# Probabilistic Programming Languages

Guillaume Baudart

*MPRI 2023-2024*

# Reminders

Probabilistic Programming Languages

# Probabilistic programming

**Programming and reasoning with uncertainty**

- Sample from probability distributions
- Condition on observed data

**Bayesian Inference: learn parameters from data**

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

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

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

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: $z \sim \textit{Uniform}(0, 1)$
- Observations: for $i \in [1, n]$, $x_i \sim \textit{Bernoulli}(z)$
- Posterior: $p(z \mid x_1, \ldots, x_n)$?

# 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: $z \sim \textit{Uniform}(0, 1)$
- Observations: for $i \in [1, n]$, $x_i \sim \textit{Bernoulli}(z)$
- Posterior: $p(z \mid x_1, \ldots, x_n)$?

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

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

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

# Example: Coin

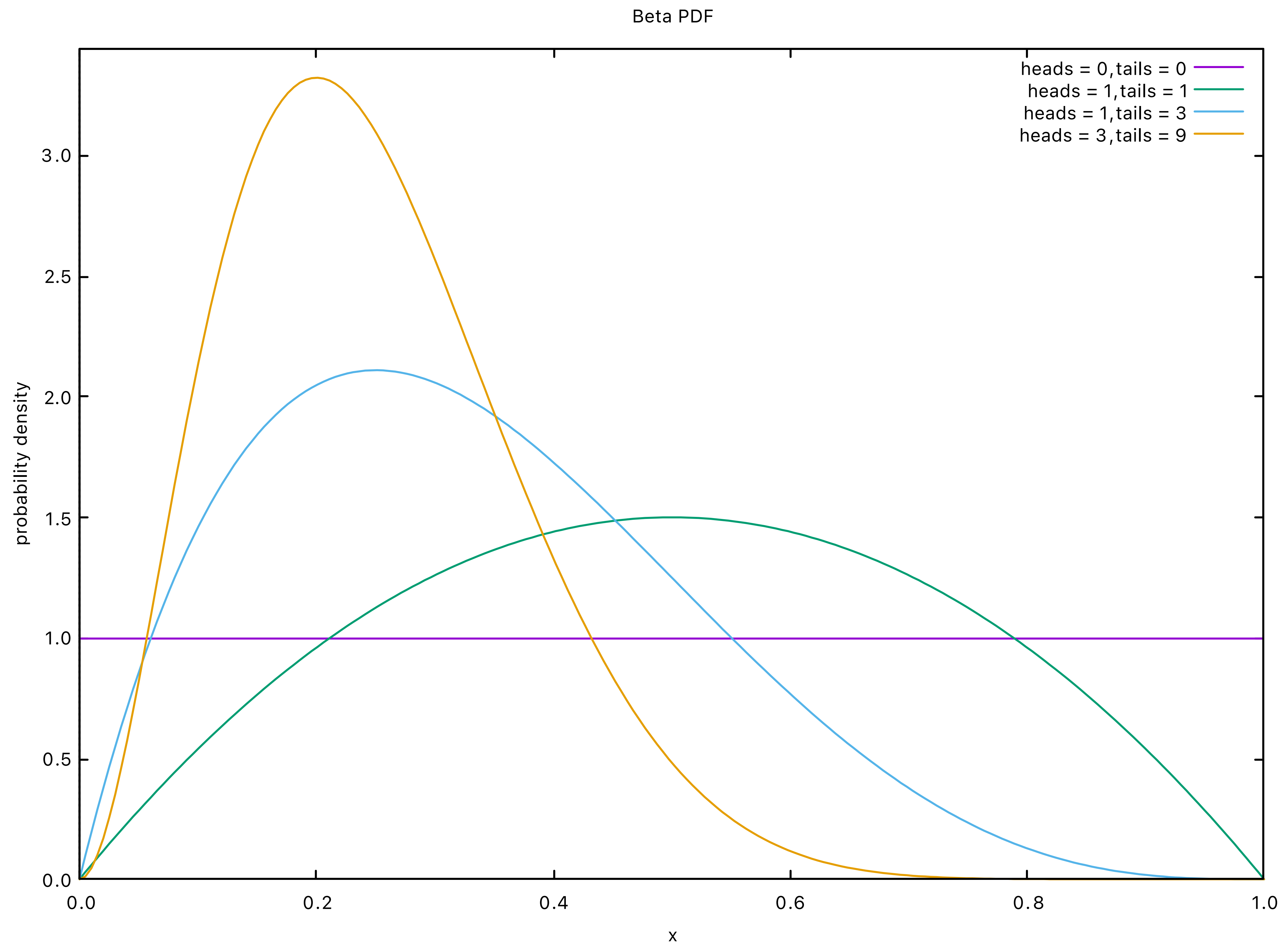
## Consider a series of coin tosses

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

## Probabilistic model

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

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



# 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

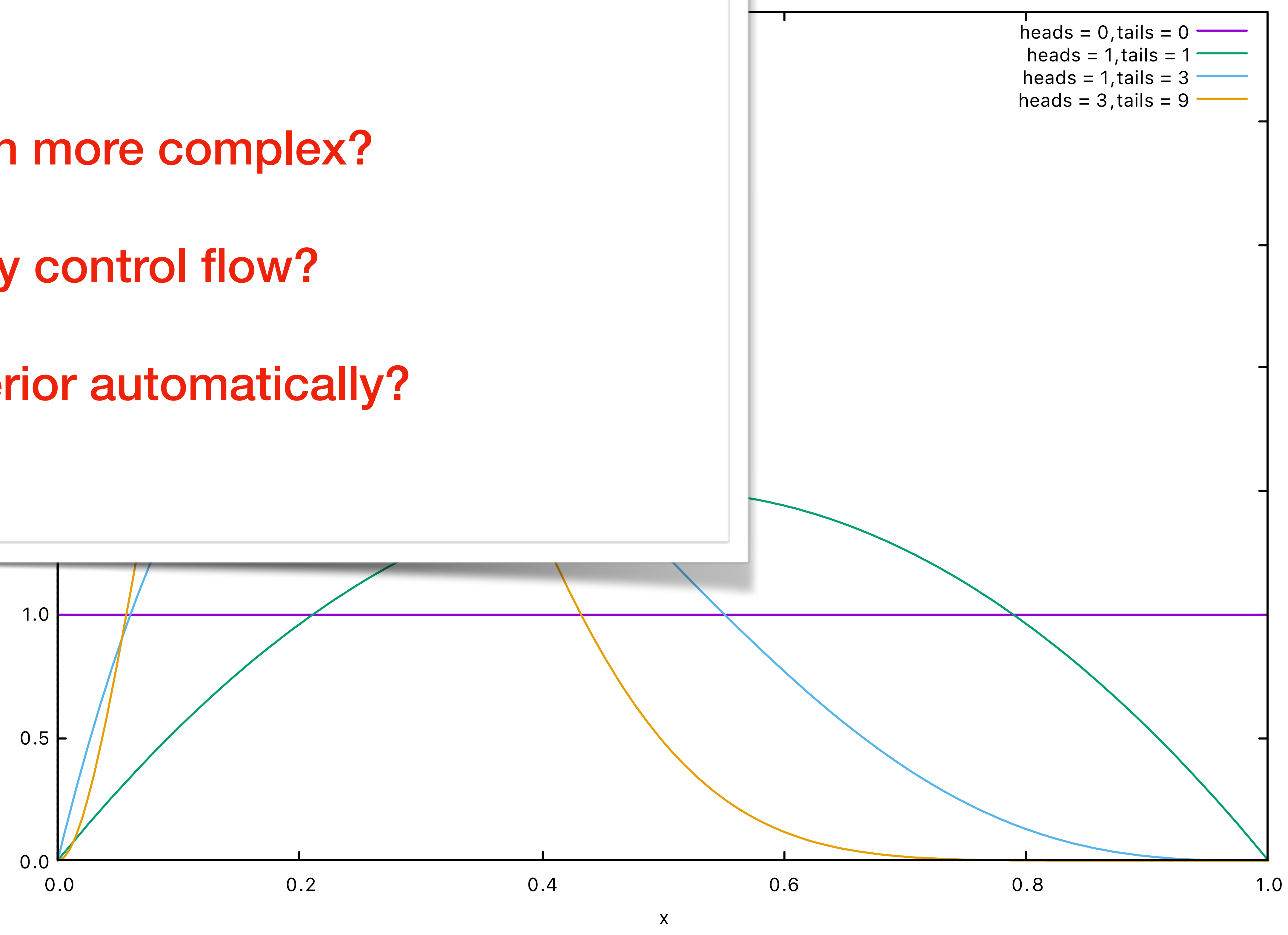
- Prior: $z \sim$ *Uniform*(
- Observations: for $i$
- Posterior: $p(z \mid x_1, .$

$z \sim$ *Beta*(#heads $+ 1,$

**What if the model is much more complex?**

**What if we use arbitrary control flow?**

**Can we compute the posterior automatically?**



# 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)

# Outline

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

I - Basic inference

- Rejection sampling
- Importance sampling

II - Typing

- Syntax: language and types
- Deterministic vs. probabilistic
- Guarding probabilistic constructs

II - Kernel Semantics

- Reminders: measure theory
- Types as measurable spaces
- Expressions as measures

