Probabilistic Programming Languages

Build Your Owl PPL

Guillaume Baudart
Christine Tasson

MPRI 2021-2022

Reminders

BYO-PPL

Probabilistic Programming

Probabilistic Programming

Programming and reasoning with uncertainty

- Sample from probability distributions
- Condition on observed data

Probabilistic Programming

Programming and reasoning with uncertainty

- Sample from probability distributions
- Condition on observed data

Bayesian Inference: learn parameters from data

- Latent parameter θ
- Observed data $x_1, \ldots, x_n$

Thomas Bayes (1701-1761)

https://en.wikipedia.org/wiki/Thomas_Bayes

Probabilistic Programming

Programming and reasoning with uncertainty

- Sample from probability distributions
- Condition on observed data

Bayesian Inference: learn parameters from data

- Latent parameter θ
- Observed data $x_1, \ldots, x_n$

$$p(\theta \mid x_1, \ldots x_n) = \frac{p(\theta)\ p(x_1, \ldots, x_n \mid \theta)}{p(x_1, \ldots, x_n)} \qquad \text{(Bayes' theorem)}$$

$$\propto p(\theta)\ p(x_1, \ldots, x_n \mid \theta) \qquad \text{(Data are constants)}$$

Thomas Bayes (1701-1761)

https://en.wikipedia.org/wiki/Thomas_Bayes

Probabilistic Programming

Programming and reasoning with uncertainty

- Sample from probability distributions
- Condition on observed data

Bayesian Inference: learn parameters from data

- Latent parameter θ
- Observed data $x_1, \ldots, x_n$

$$\boxed{p(\theta \mid x_1, \ldots x_n)} = \frac{p(\theta)\ p(x_1, \ldots, x_n \mid \theta)}{p(x_1, \ldots, x_n)} \qquad \text{(Bayes' theorem)}$$

posterior

$$\propto p(\theta)\ p(x_1, \ldots, x_n \mid \theta) \qquad \text{(Data are constants)}$$

Thomas Bayes (1701-1761)

https://en.wikipedia.org/wiki/Thomas_Bayes

Probabilistic Programming

Programming and reasoning with uncertainty

- Sample from probability distributions
- Condition on observed data

Bayesian Inference: learn parameters from data

- Latent parameter θ
- Observed data $x_1, \ldots, x_n$

$$\underbrace{p(\theta \mid x_1, \ldots x_n)}_{\textit{posterior}} = \frac{p(\theta)\ p(x_1, \ldots, x_n \mid \theta)}{p(x_1, \ldots, x_n)} \qquad \text{(Bayes' theorem)}$$

$$\propto \underbrace{p(\theta)}_{\textit{prior}}\ p(x_1, \ldots, x_n \mid \theta) \qquad \text{(Data are constants)}$$

Thomas Bayes (1701-1761)

https://en.wikipedia.org/wiki/Thomas_Bayes

Probabilistic Programming

Programming and reasoning with uncertainty

- Sample from probability distributions
- Condition on observed data

Bayesian Inference: learn parameters from data

- Latent parameter θ
- Observed data $x_1, \ldots, x_n$

$$p(\theta \mid x_1, \ldots x_n) = \frac{p(\theta)\ p(x_1, \ldots, x_n \mid \theta)}{p(x_1, \ldots, x_n)} \qquad \text{(Bayes' theorem)}$$

posterior

$$\propto p(\theta)\ p(x_1, \ldots, x_n \mid \theta) \qquad \text{(Data are constants)}$$

prior *likelihood*

Thomas Bayes (1701-1761)

https://en.wikipedia.org/wiki/Thomas_Bayes

Example: Coin

Consider a series of coin tosses

- Observations: head or tail
- Each toss is independant
- What is the probability of getting head at the next toss?

Probabilistic model

- Prior: $z \sim Uniform(0,1)$
- Observations: $\forall i \in [1, n] \, . \; x_i \sim Bernoulli(z)$
- Posterior: $p(z \,|\, x_1, x_2, \ldots, x_n)$?

Example: Coin

Consider a series of coin tosses

- Observations: head or tail
- Each toss is independant
- What is the probability of getting head at the next toss?

Probabilistic model

- Prior: $z \sim Uniform(0,1)$
- Observations: $\forall i \in [1, n] \,.\, x_i \sim Bernoulli(z)$
- Posterior: $p(z \mid x_1, x_2, \ldots, x_n)$?

$$p(z \mid x_1, \ldots, x_n) = \frac{p(x_1, \ldots, x_n \mid z) p(z)}{p(x_1, \ldots, x_n)} = \frac{p(x_1, \ldots, x_n \mid z) p(z)}{\int_z p(x_1, \ldots, x_n \mid z)}$$

$$\begin{aligned} p(x_1, \ldots, x_n \mid z) &= \prod_{i=1}^{n} p(x_i \mid z) \\ &= \prod_{i=1}^{n} z^{x_i} (1-z)^{1-x_i} \\ &= z^{\sum_{i=1}^{n} x_1} (1-z)^{\sum_{i=1}^{n} (1-x_i)} \\ &= z^{\text{#heads}} (1-z)^{\text{#tails}} \end{aligned}$$

$$\begin{aligned} p(z \mid x_1, \ldots, x_n) &= \frac{z^{\text{#heads}} (1-z)^{\text{#tails}}}{\int_z z^{\text{#heads}} (1-z)^{\text{#tails}}} \\ &= \frac{z^{\text{#heads}} (1-z)^{\text{#tails}}}{B(\text{#heads}+1, \text{#tails}+1)} \\ &= Beta_{\text{pdf}}(\text{#heads}+1, \text{#tails}+1) \end{aligned}$$

Example: Coin

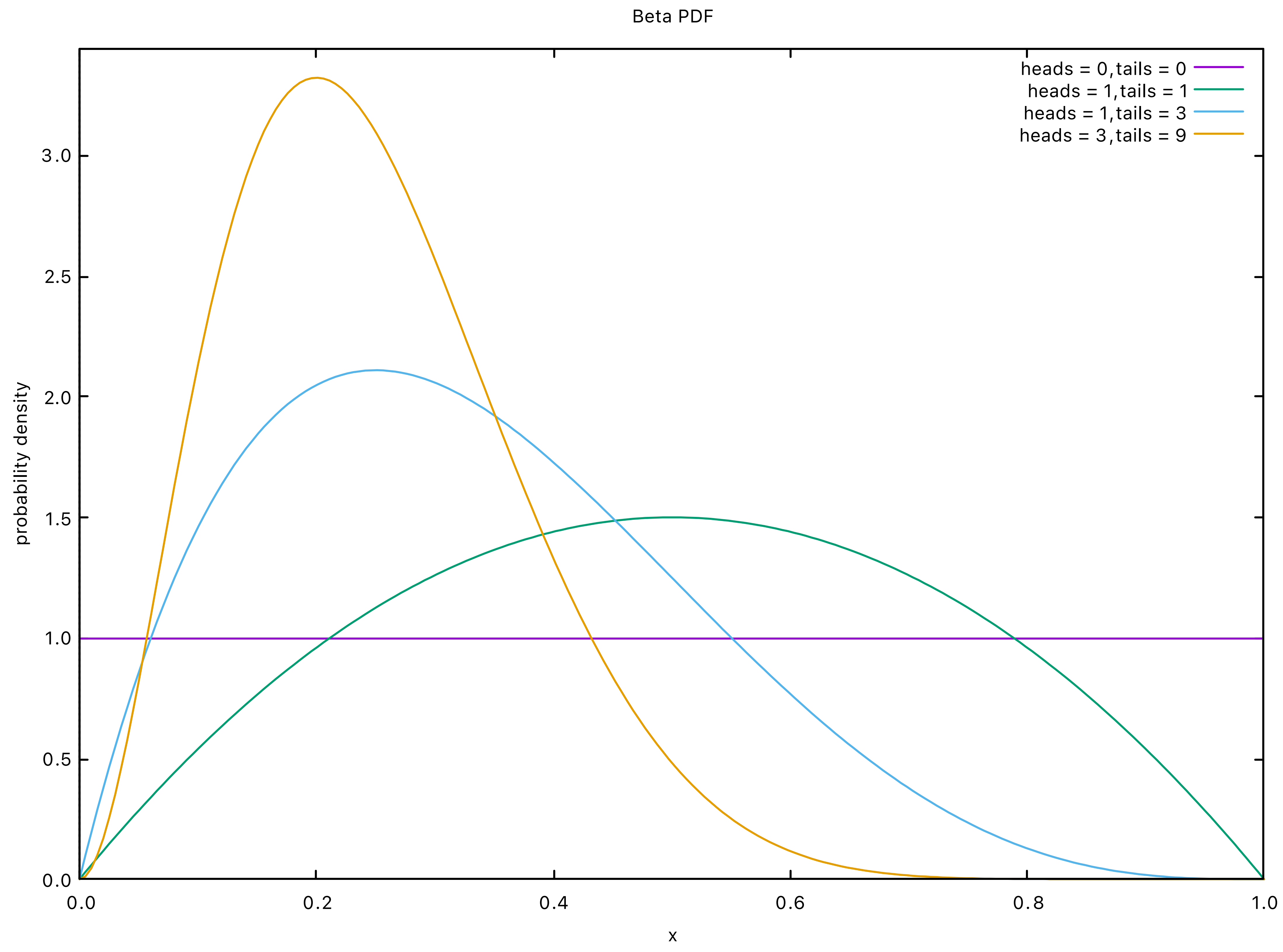
Consider a series of coin tosses

- Observations: head or tail
- Each toss is independant
- What is the probability of getting head at the next toss?

Probabilistic model

- Prior: $z \sim Uniform(0,1)$
- Observations: $\forall i \in [1, n] \,.\, x_i \sim Bernoulli(z)$
- Posterior: $p(z \mid x_1, x_2, \ldots, x_n)$?

$z \sim Beta(\#\text{heads} + 1,\ \#\text{tails} + 1)$



Example: Coin

Consider a series of coin tosses

- Observations: head or tail
- Each toss is independant
- What is the probability of getting head at the next toss?

Probabilistic model

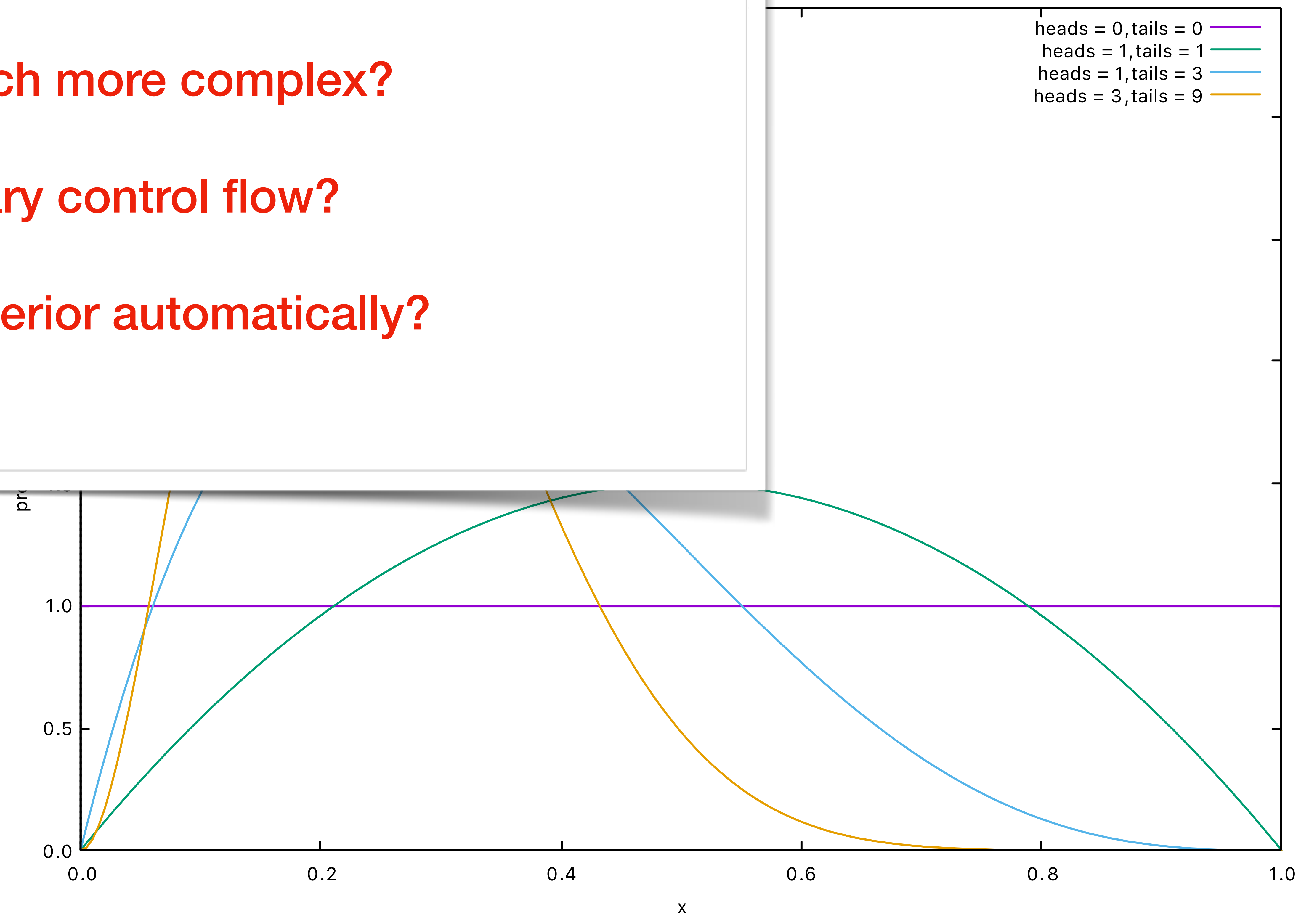
- Prior: $z \sim Uniforn$
- Observations: $\forall i \in$
- Posterior: $p(z \mid x_1, x$

What if the model is much more complex?

What if we use arbitrary control flow?

Can we compute the posterior automatically?

$z \sim Beta(\text{#heads} + 1, \text{#tails} + 1)$



Probabilistic Programming Languages

Probabilistic Programming Languages

General purpose programming languages extended with probabilistic constructs

- `sample`: draw a sample from a distribution
- `assume`, `factor`, `observe`: condition the model on inputs (e.g., observed data)
- `infer`: compute the posterior distribution of a model given the inputs

Probabilistic Programming Languages

General purpose programming languages extended with probabilistic constructs

- `sample`: draw a sample from a distribution
- `assume`, `factor`, `observe`: condition the model on inputs (e.g., observed data)
- `infer`: compute the posterior distribution of a model given the inputs

Multiple examples:

- Church, Anglican (lisp, clojure), 2008
- WebPPL (javascript), 2014
- Pyro/NumPyro (python), 2017/2019
- Gen (julia), 2018
- ProbZelus (Zelus), 2019
- ...

Probabilistic Programming Languages

General purpose programming languages extended with probabilistic constructs

- `sample`: draw a sample from a distribution
- `assume`, `factor`, `observe`: condition the model on inputs (e.g., observed data)
- `infer`: compute the posterior distribution of a model given the inputs

Multiple examples:

- Church, Anglican (lisp, clojure), 2008
- WebPPL (javascript), 2014
- Pyro/NumPyro (python), 2017/2019
- Gen (julia), 2018
- ProbZelus (Zelus), 2019
- ...

More and more, incorporating new ideas:

- New inference techniques, e.g., stochastic variational inference (SVI)
- Interaction with neural nets (deep probabilistic programming)

Build Your Owl PPL

BYO-PPL

Simplified Syntax

```
x ::= variables
c ::= constants

d ::= let x = e | d d
p ::= x | (p, p)
e ::= c | x | (e, e) | op(e) | e e
    | if e then e else e | let p = e in e | fun p → e
    | sample e | assume e | factor e | observe e e | infer e
```

A first-order functional programming language extended with probabilistic constructs

Outline

For a given inference algoritm, how to implement `sample`, `assume`, `factor`, `observe`, and `infer`?

I - Basic inference

- Rejection sampling
- Importance sampling

II - Continuation Passing Style models

III - Inference on CPS models

- Sample generation
- Importance sampling
- Particle filter

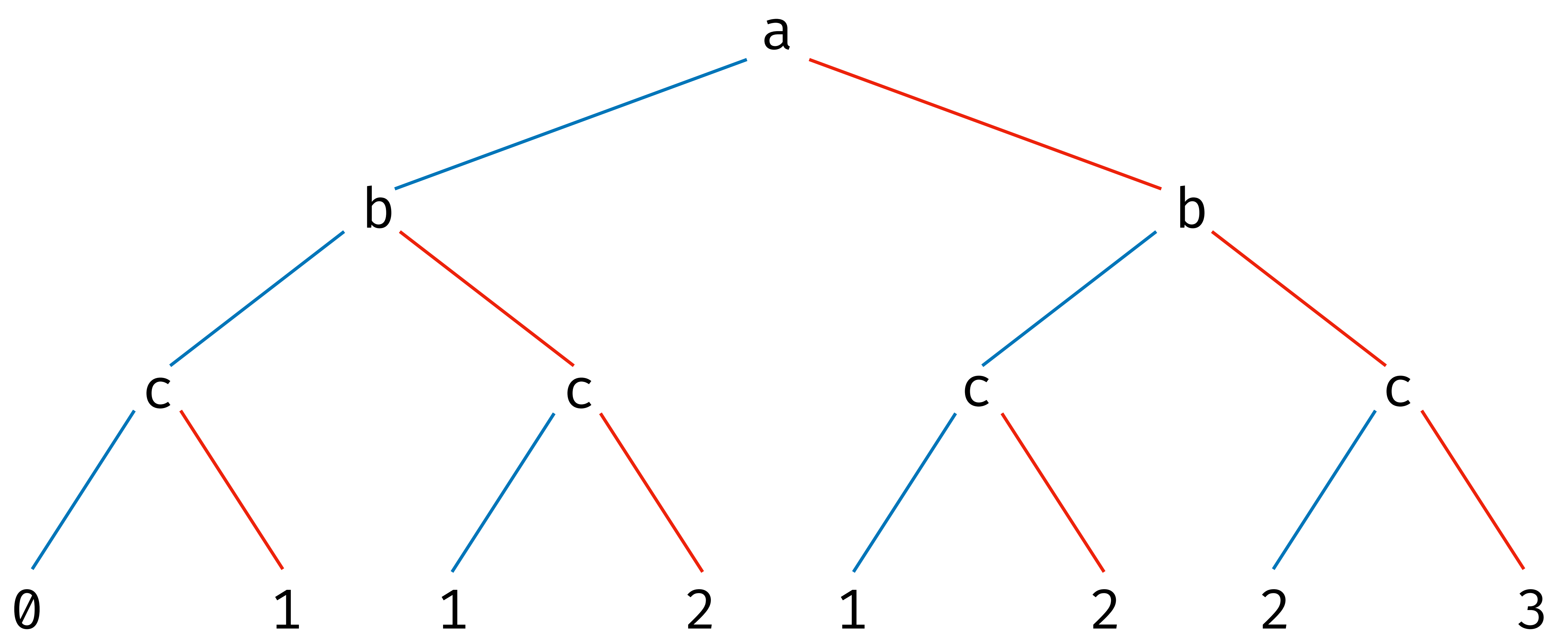
Warm-up : Rejection Sampling

BYO-PPL

Funny Bernoulli

funny_bernoulli.ml

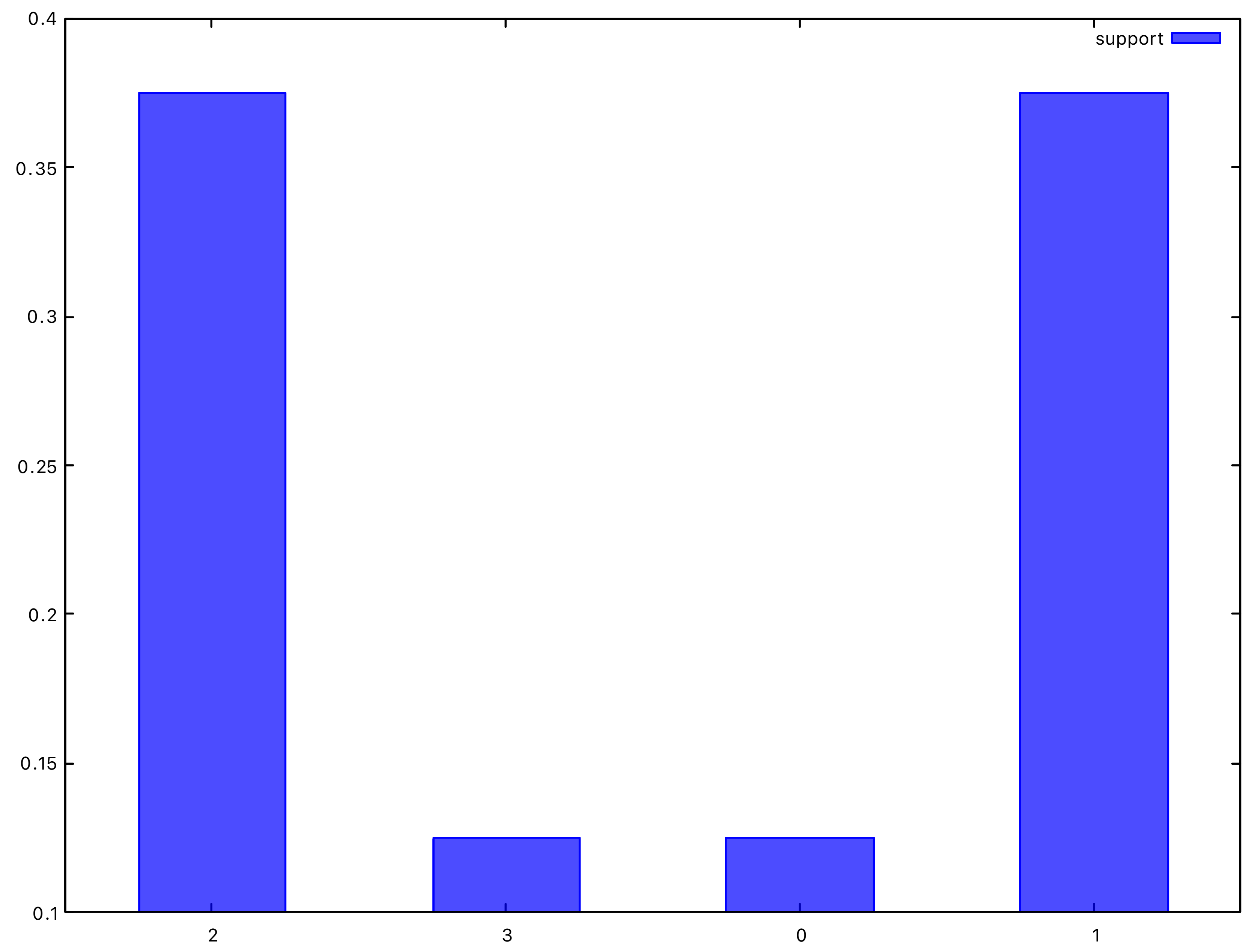
```
let funny_bernoulli () =
  let a = sample (bernoulli ~p:0.5) in
  let b = sample (bernoulli ~p:0.5) in
  let c = sample (bernoulli ~p:0.5) in
  a + b + c
```

