Probabilistic Programming Languages

Guillaume Baudart

MPRI 2024-2025

Reminders

Probabilistic Programming Languages

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 $x_1, \dots, x_n$

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

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

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 $x_1, \ldots, x_n$

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

posterior

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

prior

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 $x_1, \ldots, x_n$

Thomas Bayes (1701-1761)

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

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

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

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

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

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

Example: Coin

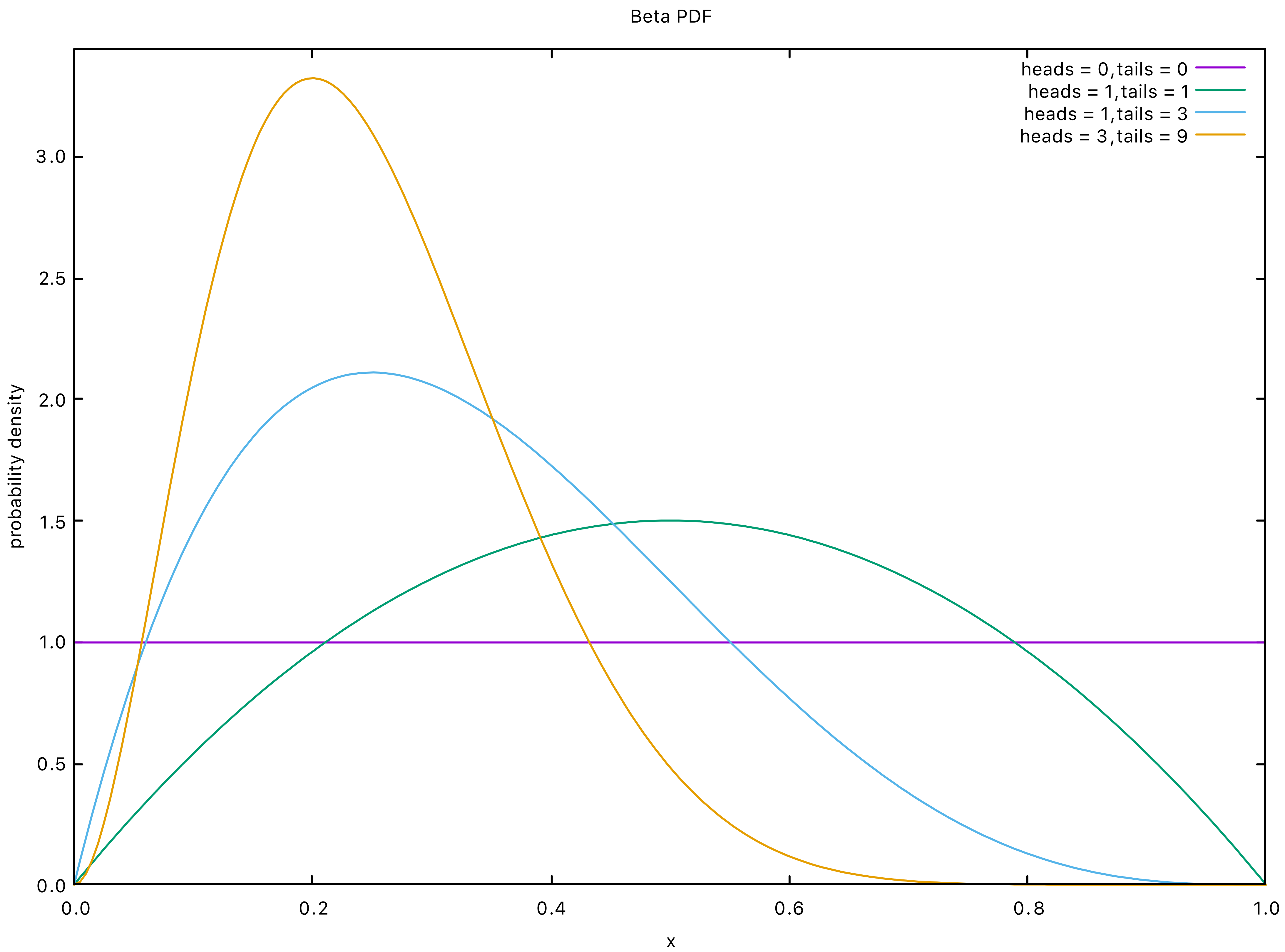
Consider a series of coin tosses

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

Probabilistic model

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

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



Example: Coin

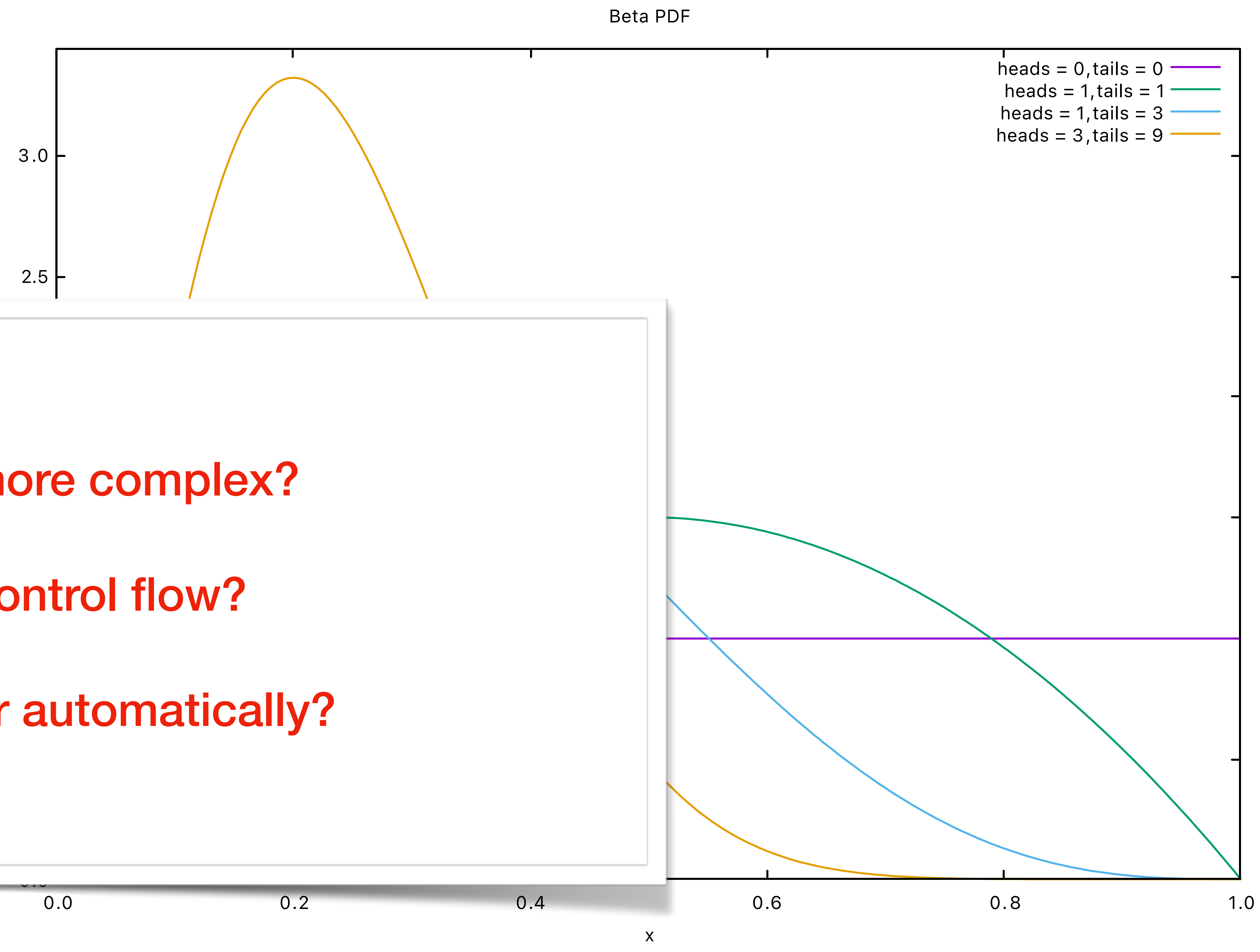
Consider a series of coin tosses

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

Probabilistic model

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

$z \sim \mathit{Beta}(\text{\#heads} +$



What if the model is much more complex?

What if we use arbitrary control flow?

Can we compute the posterior automatically?

Bayesian reasoning

$$p(x \mid y_1, \ldots, y_n) = \frac{p(x)\; p(y_1, \ldots, y_n \mid x)}{p(y_1, \ldots, y_n)} \qquad \text{(Bayes' theorem)}$$

$$\propto p(x)\; p(y_1, \ldots, y_n \mid x) \qquad \text{(Data are constants)}$$

Thomas Bayes (1701-1761)

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

Bayesian reasoning

Bayesian Inference: learn parameters from data

- Latent parameter x
- Observed data $y_1, \ldots, y_n$

$$p(x \mid y_1, \ldots, y_n) = \frac{p(x)\, p(y_1, \ldots, y_n \mid x)}{p(y_1, \ldots, y_n)} \qquad \text{(Bayes' theorem)}$$

$$\propto p(x)\, p(y_1, \ldots, y_n \mid x) \qquad \text{(Data are constants)}$$

Thomas Bayes (1701-1761)

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

Bayesian reasoning

Bayesian Inference: learn parameters from data

- Latent parameter x
- Observed data $y_1, \dots, y_n$

$$\underbrace{p(x \mid y_1, \dots, y_n)}_{\text{posterior}} = \frac{p(x)\, p(y_1, \dots, y_n \mid x)}{p(y_1, \dots, y_n)} \qquad \text{(Bayes' theorem)}$$

$$\propto \underbrace{p(x)}_{\text{prior}}\ \underbrace{p(y_1, \dots, y_n \mid x)}_{\text{likelihood}} \qquad \text{(Data are constants)}$$

Probabilistic constructs

- `x = sample(d)`: introduce a random variable `x` of distribution `d`
- `observe(d, y)`: condition on the fact that `y` was sampled from `d`
- `infer(m, y)`: compute posterior distribution of `m` given `y`

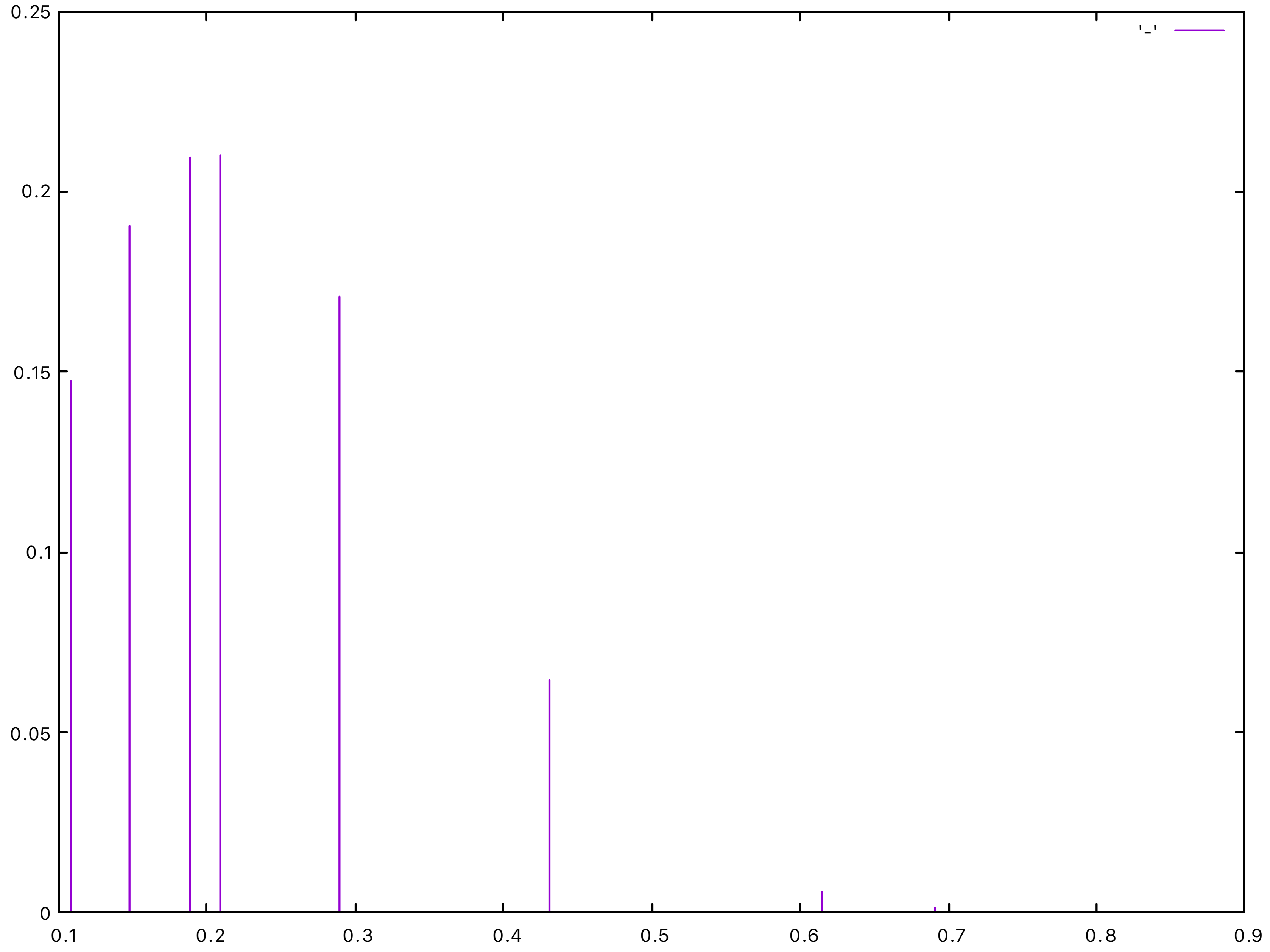
Thomas Bayes (1701-1761)

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

Example: coin

```
def coin(obs: list[int]) → float:
  p = sample(Uniform(0, 1))
  for o in obs:
    observe(Bernoulli(p), o)
  return p

with ImportanceSampling(num_particles=10):
  dist = infer(coin, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1])
  viz(dist)
```