TP : A short introduction to Stan

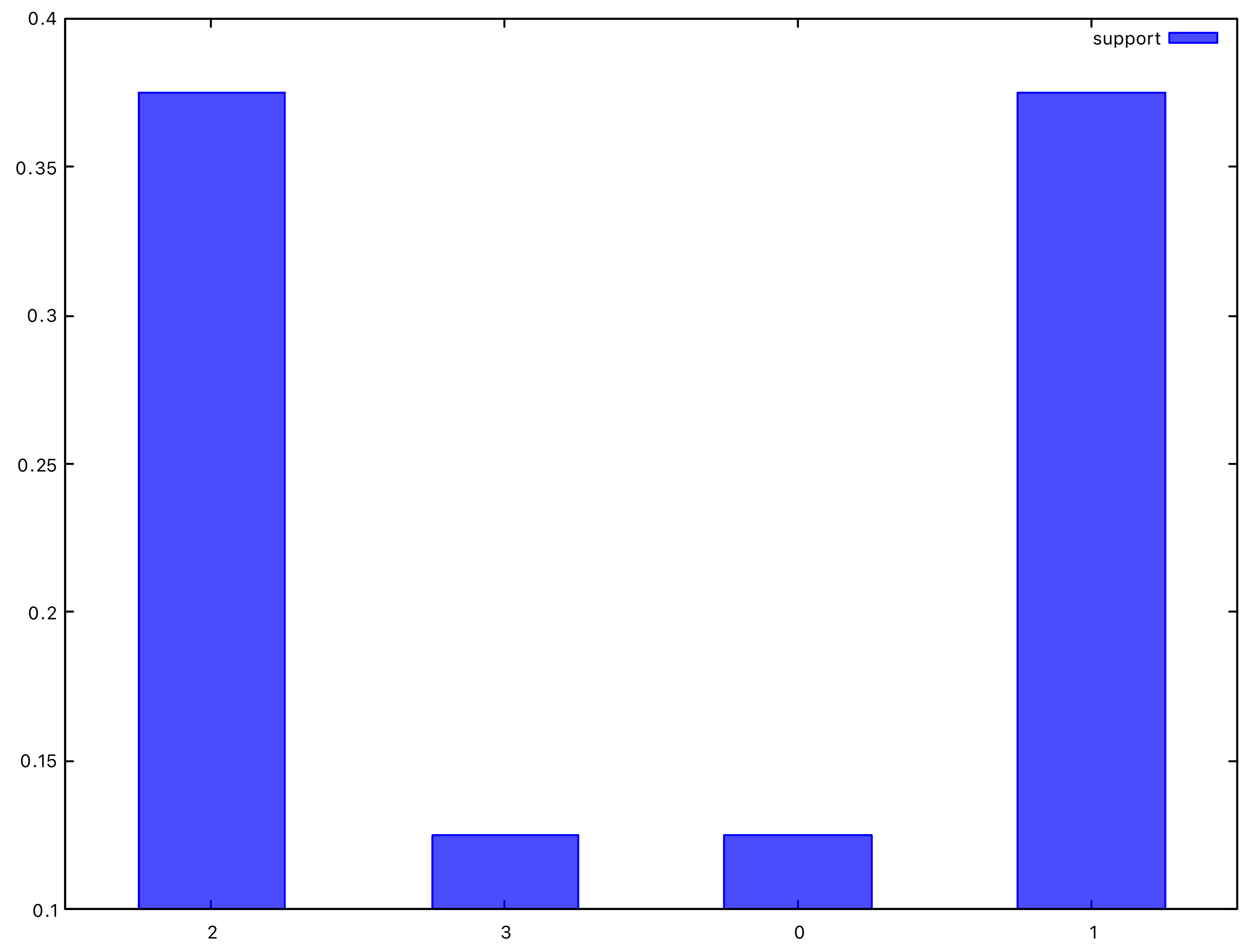
# Rejection Sampling

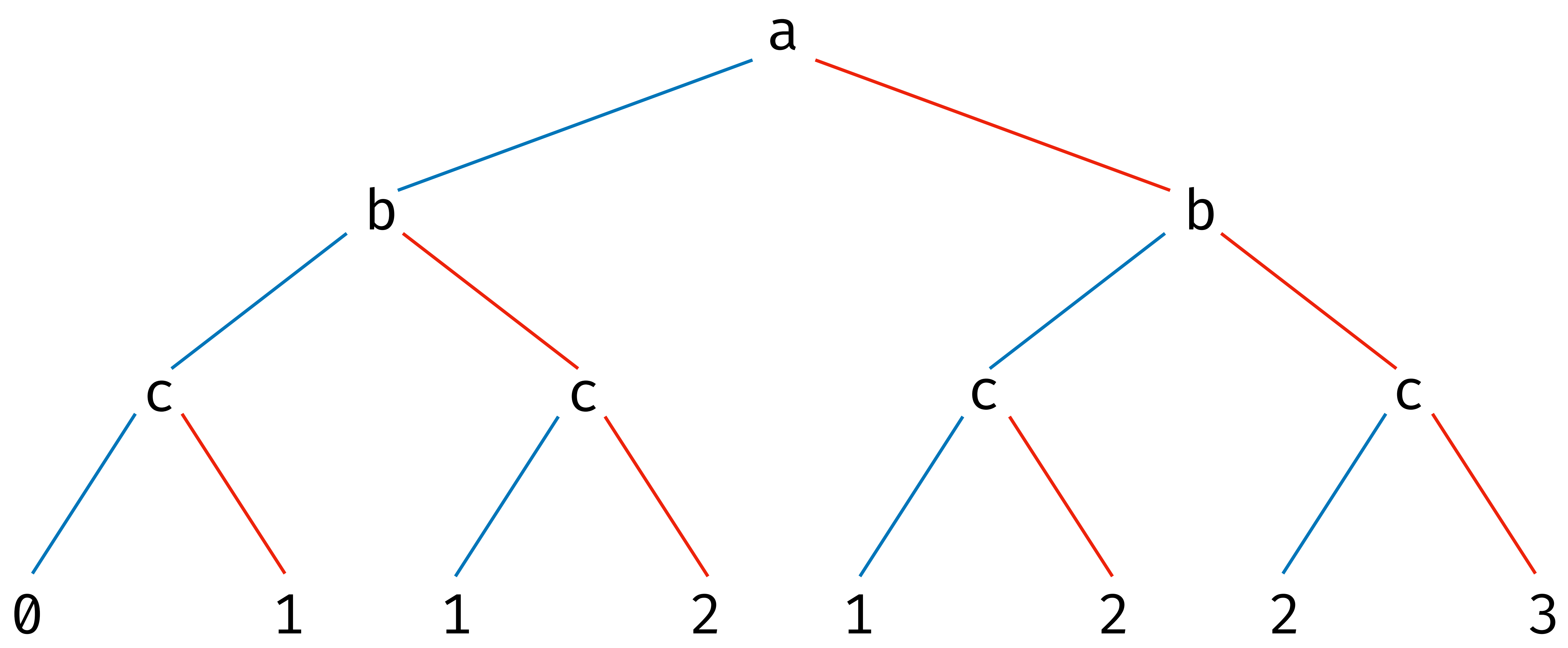
Probabilistic Programming Languages

# Example: 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
```

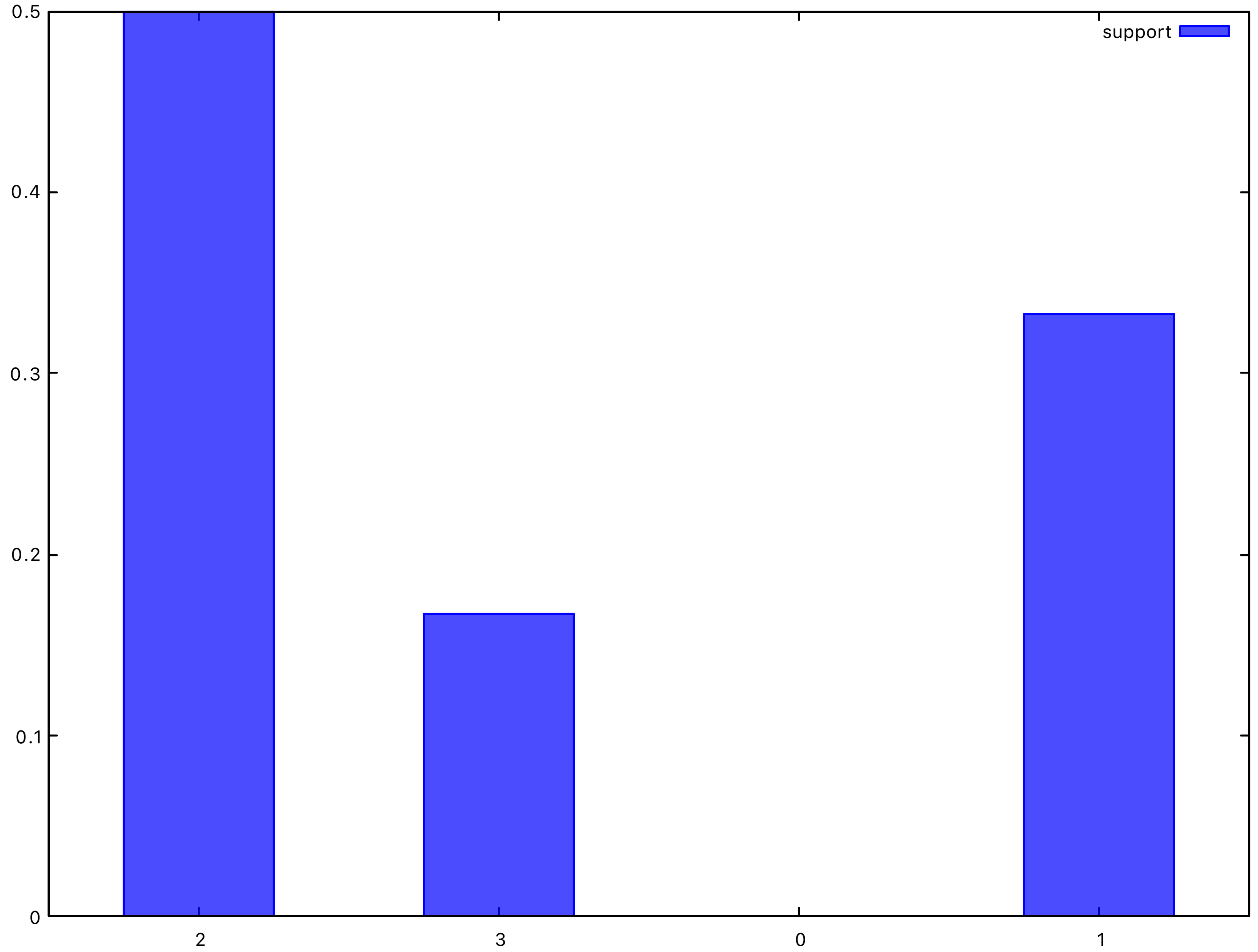


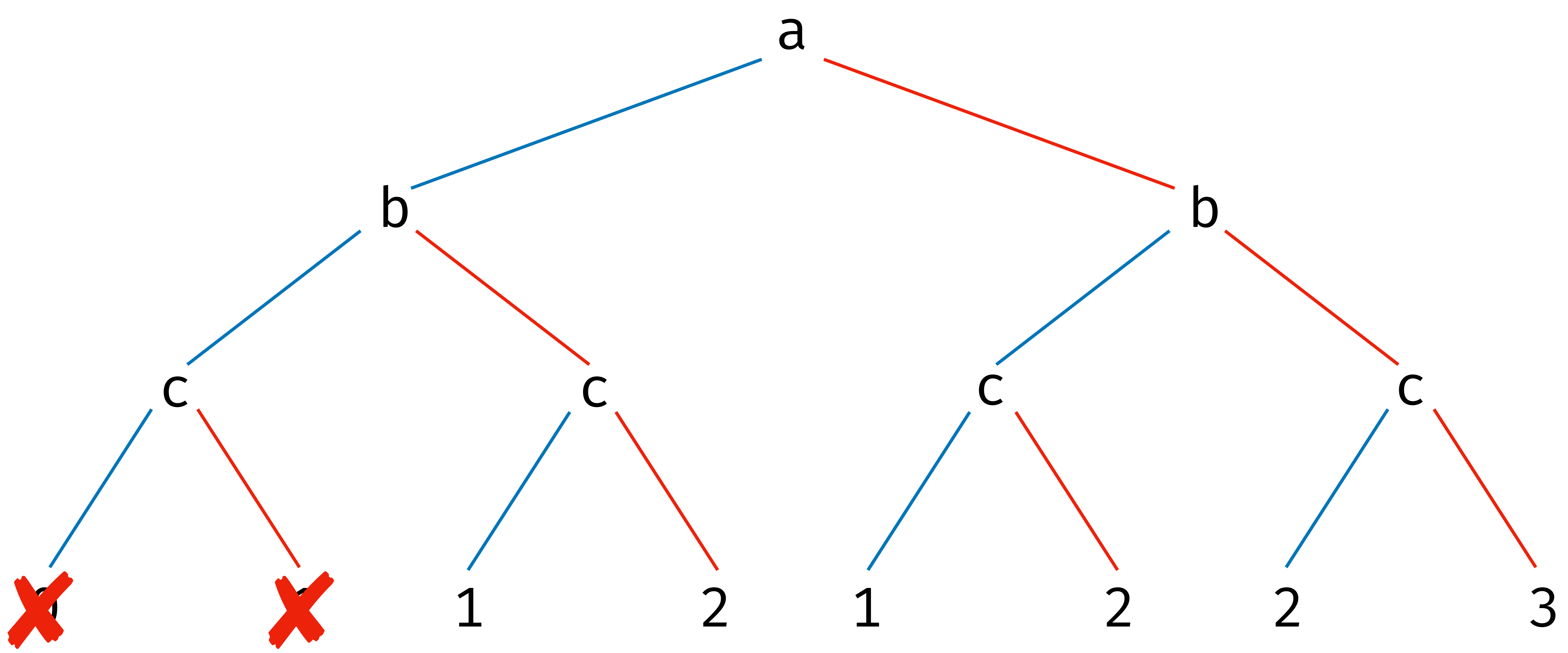



# Example: Funny Bernoulli

*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

*basic.ml*

