Introduction to Probabilistic Programming

Guillaume Baudart Christine Tasson

EJCIM 2024

Probabilistic programming languages

Probabilistic programming languages

General purpose programming languages extended with 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`

Probabilistic programming languages

General purpose programming languages extended with 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`

Multiple examples:

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

Probabilistic programming languages

General purpose programming languages extended with probabilistic constructs

- `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`

Multiple examples:

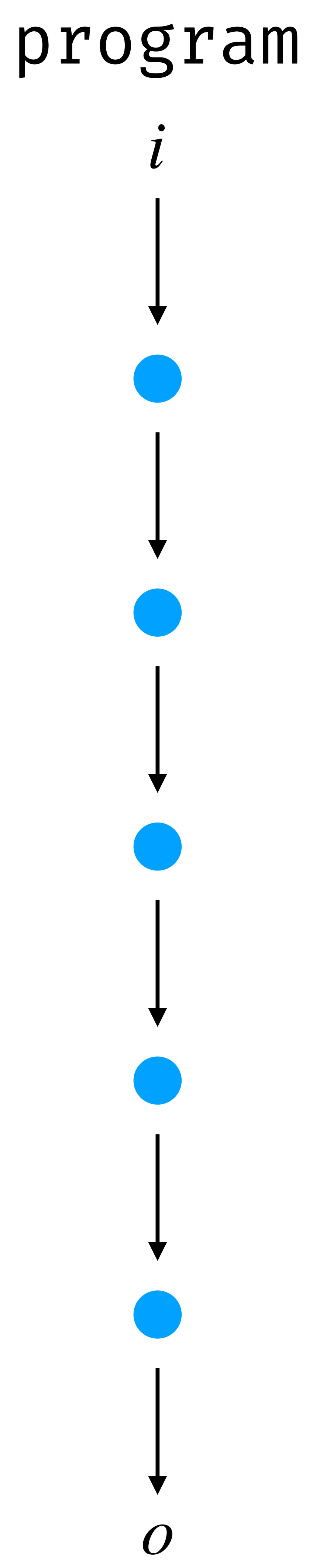
- Church, Anglican (lisp, clojure), 2008
- WebPPL (javascript), 2014
- Pyro/NumPyro (python), 2017/2019
- Gen (julia), 2018
- ProbZelus (Zelus), 2019
- ...

More and more, incorporating new ideas:

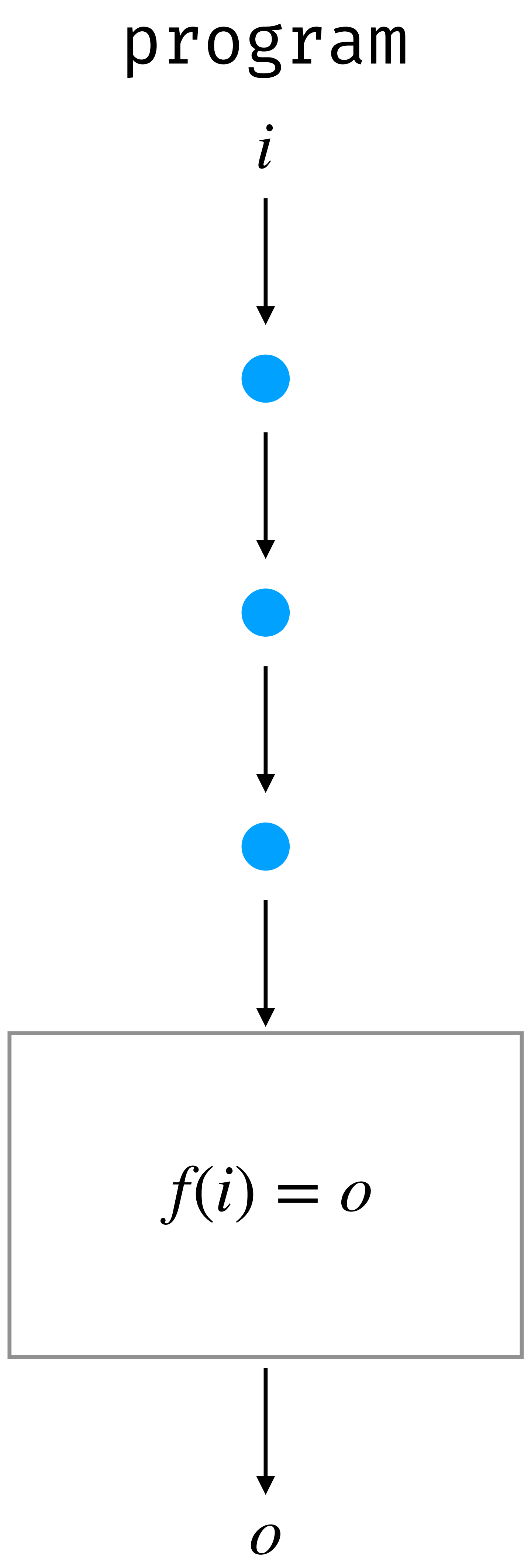
- New inference techniques, e.g., stochastic variational inference (SVI)
- Interaction with neural nets (deep probabilistic programming)

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

$\texttt{infer} : (\alpha \to \beta) \to \alpha \to \beta\ \texttt{dist}$

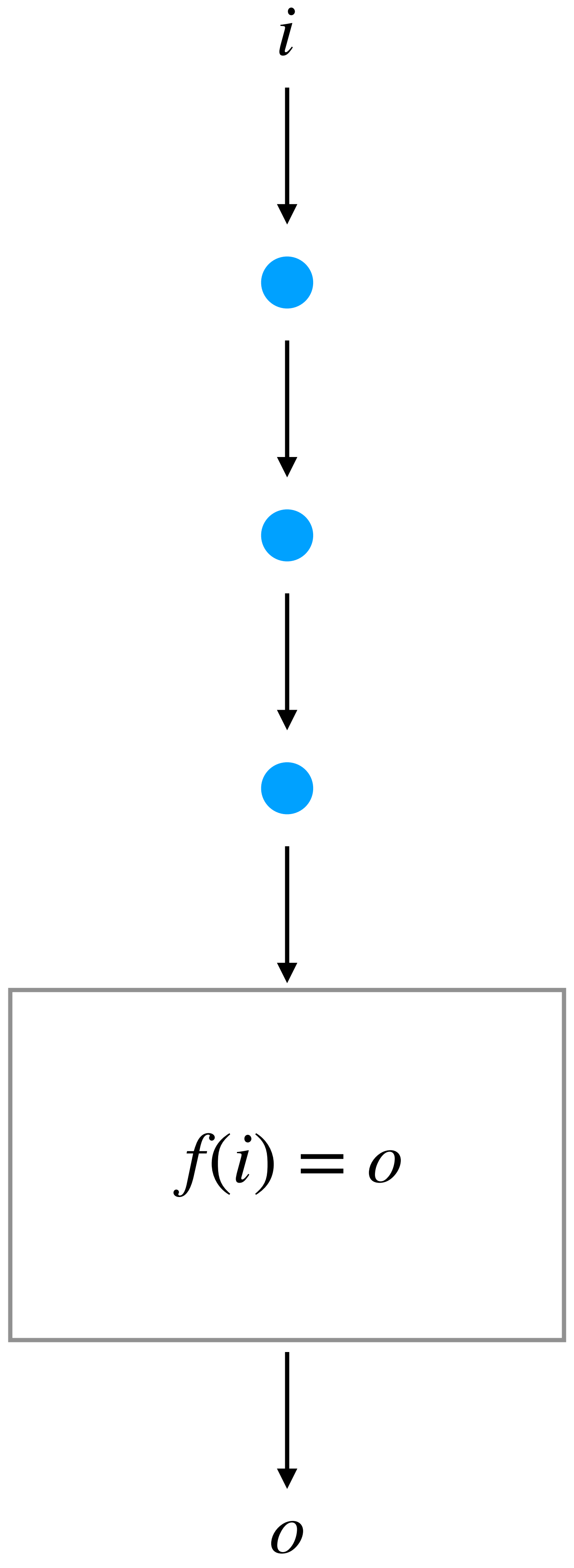


`infer` : $(\alpha \to \beta) \to \alpha \to \beta$ `dist`



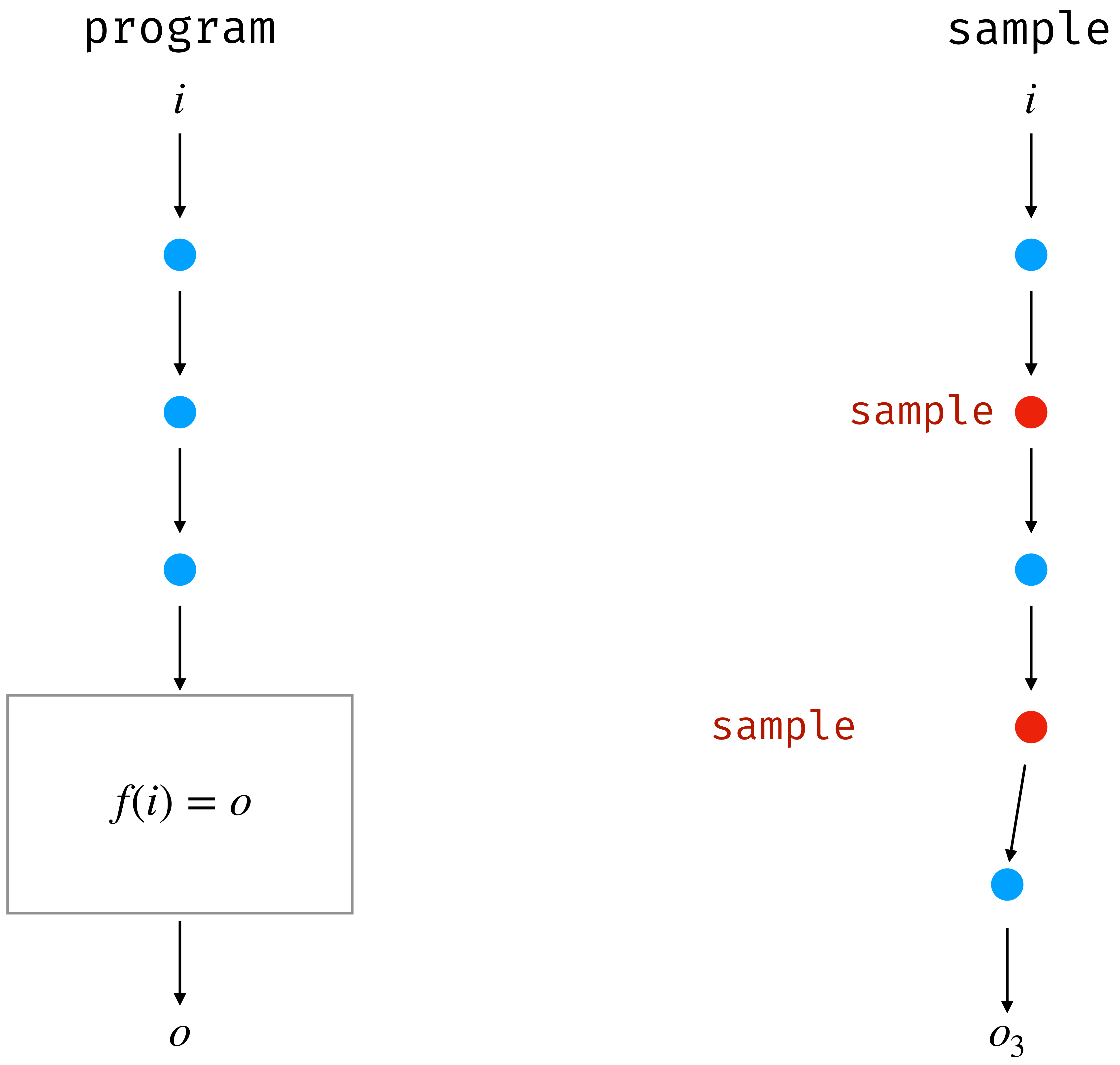
$\texttt{infer} : (\alpha \to \beta) \to \alpha \to \beta\ \texttt{dist}$

`program`

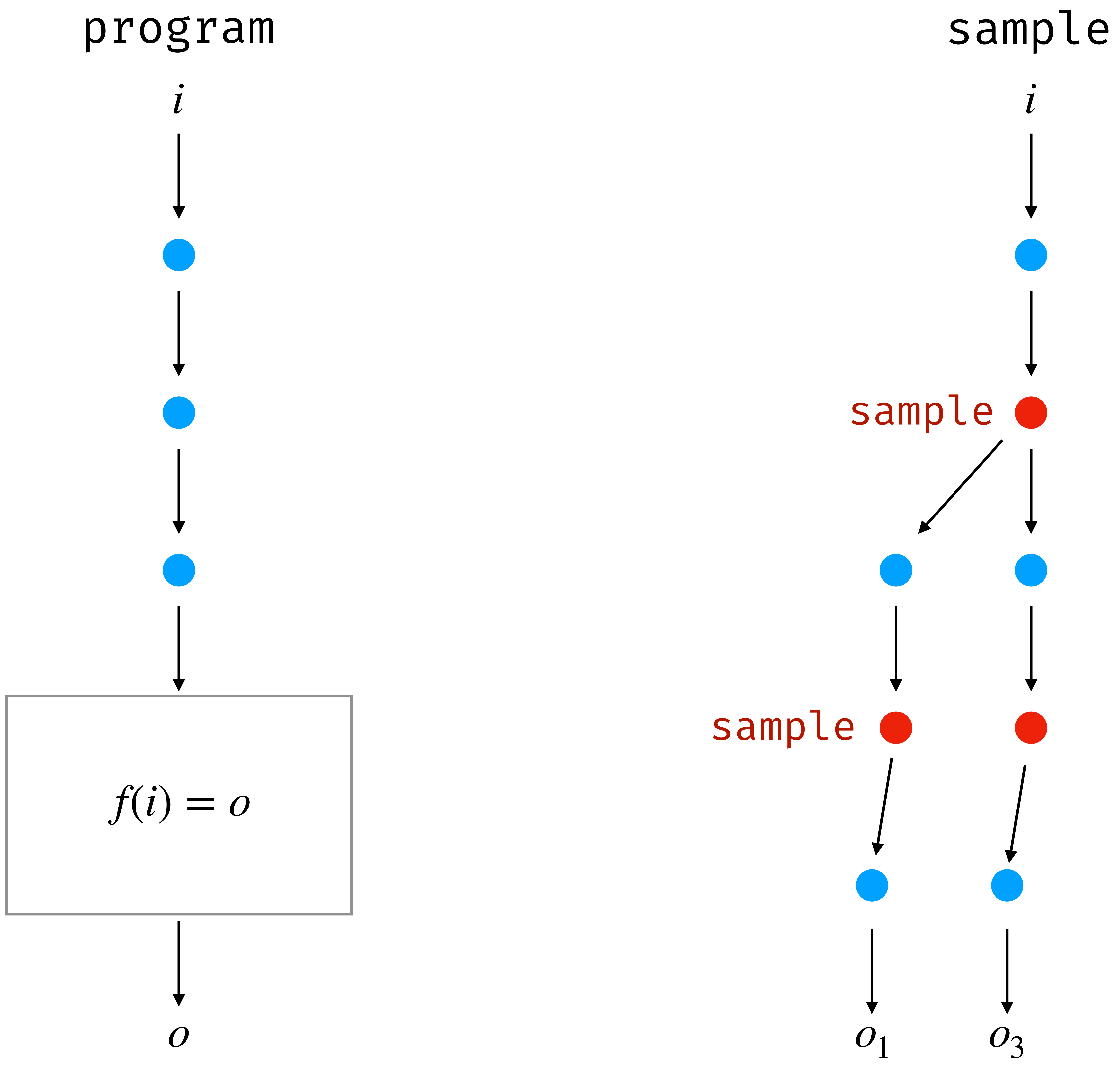


`sample`

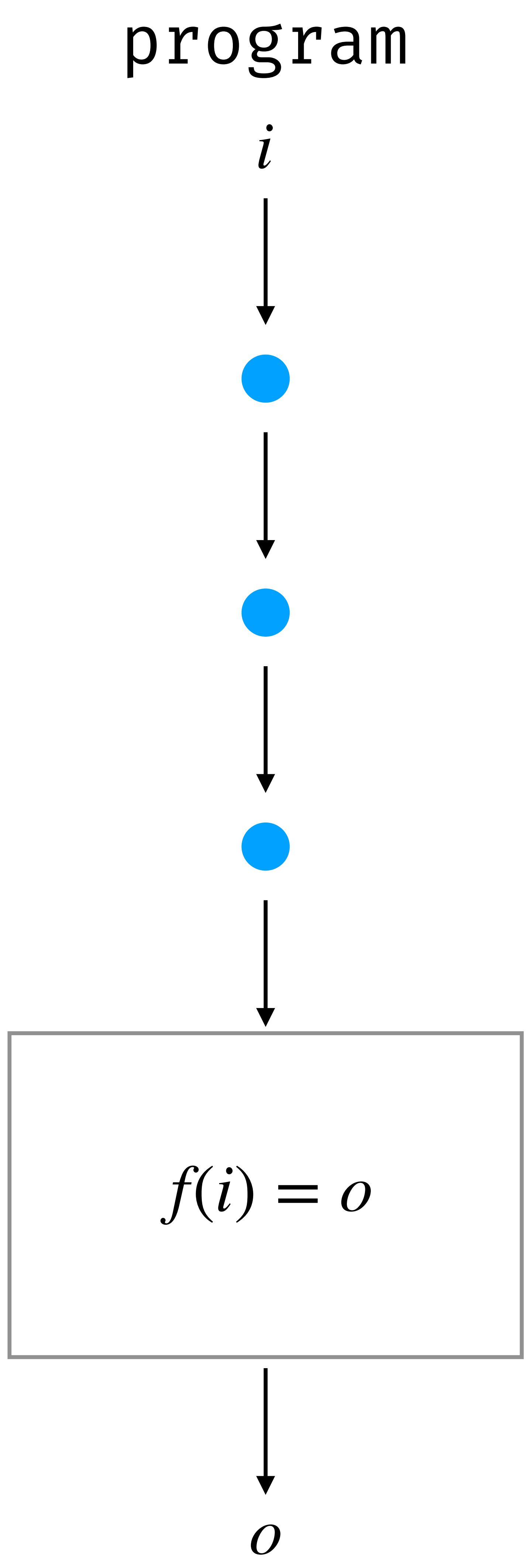
`infer` : $(\alpha \to \beta) \to \alpha \to \beta$ `dist`

