

Probabilistic Programming Languages

Guillaume Baudart

SIESTE - ENS Lyon

Probabilistic programming languages

Probabilistic programming languages

General purpose programming languages extended with probabilistic constructs

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

Probabilistic programming languages

General purpose programming languages extended with probabilistic constructs

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

Multiple examples:

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

Probabilistic programming languages

General purpose programming languages extended with probabilistic constructs

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

Multiple examples:

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

More and more, incorporating new ideas:

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

Demo: mu-ppl

Bayesian reasoning

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

$\propto p(x) p(y_1, \dots, y_n \mid x)$ (Data are constants)



Thomas Bayes (1701-1761)

Bayesian reasoning

Bayesian Inference: learn parameters from data

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

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

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



Thomas Bayes (1701-1761)

Bayesian reasoning

Bayesian Inference: learn parameters from data

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

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

posterior

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

prior

likelihood



Thomas Bayes (1701-1761)



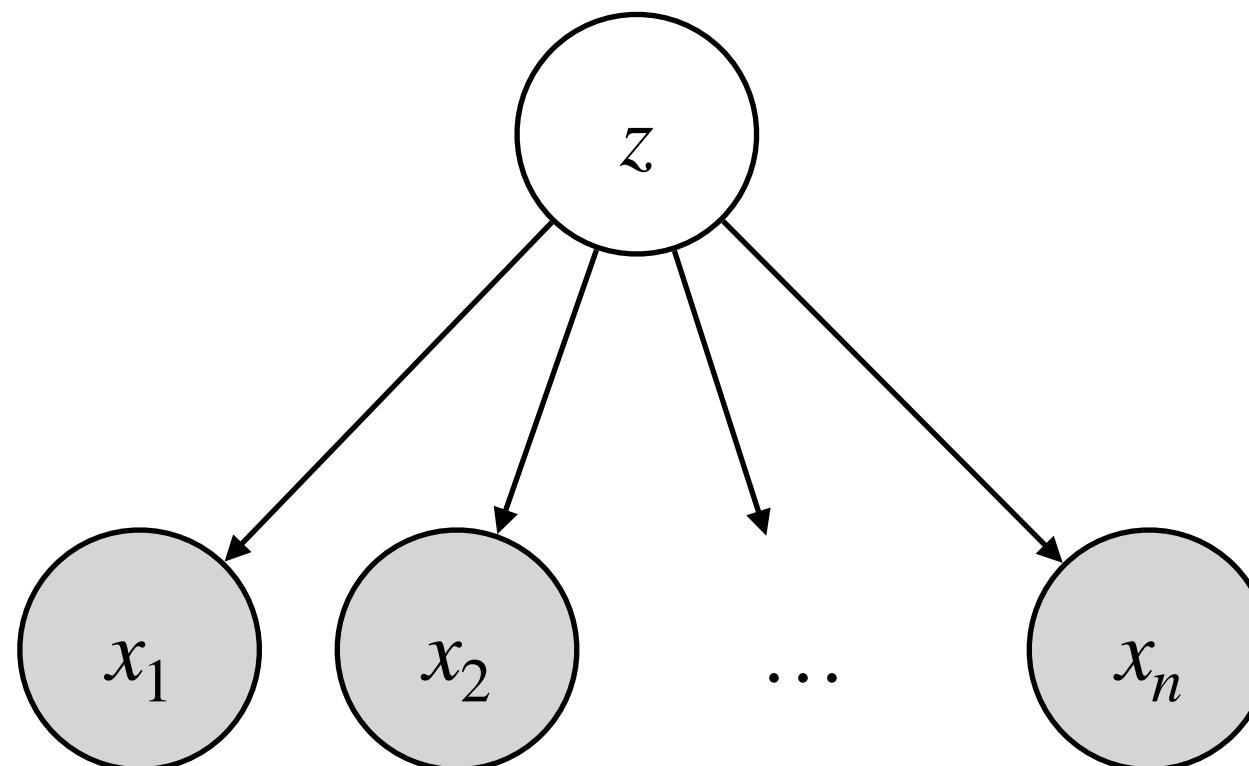
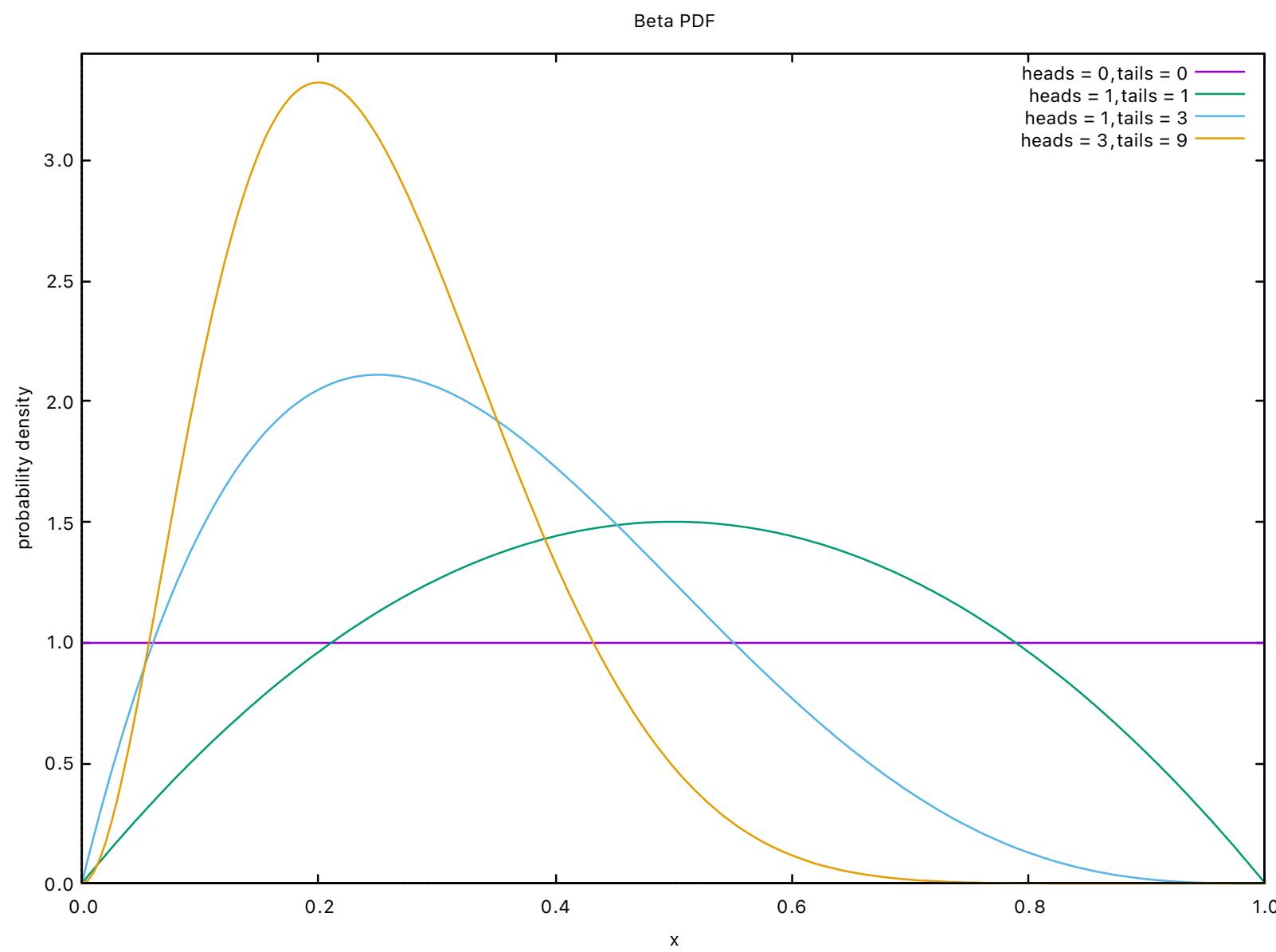
Example: coin

Consider a series of coin tosses

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

Probabilistic model

- Prior: $z \sim Uniform(0, 1)$
- Observations: for $i \in [1, n]$, $x_i \sim Bernoulli(z)$
- Posterior: $p(z | x_1, \dots, x_n)$?



latent
observed

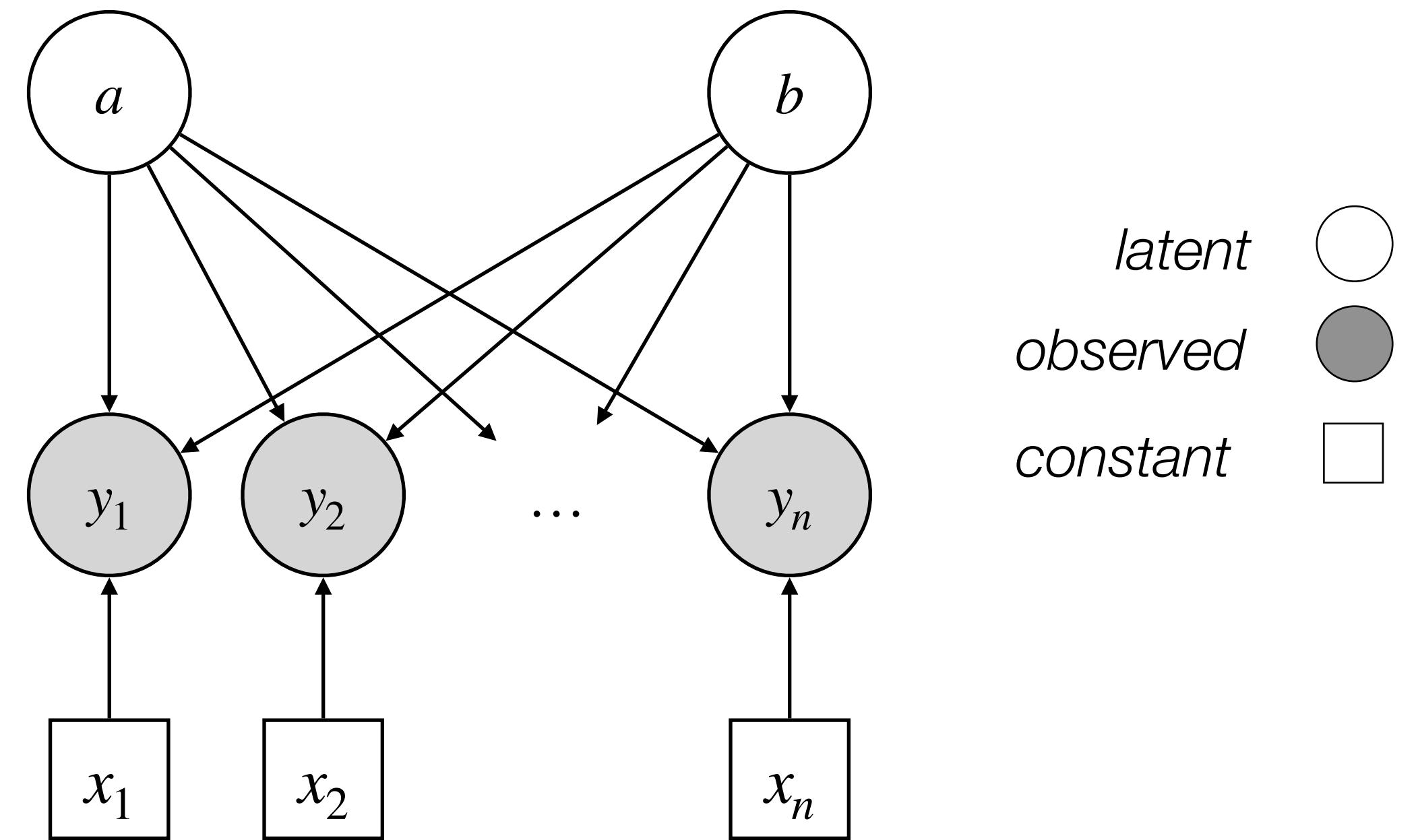
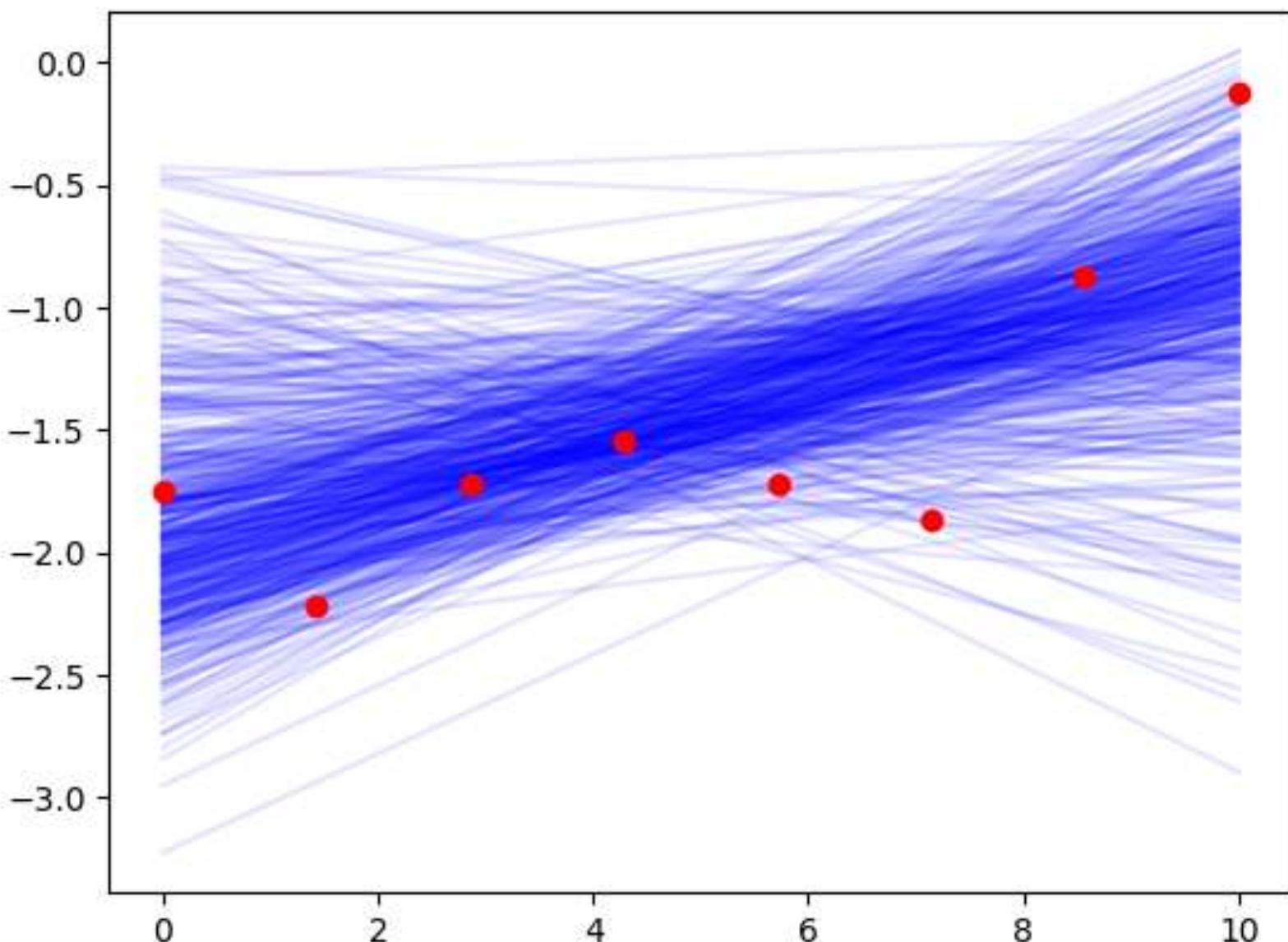
Example: linear regression

Consider a series of observations

- Observation: point (x, y)
- Each point is independent from others
- Find the distribution of possible regressions

Probabilistic model

- Prior: $a \sim \mathcal{N}(0, 1)$, and $b \sim \mathcal{N}(0, 1)$
- Observations: for $i \in [1, n]$, $y_i \sim \mathcal{N}(a \times x_i + b, \sigma)$
- Posterior: $p(a, b | (x_1, y_1), \dots, (x_n, y_n))$?



Probabilistic programming

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

$\propto p(x) p(y_1, \dots, y_n \mid x)$ (Data are constants)



Thomas Bayes (1701-1761)

Probabilistic programming

Bayesian Inference: learn parameters from data

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

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

posterior

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

prior

likelihood



Thomas Bayes (1701-1761)

Probabilistic programming

Bayesian Inference: learn parameters from data

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

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

posterior

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

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)

Probabilistic programming

Bayesian Inference: learn parameters from data

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

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

posterior

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

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

Notation: $x \sim \mu$
■ parameter (output): `sample(mu)`
■ observation (input): `observe(mu)`



Thomas Bayes (1701-1761)

Probabilistic programming

Bayesian Inference: learn parameters from data

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

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

posterior

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

prior

likelihood

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

```
infer(model, (y1, ... , yn))
```



Thomas Bayes (1701-1761)

Probabilistic programming

Bayesian Inference: learn parameters from data

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

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

posterior

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

prior

likelihood

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

```
infer(model, (y1, ... , yn))
```

A Bayesian model is describe by a program



Thomas Bayes (1701-1761)

