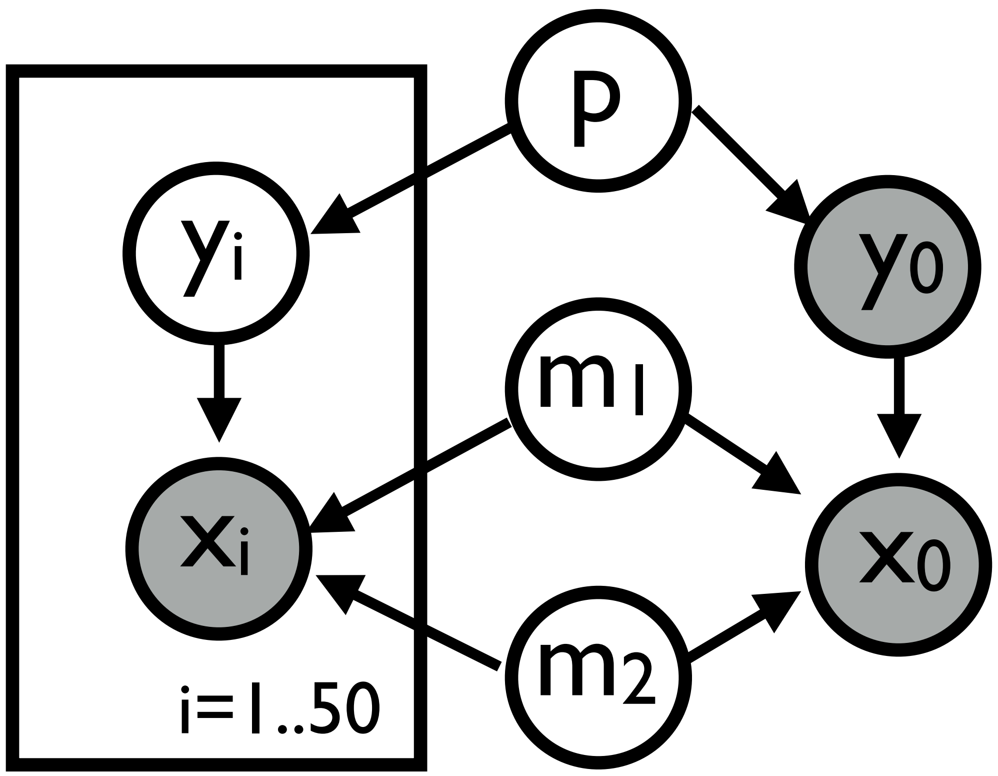

```
(assume p (sample (beta 1. 1.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))

(assume f
  (lambda (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)
```




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

(assume f
  (lambda (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) x1)
…
(observe (f p m1 m2) x50)
```

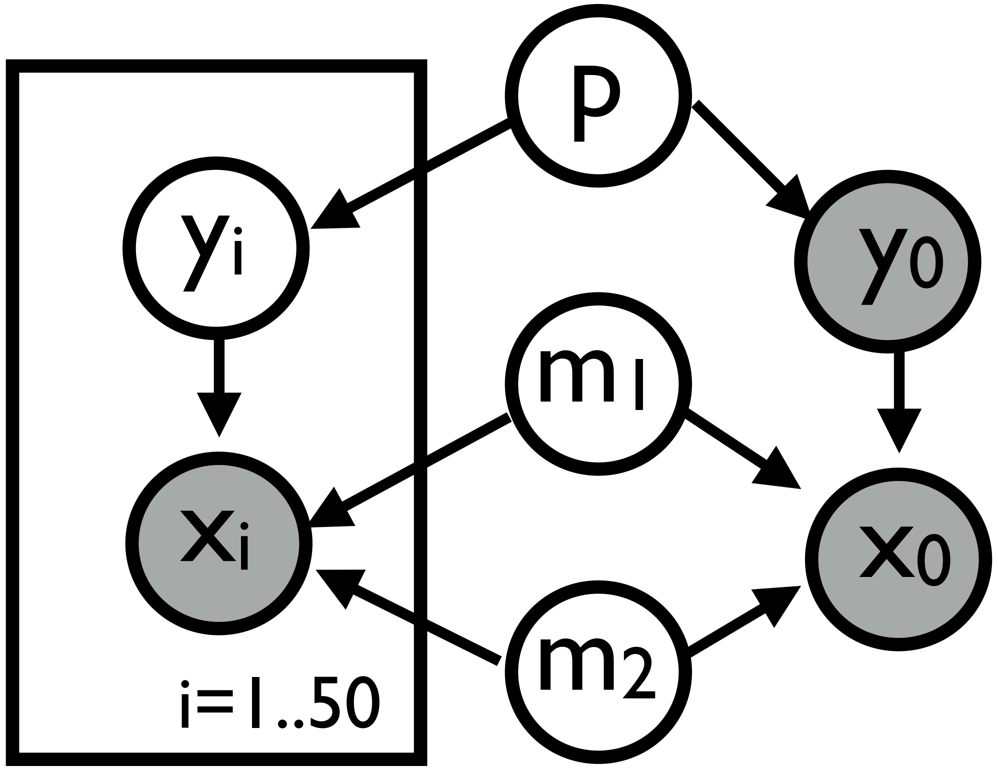



```
(assume p (sample (beta 1. 1.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))

(assume f
  (lambda (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) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```




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

(assume f (erpify-dist
  (lambda (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) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

1. ERPify.
2. Move observe backwards.




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

(assume f (erpify-dist
  (lambda (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) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

Elementary random procedure (ERP)

Examples:

`flip, normal, gamma, …`

Non-examples:

```
(lambda (P) (not (sample (flip P))))
```

Elementary random procedure (ERP)

Procedure that creates an object ob with the following two methods:

`(sample ob)` for sampling a value

`(observe ob v)` for computing log pdf / pmf

Erpification

- Generates an ERP from a normal procedure.
- Finds a formula for log pdf/pmf.
- Results expressed in terms of existing ERPs.

```
(lambda (P)
  (not (sample (flip P))))
```

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

Erpification

- Generates an ERP from a normal procedure.
- Finds a formula for log pdf/pmf.
- Results expressed in terms of existing ERPs.

```
(lambda (P)
  (not (sample (flip P))))
```

→

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

```
(lambda (P M1 M2)
  (if (sample (flip P))
    (sample (normal M1 1))
    (sample (normal M2 1))))
```

→

```
(lambda (P M1 M2)
  (mixture-dist
    [[P
      (normal M1 1)]
     [(- 1 P)
      (normal M2 1)]]))
```

Erpification

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

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

```
(lambda (P)
  (not (sample (flip P))))
```

→

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

```
(lambda (P M1 M2)
  (if (sample (flip P))
    (sample (normal M1 1))
    (sample (normal M2 1))))
```

→

```
(lambda (P M1 M2)
  (mixture-dist
    [[P
      (normal M1 1)]
     [(- 1 P)
      (normal M2 1)]]))
```

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.

```
(lambda (P)
  (not
    (sample (flip P))))
```

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

```
(lambda (P)
  (not
    (sample (flip P))))
```

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

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

```
(lambda (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.

```
(lambda (P)
  (not
    (sample (flip P))))
```

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

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

```
(lambda (P M1 M2)
  (if (sample (flip P))
    (sample (normal M1 1))
    (let ((R2 (sample (normal M2 1))))
      (sample (normal (+ R2 5) 1)))))
```

1. Symbolically enumerate all possible outputs.

2. Express the outputs using existing ERPs.

```
(lambda (P)
  (not
    (sample (flip P))))
```

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

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

```
(lambda (P M1 M2)
  (if (sample (flip P))
    (sample (normal M1 1))
    (let ((R2 (sample (normal M2 1))))
      (sample (normal (+ R2 5) 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.

```
(lambda (P)
  (not
    (sample (flip P))))
```

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

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

```
(lambda (P M1 M2)
  (if (sample (flip P))
    (sample (normal M1 1))
    (let ((R2 (sample (normal M2 1))))
      (sample (normal (+ R2 5) 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))]])
```

```
(erpify-dist
  (lambda (P M1 M2)
    (let ((y (sample (flip P))))
      (if y (normal M1 10.)
            (normal M2 10.)))))
```

```
(erpify-dist
  (lambda (P M1 M2)
    (let ((y (sample (flip P))))
      (if y (normal M1 10.)
            (normal M2 10.)))))
```

```
(lambda (P M1 M2)
  (sample
    (let ((y (sample (flip P))))
      (if y (normal M1 10.)
            (normal M2 10.)))))
```

```
(erpify-dist
  (lambda (P M1 M2)
    (let ((y (sample (flip P))))
      (if y (normal M1 10.)
            (normal M2 10.)))))
```

```
(lambda (P M1 M2)
  (sample
    (let ((y (sample (flip P))))
      (if y (normal M1 10.)
            (normal M2 10.)))))
```

```
{ (R1, P,   [R1:(normal M1 10.)]),
  (R2, 1-P, [R2:(normal M2 10.)])
}
```

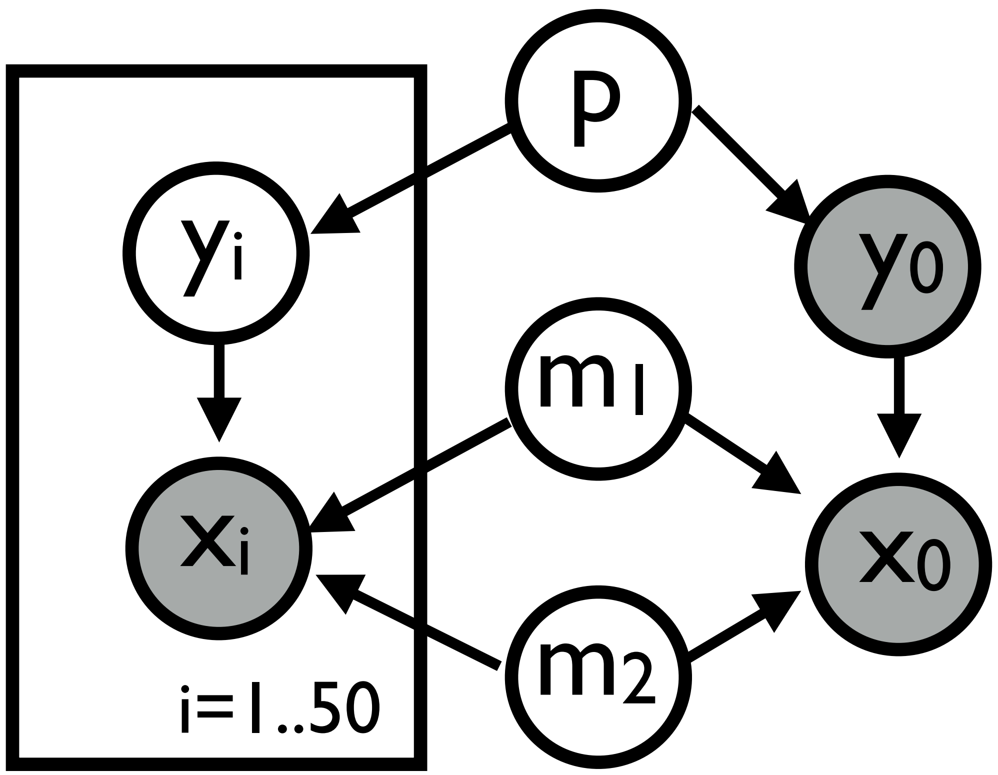
```
(erpify-dist
  (lambda (P M1 M2)
    (let ((y (sample (flip P))))
      (if y (normal M1 10.)
            (normal M2 10.)))))
```

```
(lambda (P M1 M2)
  (sample
    (let ((y (sample (flip P))))
      (if y (normal M1 10.)
            (normal M2 10.)))))
```

```
{ (R1, P,   [R1:(normal M1 10.)]),
  (R2, 1-P, [R2:(normal M2 10.)])
}
```