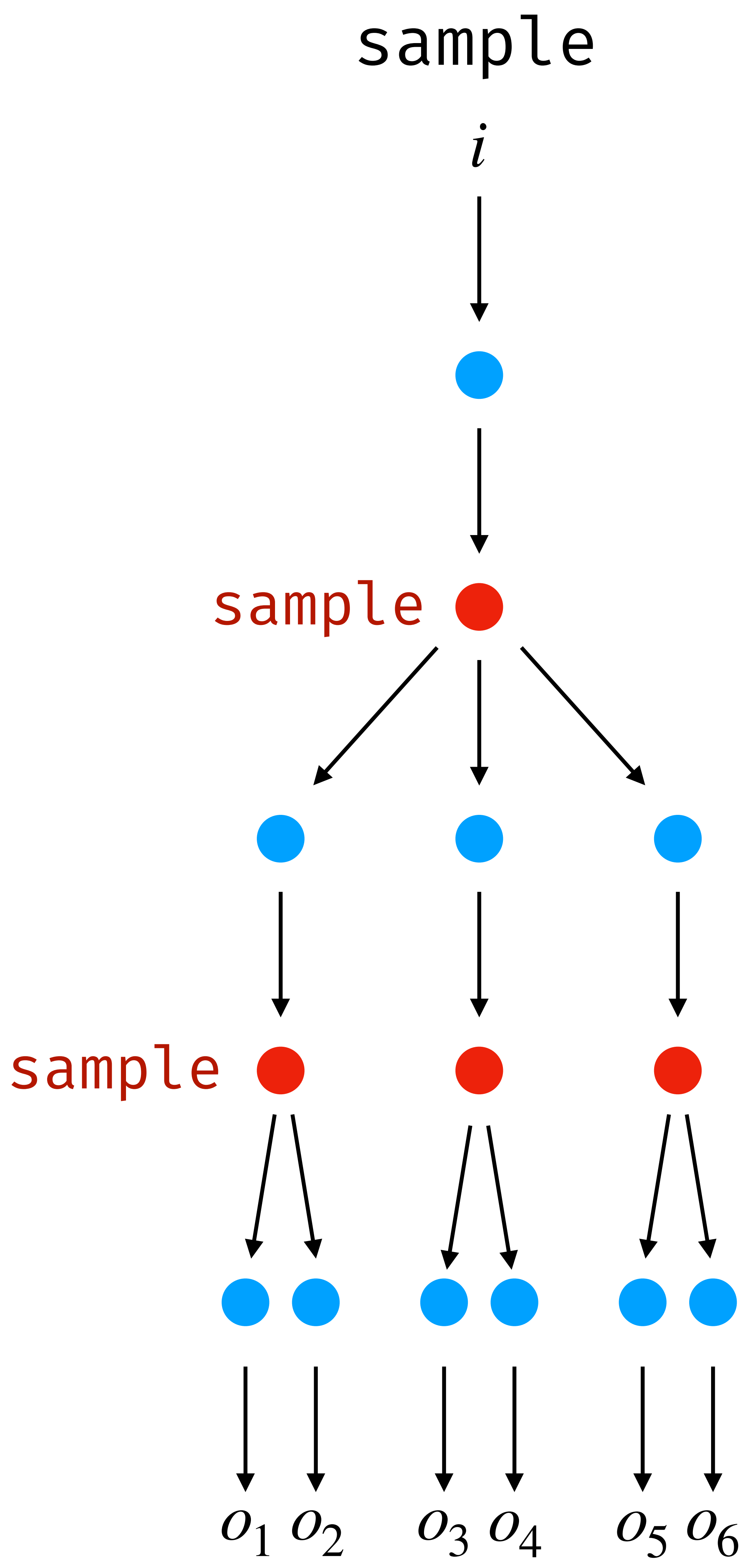


infer : $(\alpha \to \beta) \to \alpha \to \beta$ dist

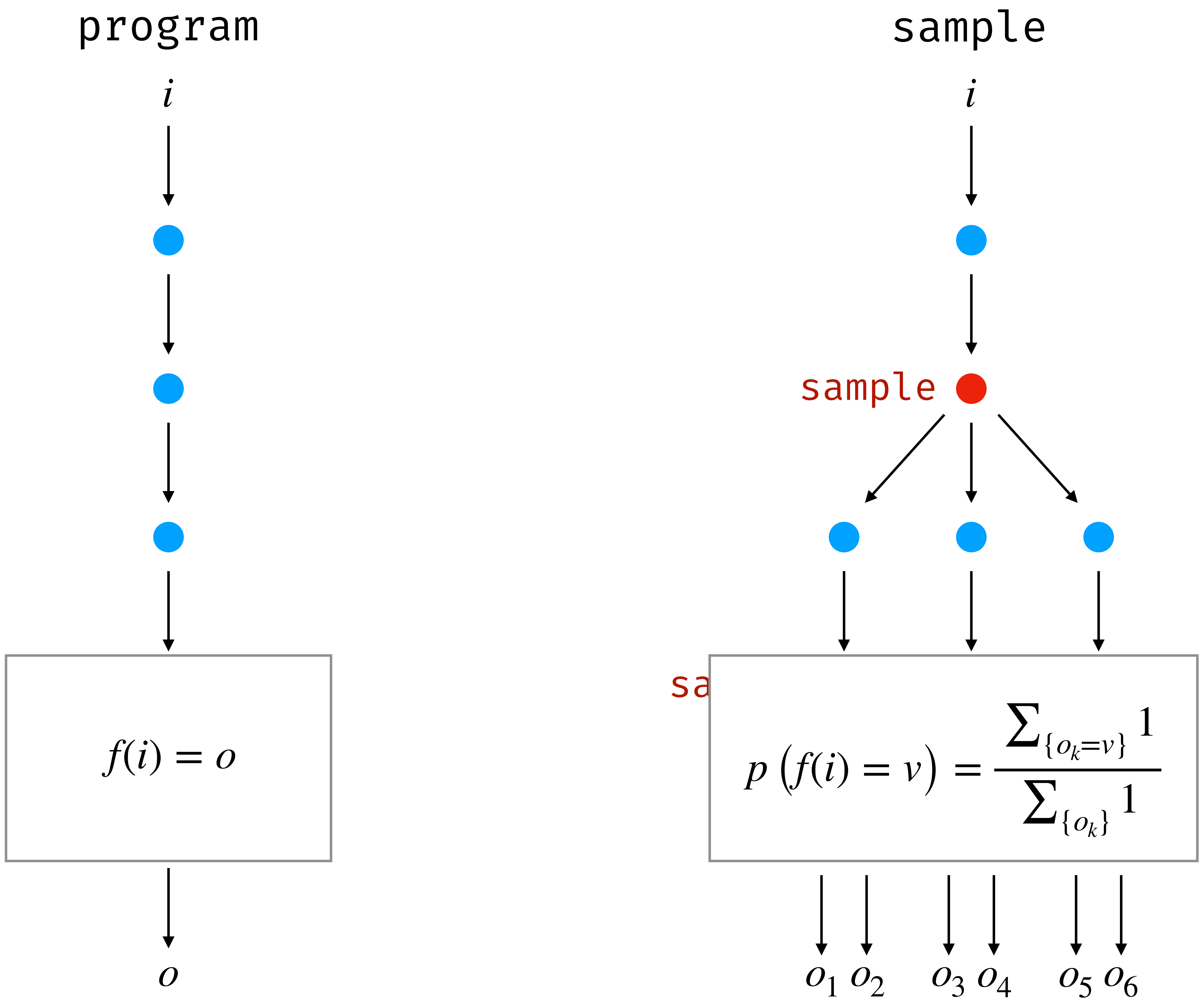


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ `dist`

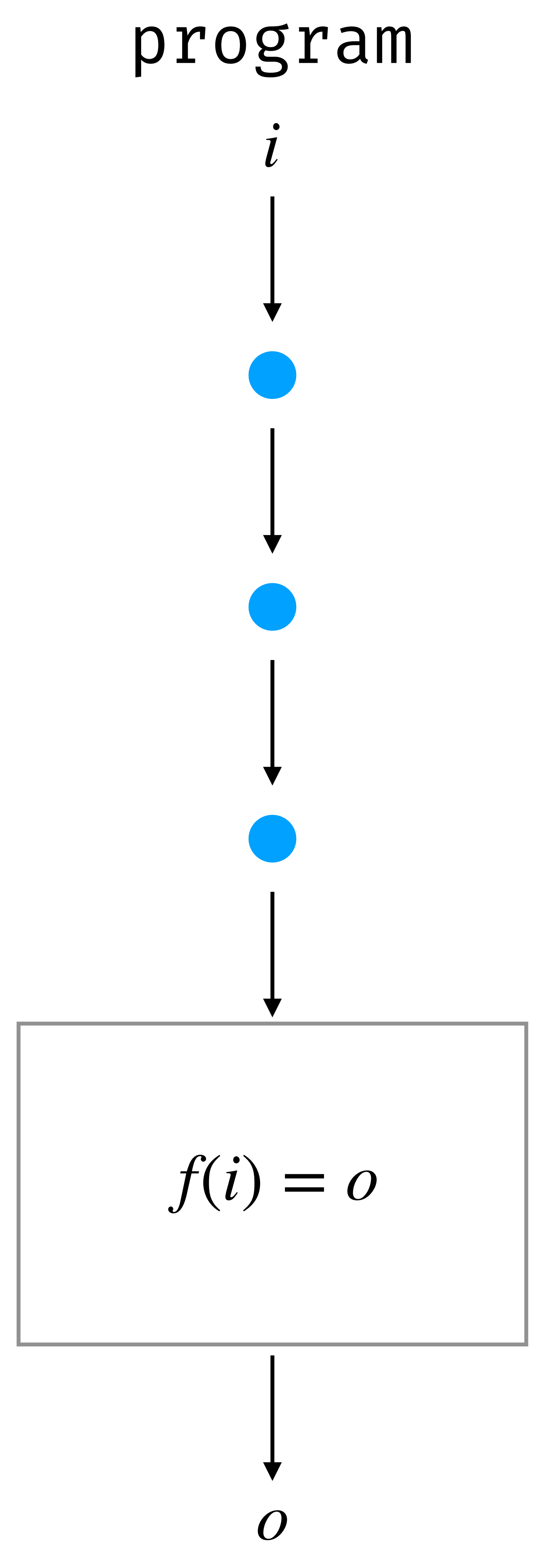


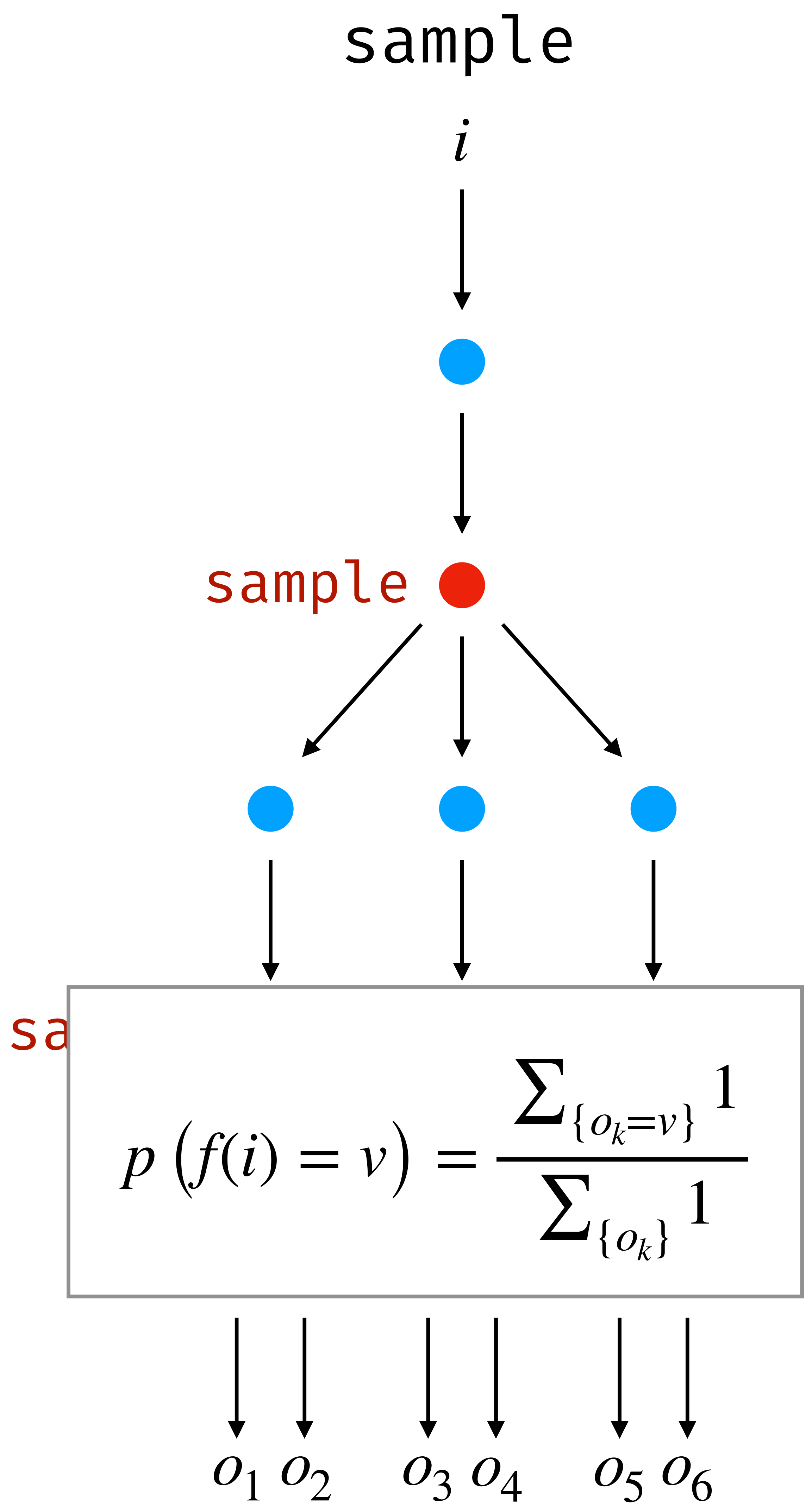


infer : $(\alpha \to \beta) \to \alpha \to \beta$ dist



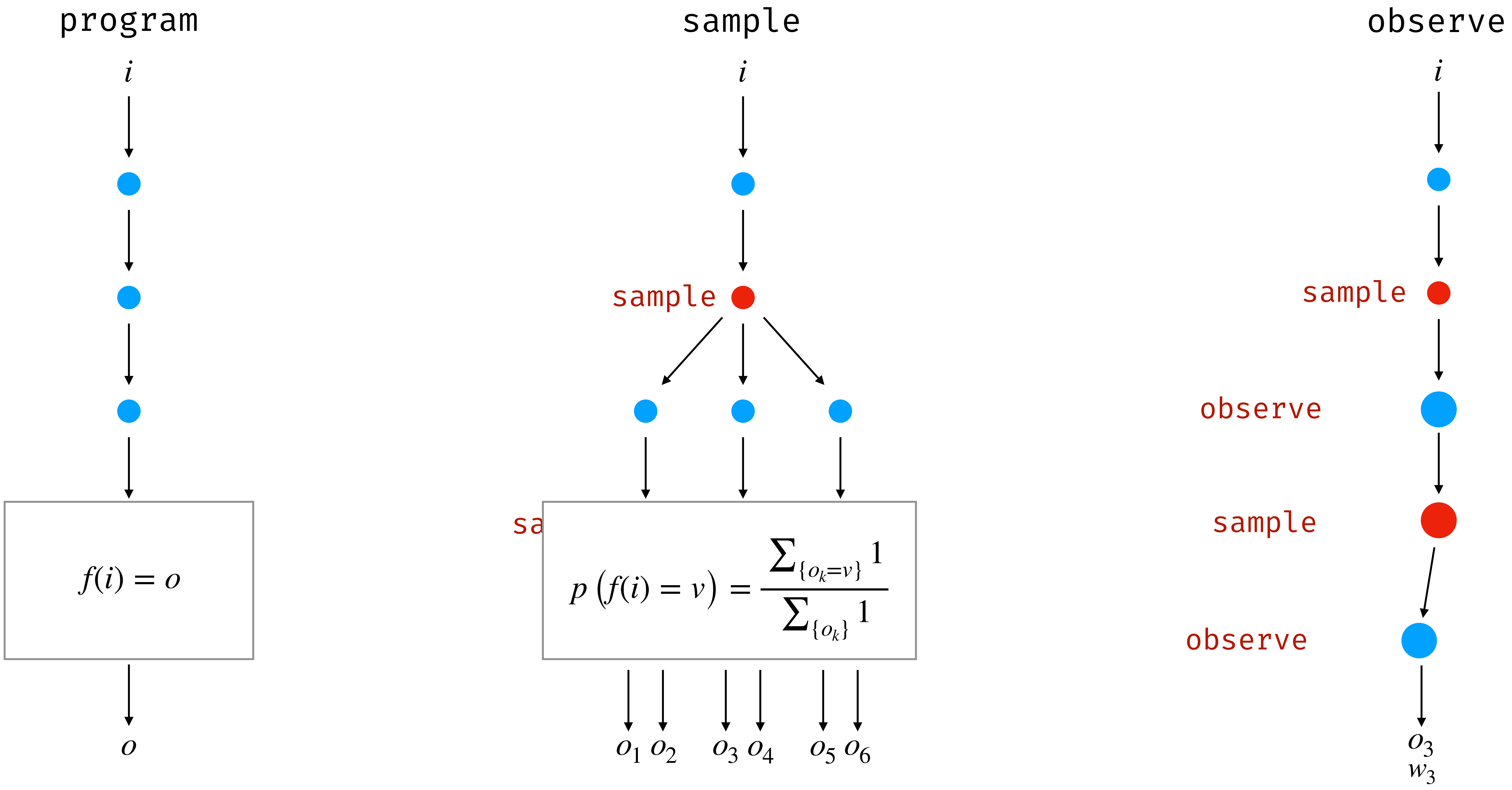
$\texttt{infer} : (\alpha \to \beta) \to \alpha \to \beta \; \texttt{dist}$



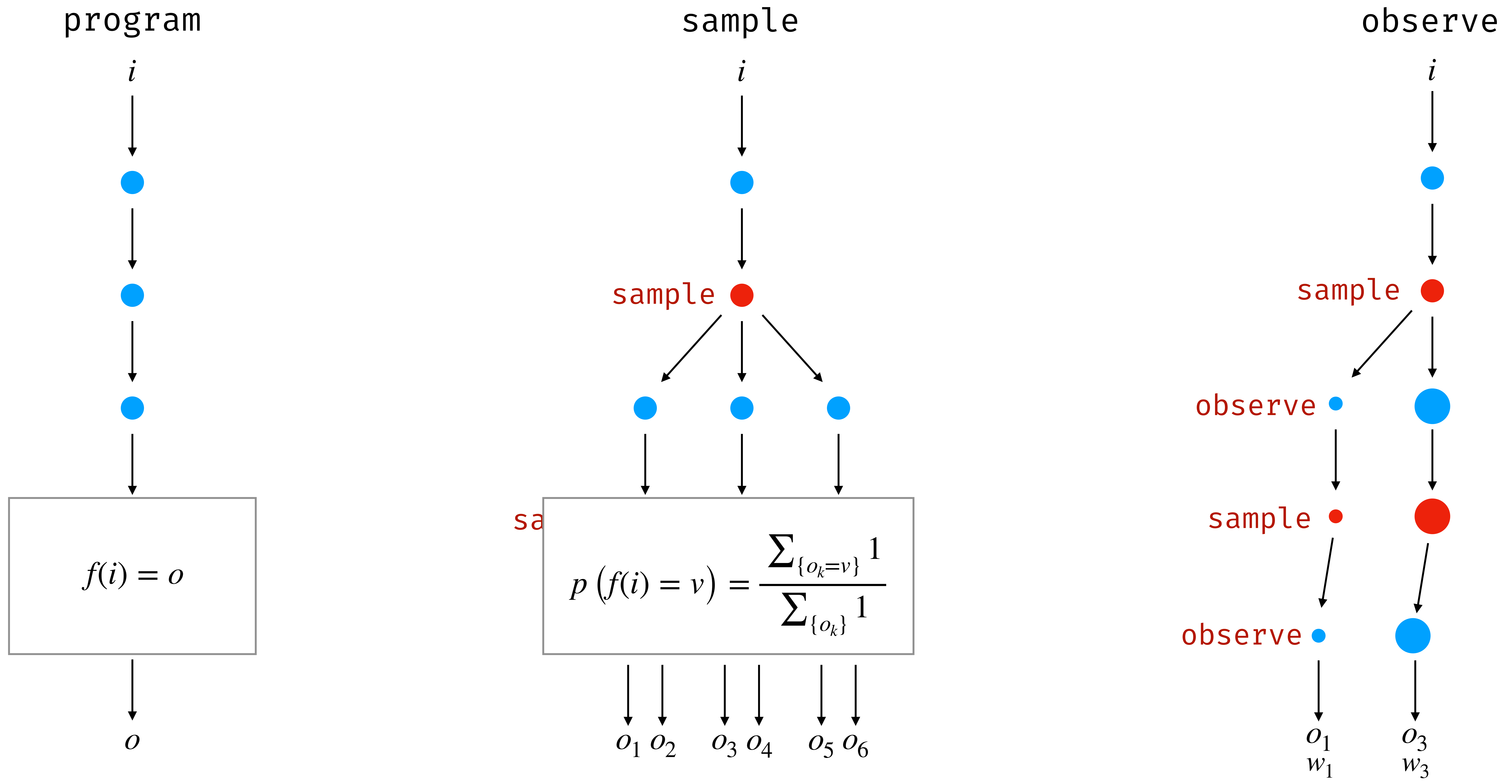


observe

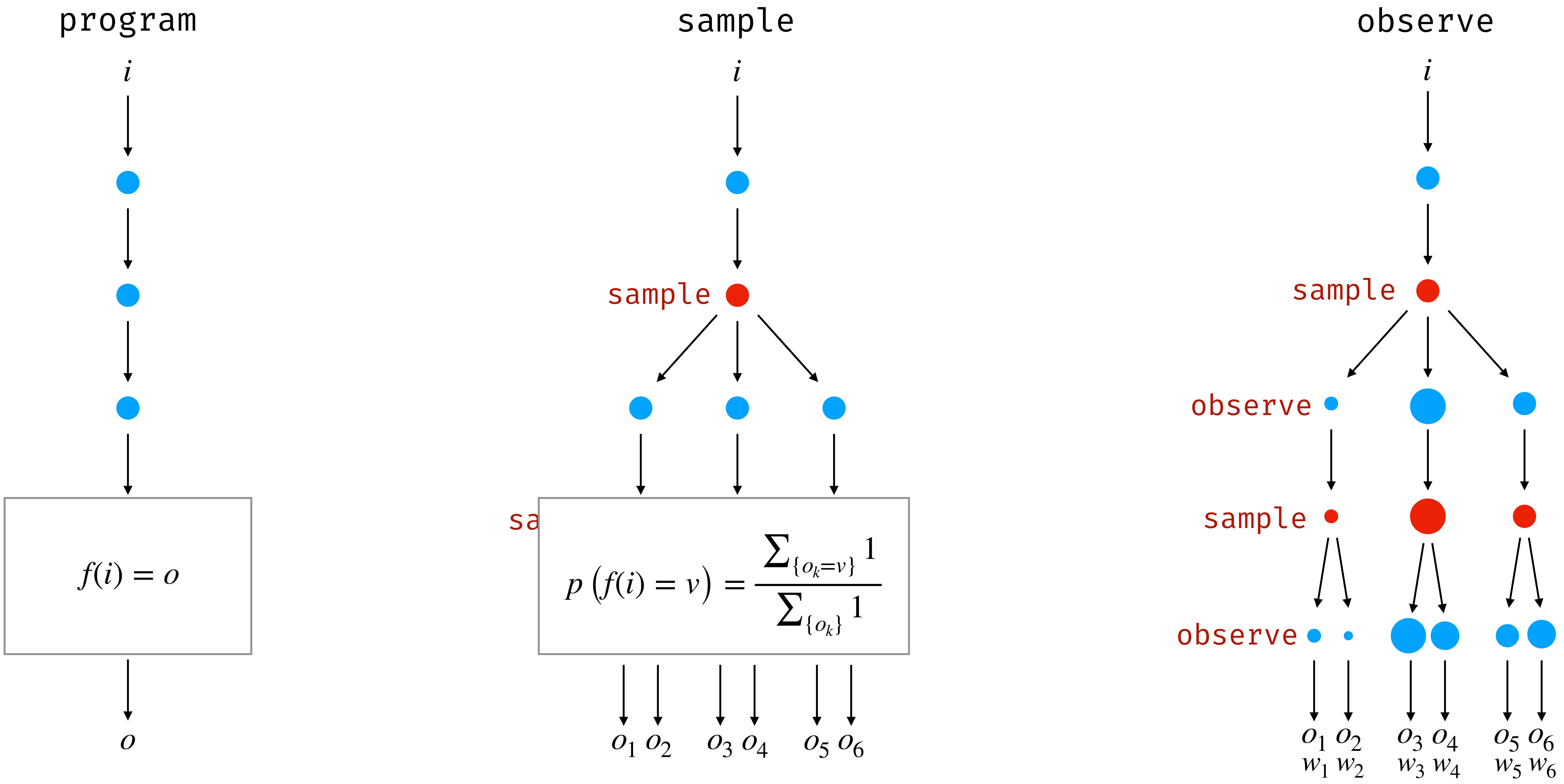
infer : $(\alpha \to \beta) \to \alpha \to \beta$ dist



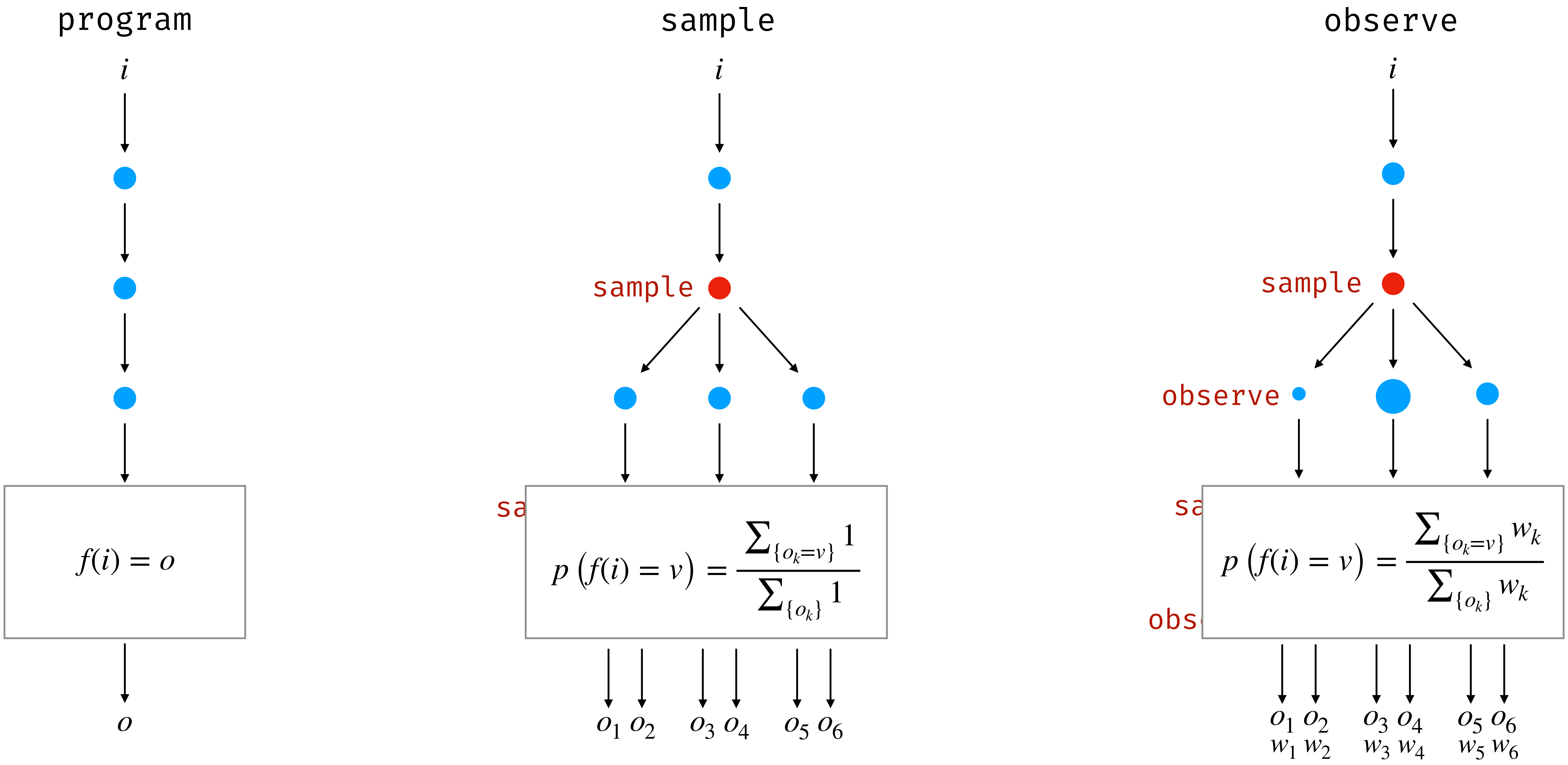
infer : $(\alpha \to \beta) \to \alpha \to \beta$ dist



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



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



Bayesian reasoning

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

$$\propto p(x)\ p(y_1, \ldots, y_n \mid x)$$ (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 $y_1, \ldots, y_n$

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

posterior

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

prior *likelihood*

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

Bayesian reasoning

Bayesian Inference: learn parameters from data

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

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

posterior

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

prior *likelihood*

```
def model(y1, … , yn):
  x = sample prior
  observe ((likelihood x), (y1, … , yn))
  return x

infer(model, (y1, … , yn))
```

Thomas Bayes (1701-1761)

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

Probabilistic programming

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`

More general than classic Bayesian Reasoning

```
def weird() =
  b = sample(Bernoulli(0.5))
  mu = 0.5 if (b = 1) else 1.0
  theta = sample(Gaussian(mu, 1.0))
  if theta > 0.:
    observe (Gaussian(mu, 0.5), theta)
    return theta
  else:
    return weird ()
```