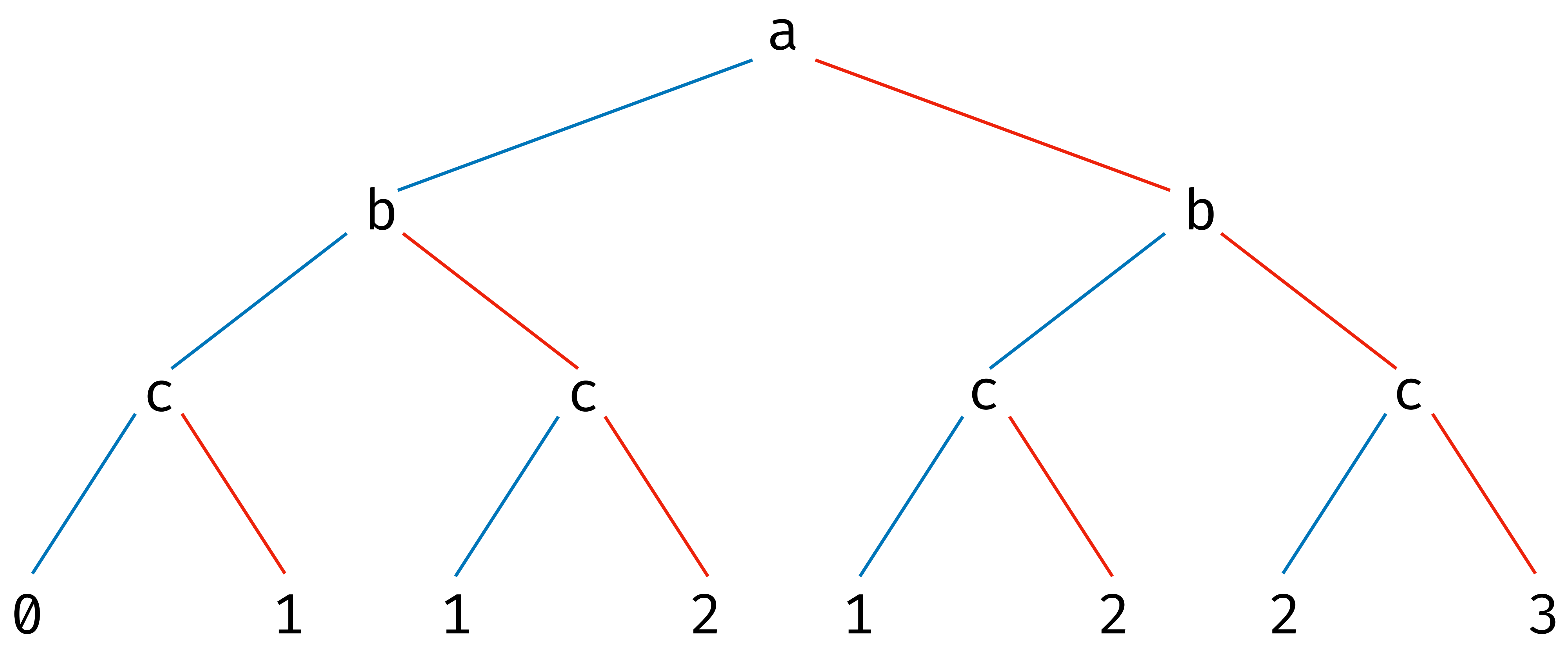


Funny Bernoulli

funny_bernoulli.ml

```
let funny_bernoulli () =
  let a = sample (bernoulli ~p:0.5) in
  let b = sample (bernoulli ~p:0.5) in
  let c = sample (bernoulli ~p:0.5) in
  a + b + c
```

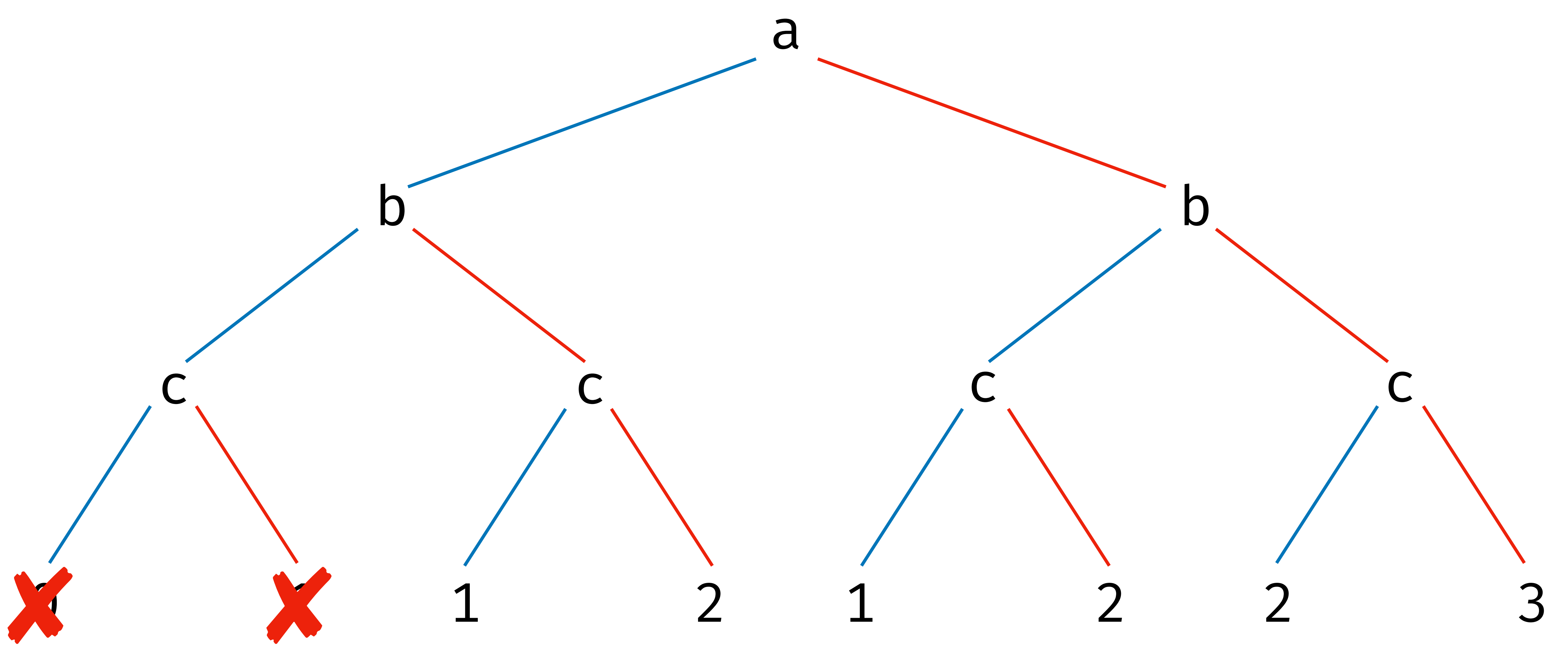




Funny Bernoulli

funny_bernoulli.ml

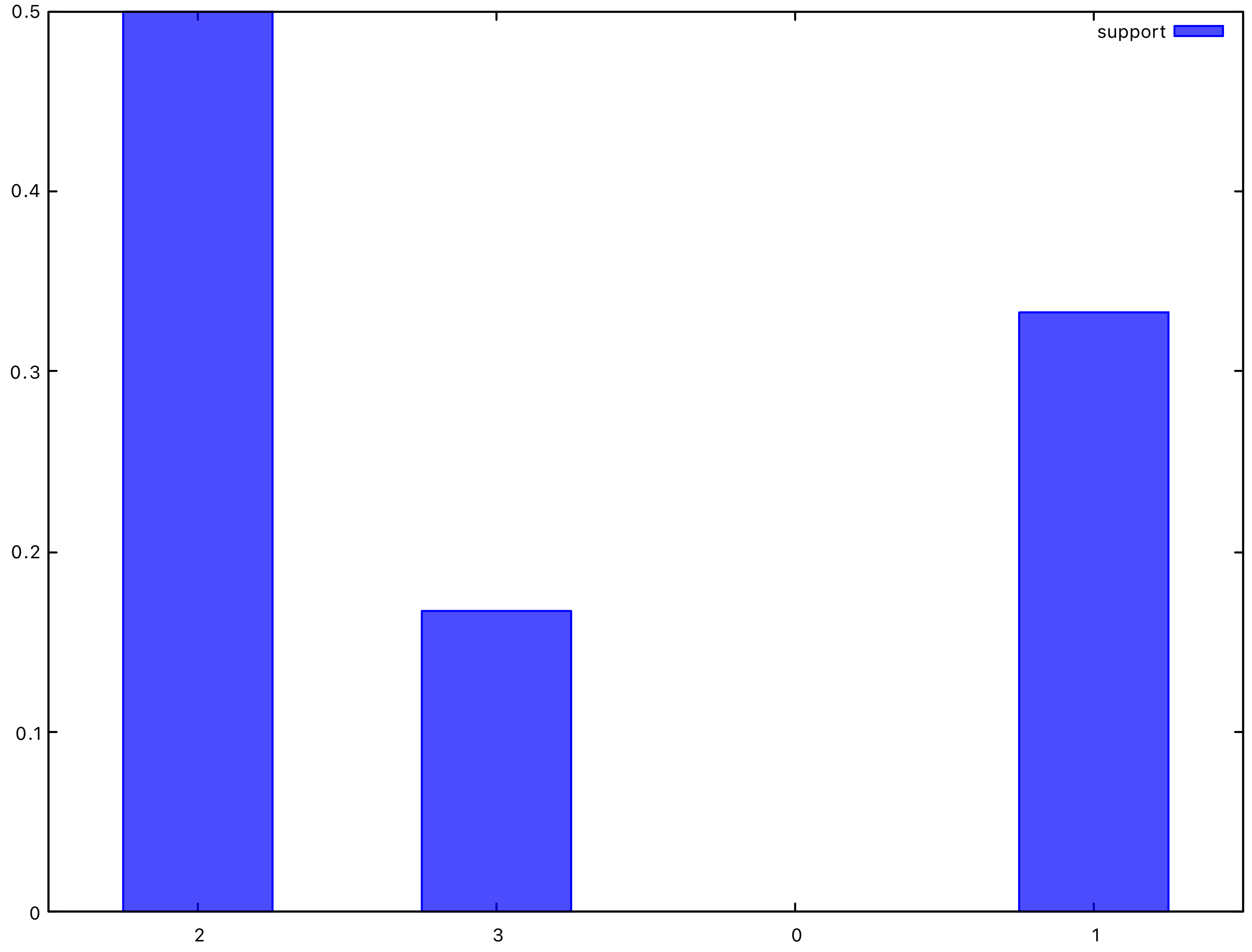
```
let funny_bernoulli () =
  let a = sample (bernoulli ~p:0.5) in
  let b = sample (bernoulli ~p:0.5) in
  let c = sample (bernoulli ~p:0.5) in
  let () = assume (a = 1 || b = 1) in
  a + b + c
```

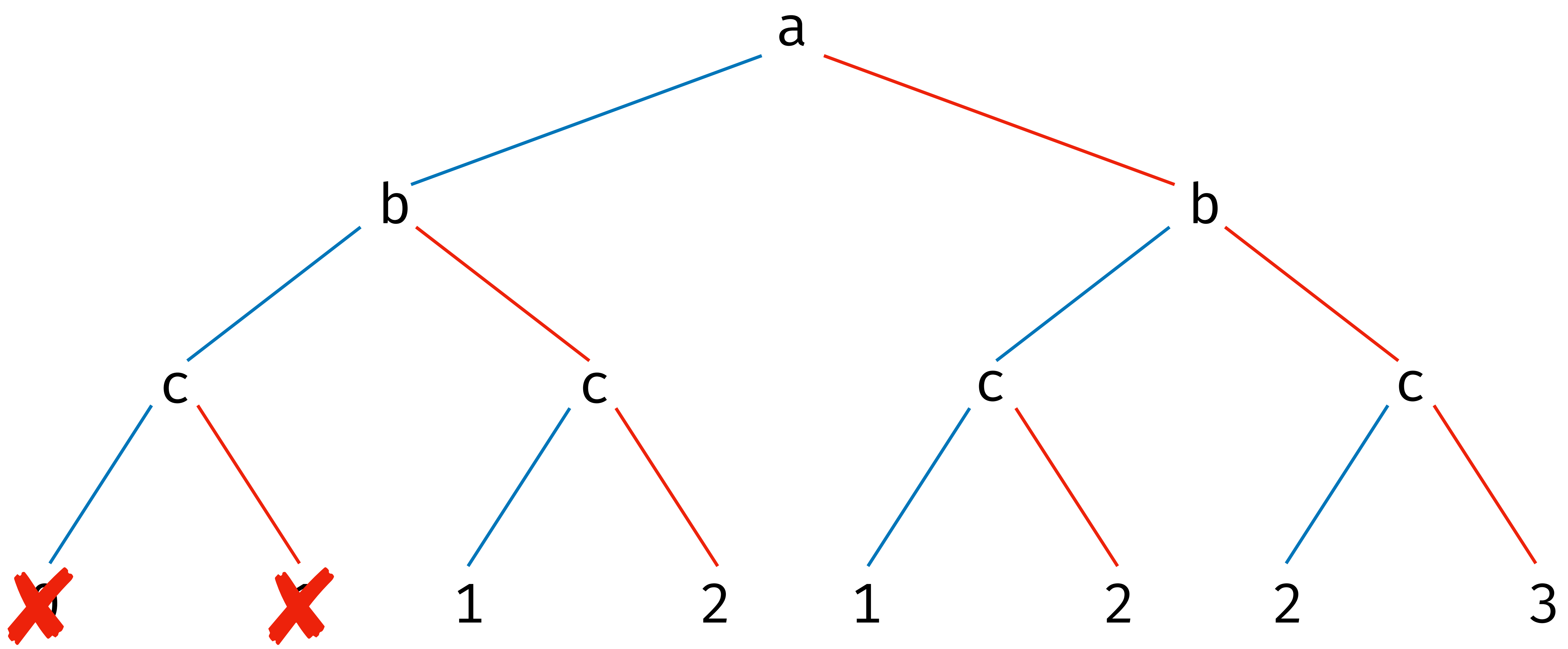


Funny Bernoulli

funny_bernoulli.ml

```
let funny_bernoulli () =
  let a = sample (bernoulli ~p:0.5) in
  let b = sample (bernoulli ~p:0.5) in
  let c = sample (bernoulli ~p:0.5) in
  let () = assume (a = 1 || b = 1) in
  a + b + c
```





Rejection Sampling


```
module Rejection_sampling : sig
  val sample : 'a Distribution.t → 'a
  val assume : bool → unit
  val infer : ?n:int → ('a → 'b) → 'a → 'b Distribution.t
end = struct ... end
```

Inference algorithm

- Run the model to get a sample
- `sample` : draw a value from a distribution
- `assume` : accept / reject a sample
- If a sample is rejected, re-run the model to get another sample

Rejection Sampling

basic.ml

```
module Rejection_sampling = struct

  let sample d = assert false
  let assume p = assert false

  let infer ?(n = 1000) model obs = assert false
end
```

Rejection Sampling

basic.ml

```
module Rejection_sampling = struct
  exception Reject

  let sample d = Distribution.draw d
  let assume p = if not p then raise Reject

  let infer ?(n = 1000) model obs =
    let rec exec i = try model obs with Reject → exec i in
    let values = Array.init n exec in
    Distribution.uniform_support ~values
end
```

The type `prob` trick

```
module Rejection_sampling : sig
  type prob
  val sample : prob → 'a Distribution.t → 'a
  val assume : prob → bool → unit
  val infer : ?n:int → (prob → 'a → 'b) → 'a → 'b Distribution.t
end = struct ... end
```

Forbid the use of probabilistic construct outside a model

- Define a simple abstract type `prob`
- Probabilistic constructs and models all require an argument of type `prob`
- Such a value can only be build by `infer`

Rejection Sampling

basic.ml

```
module Rejection_sampling = struct
  type prob = Prob

  exception Reject

  let sample _prob d = Distribution.draw d
  let assume _prob p = if not p then raise Reject

  let infer ?(n = 1000) model obs =
    let rec exec i = try model Prob obs with Reject → exec i in
    let values = Array.init n exec in
    Distribution.uniform_support ~values
end
```