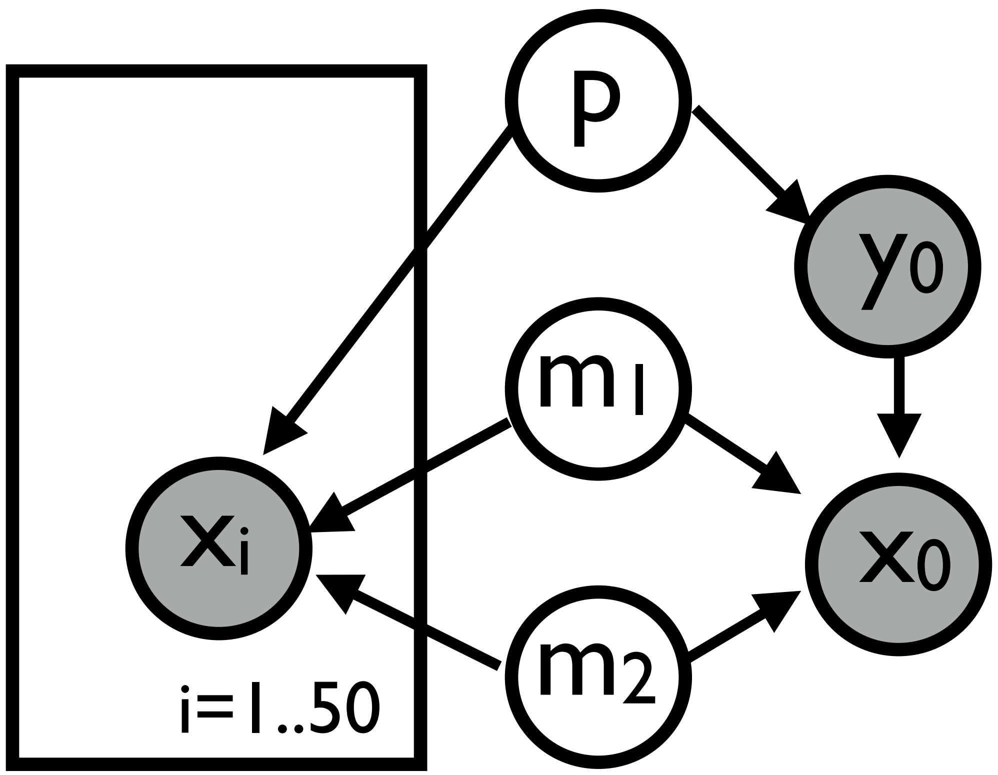
```
(mixture-dist
  [[P       (normal M1 10.)]
   [(- 1. P) (normal M2 10.)]])
```

1. ERPify.
2. Move observe backwards.



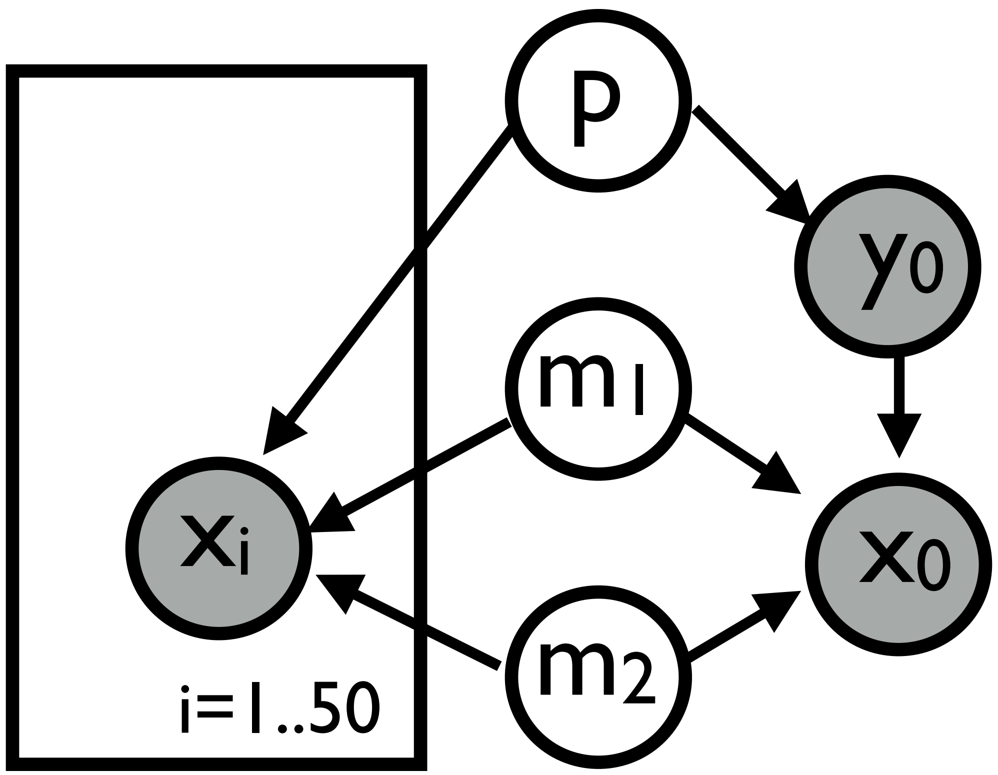

```
(assume p (sample (beta 1. 1.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))

(assume f (erpify-dist
  (lambda (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) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

1. ERPify.
2. Move observe backwards.



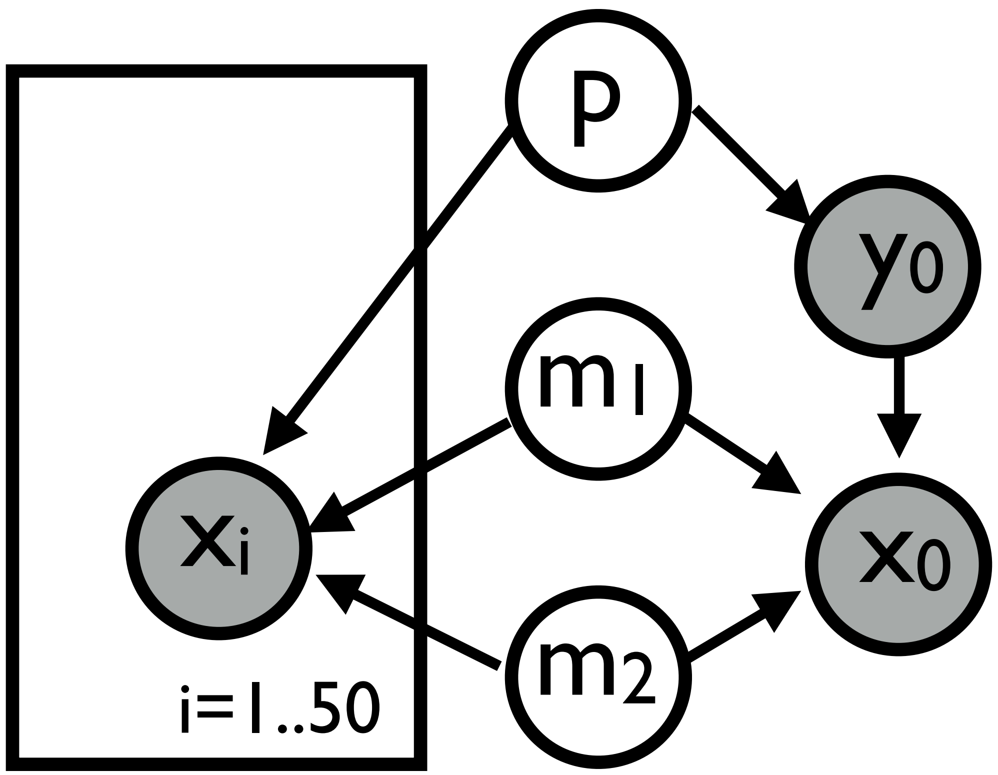

```
(assume p (sample (beta 1. 1.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P        (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (flip p) false)
(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

1. ERPify.
2. Move observe backwards.




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

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P        (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (flip p) false)
(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

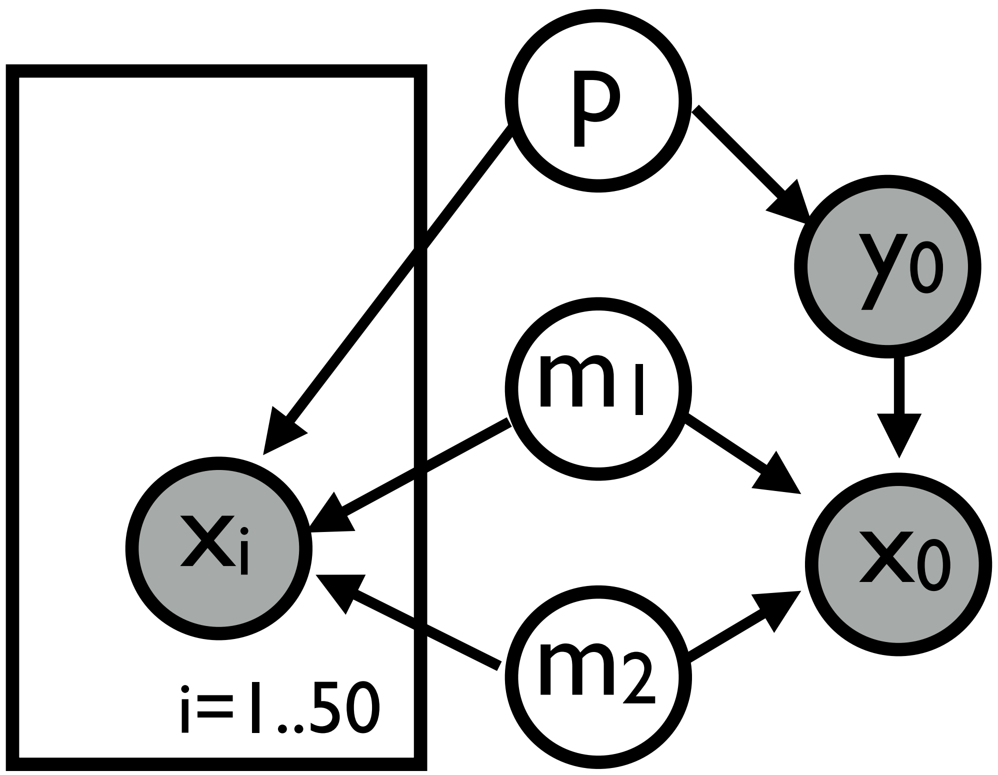
1. ERPify.
2. Move observe backwards.



p not defined.
Hence, independent.