```
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

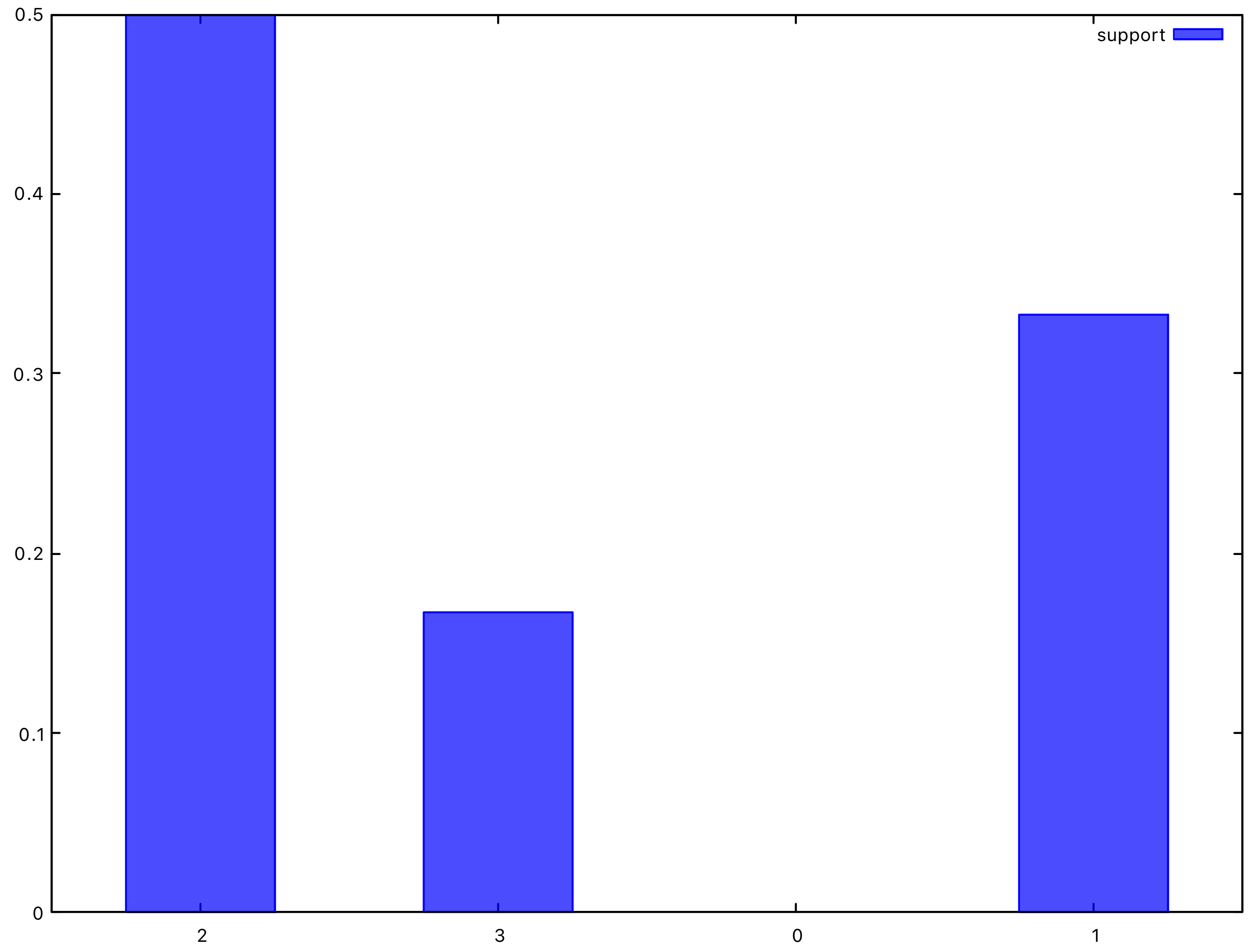
*basic.ml*

```
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
```

# Example: 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
```

# Example: 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
```

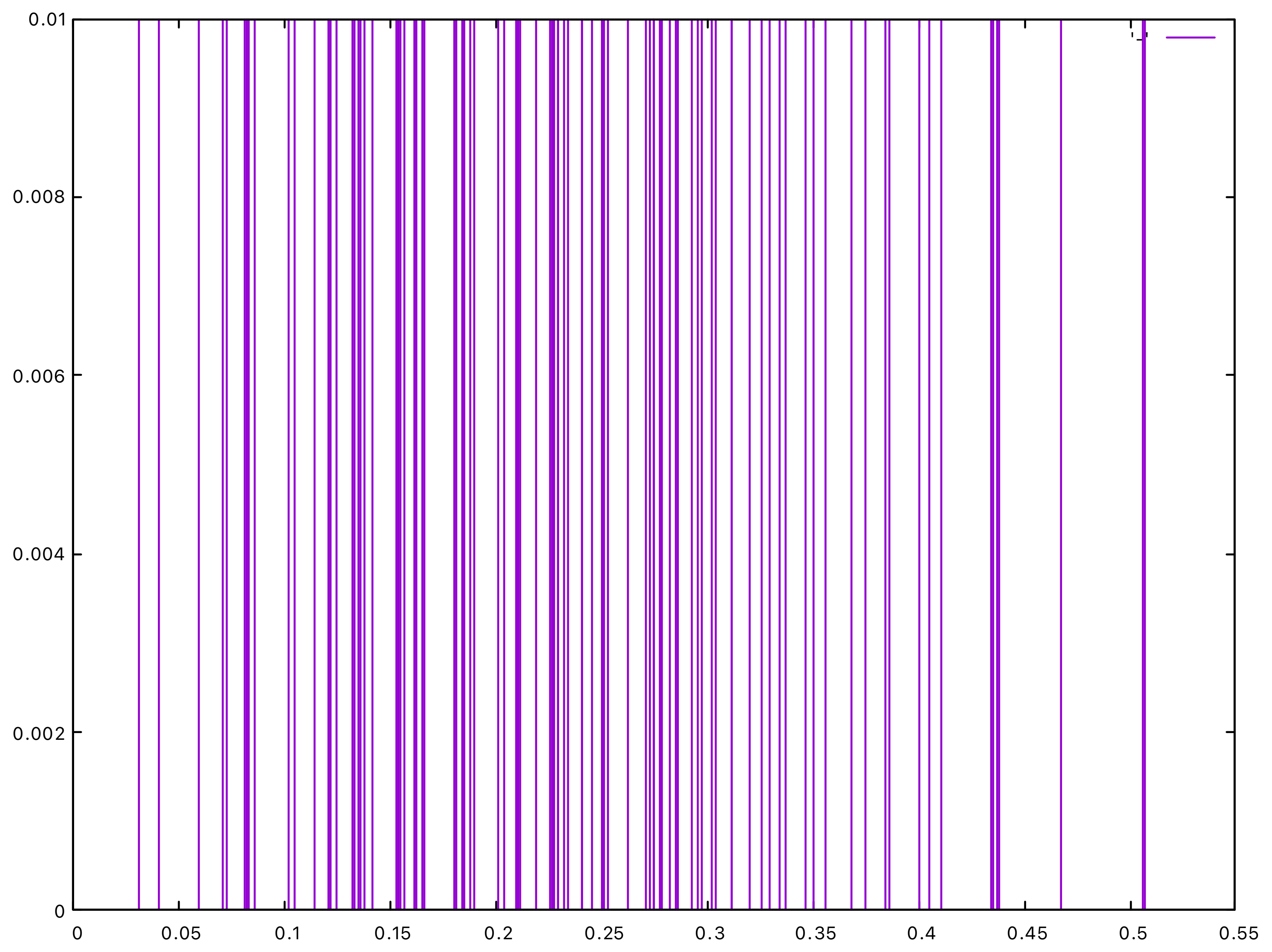
# Example: Coin

*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
```

100 particles



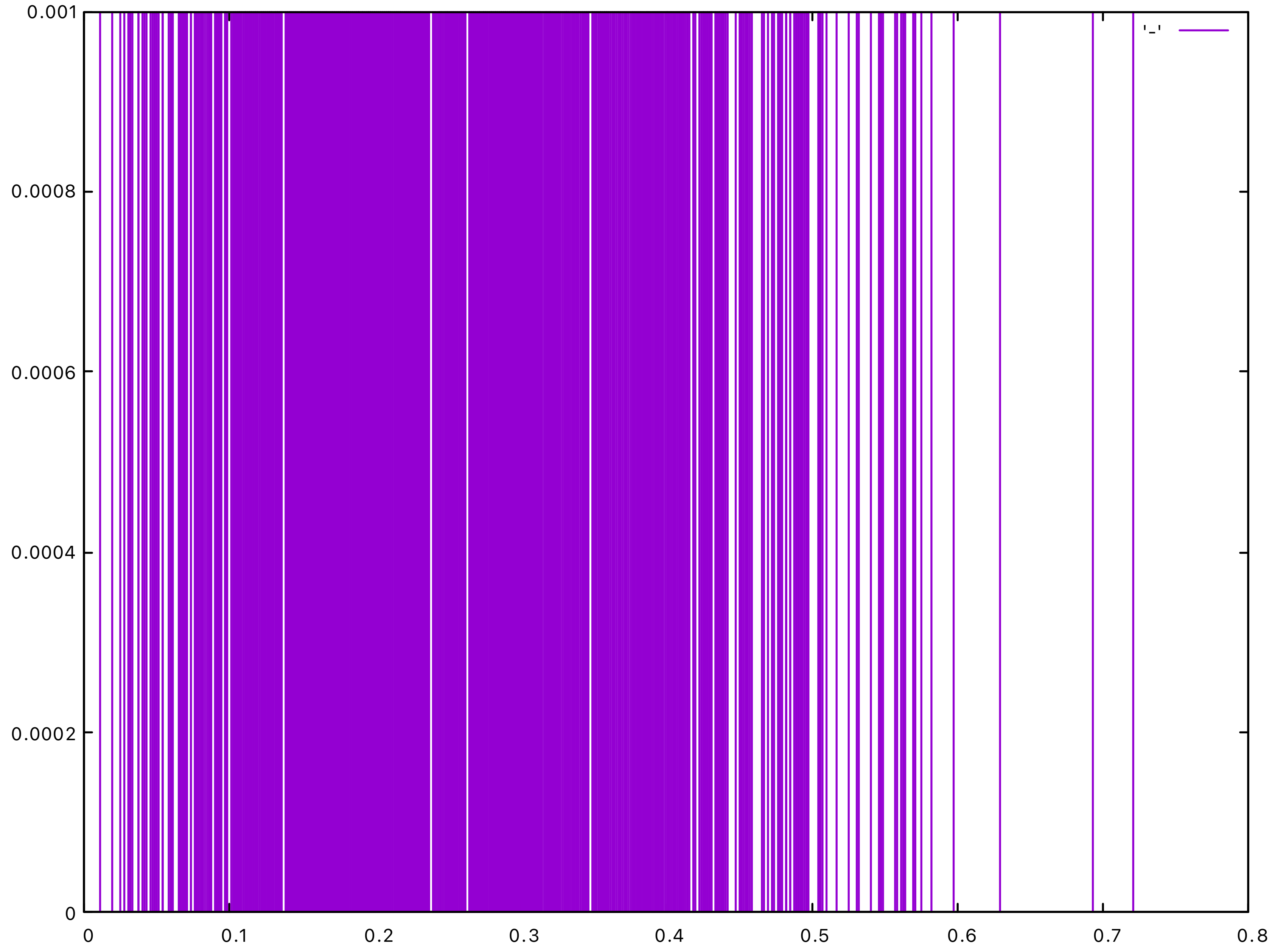
# Example: Coin

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
```