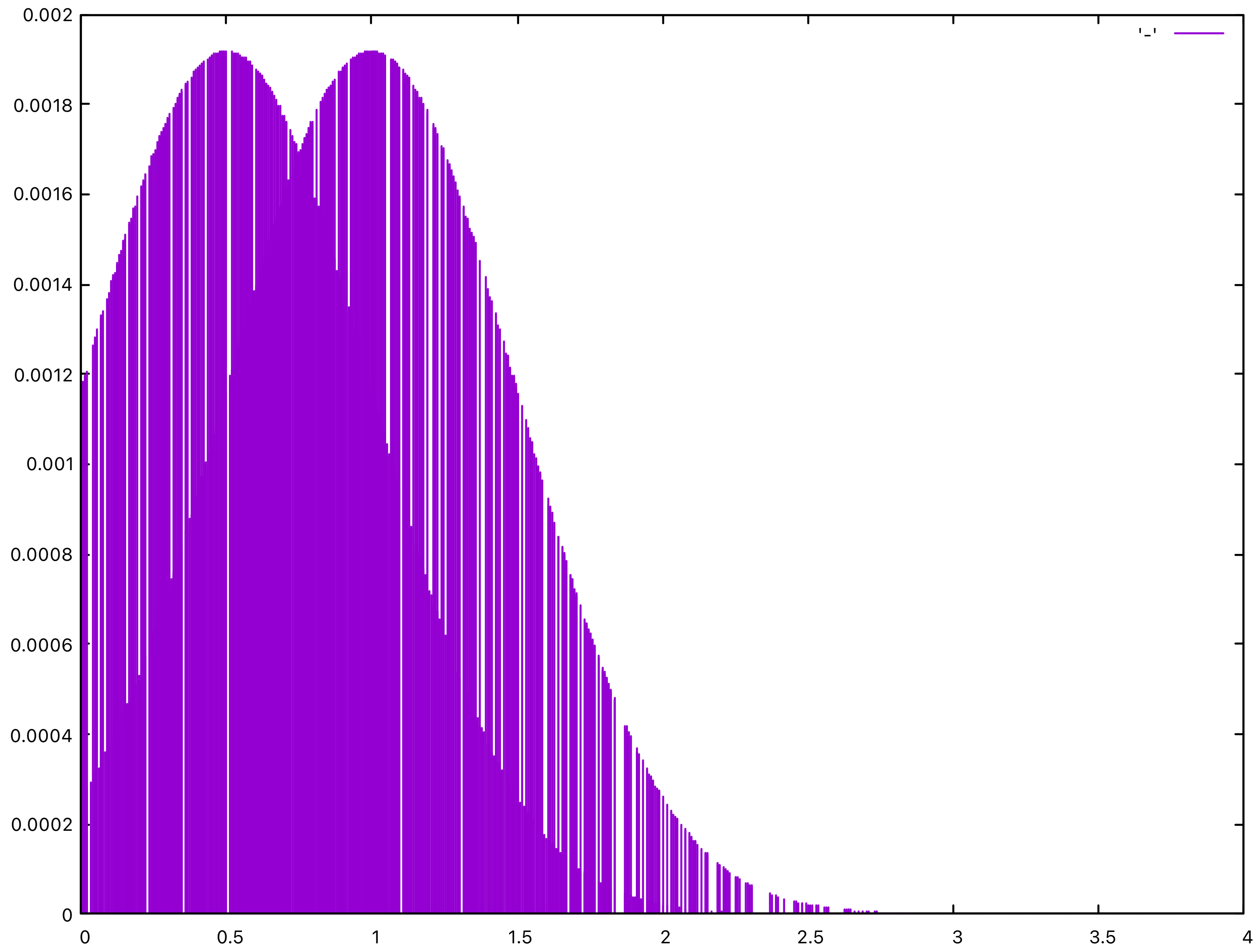
Probabilistic programming

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`

More general than classic Bayesian Reasoning

```
def weird() =
  b = sample(Bernoulli(0.5))
  mu = 0.5 if (b = 1) else 1.0
  theta = sample(Gaussian(mu, 1.0))
  if theta > 0.:
    observe (Gaussian(mu, 0.5), theta)
    return theta
  else:
    return weird ()
```



Outline

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

I - Approximate inference

- Importance sampling
- Markov Chain Monte Carlo (MCMC) with Metropolis-Hastings

II - Labs: Introduction to Sequential Monte Carlo methods

- State-space models
- Resampling

III - Density semantics

- Weighted samplers
- Measures on oracles

Part I. Approximate Inference

Introduction to Probabilistic Programming

Interpreter and handlers

Interpretation of the probabilistic constructs depends on the inference

- 1 inference = 1 `Handler` class
- Implement methods `sample`, `factor`, `infer`, ...
- Handlers are context managers

```
def model(data): [...]

with ImportanceSampling(num_particles=1000):
   dist = infer(model, data)

with MetropolisHastings(num_samples=1000):
   dist = infer(model, data)
```

Interpreter and handlers


```
class Handler: [...]

  def __init__(self, *args, **kwargs) → T: pass

  def sample(self, dist: Distribution[T]) → T: pass

  def assume(self, p: bool) → None: pass

  def factor(self, weight: float) → None: pass

  def observe(self, dist: Distribution[T], value: T) → None: pass

  def infer(self, model: Callable[P, T], data: P) → Distribution[T]: pass
```

Importance sampling

Importance sampling

Inference algorithm

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

$$p_i = \frac{W_i}{\sum_{1 \le i \le n} W_i}$$

Importance sampling

Inference algorithm

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

$$p_i = \frac{W_i}{\sum_{1 \leq i \leq n} W_i}$$

Conditioning

- `assume(p)`: sets the score to 0 if `p` is false
- `observe(d,v)`: multiply the score by the likelihood of `d` at `v` (density function of `d`)

Importance sampling

0

```
def model(data):
  ...
  x = sample ...
  ...
  factor ...
  ...
  return output
```

1

```
def model(data):
  ...
  x = sample ...
  ...
  factor ...
  ...
  return output
```

2

```
def model(data):
  ...
  x = sample ...
  ...
  factor ...
  ...
  return output
```

Importance sampling

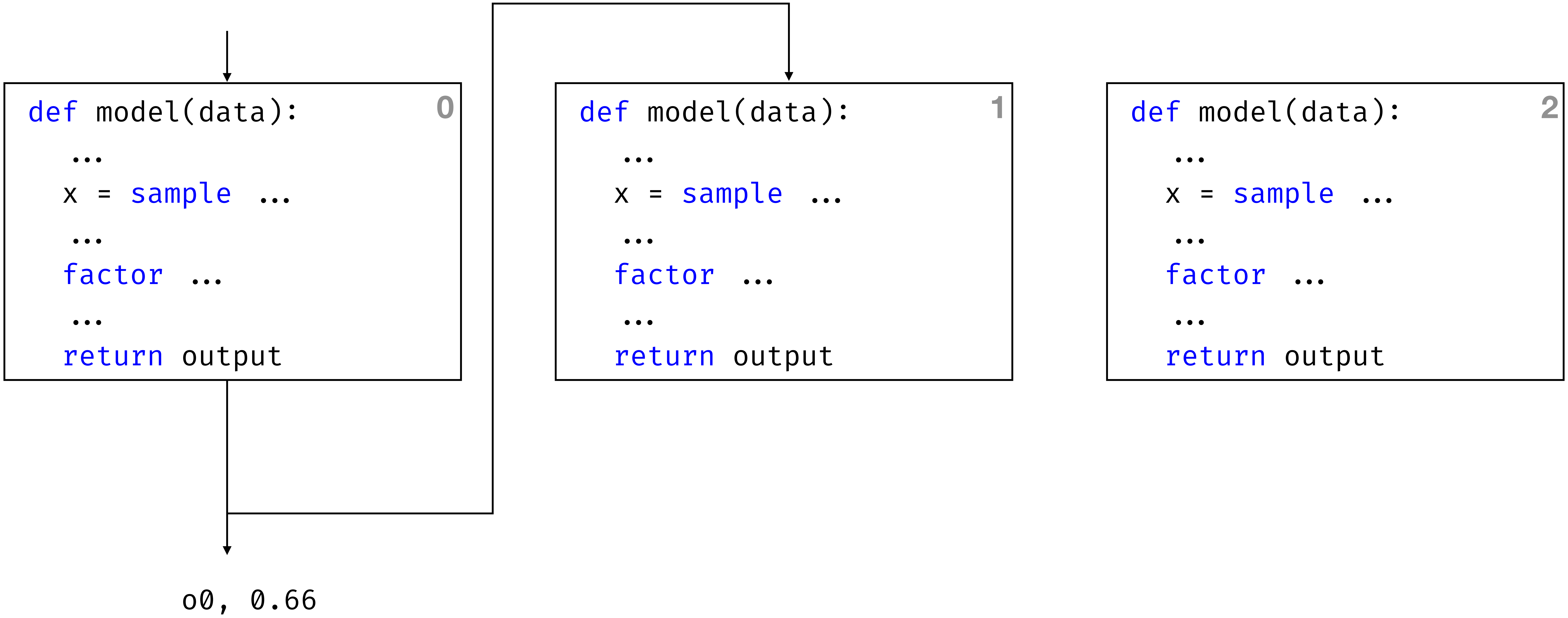
0
```
def model(data):
  ...
  x = sample ...
  ...
  factor ...
  ...
  return output
```

1
```
def model(data):
  ...
  x = sample ...
  ...
  factor ...
  ...
  return output
```

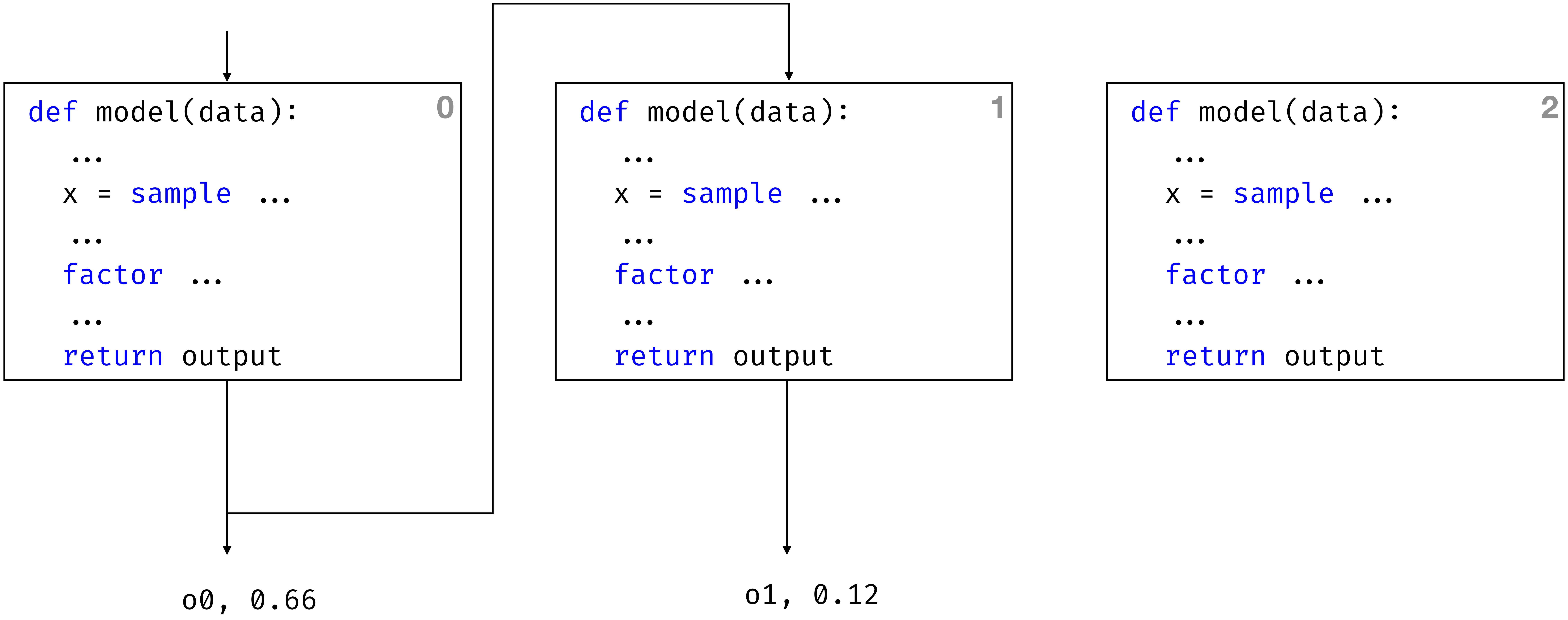
2
```
def model(data):
  ...
  x = sample ...
  ...
  factor ...
  ...
  return output
```

o0, 0.66

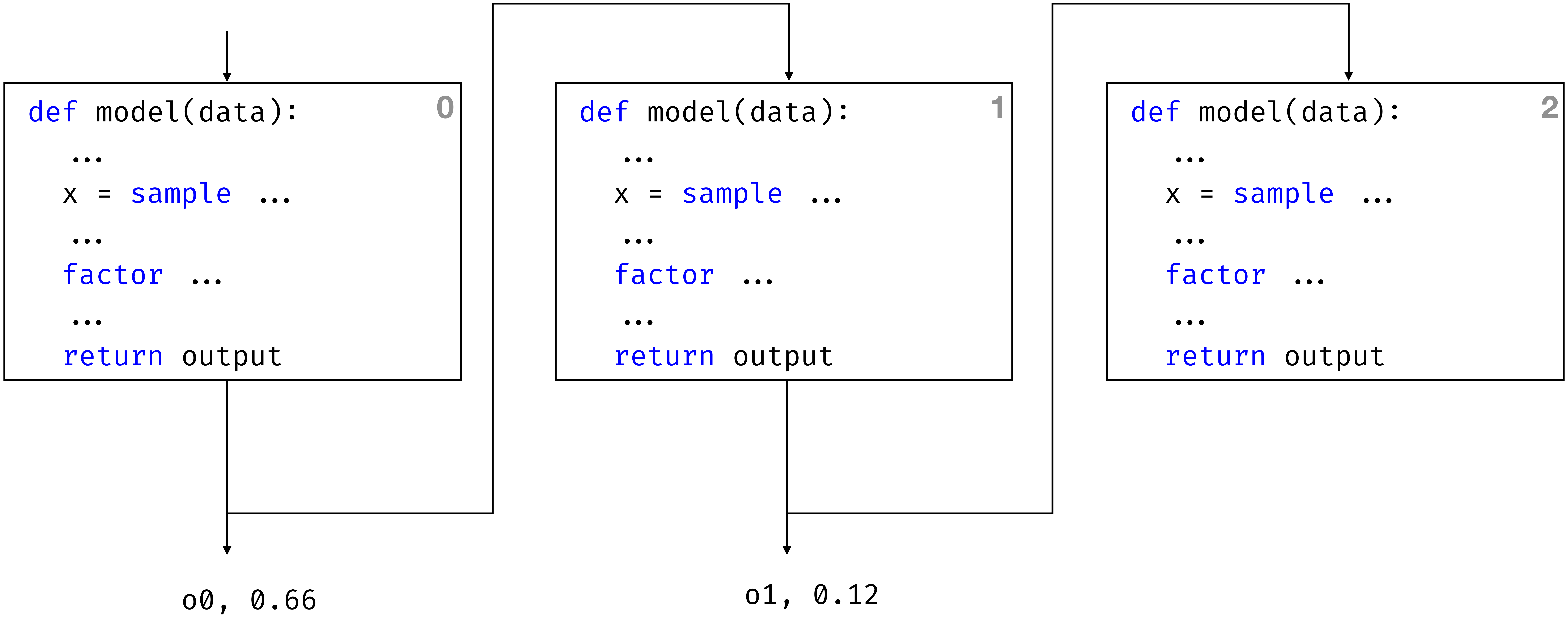
Importance sampling



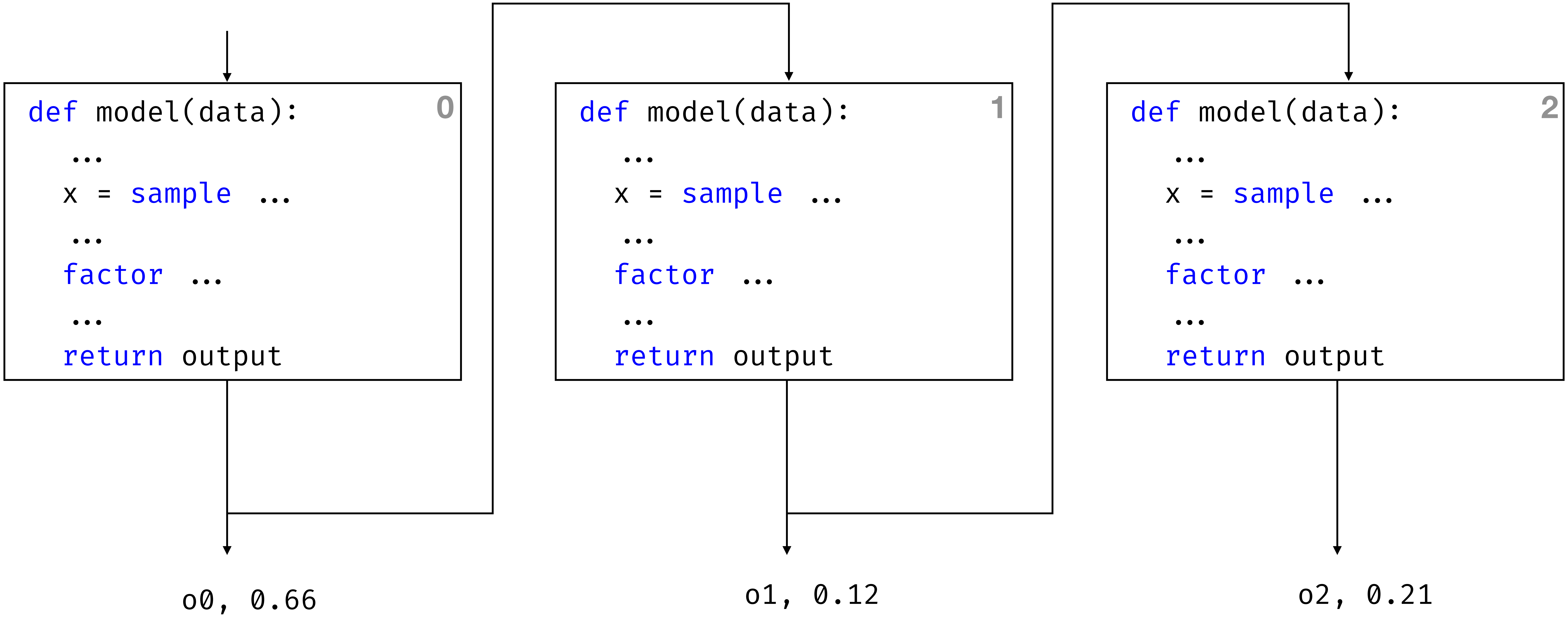
Importance sampling



Importance sampling



Importance sampling



Importance sampling

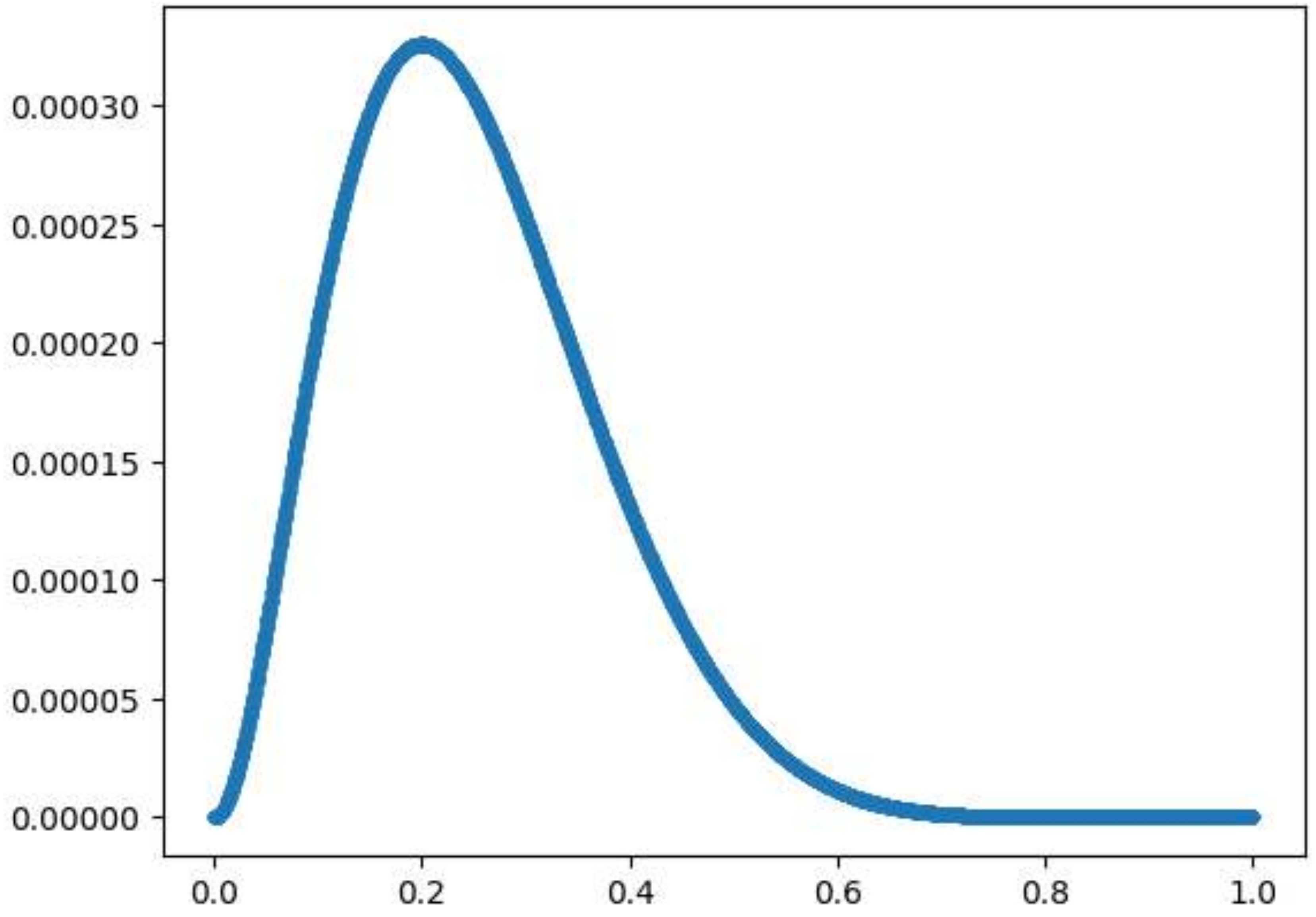

```
class Importance_sampling(Handler):
  def __init__(self, num_particles: int = 1000) → None:
    self.num_particles = num_particles
    self.score: float = 0   # current score

  def sample(self, dist: Distribution[T]) → T:
    return dist.sample()   # draw sample

  def factor(self, weight: float) → None:
    self.score += weight   # update the score (log scale)

  def infer(self, model: Callable[P, T], data: P) → Distribution[T]:
    samples: List[Tuple[T, float]] = []
    for _ in range(self.num_particles):   # run num_particles executions
      self.score = 0   # reset the score
      samples.append((model(data), self.score)) # store value and score
    return Categorical(samples)
```

Example: bias of a coin

```python
from mu_ppl import *

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=10000):
  dist = infer(coin, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1])
  viz(dist)
```

Example: bias of a coin

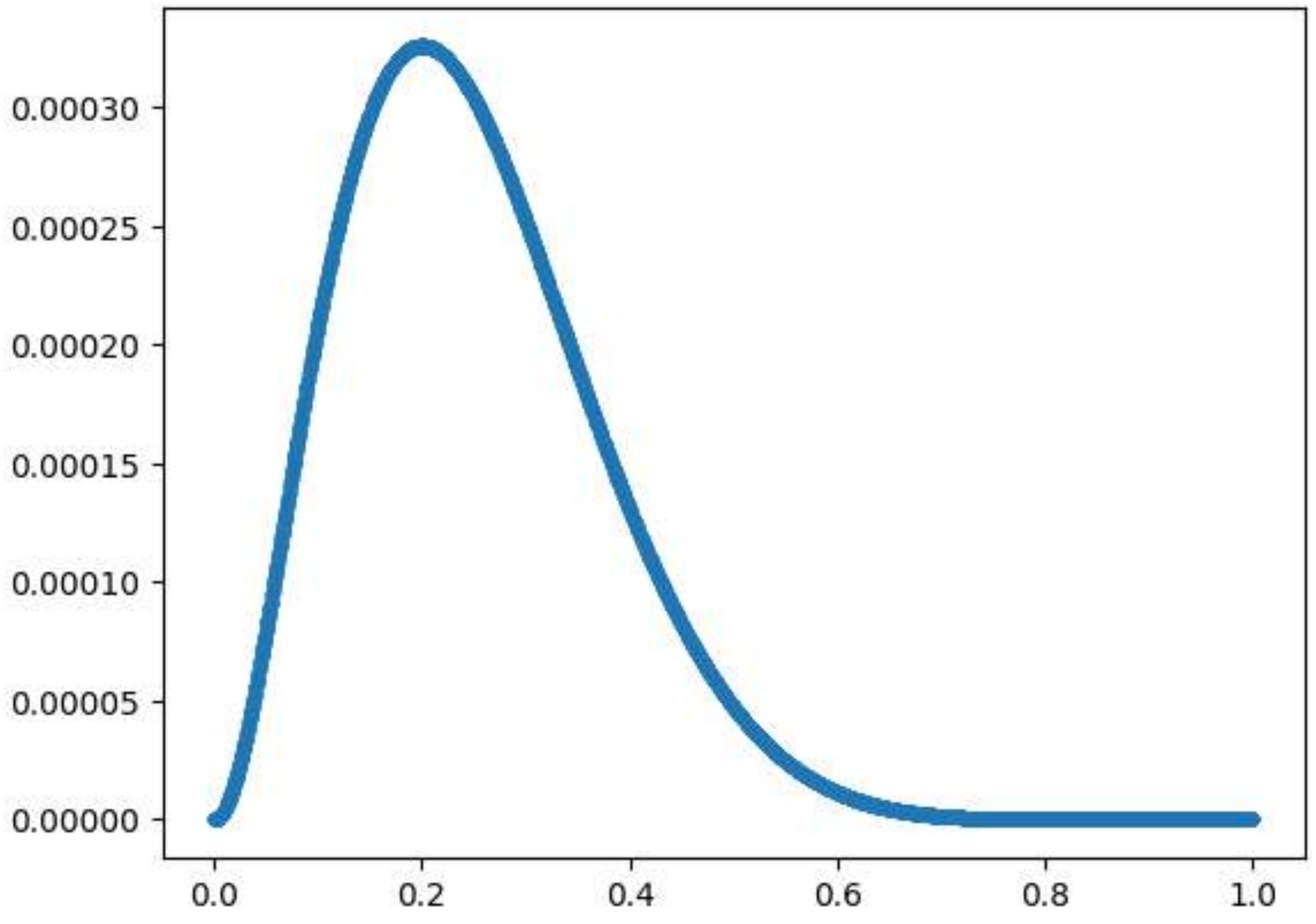
```
from mu_ppl import *

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=10000):
  dist = infer(coin, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1])
  viz(dist)
```



Example: bias of a coin

```
from mu_ppl import *

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=10000):
  dist = infer(coin, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1])
  viz(dist)
```

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



The curse of dimensionality

Problem becomes harder as the dimension increases

Basic inference: importance sampling

- Performances decrease exponentially when the dimension increases
- Only use for low-dimension models

How to mitigate this problem?