Probabilistic programming

Probabilistic constructs

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

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 ()
```



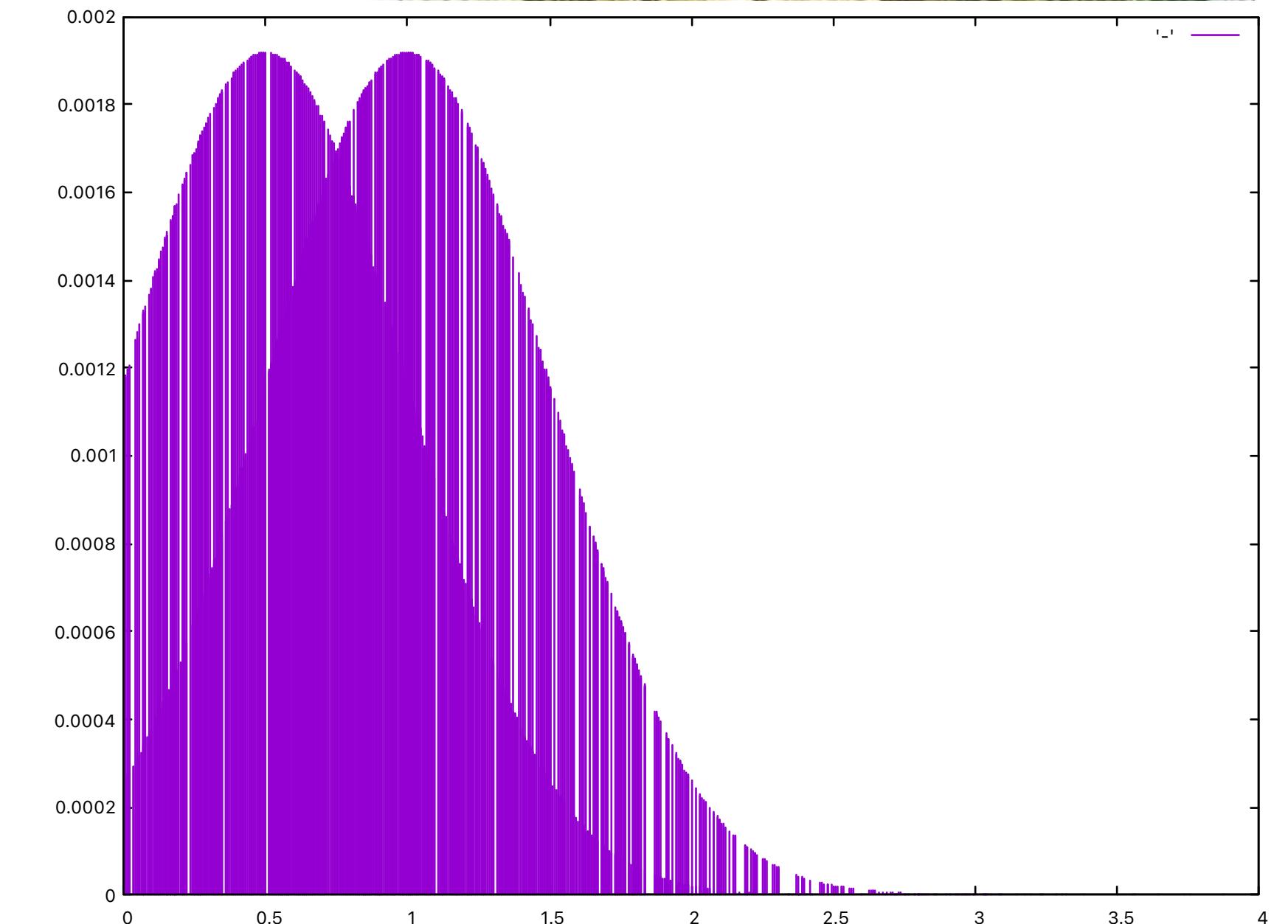
Probabilistic programming

Probabilistic constructs

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

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

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

More general than classic Bayesian reasoning

But: general case....

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

$$= \frac{p(\theta) p(x_1, \dots, x_n \mid \theta)}{\int p(\theta) p(x_1, \dots, x_n \mid \theta) d\theta}$$



Probabilistic programming