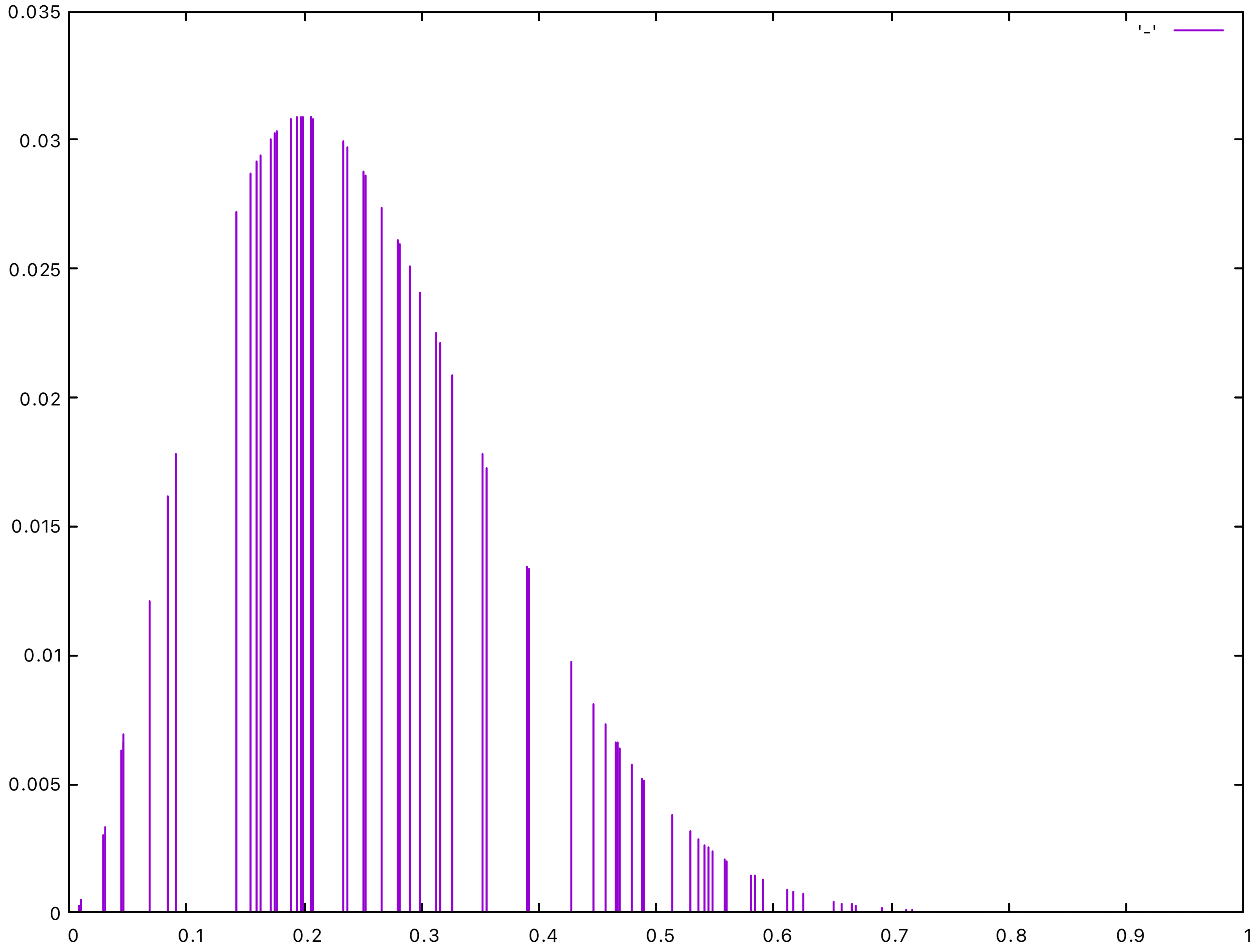
10 particles



Example: coin

```python
def coin(obs: list[int]) -> float:
  p = sample(Uniform(0, 1))
  for o in obs:
    observe(Bernoulli(p), o)
  return p

with ImportanceSampling(num_particles=100):
  dist = infer(coin, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1])
  viz(dist)
```

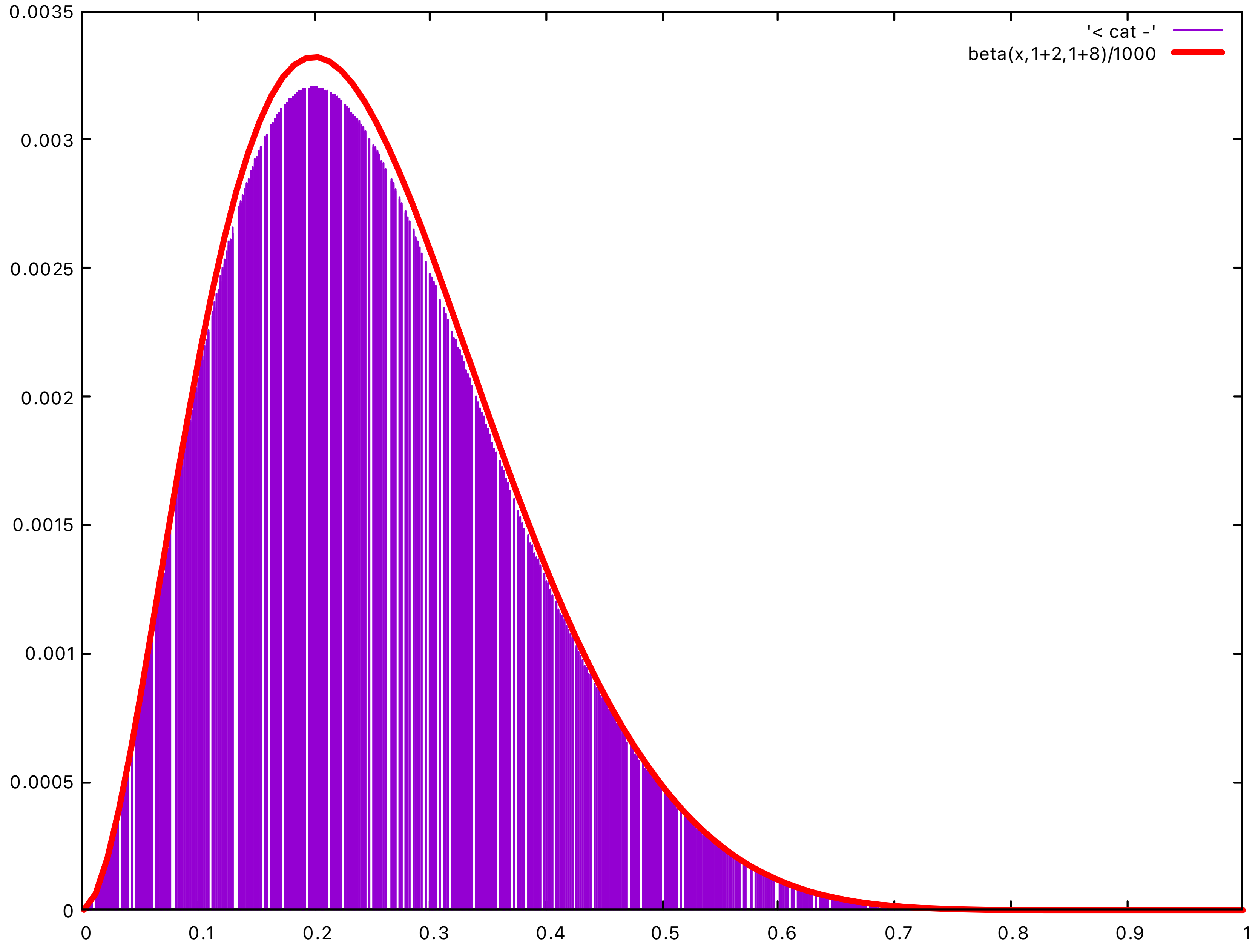
100 particles



Example: coin

```python
def coin(obs: list[int]) → float:
  p = sample(Uniform(0, 1))
  for o in obs:
    observe(Bernoulli(p), o)
  return p

with ImportanceSampling(num_particles=1000):
  dist = infer(coin, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1])
  viz(dist)
```

1000 particles

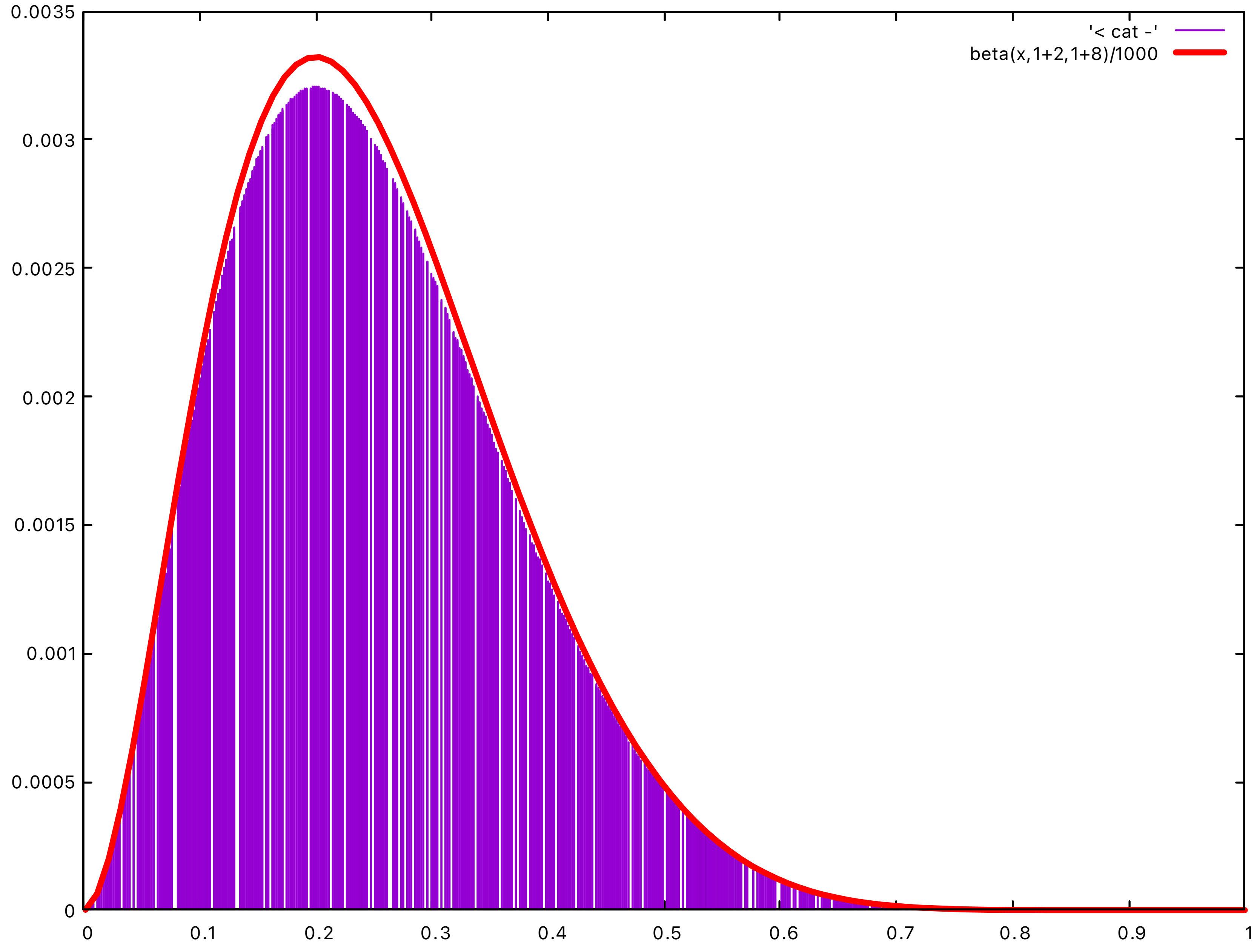


Example: coin

```python
def coin(obs: list[int]) → float:
  p = sample(Uniform(0, 1))
  for o in obs:
    observe(Bernoulli(p), o)
  return p

with ImportanceSampling(num_particles=1000):
  dist = infer(coin, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1])
  viz(dist)
```

Exact solution: Beta(#heads + 1, #tails + 1)

1000 particles



Outline

I - Language

- Syntax: language and types
- Types and kinds: deterministic vs. probabilistic

II - Runtime: basic inference

- Rejection sampling (hard)
- Importance sampling

III - Kernel Semantics

- Types as measurable spaces
- Expressions as measures

Language

Probabilistic Programming Languages

Language and types

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)

Types and kinds

expression

Context

type

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

kind: D (deterministic) | P (probabilistic)

Kind P guards what can be expressed
in a probabilistic model

Typing declarations

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

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

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

Typing declarations

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\mathit{typeOf}(\mathit{op}) = t_1 \to^D t_2 \qquad G \vdash^D e : t_1}{G \vdash^D \mathit{op}(e) : t_2}$$

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

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

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

Typing expressions

$$\frac{\textit{typeOf}(c) = t}{G \vdash^D c : t} \qquad \frac{G(x) = t}{G \vdash^D x : t} \qquad \frac{G \vdash^D e_1 : t_1 \qquad G \vdash^D e_2 : t_2}{G \vdash^D (e_1, e_2) : t_1 \times t_2}$$

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

$$\frac{G \vdash^D e_1 : \texttt{bool} \qquad G \vdash^k e_2 : t \qquad G \vdash^k e_3 : t}{G \vdash^k \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t} \qquad \frac{G \vdash^k e_1 : t_1 \qquad G + [p \leftarrow t_1] \vdash^k e_2 : t_2}{G \vdash^k \texttt{let}\ p\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : t_2}$$

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)
```

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)
```

[coin :???]

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)
```

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

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

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)
```

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

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

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

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)
```

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

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

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

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

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)
```