1000 particles



# Example: Coin

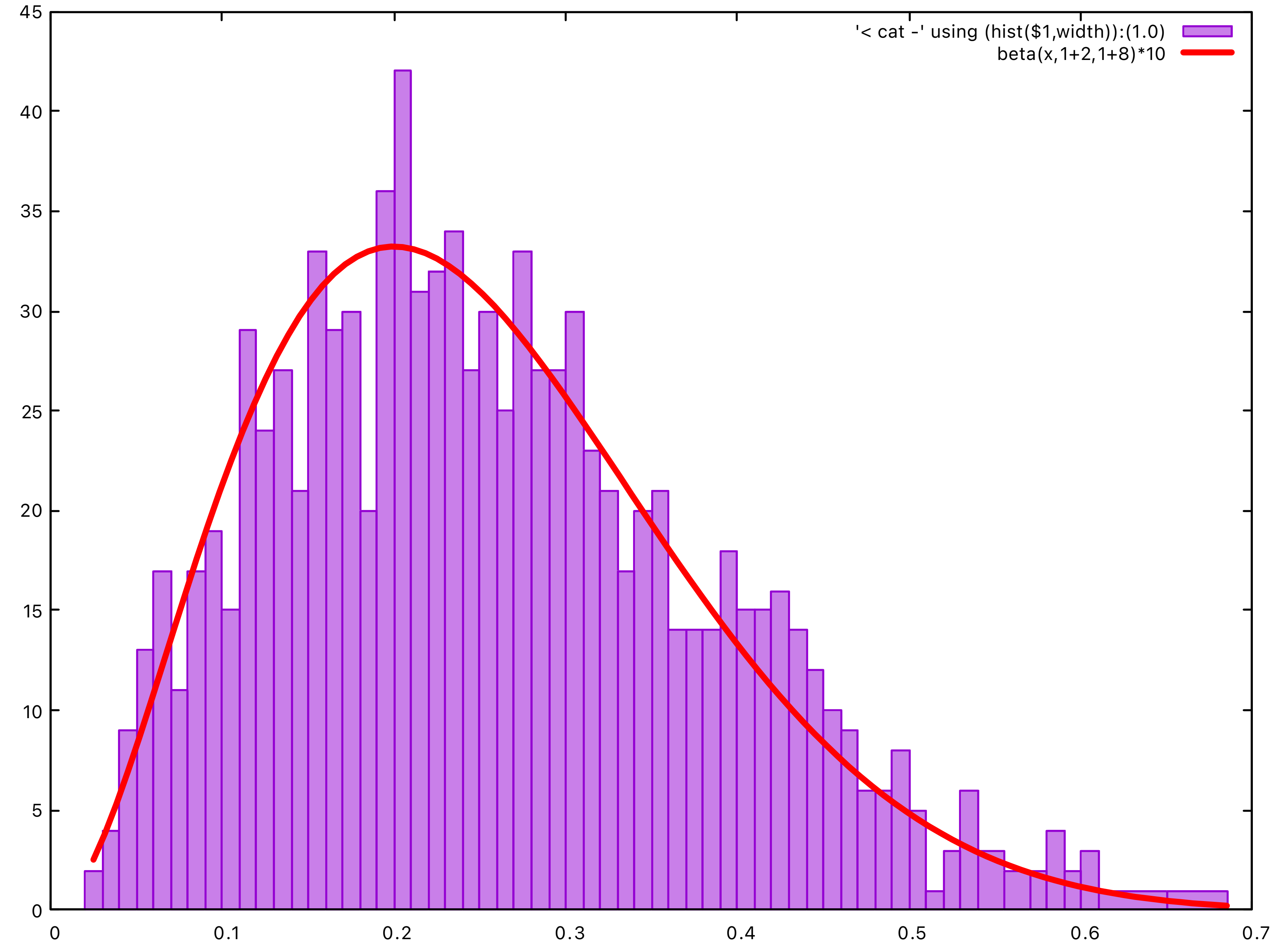
coin.ml

```
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!**

1000 particles



# Example: 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
```

**Never terminate!**

# Importance Sampling

Probabilistic Programming Languages

# 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)`

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

# Example: 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
```

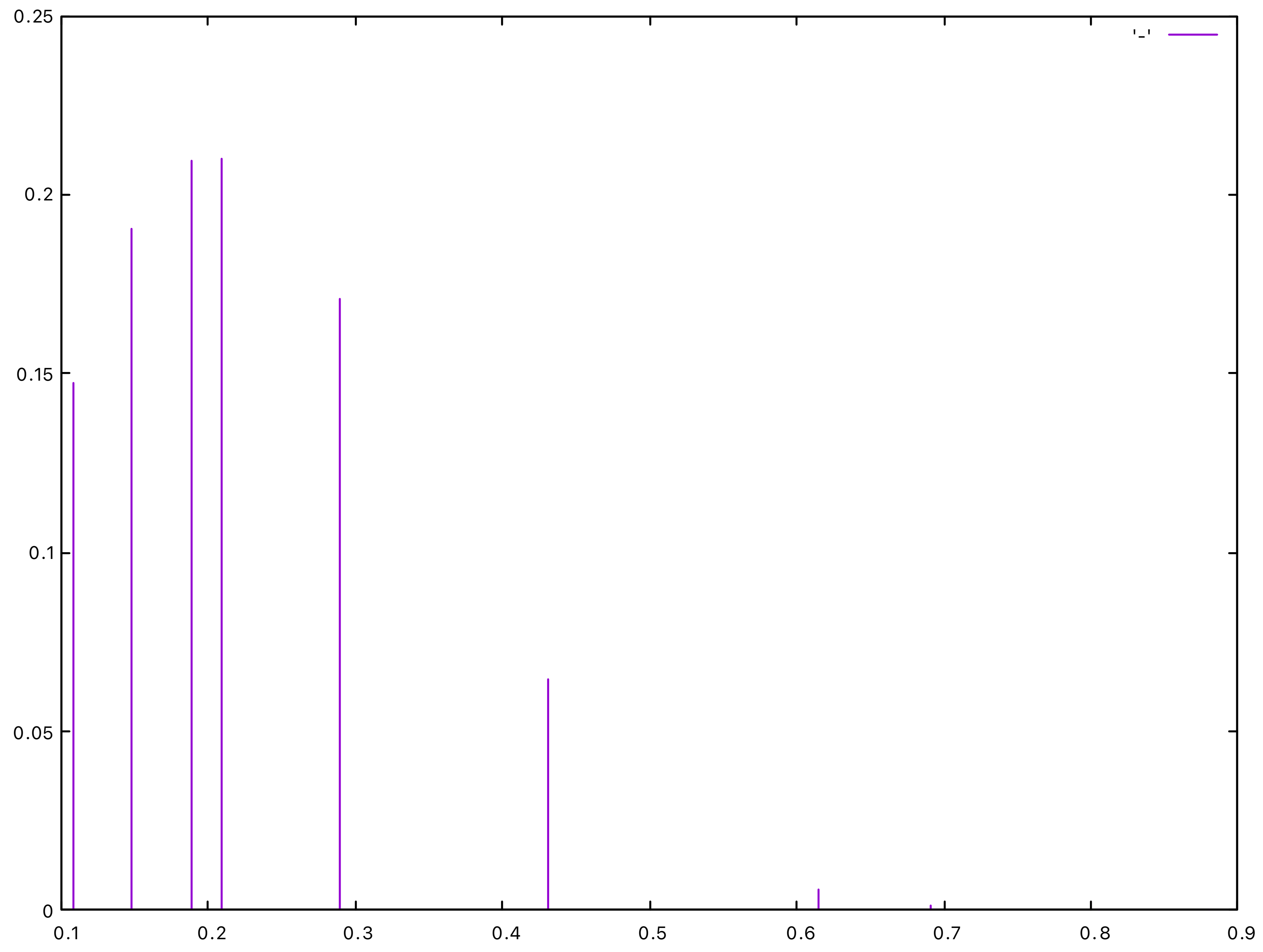
# Example: Coin

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
```