Probabilistic constructs

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

More general than classic Bayesian reasoning

But: general case....

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

$$= \frac{p(\theta) p(x_1, \dots, x_n | \theta)}{\int p(\theta) p(x_1, \dots, x_n | \theta) d\theta} = \underline{\underline{\hspace{1cm}}}$$



Probabilistic programming

Probabilistic constructs

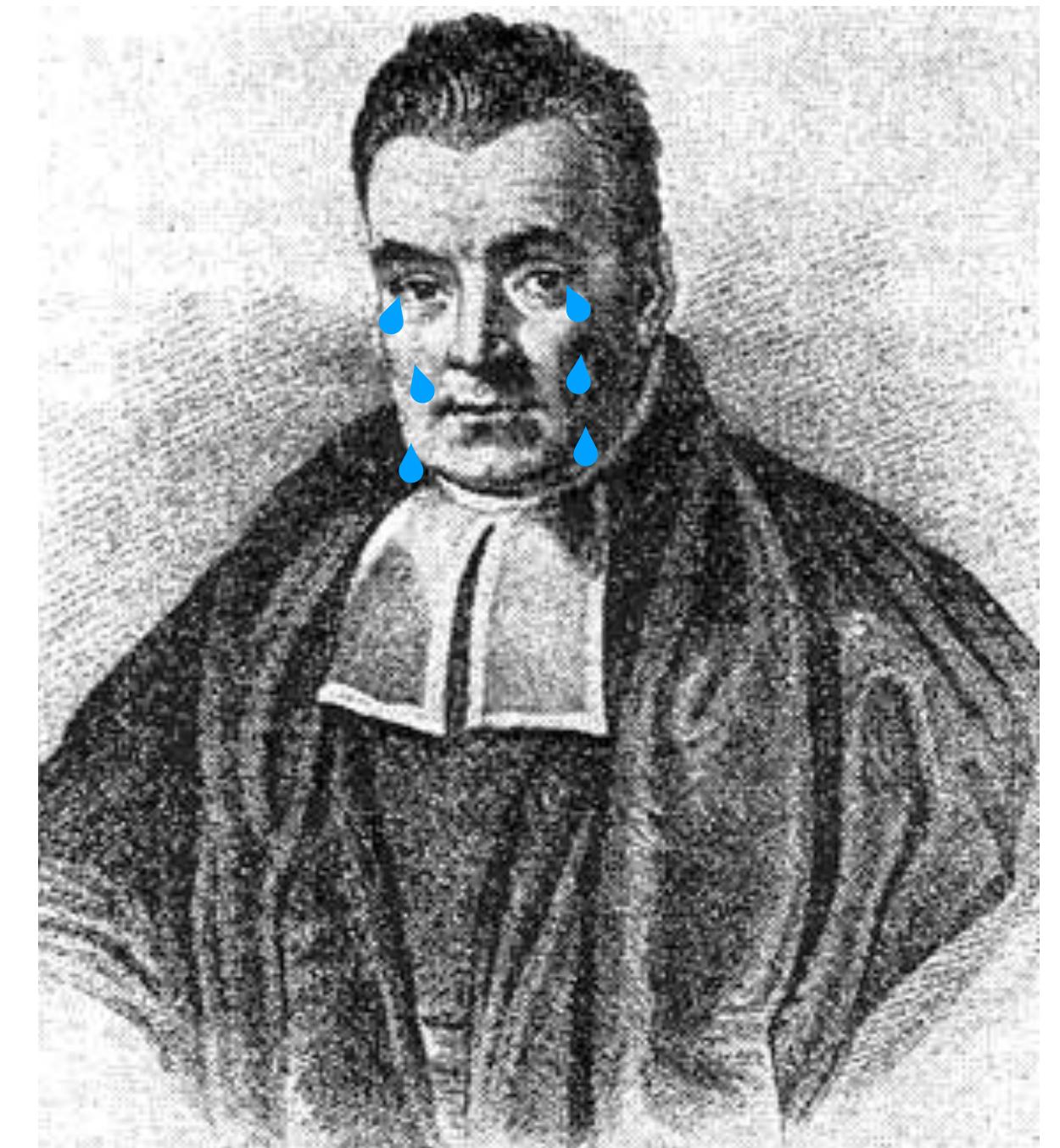
- **sample**: draw a sample from a distribution
- **assume, factor, observe**: condition the model on inputs (e.g., observed data)
- **infer**: compute the posterior distribution of a model given the inputs

More general than classic Bayesian reasoning