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

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P          (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (flip p) false)
(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

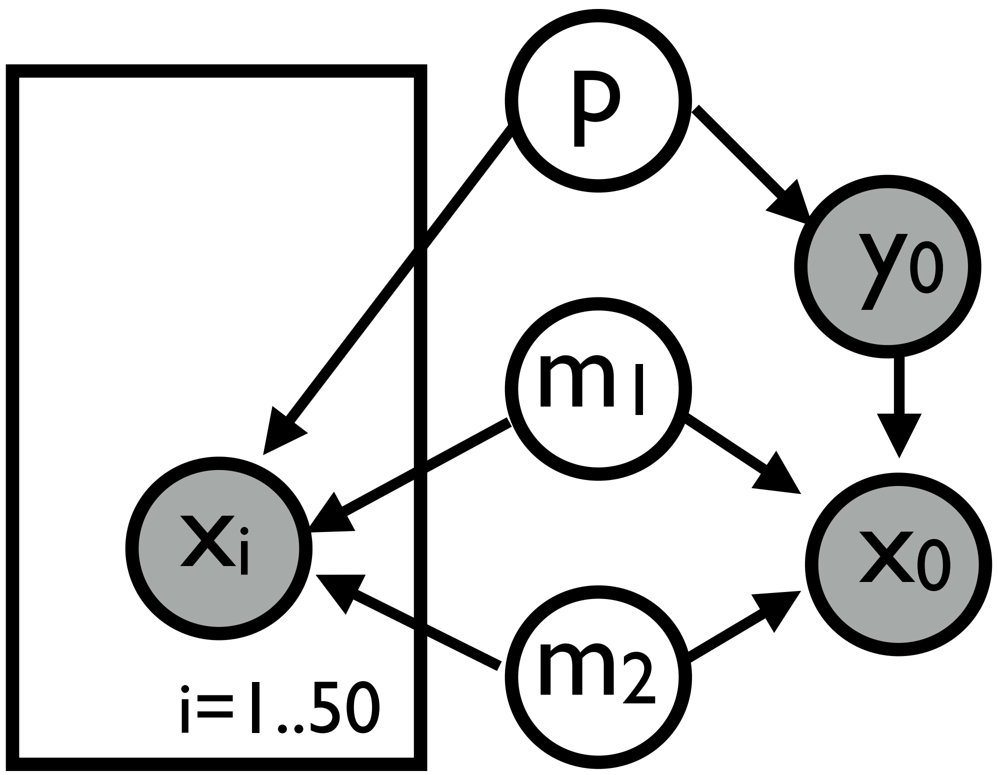
1. ERPify.
2. Move observe backwards.



p not defined.
Hence, independent.

```
(assume p (sample (beta 1. 1.)))
(observe (flip p) false)
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P         (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

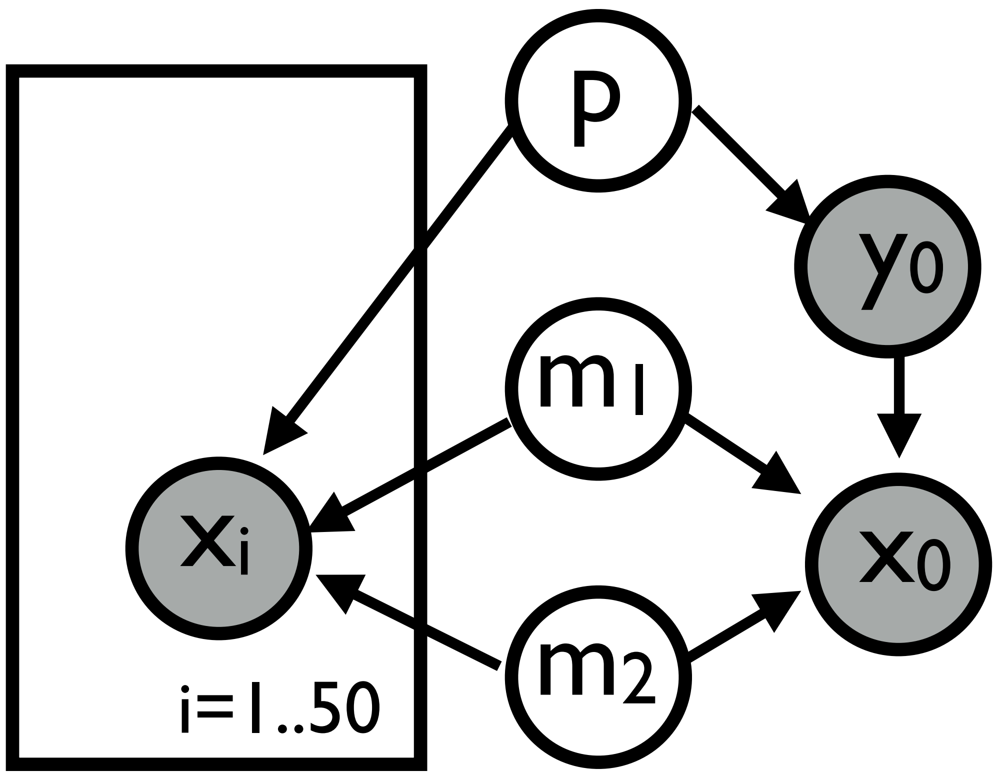
1. ERPify.
2. Move observe backwards.



p not defined.
Hence, independent.
Beta-Flip conjugacy.

```
(assume p (sample (beta 1. 1.)))
(observe (flip p) false)
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P          (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

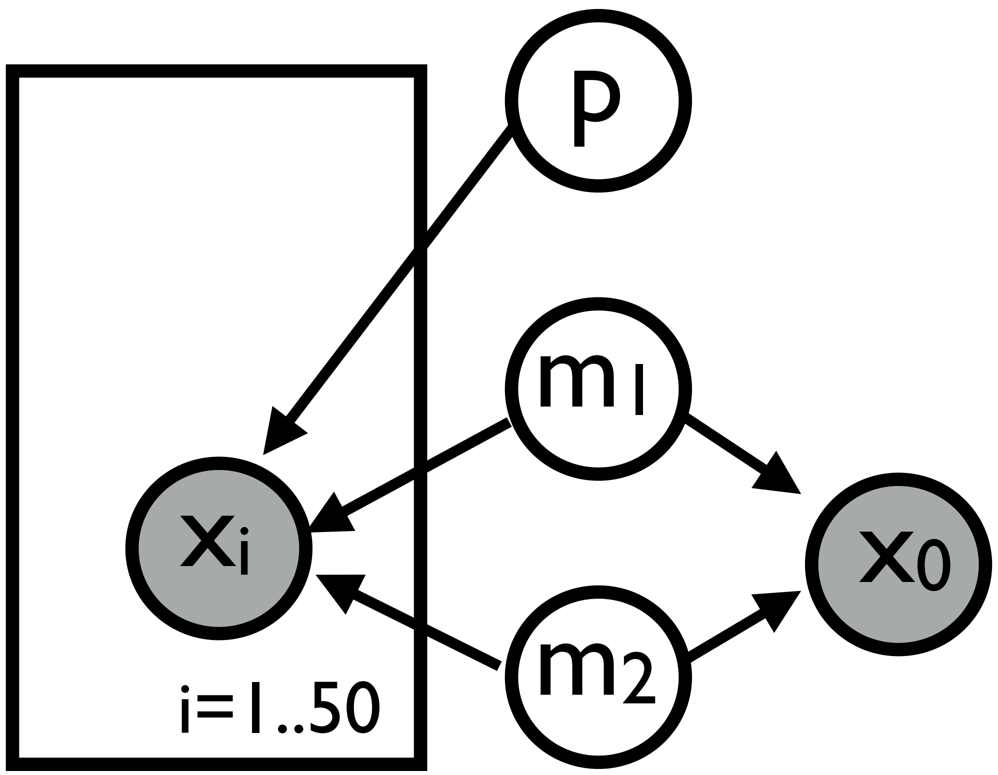
1. ERPify.
2. Move observe backwards.



p not defined.
Hence, independent.
Beta-Flip conjugacy.

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

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P        (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

1. ERPify.
2. Move observe backwards.



p not defined.

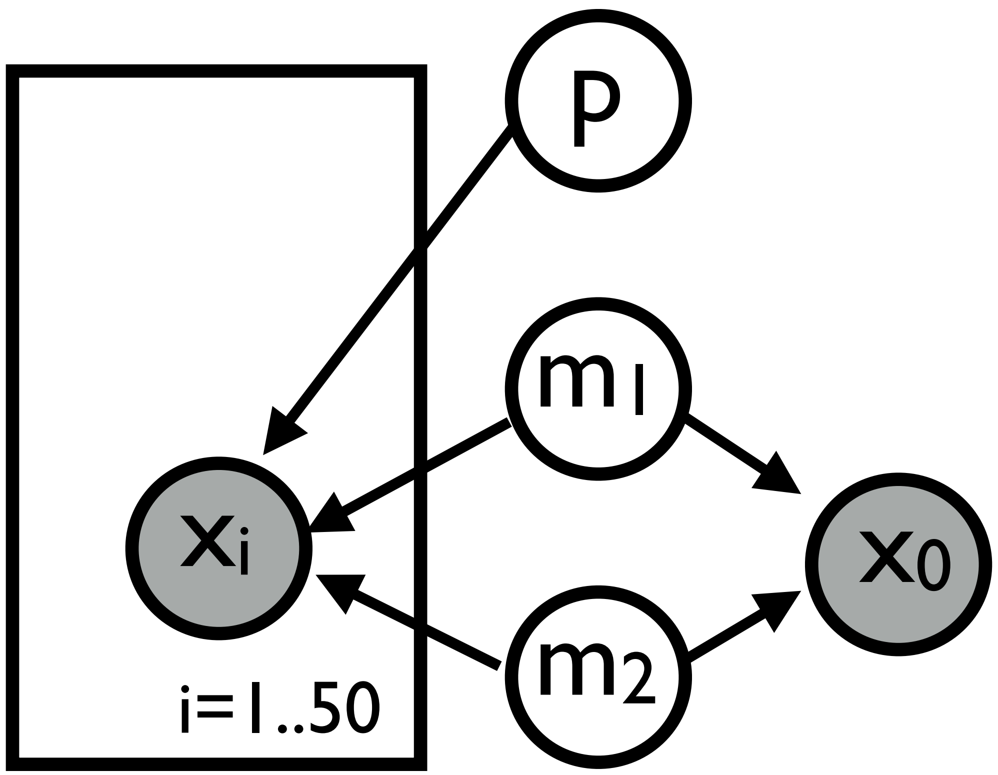
Hence, independent.