- Make assumptions about the posterior distributions
- Break the problem into simpler, smaller problems

Sven Sachsalber, *Looking for a needle in the Haystack*, 2014

The curse of dimensionality

Problem becomes harder as the dimension increases

Basic inference: importance sampling

- Performances decrease exponentially when the dimension increases
- Only use for low-dimension models

How to mitigate this problem?

- Make assumptions about the posterior distributions
- Break the problem into simpler, smaller problems

17h45mn

Sven Sachsalber, *Looking for a needle in the Haystack*, 2014

Markov Chain Monte Carlo (MCMC)

Main idea

- Create a Markov chain over executions that converges to the posterior distribution
- Iterate the process until convergence
- Then, each iteration generates a valid sample
- Accumulate samples to approximate the posterior distribution

Pros

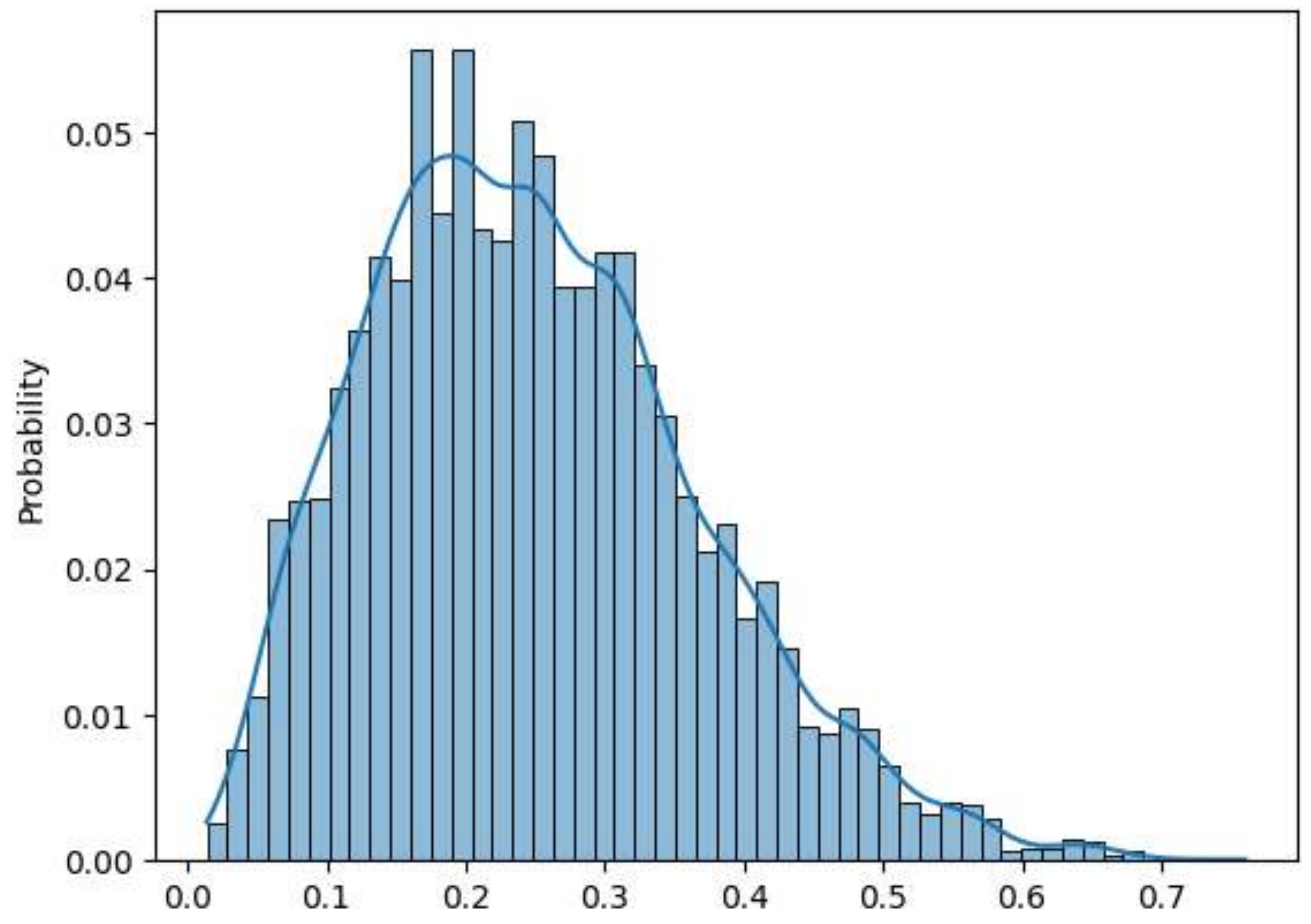
- Faster convergence
- Better results for high-dimensional models
- Advanced state-of-the-art optimizations (e.g., HMC, NUTS).

Cons

- Convergences?
- Traps: multimodal, funnel
- Samples correlation

Example: bias of a coin

```
from mu_ppl import *

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

with MetropolisHastings(num_particles=10000):
  dist = infer(coin, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1])
  viz(dist)
```



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

Metropolis Hastings

Metropolis Hastings

Trace

- X: set of random variables (`sample`)
- $P(X)$: prior distribution of X
- A trace characterize one posible execution
- W: score of the execution (same as importance sampling)

Metropolis Hastings

Trace

- X: set of random variables (`sample`)
- $P(X)$: prior distribution of X
- A trace characterize one posible execution
- W: score of the execution (same as importance sampling)

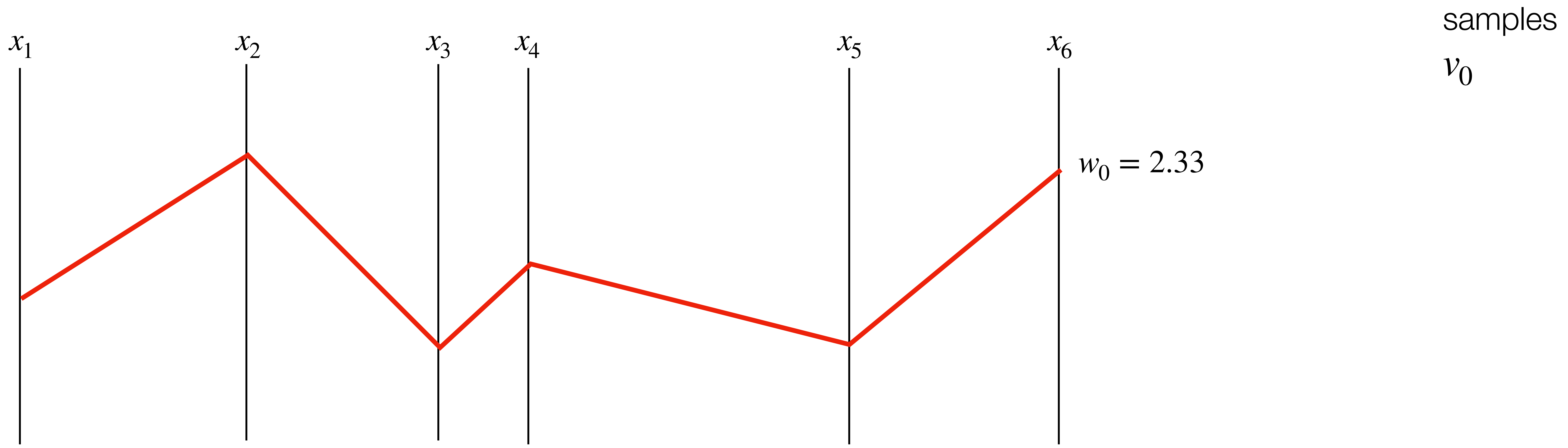
Metropolis-Hastings algorithm

- Initialization: draw X_0 at random to get a pair (v_0, W_0).
- At each step:
  1. Draw a *candidate* $X' \sim Q(X' \mid X_i)$ to get (v', W')
  2. Acceptance rate: $\alpha = \dfrac{P(X')\, W'\, Q(X_i \mid X')}{P(X_i)\, W_i\, Q(X' \mid X_i)}$.
  3. Draw $u \sim U(0,1)$.
  4. If $u \leq \alpha$ (*accept*) $\begin{cases} X_{i+1} & = & X' \\ v_{i+1} & = & v' \\ W_{i+1} & = & W' \end{cases}$ else (*reject*) $\begin{cases} X_{i+1} & = & X_i \\ v_{i+1} & = & v_i \\ W_{i+1} & = & W_i \end{cases}$

Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

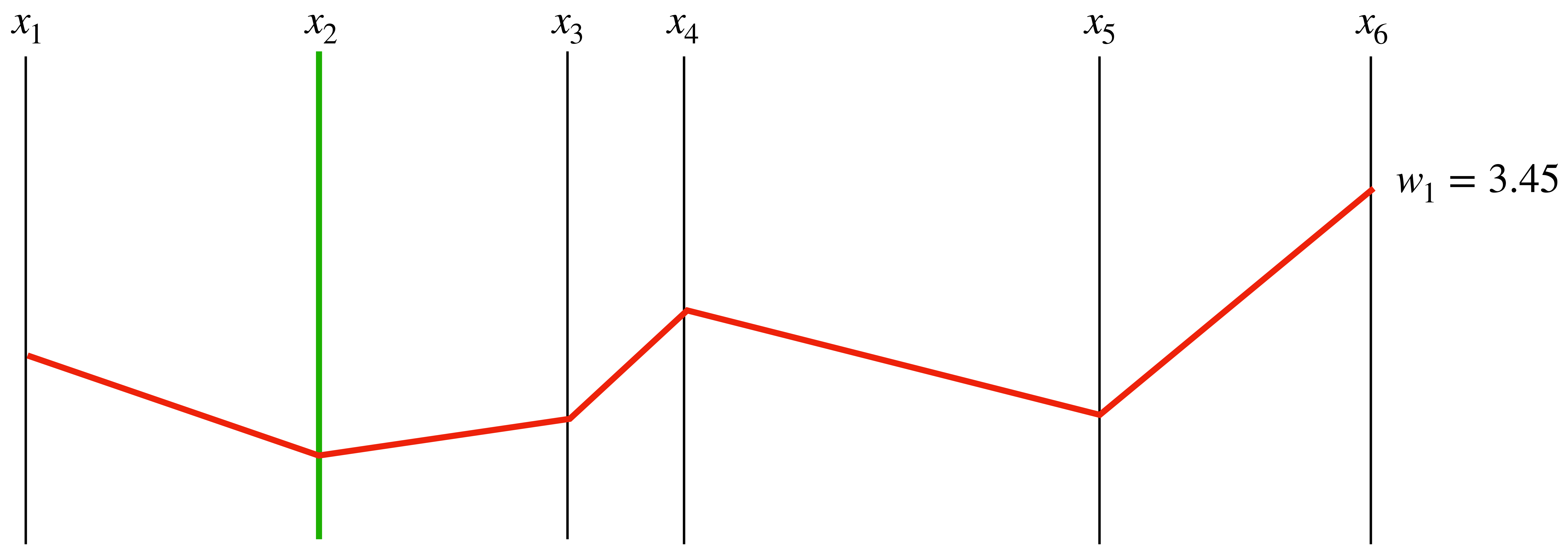
- Choose one random variable x_{regen} to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable x_{regen} to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



samples

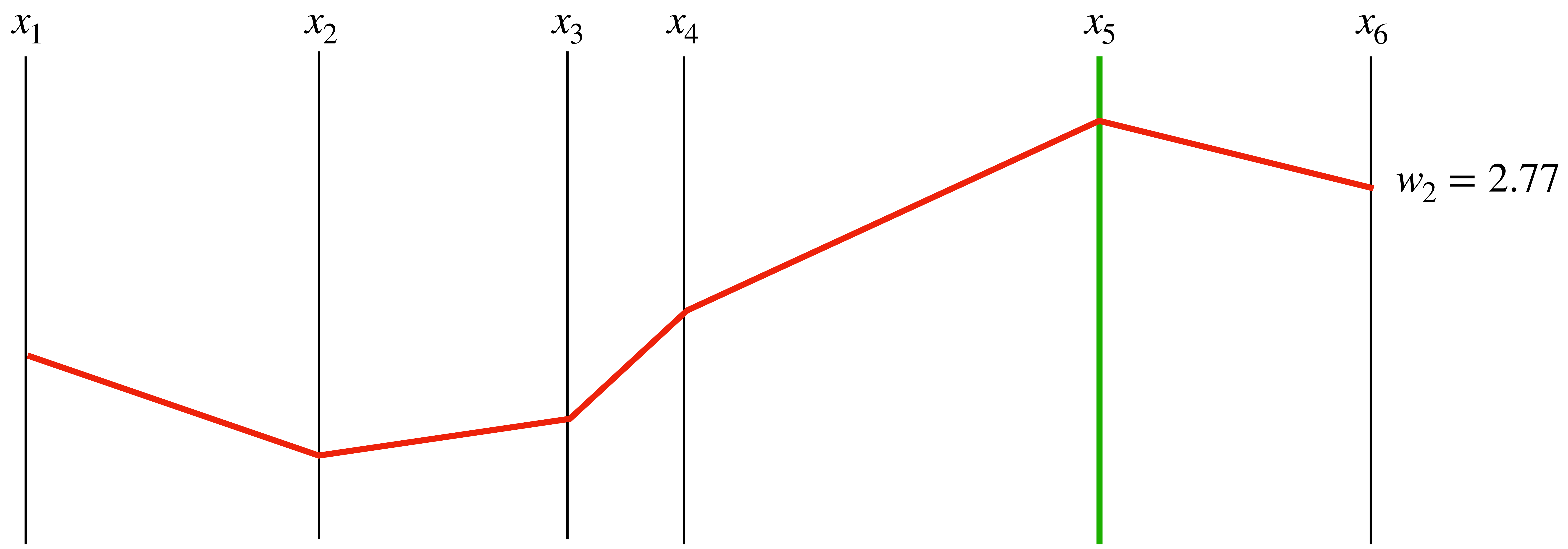
v_0

v_1

Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable x_{regen} to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



samples

v_0

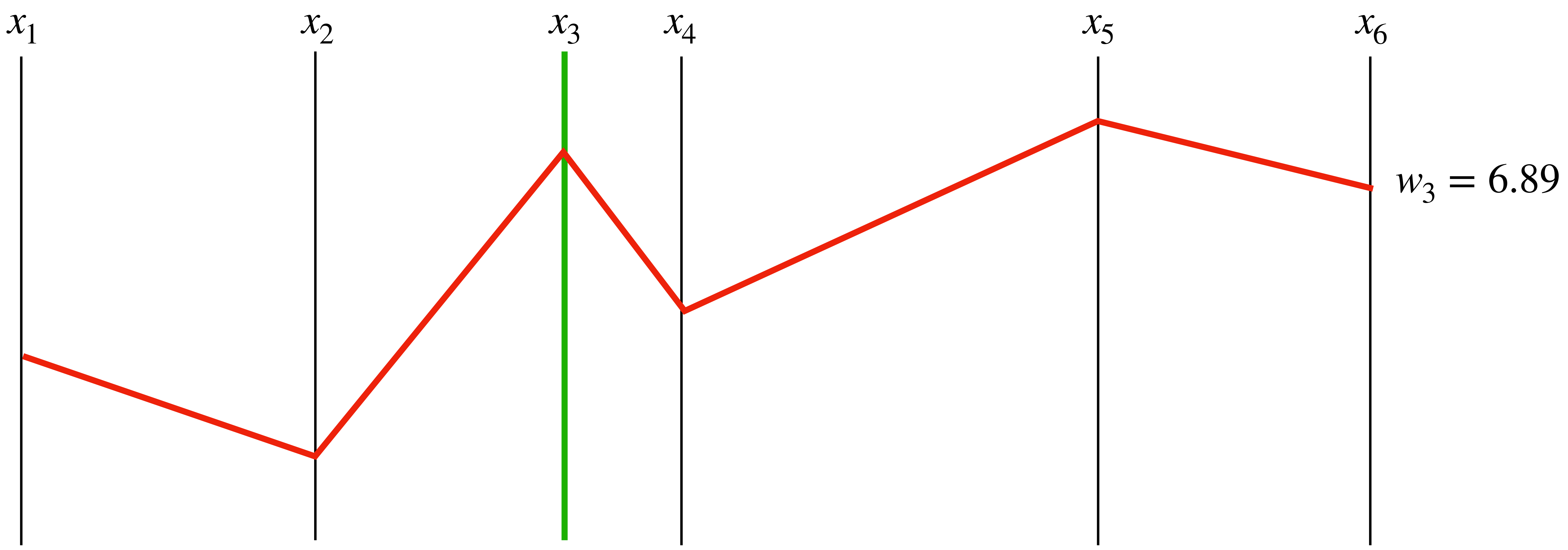
v_1

v_1

Single-site Metropolis Hastings

Reuse most of the previous trace (i.e., sampled values)

- Choose one random variable x_{regen} to resample to obtain a new execution
- Accept the trace with probability α
- Otherwise use the previous trace