10 particles

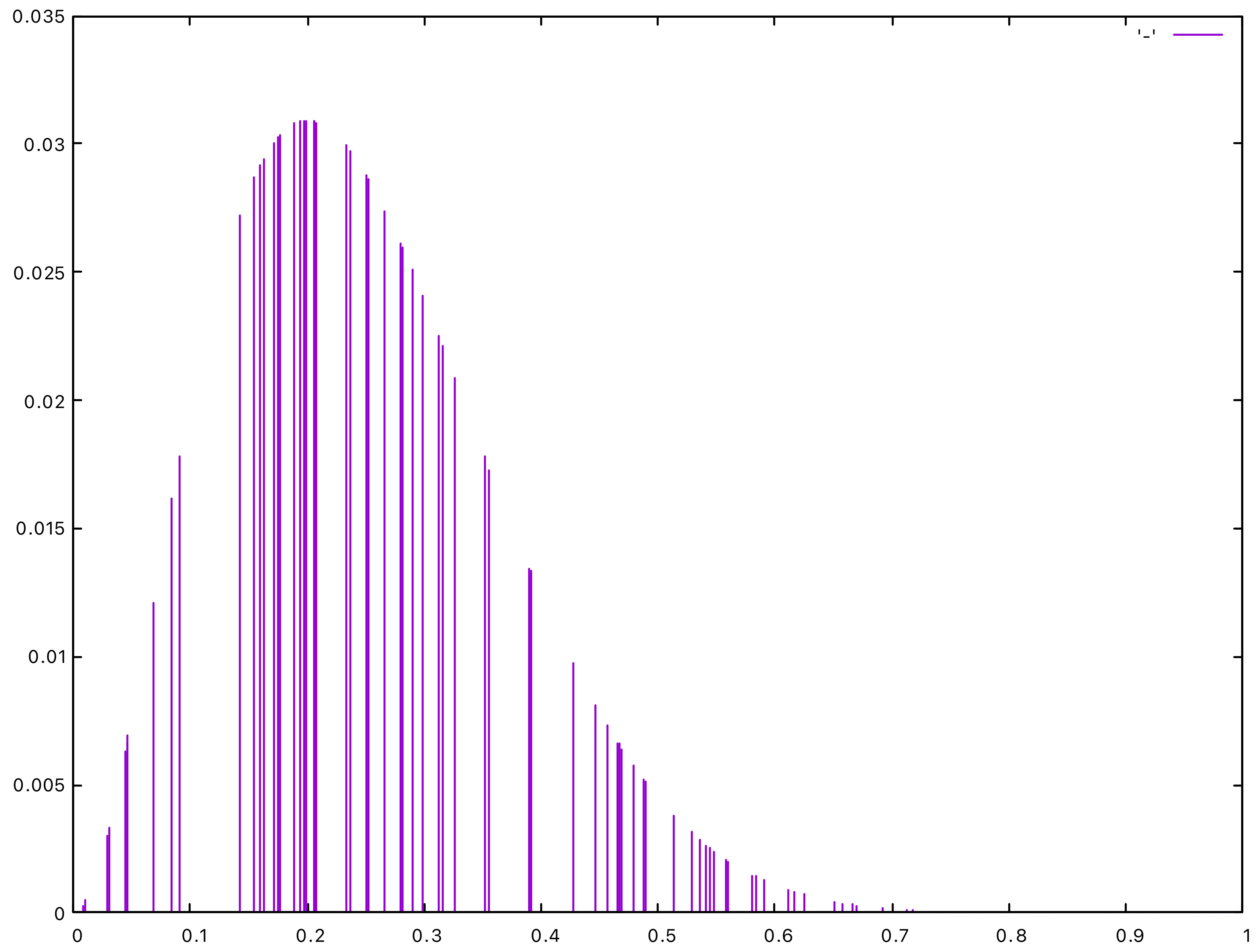


# Example: Coin

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
```

100 particles




```
❯ dune exec ./examples/coin.exe

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

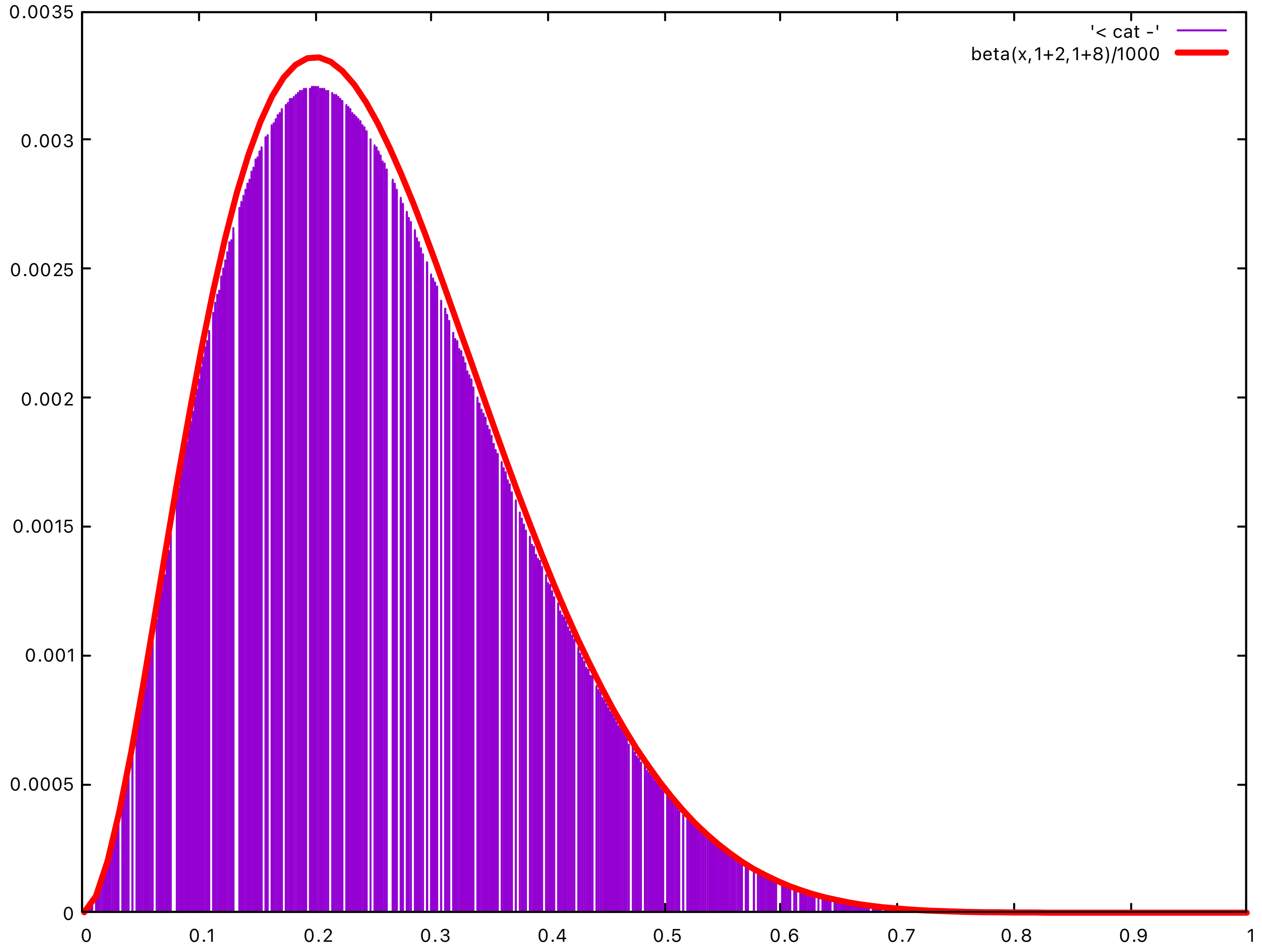
# Example: Coin

*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
```

1000 particles



# Conditioning

basic.ml

```
module Rejection_sampling = struct  ...

  (* Reject if [p] is not true. *)
  let assume prob p =
    if not p then raise Reject

  (* Assume [x] was sampled from [d]. *)
  let observe prob d x =
    let v = sample d in
    assume prob (v = x)
```

Hard conditioning

# Conditioning

basic.ml

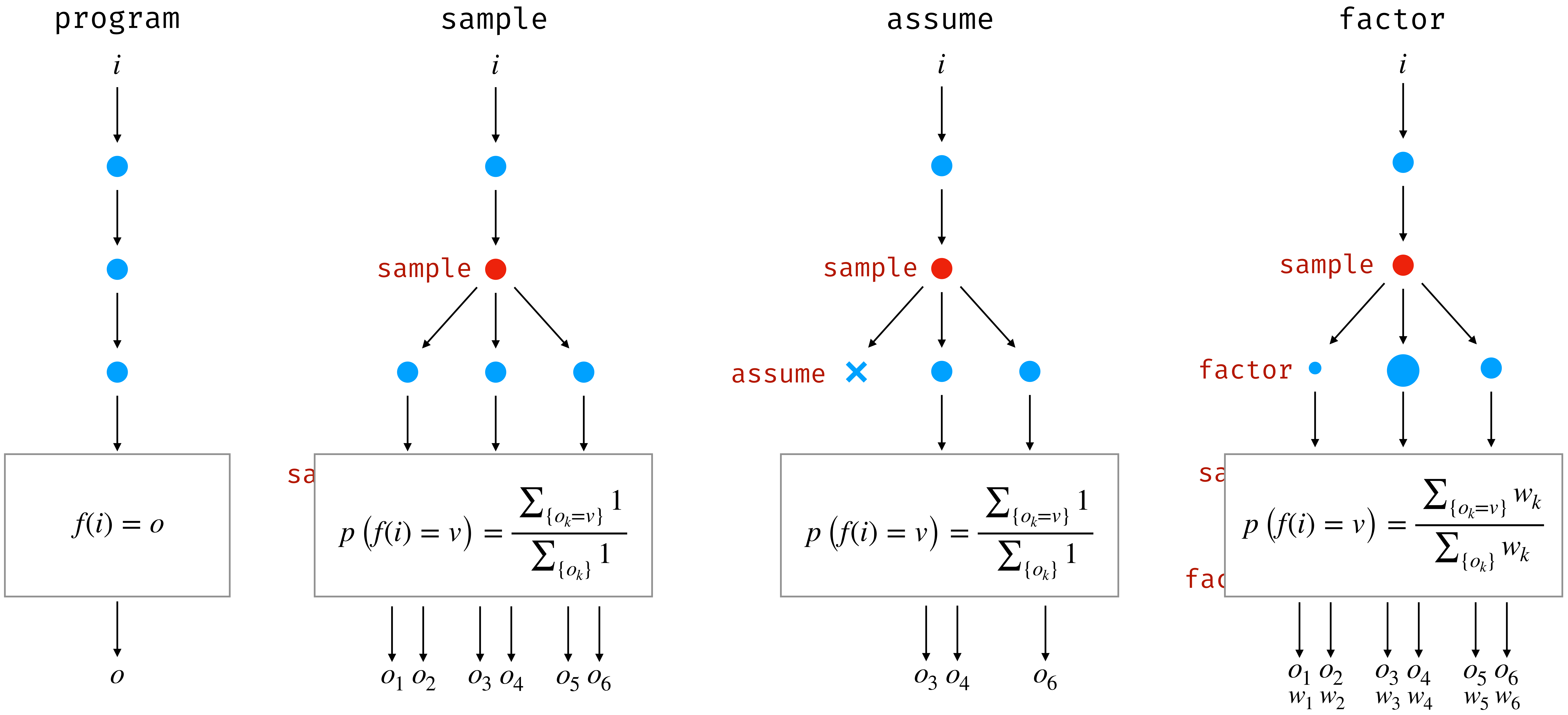
```
module Rejection_sampling = struct ...

  (* Reject if [p] is not true. *)
  let assume prob p =
    if not p then raise Reject

  (* Assume [x] was sampled from [d]. *)
  let observe prob d x =
    let v = sample d in
    assume prob (v = x)
```

Hard conditioning

```
module Importance_sampling = struct ...

  (* Update the (log)score. *)
  let factor prob s =
    prob.(prob.id) ← prob.(prob.id) +. s

  (* Assume [x] was sampled from [d]. *)
  let observe prob d x =
    factor prob (logpdf d x)
```

Soft conditioning

# infer : $(\alpha \to \beta) \to \alpha \to \beta$ dist



# Bayesian Reasoning

Bayesian Inference: learn parameters from data

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

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

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

```
let model (x1, ... , xn) =
  let theta = sample prior in
  let () = observe (likelihood theta) (x1, ... , xn) in
  theta

let posterior = infer model (x1, ... , xn)
```

Thomas Bayes (1701-1761)

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

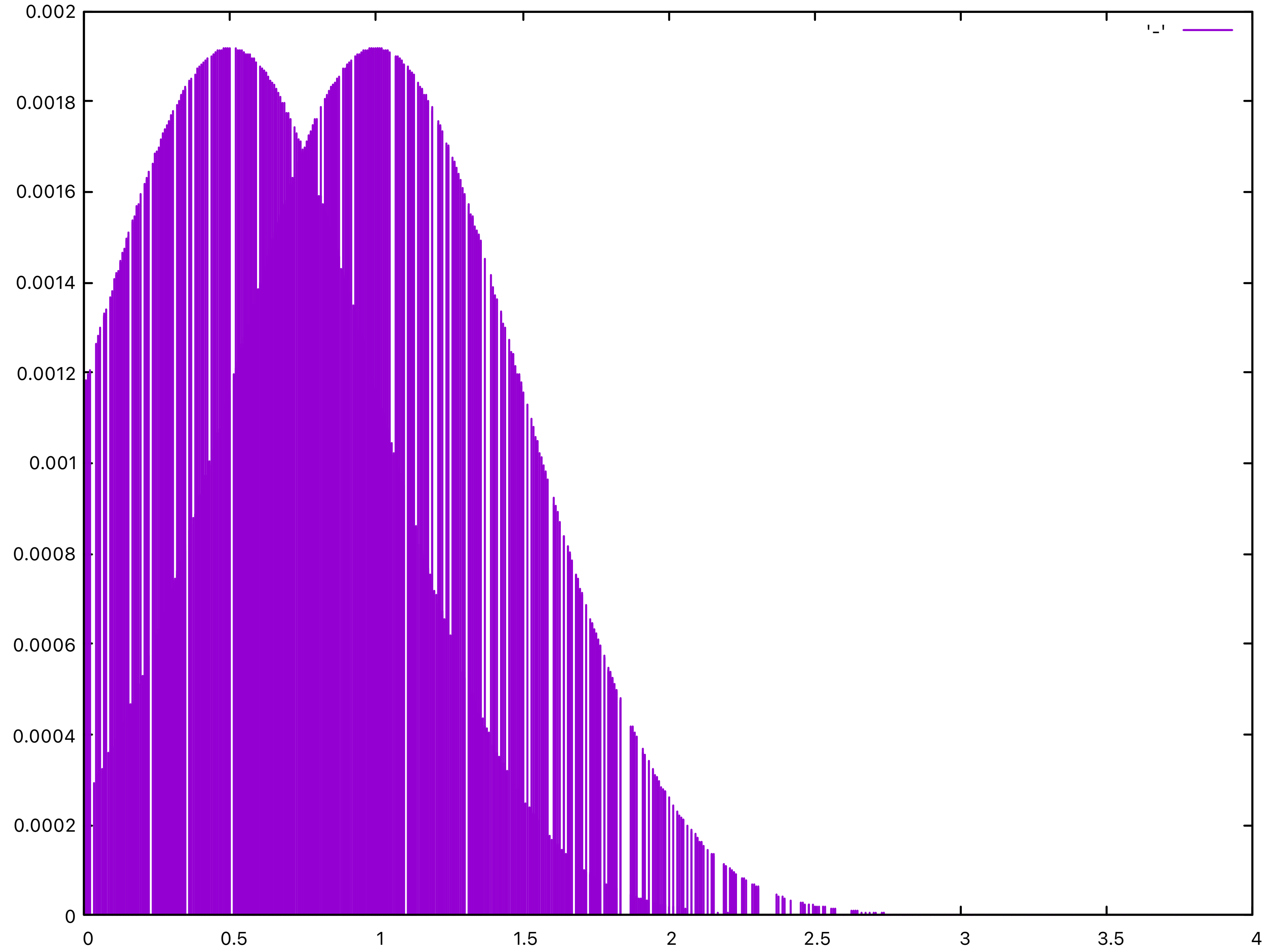
# Probabilistic programming

Programming with probability distributions

- Draw a sample from a distribution
- Condition the model on assumption

More general than classic Bayesian Reasoning