Funny Bernoulli

funny_bernoulli.ml

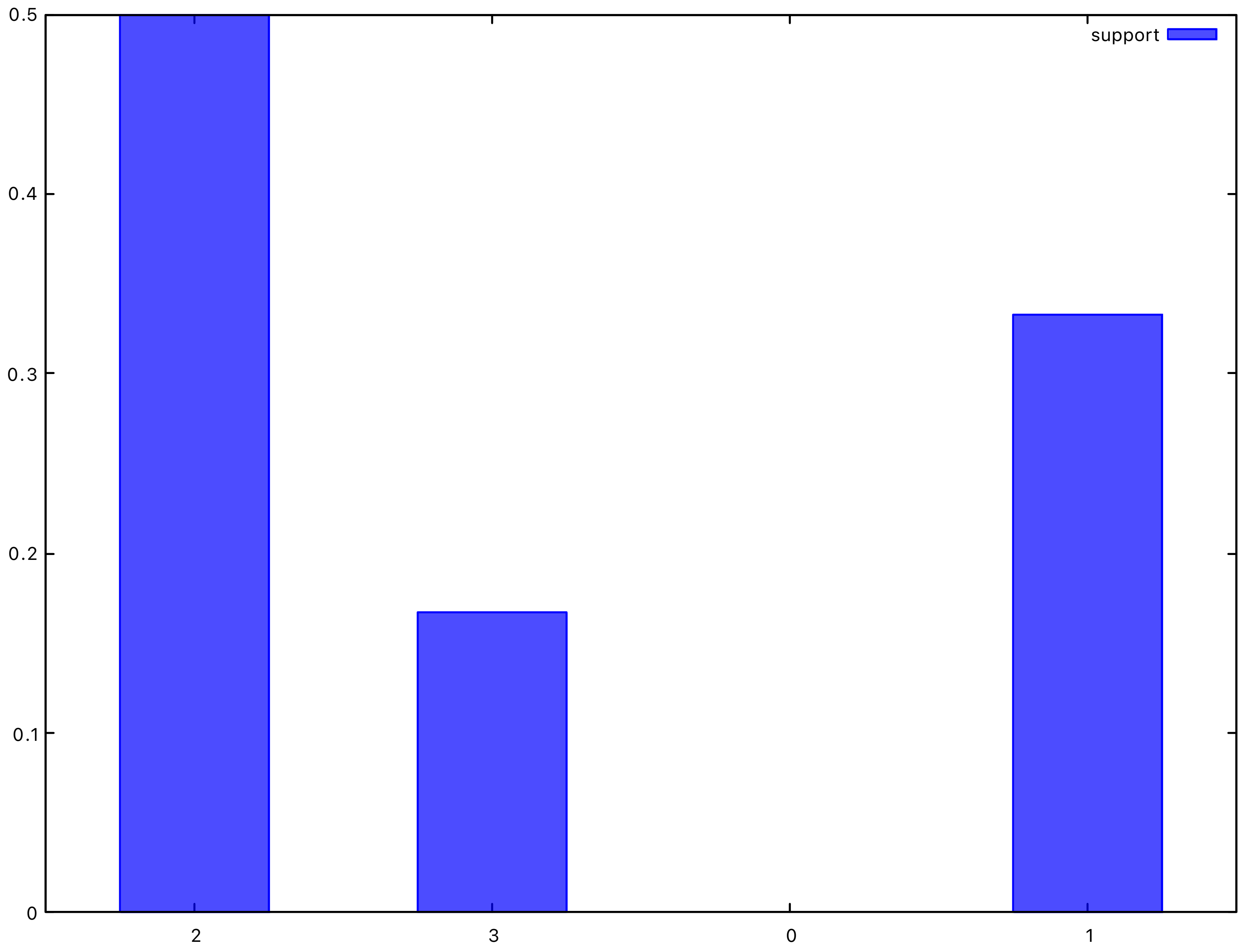
```
open Byoppl
open Distribution
open Basic.Rejection_sampling

let funny_bernoulli prob () =
  let a = sample prob (bernoulli ~p:0.5) in
  let b = sample prob (bernoulli ~p:0.5) in
  let c = sample prob (bernoulli ~p:0.5) in
  let () = assume prob (a = 1 || b = 1) in
  a + b + c

let _ =
  let dist = infer funny_bernoulli () in
  let { values; probs; _ } = get_support ~shrink:true dist in
  Array.iteri (fun i x -> Format.printf "%d %f@." x probs.(i)) values
```

```
❯ dune exec ./examples/funny_bernoulli.exe
```

Funny Bernoulli

funny_bernoulli.ml

```
open Byoppl
open Distribution
open Basic.Rejection_sampling

let funny_bernoulli prob () =
  let a = sample prob (bernoulli ~p:0.5) in
  let b = sample prob (bernoulli ~p:0.5) in
  let c = sample prob (bernoulli ~p:0.5) in
  let () = assume prob (a = 1 || b = 1) in
  a + b + c

let _ =
  let dist = infer funny_bernoulli () in
  let { values; probs; _ } = get_support ~shrink:true dist in
  Array.iteri (fun i x → Format.printf "%d %f@." x probs.(i)) values
```




```
❯ dune exec ./examples/funny_bernoulli.exe
```

Importance Sampling

BYO-PPL

Laplace and Gender Bias

laplace.ml

```
open Basic.Rejection_sampling

let laplace prob () =
  let p = sample prob (uniform ~a:0. ~b:1.) in
  let g = sample prob (binomial ~p ~n:493_472) in
  let () = assume prob (g = 241_945) in
  p

let _ =
  let dist = infer ~n:1000 laplace () in
  let m, s = Distribution.stats dist in
  Format.printf "Gender bias, mean:%f std:%f@." m s
```

```
let () = observe prob
  (binomial ~p ~n:493_472) 241_945
```

```
❯ dune exec ./examples/laplace.exe
```

Laplace and Gender Bias

laplace.ml

```
open Basic.Rejection_sampling

let laplace prob () =
  let p = sample prob (uniform ~a:0. ~b:1.) in
  let g = sample prob (binomial ~p ~n:493_472) in
  let () = assume prob (g = 241_945) in
  p

let _ =
  let dist = infer ~n:1000 laplace () in
  let m, s = Distribution.stats dist in
  Format.printf "Gender bias, mean:%f std:%f@." m s
```

$\longmapsto$

```
let () = observe prob
  (binomial ~p ~n:493_472) 241_945
```

```
❯ dune exec ./examples/laplace.exe
```

Never terminate!

Coin

coin.ml

```
open Basic.Rejection_sampling

let coin prob x =
  let z = sample prob (uniform ~a:0. ~b:1.) in
  let () = List.iter (observe prob (bernoulli ~p:z)) x in
  z

let _ =
  let dist = infer coin [1; 1; 0; 0; 0; 0; 0; 0; 0; 0] in
  let m, s = Distribution.stats dist in
  Format.printf "Coin bias, mean:%f std:%f@." m s
```

```
❯ dune exec ./examples/coin.exe

Coin bias, mean:0.246161, std:0.119687
```

Coin

coin.ml

```
open Basic.Rejection_sampling

let coin prob x =
  let z = sample prob (uniform ~a:0. ~b:1.) in
  let () = List.iter (observe prob (bernoulli ~p:z)) x in
  z

let _ =
  let dist = infer coin [1; 1; 0; 0; 0; 0; 0; 0; 0; 0] in
  let m, s = Distribution.stats dist in
  Format.printf "Coin bias, mean:%f std:%f@." m s
```

```
❯ dune exec ./examples/coin.exe

Coin bias, mean:0.246161, std:0.119687
```

Slow!

Importance Sampling

basic.ml

```
module Importance_sampling : sig
  type prob
  val sample : prob → 'a Distribution.t → 'a
  val factor : prob → float → unit
  val infer : ?n:int → (prob → 'a → 'b) → 'a → 'b Distribution.t
end = struct ... end
```

Inference algorithm

- Run a set of n independent executions
- `sample`: draw a sample from a distribution
- `factor`: associate a score to the current execution
- Gather output values and score to approximate the posterior distribution

Likelihood weighting

- `observe d x := factor (logpdf d x)`
- `assume p := factor (if p then 0. else -.infinity)`

Importance Sampling

basic.ml

```
module Importance_sampling = struct
  type prob = ...

  let sample prob d = assert false
  let factor prob s = assert false
  let observe prob d x = factor prob (Distribution.logpdf d x)
  let assume prob p = factor prob (if p then 0. else -. infinity)

  let infer ?(n = 1000) model obs = assert false
end
```

Importance Sampling

basic.ml

```
module Importance_sampling = struct
  type prob = { id : int; scores : float array }

  let sample _prob d = Distribution.draw d
  let factor prob s = prob.scores.(prob.id) <- prob.scores.(prob.id) +. s
  let observe prob d x = factor prob (Distribution.logpdf d x)
  let assume prob p = factor prob (if p then 0. else -. infinity)

  let infer ?(n = 1000) model obs =
    let scores = Array.make n 0. in
    let values = Array.mapi (fun i _ -> model { id = i; scores } obs) scores in
    Distribution.support ~values ~logits:scores
end
```

Coin

coin.ml

```
open Basic.Importance_sampling

let coin prob x =
  let z = sample prob (uniform ~a:0. ~b:1.) in
  let () = List.iter (observe prob (bernoulli ~p:z)) x in
  z

let _ =
  let dist = infer coin [1; 1; 0; 0; 0; 0; 0; 0; 0; 0] in
  let m, s = Distribution.stats dist in
  Format.printf "Coin bias, mean:%f, std:%f@." m s
```

```
❯ dune exec ./examples/coin.exe

Coin bias, mean:0.247876, std:0.118921
Beta(2+1, 8+1), mean:0.250000, std:0.120096
```

Particle Filter

BYO-PPL

HMM: Hidden Markov Model

Track the position of an agent from noisy observations

- The current position should not be too far from the previous position
- The observations should not be too far from the current position

Probabilistic model: $\forall t \in \mathbb{N}$.

- $x_t \sim \mathcal{N}(x_{t-1}, \text{ speed})$
- $y_t \sim \mathcal{N}(x_t, \text{ noise})$

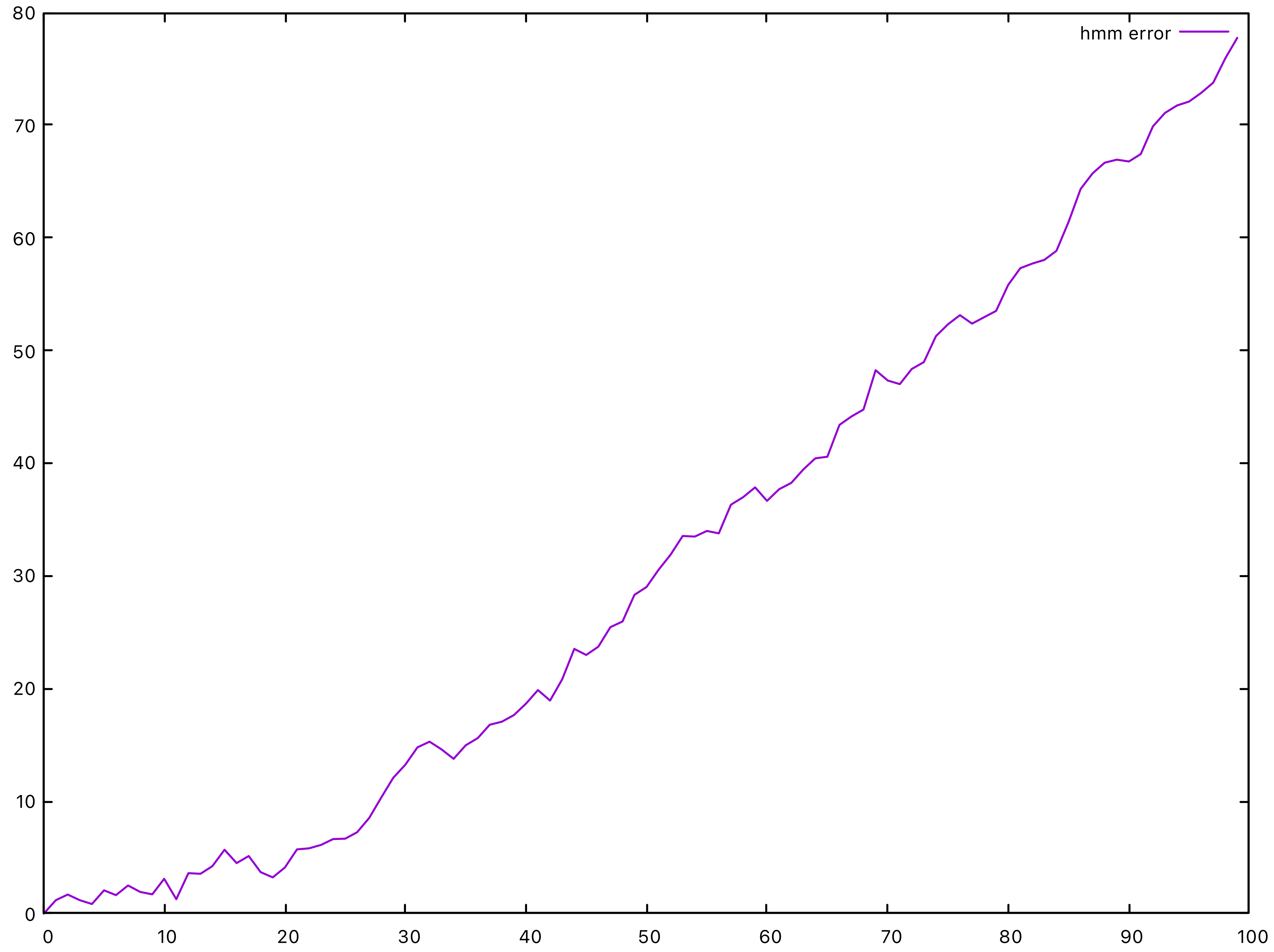
HMM: Hidden Markov Model

hmm.ml

```
open Basic.Importance_sampling

let hmm prob data =
  let rec gen states data =
    match (states, data) with
    | [], y :: data -> gen [ y ] data
    | states, [] -> states
    | pre_x :: _, y :: data ->
        let x = sample prob (gaussian ~mu:pre_x ~sigma:1.0) in
        let () = observe prob (gaussian ~mu:x ~sigma:1.0) y in
        gen (x :: states) data
  in
  gen [] data

let _ =
  let data = Owl.Arr.linspace 0. 20. 20 |> Owl.Arr.to_array |> Array.to_list in
  let dist = Distribution.split_list (infer hmm data) in
  let m_x = List.rev (List.map Distribution.mean dist) in
  List.iter2 (Format.printf "%f >> %f@.") data m_x
```

HMM: Hidden Markov Model

```
❯ dune exec ./examples/hmm.exe

0.000000 >> 0.000000
1.052632 >> 0.278989
2.105263 >> 2.923428
3.157895 >> 2.812035
4.210526 >> 2.328341
5.263158 >> 1.742109
6.315789 >> 2.518105
7.368421 >> 3.958375
8.421053 >> 5.946233
9.473684 >> 7.329554
10.526316 >> 9.293653
11.578947 >> 10.181831
12.631579 >> 8.549409
13.684211 >> 9.323073
14.736842 >> 9.280692
15.789474 >> 9.352218
...
```



HMM: Importance Sampling

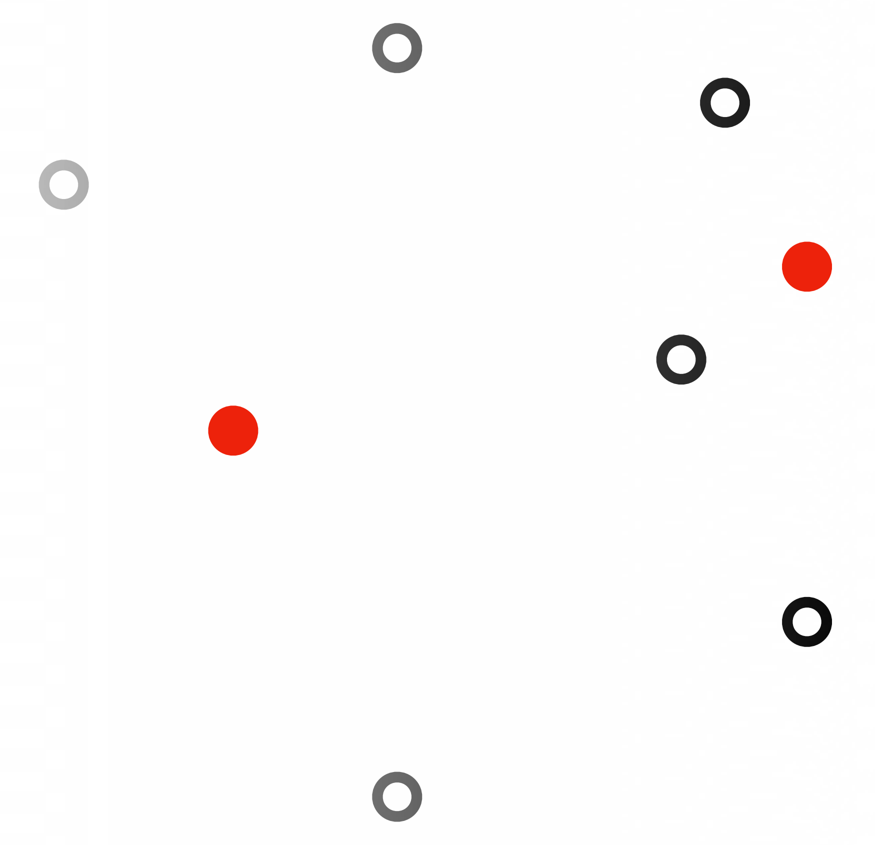
Problem:

- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement

HMM: Importance Sampling

Problem:

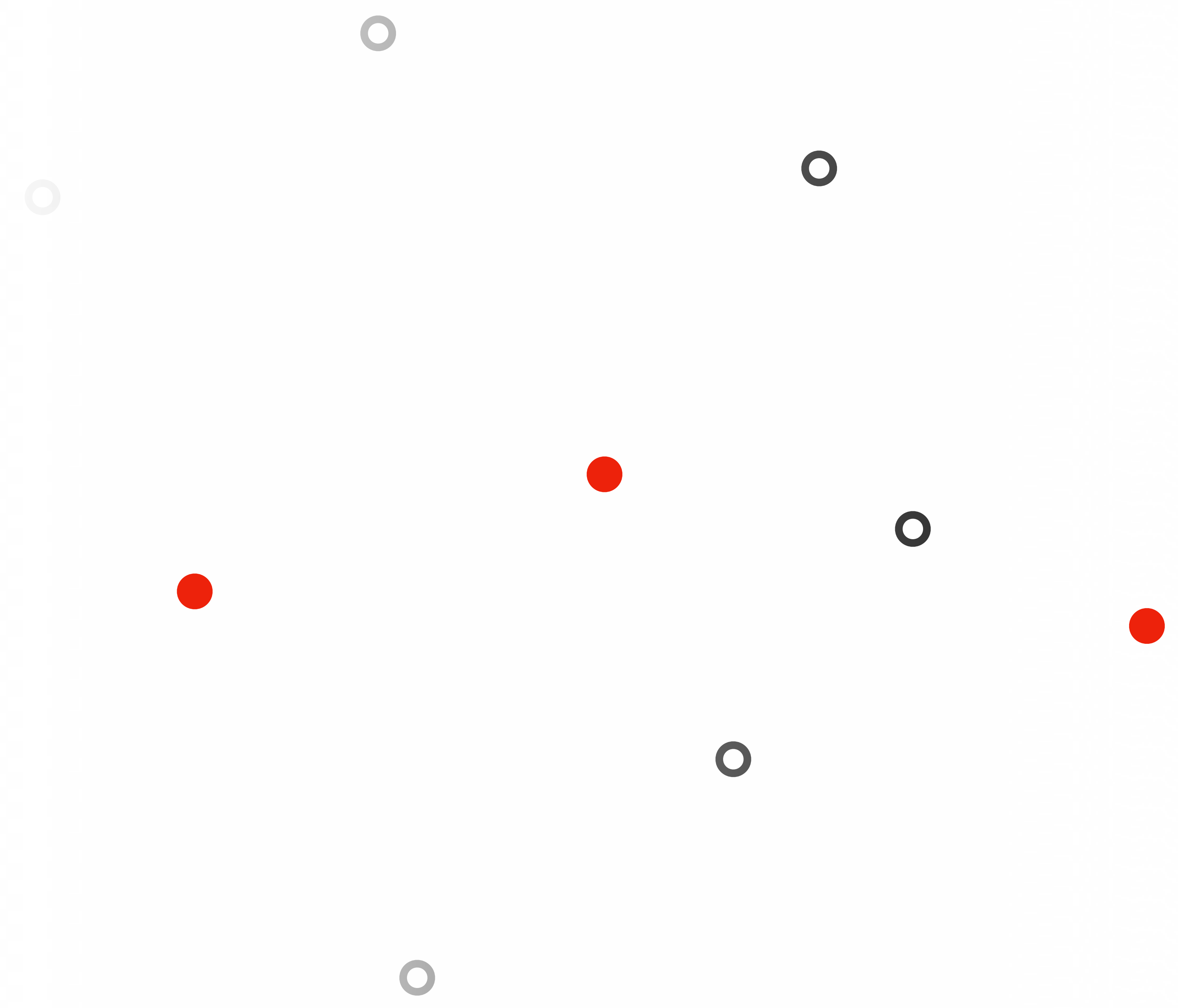
- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



HMM: Importance Sampling

Problem:

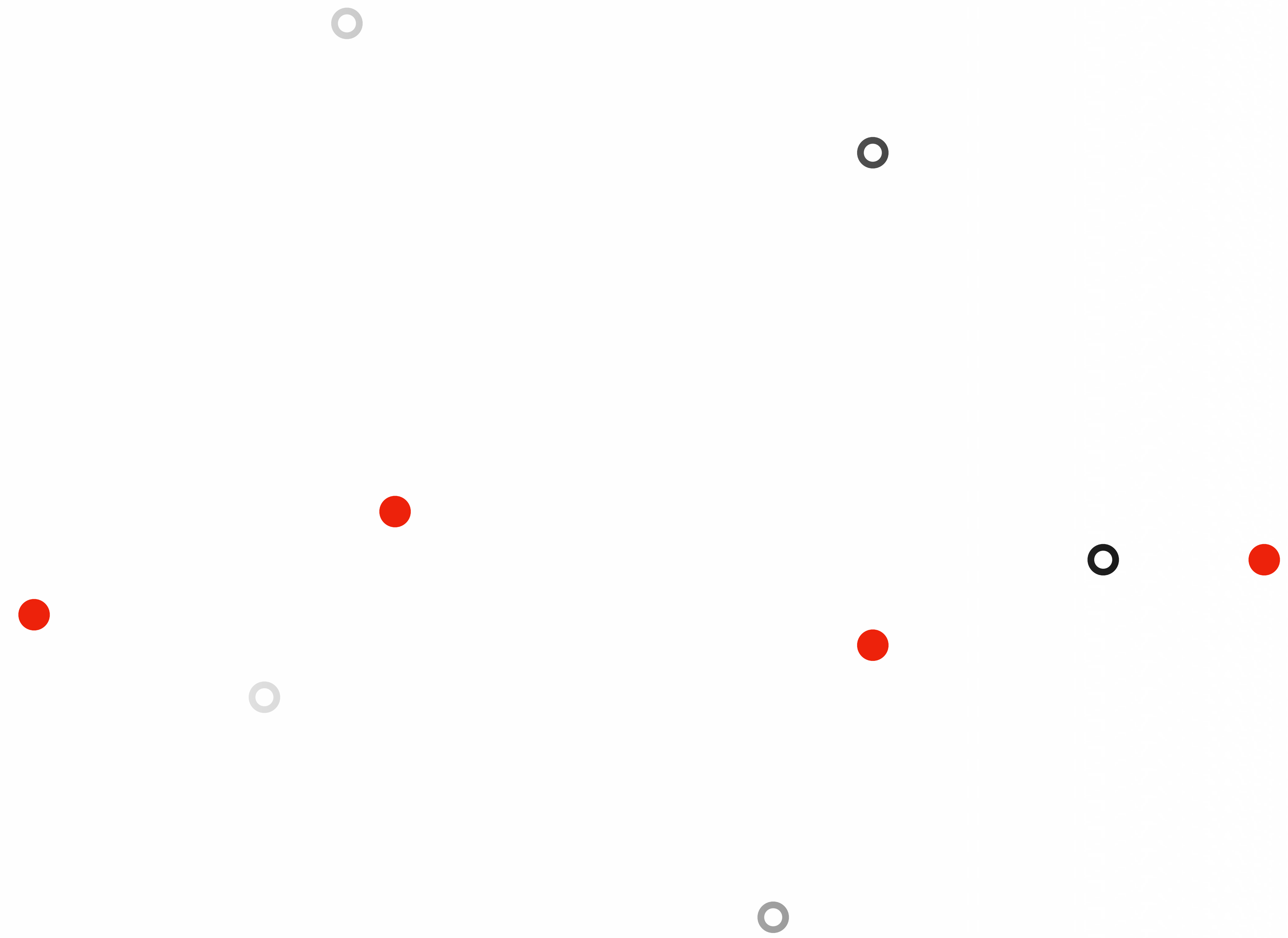
- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



HMM: Importance Sampling

Problem:

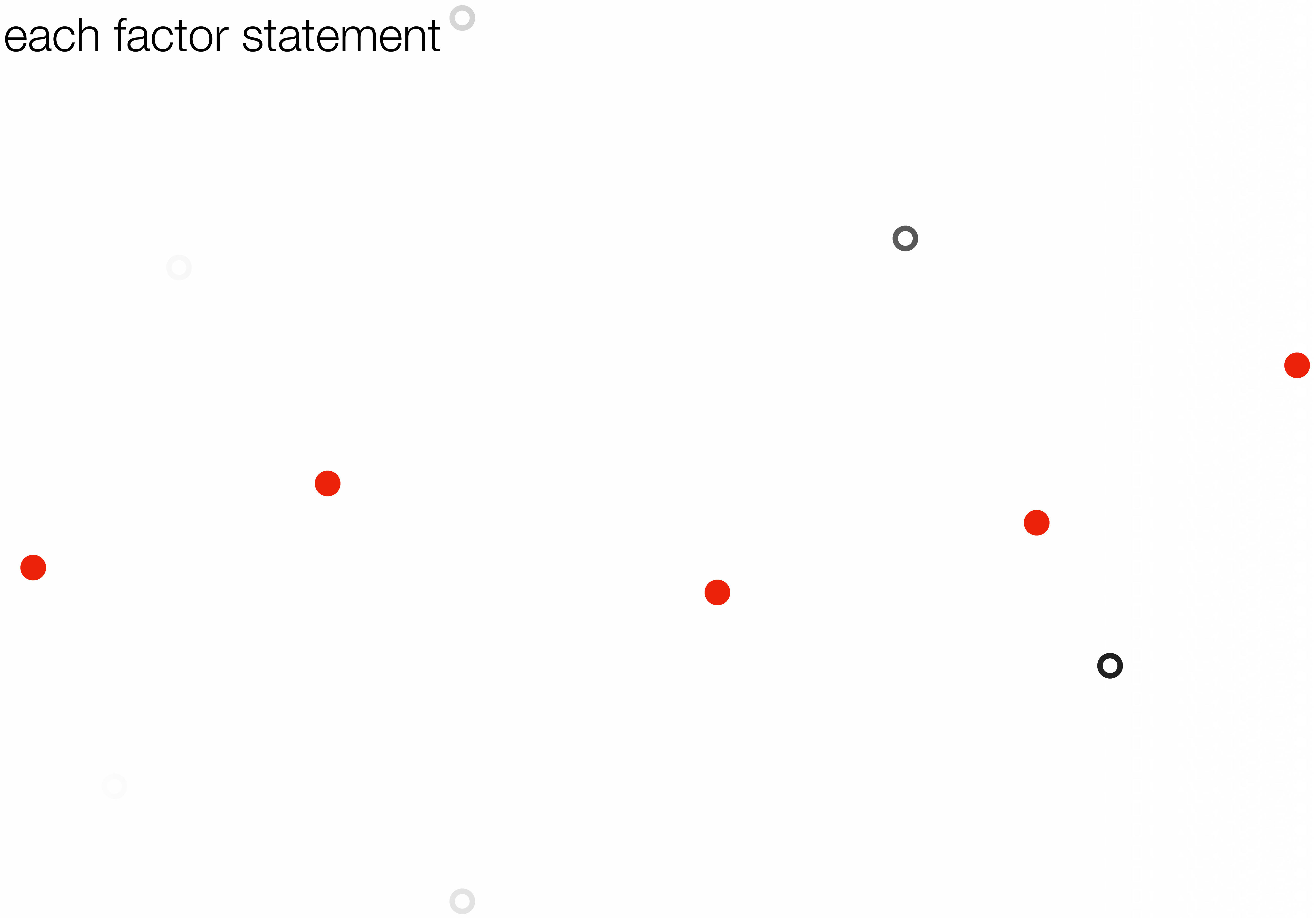
- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



HMM: Importance Sampling

Problem:

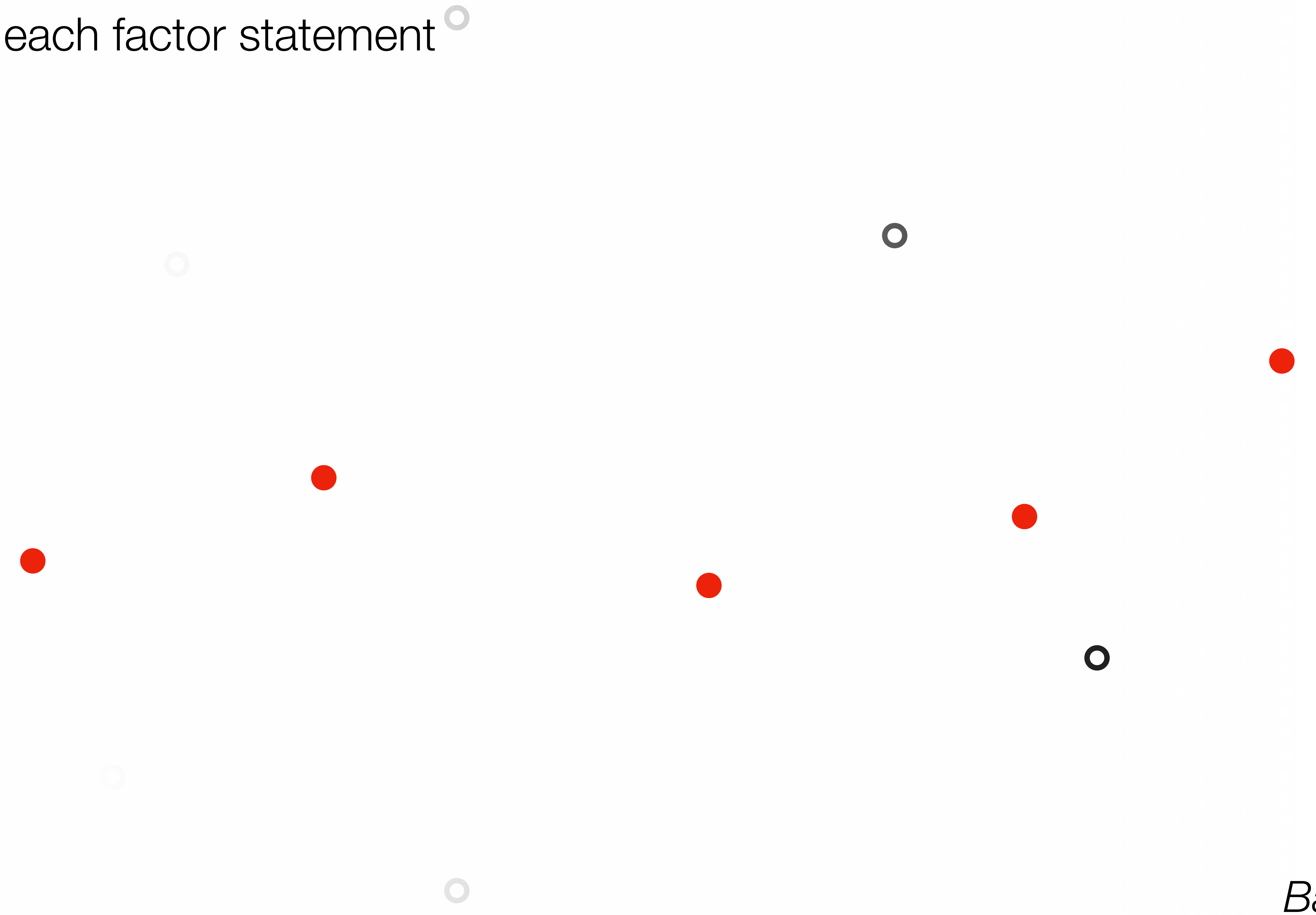
- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



HMM: Importance Sampling

Problem:

- Each particle does a random walk and compute a score
- No re-calibration based on observations
- Score decreases at each factor statement



Bad estimation

HMM: Particle Filter

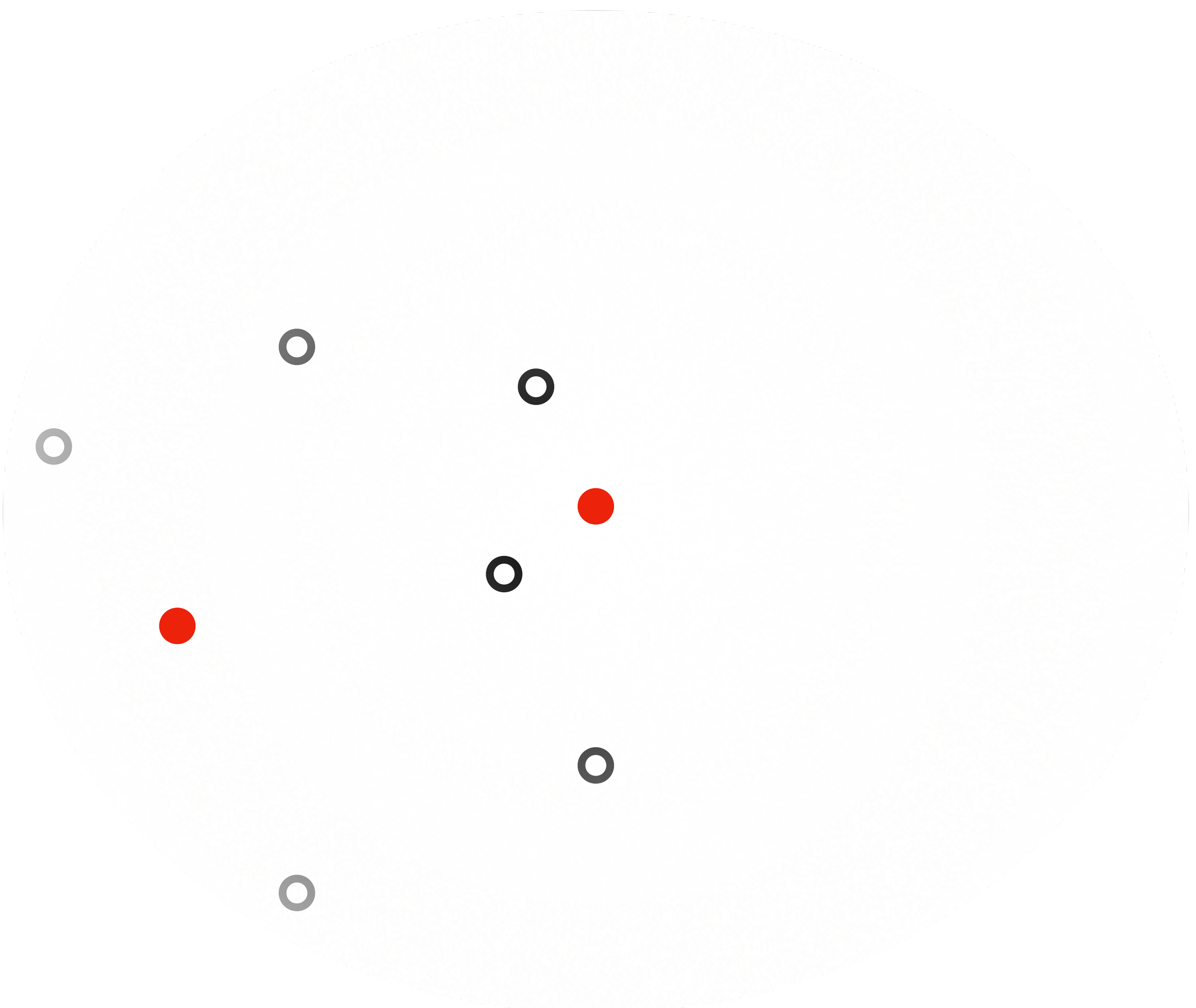
Add a resampling step

- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

HMM: Particle Filter

Add a resampling step

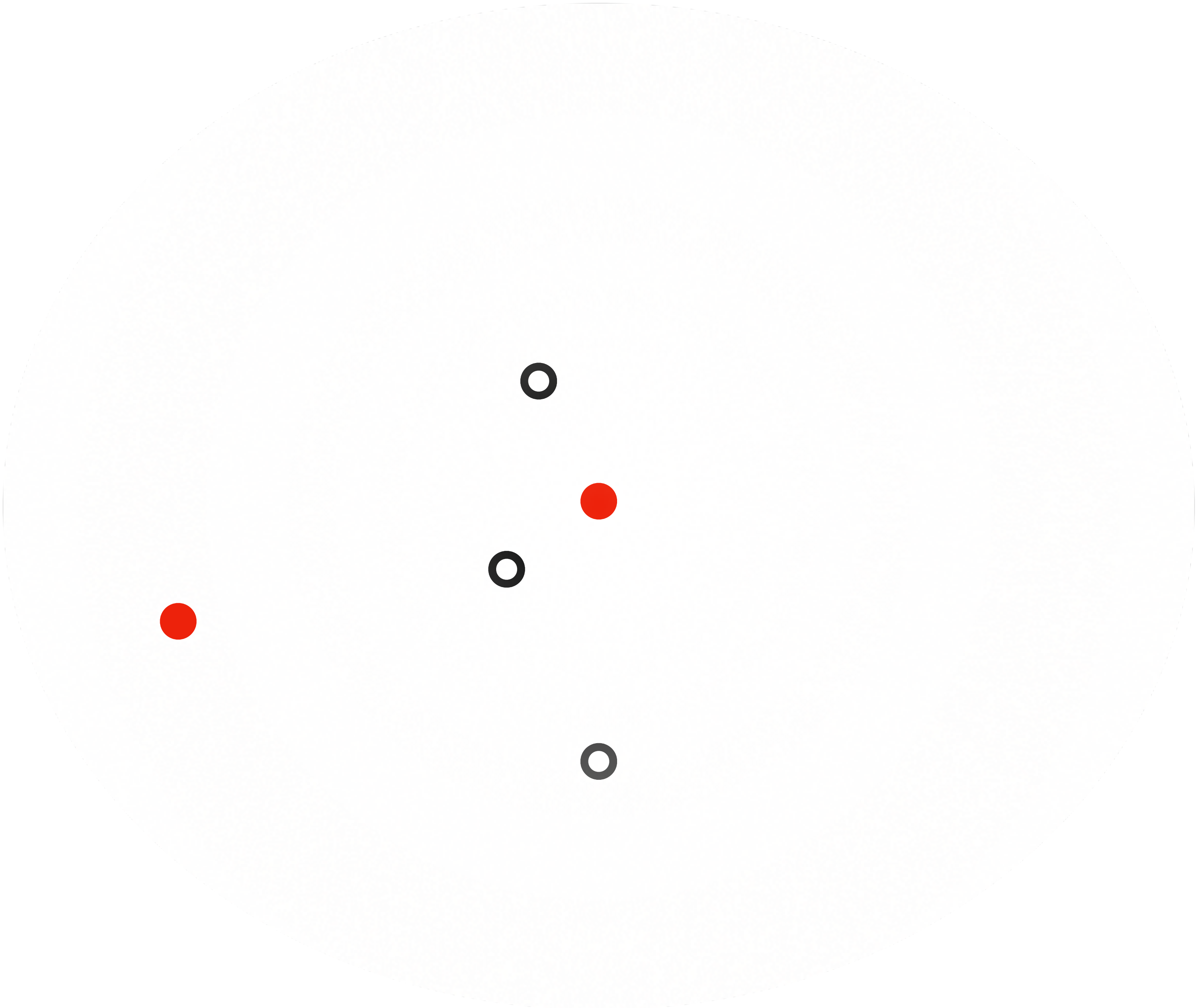
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle Filter

Add a resampling step

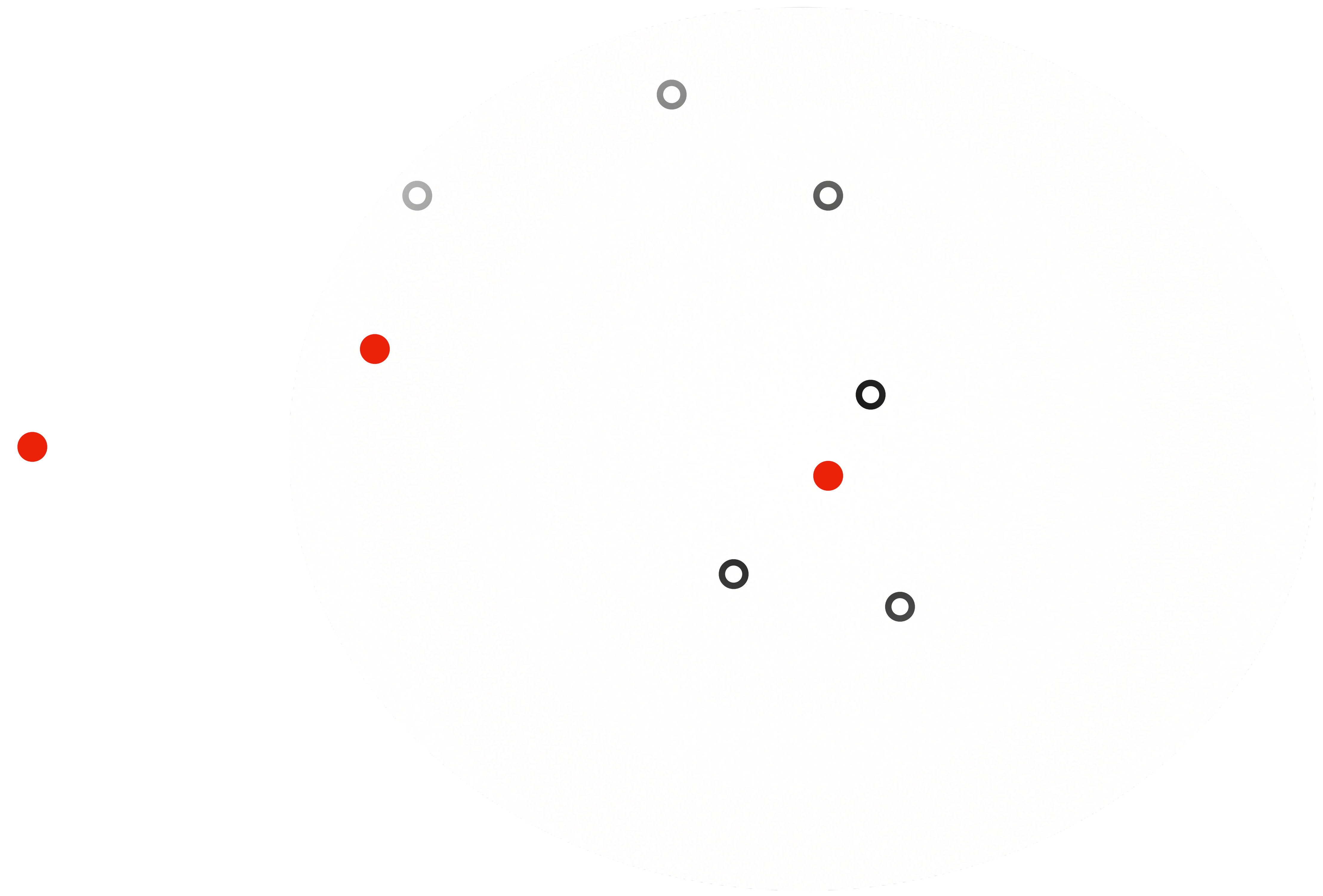
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle Filter