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

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

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

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

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

Example : Coin

coin.ml

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

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

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

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

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

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

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

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

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)
```

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

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

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

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

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

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

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

Runtime

Probabilistic Programming Languages

Hands-on: BYO-PPL

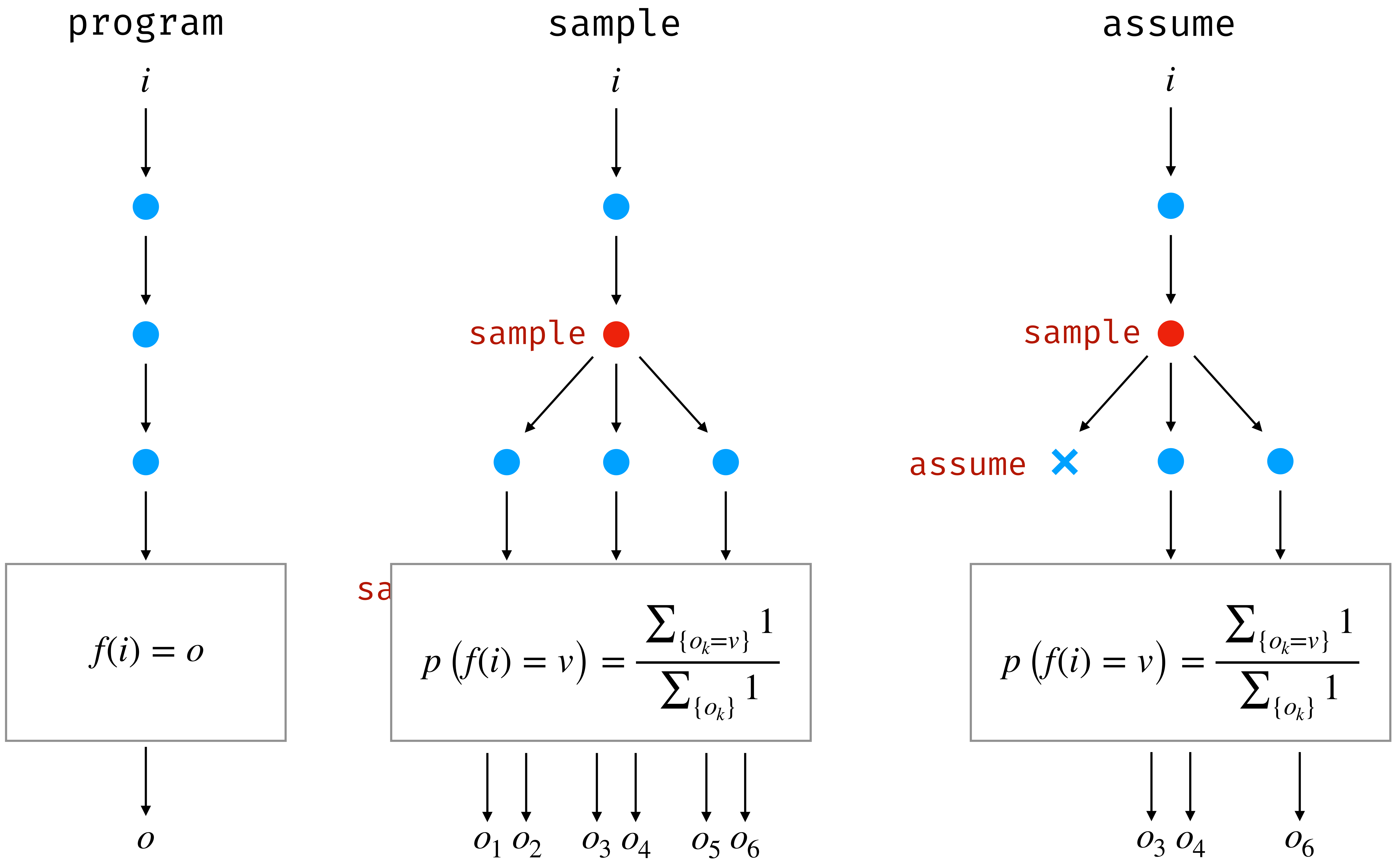
Install

- Clone https://github.com/mpri-probprog/probprog-24-25
- `cd byo-ppl`
- `opam install --deps-only .`

TODO

- Add a new distribution to distribution.ml (e.g, exponential, Poisson)
- Complete the code of `Rejection_sampling_hard` and `Importance_sampling`
- Implement and test the two models `coin` and `regression`

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



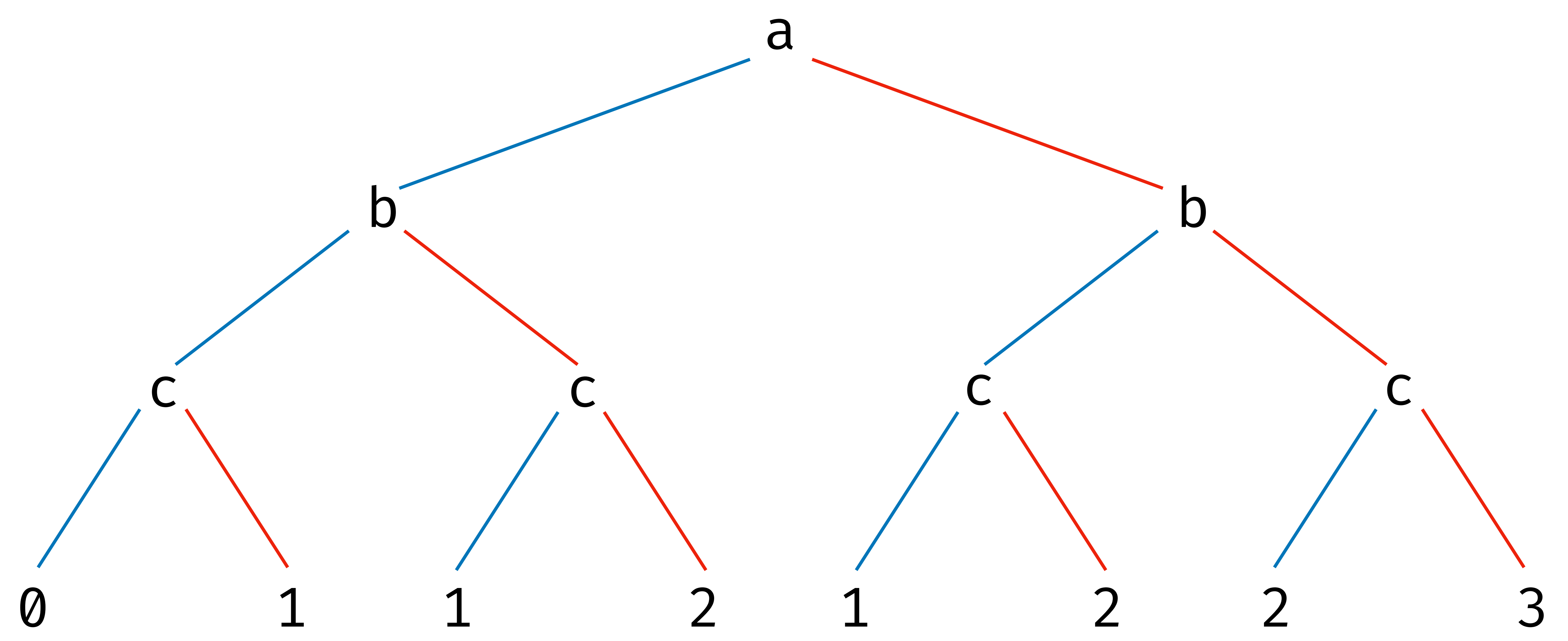
Rejection Sampling

Runtime

Example: Funny Bernoulli

funny_bernoulli.ml

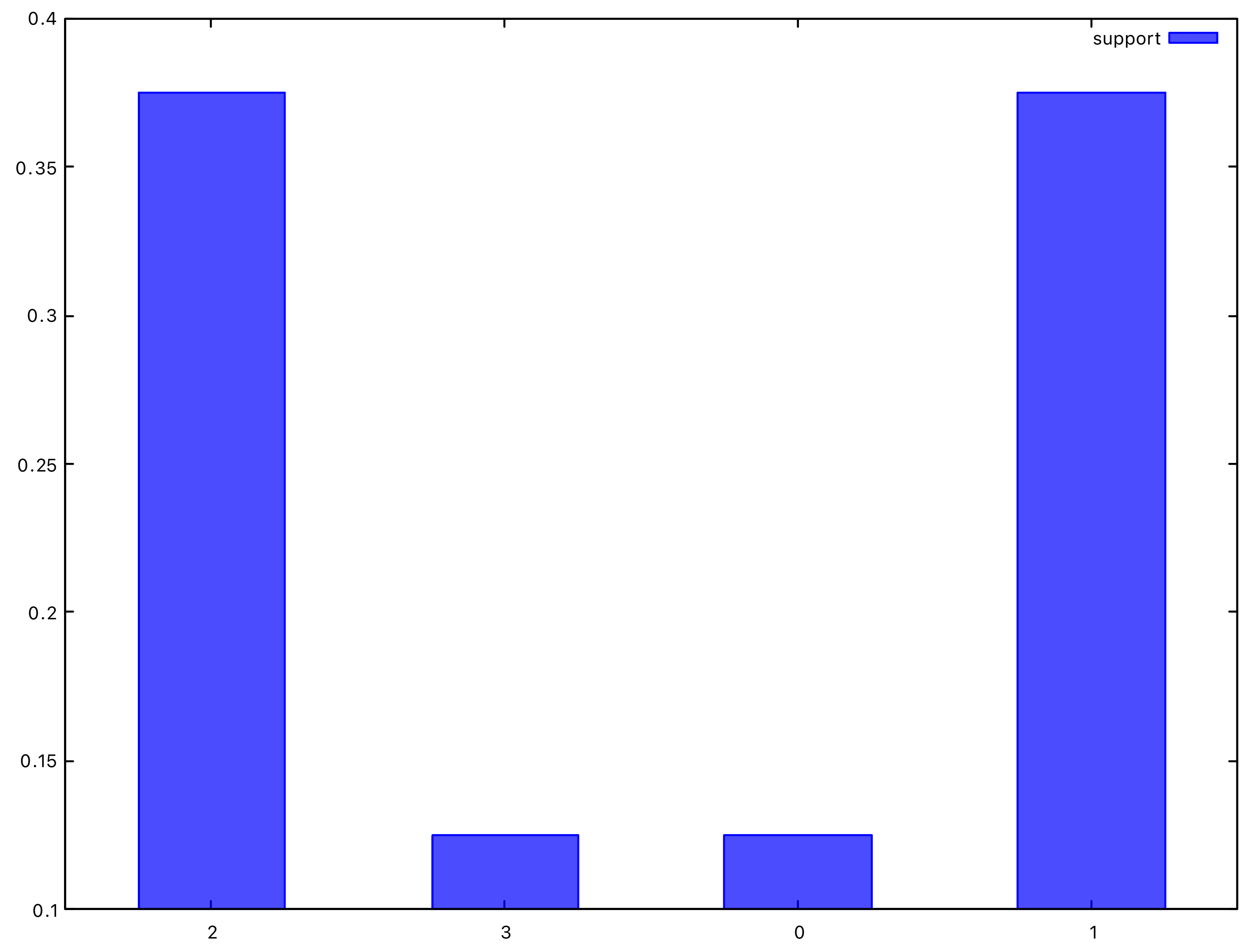
```
let funny_bernoulli () =
  let a = sample (bernoulli ~p:0.5) in
  let b = sample (bernoulli ~p:0.5) in
  let c = sample (bernoulli ~p:0.5) in
  a + b + c
```

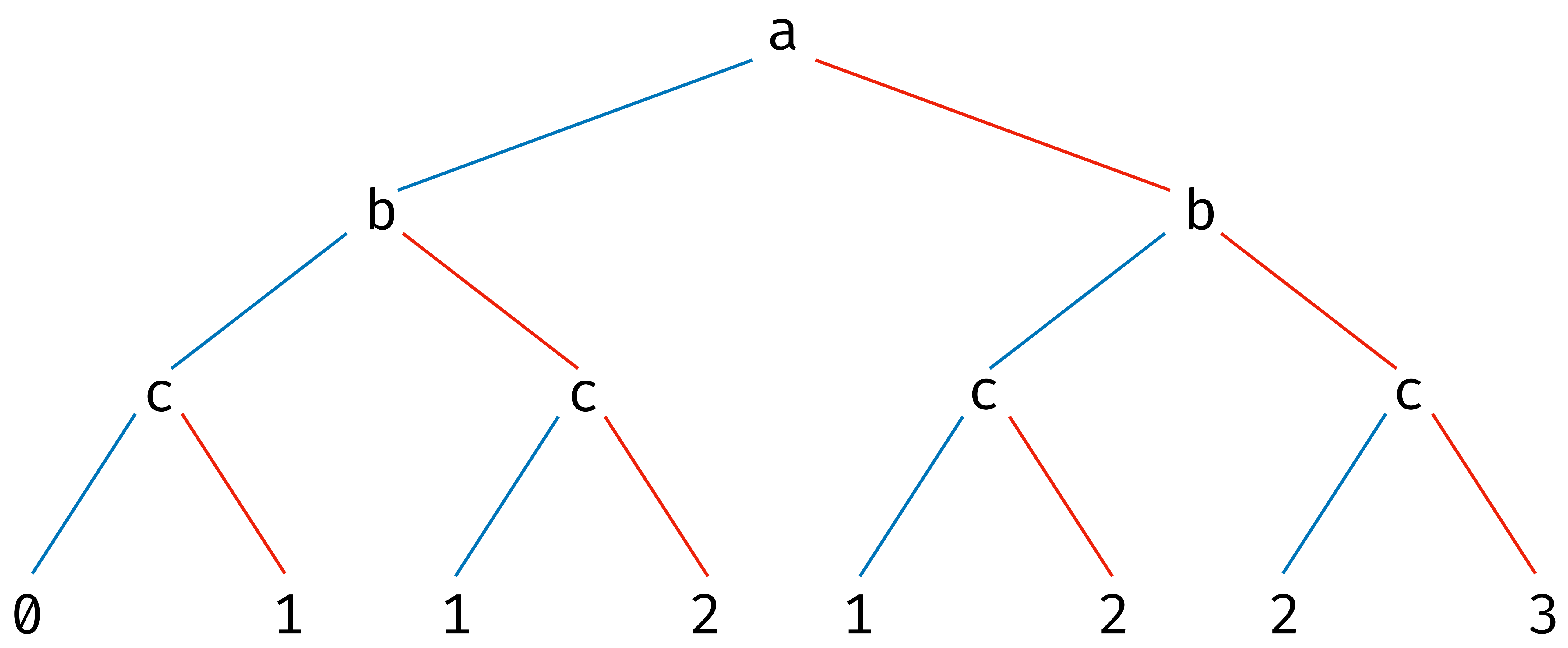


Example: Funny Bernoulli

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
  a + b + c
```

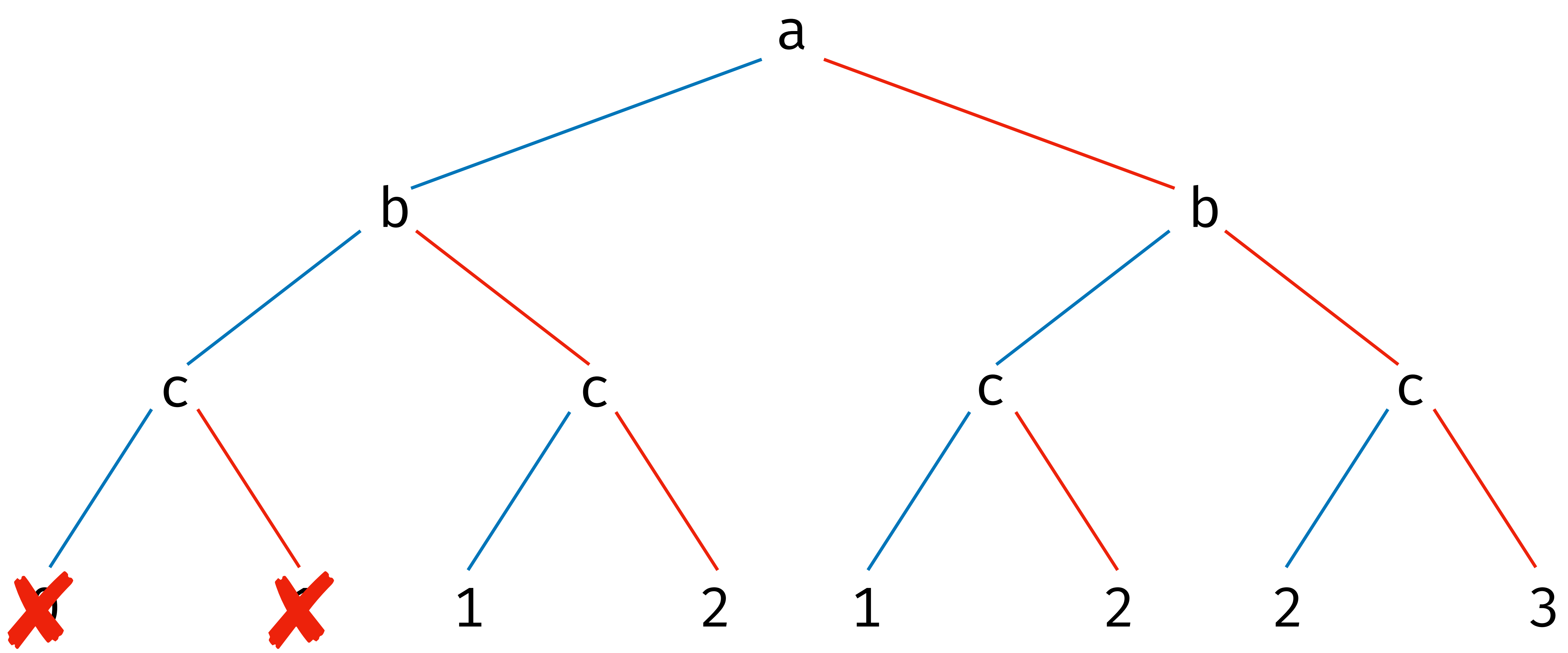




Example: Funny Bernoulli

funny_bernoulli.ml

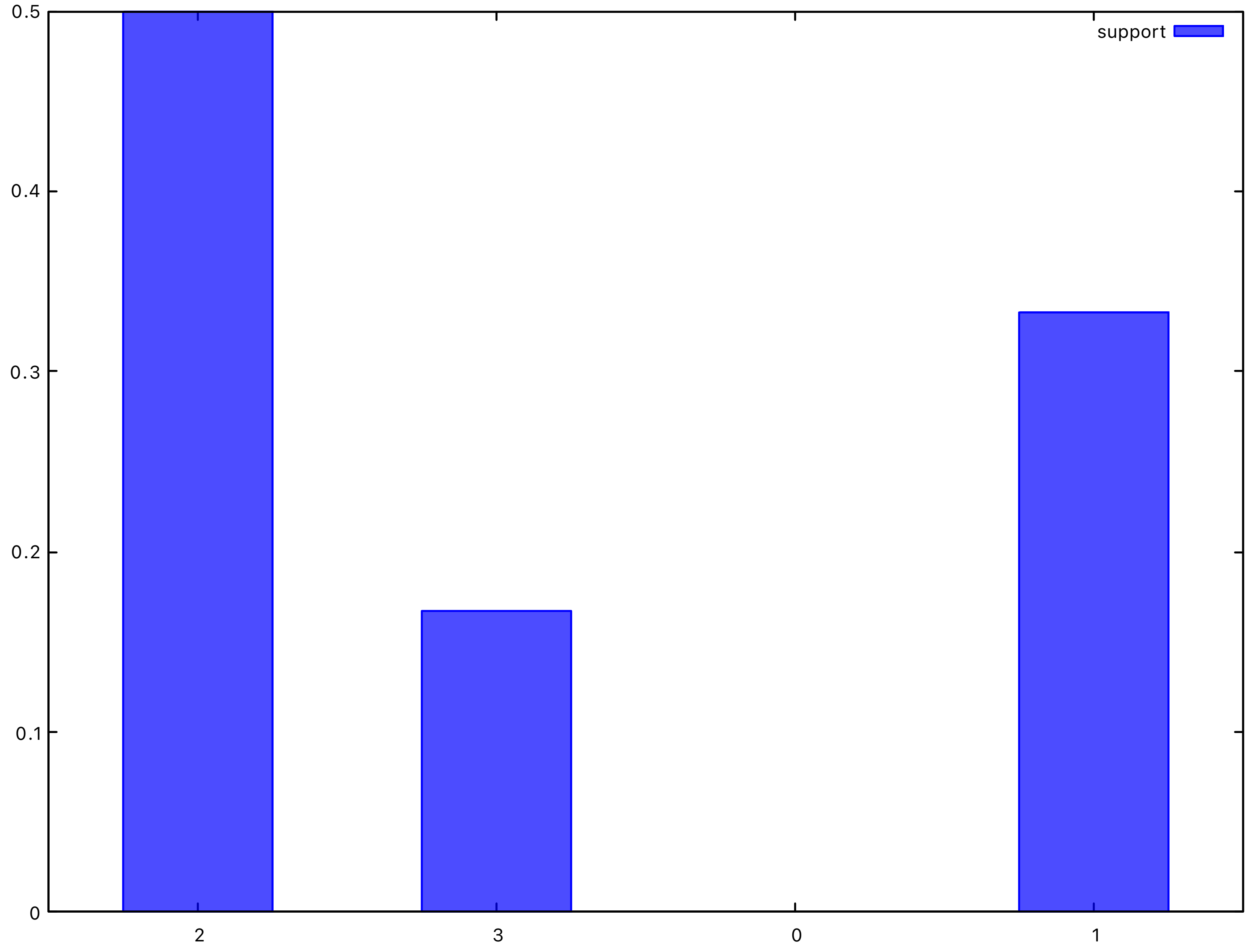
```
let funny_bernoulli () =
  let a = sample (bernoulli ~p:0.5) in
  let b = sample (bernoulli ~p:0.5) in
  let c = sample (bernoulli ~p:0.5) in
  let () = assume (a = 1 || b = 1) in
  a + b + c
```