samples

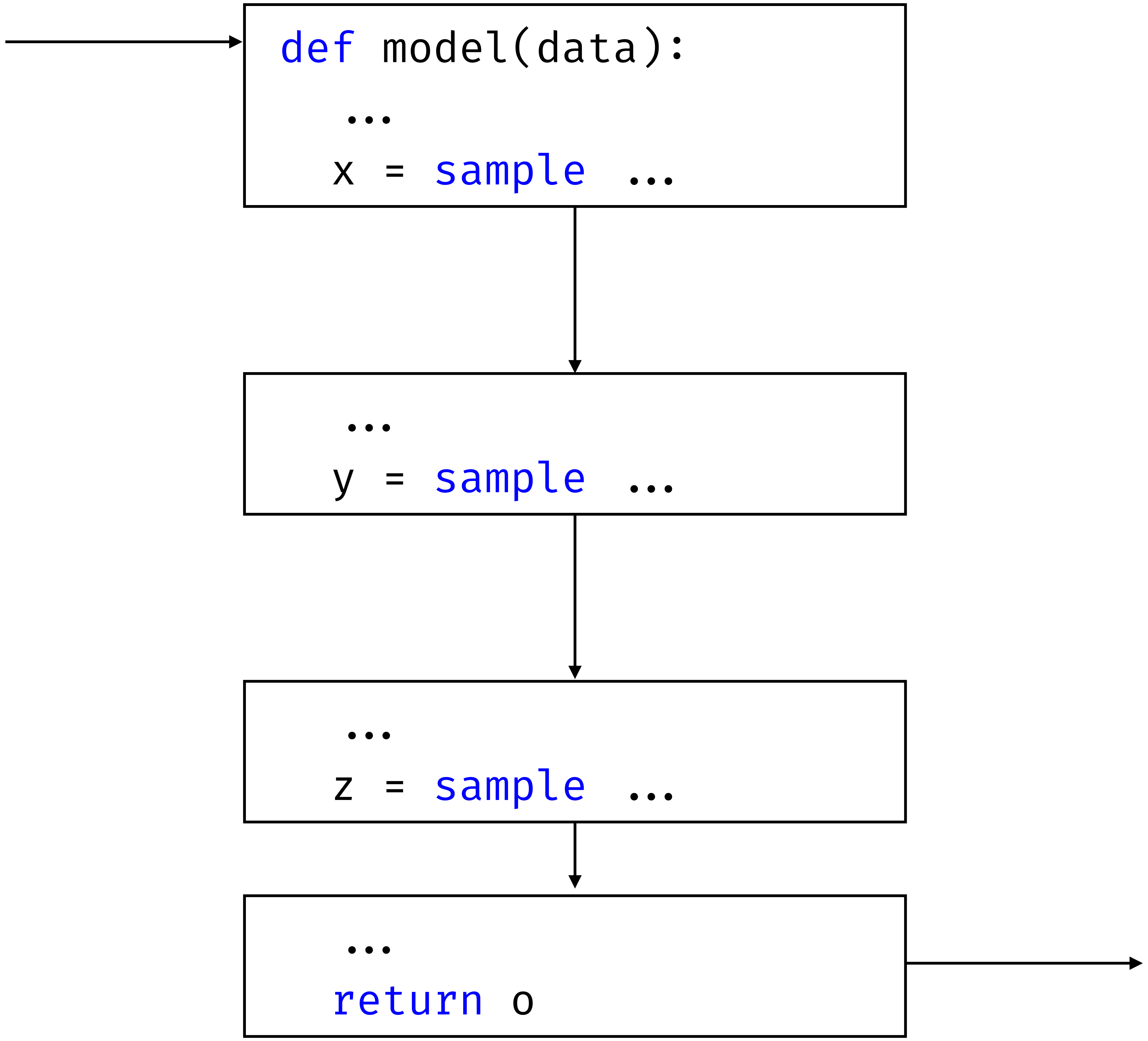
v_0

v_1

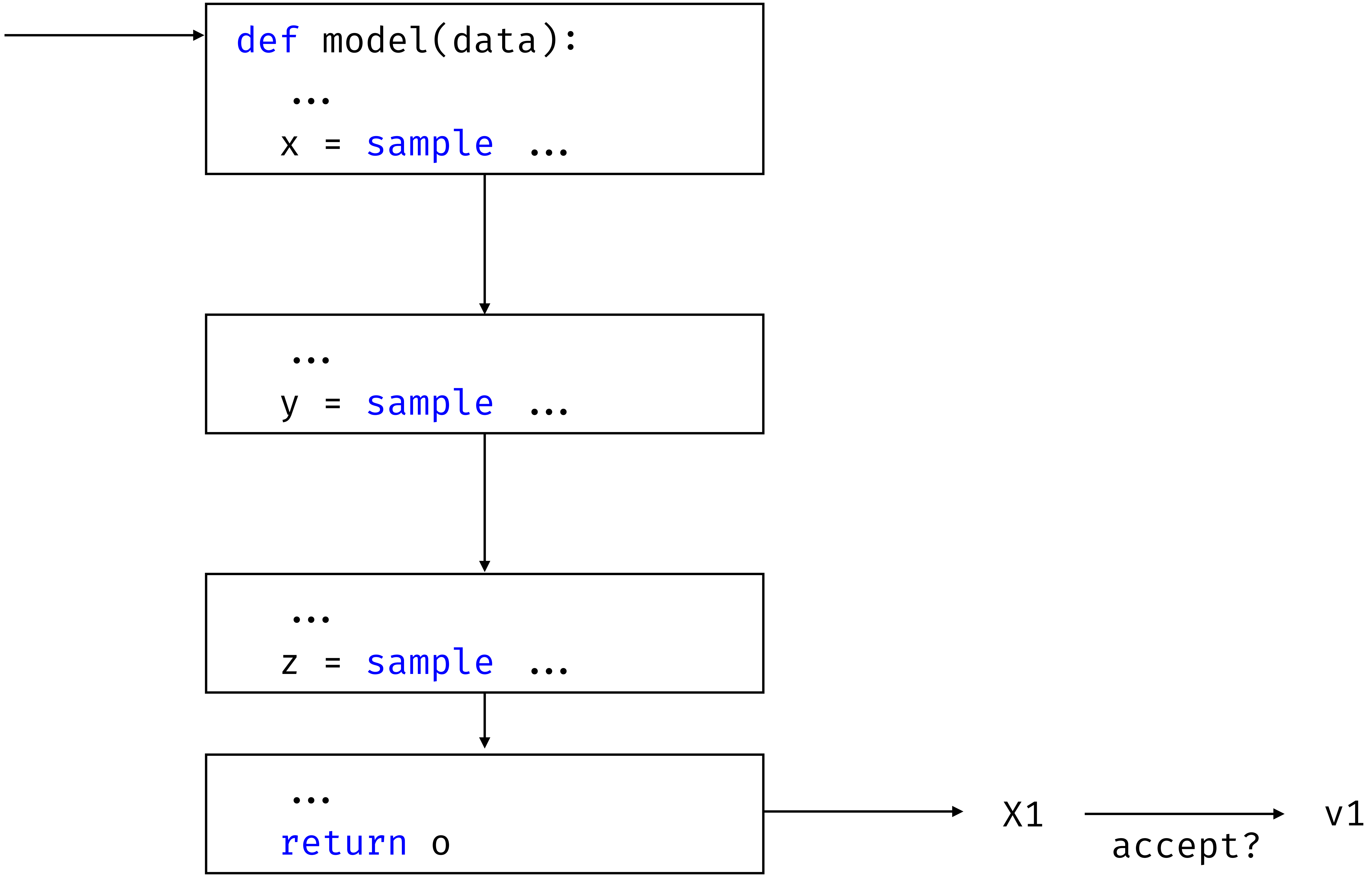
v_1

v_3

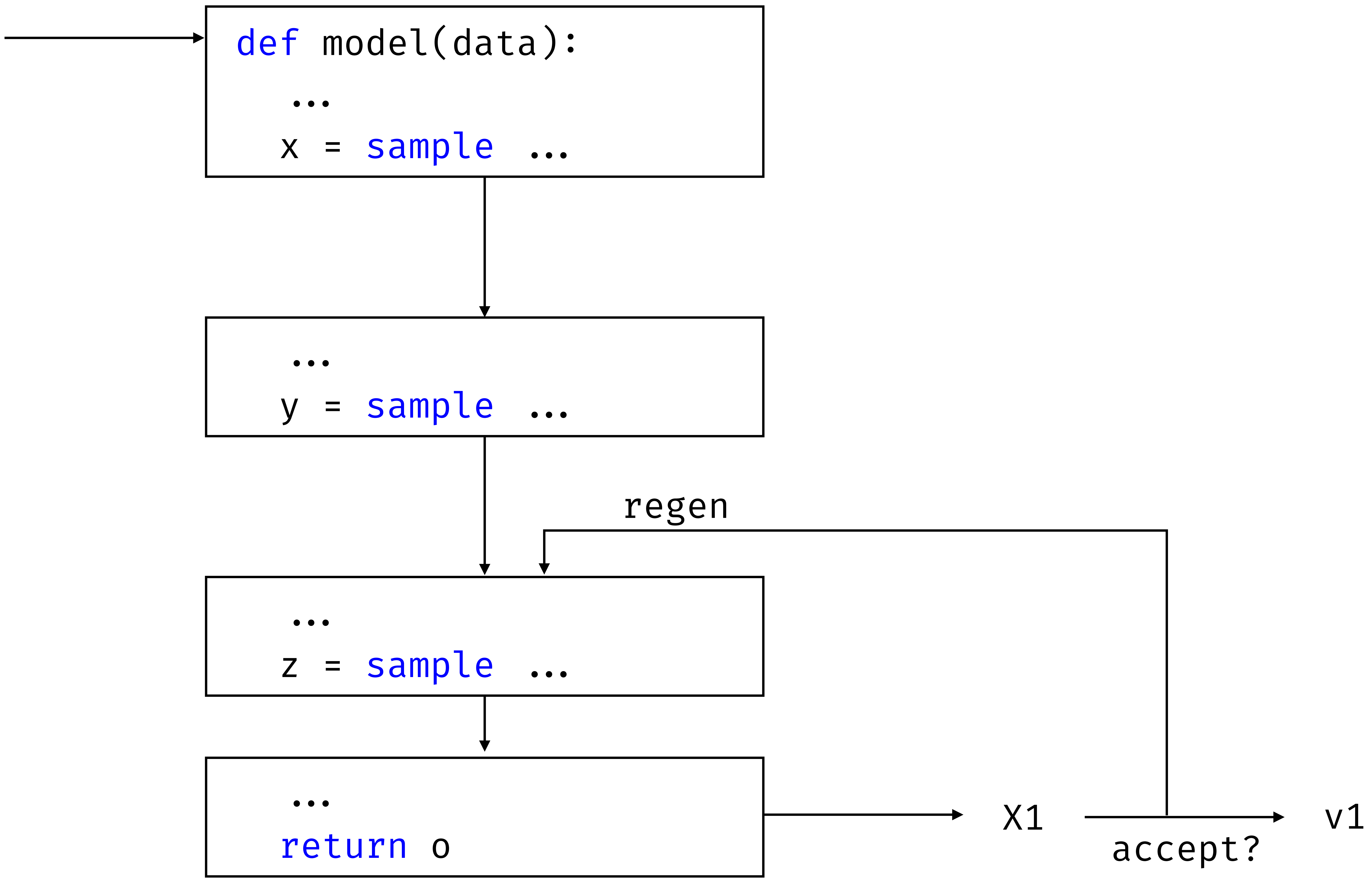
Single-site Metropolis Hastings



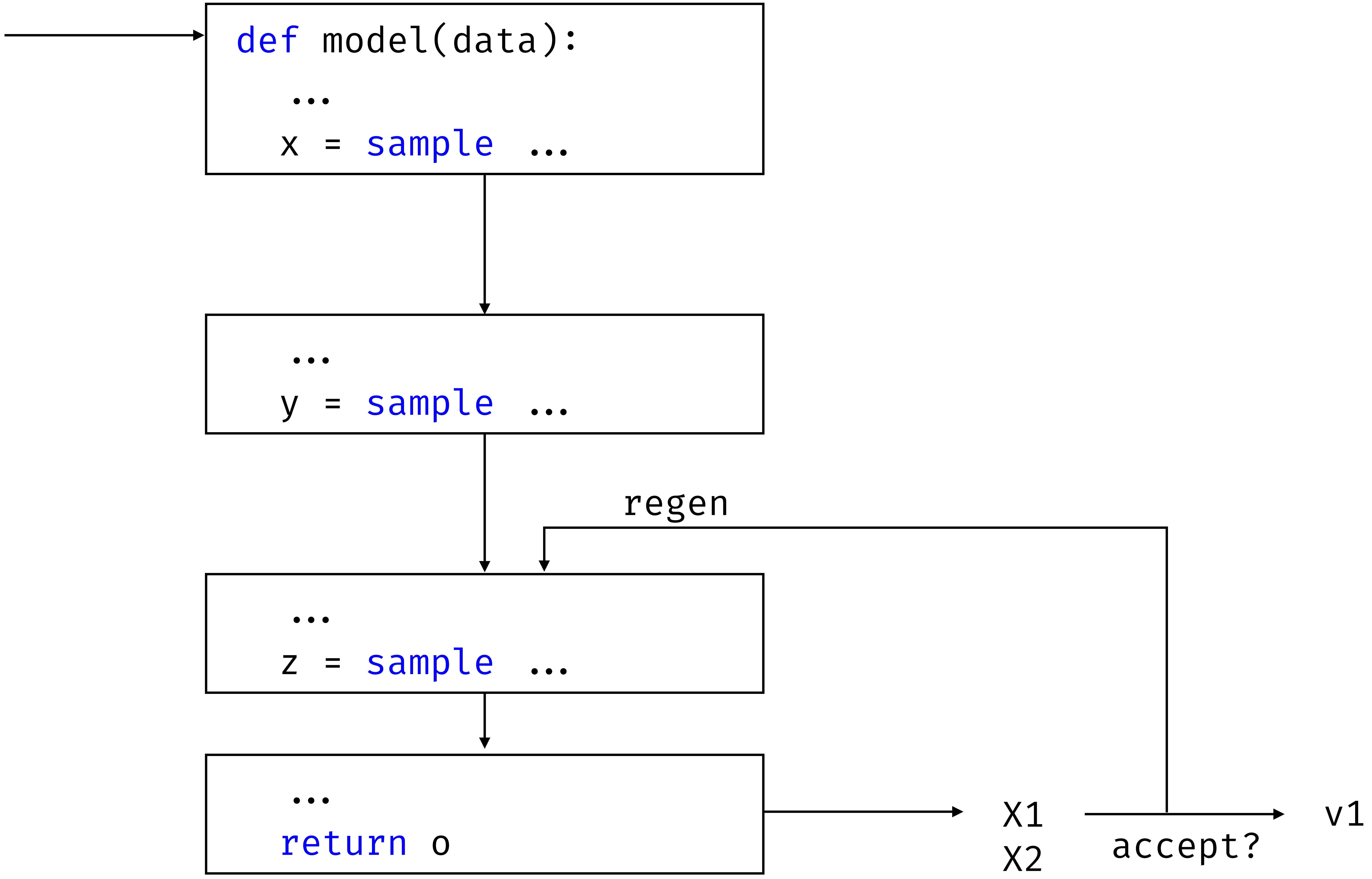
Single-site Metropolis Hastings



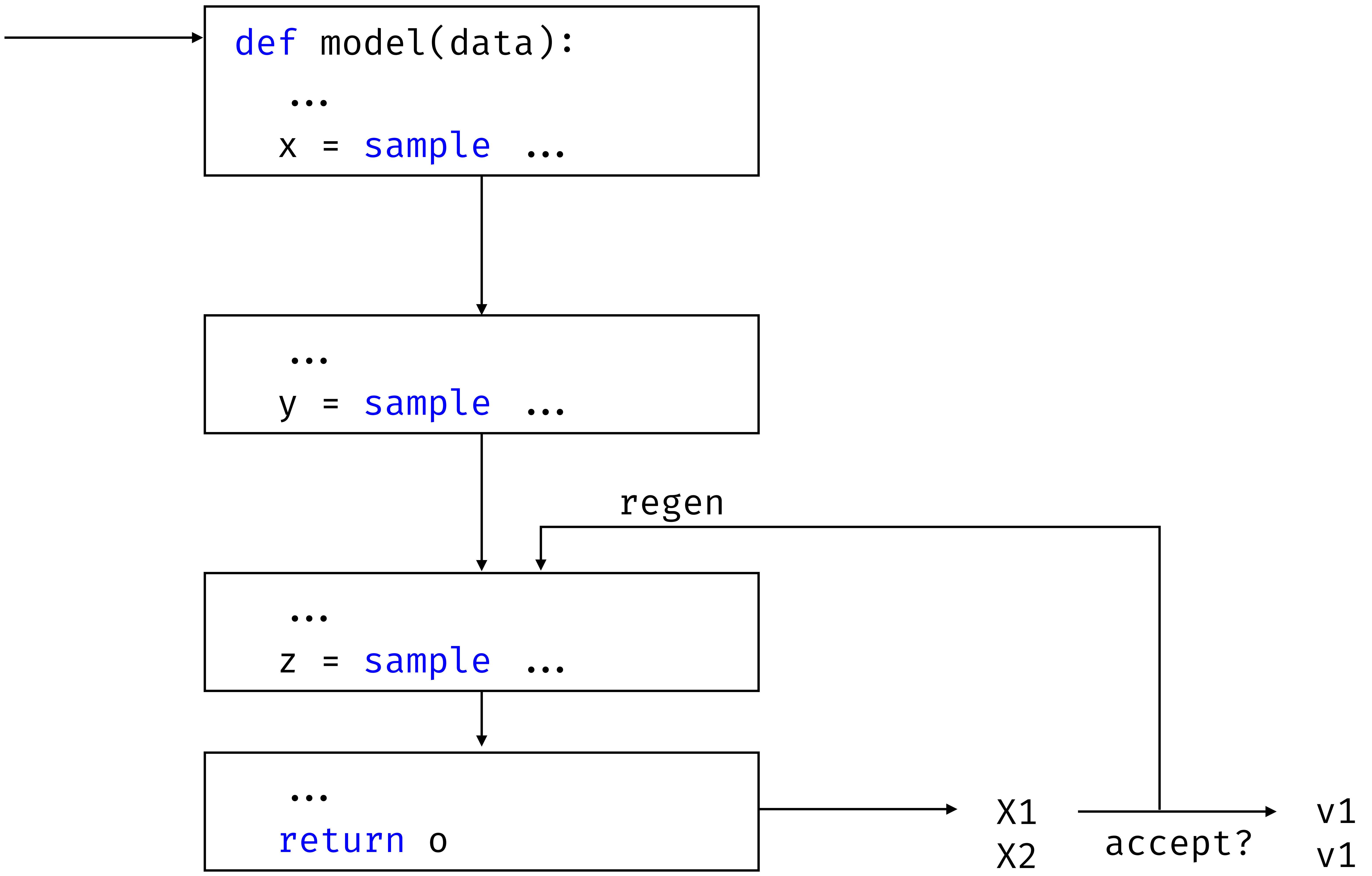
Single-site Metropolis Hastings



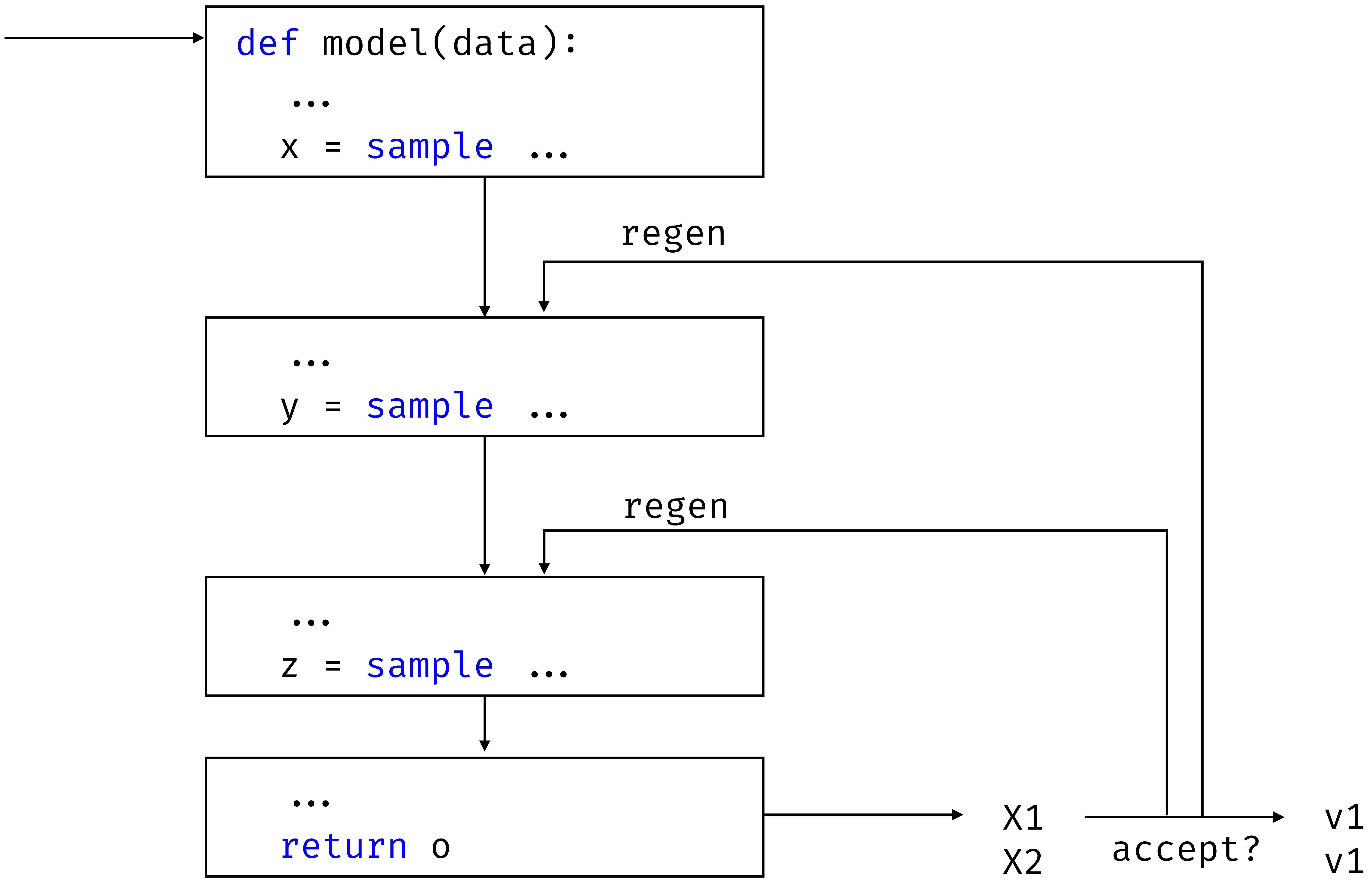
Single-site Metropolis Hastings



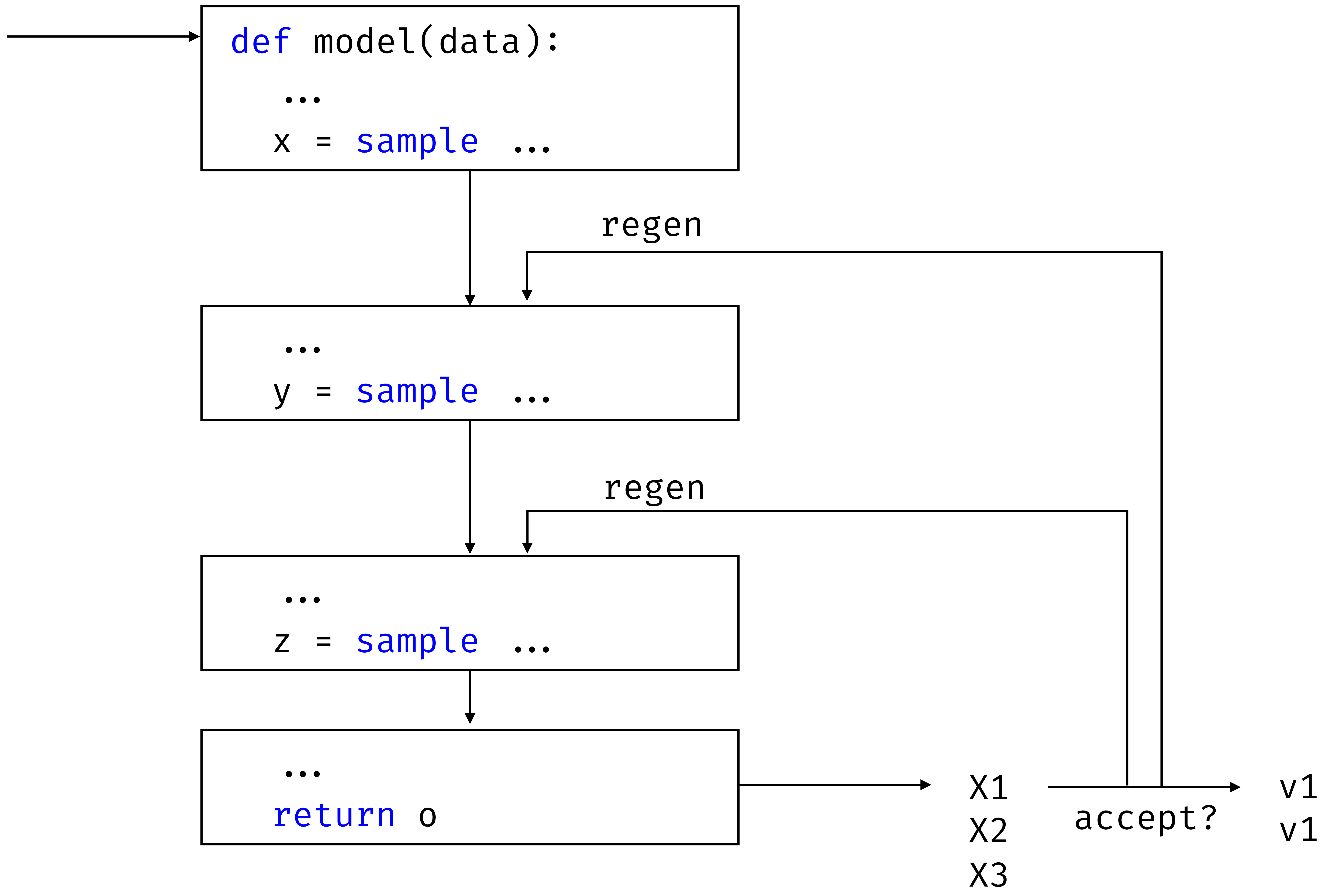
Single-site Metropolis Hastings



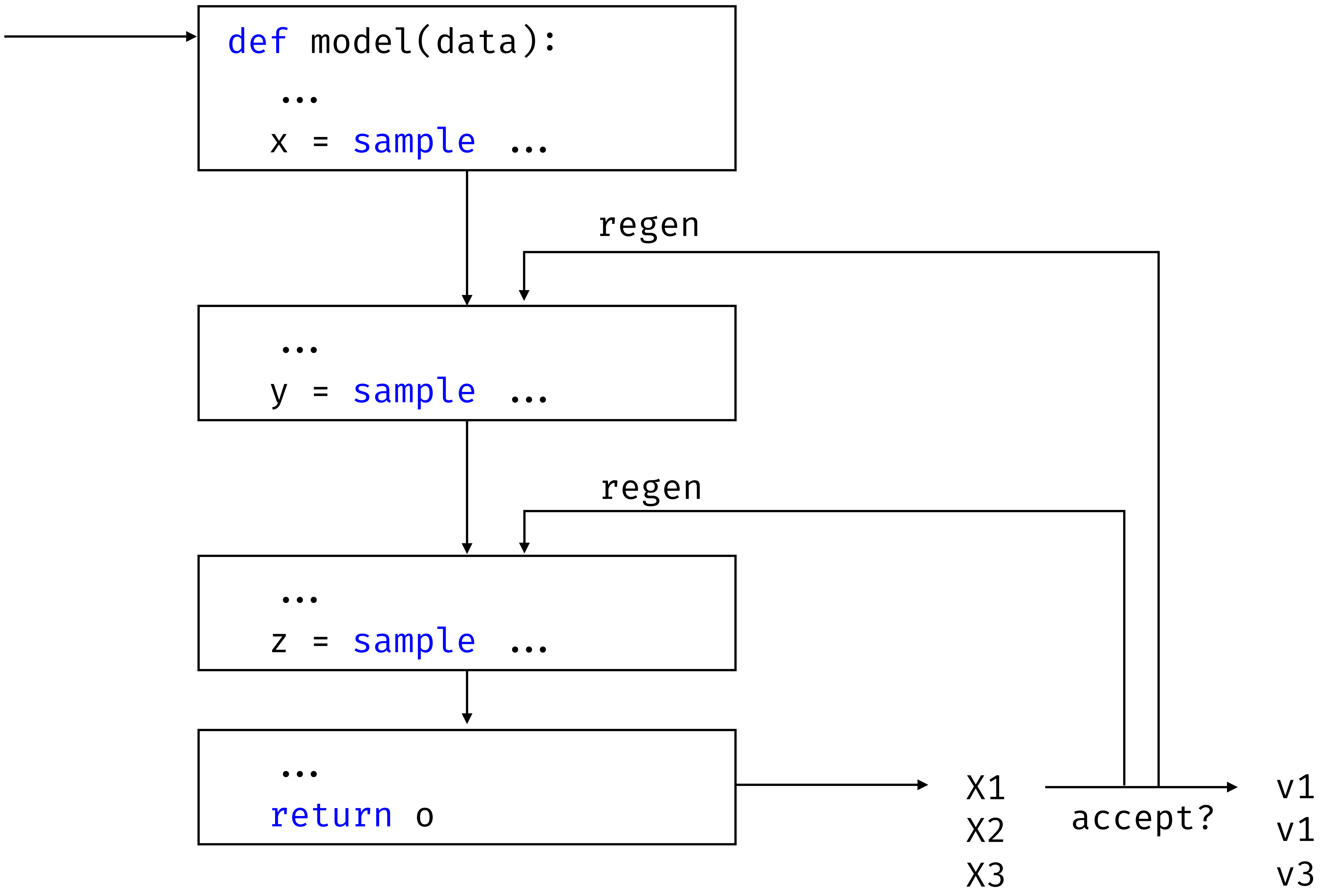
Single-site Metropolis Hastings



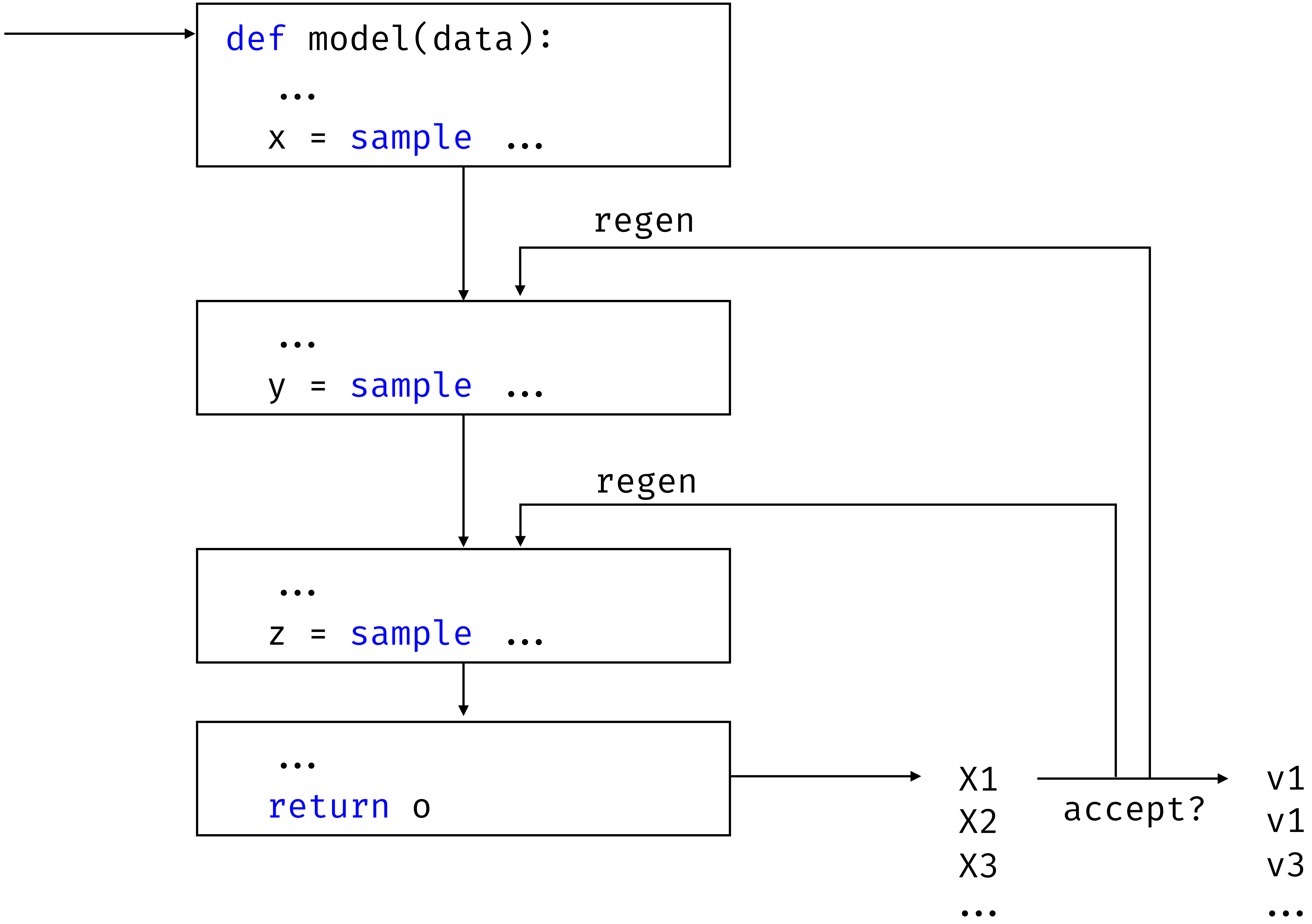
Single-site Metropolis Hastings



Single-site Metropolis Hastings



Single-site Metropolis Hastings



Single-site Metropolis Hastings: acceptation

Single-site Metropolis Hastings: acceptation

Notations

- For $x \in X$, $w(x)$: density of sample x (same as `observe`)
- $C = (X' \cap X - \{x_{\text{regen}}\})$: *cache*, i.e., reused variables between X and X'

$$P(X) = \prod_{x \in X} w(x)$$ *prior distribution*

$$Q(X' \mid X) = \frac{1}{|X|} \prod_{x \in (X' - C)} w'(x)$$ *choice of X' from X*

Single-site Metropolis Hastings: acceptation

Notations

- For $x \in X$, $w(x)$: density of sample x (same as `observe`)
- $C = (X' \cap X - \{x_{\text{regen}}\})$: *cache*, i.e., reused variables between X and X'

$$P(X) = \prod_{x \in X} w(x)$$ *prior distribution*

$$Q(X' \mid X) = \frac{1}{|X|} \prod_{x \in (X' - C)} w'(x)$$ *choice of X' from X*

$$\begin{aligned}
\alpha &= \frac{P(X')\, W'\, Q(X_i \mid X')}{P(X_i)\, W_i\, Q(X' \mid X_i)} \\
&= \frac{\prod_{x \in X'} w'(x)}{\prod_{x \in X_i} w(x)} \frac{W'}{W_i} \frac{|X_i| \prod_{x \in (X_i - C)} w(x)}{|X'| \prod_{x \in (X' - C)} w'(x)} \\
&= \frac{|X_i|}{|X'|} \frac{W'}{W_i} \frac{\prod_{x \in C} w'(x)}{\prod_{x \in C} w(x)}
\end{aligned}$$

Single-site Metropolis Hastings: acceptation

Notations

- For $x \in X$, $w(x)$: density of sample x (same as `observe`)
- $C = (X' \cap X - \{x_{\text{regen}}\})$: *cache*, i.e., reused variables between X and X'

$$P(X) = \prod_{x \in X} w(x)$$ *prior distribution*

$$Q(X' \mid X) = \frac{1}{|X|} \prod_{x \in (X'-C)} w'(x)$$ *choice of X' from X*

$$\alpha = \frac{P(X')\, W'\, Q(X_i \mid X')}{P(X_i)\, W_i\, Q(X' \mid X_i)}$$
$$= \frac{\prod_{x \in X'} w'(x)}{\prod_{x \in X_i} w(x)} \frac{W'}{W_i} \frac{|X_i| \prod_{x \in (X_i - C)} w(x)}{|X'| \prod_{x \in (X'-C)} w'(x)}$$
$$= \frac{|X_i|}{|X'|} \frac{W'}{W_i} \frac{\prod_{x \in C} w'(x)}{\prod_{x \in C} w(x)}$$

Reused variables are treated as observations

Single-site Metropolis Hastings

Rerun (part of) a trace

- Assign a unique name to each random variable `sample` (can be added by a compiler)
- `cache` : *name* $\to V$, reused samples
- `x_samples` : *name* $\to V$, samples for each random variable
- `x_scores` : *name* $\to \mathbb{R}^+$, corresponding score $w(x)$

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

Single-site Metropolis Hastings


```
class MetropolisHastings(ImportanceSampling):