```
let rec weird () =
  let b = sample (bernoulli ~p:0.5) in
  let mu = if (b = 1) then 0.5 else 1.0 in
  let theta = sample (gaussian ~mu ~sigma:1.0) in
  if theta > 0. then
    observe (gaussian ~mu ~sigma:0.5) theta;
    theta
  else weird ()

let weird_dist = infer weird ()
```



# Typing

## Probabilistic Programming Languages

# Language and types

Simplified syntax

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

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

Types

$t$ ::= unit | bool | float | $t$ dist | $t$ dist$^*$ | $t \times t$ | $t \to t$

- $t$ dist : distribution over values of type $t$
- $t$ dist$^*$: distribution with densities (pdf$(d) : V \to [0, \infty)$ is defined)

# Types and kinds

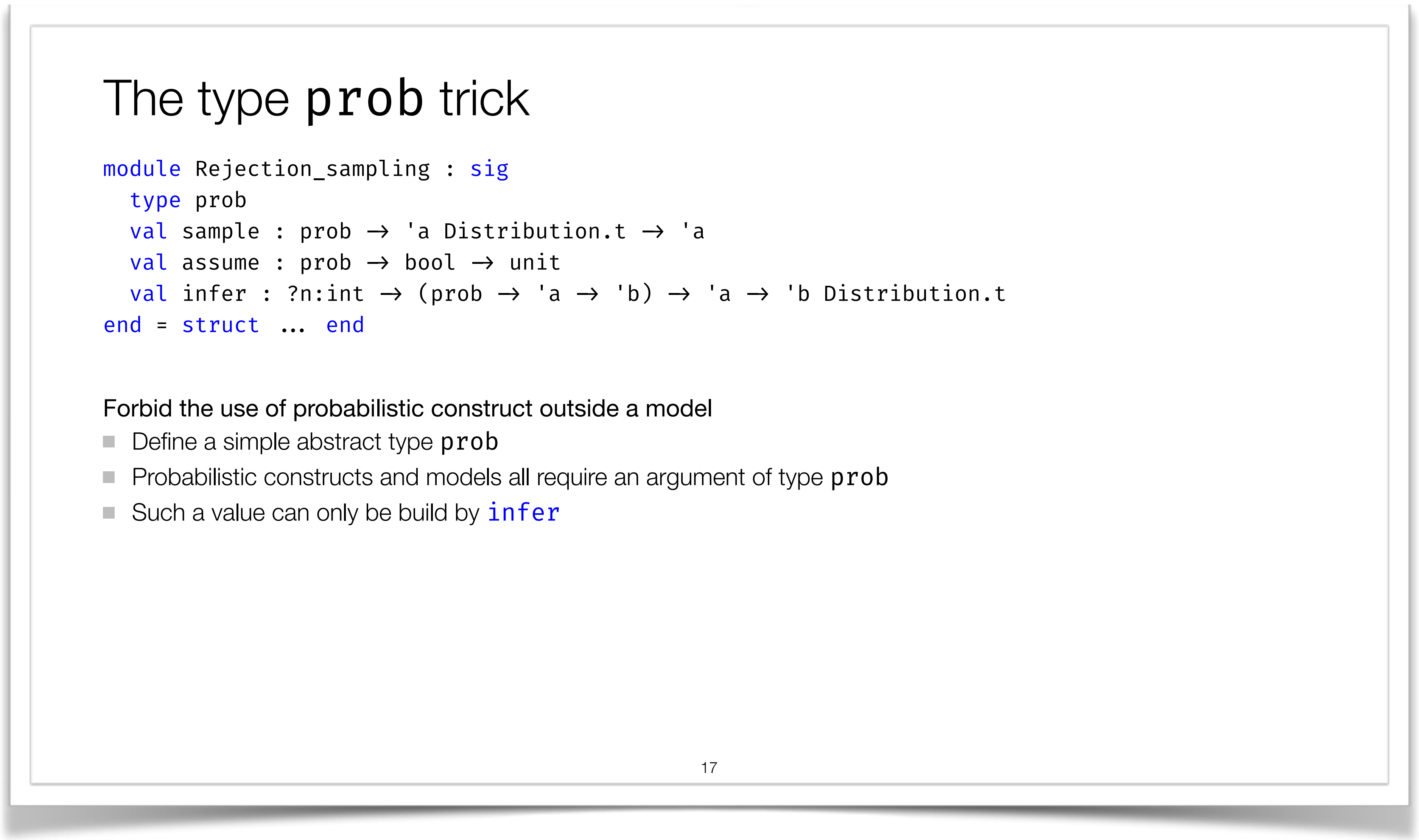
expression

Context

type

$$G \vdash^{k} e : t$$

kind: $D$ (deterministic) | $P$ (probabilistic)



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`

17

Kind P guards what can be expressed in a probabilistic model

# Typing declarations

$$\frac{G \vdash^{D} e : t}{G \vdash^{D} \texttt{let}\ p\ \texttt{=}\ e : G + [p \leftarrow t]}$$

$$\frac{k \in \{D, P\} \qquad G + [p \leftarrow t_1] \vdash^{k} e : t_2}{G \vdash^{D} \texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ p \to e : G + \left[f \leftarrow (t_1 \to^{k} t_2)\right]}$$

Declarations are deterministic
Functions can be D or P

$$\frac{G \vdash^{D} d_1 : G_1 \qquad G_1 \vdash^{D} d_2 : G_2}{G \vdash^{D} d_1\ d_2 : G_2}$$

# Typing probabilistic constructs

$$\frac{G \vdash^P e : t}{G \vdash^D \texttt{infer}(e) : t\ \texttt{dist}}$$

$$\frac{G \vdash^D e : t\ \texttt{dist}}{G \vdash^P \texttt{sample}(e) : t}$$

$$\frac{G \vdash^D e : \texttt{float}}{G \vdash^P \texttt{factor}(e) : \texttt{unit}}$$

$$\frac{G \vdash^D e_1 : t\ \texttt{dist}^* \qquad G \vdash^D e_2 : t}{G \vdash^P \texttt{observe}(e_1, e_2) : \texttt{unit}}$$

$$\frac{G \vdash^D e : t}{G \vdash^P e : t}$$

$$\frac{G \vdash^D e : t\ \texttt{dist}^*}{G \vdash^D e : t\ \texttt{dist}}$$

Subtyping

# Typing expressions

$$\frac{\textit{typeOf}(c) = t}{G \vdash^D c : t}$$

$$\frac{G(x) = t}{G \vdash^D x : t}$$

$$\frac{G \vdash^D e_1 : t_1 \qquad G \vdash^D e_2 : t_2}{G \vdash^D (\, e_1 \, \mathtt{,} \, e_2 \,) : t_1 \times t_2}$$

$$\frac{\textit{typeOf}(\textit{op}) = t_1 \rightarrow^D t_2 \qquad G \vdash^D e : t_1}{G \vdash^D \textit{op}(\, e \,) : t_2}$$

$$\frac{G(f) = t_1 \rightarrow^k t_2 \qquad G \vdash^D e : t_1}{G \vdash^k f(\, e \,) : t_2}$$

$$\frac{G \vdash^D e_1 : \mathtt{bool} \qquad G \vdash^k e_2 : t \qquad G \vdash^k e_3 : t}{G \vdash^k \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : t}$$

$$\frac{G \vdash^k e_1 : t_1 \qquad G + [p \leftarrow t_1] \vdash^k e_2 : t_2}{G \vdash^k \mathtt{let}\ p\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 : t_2}$$

Polymorphic kind

# Example : Coin

coin.ml

```
let coin (x1, … , xn) =
  let z = sample (uniform (0., 1.)) in
  observe (bernoulli (z), x1);
  … ;
  observe (bernoulli (z), xn);
  z

let _ =
  let d = infer (coin (1; 1; 0; 0; … )) in
  plot (d)
```

$[\mathtt{coin} : ???]$

$[\mathtt{x1} : \alpha_1, \ldots, \mathtt{xn} : \alpha_n] \vdash^P \mathtt{z} : \mathtt{float}$

$[\mathtt{x1} : \mathtt{int}, \ldots, \mathtt{xn} : \alpha_n, z : \mathtt{float}] \vdash^P \_ : \mathtt{unit}$

$[\mathtt{x1} : \mathtt{int}, \ldots, \mathtt{xn} : \mathtt{int}, z : \mathtt{float}] \vdash^P \_ : \mathtt{unit}$

$[\mathtt{x1} : \mathtt{int}, \ldots, \mathtt{xn} : \mathtt{int}, z : \mathtt{float}] \vdash^P \_ : \mathtt{float}$

# Example : Coin

coin.ml

```
let coin (x1, ..., xn) =
  let z = sample (uniform (0., 1.)) in
  observe (bernoulli (z), x1);
  ... ;
  observe (bernoulli (z), xn);
  z

let _ =
  let d = infer (coin (1; 1; 0; 0; ...)) in
  plot (d)
```

$[\mathtt{coin} : (\mathtt{int} \times \cdots \times \mathtt{int}) \to^{P} \mathtt{float}]$

$[\mathtt{x1} : \alpha_1, \ldots, \mathtt{xn} : \alpha_n] \vdash^{P} \mathtt{z} : \mathtt{float}$

$[\mathtt{x1} : \mathtt{int}, \ldots, \mathtt{xn} : \alpha_n, z : \mathtt{float}] \vdash^{P} \_ : \mathtt{unit}$

$[\mathtt{x1} : \mathtt{int}, \ldots, \mathtt{xn} : \mathtt{int}, z : \mathtt{float}] \vdash^{P} \_ : \mathtt{unit}$

$[\mathtt{x1} : \mathtt{int}, \ldots, \mathtt{xn} : \mathtt{int}, z : \mathtt{float}] \vdash^{P} \_ : \mathtt{float}$

$[\mathtt{coin} : (\mathtt{int} \times \cdots \times \mathtt{int}) \to^{P} \mathtt{float}] \vdash^{D} \mathtt{d} : \mathtt{float\ dist}$

$[\mathtt{coin} : (\mathtt{int} \times \cdots \times \mathtt{int}) \to^{P} \mathtt{float}, \mathtt{d} : \mathtt{float\ dist}] \vdash^{D} \_ : \mathtt{unit}$

# Example : Coin

*coin.ml*