Beta-Flip conjugacy.

Constant importance weight.

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

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P          (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

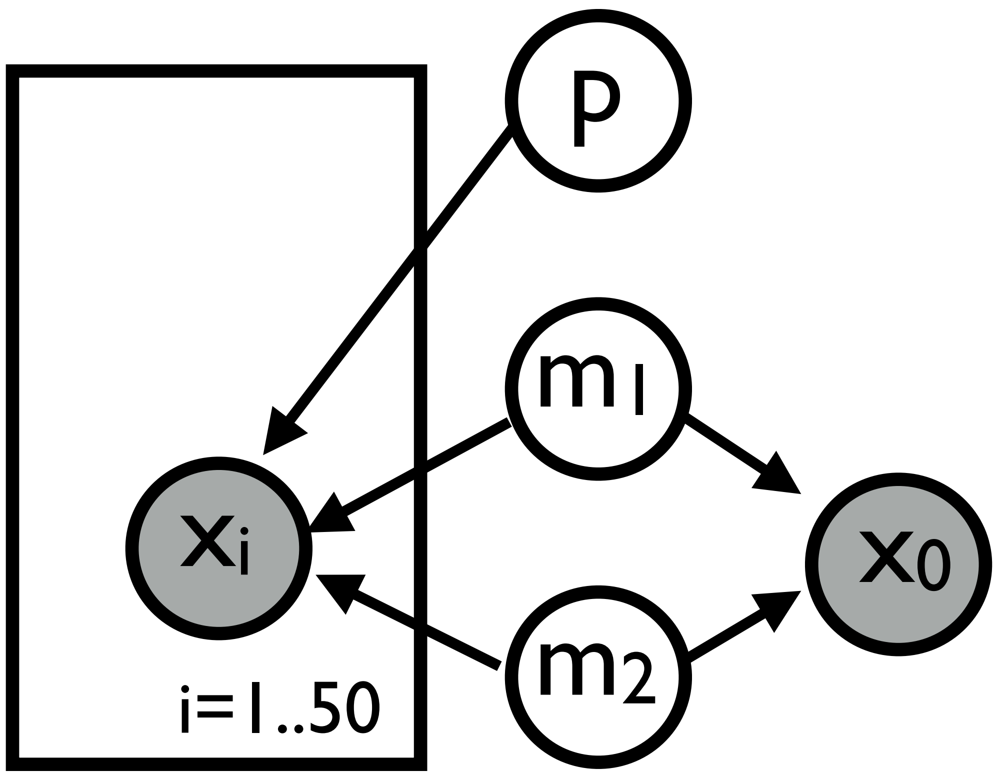
1. ERPify.
2. Move observe backwards.




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

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P        (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

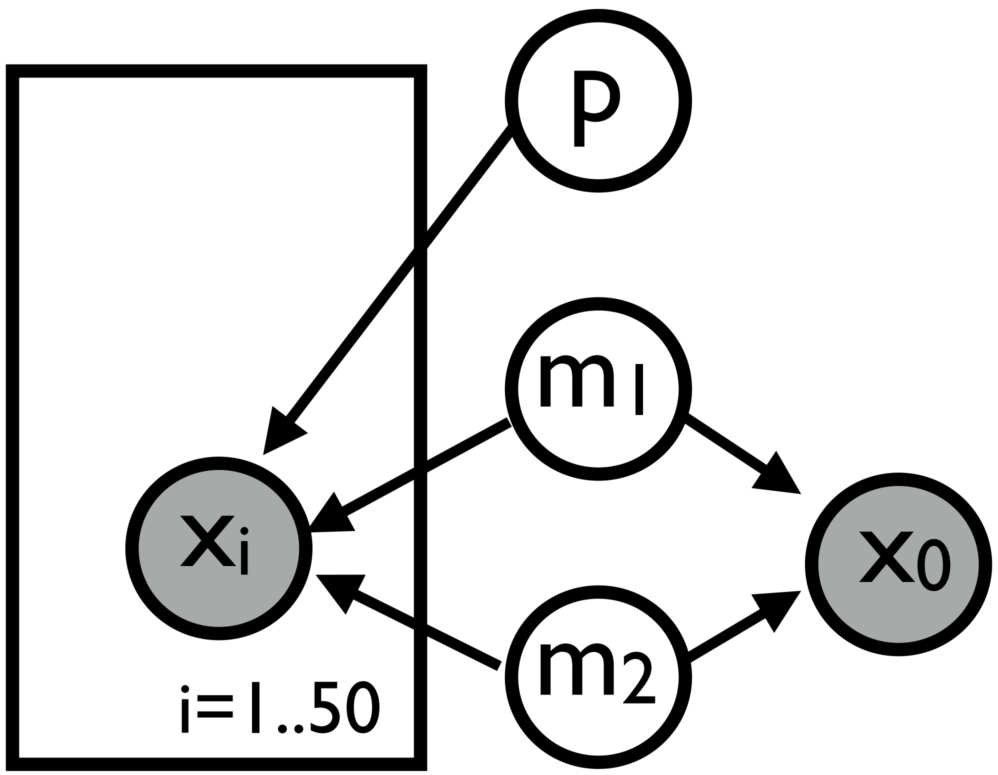
p not defined.
Hence, independent.
Beta-Flip conjugacy.
Constant importance weight.

1. ERPify.
2. Move observe backwards.




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

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P        (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

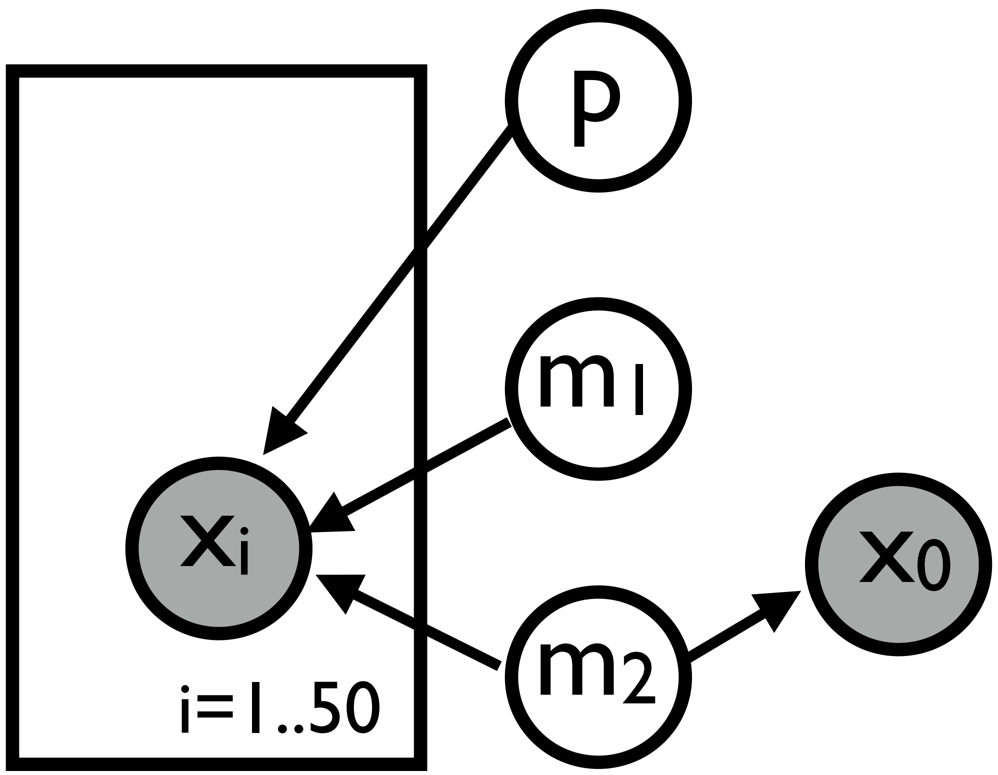
1. ERPify.
2. Move observe backwards.



Partial evaluation.

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

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P          (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

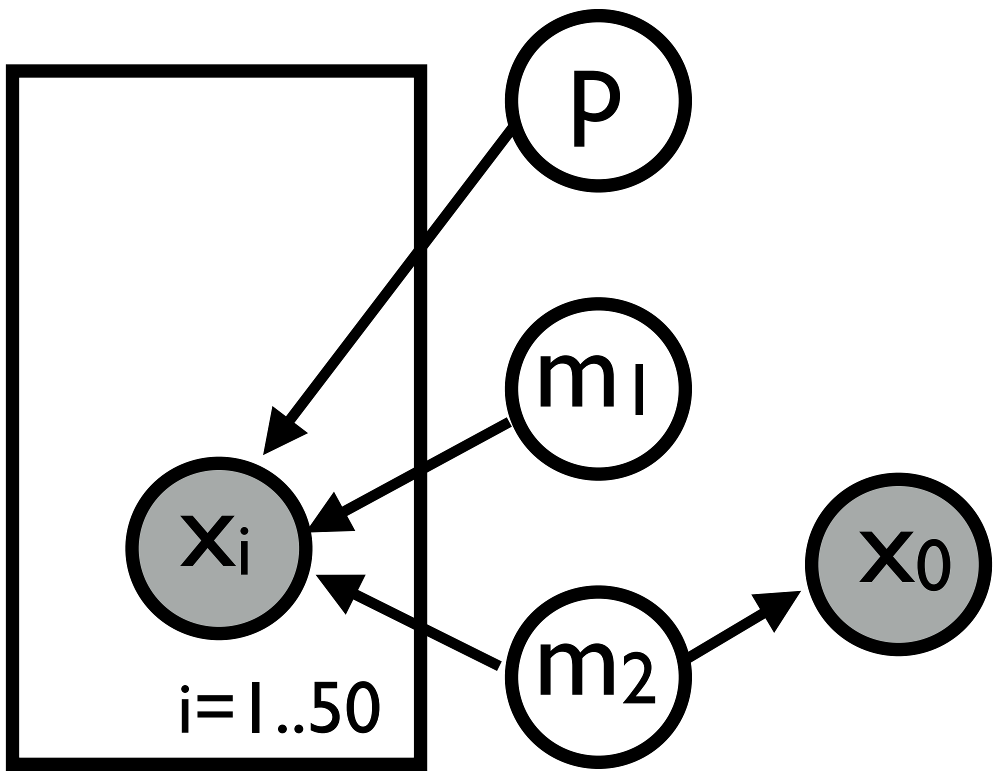
1. ERPify.
2. Move observe backwards.