  def __init__(self, num_samples: int) → None:
    self.num_samples = num_samples
    self.score: float = 0 # current score
    self.x_samples: Dict[str, Any] = {}   # samples store
    self.x_scores: Dict[str, float] = {}   # X scores
    self.cache: Dict[str, Any] = {}   # sample cache to be reused

  def sample(self, dist: Distribution[T], name: str) → T:
    try:   # reuse if possible
      v = self.cache[name]
    except KeyError:
      v = dist.sample()   # otherwise draw a sample
    self.x_samples[name] = v   # store the sample
    self.x_scores[name] = dist.log_prob(v)   # store the score
    return v
```

Single-site Metropolis Hastings


```
def mh(self, p_state):
  p_score, cache, p_x_scores = p_state
  alpha = np.log(len(p_x_scores)) - np.log(len(self.x_scores))
  alpha += self.score - p_score
  for x in self.cache:
    alpha += self.x_scores[x]
    alpha -= p_x_scores[x]
  return np.exp(alpha)
```

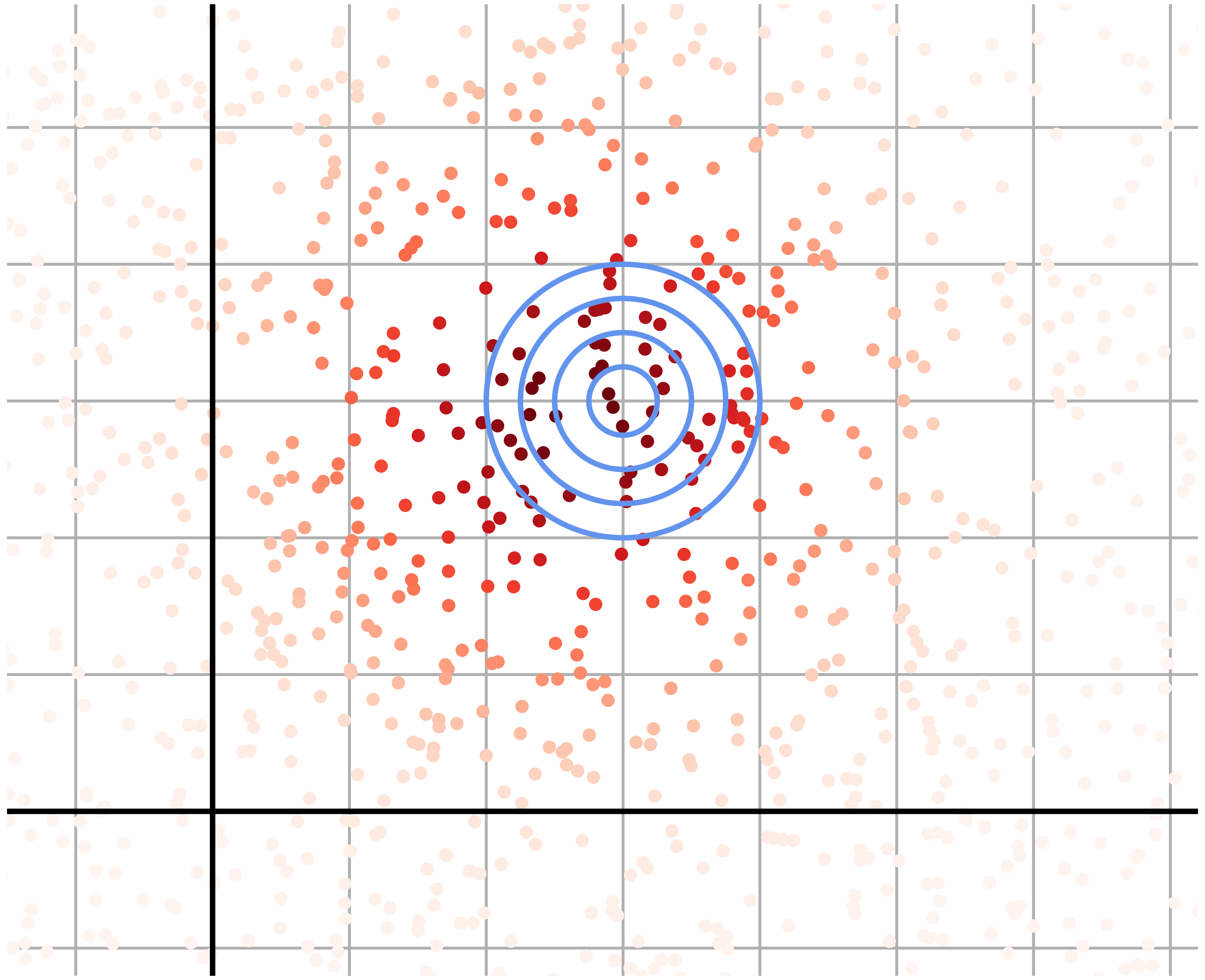
$$\alpha = \frac{|X_i|}{|X'|} \frac{W'}{W_i} \frac{\prod_{x \in C} w'(x)}{\prod_{x \in C} w(x)}$$

Single-site Metropolis Hastings


```
def infer(self, model: Callable[P, T], data: P) → Empirical[T]:
  samples: List[T] = []
  new_value = model(data)   # generate first trace
  for _ in range(self.num_samples):
    p_state = self.score, self.x_samples, self.x_scores   # store previous state
    p_value = new_value   # store current value
    regen = np.random.choice([n for n in self.x_samples])
    self.cache = deepcopy(self.x_samples)   # use samples as next cache
    del self.cache[regen]   # force regen to be resampled
    self.score, self.x_samples, self.x_scores = 0, {}, {}   # reset the state
    new_value = model(data)   # generate a new trace
    alpha = self.mh(p_state)
    u = np.random.random()
    if not (u < alpha):   # reject
      self.score, self.x_samples, self.scores = p_state   # rollback
      new_value = p_value
    samples.append(new_value)
  return Empirical(samples)
```

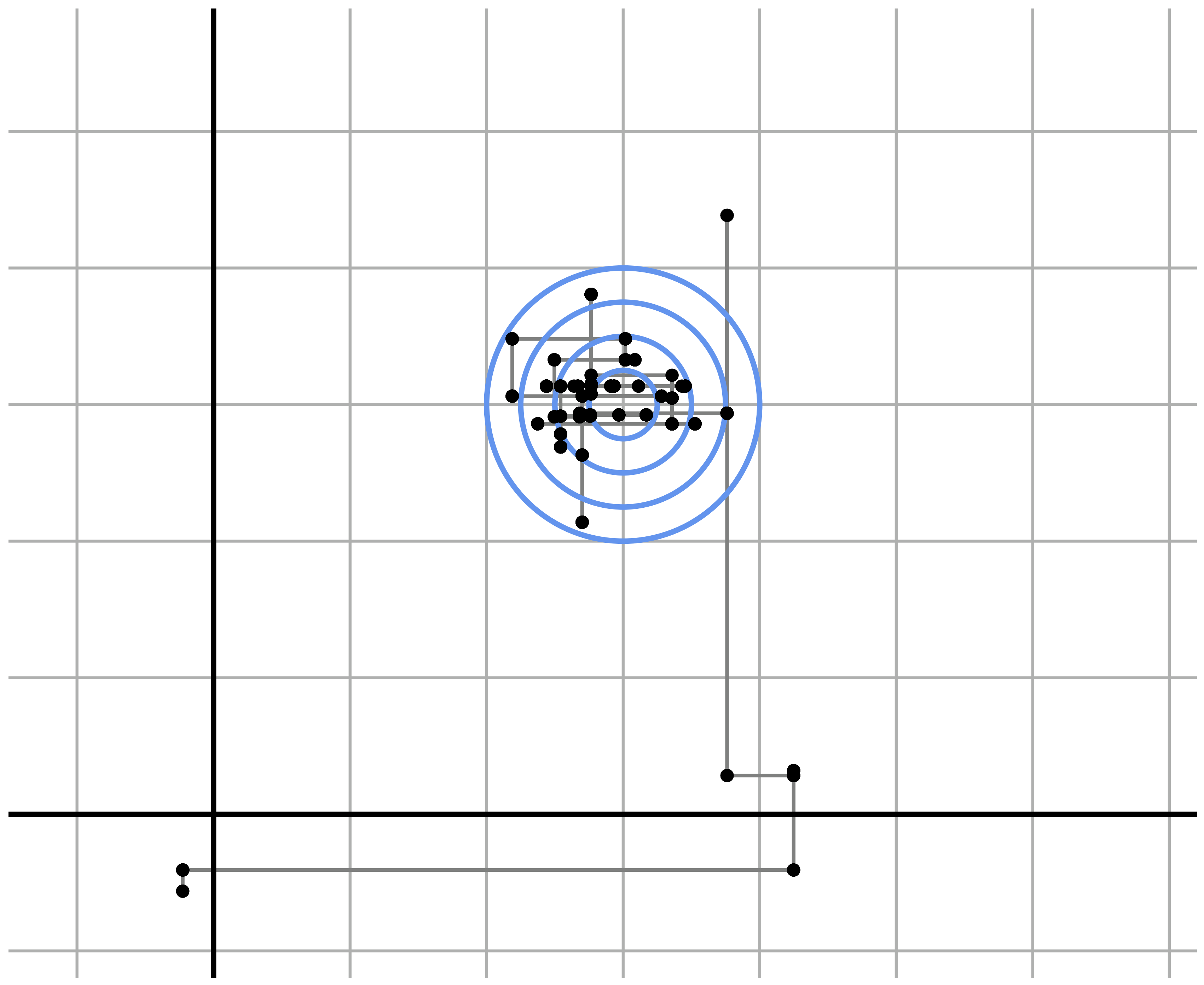
Example: Noisy position

```
def gauss(obs: List[Tuple[float, float]]) → Tuple[float, float]:
  x = sample(Gaussian(0, 10), name="x")
  y = sample(Gaussian(0, 10), name="y")
  for (xo, yo) in obs:
    observe(Gaussian(x, 1), xo)
    observe(Gaussian(y, 1), yo)
  return x, y

with ImportanceSampling(num_particles=10000):
  dist = infer(gauss, data)
```



Example: Noisy position

```
def gauss(obs: List[Tuple[float, float]]) → Tuple[float, float]:
  x = sample(Gaussian(0, 10), name="x")
  y = sample(Gaussian(0, 10), name="y")
  for (xo, yo) in obs:
    observe(Gaussian(x, 1), xo)
    observe(Gaussian(y, 1), yo)
  return x, y

with MetropolisHastings(num_samples=1000):
  dist = infer(gauss, data)
```



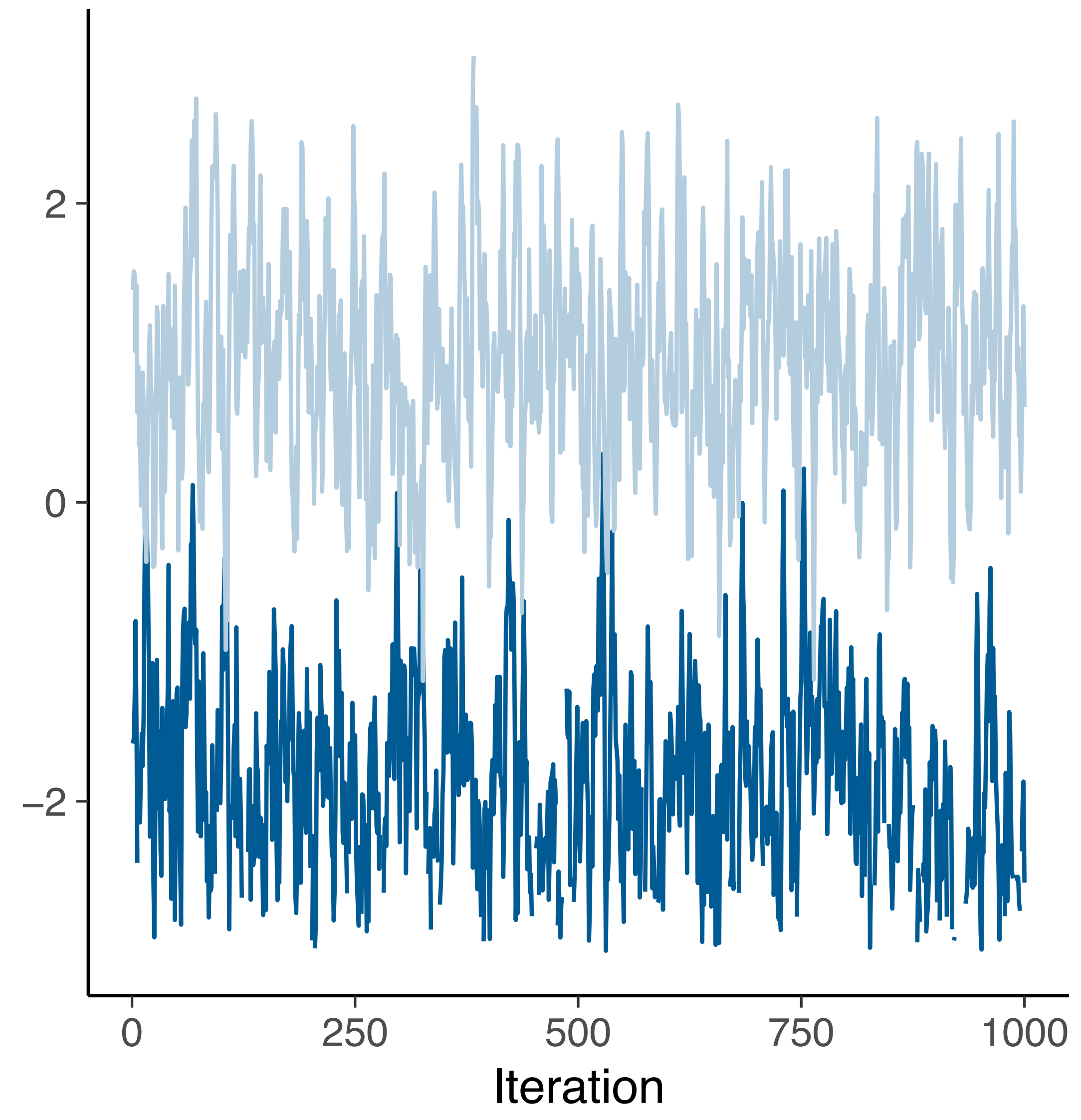
Limitations

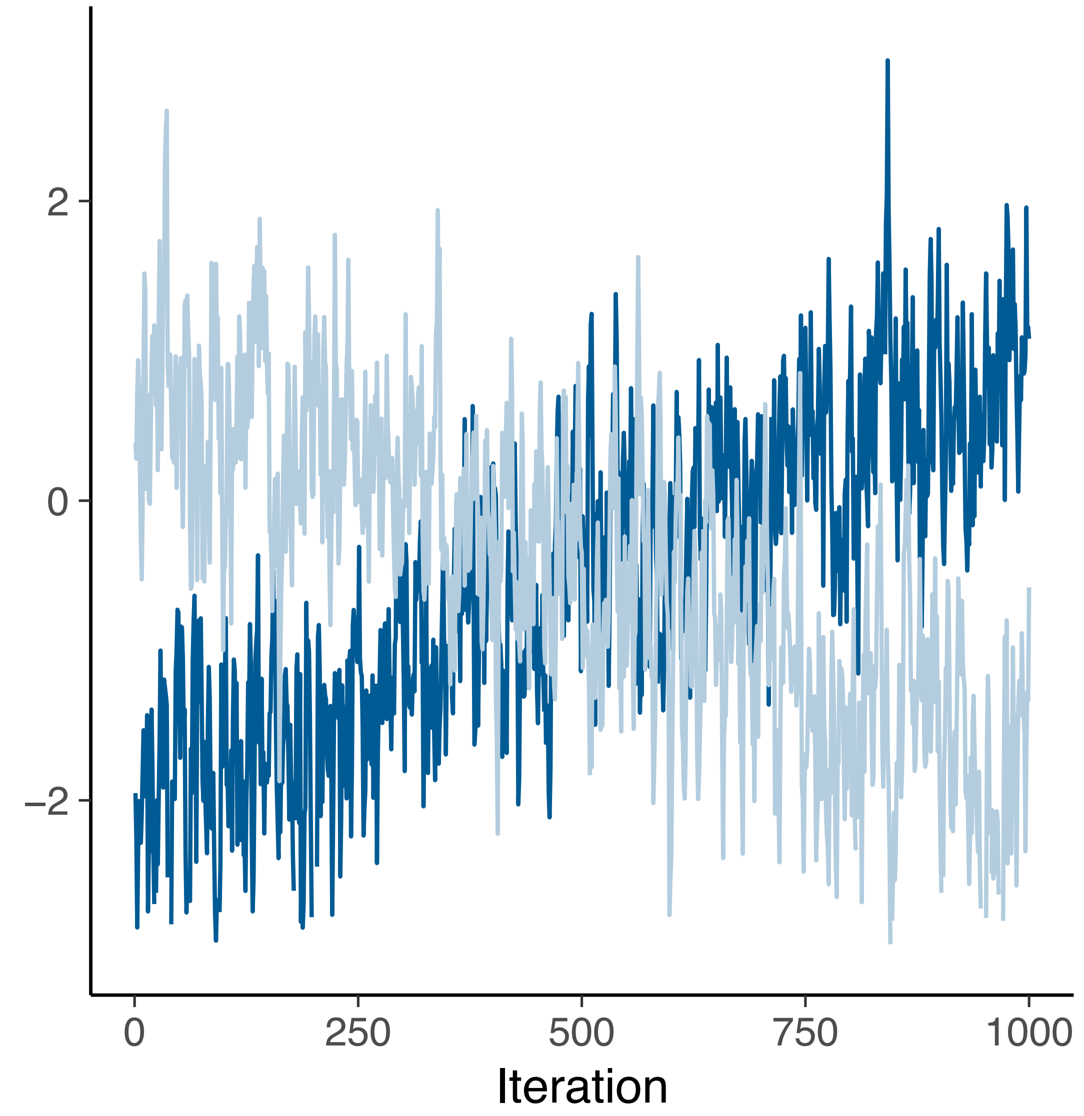
Convergence: theoretical conditions are complex

- Check experimentally: trace plot, R-hat (multi-chains)
- Solution: warmup, change initial conditions, reparameterization, ...