Add a resampling step

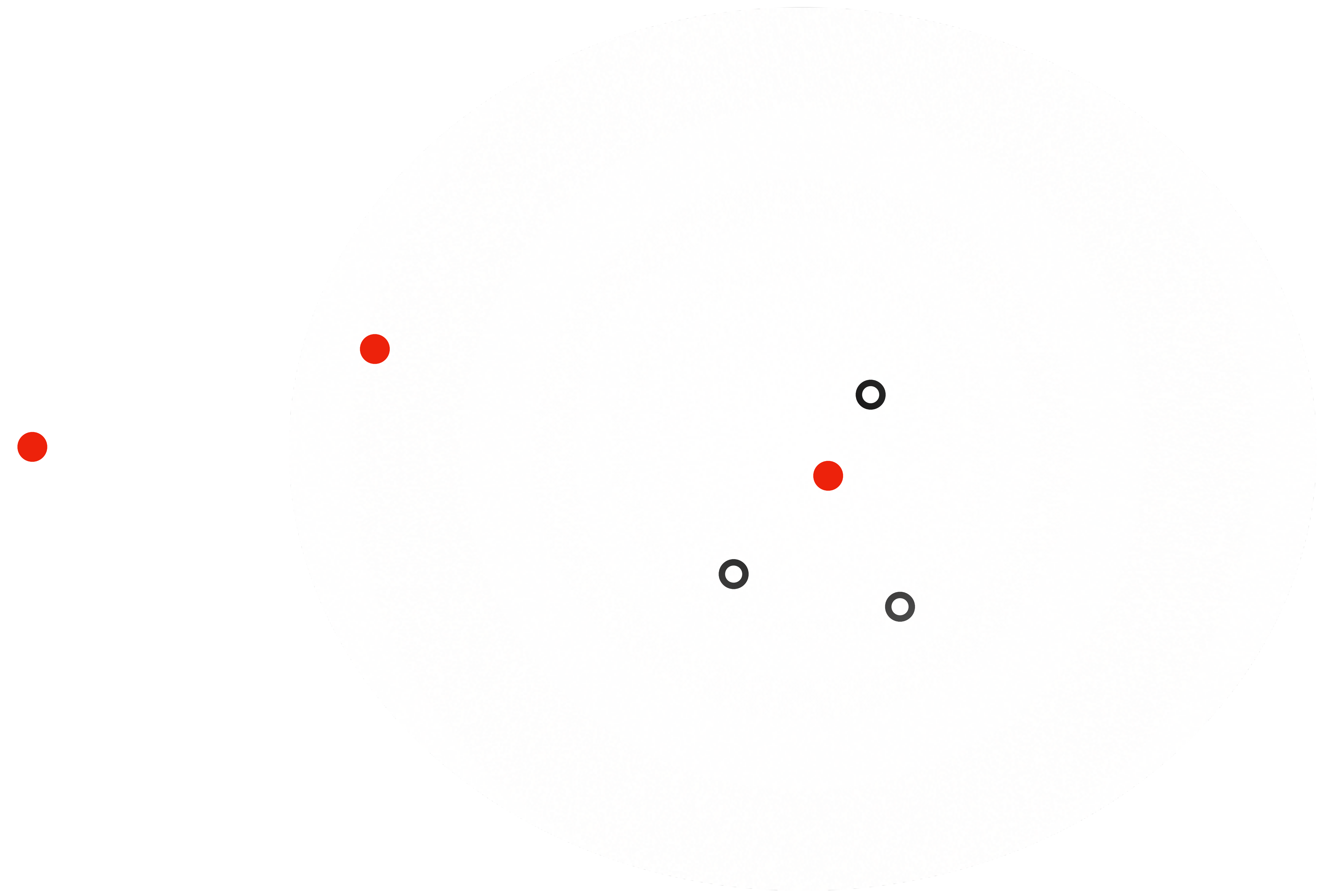
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle Filter

Add a resampling step

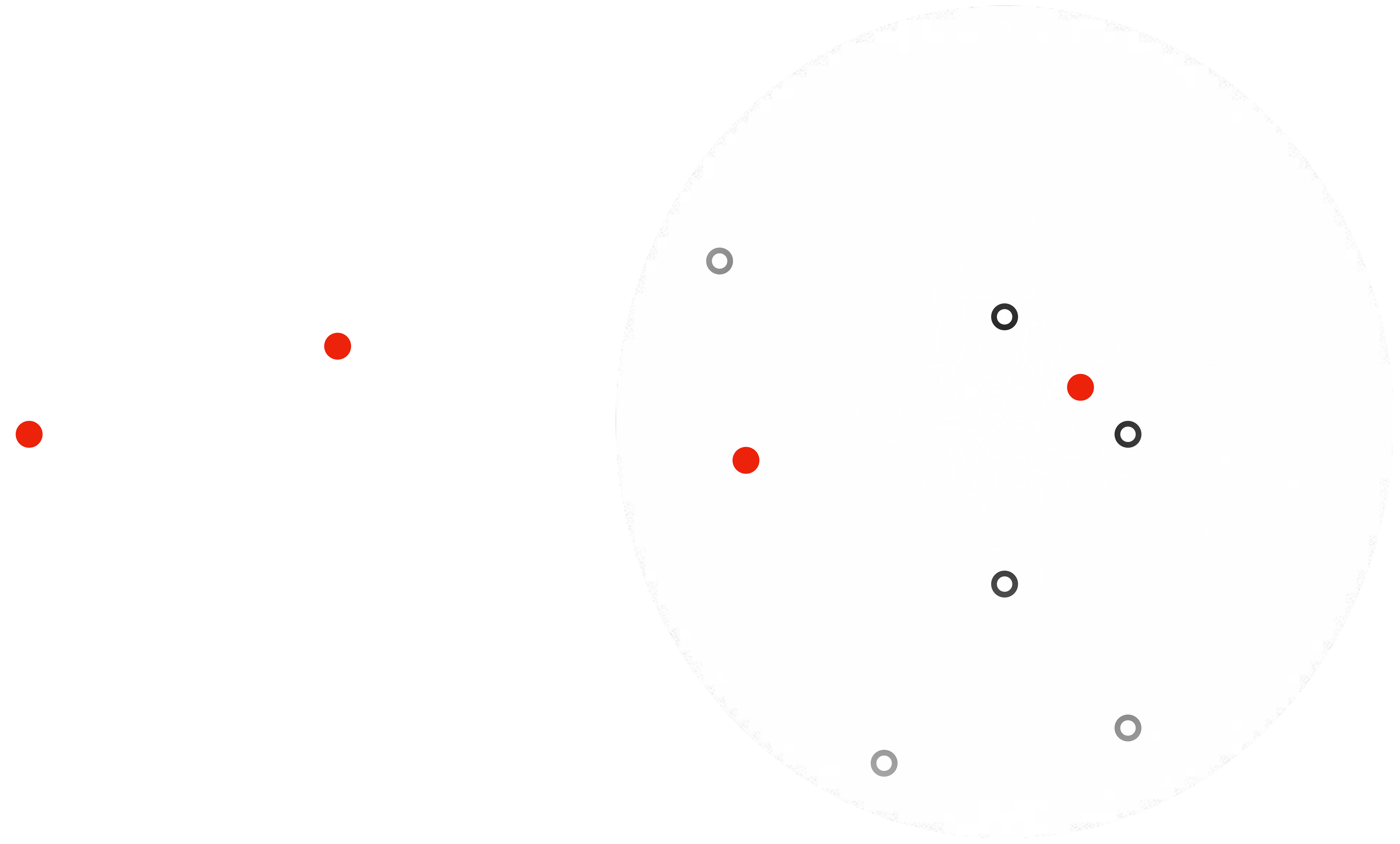
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle Filter

Add a resampling step

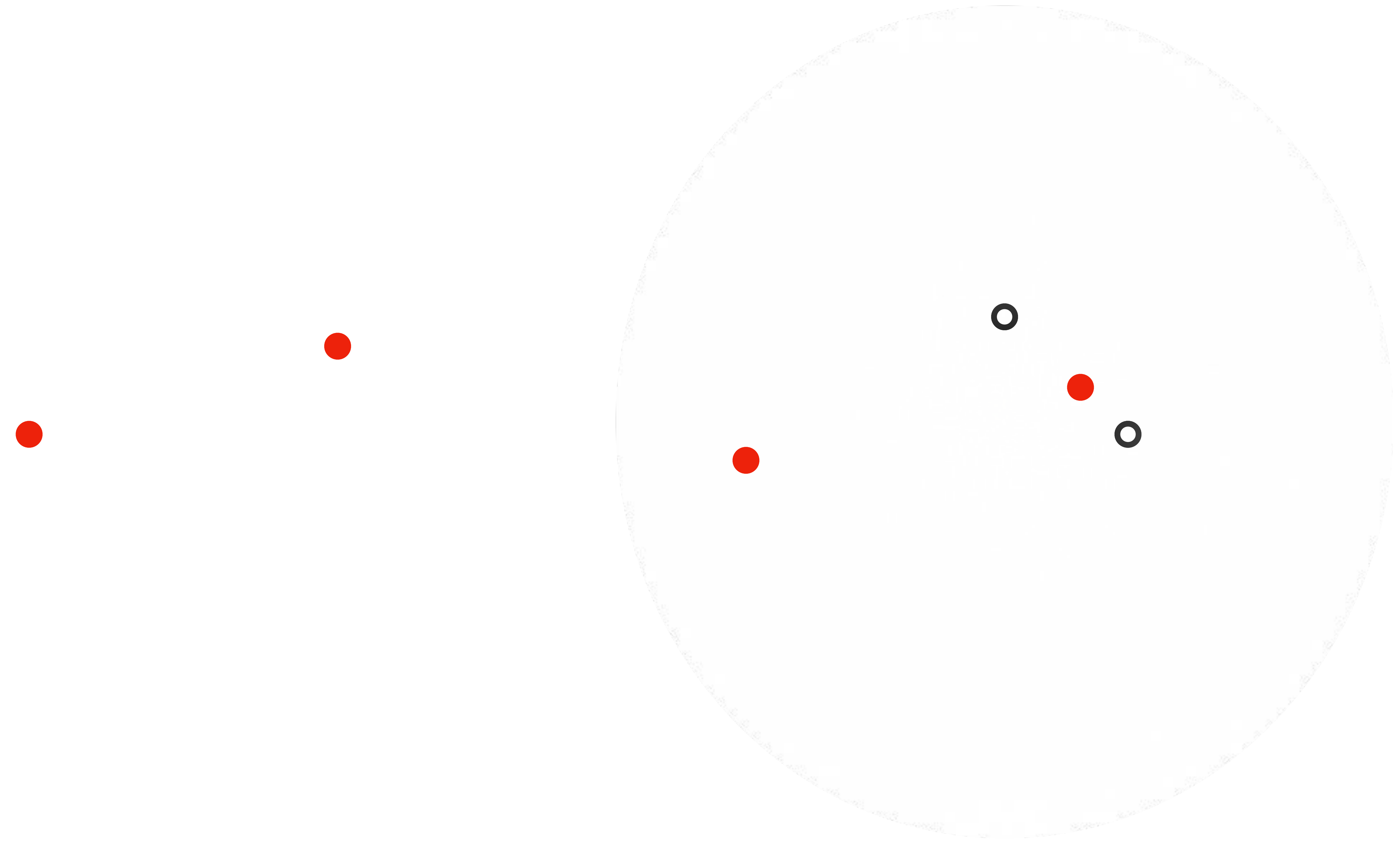
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle Filter

Add a resampling step

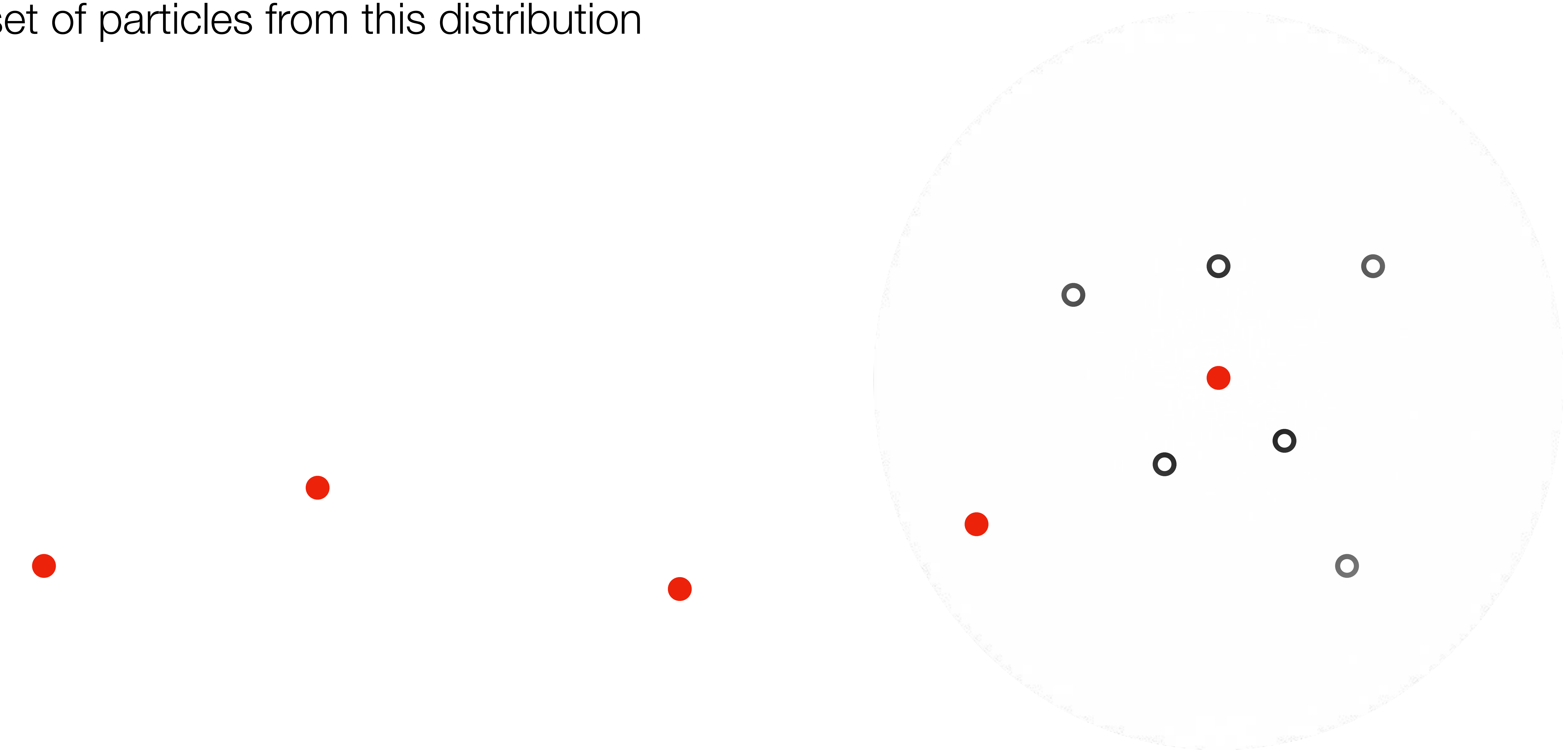
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle Filter

Add a resampling step

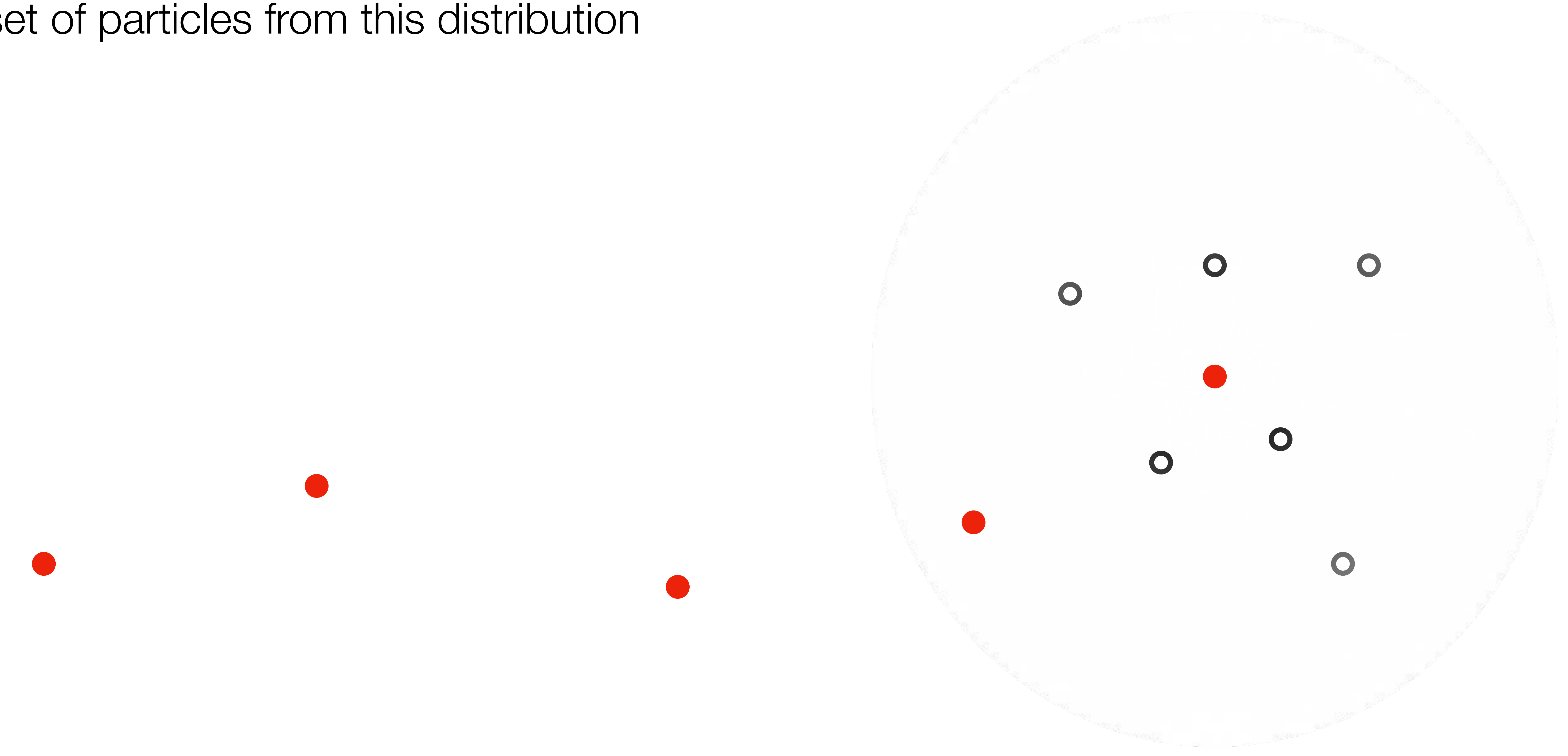
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



HMM: Particle Filter

Add a resampling step

- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution



Good estimation

Problem: Duplications

Problem: Duplications

How can we duplicate a particle during execution?