Partial evaluation.

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

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P         (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (normal m2 10.) 102.3)


(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

1. ERPify.
2. Move observe backwards.

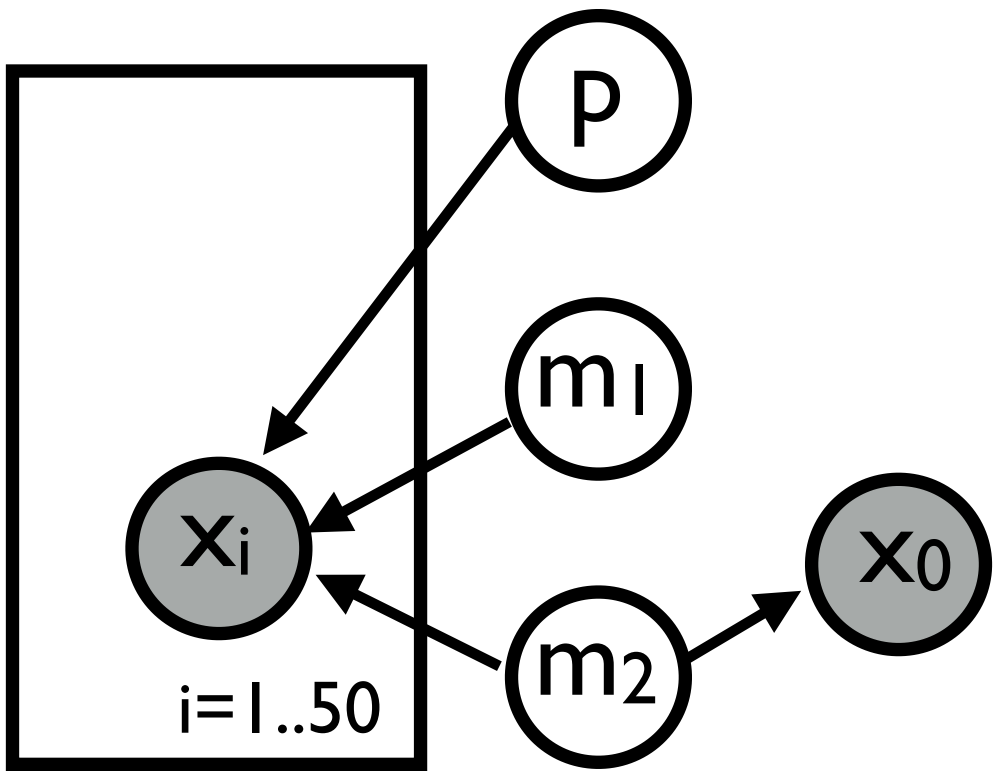



```
(assume p (sample (beta 1. 2.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P          (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (normal m2 10.) 102.3)


(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

Partial evaluation.

m2 and f are different.

1. ERPify.
2. Move observe backwards.



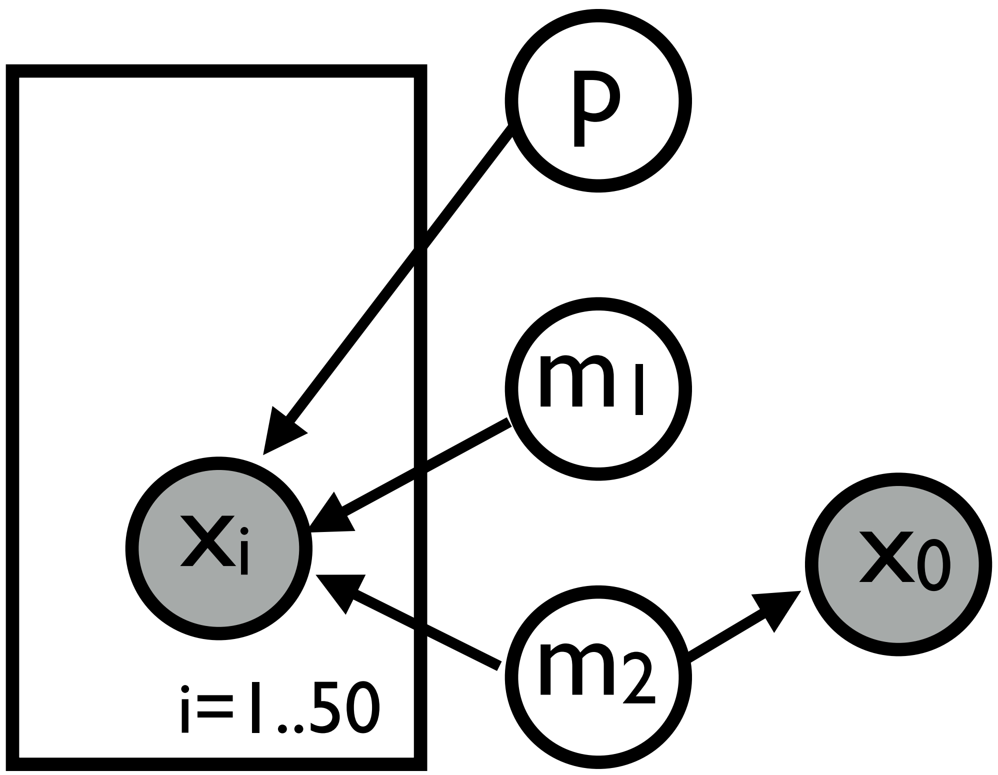

```
(assume p (sample (beta 1. 2.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))
(observe (normal m2 10.) 102.3)
(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P          (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))
```

Partial evaluation.

m2 and f are different.

```
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

1. ERPify.
2. Move observe backwards.



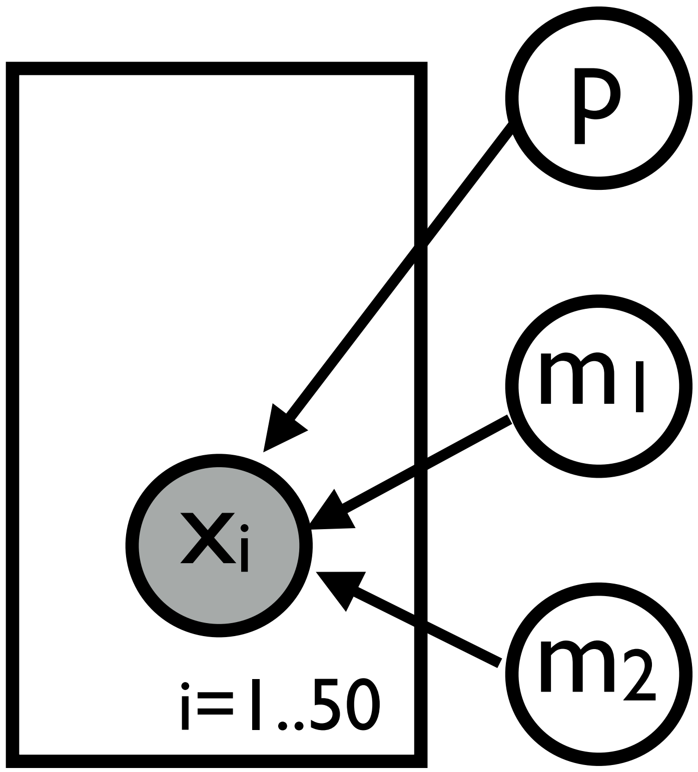

```
(assume p (sample (beta 1. 2.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))
(observe (normal m2 10.) 102.3)
(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P         (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

Partial evaluation.

m2 and f are different.

Normal-Normal conjugacy.

1. ERPify.
2. Move observe backwards.




```
(assume p (sample (beta 1. 2.)))
(assume m1 (sample (normal 10. 10.)))
(observe (normal … …) 102.3)
(assume m2 (sample (normal … …)))
(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P        (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))
```

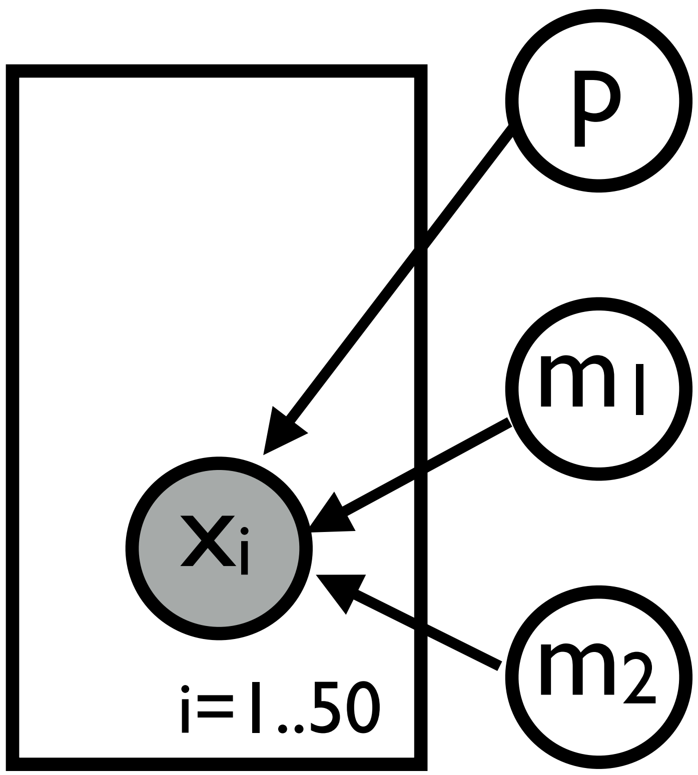
Partial evaluation.

m2 and f are different.

Normal-Normal conjugacy.