But: general case....

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

$$= \frac{p(\theta) p(x_1, \dots, x_n | \theta)}{\int p(\theta) p(x_1, \dots, x_n | \theta) d\theta} = \frac{\text{purple cake}}{\text{red skull}} \quad \text{intractable...}$$



Semantics

Probabilistic Programming Languages

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

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

program

i



i



i



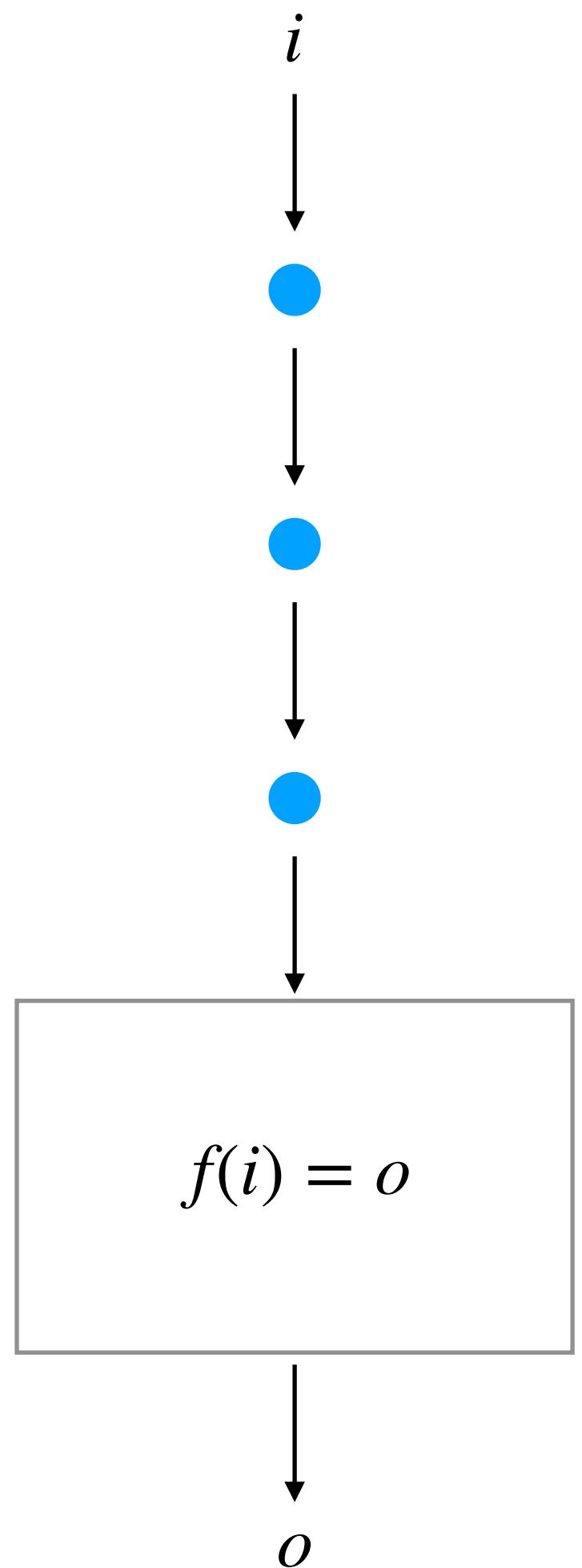
i



o

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

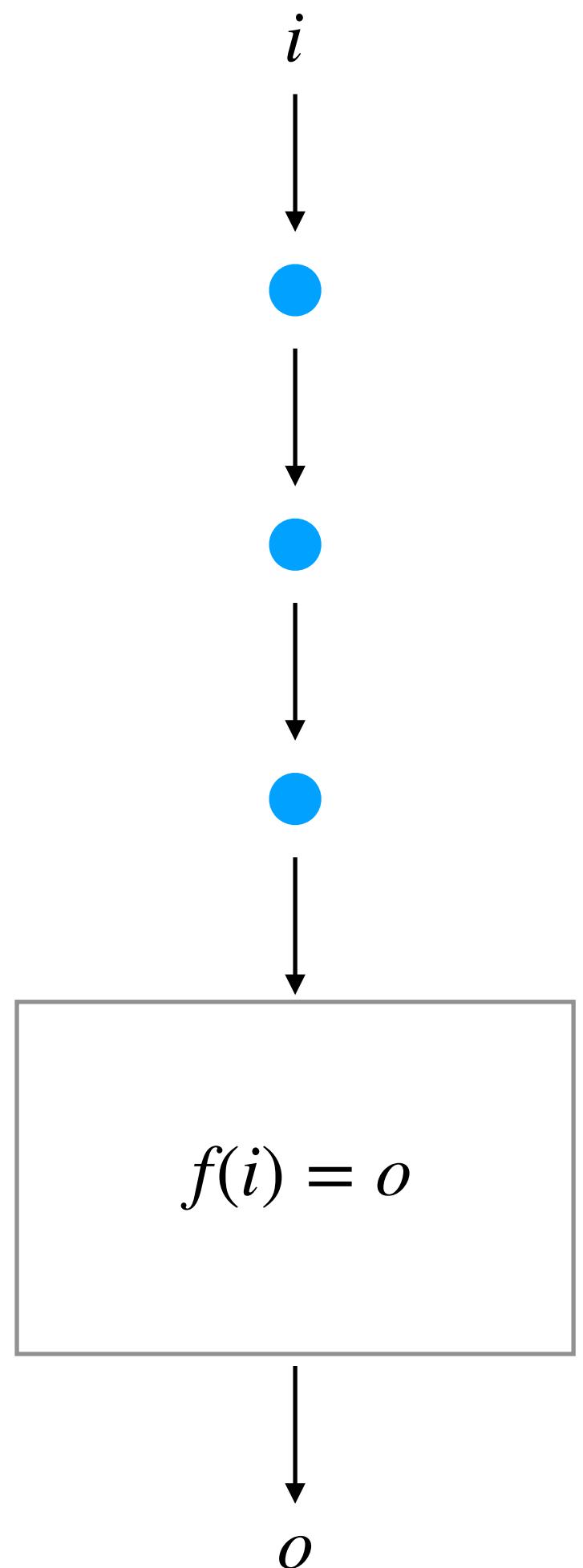
program



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

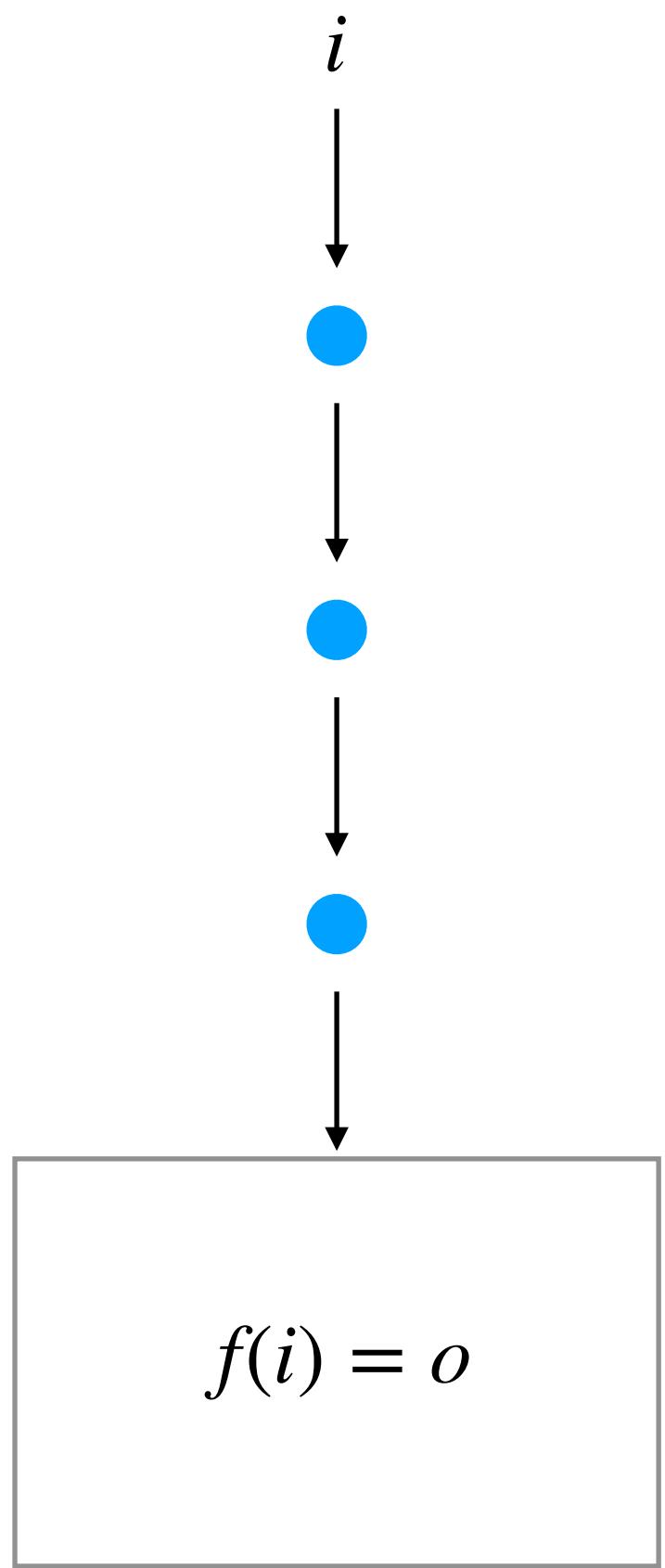
program

sample