- Rerun the particle from the start?
- Force reuse sampled values?
- Clone the memory state?

Problem: Duplications

How can we duplicate a particle during execution?

- Rerun the particle from the start?
- Force reuse sampled values?
- Clone the memory state?

Continuation Passing Style

- Functions take an extra argument `k`: the continuation
- `k` implements what should be done with the result of the function
- In our context, we can use continuation to interrupt/restart the execution of a model

Continuation Passing Style (CPS)

BYO-PPL

Reminders: CPS

tree.ml

```
let rec tree_height t =
  match t with
  | Empty → 0
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

Reminders: CPS

tree.ml

```
let rec tree_height t =
  match t with
  | Empty → 0
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

```
let rec tree_height t =
  match t with
  | Empty → 0
  | Node (_, l, r) →
    let hl = tree_height l in
    let hr = tree_heigh r in
    (1 + max hl hr)
```

1. Add intermediate values

Reminders: CPS

tree.ml

```
let rec tree_height t =
  match t with
  | Empty → 0
  | Node (_, l, r) → 1 + max (tree_height l) (tree_height r)
```

```
let rec tree_height t k =
  match t with
  | Empty → k 0
  | Node (_, l, r) →
    let hl = tree_height l in
    let hr = tree_heigh r in
    k (1 + max hl hr)
```

1. Add intermediate values
2. Add call to continuation

Reminders: CPS

tree.ml

```
let rec tree_height t =
  match t with
  | Empty -> 0
  | Node (_, l, r) -> 1 + max (tree_height l) (tree_height r)
```

```
let rec tree_height t k =
  match t with
  | Empty -> k 0
  | Node (_, l, r) ->
    tree_height l (fun hl ->
      tree_heigh r (fun hr ->
        k (1 + max hl hr)))
```

1. Add intermediate values
2. Add call to continuation
3. Turn let/in into nested function call

Funny Bernoulli CPS

funny_bernoulli.ml

```
let funny_bernoulli () =
  let a = sample (bernoulli ~p:0.5) in
  let b = sample (bernoulli ~p:0.5) in
  let c = sample (bernoulli ~p:0.5) in
  let () = assume (a = 1 || b = 1) in
  a + b + c
```

1. Add intermediate values
2. Add call to continuation
3. Turn let/in into nested function call

Funny Bernoulli CPS

funny_bernoulli.ml

```
let funny_bernoulli () =
  let a = sample (bernoulli ~p:0.5) in
  let b = sample (bernoulli ~p:0.5) in
  let c = sample (bernoulli ~p:0.5) in
  let () = assume (a = 1 || b = 1) in
  a + b + c
```

```
let funny_bernoulli () k =
  sample (bernoulli ~p:0.5) (fun a →
    sample (bernoulli ~p:0.5) (fun b →
      sample (bernoulli ~p:0.5) (fun c →
        assume (a = 1 || b = 1) (fun () →
          k (a + b + c))))
```

1. Add intermediate values
2. Add call to continuation
3. Turn let/in into nested function call

CPS Monadic Operators

cps_operators.ml

```
let return e k = k e

let ( let* ) e f k = e (fun x → f x k)    (* let* x = e in f(x) *)
```

CPS Monadic Operators

cps_operators.ml

```
let return e k = k e

let ( let* ) e f k = e (fun x → f x k)     (* let* x = e in f(x) *)
```

```
let funny_bernoulli () k =
  sample (bernoulli ~p:0.5) (fun a →
    sample (bernoulli ~p:0.5) (fun b →
      sample (bernoulli ~p:0.5) (fun c →
        assume (a = 1 || b = 1) (fun () →
          k (a + b + c))))
```

```
let funny_bernoulli () =
  let* a = sample (bernoulli ~p:0.5) in
  let* b = sample (bernoulli ~p:0.5) in
  let* c = sample (bernoulli ~p:0.5) in
  let* () = assume (a = 1 || b = 1) in
  return (a + b + c)
```

Sample Generation (CPS)

BYO-PPL

CPS Models

infer.ml

```
module Gen : sig
  type 'a prob
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  val sample : 'a Distribution.t → ('a → 'b next) → 'b next
  val factor : float → (unit → 'b next) → 'b next
  val draw: ('a, 'b) model → 'a → 'b
end = struct ... end
```

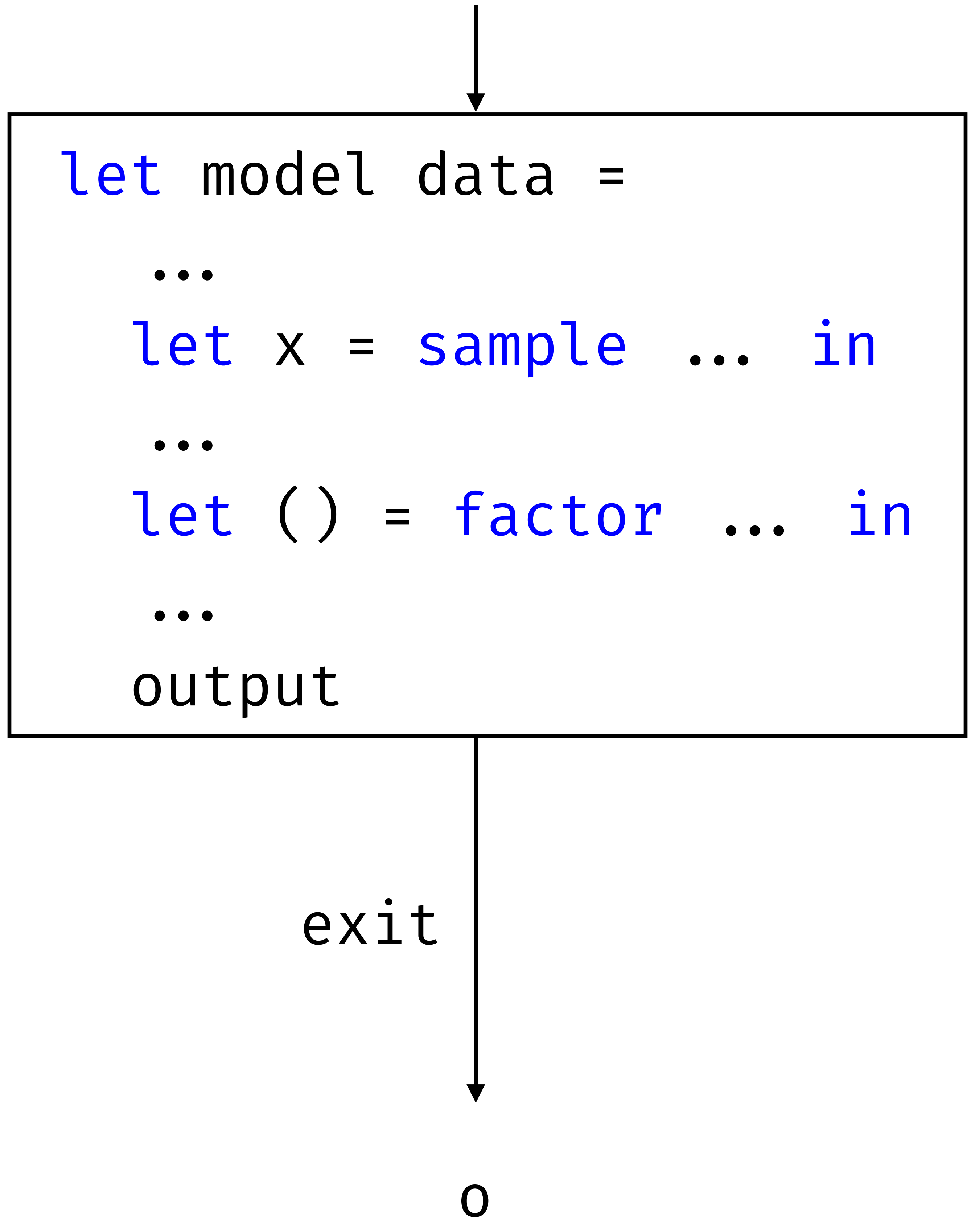
Type 'a prob

- Store all information required for inference (e.g., particles array)
- Type `('a, 'b) model` capture input/output types

Models ad probabilistic constructs are CPS functions

- Two arguments: input `'a` and a continuation on the return value `('b → 'b next)`.
- The return value is a continuation `'a next` that updates a probabilistic state of type `'a prob`.

Sample Generation



Sample Generation


```
module Gen = struct
  type 'a prob = 'a option
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  let exit v _prob = Some v

  let sample d k prob =
    let v = Distribution.draw d in
    k v prob

  let factor _s k prob = k () prob

  let draw m data =
    let v = (m data) exit None in
    Option.get v
end
```

Funny Bernoulli

funny_bernoulli.ml

```
open Infer.Gen

let funny_bernoulli () =
  let* a = sample (bernoulli ~p:0.5) in
  let* b = sample (bernoulli ~p:0.5) in
  let* c = sample (bernoulli ~p:0.5) in
  let* () = assume (a = 1 || b = 1) in
  return (a + b + c)

let _ =
  for _ = 1 to 10 do
    let v = draw funny_bernoulli () in
    Format.printf "%d " v
  done
```

```
❯ dune exec ./examples/funny_bernoulli.exe

1 1 2 2 2 2 2 1 3 2
```

Importance Sampling (CPS)

BYO-PPL

Importance Sampling


```
module Importance_sampling : sig
  type 'a prob
  and 'a next = 'a prob → 'a prob
  and ('a, 'b) model = 'a → ('b → 'b next) → 'b next

  val sample : 'a Distribution.t → ('a → 'b next) → 'b next
  val factor : float → (unit → 'b next) → 'b next
  val infer : ('a, 'b) model → 'a → 'b Distribution.t
end = struct ... end
```

Inference algorithm

- Run a set of n independent executions
- `sample`: draw a sample from a distribution
- `factor`: associate a score to the current execution
- Gather output values and score to approximate the posterior distribution

Importance Sampling

0

```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
  …
  output
```

1

```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
  …
  output
```

2

```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
  …
  output
```

Importance Sampling

0
```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
  …
  output
```

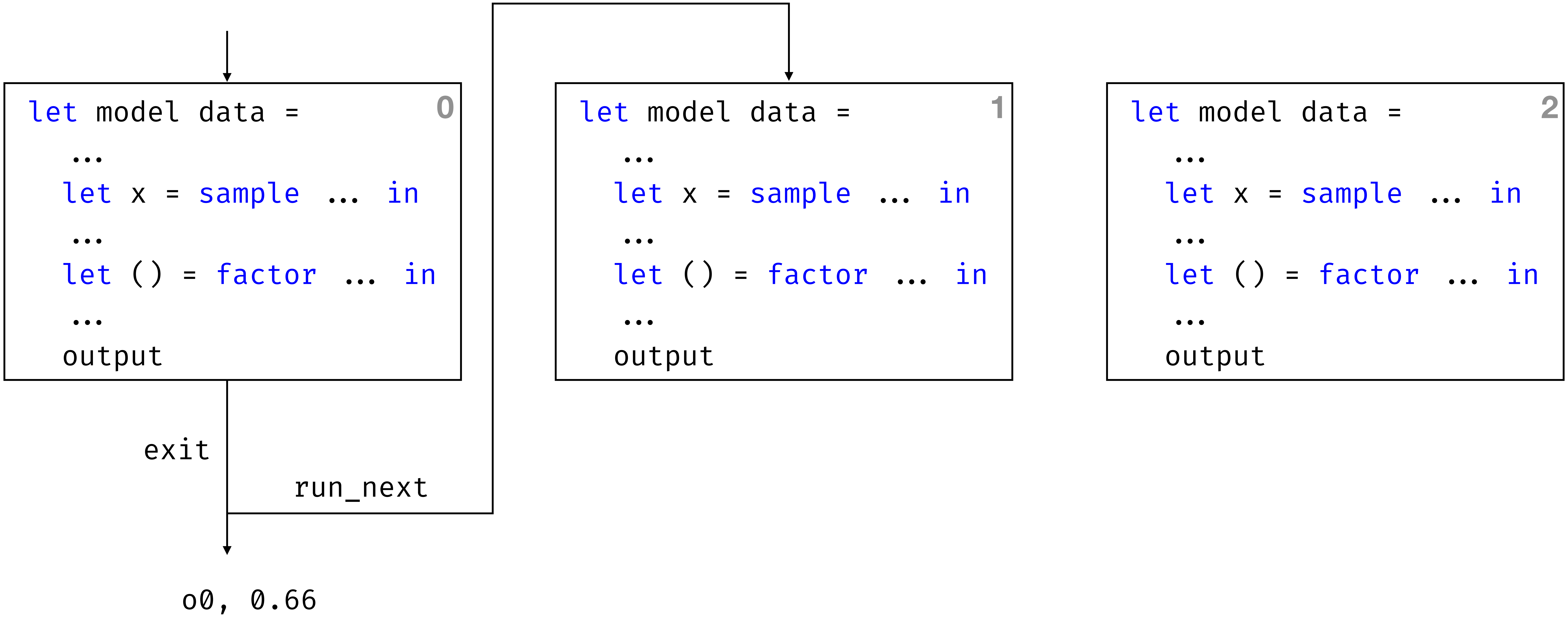
1
```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
  …
  output
```

2
```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
  …
  output
```

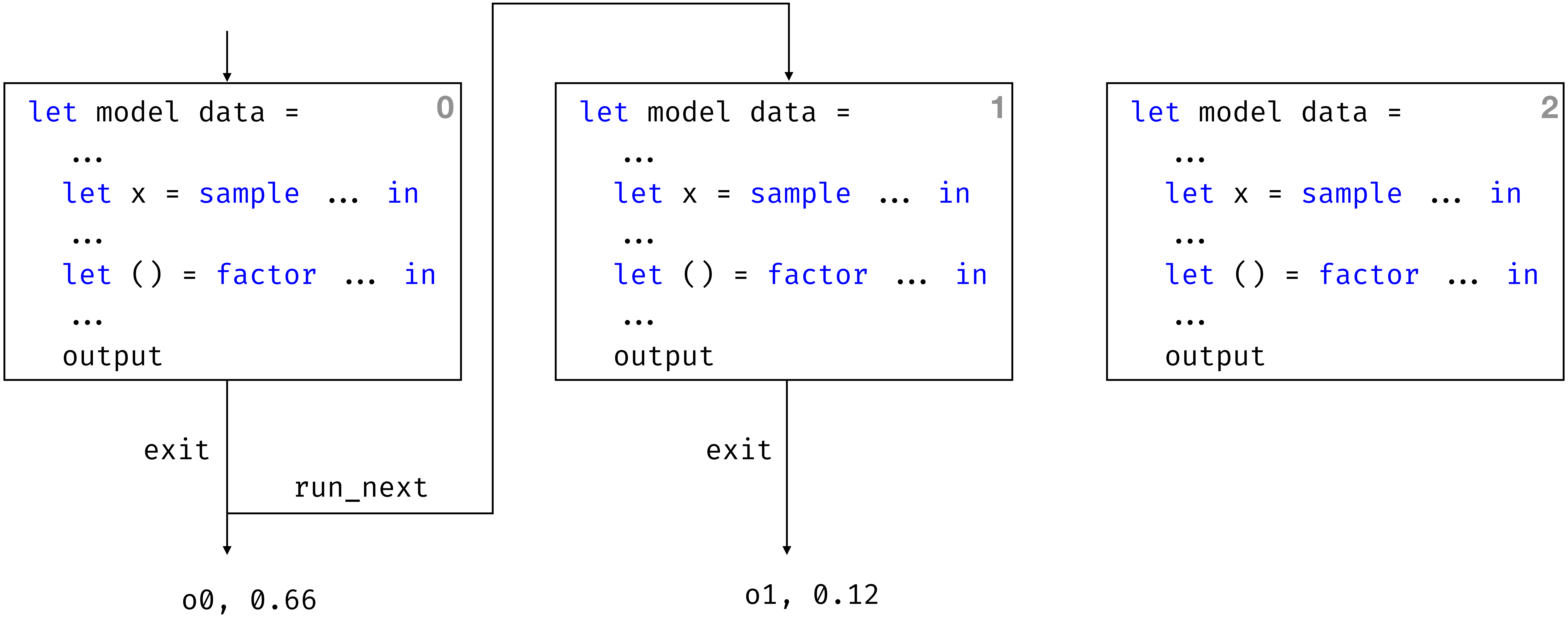
exit

o0, 0.66

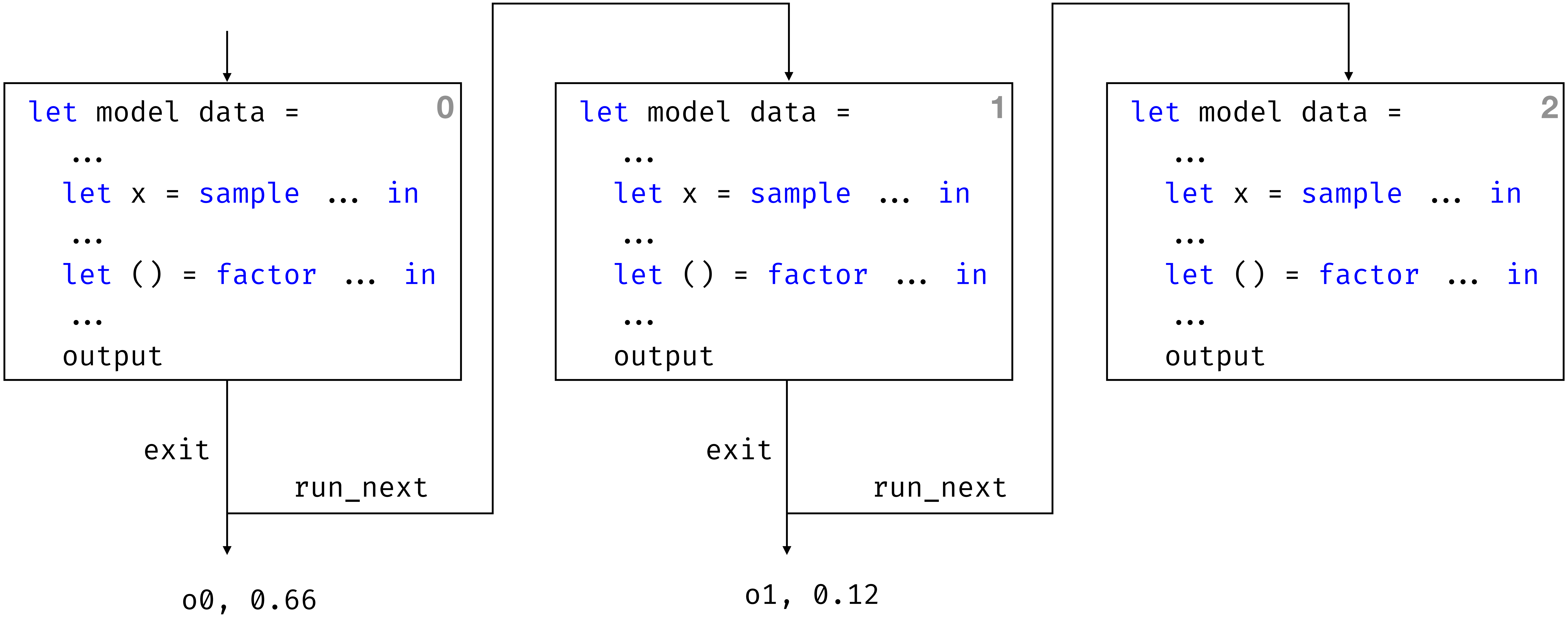
Importance Sampling



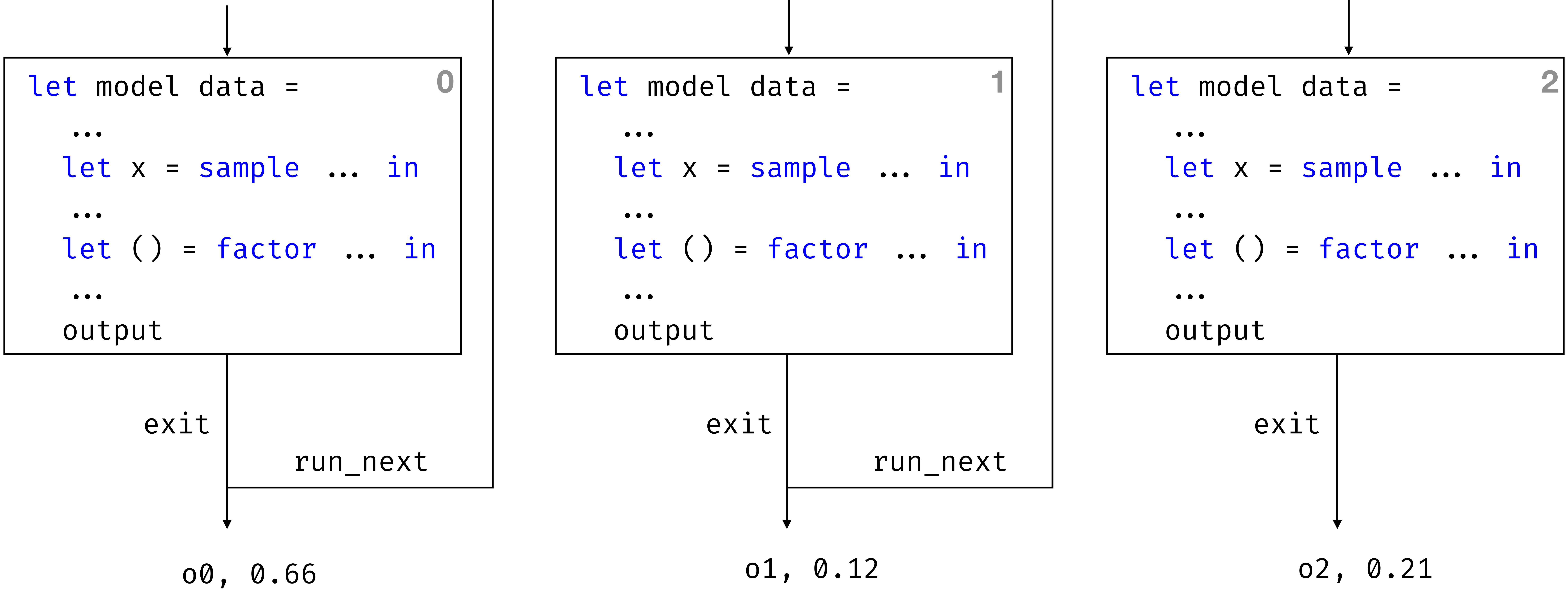
Importance Sampling



Importance Sampling



Importance Sampling



Importance Sampling


```
module Importance_sampling = struct
  type 'a prob =  ...

  let sample d k prob = assert false
  let factor s k prob = assert false

  let infer ?(n = 1000) m data = assert false
end
```