```
(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

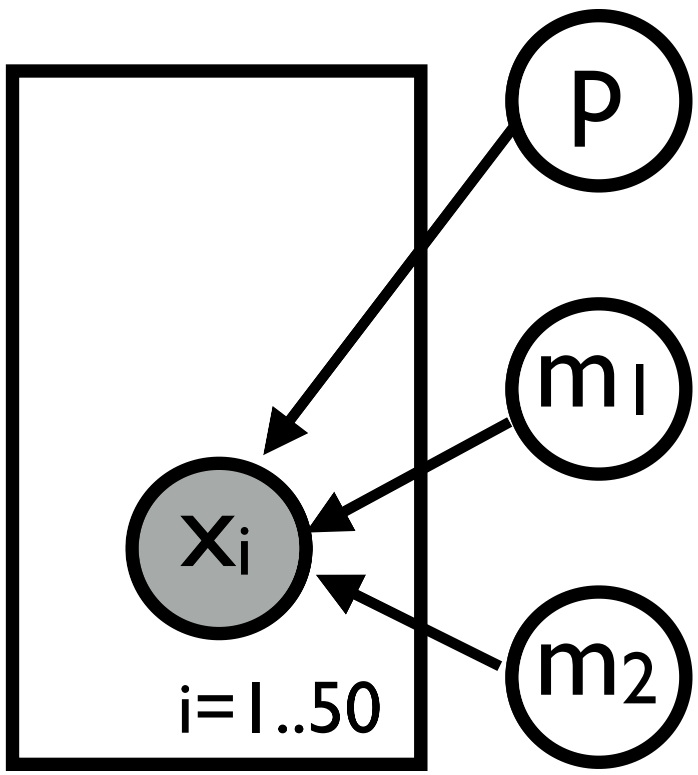
1. ERPify.
2. Move observe backwards.




```
(assume p (sample (beta 1. 2.)))
(assume m1 (sample (normal 10. 10.)))
(observe (normal … …) 102.3)
(assume m2 (sample (normal … …)))
(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P         (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

Partial evaluation.
m2 and f are different.
Normal-Normal conjugacy.
Constant importance weight.

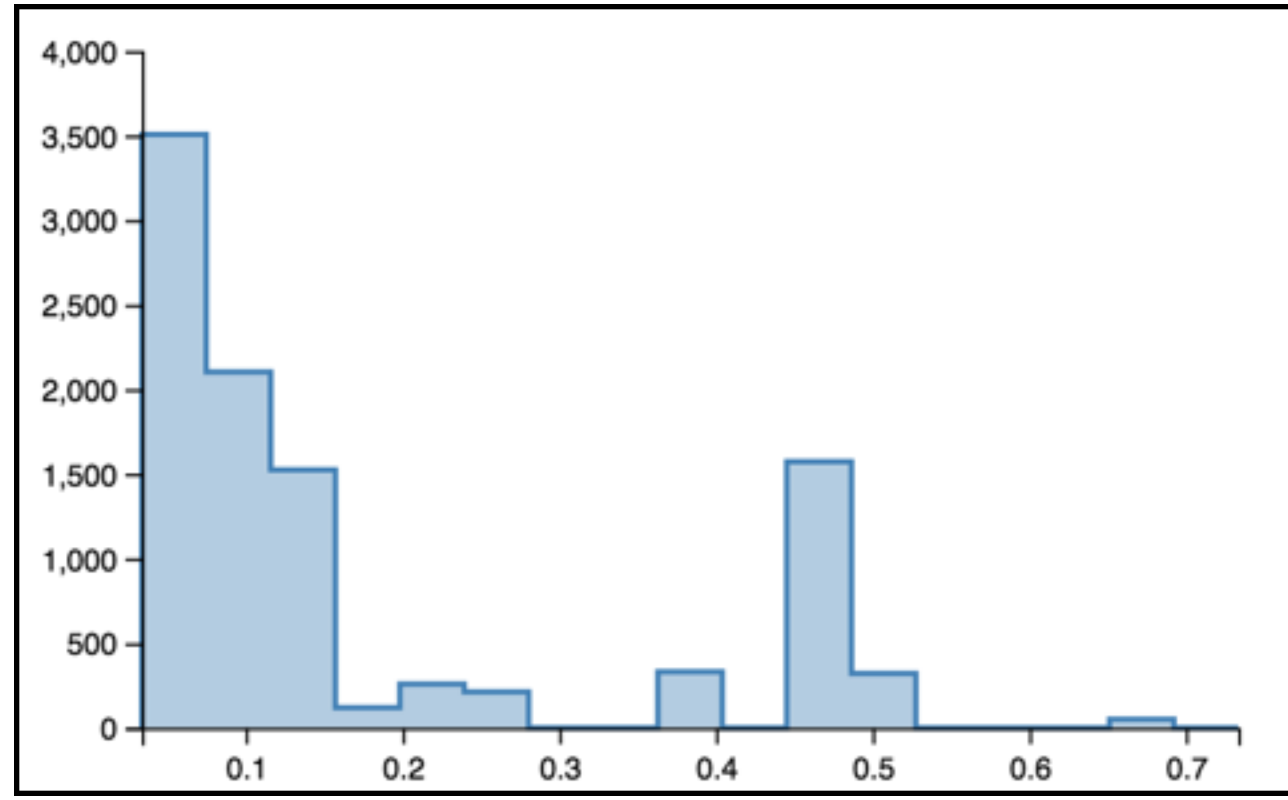
1. ERPify.
2. Move observe backwards.



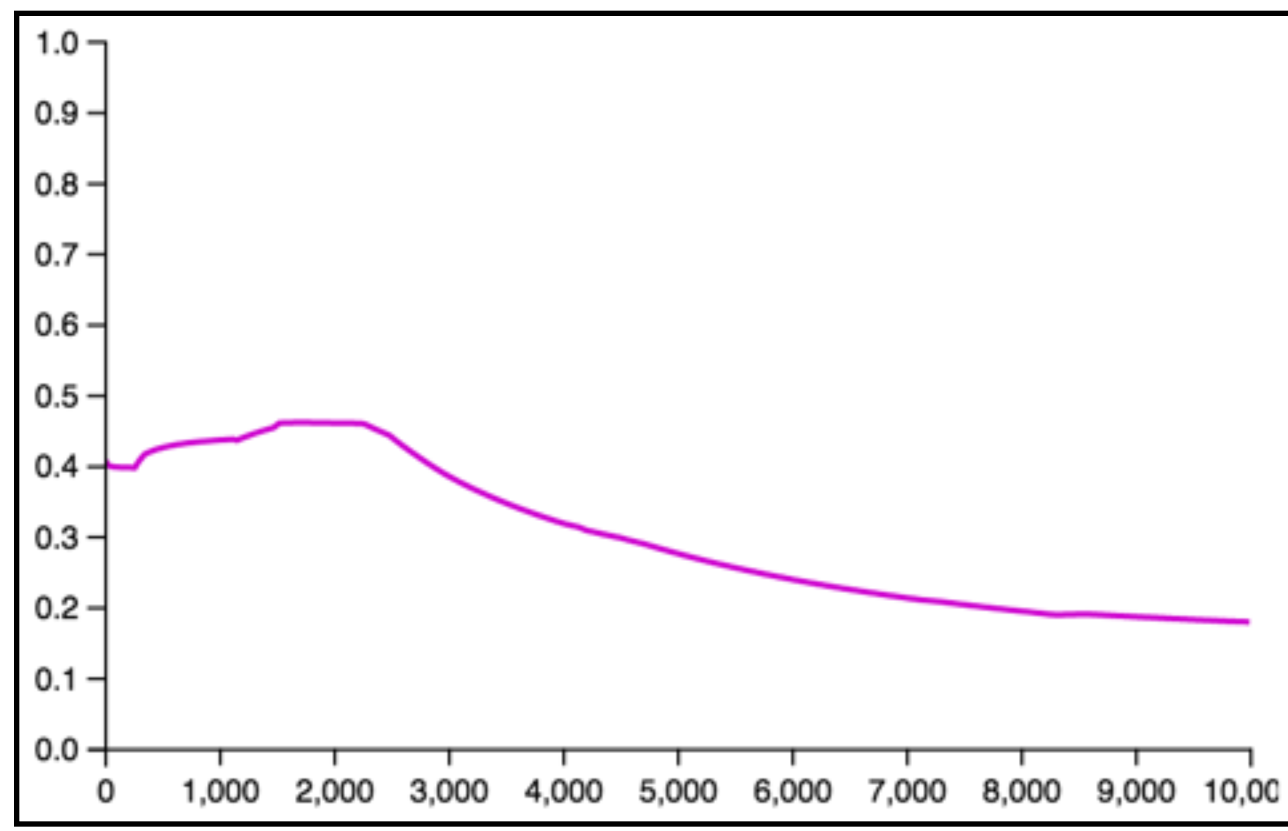

```
(assume p (sample (beta 1. 2.)))
(assume m1 (sample (normal 10. 10.)))

(assume m2 (sample (normal … …)))
(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P        (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (f p m1 m2) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```

Partial evaluation.
m2 and f are different.
Normal-Normal conjugacy.
Constant importance weight.

Gibbs-like algorithm (LHM) applied to original program

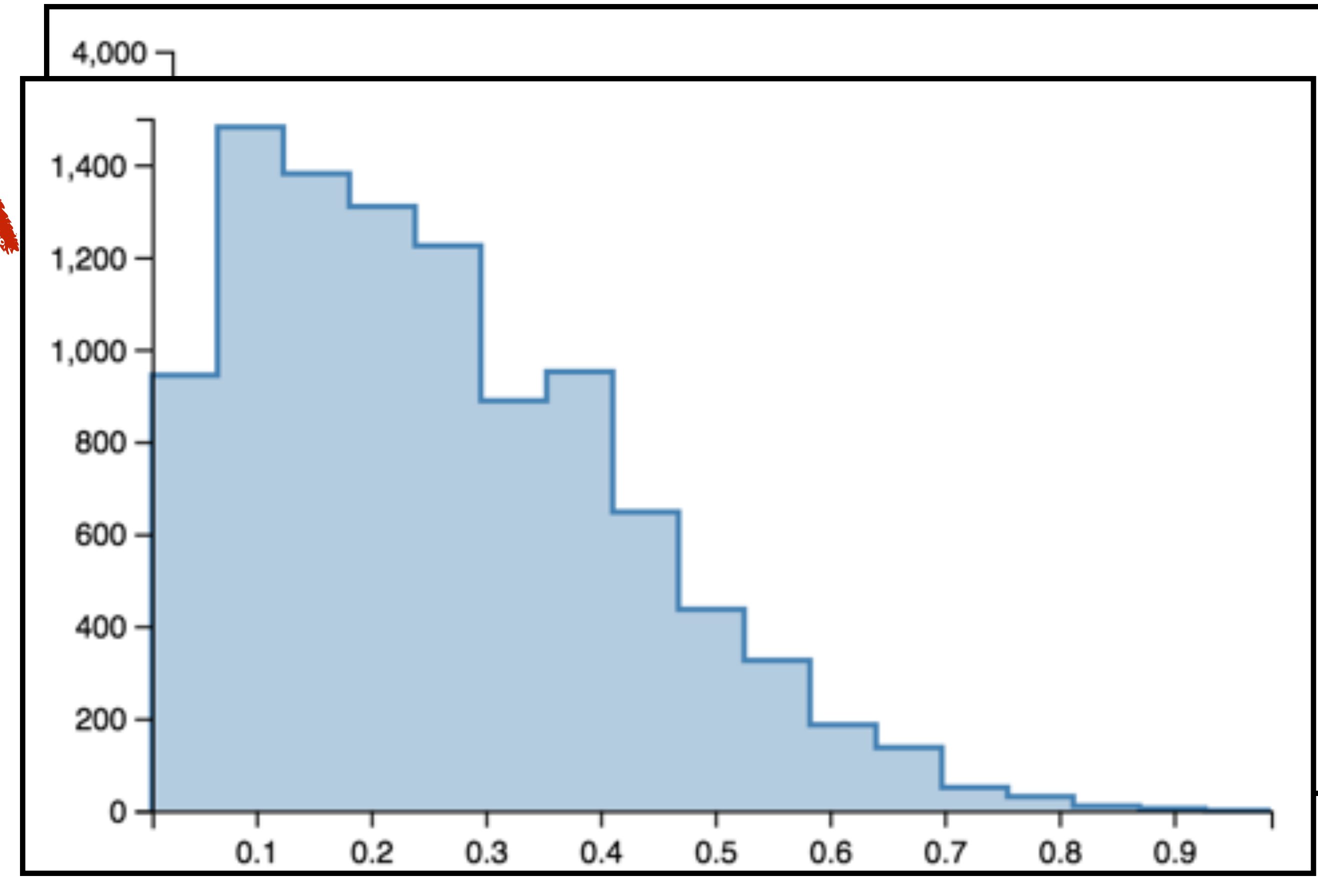


10K samples.
26 different samples.