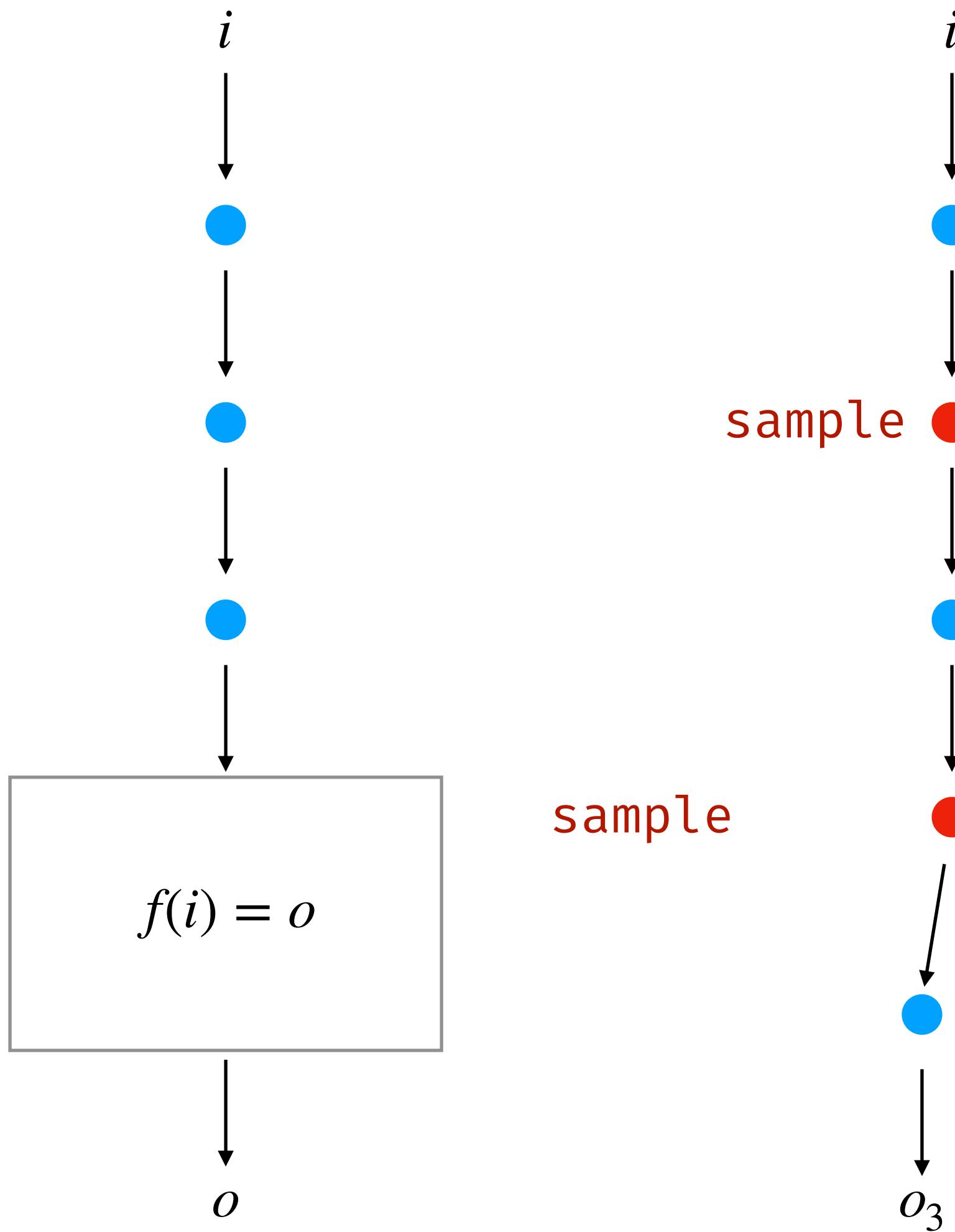


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

program

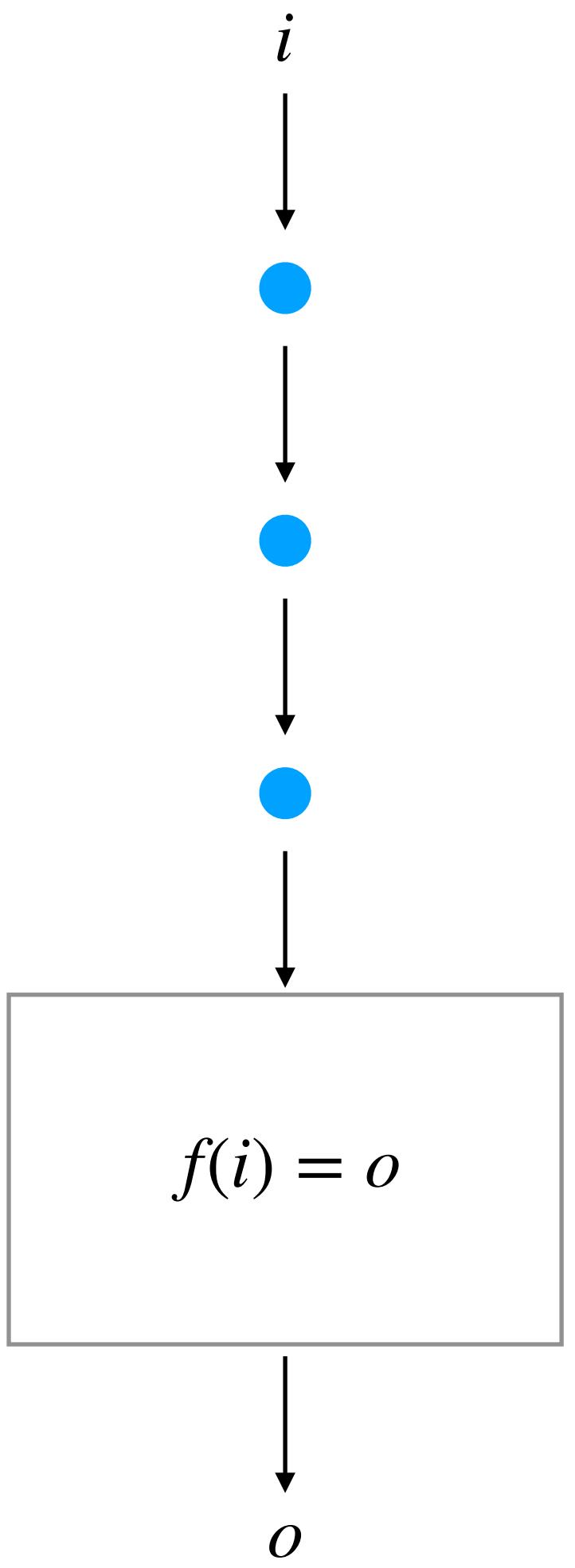


sample

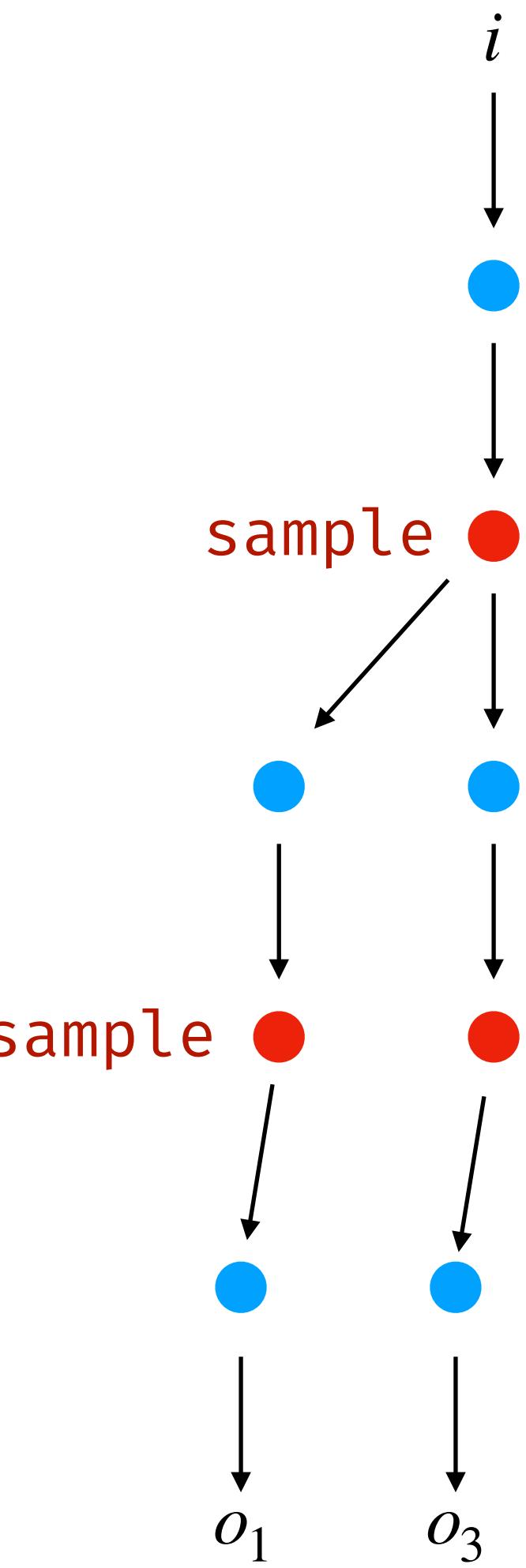


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

program

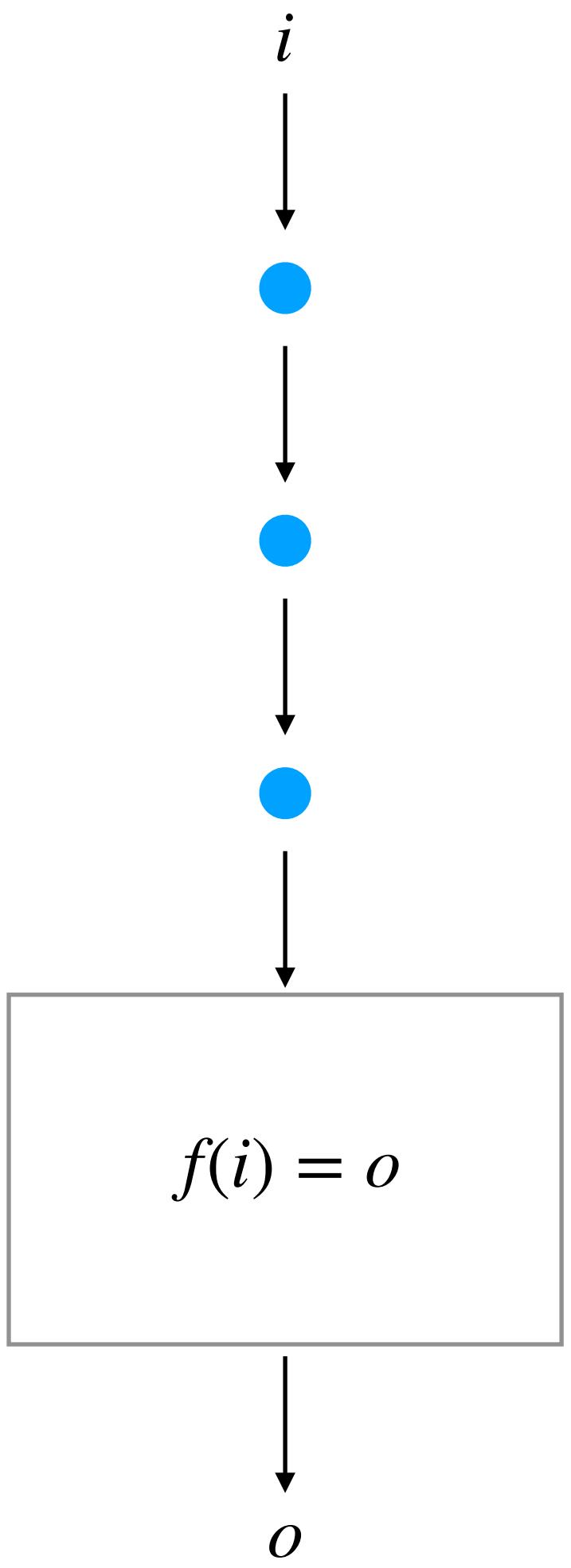


sample

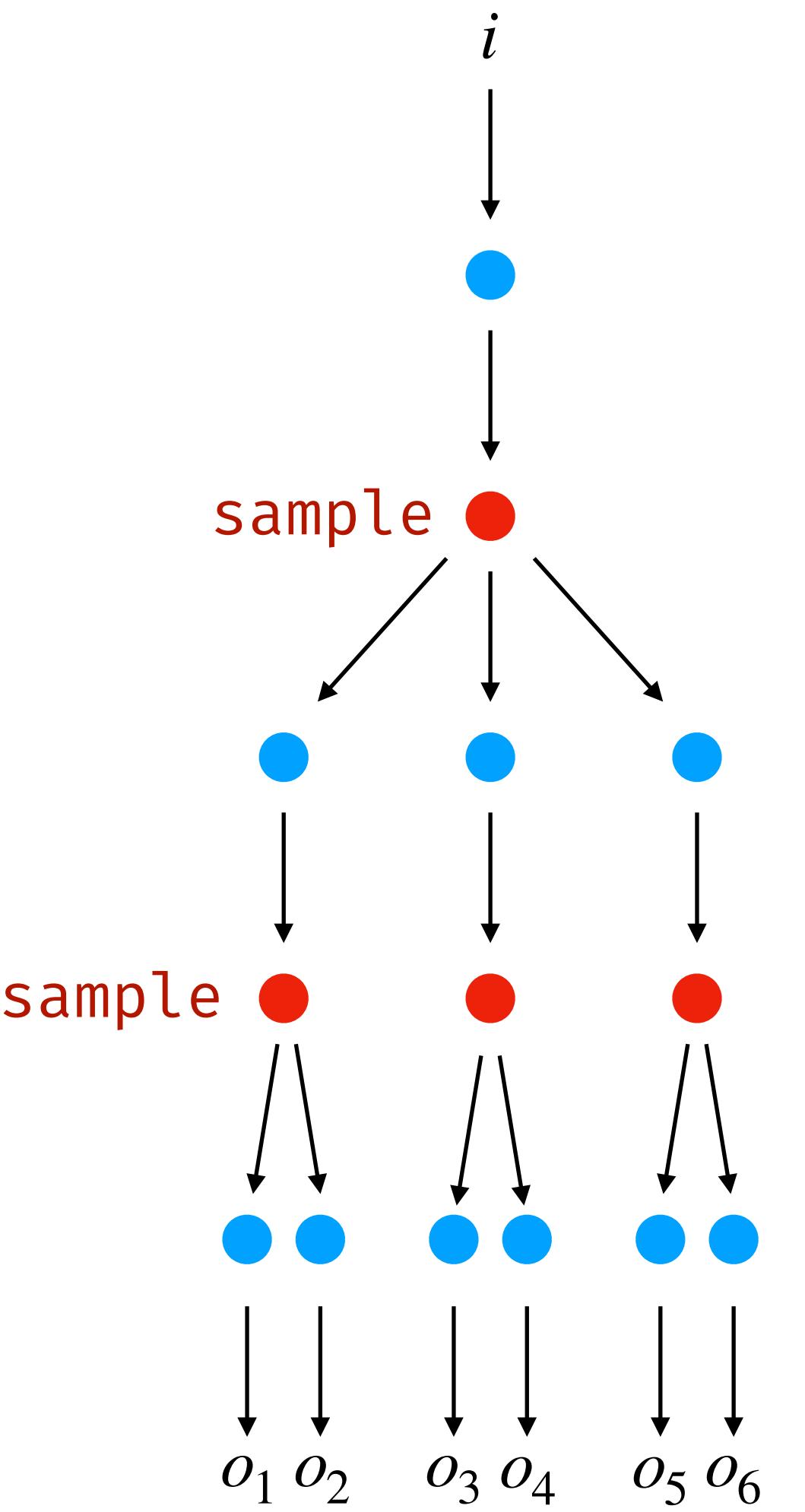


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

program

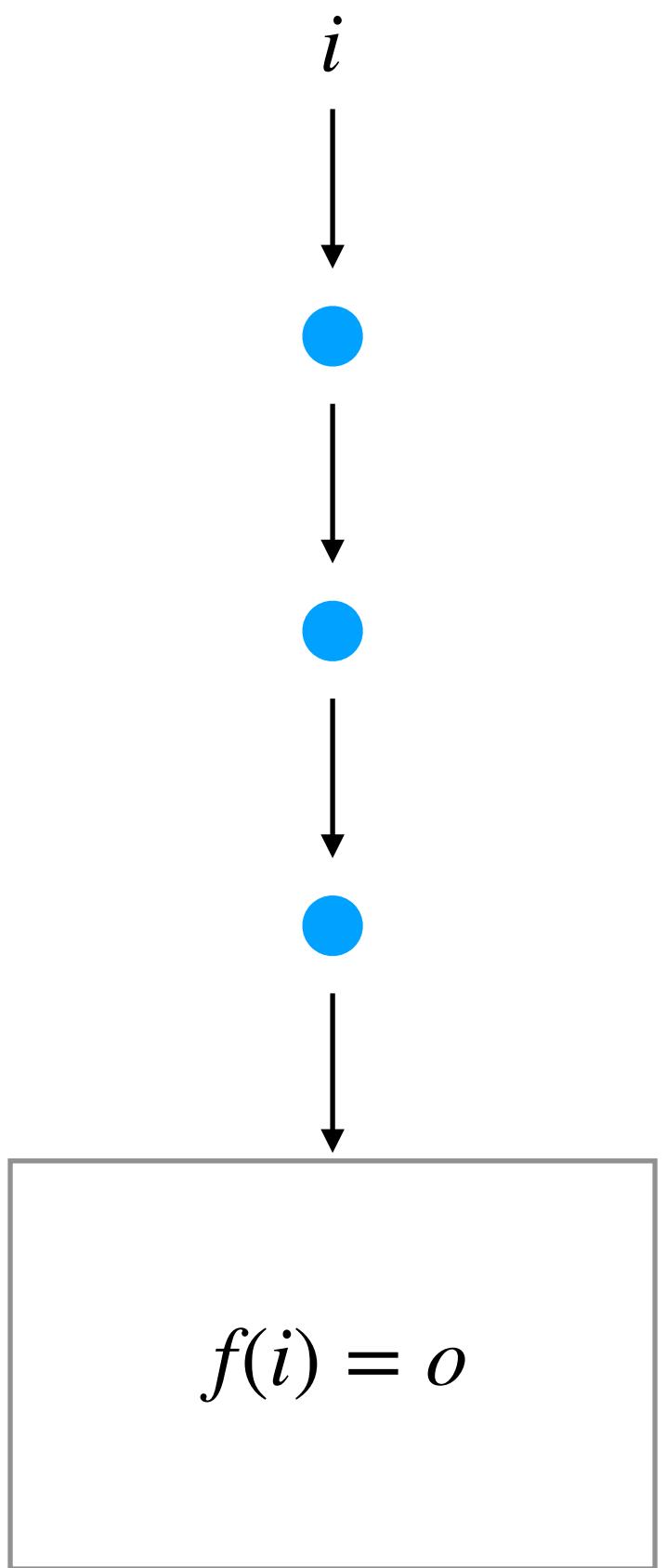


sample

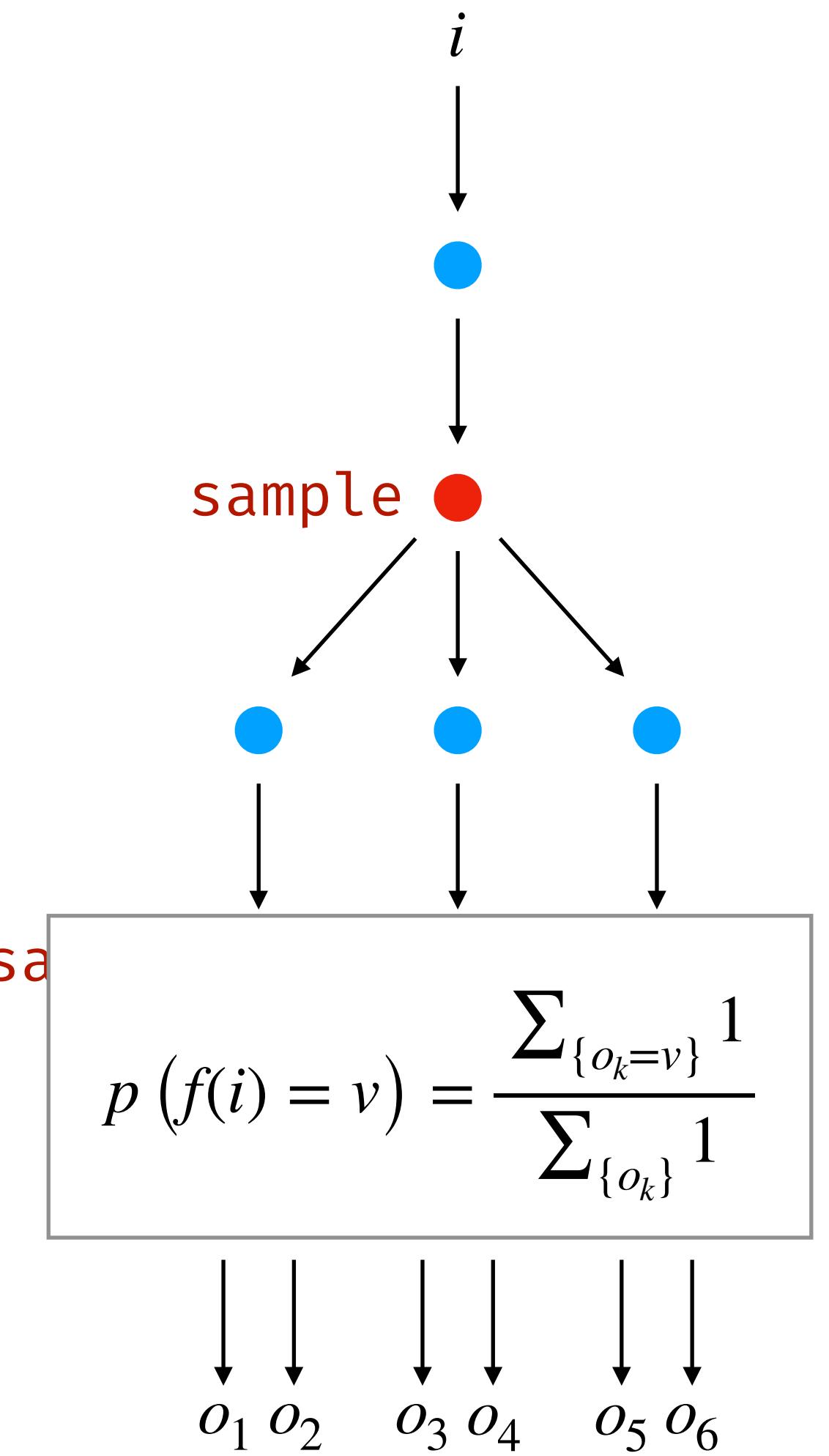


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

program

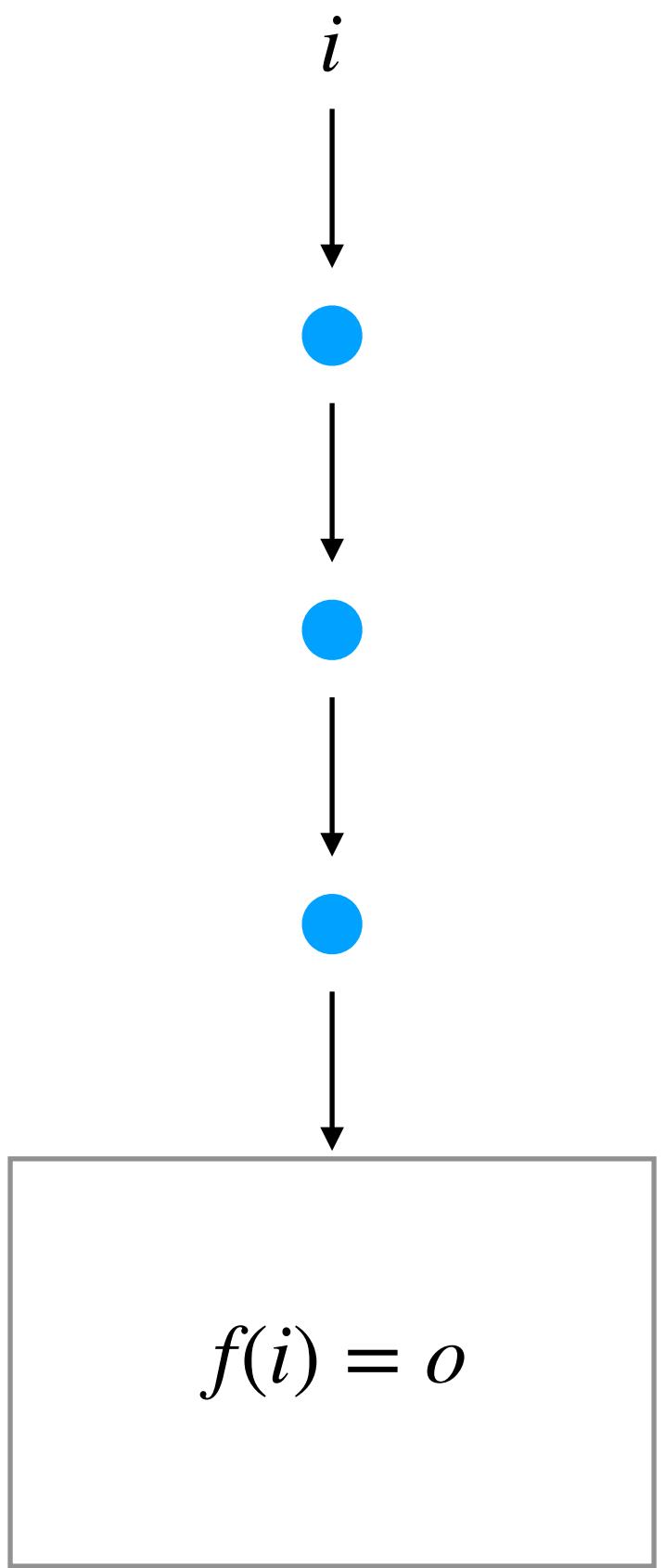


sample

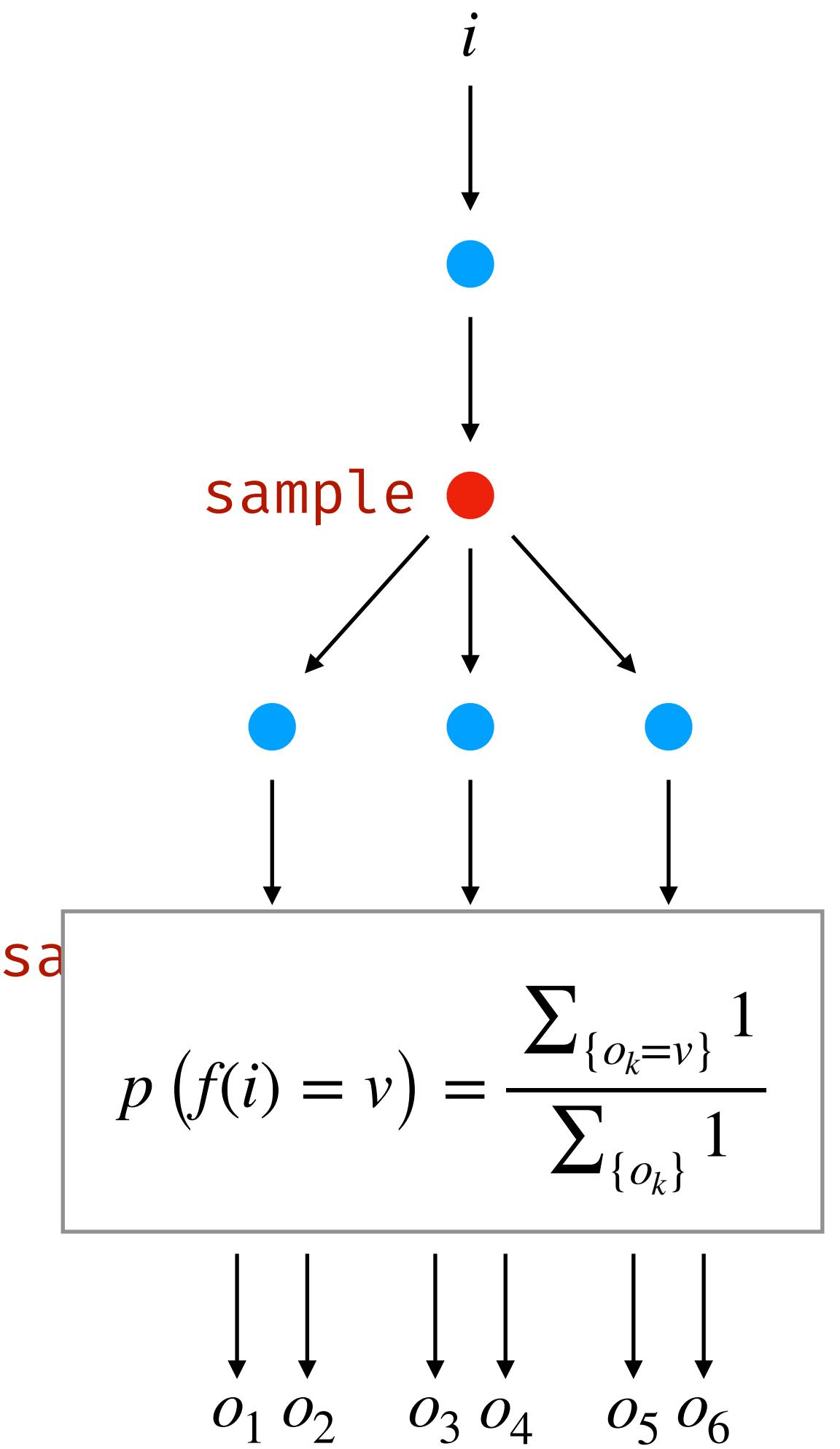