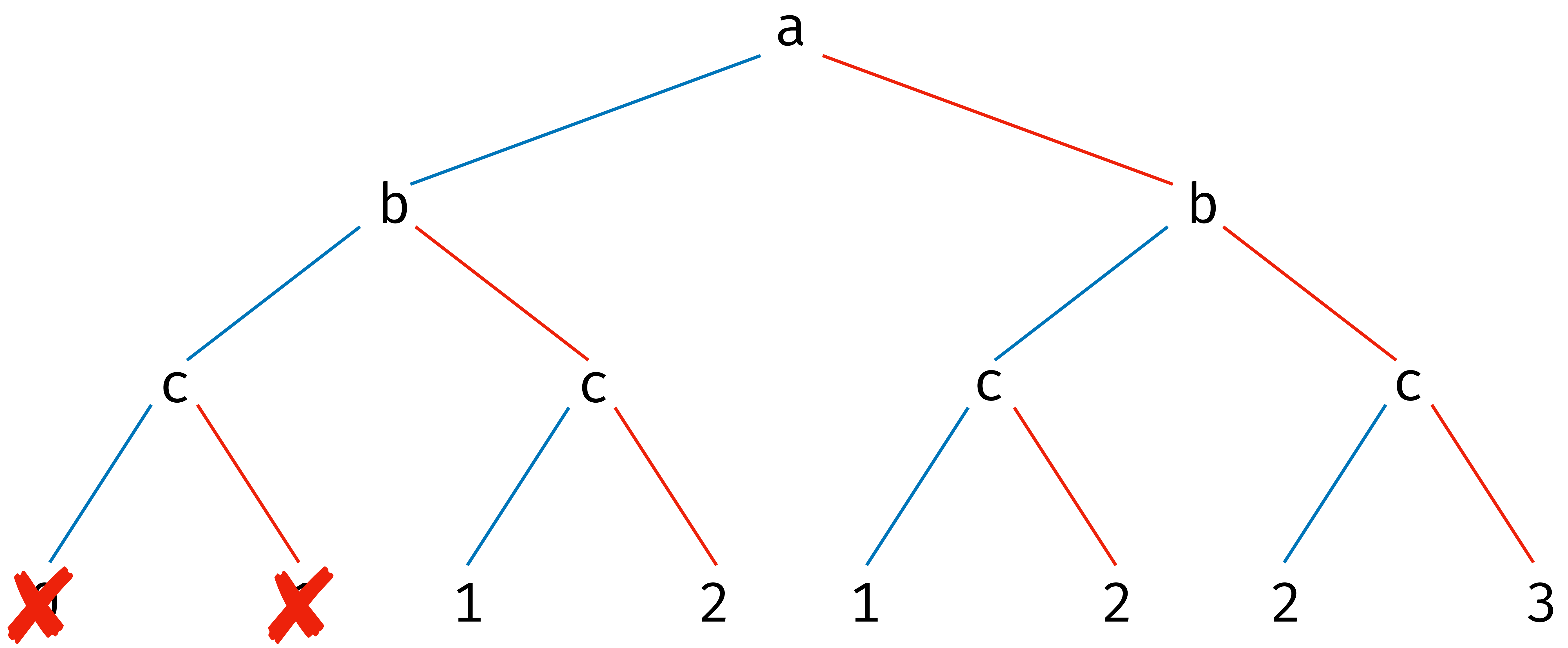


Example: Funny Bernoulli

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 (hard)

basic.ml

```
module Rejection_sampling_hard : 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

Hard conditioning

- `val observe : 'a Distribution.t → 'a → unit`
- Assume that a value was sampled from a distribution (??)

Rejection sampling (hard)

basic.ml

```
module Rejection_sampling_hard = struct

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

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

Rejection sampling (hard)

basic.ml

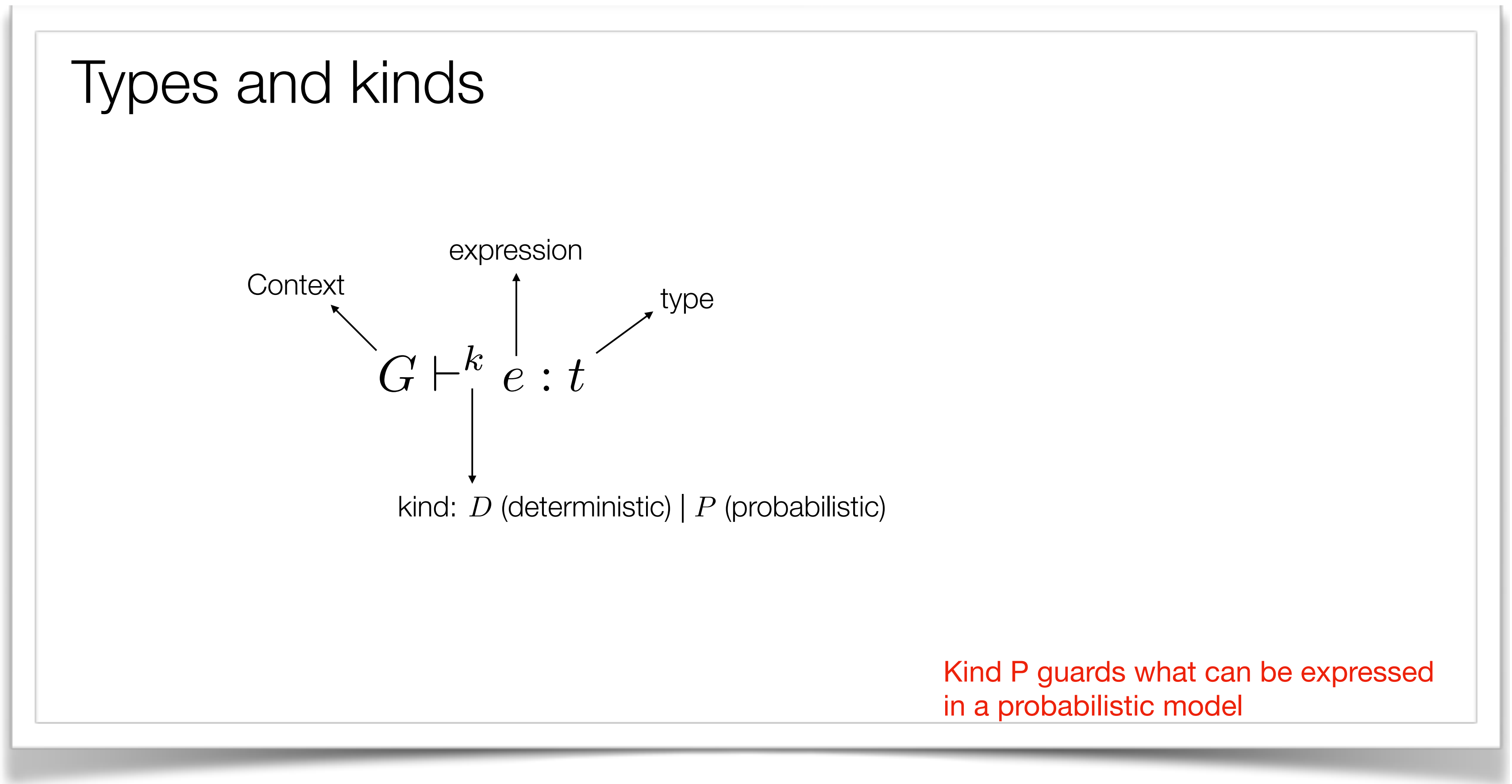
```
module Rejection_sampling_hard = struct
  exception Reject

  let sample d = Distribution.draw d
  let assume p = if not p then raise Reject
  let observe d x = assume (Distribution.draw d = x)


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

The type `prob` trick

```
module Rejection_sampling_hard : sig
  type prob
  val sample : prob -> 'a Distribution.t -> 'a
  val assume : prob -> bool -> unit
  val observe : prob -> 'a Distribution.t -> 'a -> 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 (hard)

basic.ml

```
module Rejection_sampling_hard = struct
  type prob = Prob

  exception Reject

  let sample _prob d = Distribution.draw d
  let assume _prob p = if not p then raise Reject
  let observe _prob d x = assume (Distribution.draw d = x)

  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

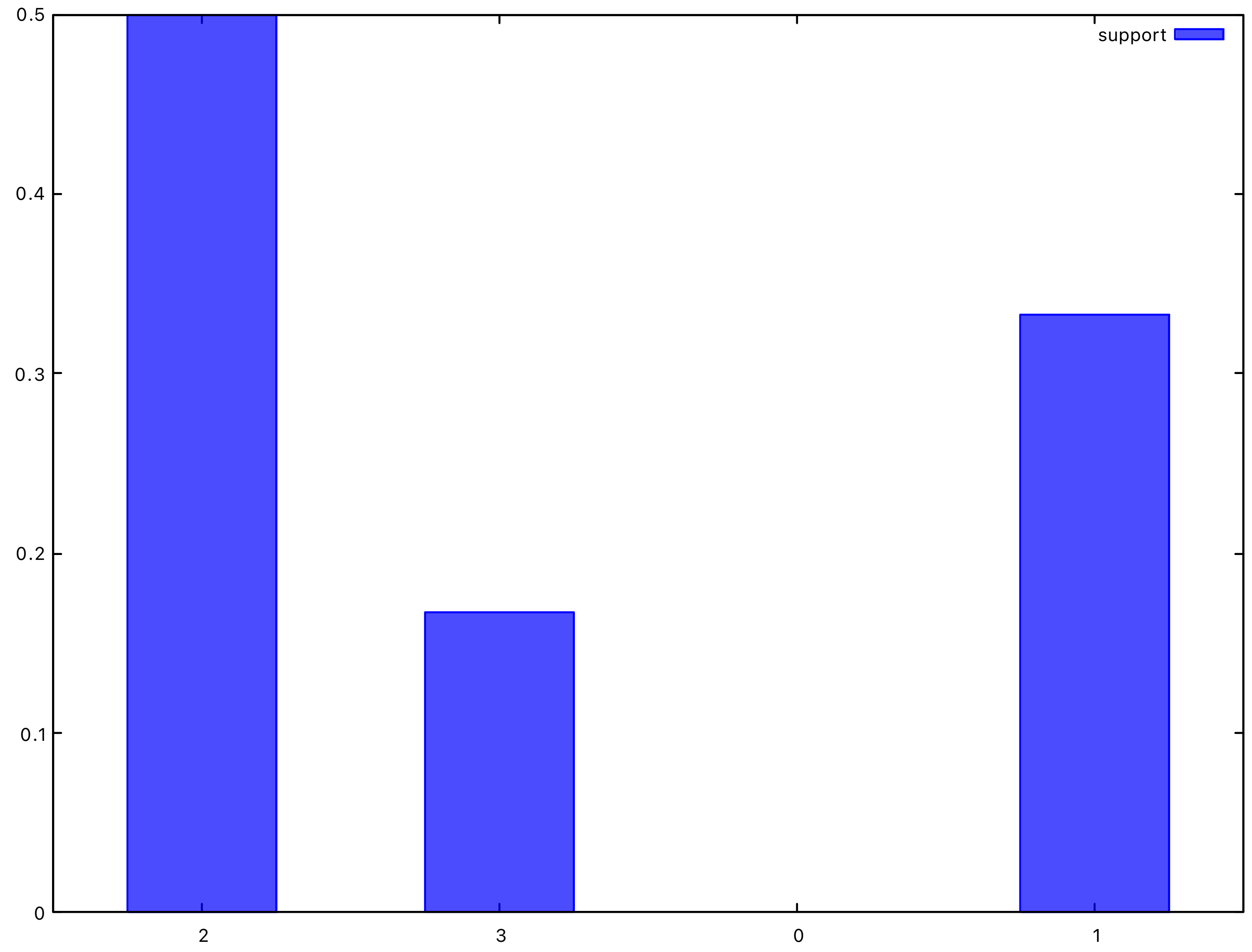
```
open Byoppl
open Distribution
open Basic.Rejection_sampling_hard

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 support = categorical_to_list dist in
  List.iter (fun (v, w) -> Format.printf "%d %f@." v w) support
```

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

Example: Funny Bernoulli

funny_bernoulli.ml

```
open Byoppl
open Distribution
open Basic.Rejection_sampling_hard

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 support = categorical_to_list dist in
  List.iter (fun (v, w) -> Format.printf "%d %f@." v w) support
```




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

Example: Coin

coin.ml

```
open Basic.Rejection_sampling_hard

let coin prob data =
  let z = sample prob (uniform ~a:0. ~b:1.) in
  let tosses = List.map (fun _ → sample prob (bernoulli ~p:z)) data in
  let () = assume prob (data = tosses) in
  z

let data = [false; true; true; false; false; false; false; false; false; false]

let _ =
  let dist = infer coin data 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_hard

let coin prob data =
  let z = sample prob (uniform ~a:0. ~b:1.) in
  let tosses = List.map (fun _ → sample prob (bernoulli ~p:z)) data in
  let () = assume prob (data = tosses) in
  z

let data = [false; true; true; false; false; false; false; false; false; false]

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

↦ `observe d x`

```
❯ dune exec ./examples/coin.exe

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

Example: Coin

coin.ml