posterior mean
of mixing prob. p

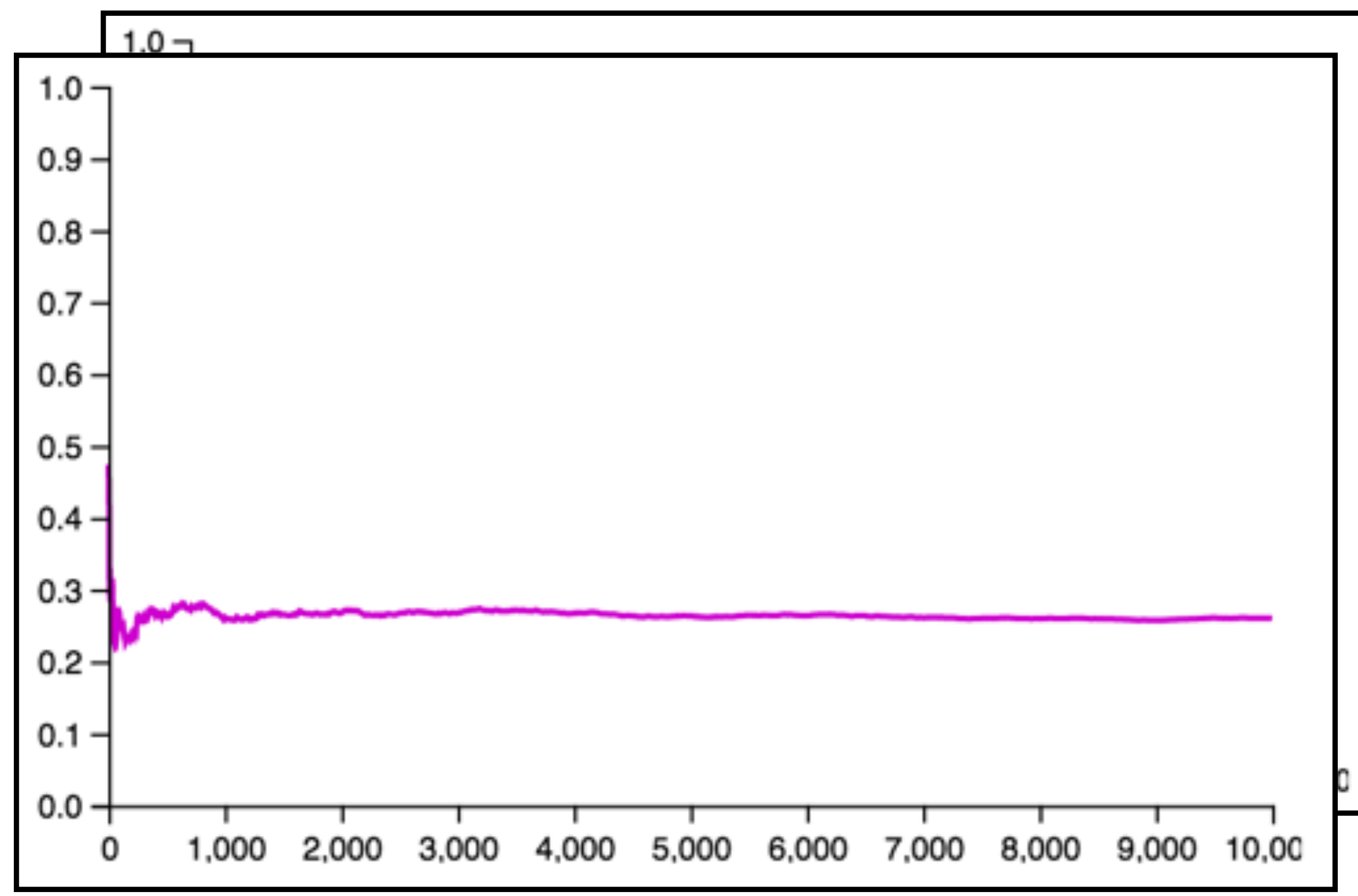
Gibbs-like algorithm (LHM) applied to ~~original program~~ transformed program



10K samples.

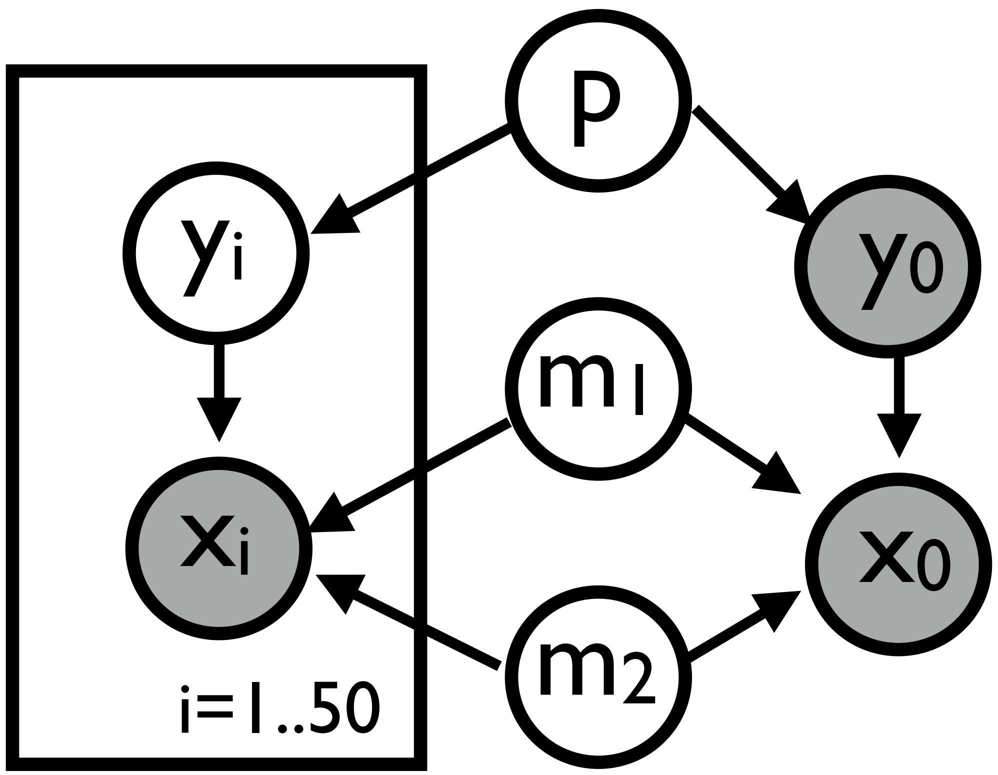
~~26~~ different samples.