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

program



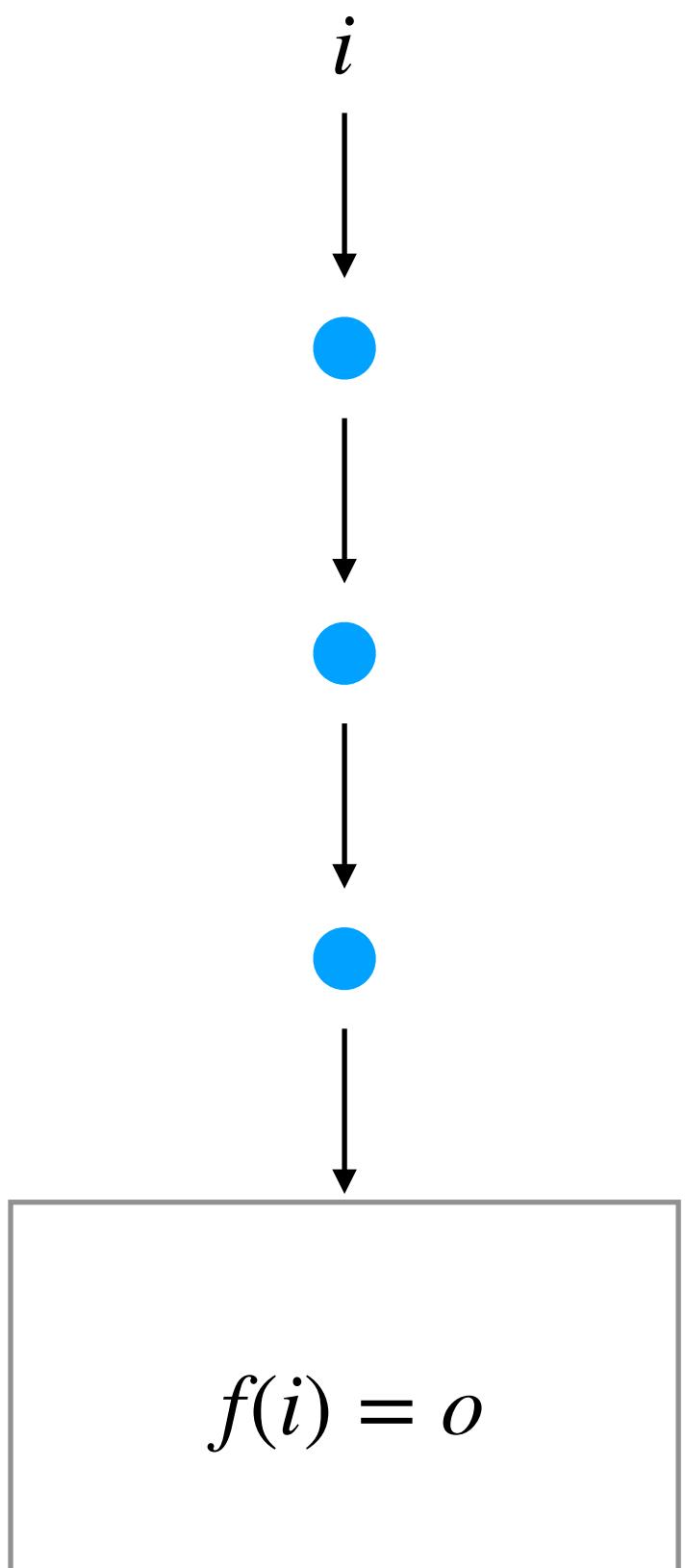
sample



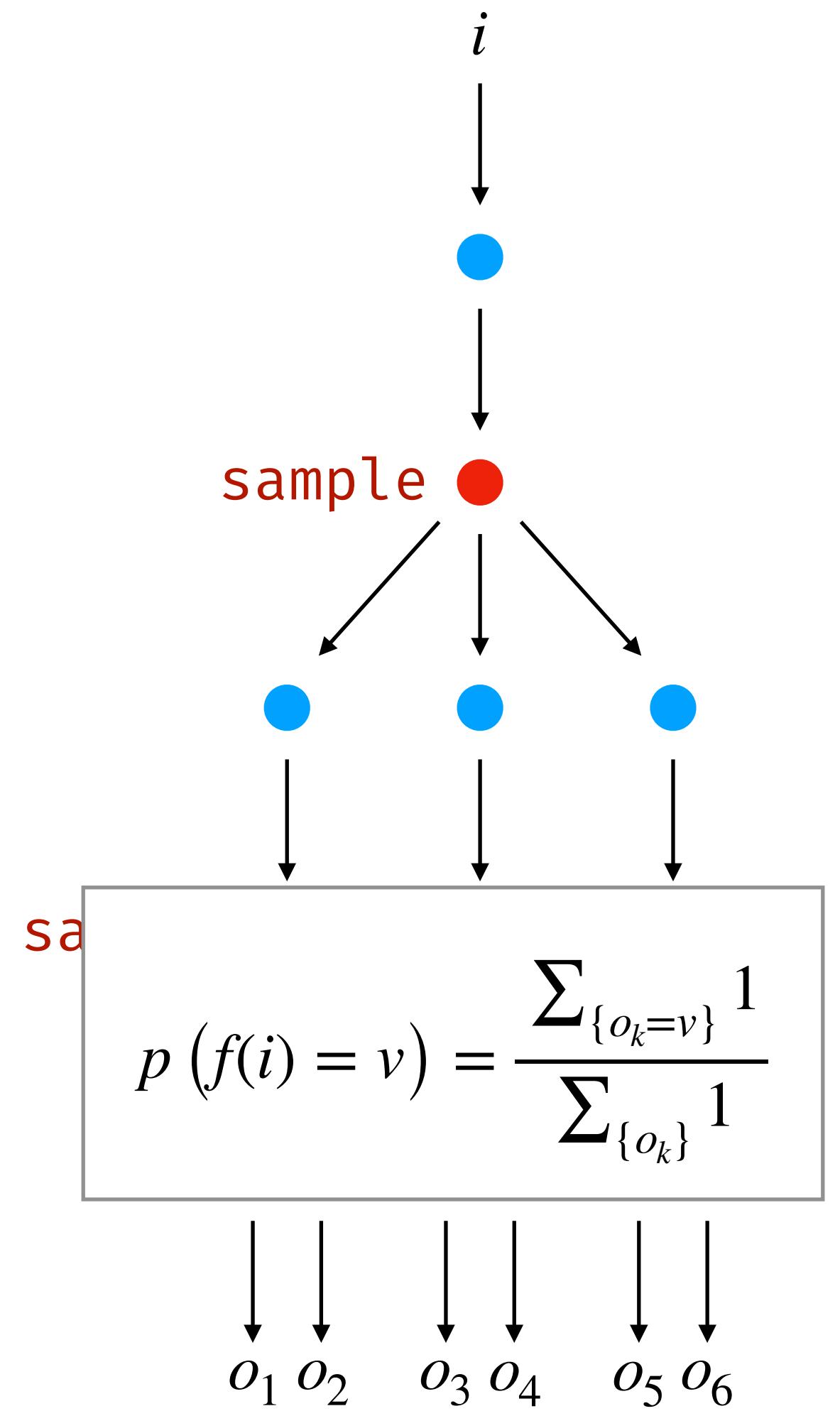
assume

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

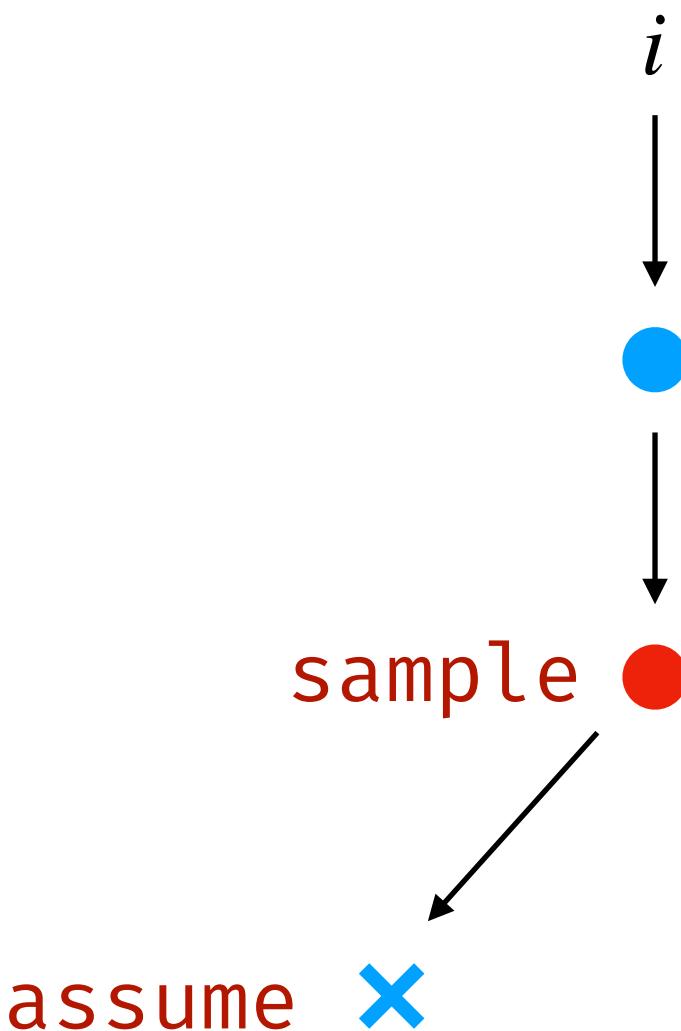
program



sample

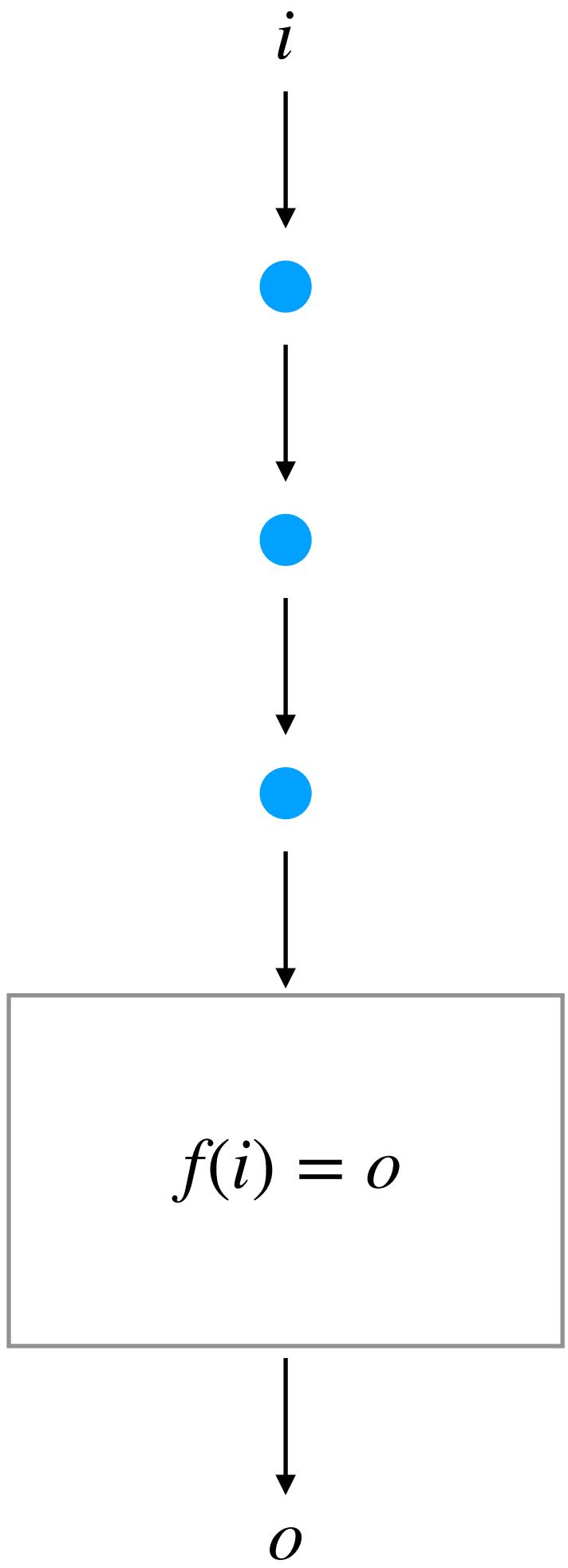


assume

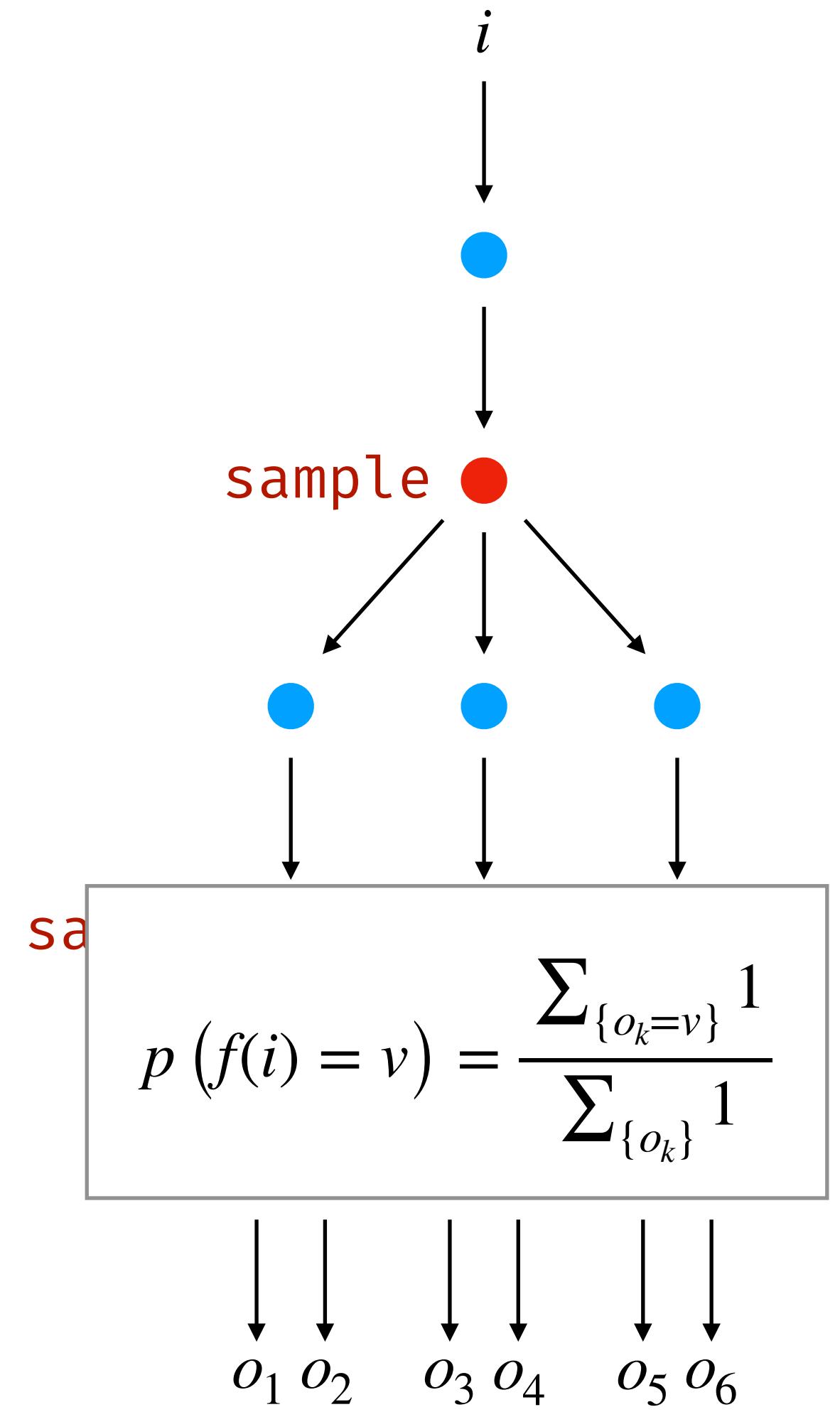


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

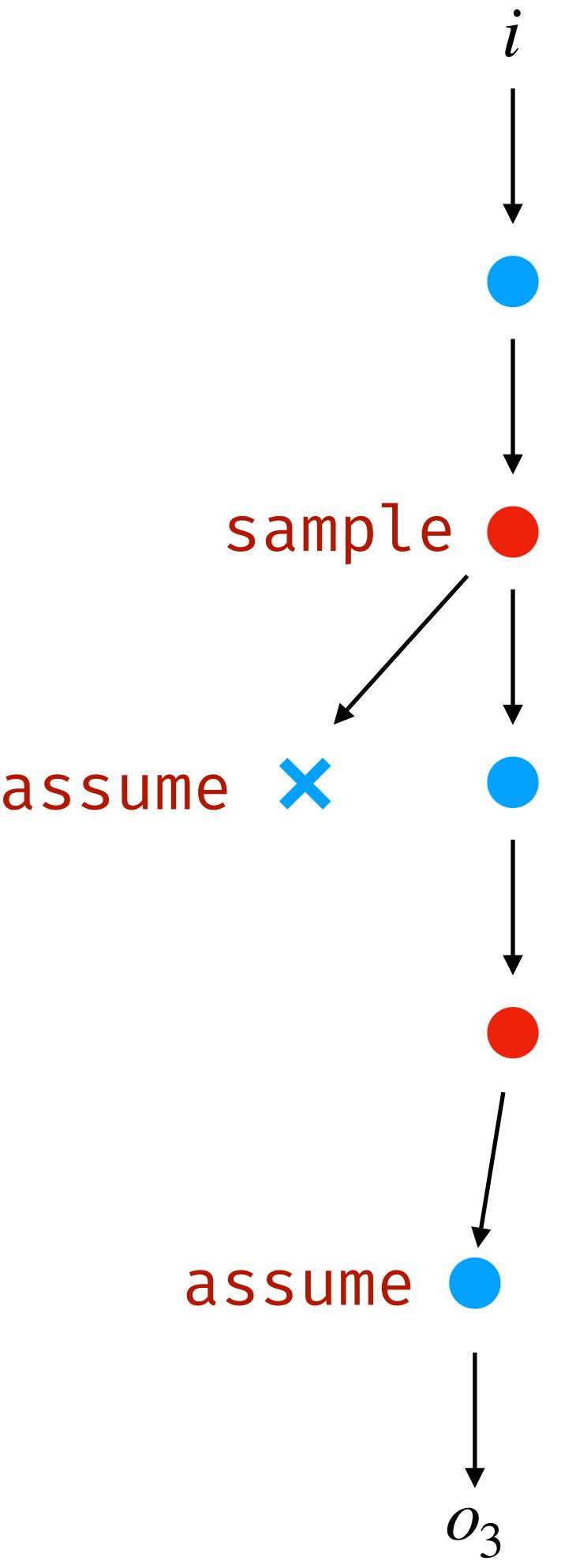
program



sample

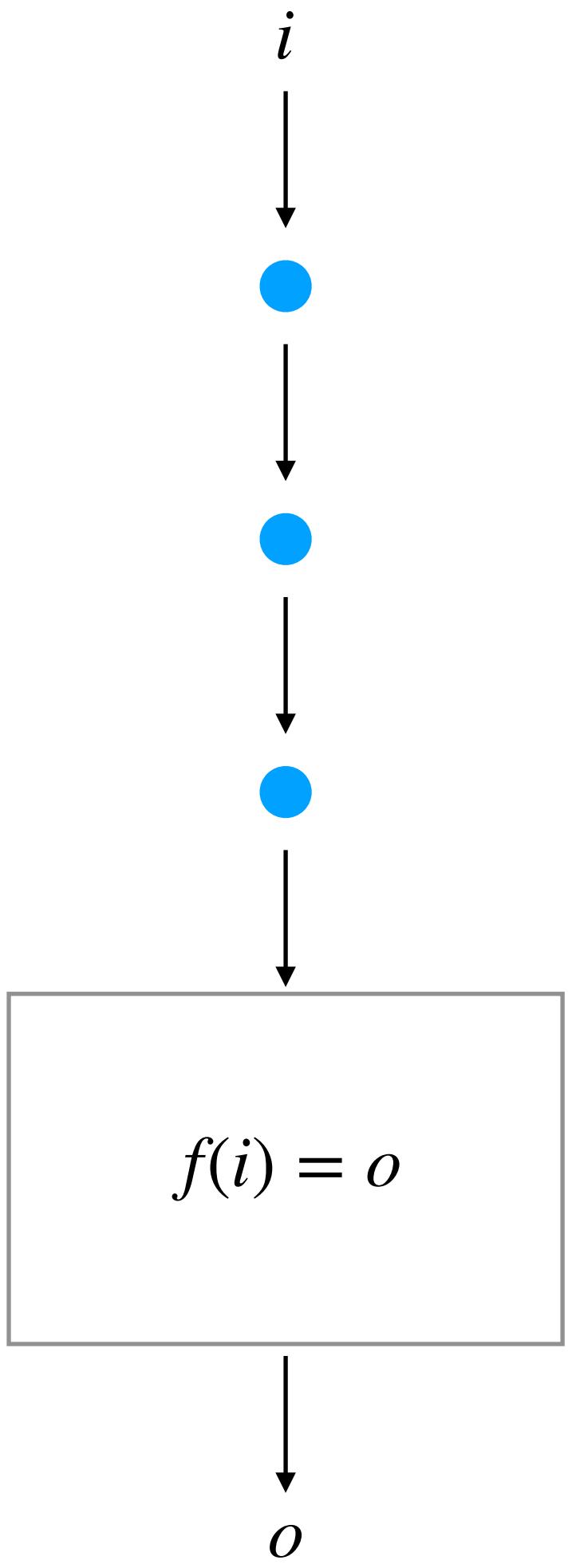


assume

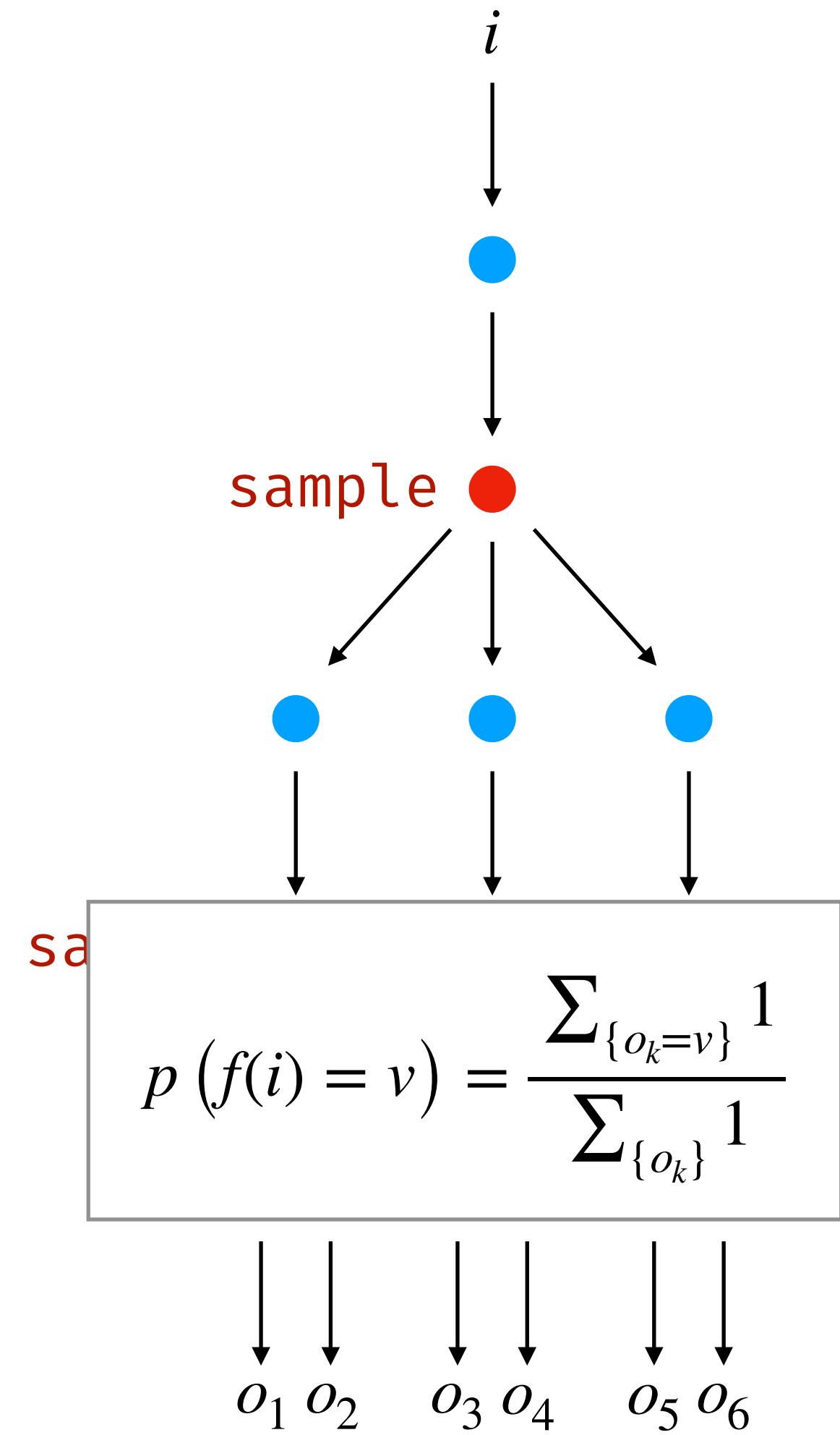


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

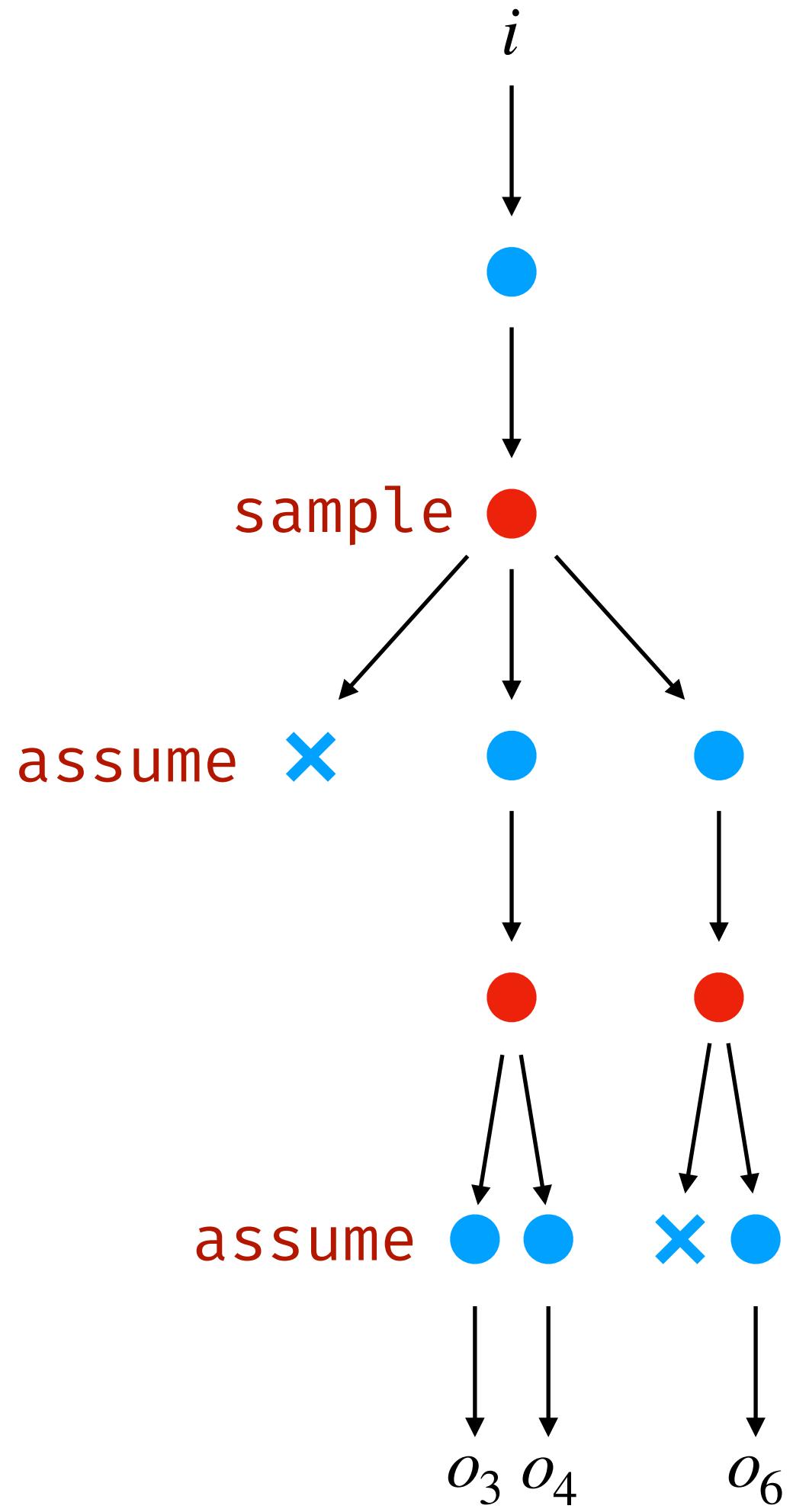
program



sample

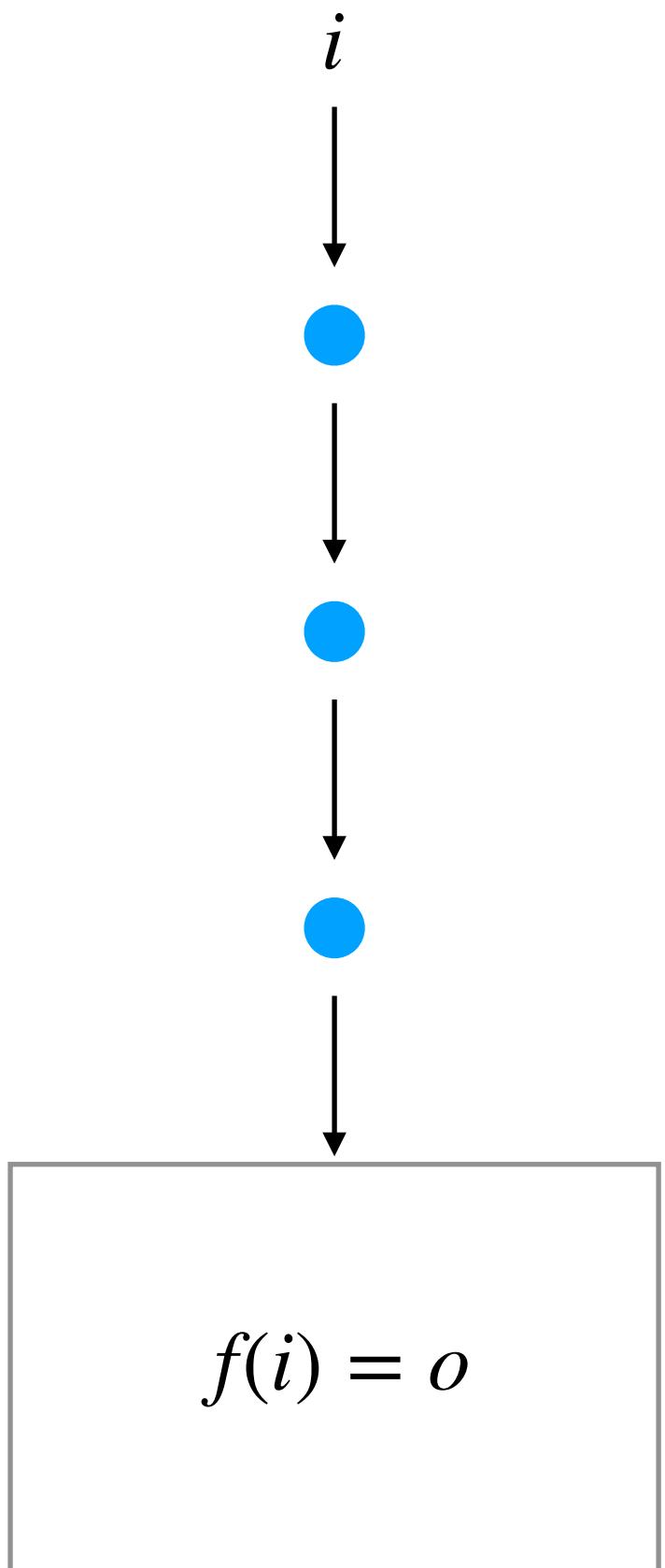