```
open Basic.Rejection_sampling_hard

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

let data = [false; true; true; false; false; false; false; false; false; false]

let _ =
  let dist = infer coin data 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
```

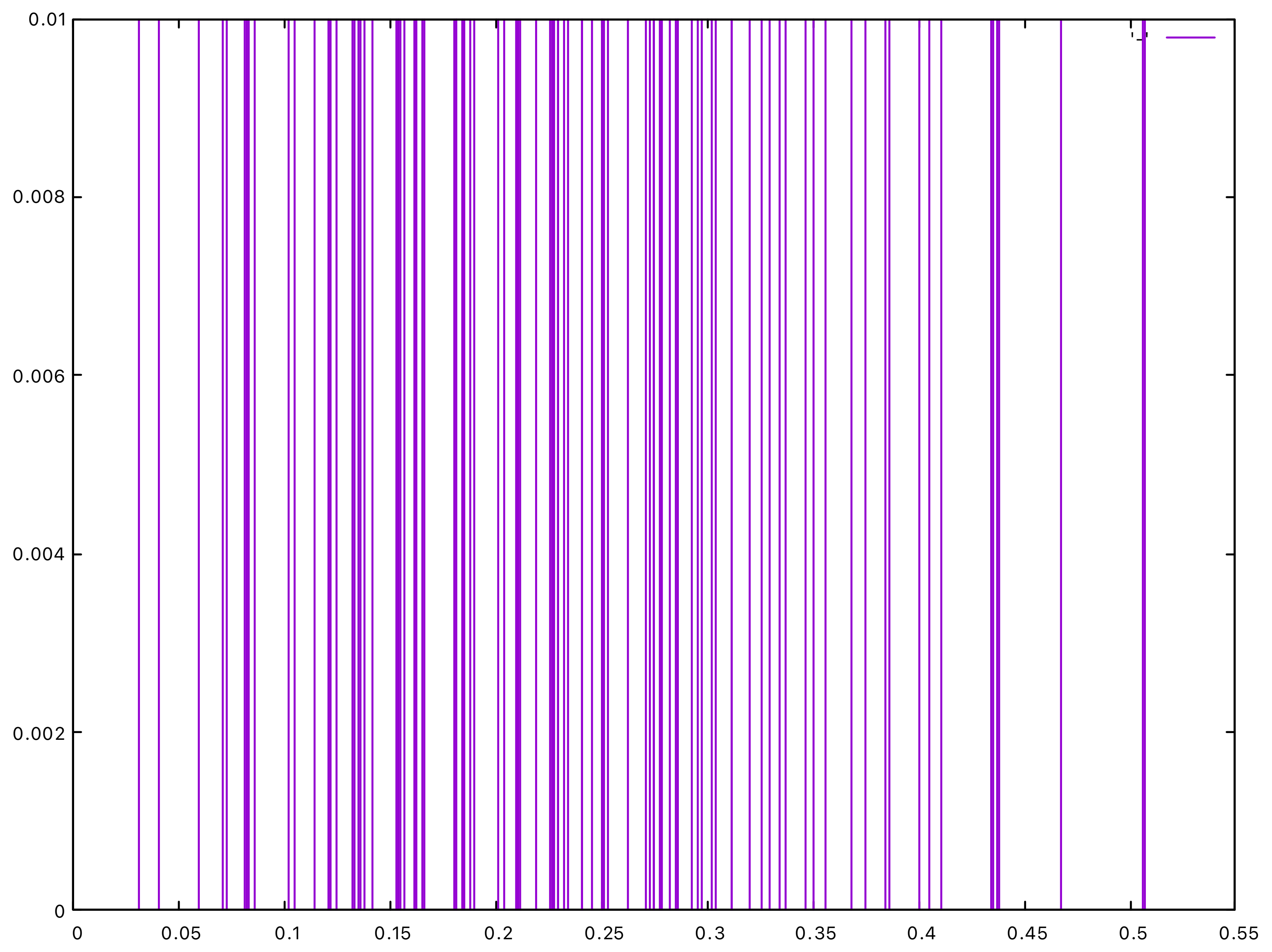
Example: Coin

coin.ml

```
open Basic.Rejection_sampling_hard

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

let data = [false; true; true; false; false; false; false; false; false; false]

let _ =
  let dist = infer coin data 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



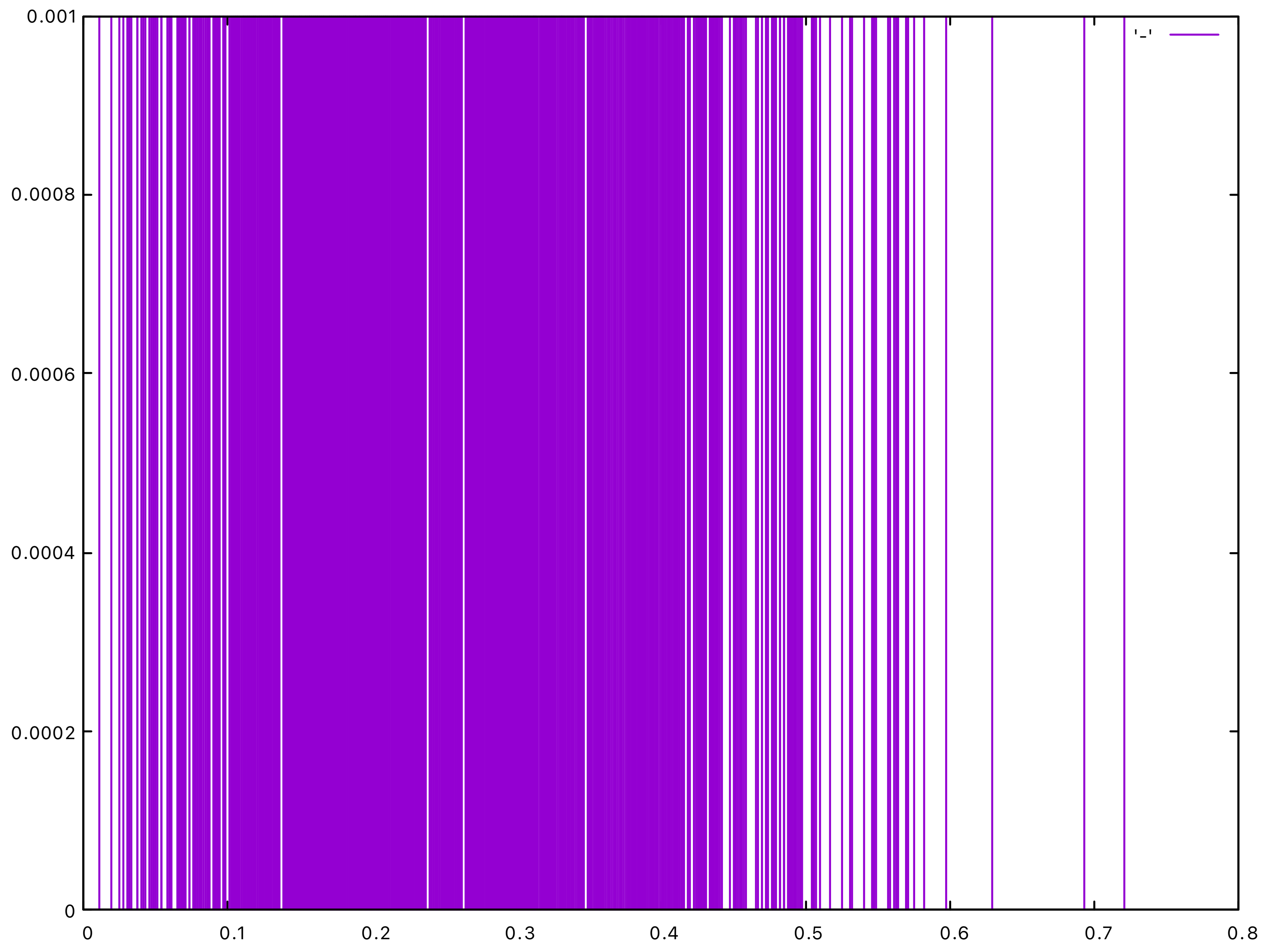
Example: Coin

coin.ml

```
open Basic.Rejection_sampling_hard

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

let data = [false; true; true; false; false; false; false; false; false; false]

let _ =
  let dist = infer coin data 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

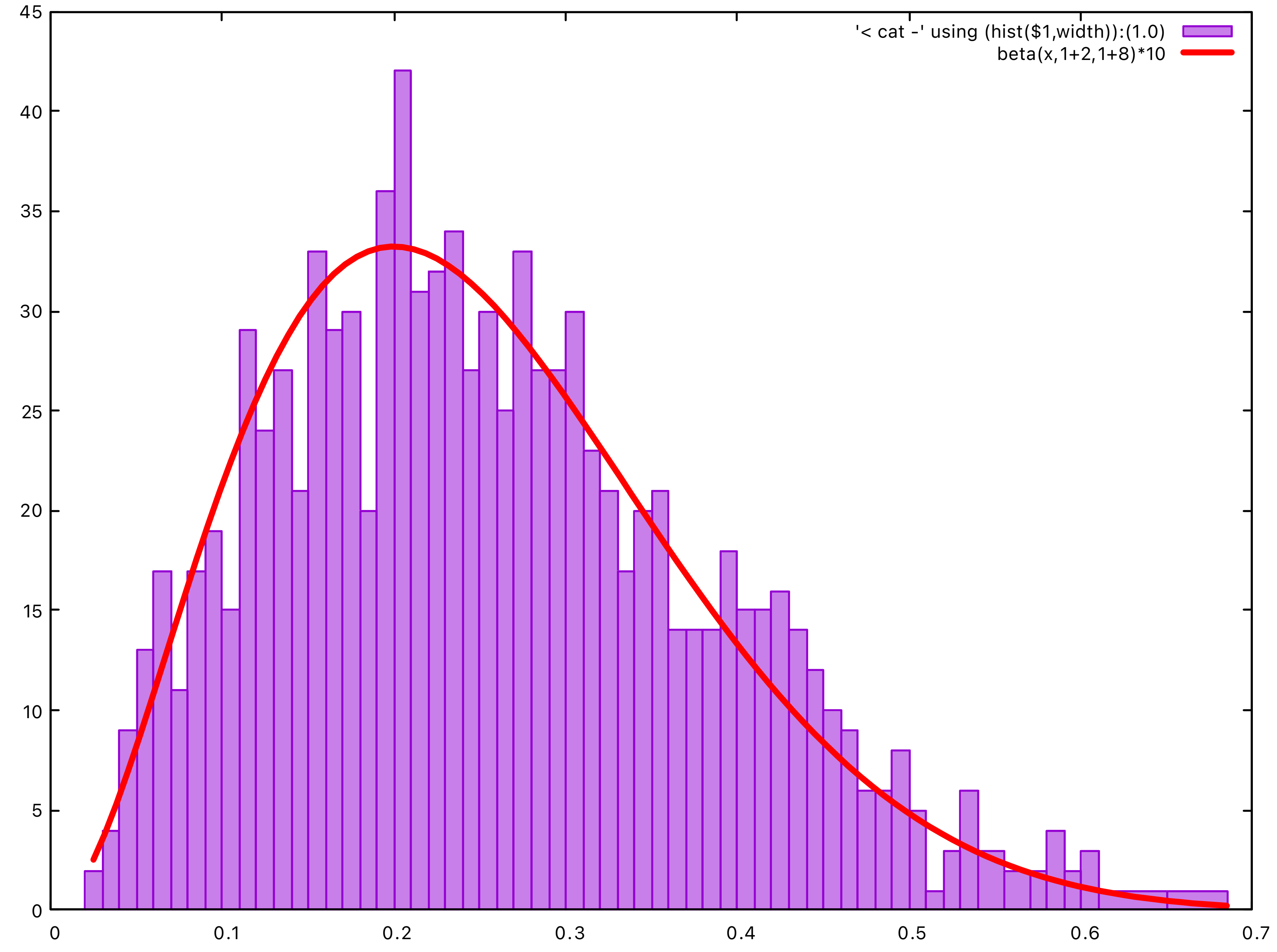


Example: Coin

coin.ml

```
open Basic.Rejection_sampling_hard

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

let data = [false; true; true; false; false; false; false; false; false; false]

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

1000 particles



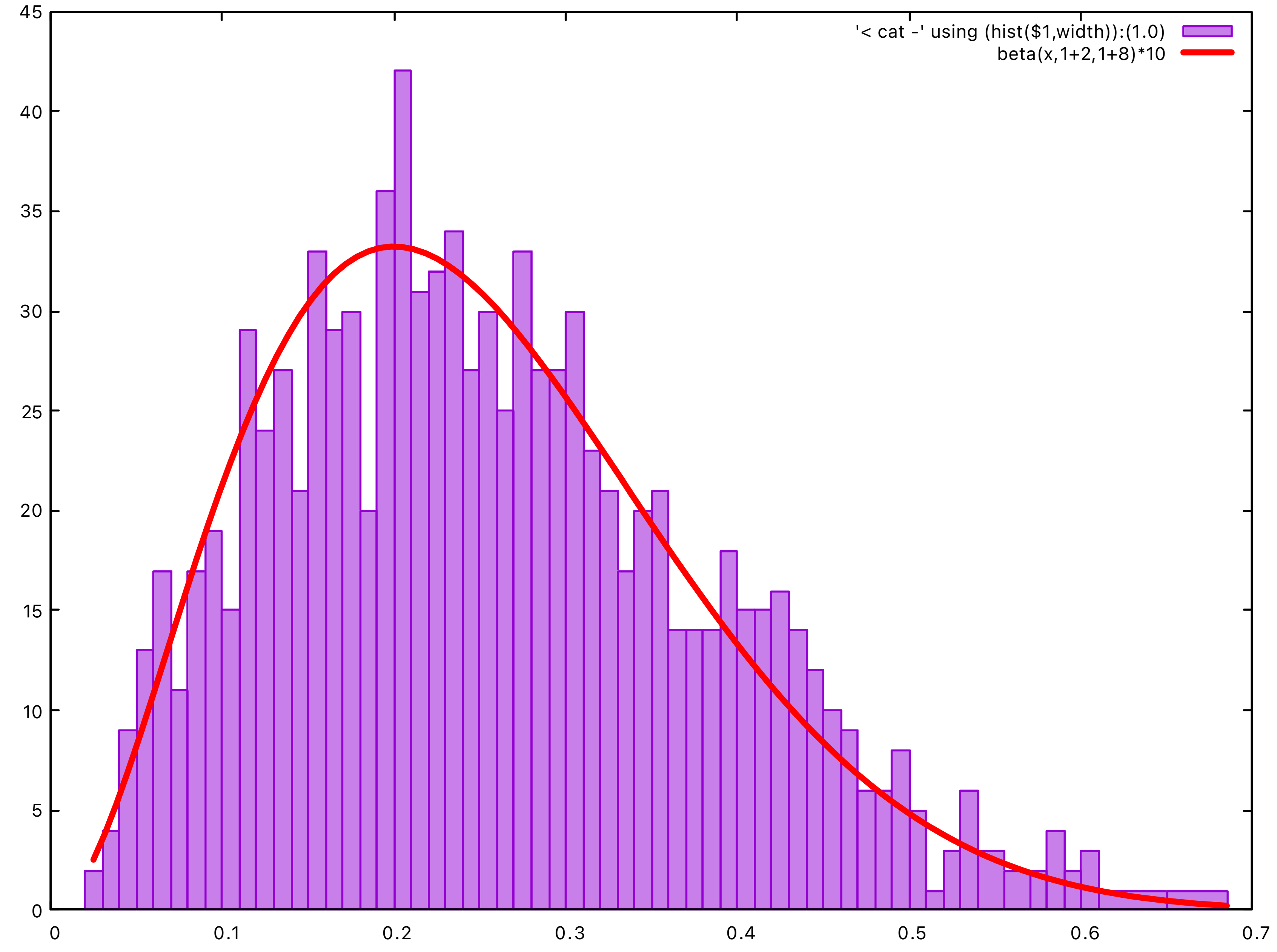

```
❯ dune exec ./examples/coin.exe

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

Example: Coin

coin.ml

```
open Basic.Rejection_sampling_hard

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

let data = [false; true; true; false; false; false; false; false; false; false]