```
let coin prob (x1, ... , xn) =
  let z = sample prob (uniform (0., 1.)) in
  observe prob (bernoulli (z), x1);
  ... ;
  observe prob (bernoulli (z), xn);
  z

let _ =
  let d = infer (coin (1; 1; 0; 0; ... )) in
  plot (d)
```

$[\mathtt{coin} : (\mathtt{int} \times \cdots \times \mathtt{int}) \rightarrow^P \mathtt{float}]$

$[\mathtt{x1} : \alpha_1, \ldots, \mathtt{xn} : \alpha_n] \vdash^P \mathtt{z} : \mathtt{float}$

$[\mathtt{x1} : \mathtt{int}, \ldots, \mathtt{xn} : \alpha_n, z : \mathtt{float}] \vdash^P \_ : \mathtt{unit}$

$[\mathtt{x1} : \mathtt{int}, \ldots, \mathtt{xn} : \mathtt{int}, z : \mathtt{float}] \vdash^P \_ : \mathtt{unit}$

$[\mathtt{x1} : \mathtt{int}, \ldots, \mathtt{xn} : \mathtt{int}, z : \mathtt{float}] \vdash^P \_ : \mathtt{float}$

$[\mathtt{coin} : (\mathtt{int} \times \cdots \times \mathtt{int}) \rightarrow^P \mathtt{float}] \vdash^D \mathtt{d} : \mathtt{float\ dist}$

$[\mathtt{coin} : (\mathtt{int} \times \cdots \times \mathtt{int}) \rightarrow^P \mathtt{float}, \mathtt{d} : \mathtt{float\ dist}] \vdash^D \_ : \mathtt{unit}$

# Reminders: Measure Theory

Probabilistic Programming Languages

# Measurable spaces

A $\sigma$-algebra on a set $X$ is a collection of subsets:

- containing $\emptyset$
- closed under complement
- closed under countable union

A measurable space is a pair $(X, \Sigma_X)$

- $X$: set
- $\Sigma_X$: measurable sets

Examples:

- $\emptyset$...
- Singleton: $\{\emptyset, \{()\}\}$
- Booleans: $\{\emptyset, \{\mathtt{true}\}, \{\mathtt{false}\}, \{\mathtt{true}, \mathtt{false}\}\}$
- Borel sets (intervals) on $\mathbb{R}$
- Product : $\Sigma_{A \times B} = \{(U, V) \mid X \in \Sigma_A, Y \in \Sigma_B\}$

# Measure

A measure maps a measurable set to a positive score: $\mu : \Sigma_X \to [0, \infty)$

- $\mu(U) \geq 0$ for all set $U \in \Sigma_X$
- $\mu(\emptyset) = 0$
- $\mu(\bigcup_{i \in I} U_i) = \sum_{i \in I} \mu(U_i)$ for all countable collection $\bigcup_{i \in I} U_i$ of pairwise disjoint sets in $\Sigma_X$

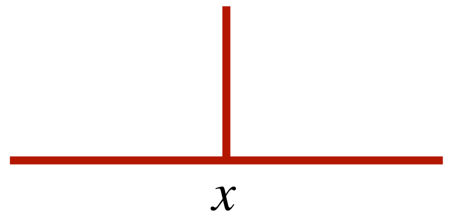
Probability distributions are normalized measure, i.e., $\mu(X) = 1$

Examples

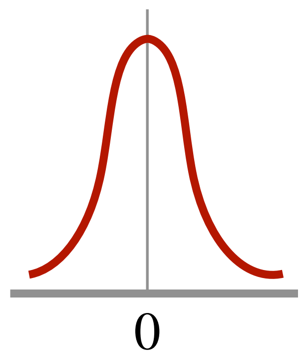
- Lebesgue measure
  $\lambda([a, b]) = b - a$
- Bernoulli distribution (discrete)
  $\mathcal{B}(0.3)(\{\mathtt{true}\}) = 0.3$
  $\mathcal{B}(0.3)(\{\mathtt{true}, \mathtt{false}\}) = 1$

- Dirac delta measure

$$\delta_x(U) = \begin{cases} 1 & \text{if } x \in U \\ 0 & \text{otherwise} \end{cases}$$




- Normal distribution
  $\mathcal{N}(0, 1)([0, 1]) \approx 0.34$
  $\mathcal{N}(0, 1)((-\infty, 0]) = 0.5$



# Measurable functions

$f : X \to Y$ is measurable if $f^{-1}(U) \in \Sigma_X$ for all $U \in \Sigma_Y$

Pushforward: transfer a measure from a measurable space to another one

- $f : X \to Y$ measurable
- $\mu : \Sigma_X \to [0, \infty)$
- $f_*(\mu) : \Sigma_Y \to [0, \infty)$

Examples

- $\mu : \Sigma_{A \times B} \to [0, \infty)$
- $\pi_{1*}(\mu) : \Sigma_A \to [0, \infty)$
- $\pi_{2*}(\mu) : \Sigma_B \to [0, \infty)$

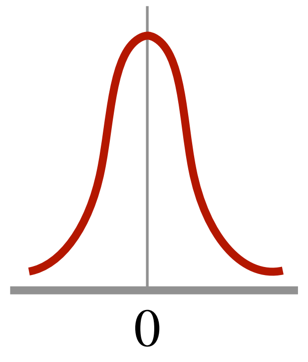
# Integration

Given $f, g : X \to [0,\infty)$ measurable functions and a measure $\mu : \Sigma_X \to [0,\infty)$

- $\displaystyle\int f(x)\mu(dx) \in [0,\infty)$
- $\displaystyle\int \mu(dx)\delta_x(U) = \int_U \mu(dx) = \mu(U)$ for all $U \in \Sigma_X$
- $\displaystyle\int f(x) r \mu(dx) = r \int f(x)\mu(dx)$ for $r \in \mathbb{R}$
- $\displaystyle\int (f(x) + g(x))\, \mu(dx) = \int f(x)\mu(dx) + \int g(x)\mu(dx)$

We can also define new measures from integration

- $f(x) = \dfrac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$
- $\displaystyle\int_U f(x)\lambda(dx) = \mathcal{N}(0,1)(U)$ where $\lambda$ is the Lebesgue measure



# Kernel

A kernel $k : X \times \Sigma_Y \to [0, \infty)$ is a function such that:

- $k(x, \_) : \Sigma_Y \to [0, \infty)$ for all $x \in X$ is a measure
- $k(\_, U) : X \to [0, \infty)$ for all $U \in \Sigma_Y$ is measurable

A probability kernel is such that $k(x, Y) = 1$ for all $x \in X$

We now have everything to define
a kernel based denotational semantics!

# Semantics

## Probabilistic Programming Languages

# Types as mesurable spaces

A ground type $t$ is interpreted as a measurable space $[\![t]\!]$

- $[\![\texttt{unit}]\!]$: discrete measurable space over the unique value $(\,)$
- $[\![\texttt{bool}]\!]$: discrete measurable space with the two values $\texttt{true}$, $\texttt{false}$
- $[\![\texttt{float}]\!]$: reals with its Borel sets (intervals)

- $A \times B$ product space $[\![A]\!] \times [\![B]\!]$
  with the rectangles $U \times V$ for $U \in \Sigma_A$ and $B \in \Sigma_B$

- $[\![t\ \texttt{dist}]\!]$: set of probability measures on $[\![t]\!]$
  with the sets $\{\mu \mid \mu(U) < r\}$ for $U \in \Sigma_{[\![t]\!]}$ and $r \in [0, 1]$ (Giry monad)

- A context $G = [x_1 : A_1, \ldots, x_n : A_n]$ maps variables to types
  $[\![G]\!] = \prod_{i=1}^{n} [\![A_i]\!]$ is also a measurable space (product of all variables spaces)

What about function types?

# Deterministic vs. probabilistic

Deterministic semantics $G \vdash^D e : t$

- Classic denotational semantics
- Environments: $\phi$ (global declarations), $\gamma$ (local variables)
- Given the declarations $\phi$, $[\![e]\!]^\phi : \Gamma \to t$ is a measurable function
- $[\![e]\!]^\phi_\gamma$ is a value of type $t$

Probabilistic semantics $G \vdash^P e : t$

- Given the declarations $\phi$, expressions are interpreted as kernels
- $\{\![e]\!\}^\phi : \Gamma \times \Sigma_{[\![t]\!]} \to [0, \infty)$
- $\{\![e]\!\}^\phi_\gamma$ is a measure on values of type $t$

# (Un)normalized measures

set of possible outcomes

model

$$\{\!\![ e ]\!\!\}^{\phi}_{\gamma} : \Sigma_{[\![ t ]\!]} \to [0, \infty)$$

score

environments

Unnormalized measure

# (Un)normalized measures

set of possible outcomes

model

$$\{\!\![ e ]\!\!\}^{\phi}_{\gamma} : \Sigma_{[\![t]\!]} \rightarrow [0, \infty)$$

Unnormalized measure

score

environments

$$[\![\texttt{infer}(e)]\!]^{\phi}_{\gamma} = \frac{\{\!\![ e ]\!\!\}^{\phi}_{\gamma}}{\{\!\![ e ]\!\!\}^{\phi}_{\gamma}\left([\![\mathit{typeOf}(e)]\!]\right)}$$

Distribution

normalize over all possible values

# Deterministic semantics

$$
\begin{aligned}
[\![\texttt{let}\ p\ \texttt{=}\ e]\!]^{\phi} &= \phi + \left[p \leftarrow [\![e]\!]^{\phi}_{\emptyset}\right] \\
[\![\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ p \rightarrow e]\!]^{\phi} &= \phi + \left[f \leftarrow \lambda v.\ [\![e]\!]^{\phi}_{[p\leftarrow v]}\right] \\
[\![d_1\ d_2]\!]^{\phi} &= \mathit{let}\ \phi_1 = \phi + [\![d_1]\!]^{\phi}\ \ \mathit{in}\ \ [\![d_2]\!]^{\phi_1}
\end{aligned}
$$

$$
\begin{aligned}
[\![c]\!]^{\phi}_{\gamma} &= c \\
[\![x]\!]^{\phi}_{\gamma} &= (\gamma + \phi)(x) \\
[\![\texttt{(}e_1\texttt{,}e_2\texttt{)}]\!]^{\phi}_{\gamma} &= ([\![e_1]\!]^{\phi}_{\gamma}, [\![e_2]\!]^{\phi}_{\gamma}) \\
[\![\mathit{op}\texttt{(}e\texttt{)}]\!]^{\phi}_{\gamma} &= \mathit{op}([\![e]\!]^{\phi}_{\gamma}) \\
[\![f\texttt{(}e\texttt{)}]\!]^{\phi}_{\gamma} &= \phi(f)([\![e]\!]^{\phi}_{\gamma}) \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]^{\phi}_{\gamma} &= \mathit{if}\ \ [\![e_1]\!]^{\phi}_{\gamma}\ \ \mathit{then}\ \ [\![e_2]\!]^{\phi}_{\gamma}\ \ \mathit{else}\ \ [\![e_3]\!]^{\phi}_{\gamma} \\
[\![\texttt{let}\ p\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!]^{\phi}_{\gamma} &= \mathit{let}\ v = [\![e_1]\!]^{\phi}_{\gamma}\ \ \mathit{in}\ \ [\![e_2]\!]^{\phi}_{\gamma+[p\leftarrow v]}
\end{aligned}
$$

# Probabilistic semantics

$$\llbracket \texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ p \to e \rrbracket^{\phi} = \phi + \left[ f \leftarrow \lambda v.\ \{\!\!\{ e \}\!\!\}^{\phi}_{[p \leftarrow v]} \right] \textit{ if kindOf}(e) = P$$

$$\begin{aligned}
\{\!\!\{ e \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ \delta_{\llbracket e \rrbracket^{\phi}_{\gamma}}(U) \textit{ if kindOf}(e) = D \\
\{\!\!\{ f(e) \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ \phi(f)(\llbracket e \rrbracket^{\phi}_{\gamma})(U) \\
\{\!\!\{ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ \textit{if}\ \llbracket e_1 \rrbracket^{\phi}_{\gamma}\ \textit{then}\ \{\!\!\{ e_2 \}\!\!\}^{\phi}_{\gamma}(U)\ \textit{else}\ \{\!\!\{ e_3 \}\!\!\}^{\phi}_{\gamma}(U) \\
\{\!\!\{ \texttt{let}\ p\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ \int_{\llbracket \textit{typeOf}(e_1) \rrbracket} \{\!\!\{ e_1 \}\!\!\}^{\phi}_{\gamma}(dv)\ \{\!\!\{ e_2 \}\!\!\}^{\phi}_{\gamma + [p \leftarrow v]}
\end{aligned}$$

$$\begin{aligned}
\{\!\!\{ \texttt{sample}(e) \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ \llbracket e \rrbracket^{\phi}_{\gamma}(U) \\
\{\!\!\{ \texttt{factor}(e) \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ \llbracket e \rrbracket^{\phi}_{\gamma} \cdot \delta_{()}(U) \\
\{\!\!\{ \texttt{observe}(e_1, e_2) \}\!\!\}^{\phi}_{\gamma} &= \lambda U.\ \mathrm{pdf}(\llbracket e_1 \rrbracket^{\phi}_{\gamma})(\llbracket e_2 \rrbracket^{\phi}_{\gamma}) \cdot \delta_{()}(U)
\end{aligned}$$

$$\llbracket \texttt{infer}(e) \rrbracket^{\phi}_{\gamma} = \begin{cases} \dfrac{\lambda U.\ \{\!\!\{ e \}\!\!\}^{\phi}_{\gamma}(U)}{\{\!\!\{ e \}\!\!\}^{\phi}_{\gamma}(\llbracket \textit{typeOf}(e) \rrbracket)} & \textit{if } 0 < \{\!\!\{ e \}\!\!\}^{\phi}_{\gamma}(\llbracket \textit{typeOf}(e) \rrbracket) < \infty \\ \textit{Error} & \textit{otherwise} \end{cases}$$

Careful with 0, and ∞...

# Example : Gaussian

*my_gaussian.ml*