assume

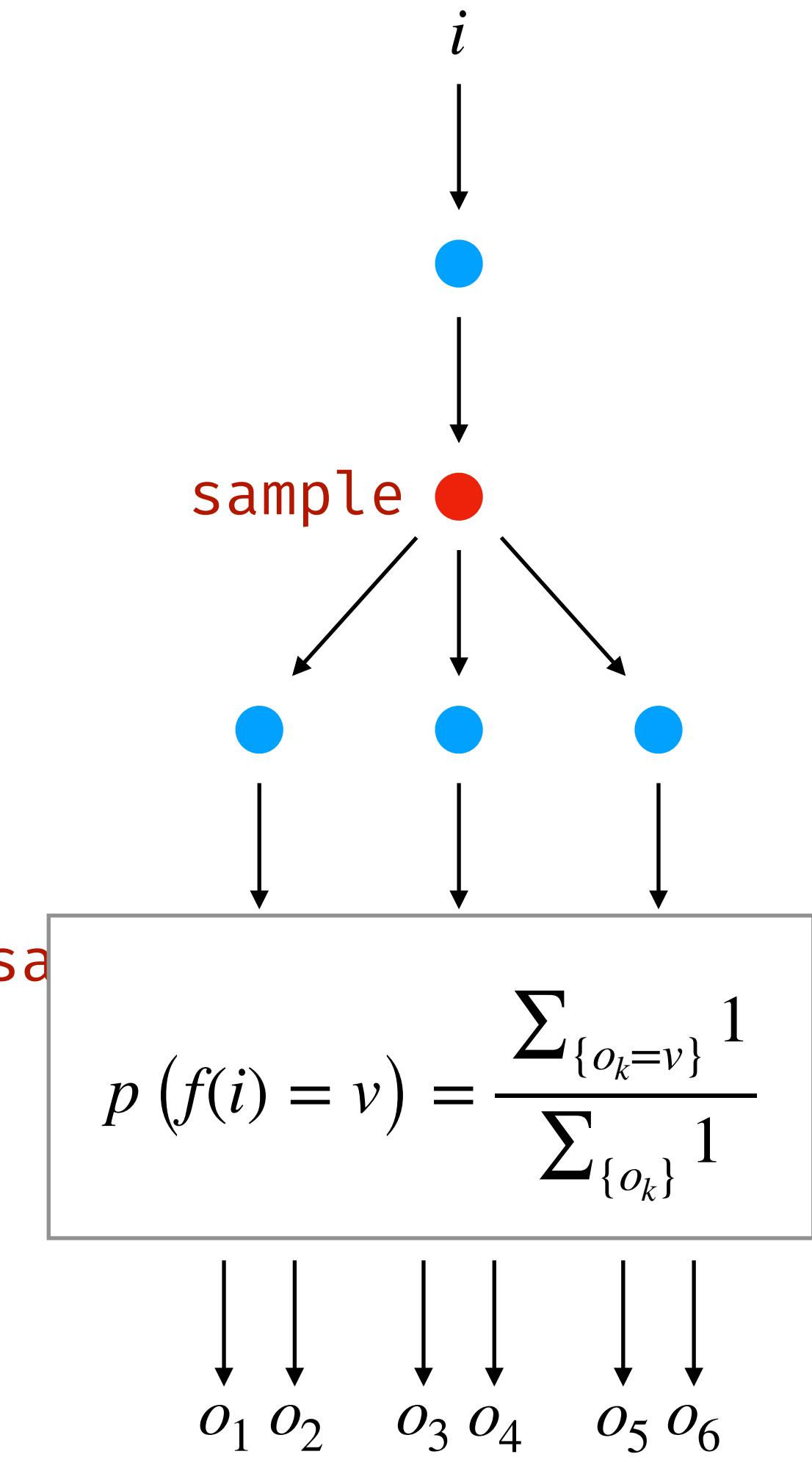


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

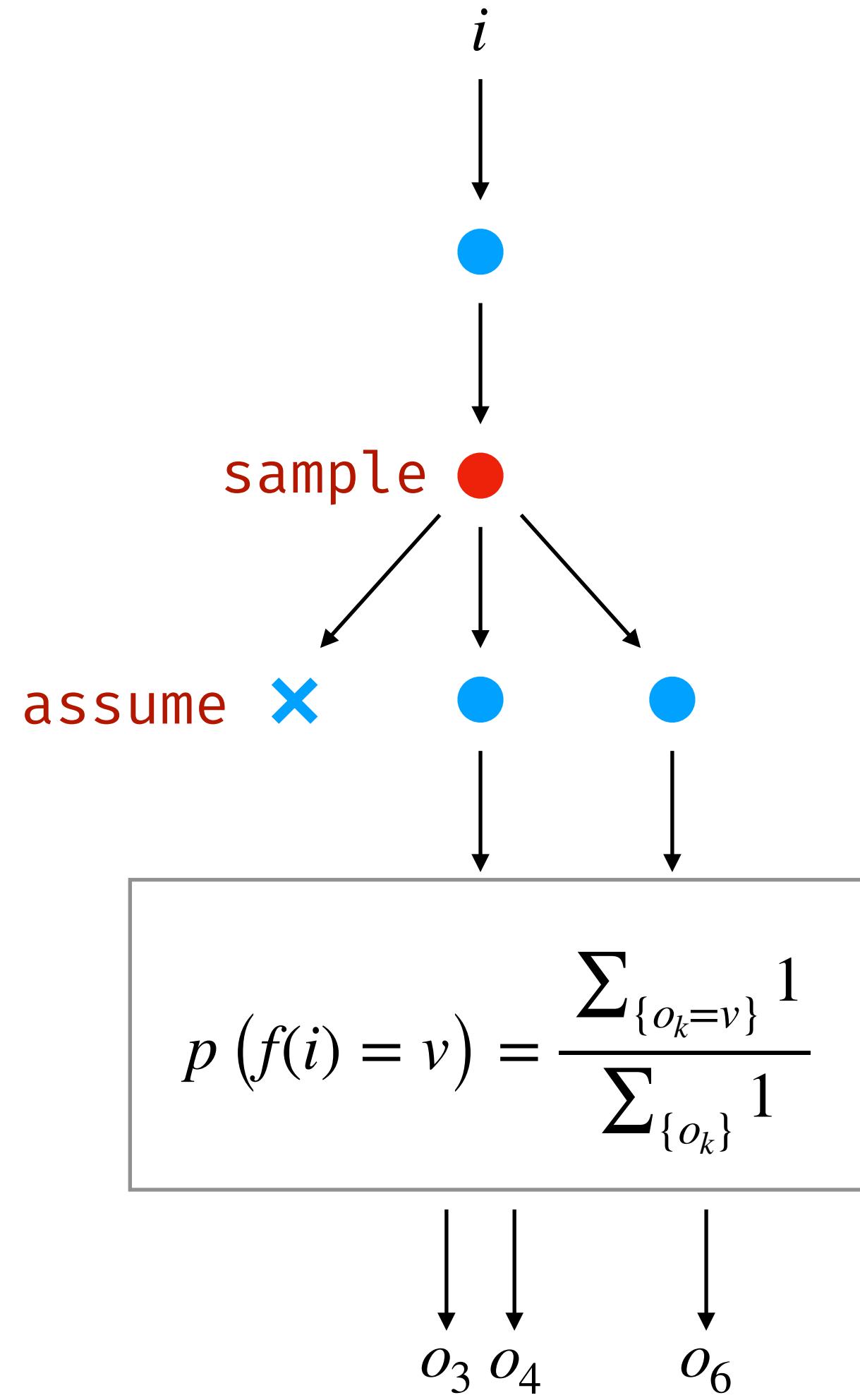
program



sample

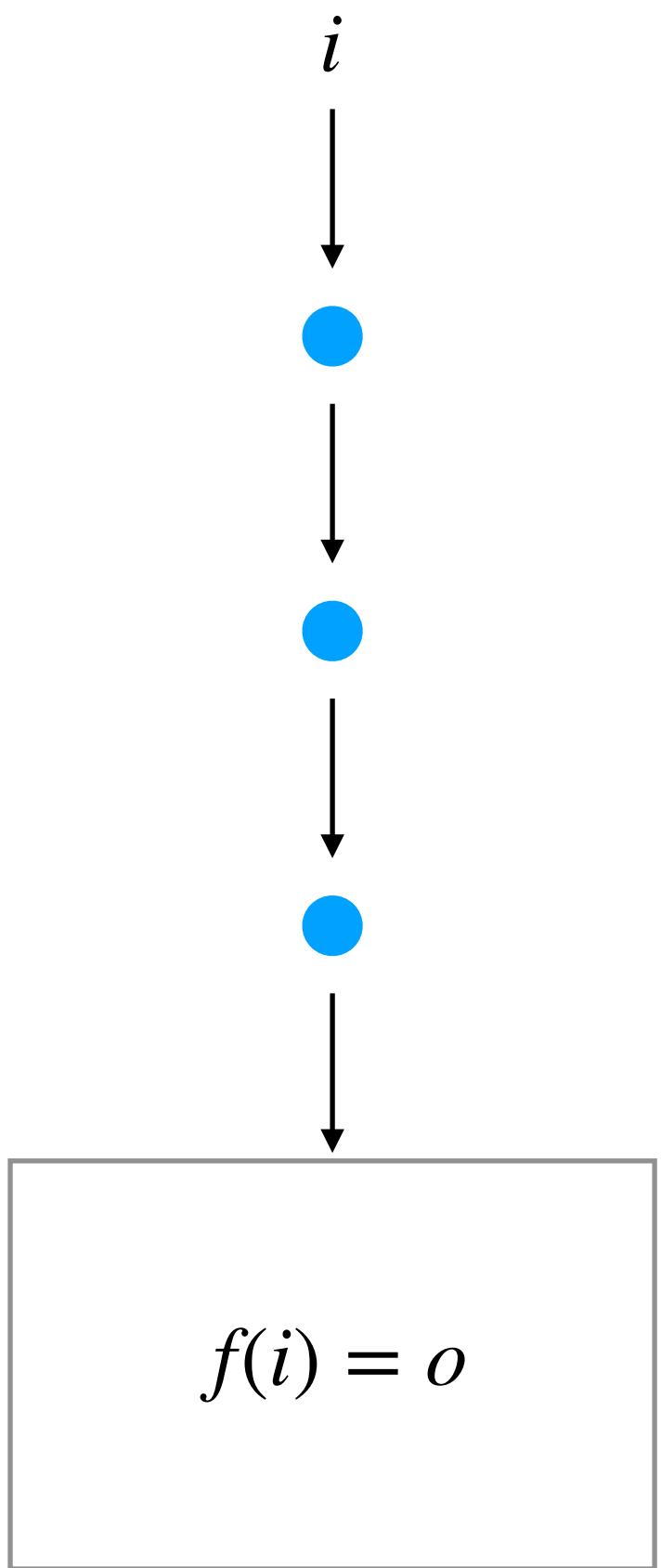


assume

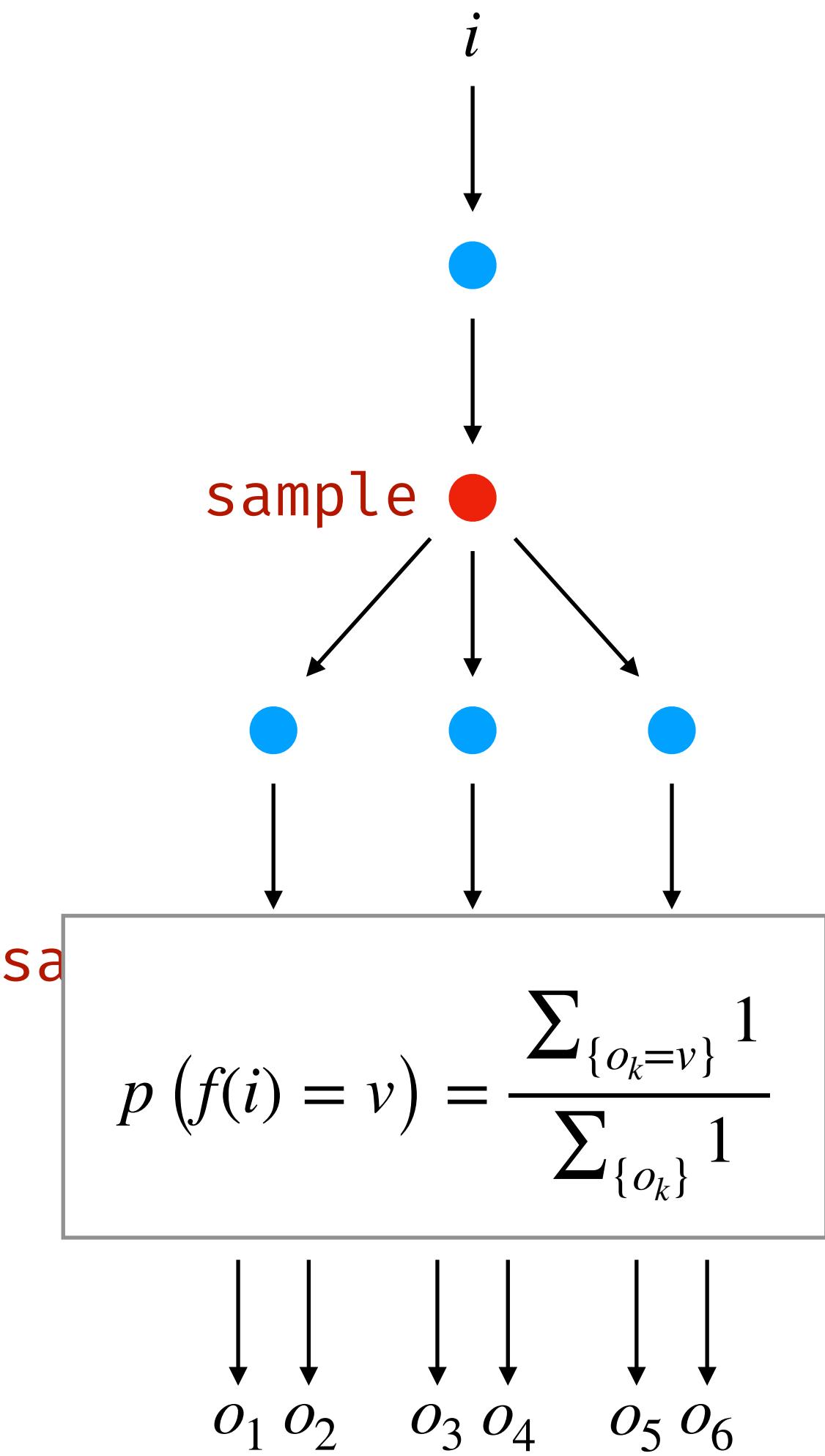


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

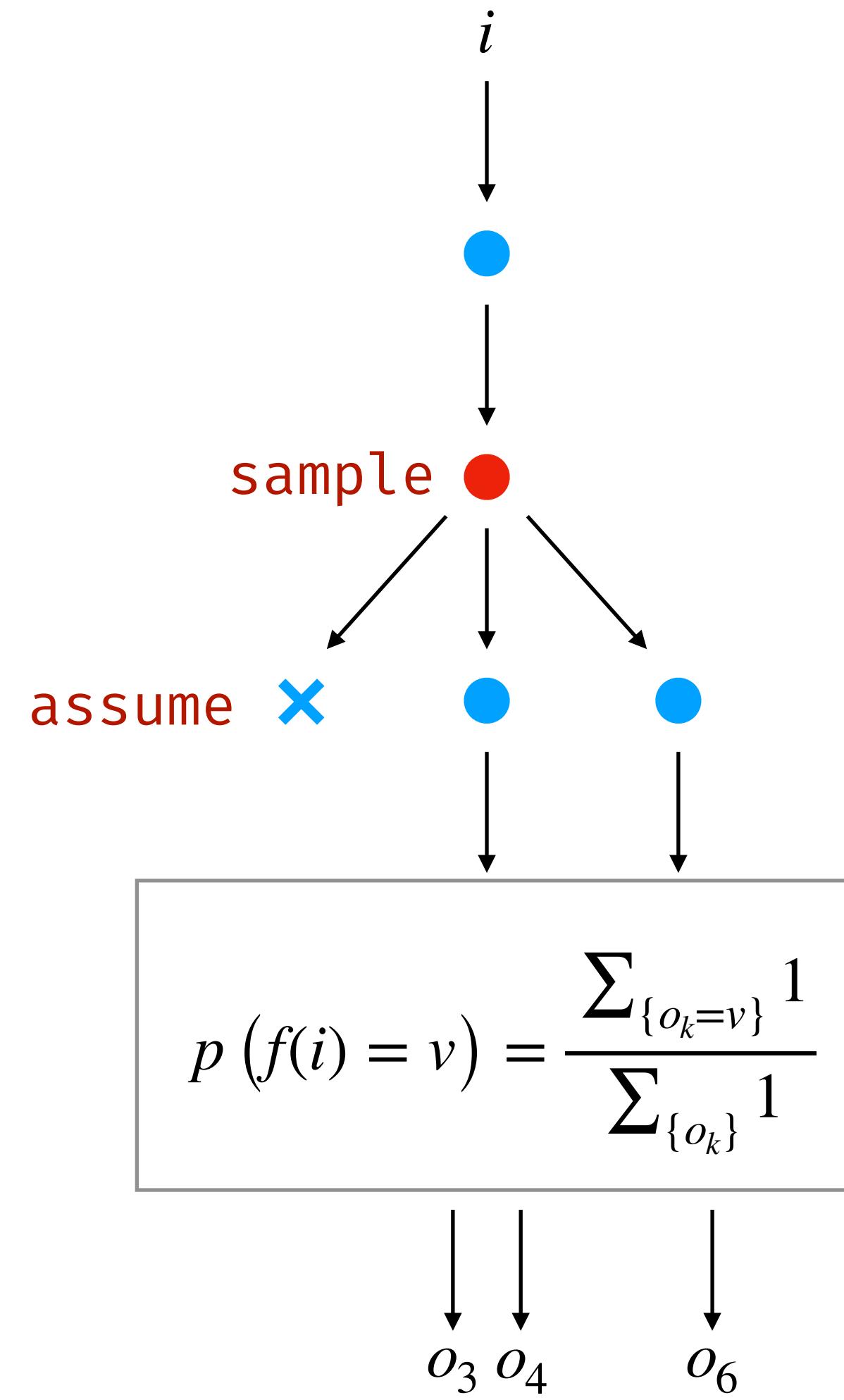
program



sample



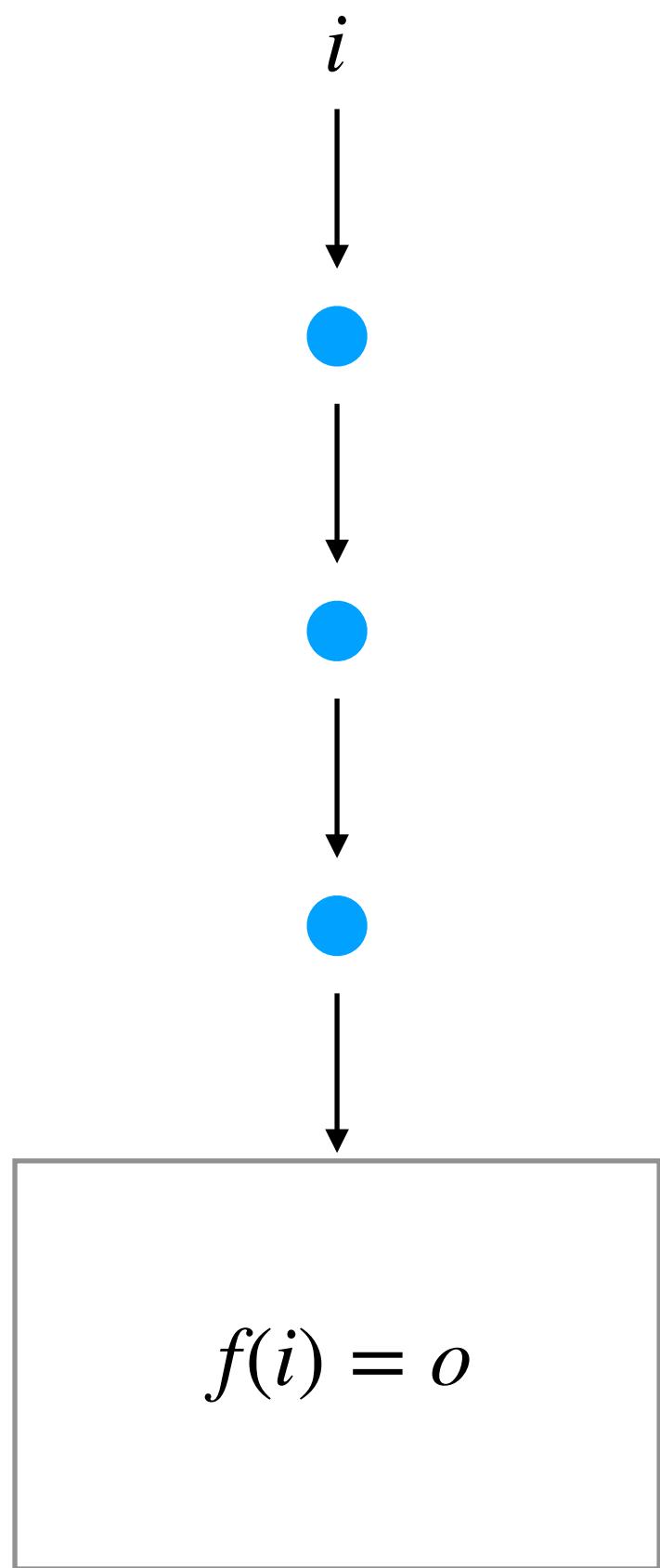
assume



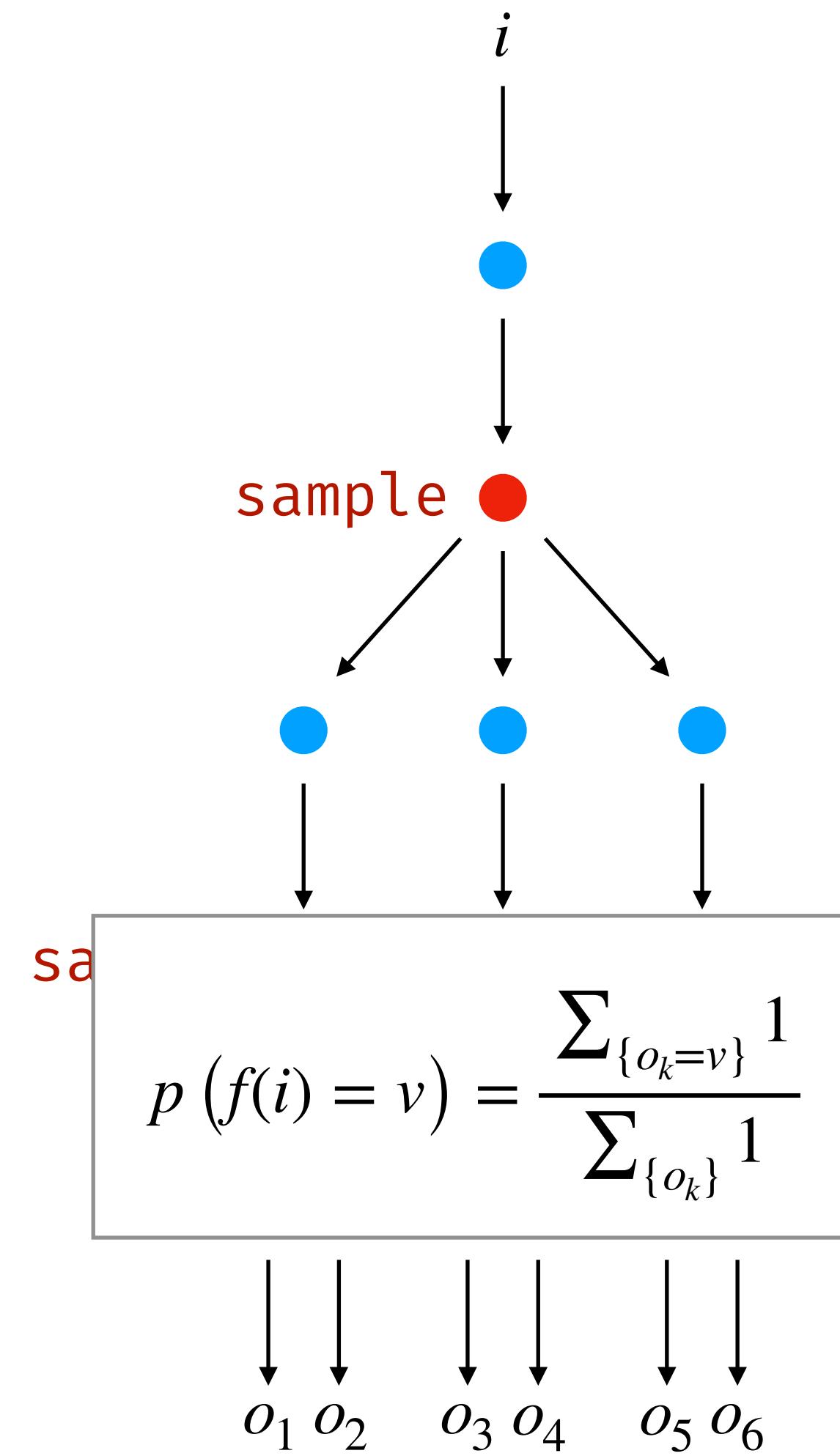
factor

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

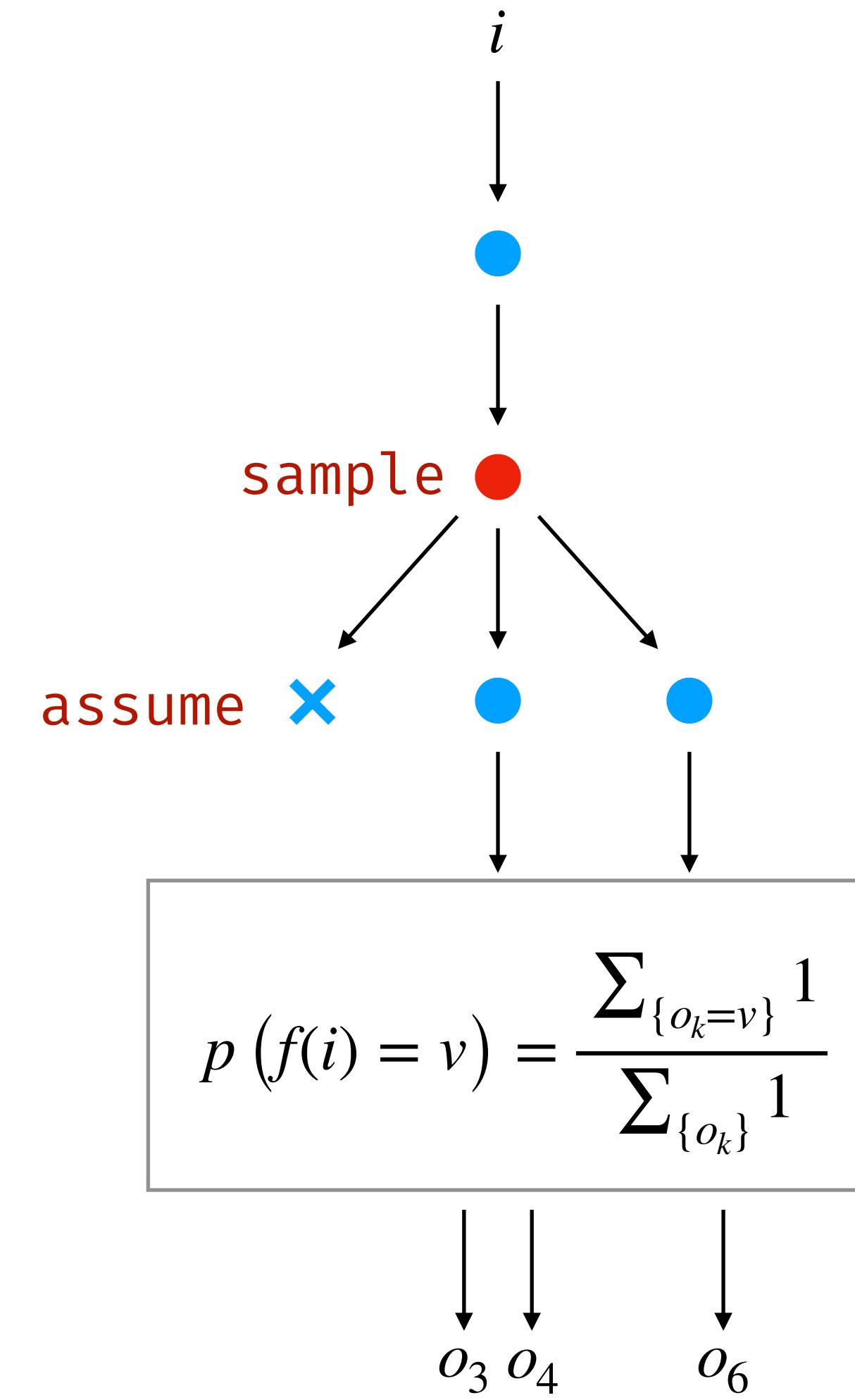
program



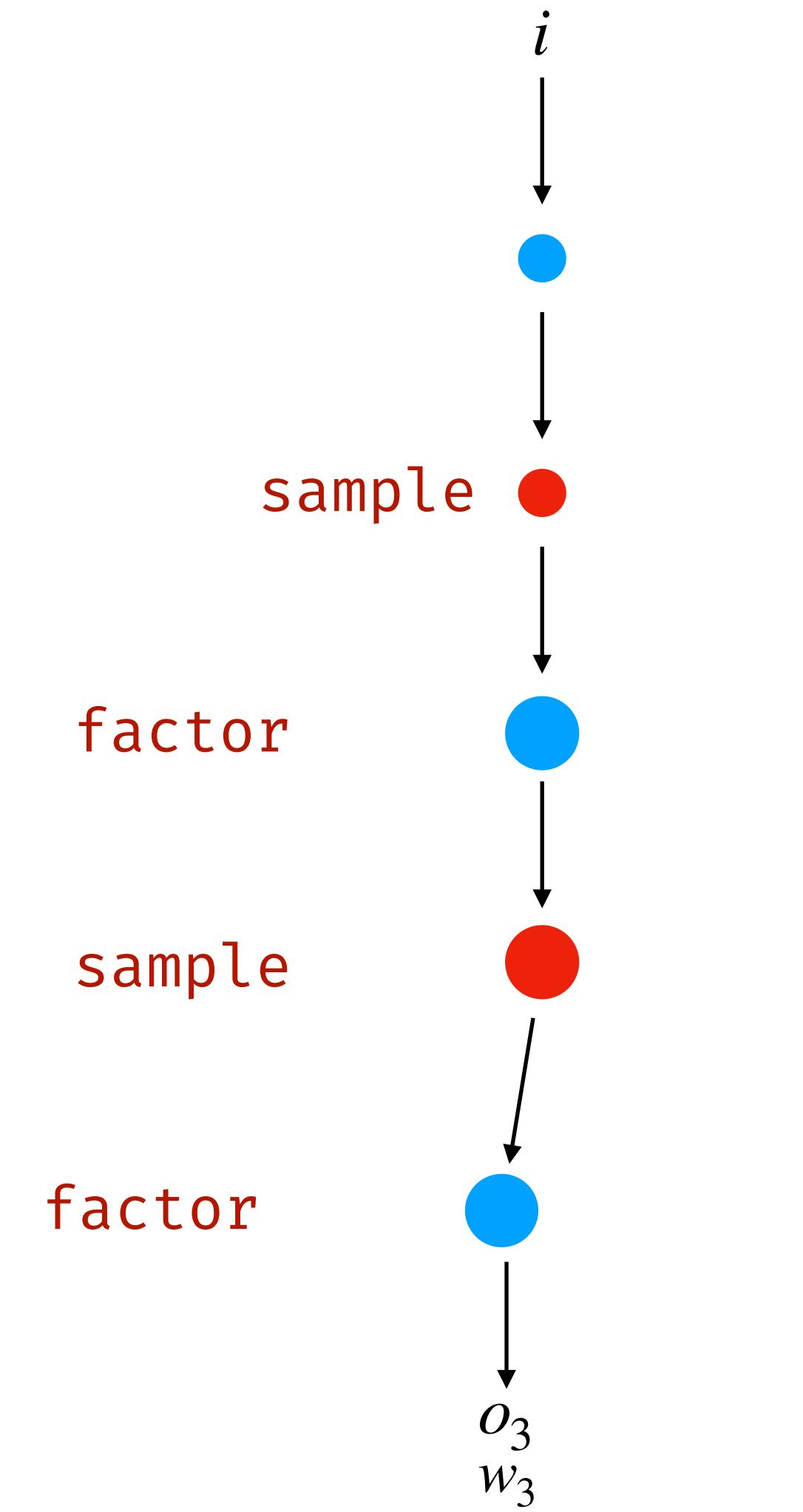
sample



assume

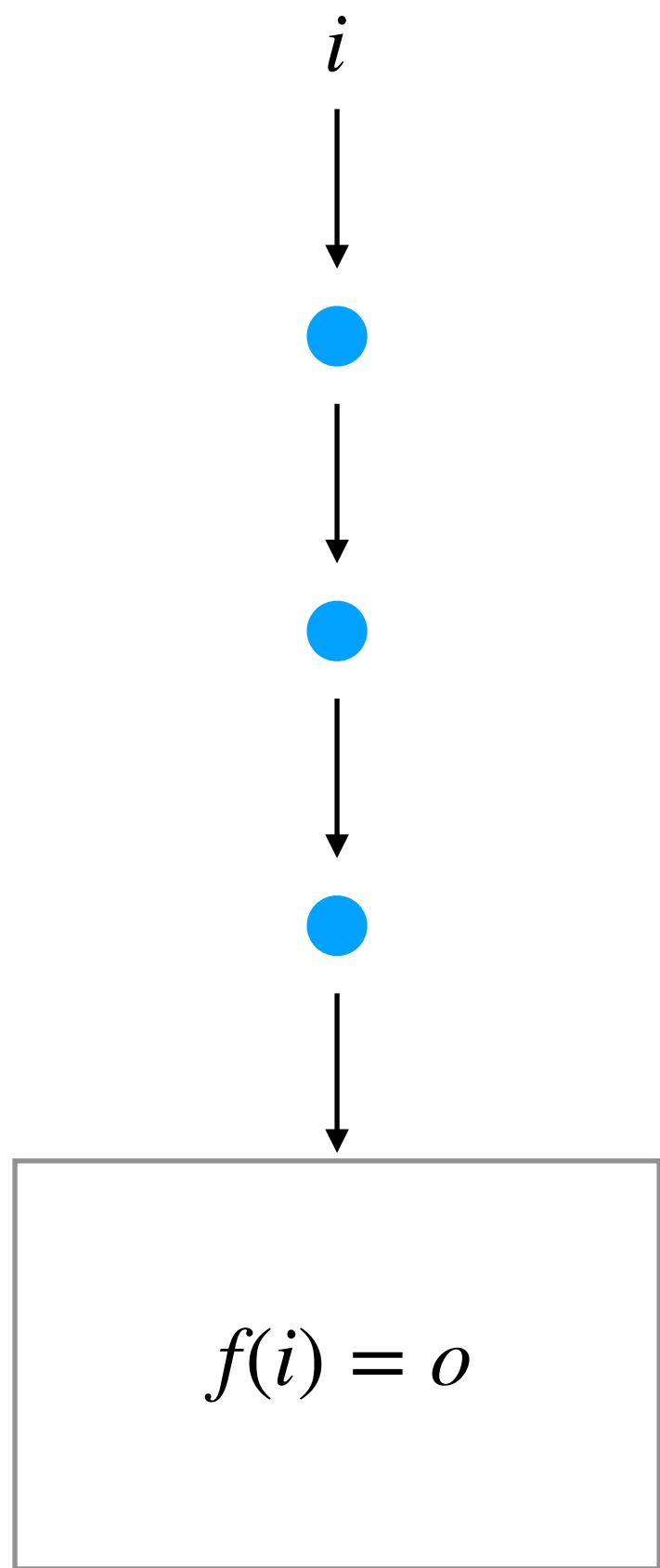