Sample correlation

- Diagnostic tools ESS (effective sample size)
- Solution: thinning, (keep one sample every n)





images from Vehtari et al. 2021

Limitations

Convergence: theoretical conditions are complex

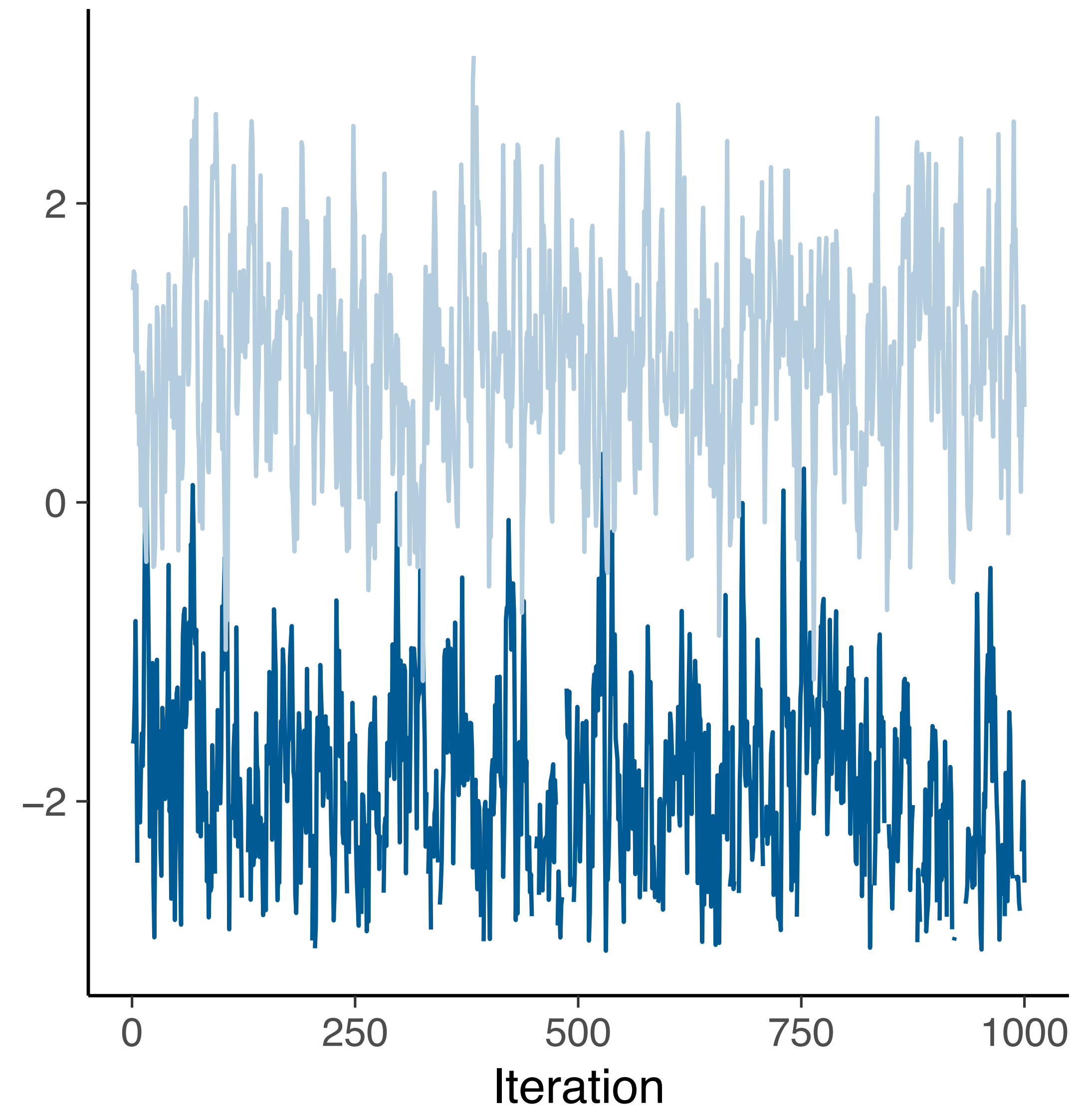
- Check experimentally: trace plot, R-hat (multi-chains)
- Solution: warmup, change initial conditions, reparameterization, ...

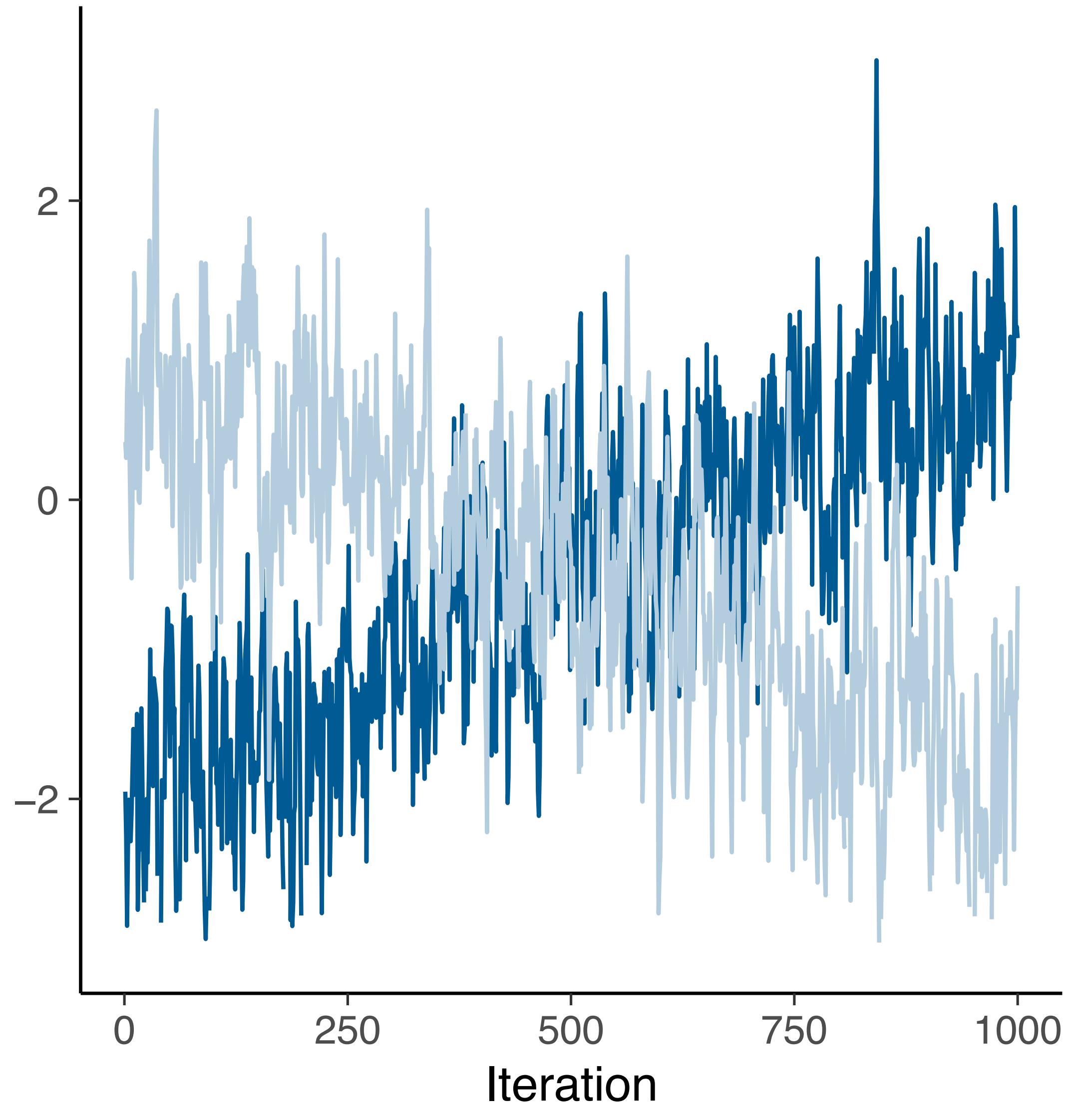
Sample correlation

- Diagnostic tools ESS (effective sample size)
- Solution: thinning, (keep one sample every n)

```
import arviz as az

with MetropolisHastings(num_particles=1000, warmups=1000):
  chains = [infer(gauss, obs).samples for _ in range(4)]
  az_data = az.convert_to_inference_data(np.array(chains))
  az.summary(az_data)
```

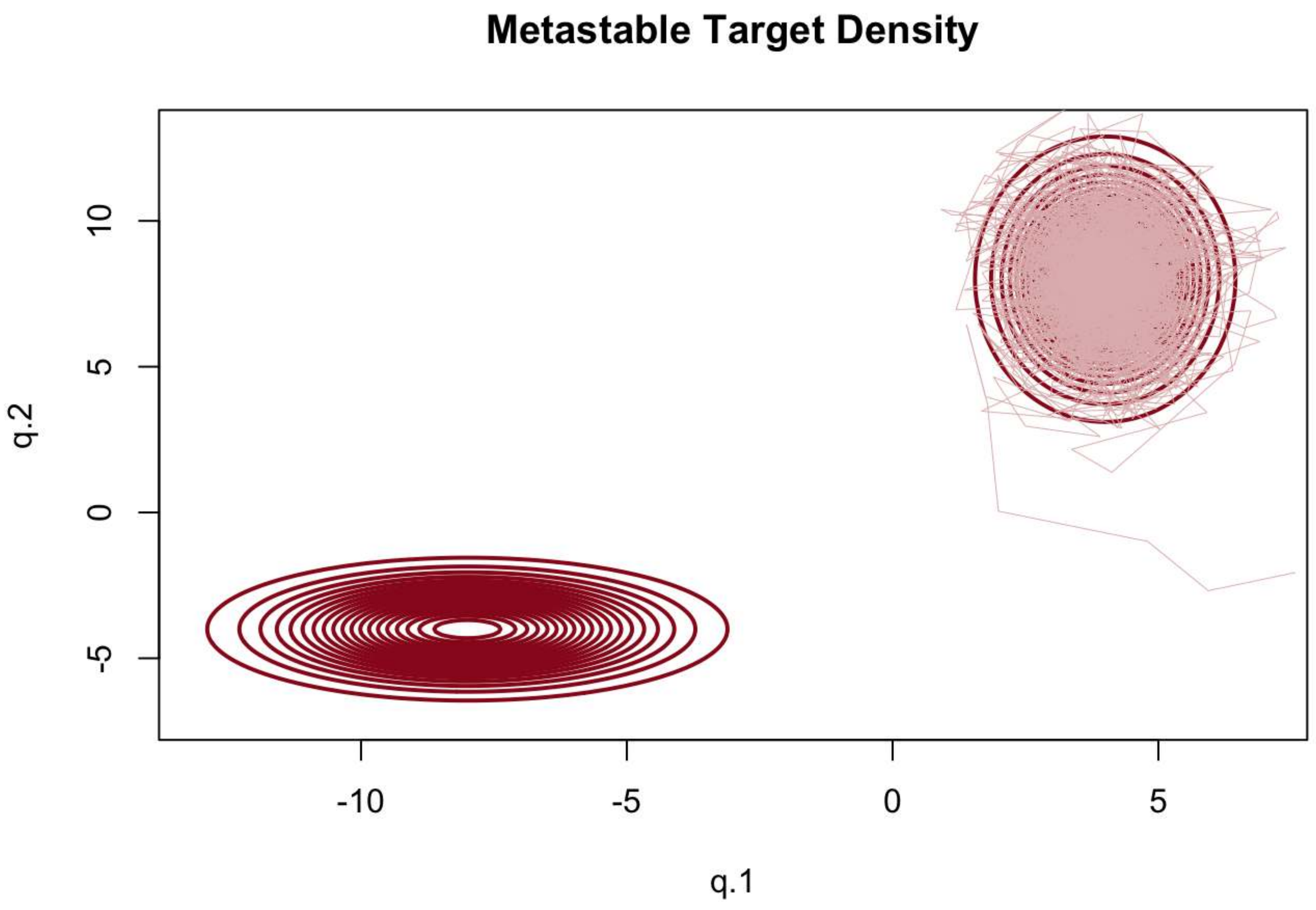
	mean	sd	ess_bulk	r_hat
x	2.793	0.373	21.0	1.23
y	3.028	0.335	43.0	1.08



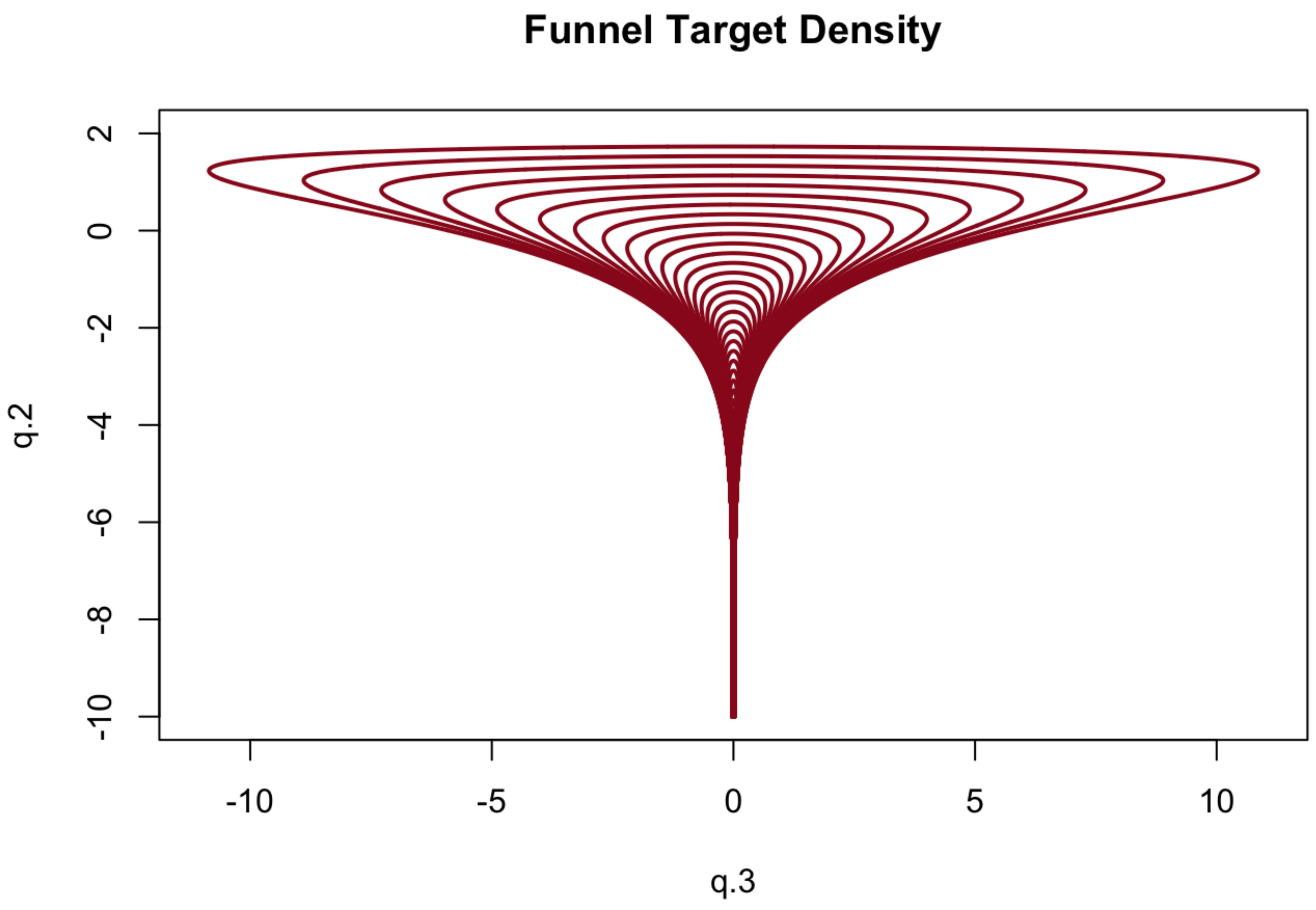


images from Vehtari et al. 2021

Pathological models



Multimodal distribution



Neal's funnel

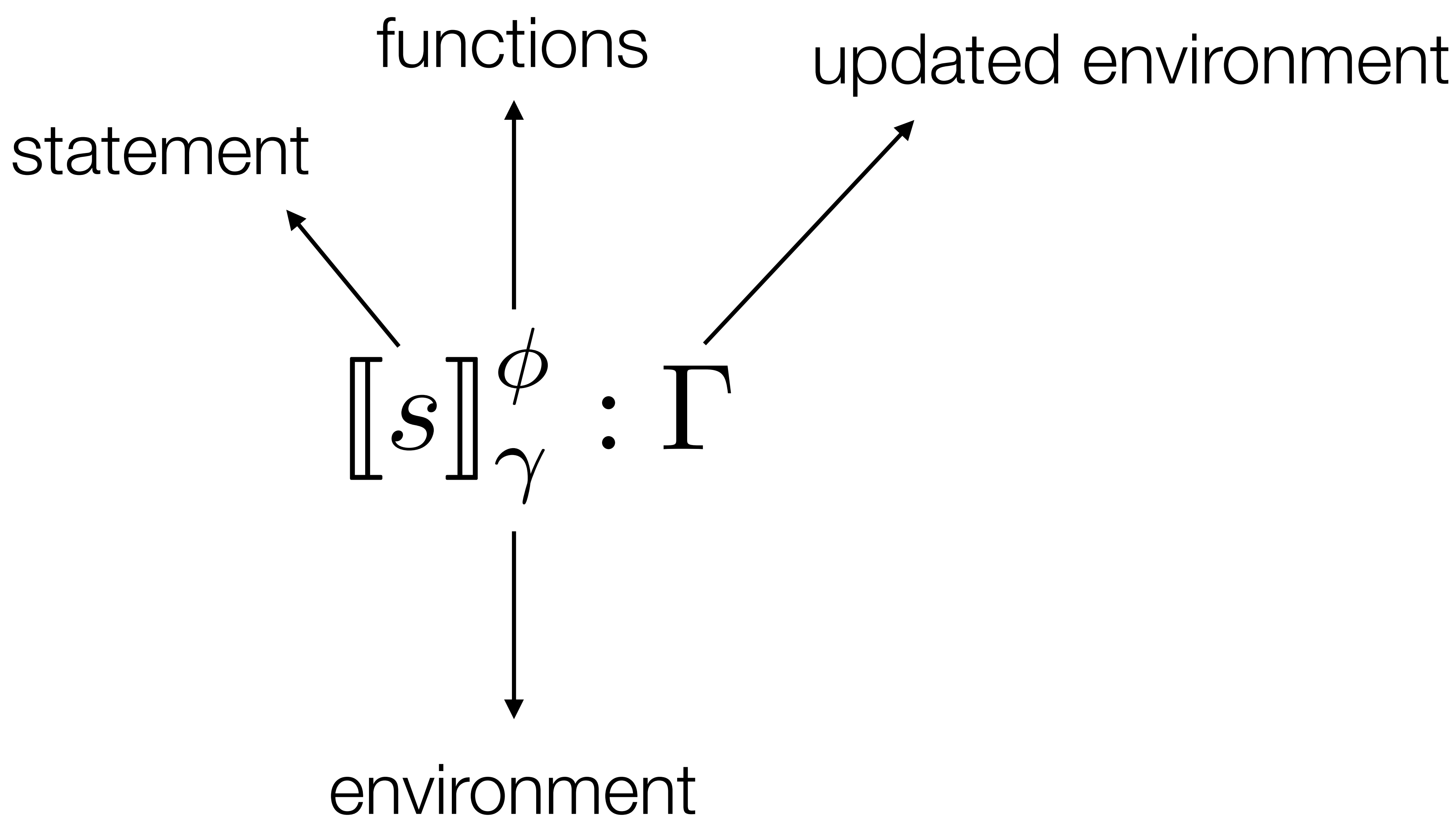
images from Bettancourt 2020

Part II. Sequential Monte Carlo methods

Part III. Density Semantics

Introduction to Probabilistic Programming

Reminders: deterministic semantics



$$[\![s]\!]_{\gamma}^{\phi} : \Gamma$$

Reminders: deterministic semantics

functions

updated environment

statement

$$[\![s]\!]_{\gamma}^{\phi} : \Gamma$$

environment

Example:

$$[\![y \texttt{ = } x + 40]\!]_{[x \leftarrow 2]}^{\emptyset} = [x \leftarrow 2,\ y \leftarrow 42]$$

Reminders: kernel semantics

functions

Measure over updated environments

statement

$$[\![s]\!]^{\phi}_{\gamma} : \mathsf{Meas}(\Gamma)$$

environment

Reminders: kernel semantics

functions

Measure over updated environments

statement

$$[\![s]\!]_{\gamma}^{\phi} : \textsf{Meas}(\Gamma)$$

environment

Example:

$$[\![x = \texttt{sample}(\mathcal{N}(0,1))]\!]_{\emptyset}^{\emptyset}(\{[x \leftarrow v] \mid v > 0\}) = 0.5$$

Reminders: kernel semantics

Unnormalized measure

functions

Measure over updated environments

statement

$$[\![s]\!]^{\phi}_{\gamma} : \mathsf{Meas}(\Gamma)$$

environment

Example:

$$[\![x \texttt{ = } \texttt{sample}(\mathcal{N}(0,1))]\!]^{\emptyset}_{\emptyset}(\{[x \leftarrow v] \mid v > 0\}) = 0.5$$

Reminders: kernel semantics

Unnormalized measure

functions

Measure over updated environments

statement

$$[\![s]\!]^{\phi}_{\gamma} : \text{Meas}(\Gamma)$$

environment

Example:

$$[\![x \texttt{ = sample}(\mathcal{N}(0,1))]\!]^{\emptyset}_{\emptyset}(\{[x \leftarrow v] \mid v > 0\}) = 0.5$$

$$[\![\texttt{infer}(f, e)]\!]^{\phi}_{\gamma}(U) = \begin{cases} \frac{\mu(U)}{\mu(\top)} & \textit{avec } \mu(U) = \phi(f)([\![e]\!]^{\phi}_{\gamma}) \quad \textit{si } 0 < \mu(\top) < \infty \\ \text{Erreur} & \textit{sinon} \end{cases}$$

Density semantics

Key idea

- A model is a function $f : P \to t \times \mathbb{R}^+$
- Associate a value $v(p)$ and a score $W(p)$ to parameters (random variables)
- Deterministic function *given an oracle* for the parameters

Interpretation close to our weighted samplers for approximate inference

Back to measure

- ρ: uniform distributions on parameters
- We get a measure by integrating f along ρ

$$\mu(U) = \int \rho(dp)W(p)\ \delta_{v(p)}(U)$$

Density semantics

Key idea

- A model is a function $f : P \to t \times \mathbb{R}^+$
- Associate a value $v(p)$ and a score $W(p)$ to parameters (random variables)
- Deterministic function *given an oracle* for the parameters

Interpretation close to our weighted samplers for approximate inference

Back to measure

- ρ: uniform distributions on parameters
- We get a measure by integrating f along ρ

$$\mu(U) = \int \rho(dp) W(p)\ \delta_{v(p)}(U)$$

Problem: Random variables can change between two executions

```
c = sample(Bernoulli(0.5))
if c: x = sample(Gaussian(0, 1))
```

Measure over parameters

Lee et al. 2020

Measure over parameters

Key ideas

- Associate a unique name to each random variable (same as Metropolis-Hastings)
- Map random elements in [0, 1] to samples using inverse transform sampling