2721

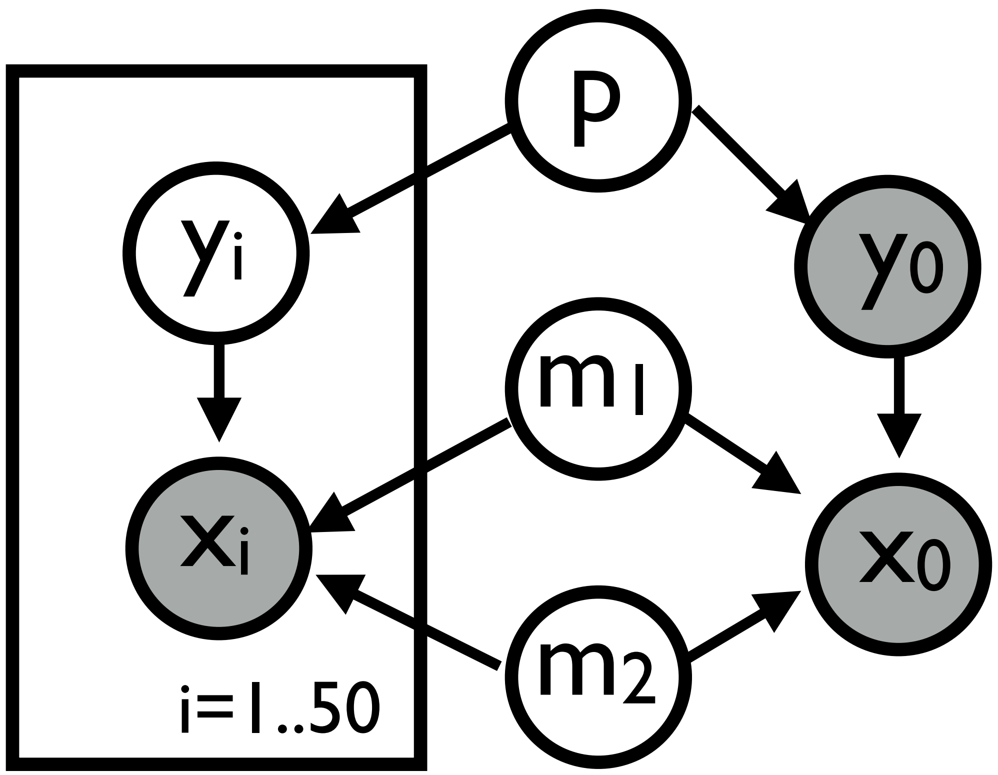


posterior mean of mixing prob. p

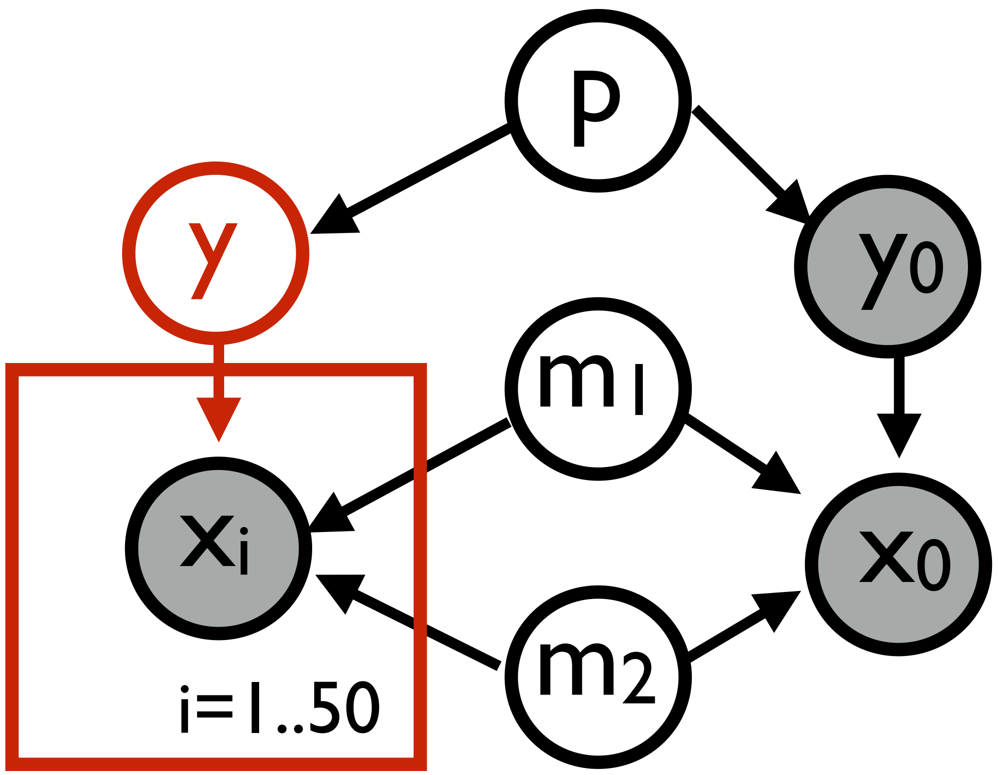
ERPification is correct sometimes, but not always.



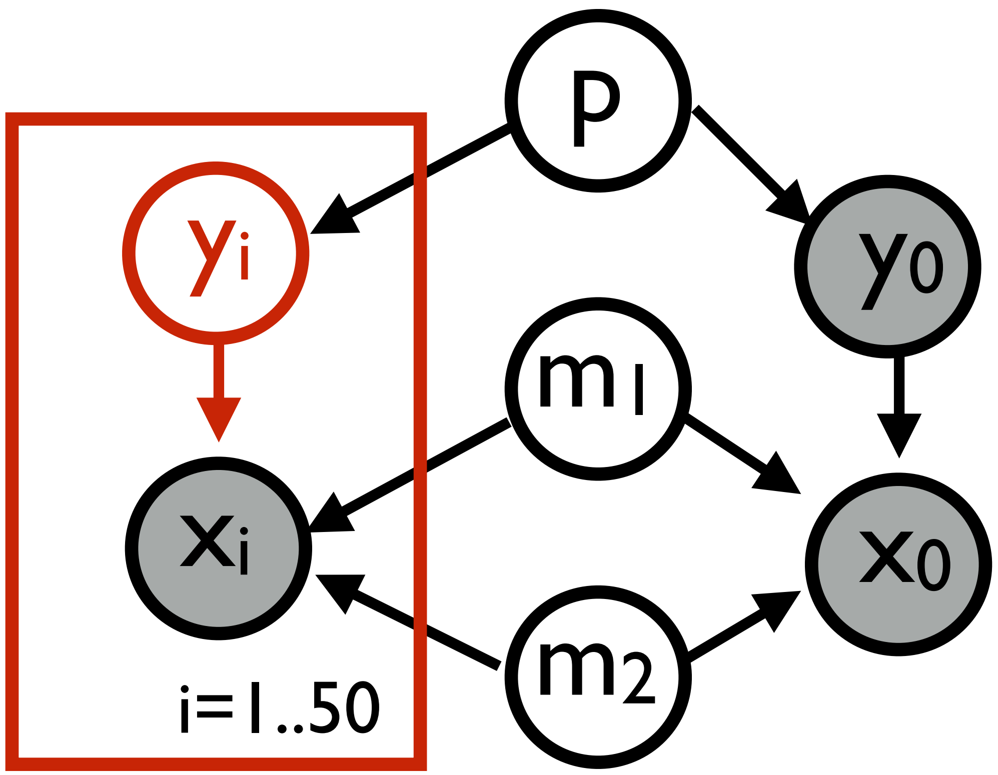

```
(assume p (sample (beta 1. 1.)))
(assume m1 (sample (normal 10. 10.)))
(assume m2 (sample (normal 10. 10.)))

(assume f (erpify-dist
  (lambda (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) x1)
…
(observe (f p m1 m2) x50)
(predict p)
```




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

(assume f (erpify-dist
  (lambda (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)
(assume dist (f p m1 m2))
(observe dist x1)
…
(observe dist x50)
(predict p)
```




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

(assume f (erpify-dist
  (lambda (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)
(assume dist (f p m1 m2))
(observe dist x1)
…
(observe dist x50)
(predict p)
```




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

(assume f
  (lambda (P M1 M2)
    (mixture-dist
      [[P        (normal M1 10.)]
       [(- 1. P) (normal M2 10.)]])))

(observe (flip p) false)
(observe (if false (normal m1 10.)
                   (normal m2 10.))
         102.3)
(assume dist (f p m1 m2))
(observe dist x1)
…
(observe dist x50)
(predict p)
```

Old

```
(lambda (P M1 M2)
  (let ((y (sample (flip P))))
    (if y (normal M1 10.)
          (normal M2 10.)))))
```

$: R \times R \times R \xrightarrow{\text{flip}} \text{dist}(R)$

New

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

$: R \times R \times R \longrightarrow \text{dist}(R)$

- Correct only if proc is called at most once.
- Need to checked by, e.g., a type system.

ERPification can fail.

Absence of density (wrt. Lebesque measure)

```
(lambda (P)
  (if (sample (flip P))
      (sample (normal 0. 1.))
      2))
```

Incompleteness of our method

```
(lambda (X)
  (+ (sample (gamma 3. 2.))
     (sample (normal X 2.))))
```

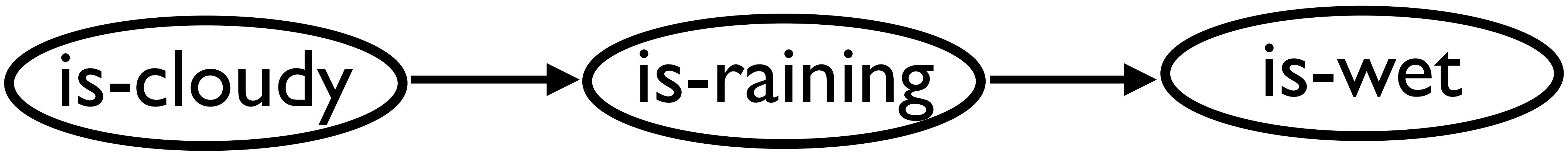
Similarity with algorithms for graphical models

- Symbolic enumeration of all outputs.
- Variable elimination.

- Moving observe backwards.
- Backward algorithm for HMM.

- Symbolic enumeration of all outputs.
- Variable elimination.

- Moving observe backwards.
- Backward algorithm for HMM.



[

```
(let [is-cloudy
        (sample (flip P))
      is-raining
        (if is-cloudy
          (sample (flip .8))
          (sample (flip .1)))
      is-wet
        (if is-raining
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```

, 1]

[
```
(let [is-raining
        (if true
          (sample (flip .8))
          (sample (flip .1))))
      is-wet
        (if is-raining
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, P]

[
```
(let [is-raining
        (if false
          (sample (flip .8))
          (sample (flip .1))))
      is-wet
        (if is-raining
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, 1-P]

[
```
(let [is-wet
        (if true
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, P*0.8]

[
```
(let [is-wet
        (if false
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, P*0.2]

[
```
(let [is-raining
        (if false
          (sample (flip .8))
          (sample (flip .1)))
      is-wet
        (if is-raining
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, 1-P]

[
```
(let [is-wet
        (if true
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, P*0.8]

[
```
(let [is-wet
        (if false
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, P*0.2]

[
```
(let [is-wet
        (if true
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, (1-P)*0.1]

[
```
(let [is-wet
        (if false
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, (1-P)*0.9]

[
```
(let [is-wet
        (if true
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, P*0.8]

[
```
(let [is-wet
        (if false
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, P*0.2]

[
```
(let [is-wet
        (if true
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, (1-P)*0.1]

[
```
(let [is-wet
        (if false
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, (1-P)*0.9]

[
```
(let [is-wet
        (if true
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, 0.1+P*0.7]

[
```
(let [is-wet
        (if false
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, P*0.2]

[
```
(let [is-wet
        (if false
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, (1-P)*0.9]

[
```
(let [is-wet
        (if true
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, $0.1+P*0.7$]

[
```
(let [is-wet
        (if false
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, $P*0.2$]

[
```
(let [is-wet
        (if false
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```
, $(1-P)*0.9$]

[
```
(let [is-wet
       (if true
         (sample (flip Q))
         (sample (flip .1)))]
  is-wet)
```
, 0.1+P*0.7]

[
```
(let [is-wet
       (if false
         (sample (flip Q))
         (sample (flip .1)))]
  is-wet)
```
, 0.9-P*0.7]

$$\left[\ \boxed{\texttt{true}}\ ,\quad Q*(0.1+P*0.7)\ \right]$$

$$\left[\ \boxed{\texttt{false}}\ ,\quad (1-Q)*(0.1+P*0.7)\ \right]$$

[

```
(let [is-wet
        (if false
          (sample (flip Q))
          (sample (flip .1)))]
  is-wet)
```

, 0.9-P*0.7]

$$[\ \boxed{\texttt{true}}\ ,\ Q*(0.1+P*0.7)\ ]$$

$$[\ \boxed{\texttt{false}}\ ,\ (1-Q)*(0.1+P*0.7)\ ]$$

$$[\ \boxed{\texttt{true}}\ ,\ 0.1*(0.9-P*0.7)\ ]$$

$$[\ \boxed{\texttt{false}}\ ,\ 0.9*(0.9-P*0.7)\ ]$$

$$\left[\ \boxed{\texttt{true}}\ ,\quad \begin{array}{l} 0.1*Q\ -\ 0.07*P \\ +\ 0.7*PQ\ +\ 0.09 \end{array} \right]$$

$$\left[\ \boxed{\texttt{false}}\ ,\quad (1-Q)*(0.1+P*0.7) \right]$$

$$\left[\ \boxed{\texttt{false}}\ ,\quad 0.9*(0.9-P*0.7) \right]$$

$$\left[\ \boxed{\texttt{true}}\ ,\ \begin{array}{l} 0.1*Q - 0.07*P \\ + 0.7*PQ + 0.09 \end{array}\ \right]$$

$$\left[\ \boxed{\texttt{false}}\ ,\ \begin{array}{l} -0.1*Q + 0.07*P \\ - 0.7*PQ + 0.91 \end{array}\ \right]$$