Importance Sampling


```
module Importance_sampling = struct
  type 'a prob = { id : int; particles : 'a particle array }
  and 'a particle = { k : 'a next; value : 'a option; score : float }
  ...

  let sample d k prob =
    let v = Distribution.draw d in
    k v prob

  let factor s k prob =
    let particle = prob.particles.(prob.id) in
    prob.particles.(prob.id) ← { particle with score = s +. particle.score };
    k () prob

  ...
end
```

Importance Sampling

infer.ml

```
module Importance_sampling = struct
  type 'a prob = { id : int; particles : 'a particle array }
  and 'a particle = { k : 'a next; value : 'a option; score : float }
   …

  (* Call the continuation of the next particle *)
  let run_next prob =
    if prob.id < Array.length prob.particles - 1 then
      let k = prob.particles.(prob.id + 1).k in
      k { prob with id = prob.id + 1 }
    else prob

  let exit v prob =
    let particle = prob.particles.(prob.id) in
    prob.particles.(prob.id) ← { particle with value = Some v };
    run_next prob
end
```

Importance Sampling


```
module Importance_sampling = struct
  type 'a prob = { id : int; particles : 'a particle array }
  and 'a particle = { k : 'a next; value : 'a option; score : float }
  ...

  let infer ?(n = 1000) m data =
    let init_particle = { k = (m data) exit; value = None; score = 0. } in
    let prob = { id = -1; particles = Array.init n (fun _ → init_particle) } in
    let prob = run_next prob in
    let values = Array.map (fun x → Option.get x.value) prob.particles in
    let logits = Array.map (fun x → x.score) prob.particles in
    Distribution.support ~values ~logits
end
```

Coin

coin.ml

```
open Infer.Importance_sampling

let coin x =
  let* z = sample (uniform ~a:0. ~b:1.) in
  let* () = Cps_list.iter (observe (bernoulli ~p:z)) x in
  return z

let _ =
  let dist = infer coin [1; 1; 0; 0; 0; 0; 0; 0; 0; 0] in
  let m, s = Distribution.stats dist in
  Format.printf "Coin bias, mean:%f, std:%f@." m s
```

```
❯ dune exec ./examples/coin.exe

Coin bias, mean:0.247876, std:0.118921
Beta(2+1, 8+1), mean:0.250000, std:0.120096
```

Particle Filter (CPS)

BYO-PPL

Particle Filter

basic.ml

```
module Particle_filter = struct
  include Importance_sampling

  let resample particles = assert false
  let factor s k prob = assert false
end
```

Inference algorithm : importance sampling, but...

- Add a resampling step at each `factor`
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

Particle Filter

0

```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
```

```
  …
  output
```

1

```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
```

```
  …
  output
```

2

```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
```

```
  …
  output
```

Particle Filter

0

```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
```

```
  …
  output
```

1

```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
```

```
  …
  output
```

2

```
let model data =
  …
  let x = sample … in
  …
  let () = factor … in
```

```
  …
  output
```

0.66

Particle Filter

run_next

```
let model data =                      0
  ...
  let x = sample ... in
  ...
  let () = factor ... in
```

```
  ...
  output
```

```
let model data =                      1
  ...
  let x = sample ... in
  ...
  let () = factor ... in
```

```
  ...
  output
```

```
let model data =                      2
  ...
  let x = sample ... in
  ...
  let () = factor ... in
```

```
  ...
  output
```

0.66

Particle Filter

run_next

```
let model data =                      0
  ...
  let x = sample ... in
  ...
  let () = factor ... in
```

```
  ...
  output
```

```
let model data =                      1
  ...
  let x = sample ... in
  ...
  let () = factor ... in
```

```
  ...
  output
```

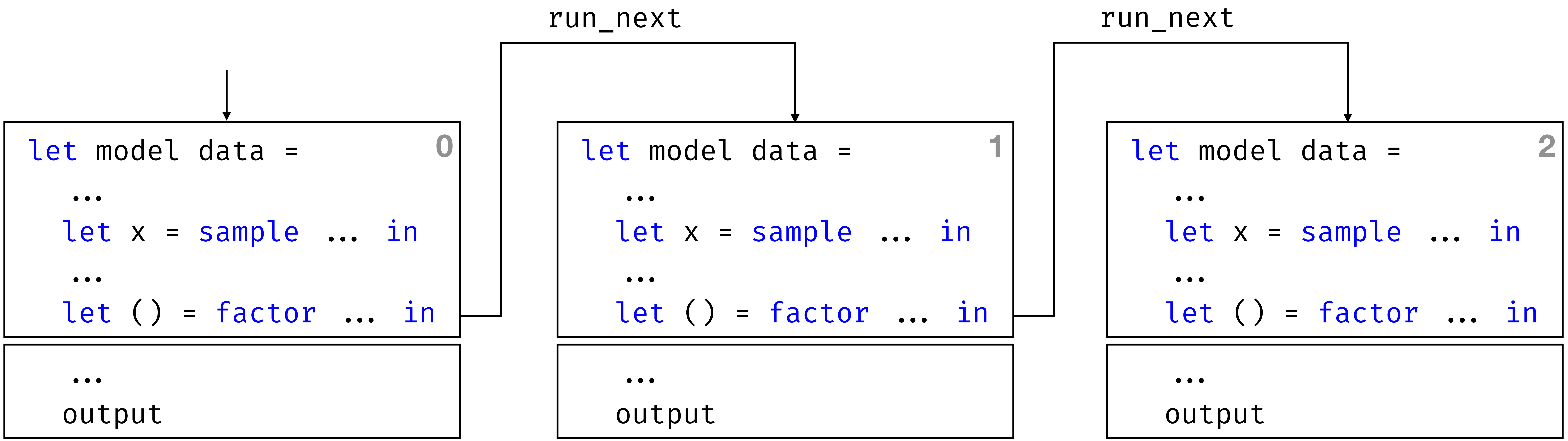
```
let model data =                      2
  ...
  let x = sample ... in
  ...
  let () = factor ... in
```

```
  ...
  output
```