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

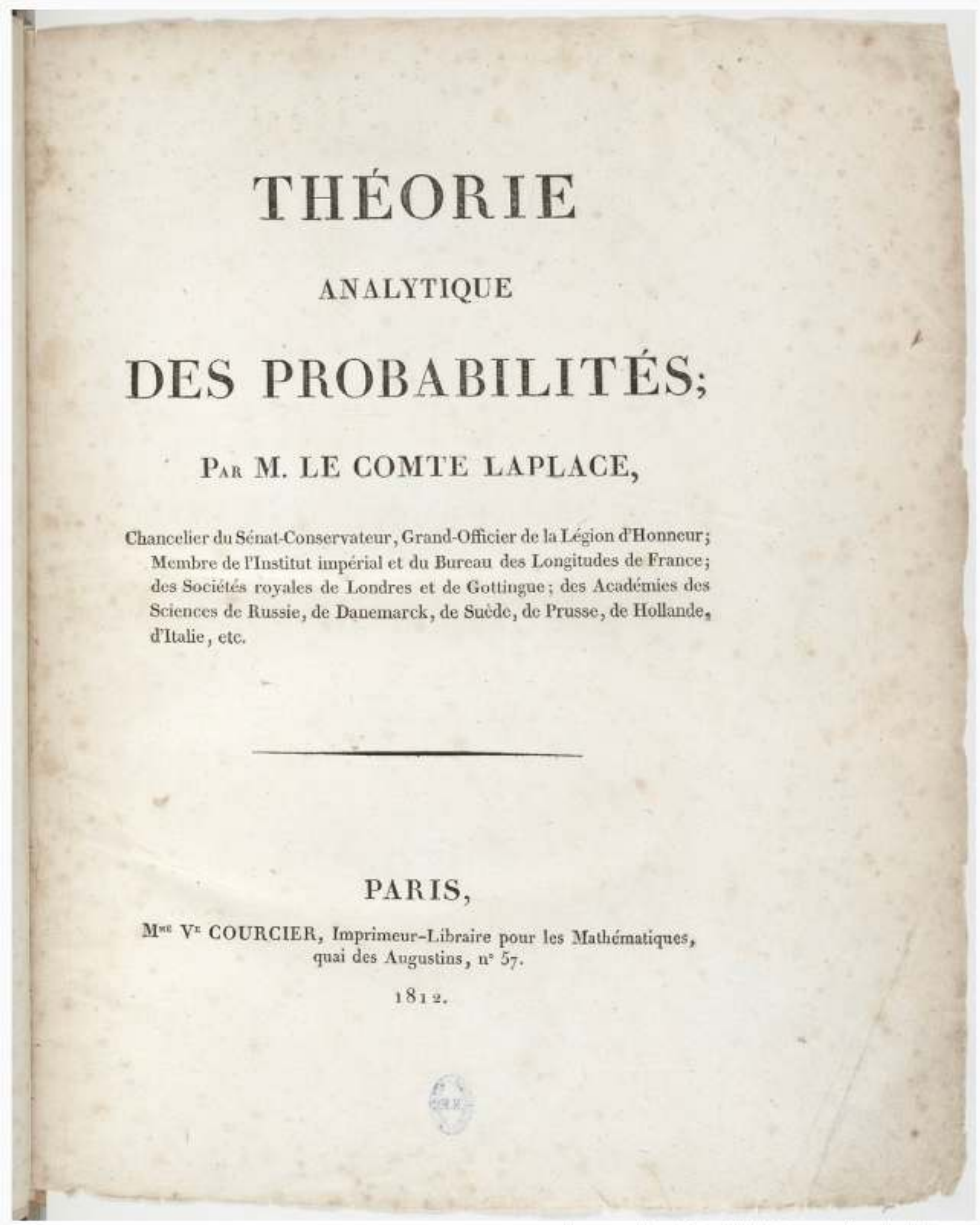
1000 particles




```
❯ dune exec ./examples/coin.exe

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

Slow!

Laplace problem


THÉORIE

ANALYTIQUE

DES PROBABILITÉS;

PAR M. LE COMTE LAPLACE,

Chancelier du Sénat-Conservateur, Grand-Officier de la Légion d'Honneur; Membre de l'Institut impérial et du Bureau des Longitudes de France; des Sociétés royales de Londres et de Gottingue; des Académies des Sciences de Russie, de Danemarck, de Suède, de Prusse, de Hollande, d'Italie, etc.

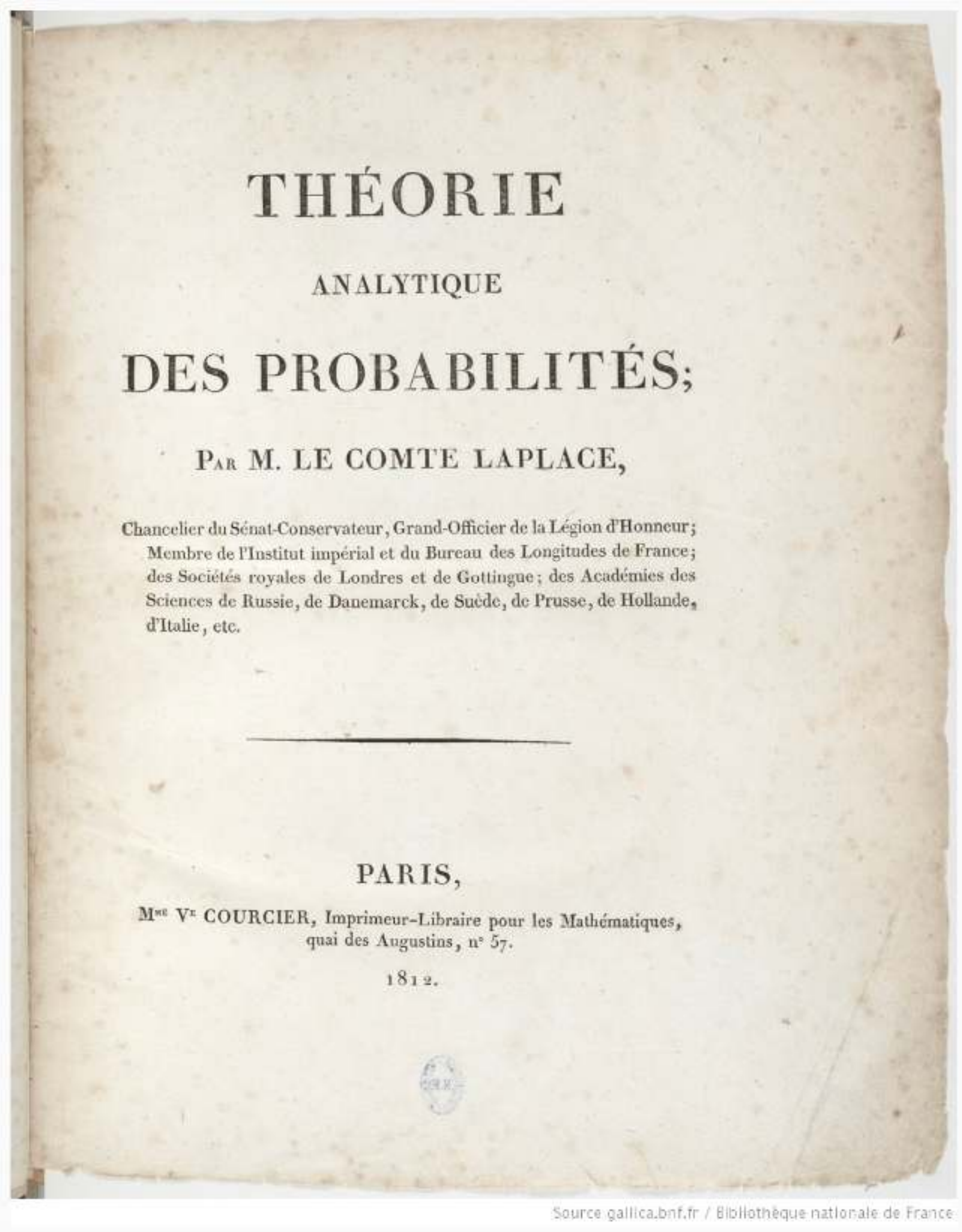
PARIS,

Mme Ve COURCIER, Imprimeur-Libraire pour les Mathématiques, quai des Augustins, n° 57.

1812.

Laplace problem

What is the probability that the proportion of boys over girls is greater in London than in Paris in the XVIII century?



THÉORIE

ANALYTIQUE

DES PROBABILITÉS;

PAR M. LE COMTE LAPLACE,

Chancelier du Sénat-Conservateur, Grand-Officier de la Légion d'Honneur; Membre de l'Institut impérial et du Bureau des Longitudes de France; des Sociétés royales de Londres et de Gottingue; des Académies des Sciences de Russie, de Danemarck, de Suède, de Prusse, de Hollande, d'Italie, etc.

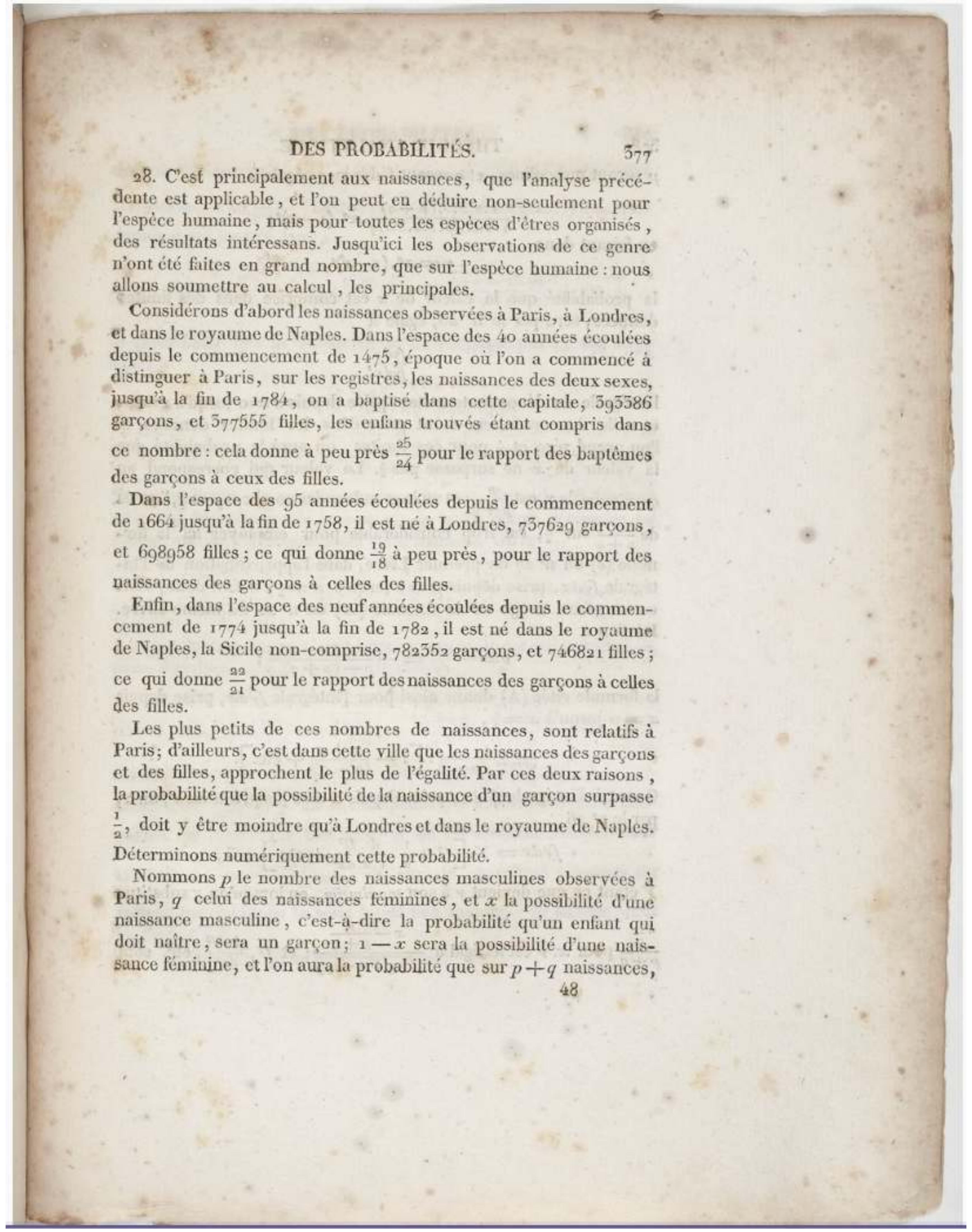
PARIS,

Mme Ve COURCIER, Imprimeur-Libraire pour les Mathématiques, quai des Augustins, n° 57.

1812.

Laplace problem

What is the probability that the proportion of boys over girls is greater in London than in Paris in the XVIII century?



DES PROBABILITÉS. 377

28. C'est principalement aux naissances, que l'analyse précédente est applicable, et l'on peut en déduire non-seulement pour l'espèce humaine, mais pour toutes les espèces d'êtres organisés, des résultats intéressans. Jusqu'ici les observations de ce genre n'ont été faites en grand nombre, que sur l'espèce humaine : nous allons soumettre au calcul, les principales.

Considérons d'abord les naissances observées à Paris, à Londres, et dans le royaume de Naples. Dans l'espace des 40 années écoulées depuis le commencement de 1475, époque où l'on a commencé à distinguer à Paris, sur les registres, les naissances des deux sexes, jusqu'à la fin de 1784, on a baptisé dans cette capitale, 393386 garçons, et 377555 filles, les enfans trouvés étant compris dans ce nombre : cela donne à peu près $\frac{25}{24}$ pour le rapport des baptêmes des garçons à ceux des filles.

Dans l'espace des 95 années écoulées depuis le commencement de 1664 jusqu'à la fin de 1758, il est né à Londres, 737629 garçons, et 698958 filles ; ce qui donne $\frac{19}{18}$ à peu près, pour le rapport des naissances des garçons à celles des filles.

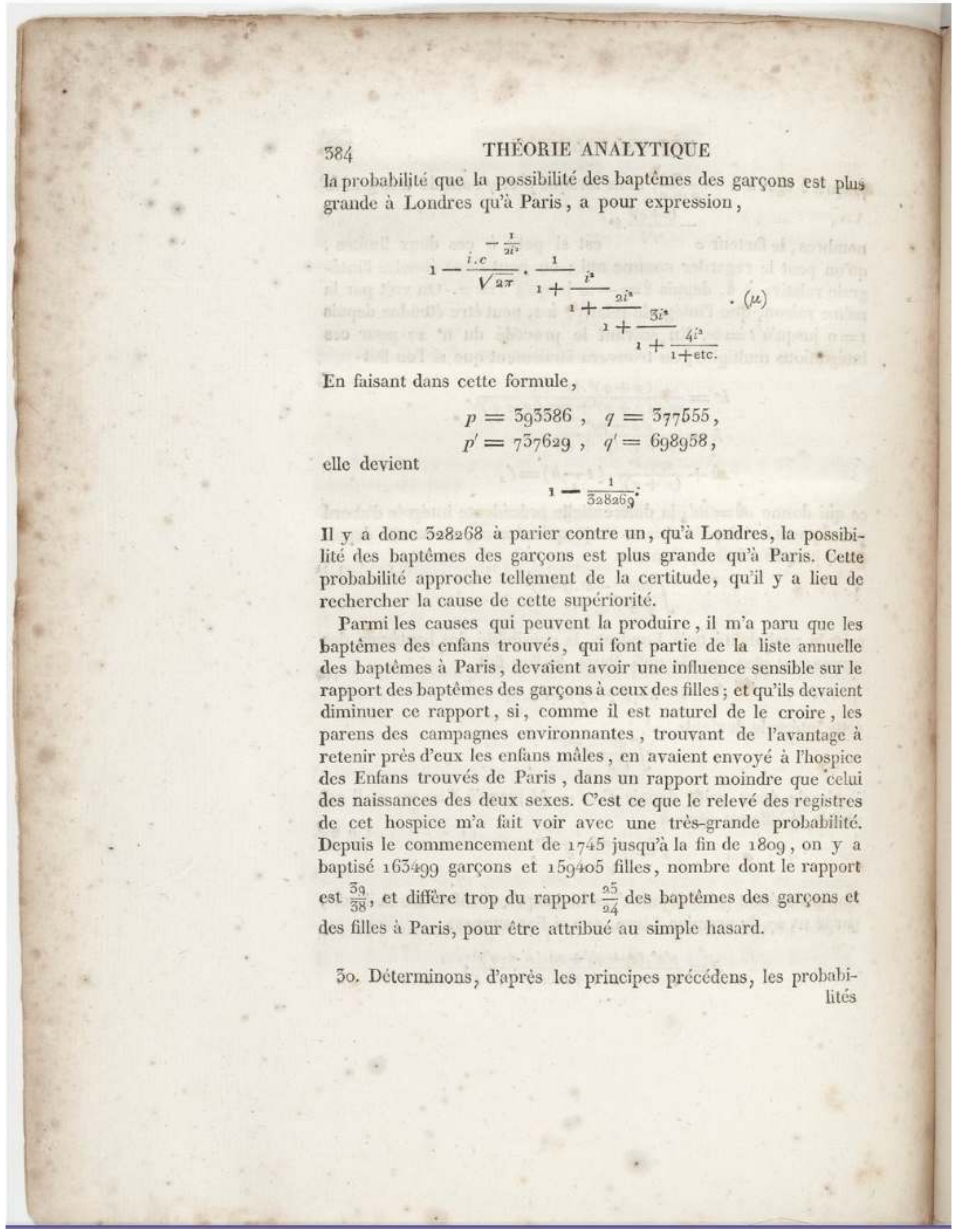
Enfin, dans l'espace des neuf années écoulées depuis le commencement de 1774 jusqu'à la fin de 1782, il est né dans le royaume de Naples, la Sicile non-comprise, 782352 garçons, et 746821 filles ; ce qui donne $\frac{22}{21}$ pour le rapport des naissances des garçons à celles des filles.

Les plus petits de ces nombres de naissances, sont relatifs à Paris ; d'ailleurs, c'est dans cette ville que les naissances des garçons et des filles, approchent le plus de l'égalité. Par ces deux raisons, la probabilité que la possibilité de la naissance d'un garçon surpasse $\frac{1}{2}$, doit y être moindre qu'à Londres et dans le royaume de Naples. Déterminons numériquement cette probabilité.

Nommons p le nombre des naissances masculines observées à Paris, q celui des naissances féminines, et x la possibilité d'une naissance masculine, c'est-à-dire la probabilité qu'un enfant qui doit naître, sera un garçon ; $1-x$ sera la possibilité d'une naissance féminine, et l'on aura la probabilité que sur $p+q$ naissances,

48

...



384 THÉORIE ANALYTIQUE

la probabilité que la possibilité des baptêmes des garçons est plus grande à Londres qu'à Paris, a pour expression,

$$1 - \frac{i.c^{-\frac{1}{2i^2}}}{\sqrt{2\pi}} \cdot \cfrac{1}{1+\cfrac{i^2}{1+\cfrac{2i^2}{1+\cfrac{3i^2}{1+\cfrac{4i^2}{1+\text{etc.}}}}}} \,. \quad (\mu)$$

En faisant dans cette formule,

$$p = 393386, \quad q = 377555,$$
$$p' = 737629, \quad q' = 698958,$$

elle devient

$$1 - \frac{1}{328269}.$$

Il y a donc 328268 à parier contre un, qu'à Londres, la possibilité des baptêmes des garçons est plus grande qu'à Paris. Cette probabilité approche tellement de la certitude, qu'il y a lieu de rechercher la cause de cette supériorité.

Parmi les causes qui peuvent la produire, il m'a paru que les baptêmes des enfans trouvés, qui font partie de la liste annuelle des baptêmes à Paris, devaient avoir une influence sensible sur le rapport des baptêmes des garçons à ceux des filles ; et qu'ils devaient diminuer ce rapport, si, comme il est naturel de le croire, les parens des campagnes environnantes, trouvant de l'avantage à retenir près d'eux les enfans mâles, en avaient envoyé à l'hospice des Enfans trouvés de Paris, dans un rapport moindre que celui des naissances des deux sexes. C'est ce que le relevé des registres de cet hospice m'a fait voir avec une très-grande probabilité. Depuis le commencement de 1745 jusqu'à la fin de 1809, on y a baptisé 163499 garçons et 159405 filles, nombre dont le rapport est $\frac{39}{38}$, et diffère trop du rapport $\frac{25}{24}$ des baptêmes des garçons et des filles à Paris, pour être attribué au simple hasard.

30. Déterminons, d'après les principes précédens, les probabilités

Example: Laplace and gender bias

laplace.ml