Inverse transform sampling

- Draw a sample $u \sim \textit{Uniform}(0, 1)$
- Compute sample value $x = \textit{icdf}(\mu)$
- $\textit{icdf}(\mu)$: generalized inverse of the cumulative distribution function

Lee et al. 2020

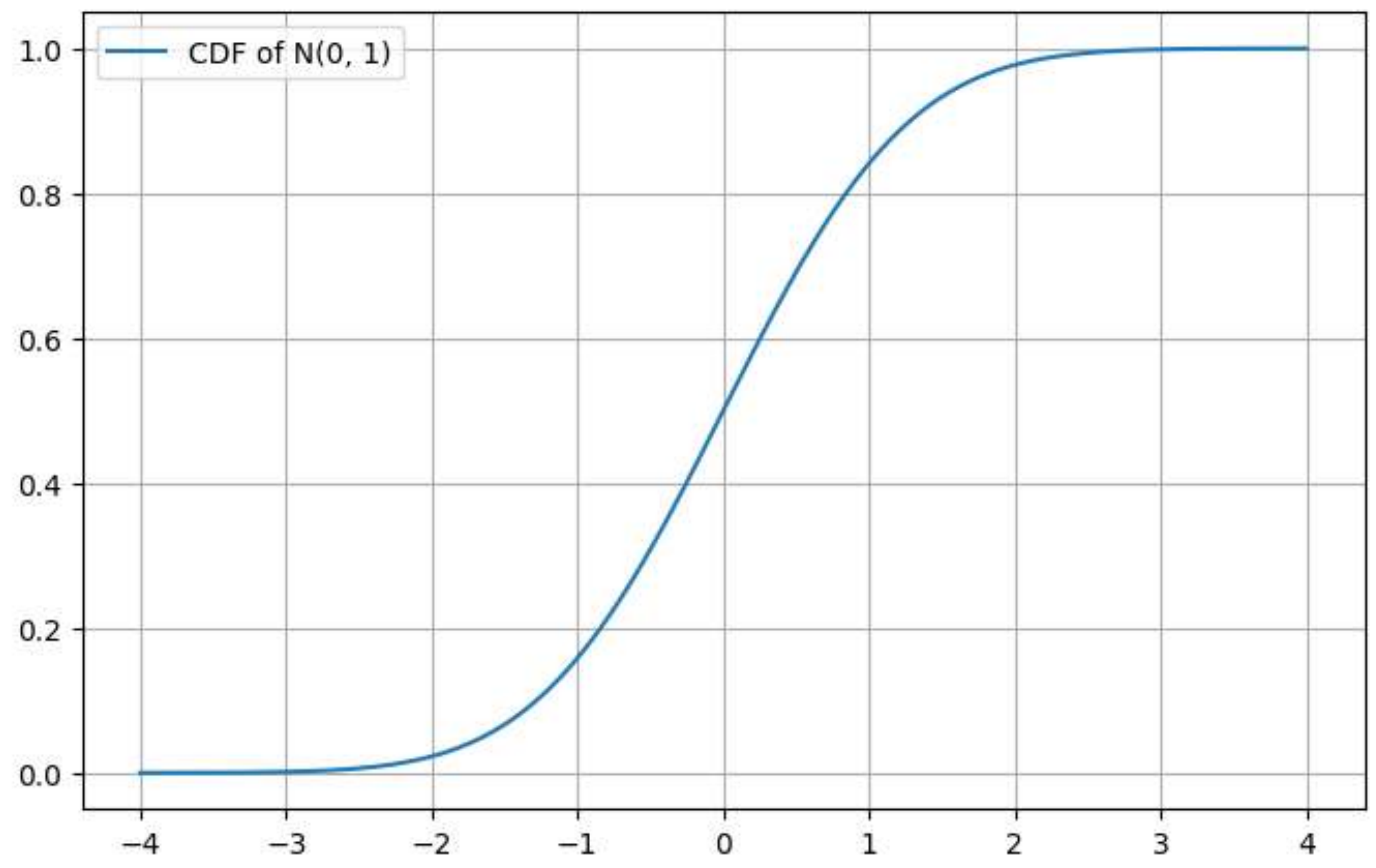
Measure over parameters

Key ideas

- Associate a unique name to each random variable (same as Metropolis-Hastings)
- Map random elements in [0, 1] to samples using inverse transform sampling

Inverse transform sampling

- Draw a sample $u \sim \textit{Uniform}(0, 1)$
- Compute sample value $x = \textit{icdf}(\mu)$
- $\textit{icdf}(\mu)$: generalized inverse of the cumulative distribution function



Lee et al. 2020

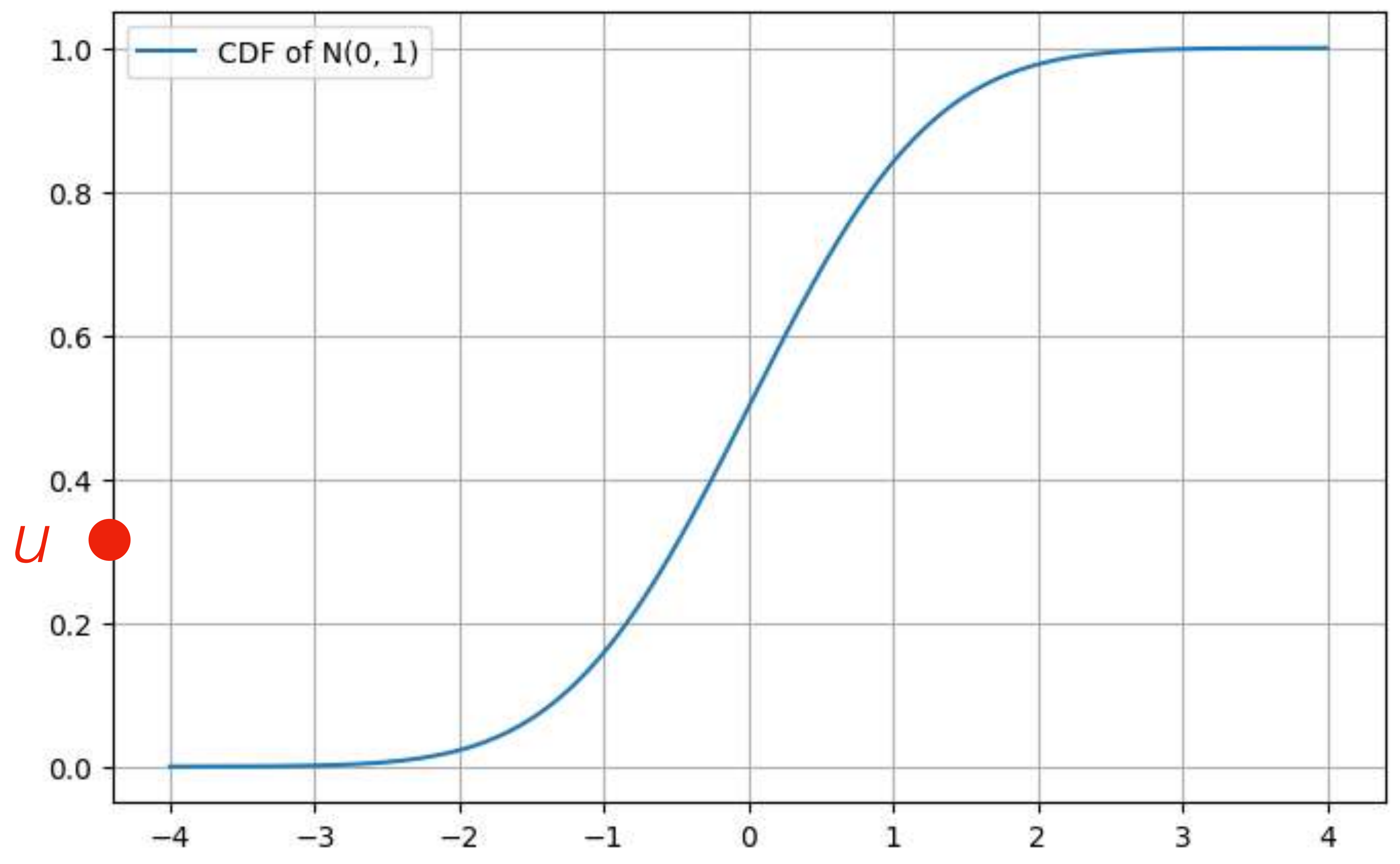
Measure over parameters

Key ideas

- Associate a unique name to each random variable (same as Metropolis-Hastings)
- Map random elements in [0, 1] to samples using inverse transform sampling

Inverse transform sampling

- Draw a sample $u \sim \textit{Uniform}(0, 1)$
- Compute sample value $x = \textit{icdf}(\mu)$
- $\textit{icdf}(\mu)$: generalized inverse of the cumulative distribution function



Lee et al. 2020

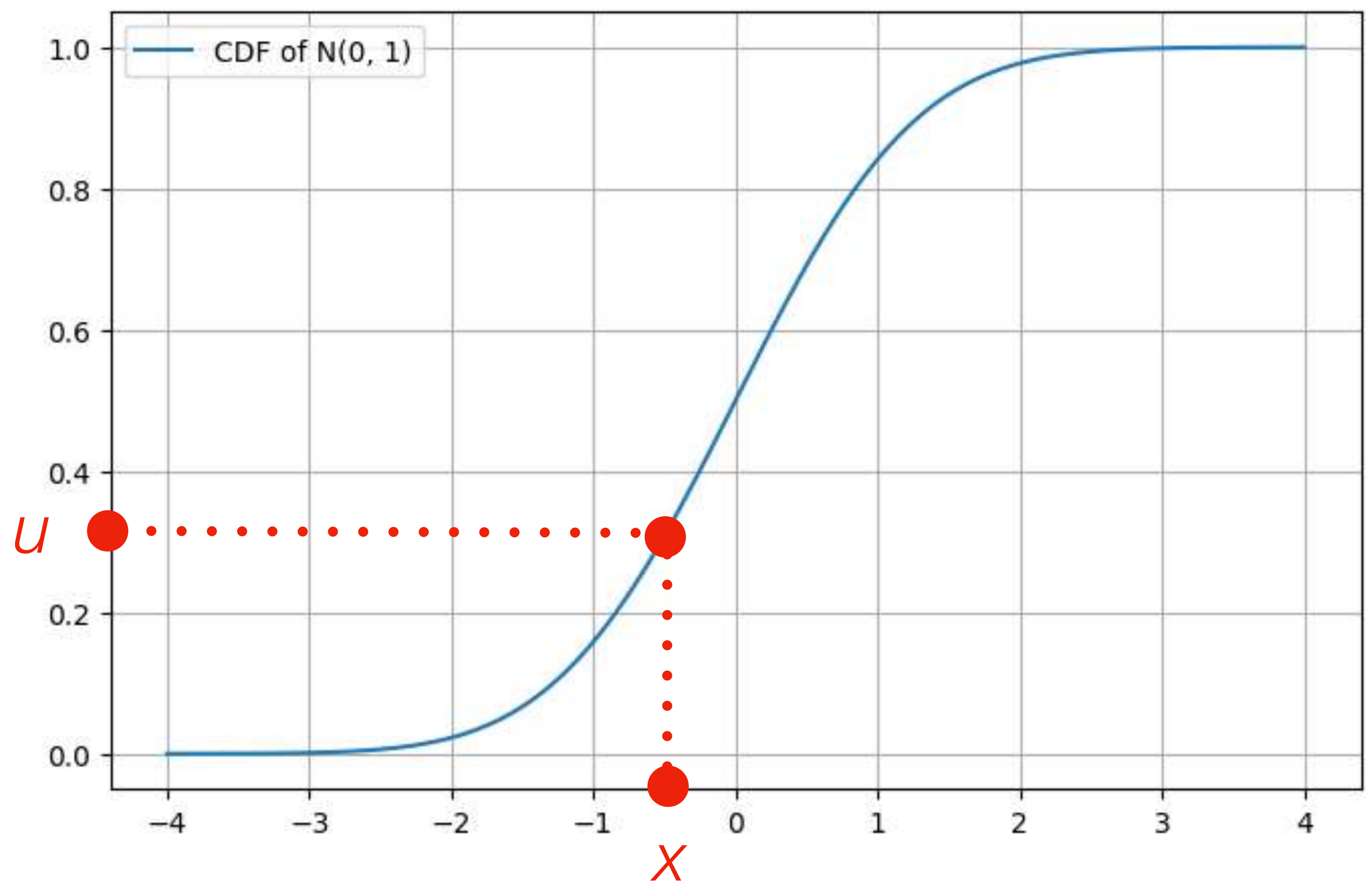
Measure over parameters

Key ideas

- Associate a unique name to each random variable (same as Metropolis-Hastings)
- Map random elements in [0, 1] to samples using inverse transform sampling

Inverse transform sampling

- Draw a sample $u \sim \textit{Uniform}(0, 1)$
- Compute sample value $x = \textit{icdf}(\mu)$
- $\textit{icdf}(\mu)$: generalized inverse of the cumulative distribution function



Lee et al. 2020

Measure over parameters

Key ideas

- Associate a unique name to each random variable (same as Metropolis-Hastings)
- Map random elements in [0, 1] to samples using inverse transform sampling

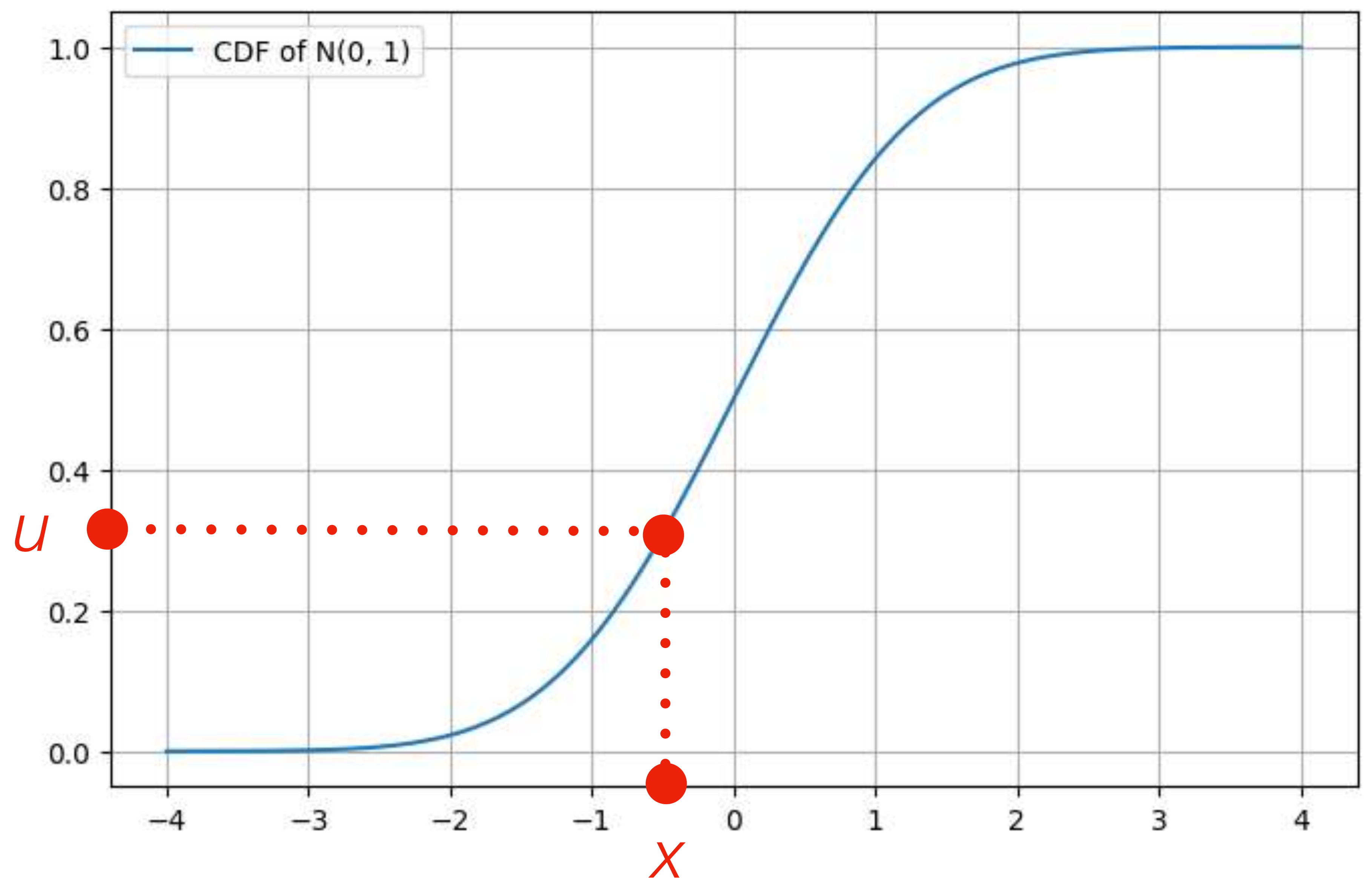
Inverse transform sampling

- Draw a sample $u \sim \textit{Uniform}(0, 1)$
- Compute sample value $x = \textit{icdf}(\mu)$
- $\textit{icdf}(\mu)$: generalized inverse of the cumulative distribution function

Uniform measure over parameters

- Group set of parameters with same domain names
- Compute the sizes with the Lesbegue measure λ over $[0, 1]$
- Sum all the results

$$\rho(r) = \left(\sum_{K \subset \textit{Str}} \bigotimes_{\alpha \in K} \lambda \right)(r)$$



Lee et al. 2020

Density semantics

functions

statement

random element

score

$$(\!| s |\!)_{\gamma}^{\phi}\,(r) : \Gamma \times \mathbb{R}^{+}$$

updated environment

environment

Density semantics

$$(\!|s|\!)_{\gamma}^{\phi}(r) : \Gamma \times \mathbb{R}^{+}$$

- statement: s
- functions: ϕ
- random element: r
- score: $\mathbb{R}^{+}$
- environment: γ
- updated environment: Γ

Example:

$$[\![x = \texttt{sample}(\mathcal{N}(0,1), \texttt{x})]\!]_{\emptyset}^{\emptyset}(r) = [x \leftarrow \textit{icdf}(r(\texttt{x}))], 1$$

Density semantics

functions

statement — random element — score

$$(\!|s|\!)_\gamma^\phi\,(r) : \Gamma \times \mathbb{R}^+$$

environment — updated environment

Example:

$$[\![x\ \texttt{=}\ \texttt{sample}(\mathcal{N}(0,1), \texttt{x})]\!]_\emptyset^\emptyset\,(r) = [x \leftarrow \textit{icdf}(r(\texttt{x}))], 1$$

$$(\!|\texttt{infer}(f, e)|\!)_\gamma^\phi\,(U) = \begin{cases} \dfrac{\mu(U)}{\mu(\top)} & \textit{avec} \begin{cases} \mu(U) = \displaystyle\int \rho(dr)\ W(r)\ \delta_{v(r)}(U) \\ v(r), W(r) = \phi(f)((\!|e|\!)_\gamma^\phi\,(r)) \end{cases} \quad \textit{si } 0 < \mu(\top) < \infty \\ \text{Erreur} & \textit{sinon} \end{cases}$$

Density semantics

$$
\begin{aligned}
(\!|c|\!)_\gamma^\phi &= c \\
(\!|x|\!)_\gamma^\phi &= \gamma(x) \\
(\!|\mathit{op}(e)|\!)_\gamma^\phi &= \mathit{op}((\!|e|\!)_\gamma^\phi) \\
(\!|(e_1, e_2)|\!)_\gamma^\phi &= \left((\!|e_1|\!)_\gamma^\phi, (\!|e_2|\!)_\gamma^\phi\right)
\end{aligned}
$$

$$
\begin{aligned}
(\!|\texttt{pass}|\!)_\gamma^\phi(r) &= (\gamma, 1) \\
(\!|\texttt{if}\ e\texttt{:}\ s_1\ \texttt{else:}\ s_2|\!)_\gamma^\phi(r) &= (\!|s_1|\!)_\gamma^\phi(r)\ \textit{si}\ (\!|e|\!)_\gamma^\phi\ \textit{sinon}\ (\!|s_2|\!)_\gamma^\phi(r) \\
(\!|s_1\texttt{;}\ s_2|\!)_\gamma^\phi(r) &= (\gamma_2, W_1 \cdot W_2)\ \textit{avec}\ (\gamma_1, W_1) = (\!|s_1|\!)_\gamma^\phi(r) \\
&\qquad \textit{et}\ (\gamma_2, W_2) = (\!|s_2|\!)_{\gamma_1}^\phi(r)
\end{aligned}
$$

$$
\begin{aligned}
(\!|x\ \texttt{=}\ e|\!)_\gamma^\phi(r) &= \left(\gamma + \left[x \leftarrow (\!|e|\!)_\gamma^\phi\right], 1\right) \\
(\!|x\ \texttt{=}\ \texttt{sample}(e, \mathit{name})|\!)_\gamma^\phi(r) &= \left(\gamma + \left[x \leftarrow \mathit{icdf}\left((\!|e|\!)_\gamma^\phi, r(\mathit{name})\right)\right], 1\right) \\
(\!|x\ \texttt{=}\ f(e)|\!)_\gamma^\phi(r) &= (\gamma + [x \leftarrow v], W) \quad \textit{avec}\ v, W = \phi(f)((\!|e|\!)_\gamma^\phi)(r)
\end{aligned}
$$

$$
(\!|\texttt{factor}(e)|\!)_\gamma^\phi(r) = \left(\gamma, (\!|e|\!)_\gamma^\phi\right)
$$

Example: my Gaussian

```
def my_gaussian(mu: float, sigma: float) → float:
    x = sample(Gaussian(mu, sigma))
    return x
```

$$\phi(\texttt{my\_gaussian})(\mu, \sigma)(r) = \mathit{icdf}(\mathcal{N}(\mu, \sigma), r(\texttt{x})), 1$$

$$\begin{aligned}\mu(U) &= \int_0^1 1\ \delta_{\mathit{icdf}(\mathcal{N}(\mu,\sigma),r_x)}(U)\ dr_x \\ &= \int \mathcal{N}(\mu, \sigma)(dv)\ \delta_v(U) \qquad \text{avec } v = \mathit{icdf}(\mathcal{N}(\mu, \sigma), r_x) \\ &= \mathcal{N}(\mu, \sigma)(U)\end{aligned}$$

Semantics equivalence

Theorem. *For a statement* s*, the density semantics of* s *is the density of the measure defined by the kernel semantics.*

$$[\![\mathsf{s}]\!]^{\phi}_{\gamma}(U) = \int \rho(dr) W_{\mathsf{s}}(r) \delta_{\gamma_{\mathsf{s}}(r)}(U) \qquad \textit{où } \gamma_{\mathsf{s}}(r), W_{\mathsf{s}}(r) = (\!| \mathsf{s} |\!)^{\phi}_{\gamma}(r)$$

Proof. By induction... (see notes) □

Semantics equivalence

Theorem. *For a statement* s*, the density semantics of* s *is the density of the measure defined by the kernel semantics.*

$$[\![\mathsf{s}]\!]^{\phi}_{\gamma}(U) = \int \rho(dr) W_{\mathsf{s}}(r) \delta_{\gamma_{\mathsf{s}}(r)}(U) \qquad \text{où } \gamma_{\mathsf{s}}(r), W_{\mathsf{s}}(r) = (\!|\mathsf{s}|\!)^{\phi}_{\gamma}(r)$$

Proof. By induction... (see notes) □

The kernel and density semantics define the same object

Takeaway

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

I - Approximate inference

- Importance sampling: weighted sampler returns pairs (value, score)
- Metropolis-Hastings: generate samples using a Markov chain over executions

II - Labs: Introduction to Sequential Monte Carlo methods

- State-space models for modeling time series
- Particle filtering with resampling

III - Density semantics

- Weighted samplers: operational semantics for approximate inference
- Measures on oracles: uniform measure and inverse transform sampling

References

An Introduction to Probabilistic Programming
Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, Frank Wood, 2018
https://arxiv.org/abs/1809.10756

Towards verified stochastic variational inference for probabilistic programs
Wonyeol Lee, Hangyeol Yu, Xavier Rival, Hongseok Yang, POPL 2020
https://arxiv.org/abs/1907.08827

Markov Chain Monte Carlo in Practice
Michael Bettancourt, 2020
https://betanalpha.github.io/assets/case_studies/markov_chain_monte_carlo.html

Rank-normalization, folding, and localization: An improved R-hat for assessing convergence of MCMC
Aki Vehtari, Andrew Gelman, Daniel Simpson, Bob Carpenter, and Paul-Christian Bürkner
https://arxiv.org/abs/1903.08008