0.66

0.12

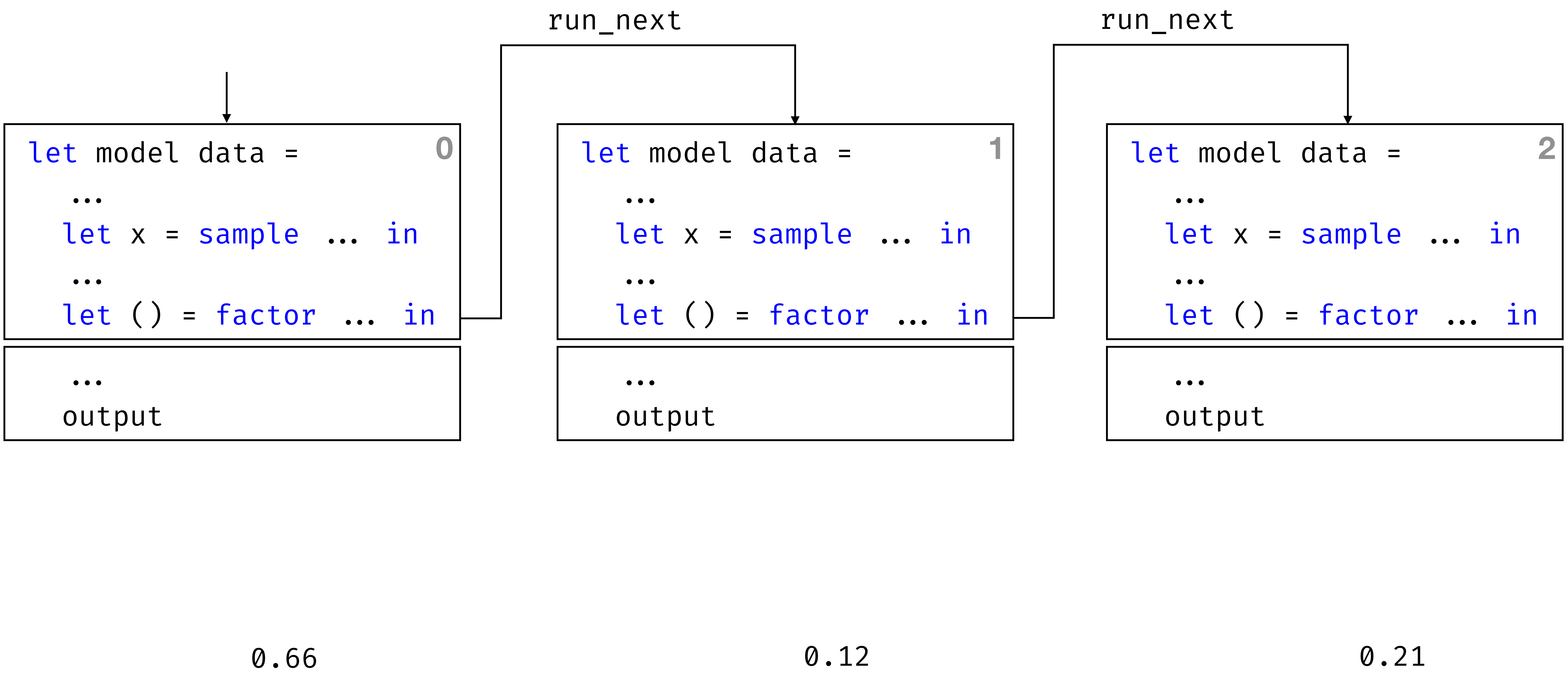
Particle Filter



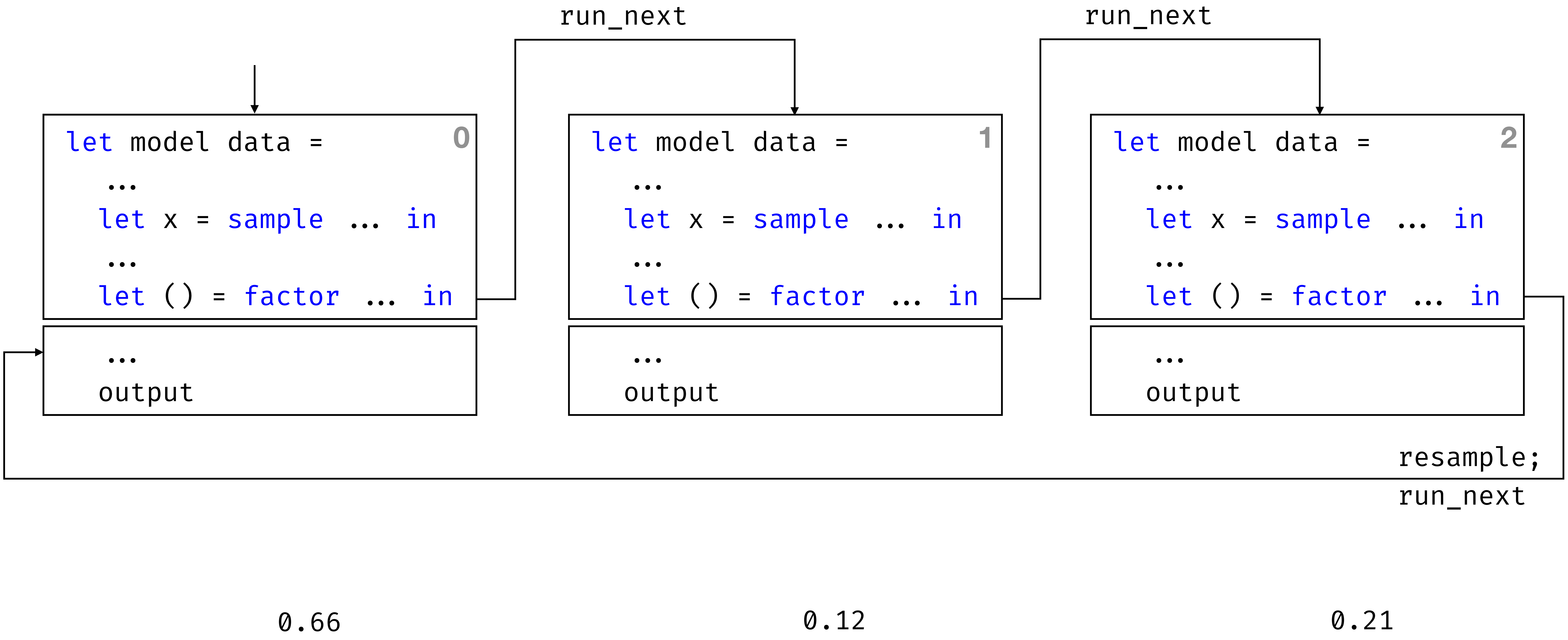
0.66

0.12

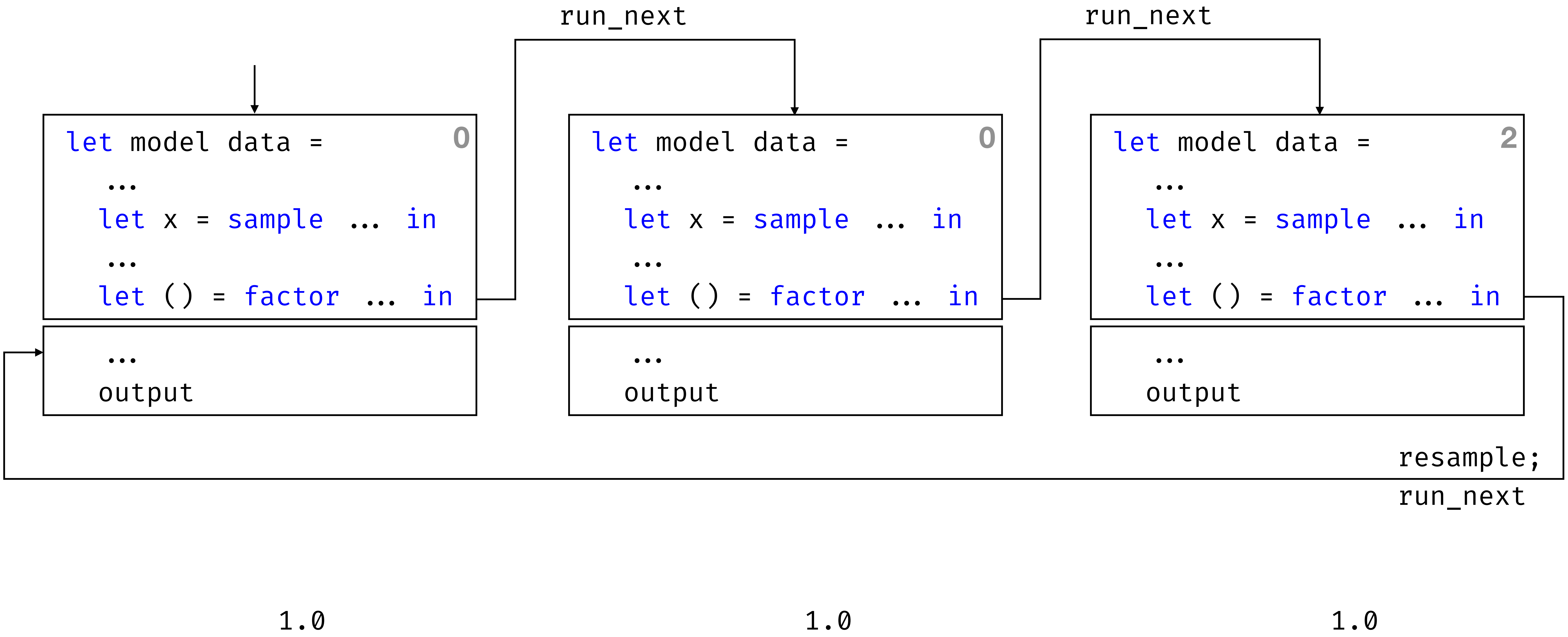
Particle Filter



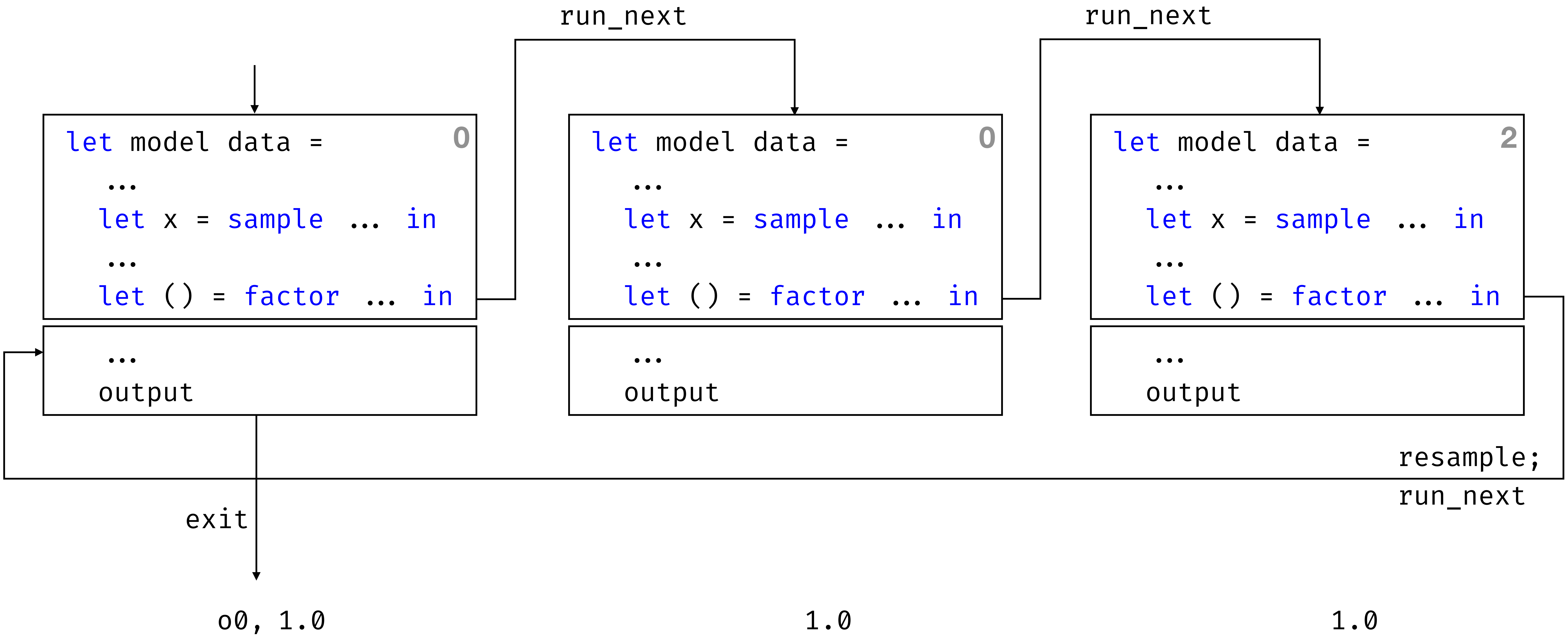
Particle Filter



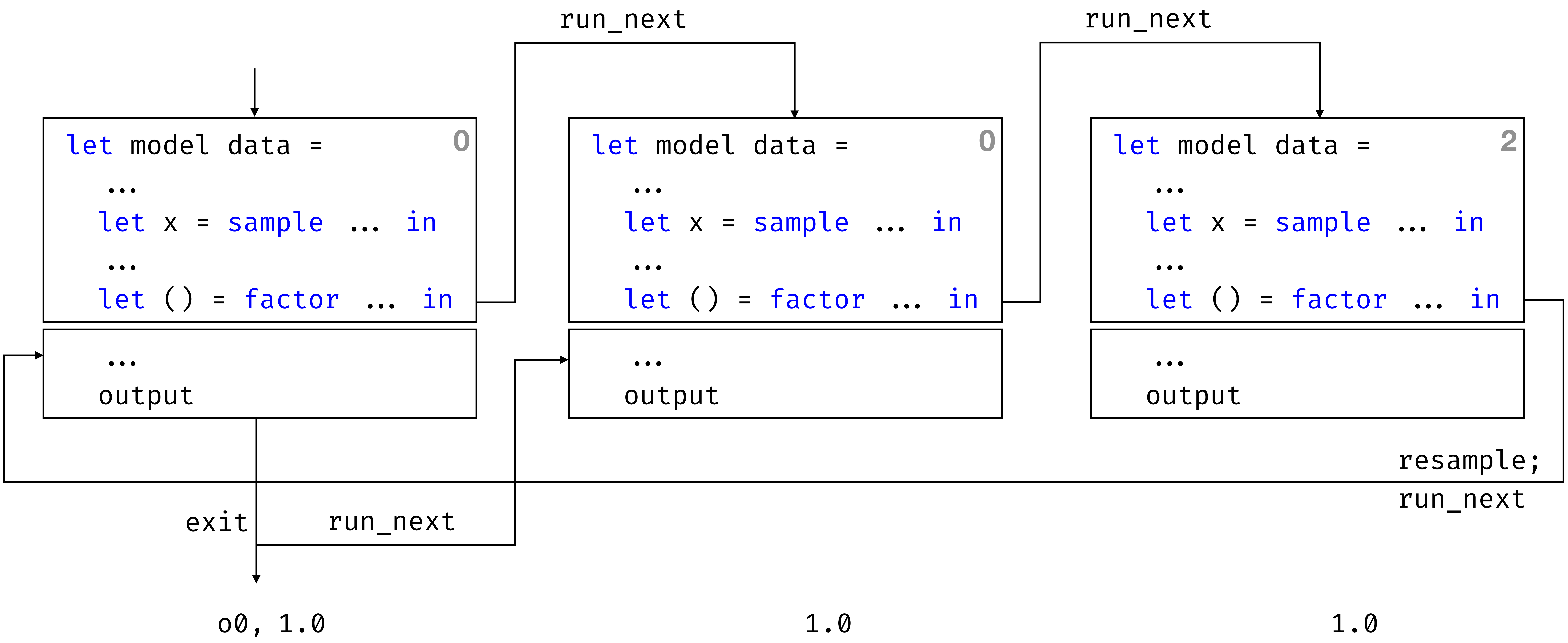
Particle Filter



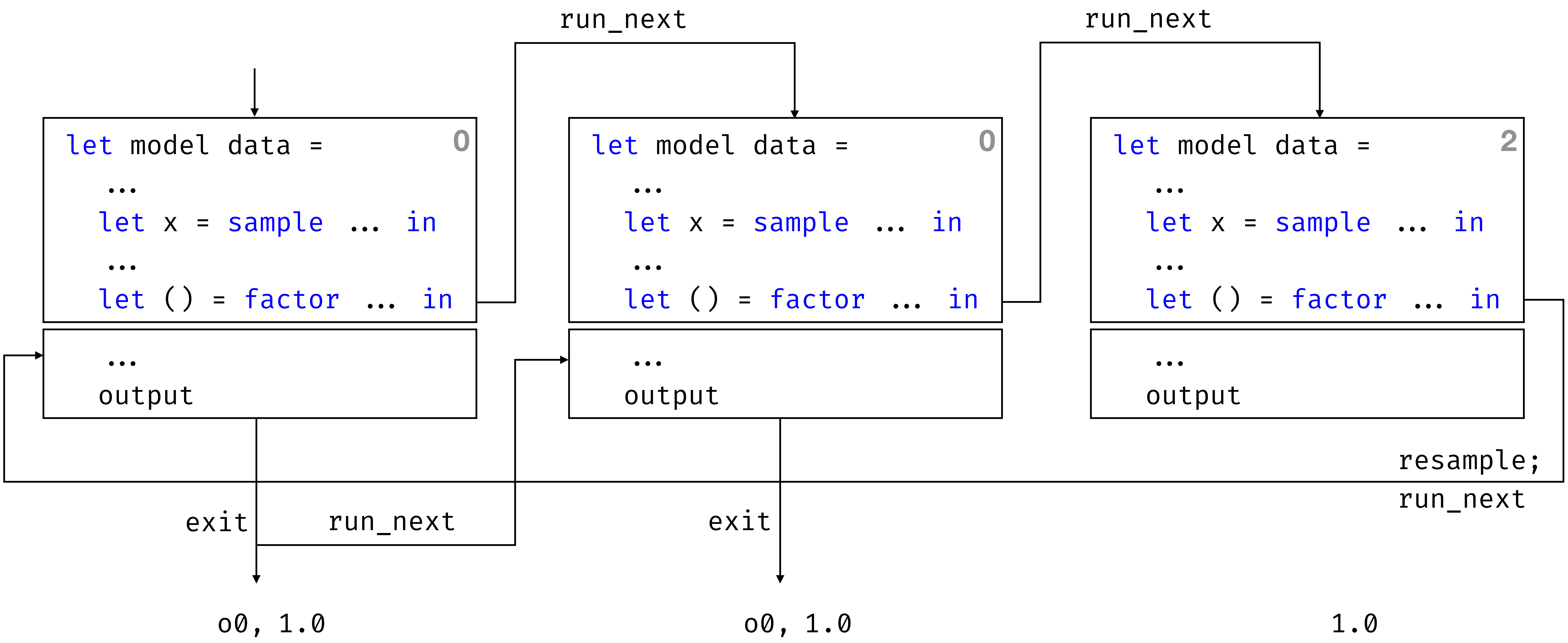
Particle Filter



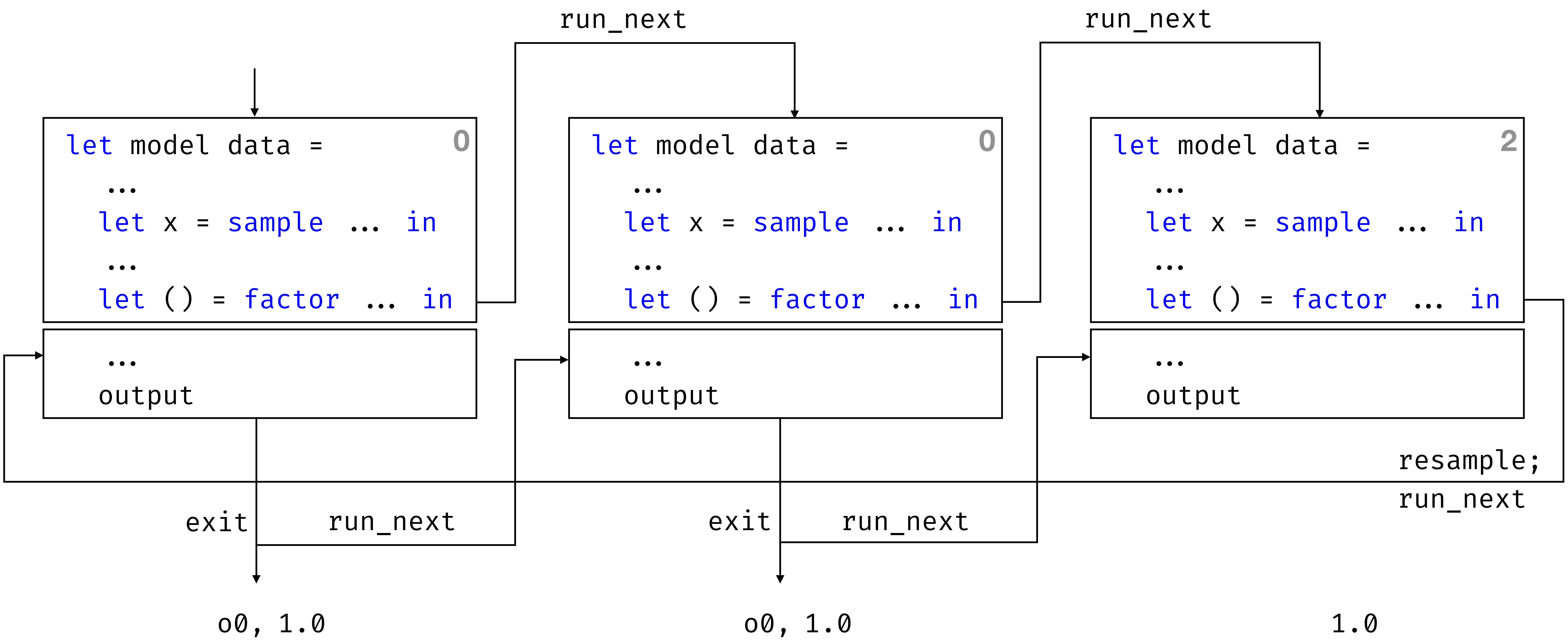
Particle Filter



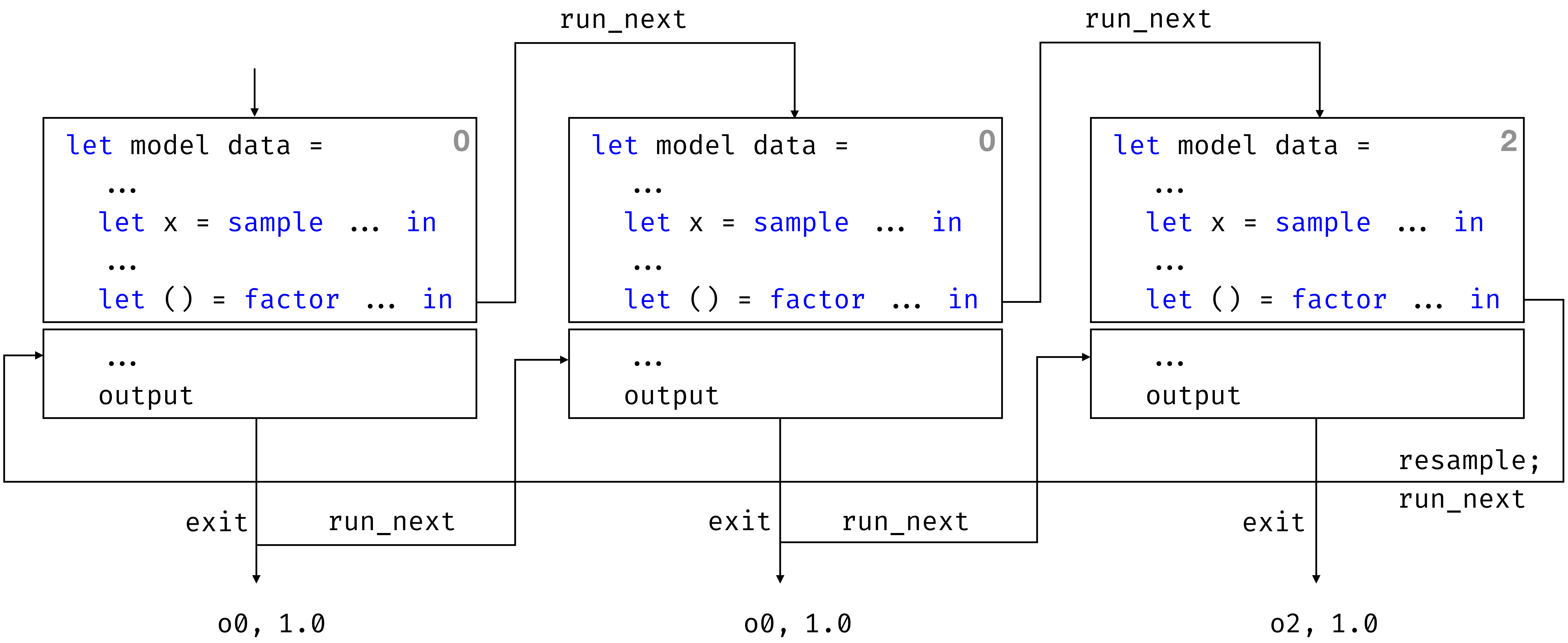
Particle Filter



Particle Filter



Particle Filter



Particle Filter


```
module Particle_filter = struct
  include Importance_sampling

  let resample particles =
    let logits = Array.map (fun x → x.score) particles in
    let values = Array.map (fun x → { x with score = 0. }) particles in
    let dist = Distribution.support ~values ~logits in
    Array.init (Array.length particles) (fun _ → Distribution.draw dist)

  let factor s k prob =
    let particle = prob.particles.(prob.id) in
    prob.particles.(prob.id) ← { particle with k = k (); score = s +. particle.score };
    let prob =
      if prob.id < Array.length prob.particles - 1 then prob
      else { id = -1; particles = resample prob.particles }
    in
    run_next prob
end
```

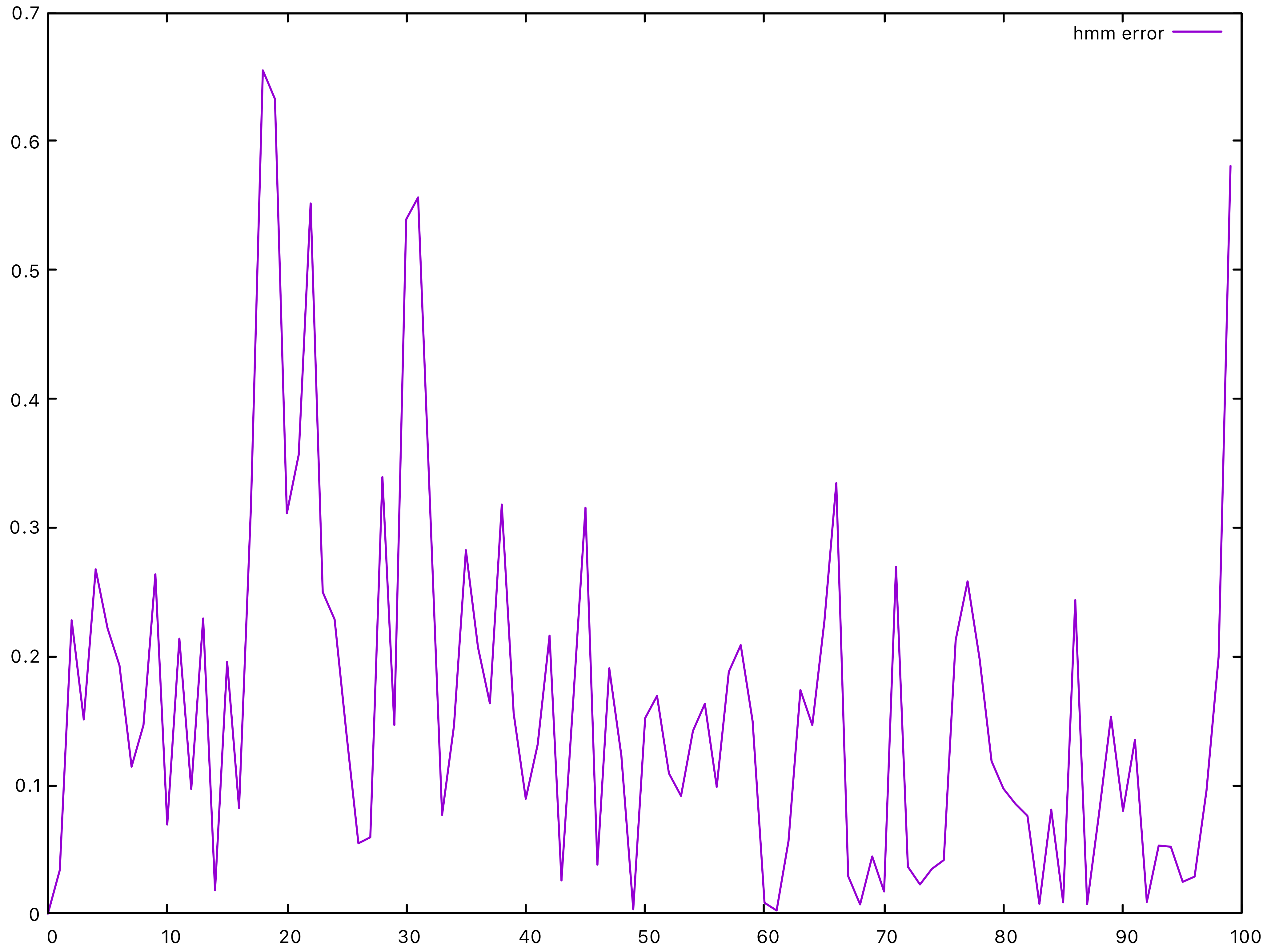
HMM: Hidden Markov Model

hmm.ml

```
open Infer.Particle_filter

let hmm prob data =
  let rec gen states data =
    match (states, data) with
    | [], y :: data → gen [ y ] data
    | states, [] → return states
    | pre_x :: _, y :: data →
        let* x = sample prob (gaussian ~mu:pre_x ~sigma:1.0) in
        let* () = observe prob (gaussian ~mu:x ~sigma:1.0) y in
        gen (x :: states) data
  in
  gen [] data

let _ =
  let data = Owl.Arr.linspace 0. 20. 20 ▷ Owl.Arr.to_array ▷ Array.to_list in
  let dist = Distribution.split_list (infer ~n:100 hmm data) in
  let m_x = List.map Distribution.mean dist in
  List.iter2 (Format.printf "%f >> %f") data m_x
```

HMM: Hidden Markov Model

```
❯ dune exec ./hmm.exe

0.000000 >> 0.000000
1.052632 >> 0.997546
2.105263 >> 2.300316
3.157895 >> 3.289649
4.210526 >> 4.857555
5.263158 >> 4.907179
6.315789 >> 6.254198
7.368421 >> 7.208341
8.421053 >> 8.432642
9.473684 >> 8.938143
10.526316 >> 9.555007
11.578947 >> 11.098199
12.631579 >> 12.823460
13.684211 >> 13.701444
14.736842 >> 14.934314
15.789474 >> 16.115058
...
```



Conclusion

For a given inference algoritm, how to implement `sample`, `assume`, `factor`, `observe`, and `infer`?

I - Basic inference

- Rejection sampling
- Importance sampling

II - Continuation Passing Style models

III - Inference on CPS models

- Sample generation
- Importance sampling
- Particle filter

References

WebPPL
Noah Goodman and Andreas Stuhlmüller
http://webppl.org/

The Design and Implementation of Probabilistic Programming Languages
Noah Goodman and Andreas Stuhlmüller
http://dippl.org/

An Introduction to Probabilistic Programming
Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, Frank Wood
https://arxiv.org/abs/1809.10756

Embedded probabilistic domain-specific language HANSEI
Oleg Kiselyov, Chung-chieh Shan
https://okmij.org/ftp/kakuritu/Hansei.html

BYO-PPL

Build Your Own Probabilistic Language

- Clone the repo: `git clone https://github.com/mpri-probprog/byo-ppl-22-23.git`
- Install the dependencies: `opam install . --deps-only`
- Build the project: `dune build`
- Test an example: `dune exec ./examples/funny_bernoulli.exe`

Implemented as an OCaml embedded domain specific language (eDSL)

- `Distribution`: small library of probability distributions and basic statistical functions.
- `Basic`: basic inference algorithms (rejection sampling inference sampling)
- `Infer`: inference algorithms for models written in Continuation Passing Style (CPS).
- `Cps_operators`: syntactic sugar to write CPS style probabilistic models.
- `Utils`: missing utilities functions used in other modules.