```
open Basic.Rejection_sampling_hard

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
```

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

Example: Laplace and gender bias

laplace.ml

```
open Basic.Rejection_sampling_hard

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
```

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

Never terminate!

Example: Laplace and gender bias

laplace.ml

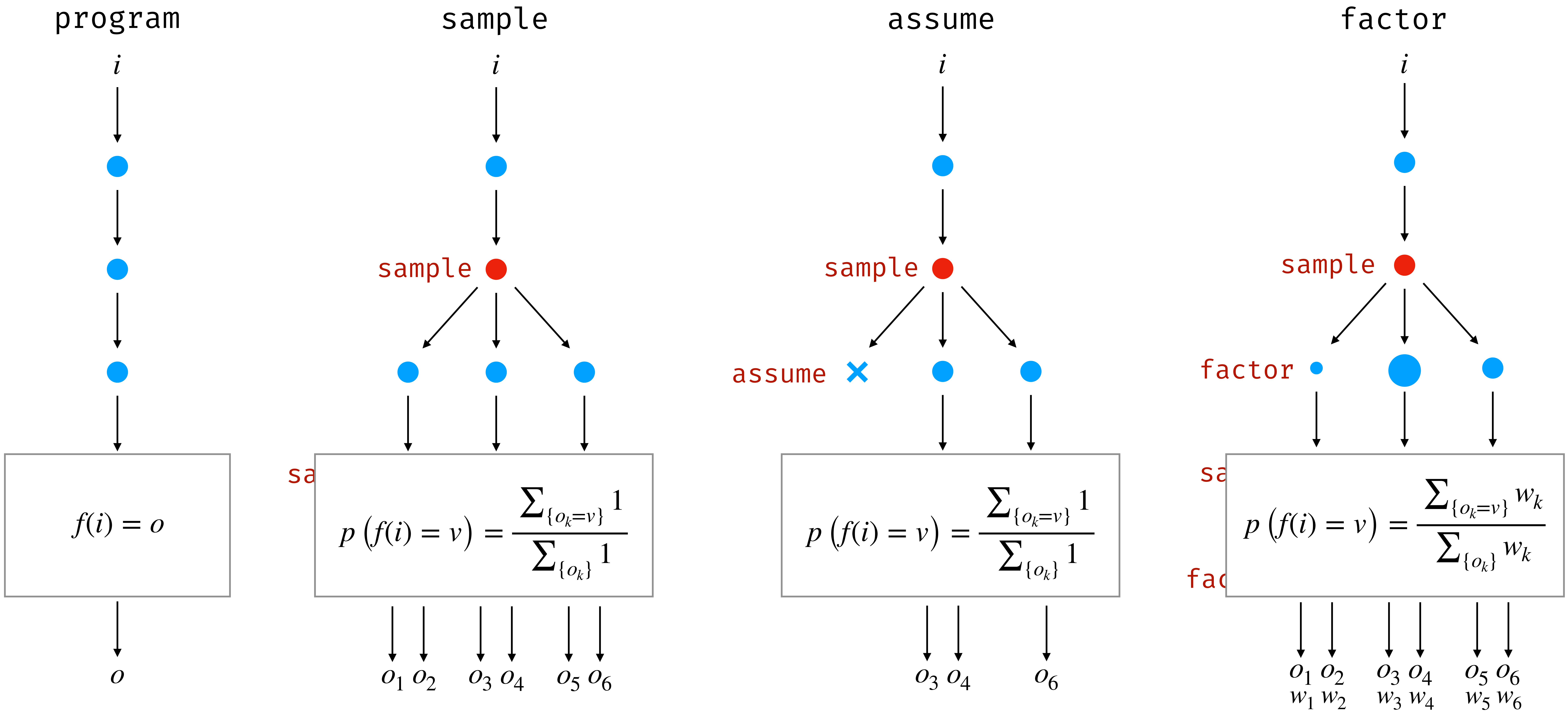
```
open Basic.Rejection_sampling_hard

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
```

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

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

Never terminate!

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



Importance Sampling

Runtime

Importance sampling

basic.ml

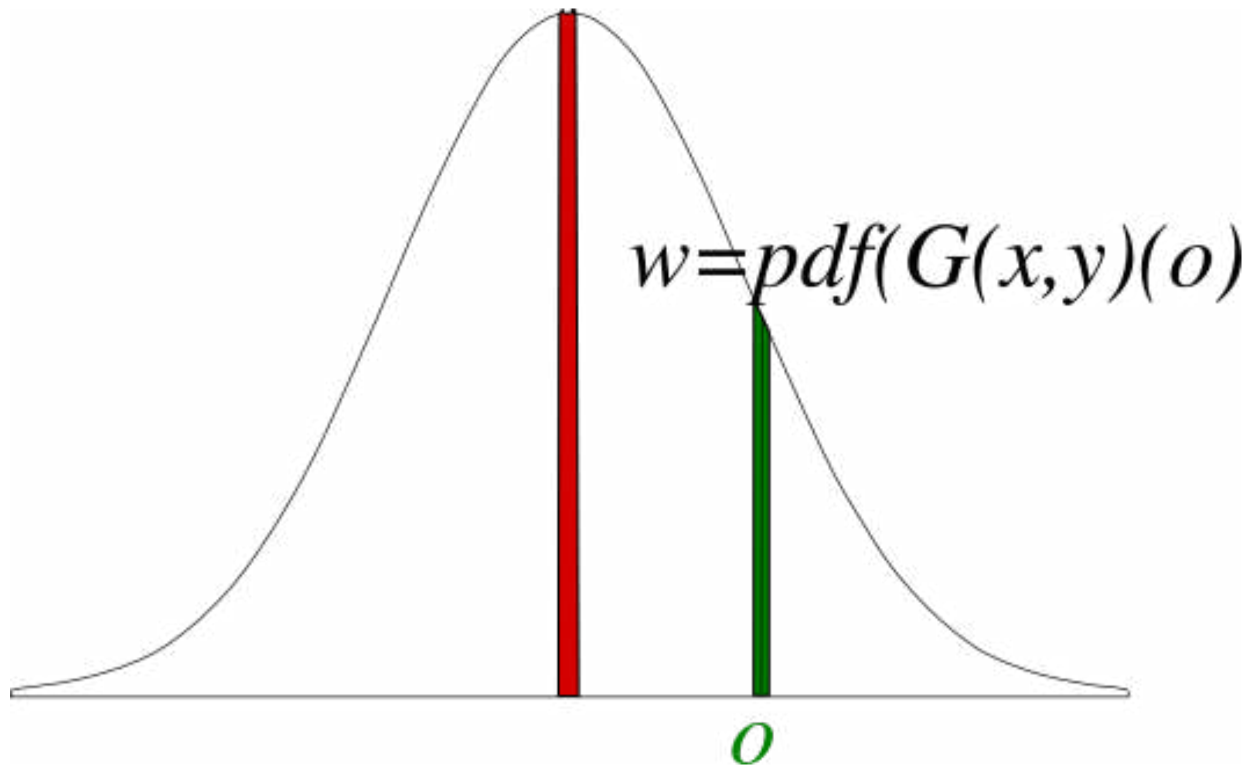
```
module Importance_sampling : sig
  type prob
  val sample : prob → 'a Distribution.t → 'a
  val factor : prob → float → unit
  val infer : ?n:int → (prob → 'a → 'b) → 'a → 'b Distribution.t
end = struct ... end
```

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 = { mutable score : float }

  let sample _prob d = Distribution.draw d
  let factor prob s = prob.score ← prob.score +. s
  let observe prob d x = factor prob (Distribution.logpdf d x)

  let infer ?(n = 1000) model obs =
    let gen _ =
      let prob = { score = 0. } in
      let value = model prob data in
      (value, prob.score)
    in
    let support = List.init n gen in
    Distribution.categorical ~support
end
```

Exercice: Can you do it with a monad instead of an explicit `prob`?

Example: Coin

coin.ml

```
open Basic.Importance_sampling

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

let data = [false; true; true; false; false; false; false; false; false; false]

let _ =
  let dist = infer coin data 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
```