```
let my_gaussian (mu, sigma) =
  let x = sample (gaussian (mu, sigma)) in
  x
```

$$
\begin{aligned}
\{\!\!\{\texttt{my\_gaussian (mu, sigma)}\}\!\!\}_{\emptyset}(U) &= \int_{\mathbb{R}} \{\!\!\{\texttt{sample (gaussian (mu, sigma))}\}\!\!\}_{[\texttt{mu}\leftarrow\mu,\texttt{sigma}\leftarrow\sigma]}(dx)\ \{\!\!\{\texttt{x}\}\!\!\}_{[\texttt{mu}\leftarrow\mu,\texttt{sigma}\leftarrow\sigma,\texttt{x}\leftarrow x]}(U) \\
&= \int_{\mathbb{R}} \mathit{Gaussian}(\mu,\sigma)(dx)\ \delta_x(U) \\
&= \int_U \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}\, dx \\
&= \mathit{Gaussian}(\mu,\sigma)(U)
\end{aligned}
$$

# Example : Beta

*my_gaussian.ml*

```
let my_beta (a, b) =
  let x = sample (uniform (0., 1.)) in
  let () = observe (beta (a, b), x) in
  x
```

$$
\begin{aligned}
\{\!\![\texttt{my\_beta (a, b)}]\!\!\}_{\emptyset}(U) &= \int_0^1 \{\!\![\texttt{sample (uniform (0, 1))}]\!\!\}_{[\mathsf{a}\leftarrow a, \mathsf{b}\leftarrow b]}(dx) \\
&\qquad \int_{()} \{\!\![\texttt{observe (beta (a, b), x)}]\!\!\}_{[\mathsf{a}\leftarrow a, \mathsf{b}\leftarrow b, \mathsf{x}\leftarrow x]}(du)\ \{\!\![\texttt{x}]\!\!\}_{[\mathsf{a}\leftarrow a, \mathsf{b}\leftarrow b, \mathsf{x}\leftarrow x]}(U) \\
&= \int_0^1 \mathit{Uniform}(dx)\ \mathrm{pdf}(\mathit{Beta}(a,b))(x)\ \delta_x(U) \\
&= \int_U \mathrm{pdf}(\mathit{Beta}(a,b))(x) dx \\
&= \mathit{Beta}(a,b)(U)
\end{aligned}
$$

# Example : Coin

coin.ml

```
let coin (x1, ... , xn) =
  let z = sample (uniform (0., 1.)) in
  observe (bernoulli (z), x1); ... ; observe (bernoulli (z), xn);
  z
```

$$
\begin{aligned}
\{\!\![\texttt{coin (x1, ... , xn)}]\!\!\}_{\emptyset}(U) = &\int_0^1 \{\!\![\texttt{sample (uniform (0, 1))}]\!\!\}_{[\texttt{x1}\leftarrow x_1,\ldots,\texttt{xn}\leftarrow x_n]}(dz) \\
&\quad \int_{()} \{\!\![\texttt{observe (bernoulli (z), x1)}]\!\!\}_{[\texttt{z}\leftarrow z,\texttt{x1}\leftarrow x_1,\ldots,\texttt{xn}\leftarrow x_n]}(du_0) \\
&\quad\quad \int_{()} \{\!\![\texttt{observe (bernoulli (z), x2)}]\!\!\}_{[\texttt{z}\leftarrow z,\texttt{x1}\leftarrow x_1,\ldots,\texttt{xn}\leftarrow x_n]}(du_1) \\
&\quad\quad\quad \ldots \\
&\quad\quad\quad\quad \int_{()} \{\!\![\texttt{observe (bernoulli (z), xn)]}]\!\!\}_{[\texttt{z}\leftarrow z,\texttt{x1}\leftarrow x_1,\ldots,\texttt{xn}\leftarrow x_n]}(du_n) \\
&\quad\quad\quad\quad\quad \{\!\![\texttt{z}]\!\!\}_{[\texttt{z}\leftarrow z,\texttt{x1}\leftarrow x_1,\ldots,\texttt{xn}\leftarrow x_n]}(U) \\
= &\int_0^1 \mathit{Uniform}(0,1)(dz) \prod_{i=1}^{n} \mathrm{pdf}(\mathit{Bernoulli}(z))(x_i)\ \delta_z(U) \\
= &\int_U z^{\#\mathrm{heads}}\ (1-z)^{\#\mathrm{tails}}\ dz
\end{aligned}
$$

Unnormalized!

# Example : Coin

coin.ml

```
let coin (x1, ..., xn) =
  let z = sample (uniform (0., 1.)) in
  observe (bernoulli (z), x1); ... ; observe (bernoulli (z), xn);
  z

let d = infer (coin (data))
```

$$\{\!\!|\texttt{coin (x1, ..., xn)}|\!\!\}_{\emptyset}(U) = \int_U z^{\#\text{heads}} (1-z)^{\#\text{tails}}\, dz$$

$$[\![\texttt{infer (coin (x1, ..., xn))}]\!]_{[\texttt{coin}]} = \frac{\int_U z^{\#\text{heads}} (1-z)^{\#\text{tails}}\, dz}{\int_0^1 z^{\#\text{heads}} (1-z)^{\#\text{tails}}\, dz} = \frac{\int_U z^{\#\text{heads}} (1-z)^{\#\text{tails}}\, dz}{\mathrm{B}(\#\text{heads}+1, \#\text{tails}+1)} = \mathit{Beta}(\#\text{heads}+1, \#\text{tails}+1)(U)$$

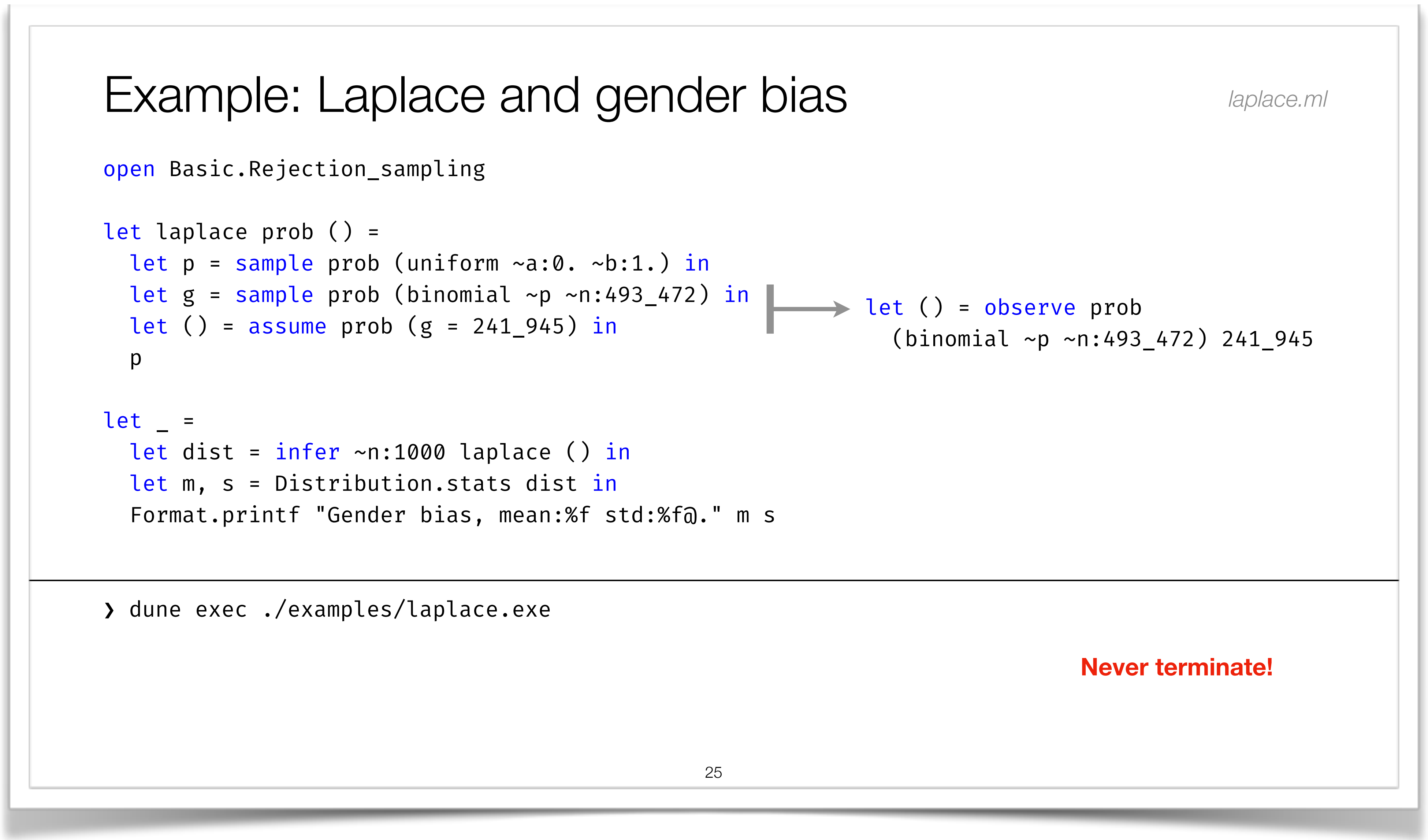
# Exercises

Prove the following properties

```
sample mu (* where mu is defined on [a, b] *)
≡
let x = sample (uniform (a, b)) in
let () = observe (mu, x) in
x
```

```
observe (mu, x) (* where mu is a discrete distribution *)
≡
let y = sample mu in
assume x = y
```

```
sample (bernoulli (0.5))
≡
let x = sample (gaussian (0., 1.)) in
x > 0
```



## Example: 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
```

**Never terminate!**

25

# Improper priors

Uniform priors on bounded domains

- If $\mu : t\ \mathtt{dist}^*$ is defined on $[a, b]$ and has a density
- $\llbracket \mathtt{sample}(\mu) \rrbracket = \llbracket \mathtt{let}\ x = \mathtt{sample}(\mathit{Uniform}(a,b))\ \mathtt{in}\ \mathtt{observe}(\mu, x);\ x \rrbracket$

Improper priors

```
let improper =
  let x = sample (gaussian 0 1) in
  factor (1. /. (pdf (gaussian 0 1) x));
  x
```

$$\llbracket \mathtt{improper} \rrbracket_{\emptyset}(U) = \int_U \mathit{Gaussian}(0,1)(dx)\, \frac{1}{f(x)}\, dx$$
$$= \int_U f(x)\, \frac{1}{f(x)}\, dx$$
$$= \lambda(U)$$



Staton 2017

# References


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

Semantics of probabilistic programs.
Dexter Kozen
Journal of Computer and System 1981

Commutative semantics for probabilistic programming
Sam Staton
ESOP 2017

Semantics of Probabilistic Programs using s-Finite Kernels in Coq
Reynald Affeldt, Cyril Cohen, Ayumu Saito
CPP 2023

# TP : A short introduction to Stan

Everything is on Github: https://github.com/mpri-probprog/probprog-23-24

- Go to `td/td2`
- Launch `jupyter notebook` (or `jupyter lab`)

Requirements

- Pandas
- CmdStanPy
- Jupyter
- Matplotlib

https://mc-stan.org/