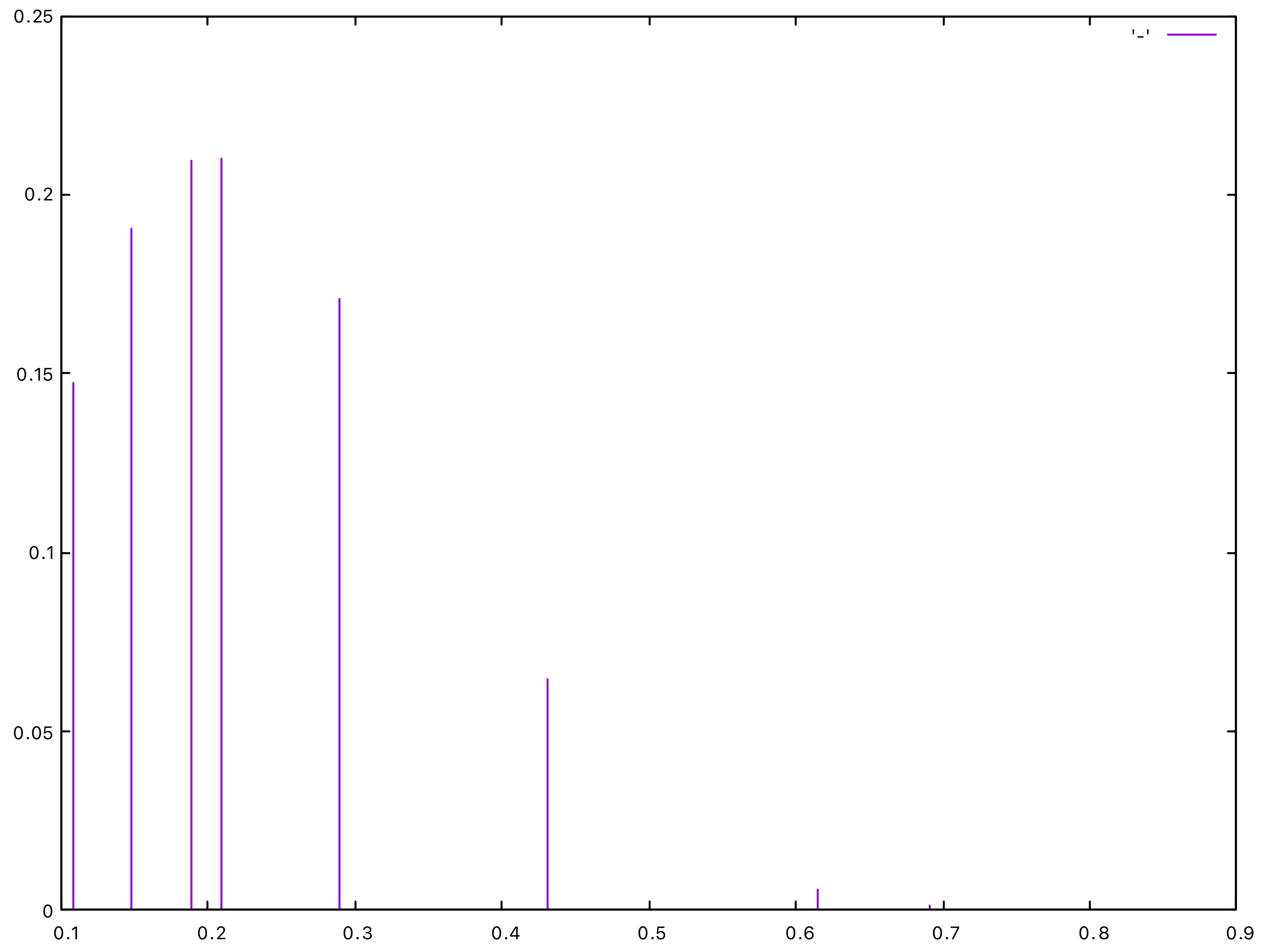
Example: Coin

coin.ml

```
open Basic.Importance_sampling

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

let data = [false; true; true; false; false; false; false; false; false; false]

let _ =
  let dist = infer coin data 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

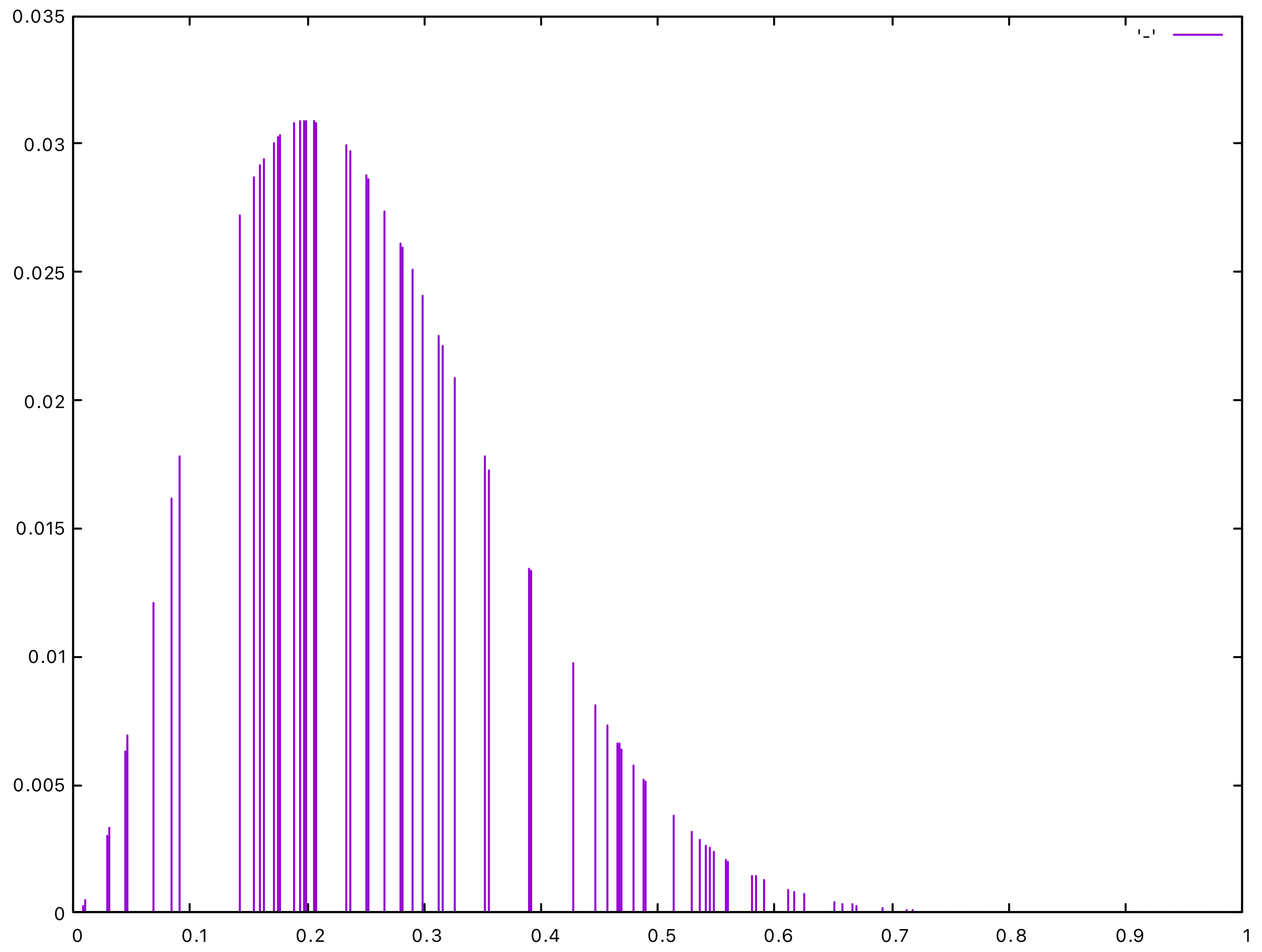


Example: Coin

coin.ml

```
open Basic.Importance_sampling

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

let data = [false; true; true; false; false; false; false; false; false; false]

let _ =
  let dist = infer coin data 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
```

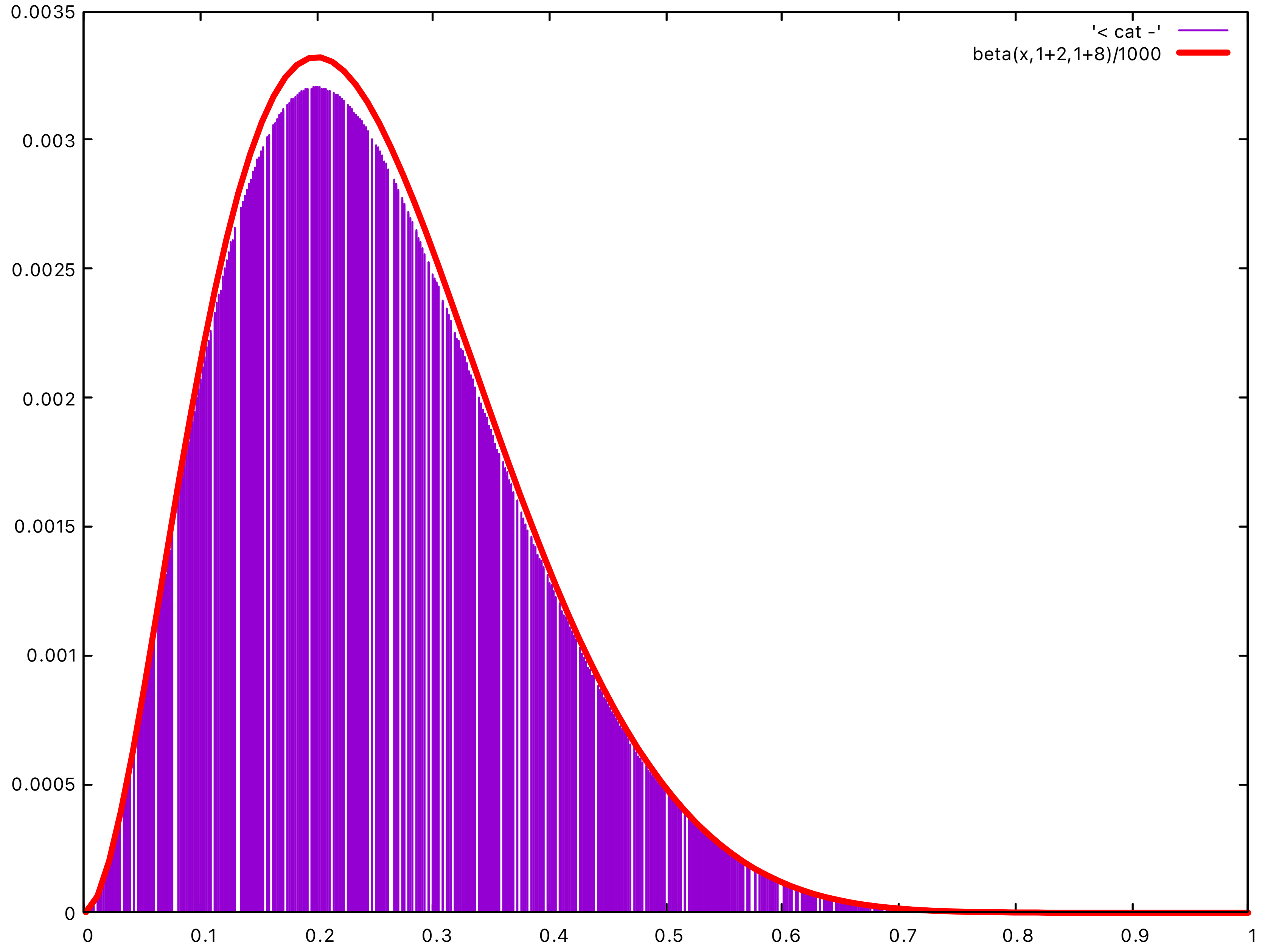
Example: Coin

coin.ml

```
open Basic.Importance_sampling

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

let data = [false; true; true; false; false; false; false; false; false; false]

let _ =
  let dist = infer coin data 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_hard = 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_hard = 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.score ← prob.score +. s

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

Soft conditioning

Kernel Semantics

Probabilistic Programming Languages

Types as mesurable spaces

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 `true`, `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)

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 `true`, `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: ϕ (global declarations), γ (local variables)
- Given the declarations ϕ, $[\![e]\!]^\phi : \Gamma \to t$ is a measurable function
- $[\![e]\!]^\phi_\gamma$ is a value of type t

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

- Given the declarations ϕ, 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

(Un)normalized measures

set of possible outcomes

model

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

Unnormalized measure

score

environments

(Un)normalized measures

set of possible outcomes

model

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

Unnormalized measure

score

environments

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

normalize over all possible values

(Un)normalized measures

model

set of possible outcomes

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

Unnormalized measure

environments

score

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

Distribution

normalize over all possible values

Deterministic semantics

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

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

Probabilistic semantics

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

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

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

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

Probabilistic semantics

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

$$\{\!\![e]\!\!\}^{\phi}_{\gamma} = \lambda U.\ \delta_{\llbracket e \rrbracket^{\phi}_{\gamma}}(U) \textit{ if kindOf}(e) = D$$

$$\{\!\![f(e)]\!\!\}^{\phi}_{\gamma} = \lambda U.\ \phi(f)(\llbracket e \rrbracket^{\phi}_{\gamma})(U)$$

$$\{\!\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!\!\}^{\phi}_{\gamma} = \lambda U.\textit{ if } \llbracket e_1 \rrbracket^{\phi}_{\gamma} \textit{ then } \{\!\![e_2]\!\!\}^{\phi}_{\gamma}(U) \textit{ else } \{\!\![e_3]\!\!\}^{\phi}_{\gamma}(U)$$

$$\{\!\![\texttt{let } p \texttt{ = } e_1 \texttt{ in } e_2]\!\!\}^{\phi}_{\gamma} = \lambda U. \int_{\llbracket \textit{typeOf}(e_1) \rrbracket} \{\!\![e_1]\!\!\}^{\phi}_{\gamma}(dv)\ \{\!\![e_2]\!\!\}^{\phi}_{\gamma + [p \leftarrow v]}$$

$$\{\!\![\texttt{sample}(e)]\!\!\}^{\phi}_{\gamma} = \lambda U.\ \llbracket e \rrbracket^{\phi}_{\gamma}(U)$$

$$\{\!\![\texttt{factor}(e)]\!\!\}^{\phi}_{\gamma} = \lambda U.\ \llbracket e \rrbracket^{\phi}_{\gamma} \cdot \delta_{()}(U)$$

$$\{\!\![\texttt{observe}(e_1, e_2)]\!\!\}^{\phi}_{\gamma} = \lambda U.\ \mathrm{pdf}(\llbracket e_1 \rrbracket^{\phi}_{\gamma})(\llbracket e_2 \rrbracket^{\phi}_{\gamma}) \cdot \delta_{()}(U)$$

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

Careful with 0, and ∞...

Example : Gaussian

my_gaussian.ml

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

Example : Gaussian

my_gaussian.ml

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

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

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
```

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
```

Example : Coin

coin.ml

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

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

Example : Coin

coin.ml

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

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

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))
```

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))
```

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

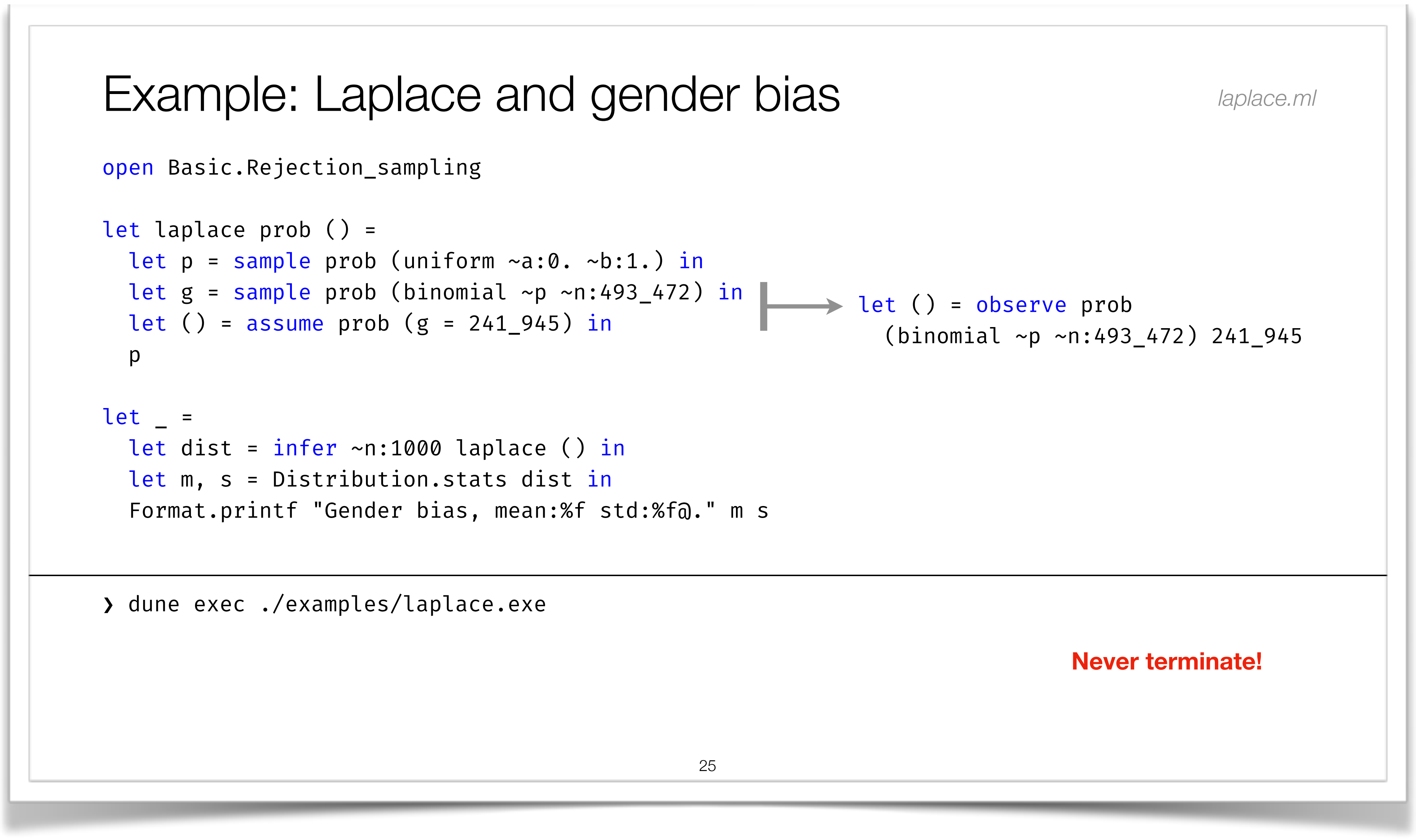
$$[\![\texttt{infer (coin (x1, ... , xn))}]\!]_{[\texttt{coin}]} = \frac{\int_U z^{\text{\#heads}} (1-z)^{\text{\#tails}} \, dz}{\int_0^1 z^{\text{\#heads}} (1-z)^{\text{\#tails}} \, dz} = \frac{\int_U z^{\text{\#heads}} (1-z)^{\text{\#tails}} \, dz}{\mathrm{B}(\text{\#heads}+1, \text{\#tails}+1)} =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

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
  ```
  ```

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
  ```


  ```

Improper priors

Uniform priors on bounded domains

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

Improper priors

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

$$\begin{aligned}[\![\mathtt{improper}]\!]_\emptyset (U) &= \int_U \mathit{Gaussian}(0,1)(dx)\ \frac{1}{f(x)}\ dx \\ &= \int_U f(x)\ \frac{1}{f(x)}\ dx \\ &= \lambda(U)\end{aligned}$$

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-24-25

- Go to `td/td4-stan`
- Launch `jupyter notebook` (or `jupyter lab`)

Requirements

- Pandas
- CmdStanPy
- Jupyter
- Matplotlib

https://mc-stan.org/

TP : A short introduction to Stan

Everything is on Github: https://github.com/mpri-probprog/probprog-24-25

- Go to `td/td4-stan`
- Launch `jupyter notebook` (or `jupyter lab`)

Requirements

- Pandas
- CmdStanPy
- Jupyter
- Matplotlib

https://mc-stan.org/