factor

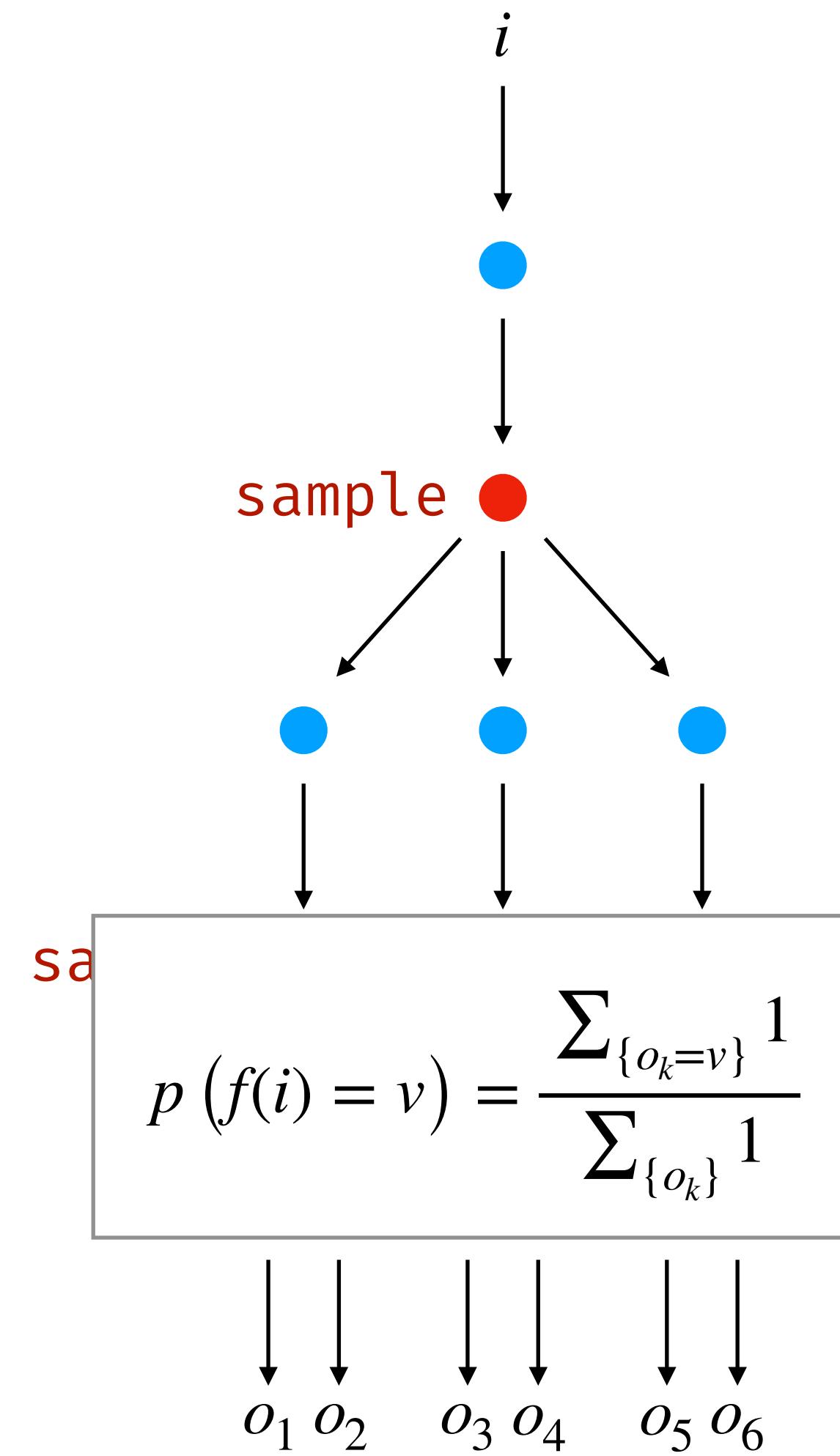


infer : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

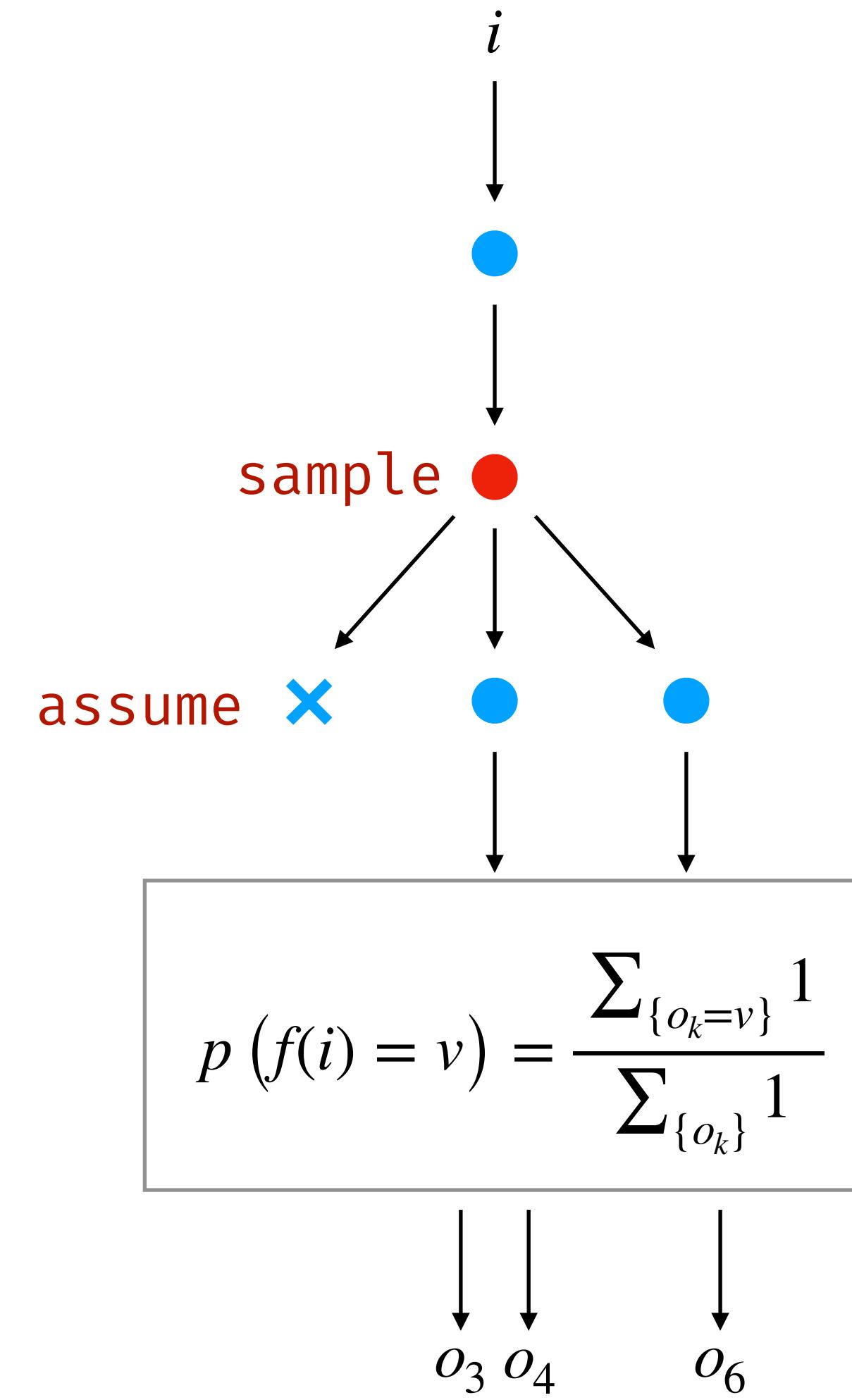
program



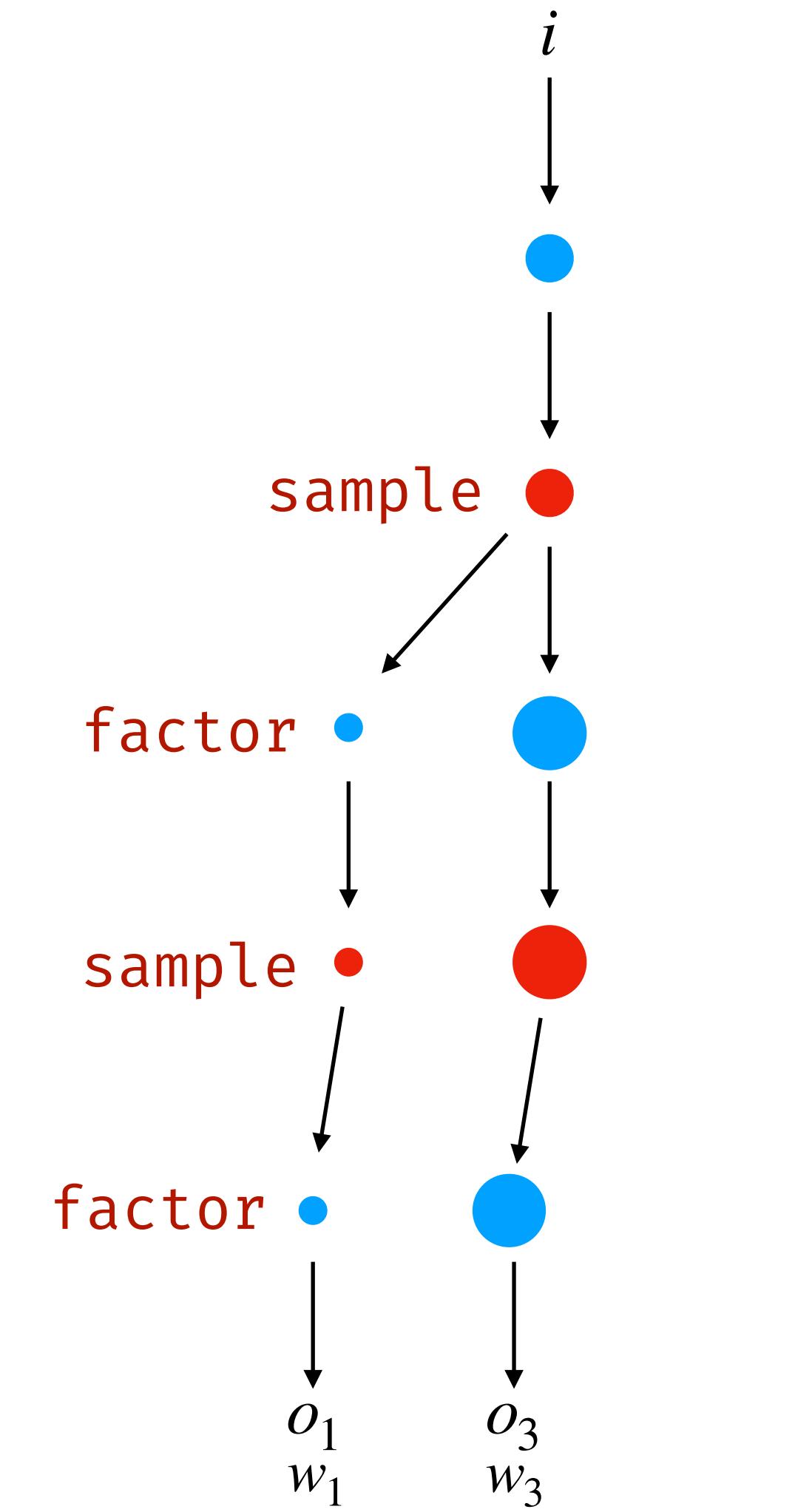
sample



assume

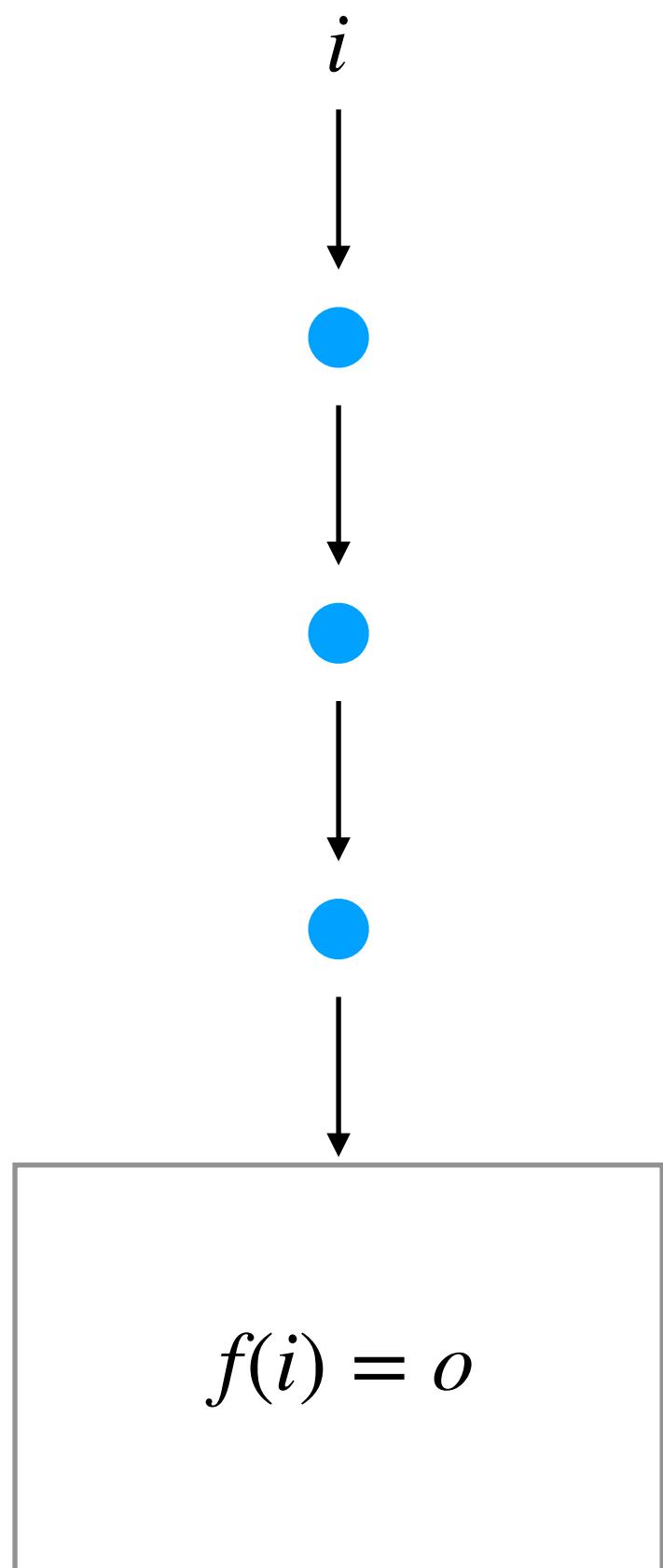


factor

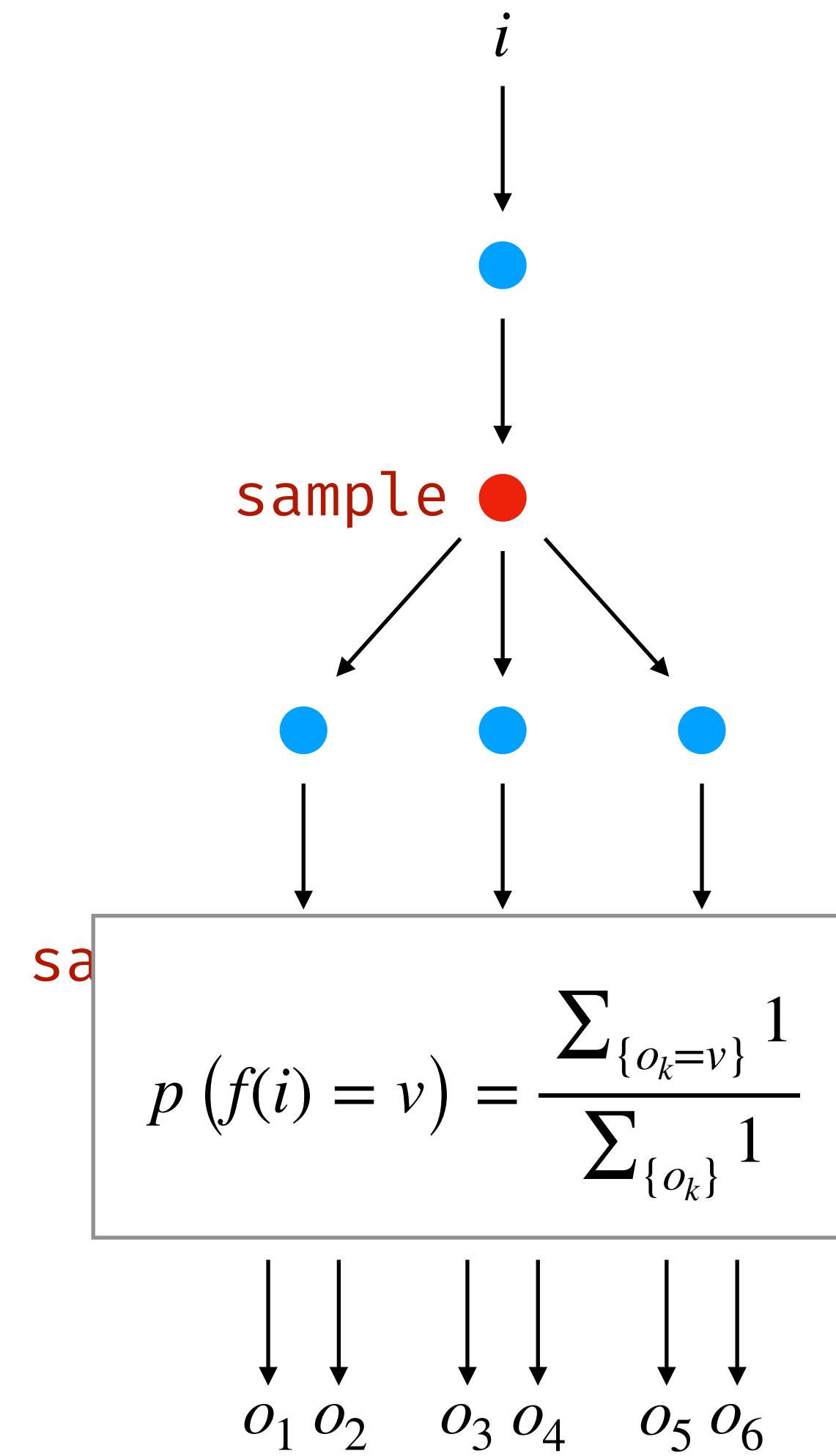


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

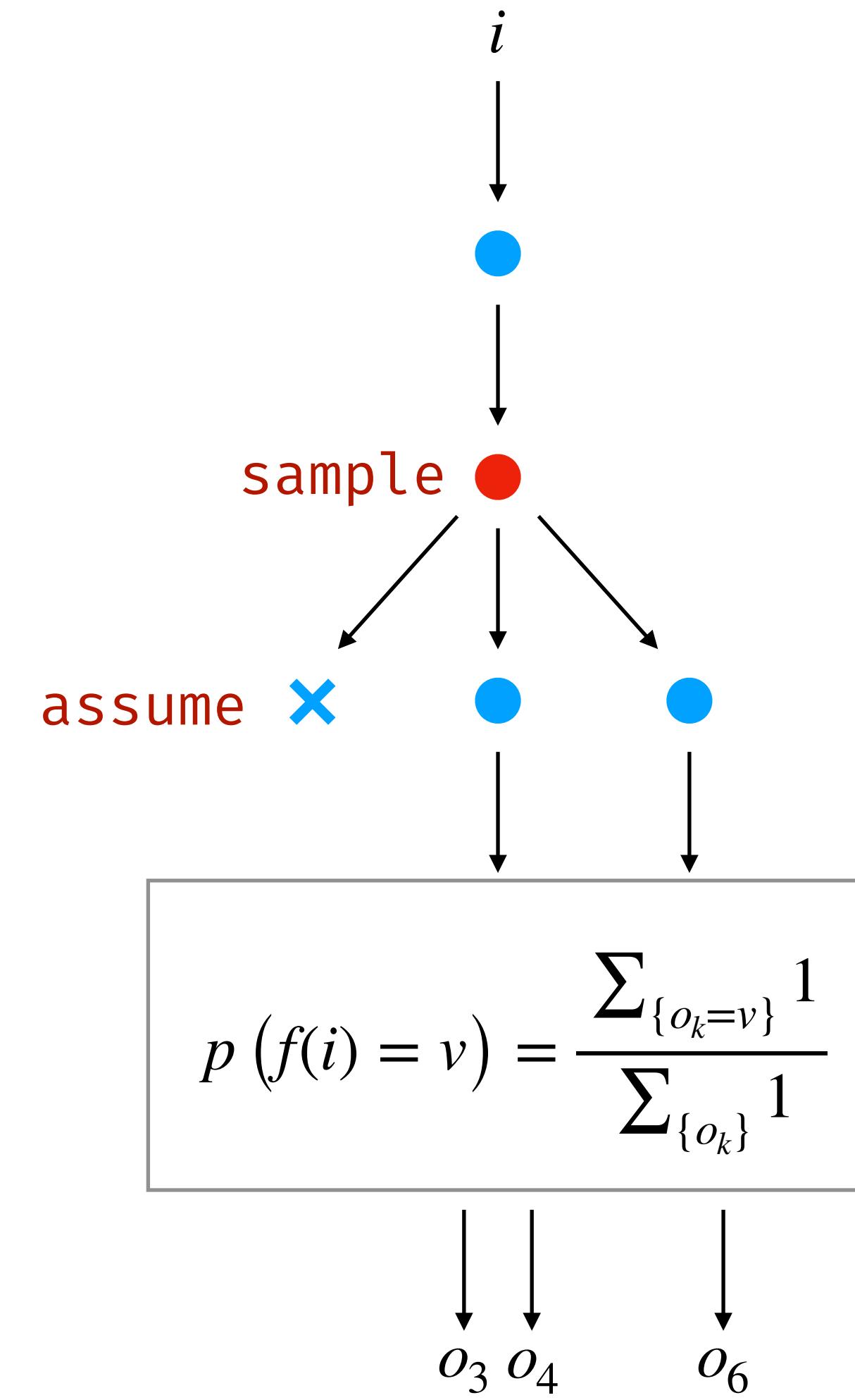
program



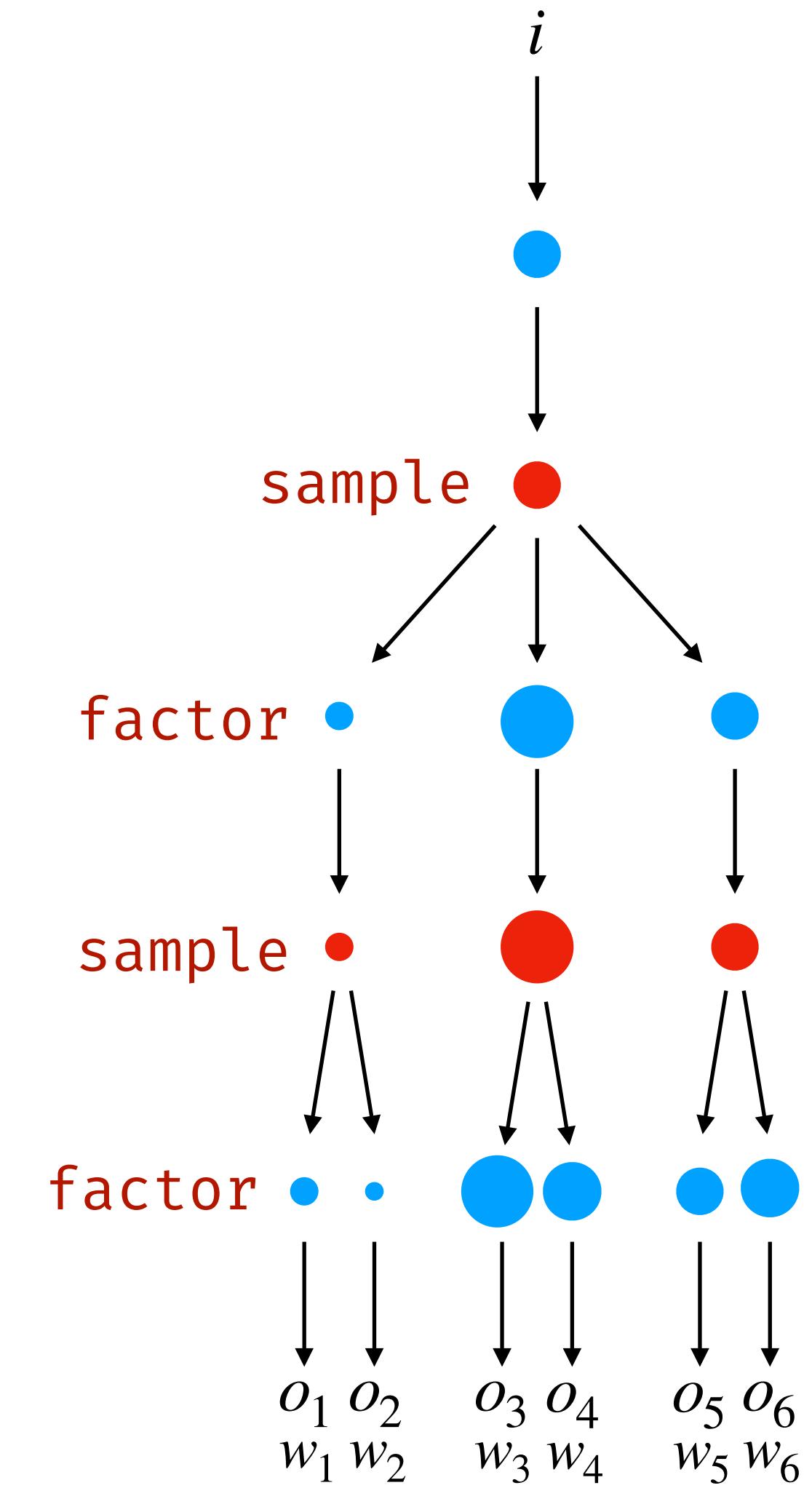
sample



assume

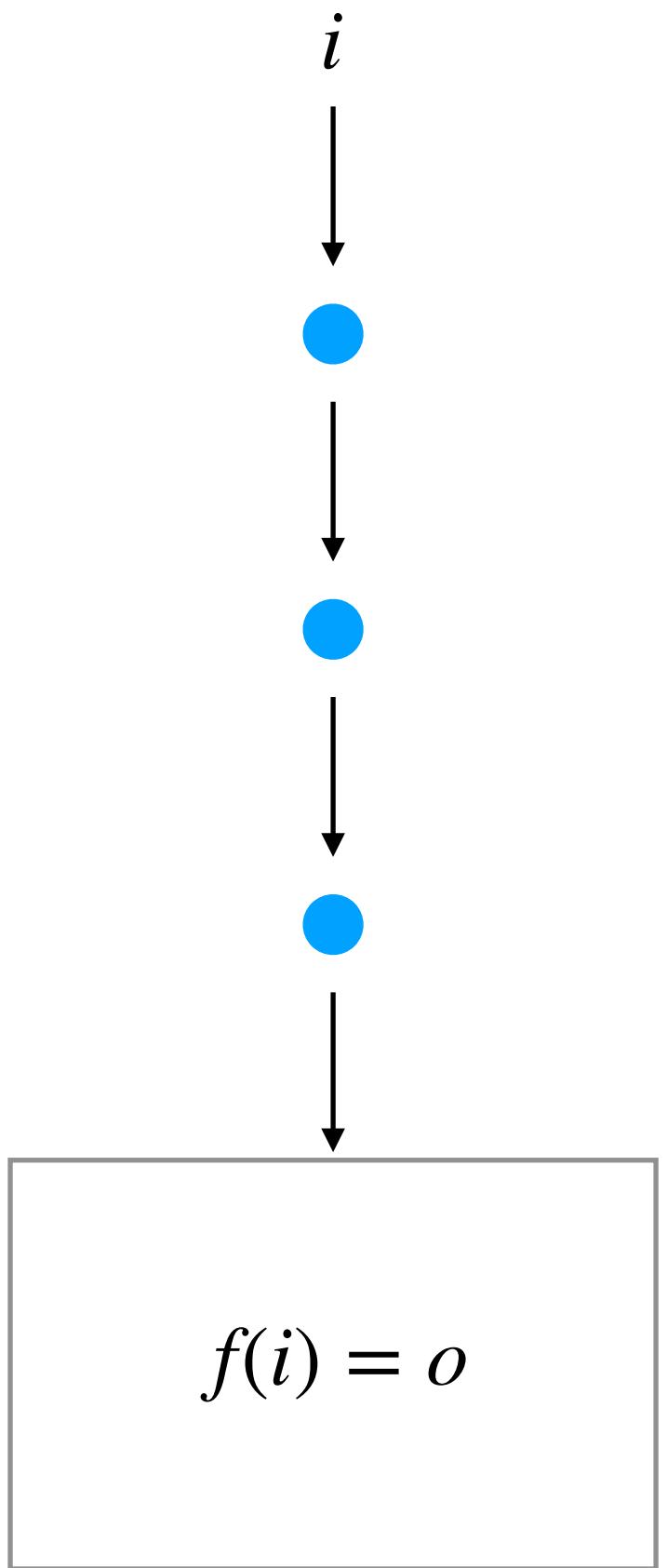


factor

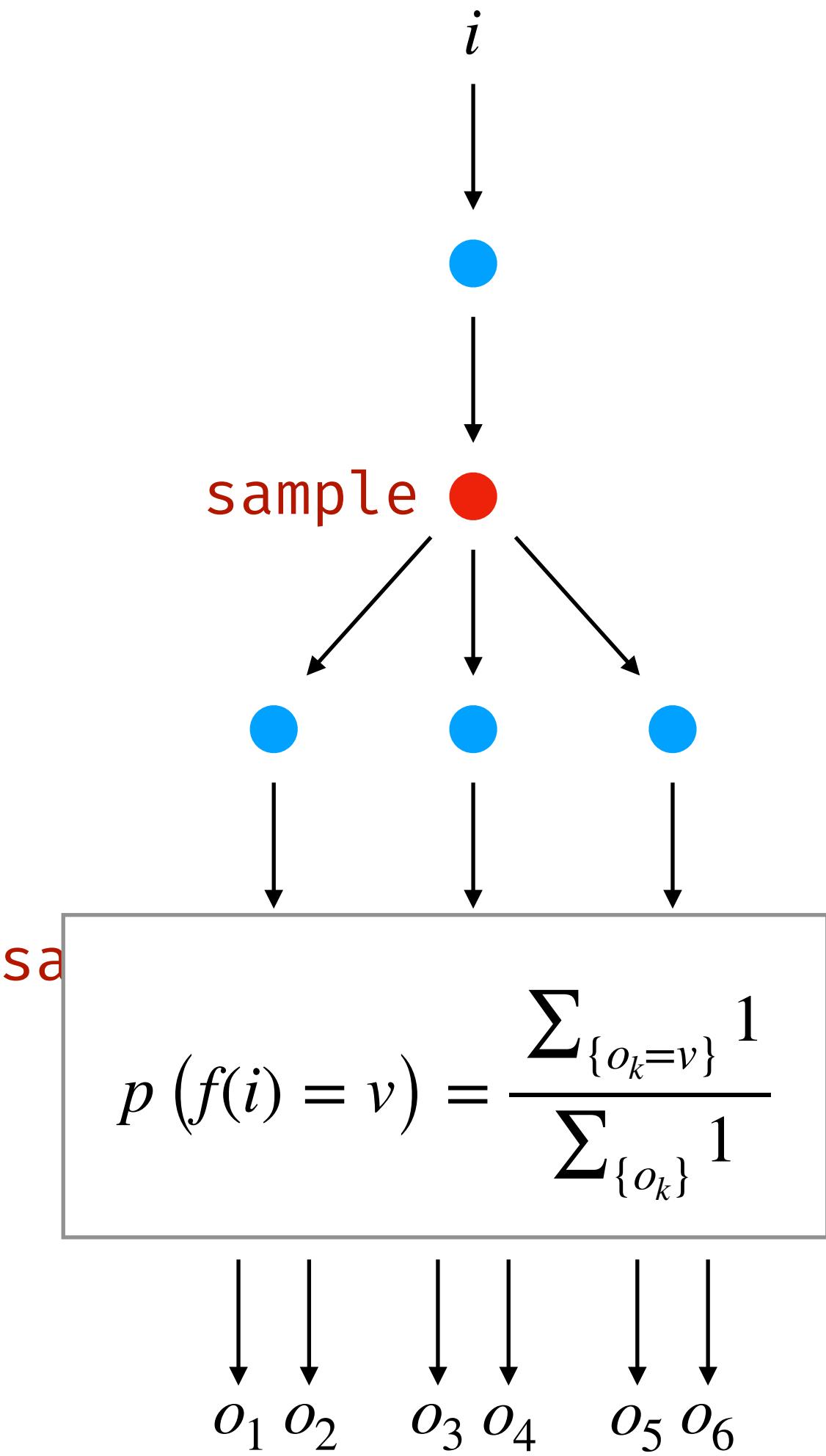


`infer` : $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ dist

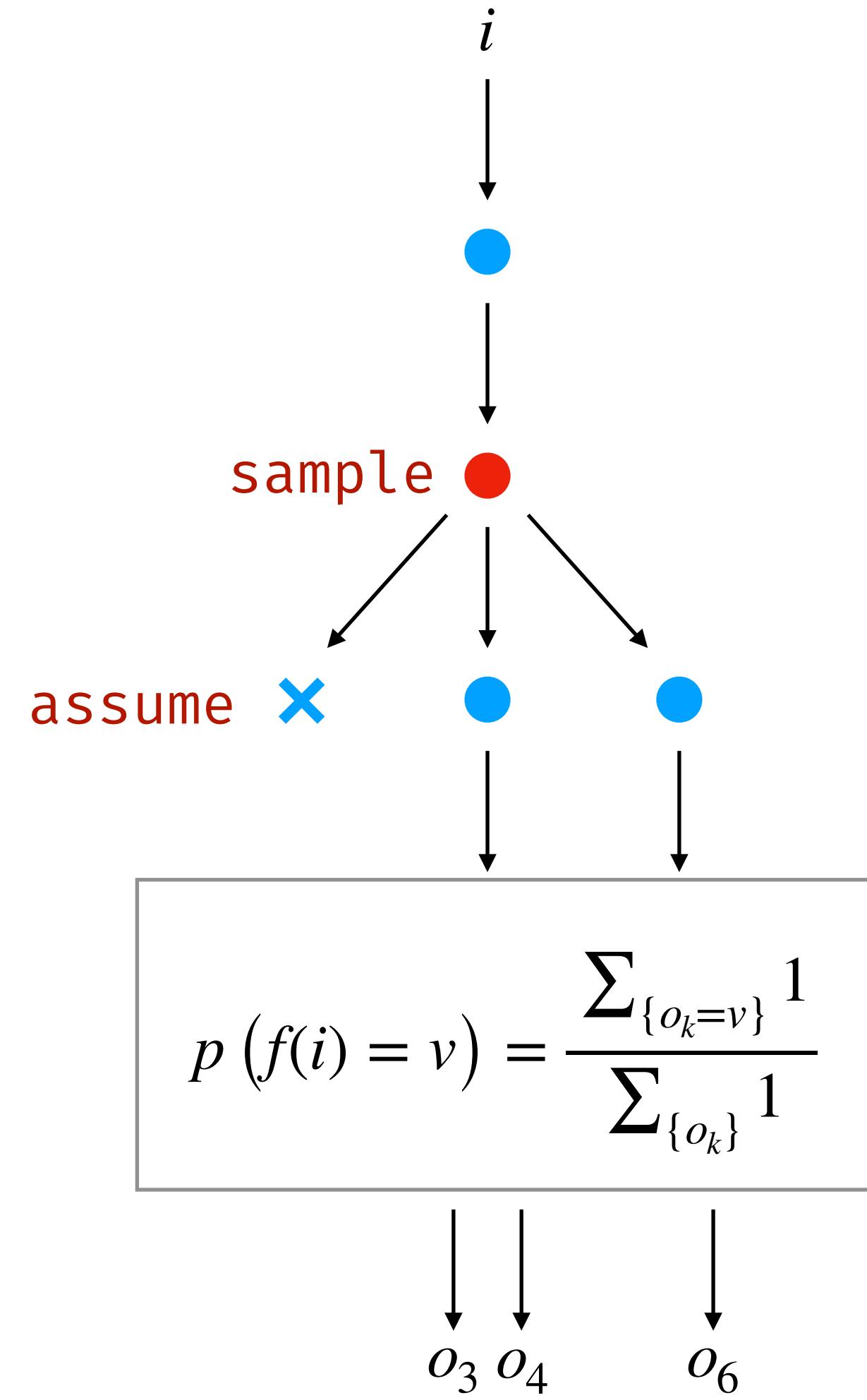
program



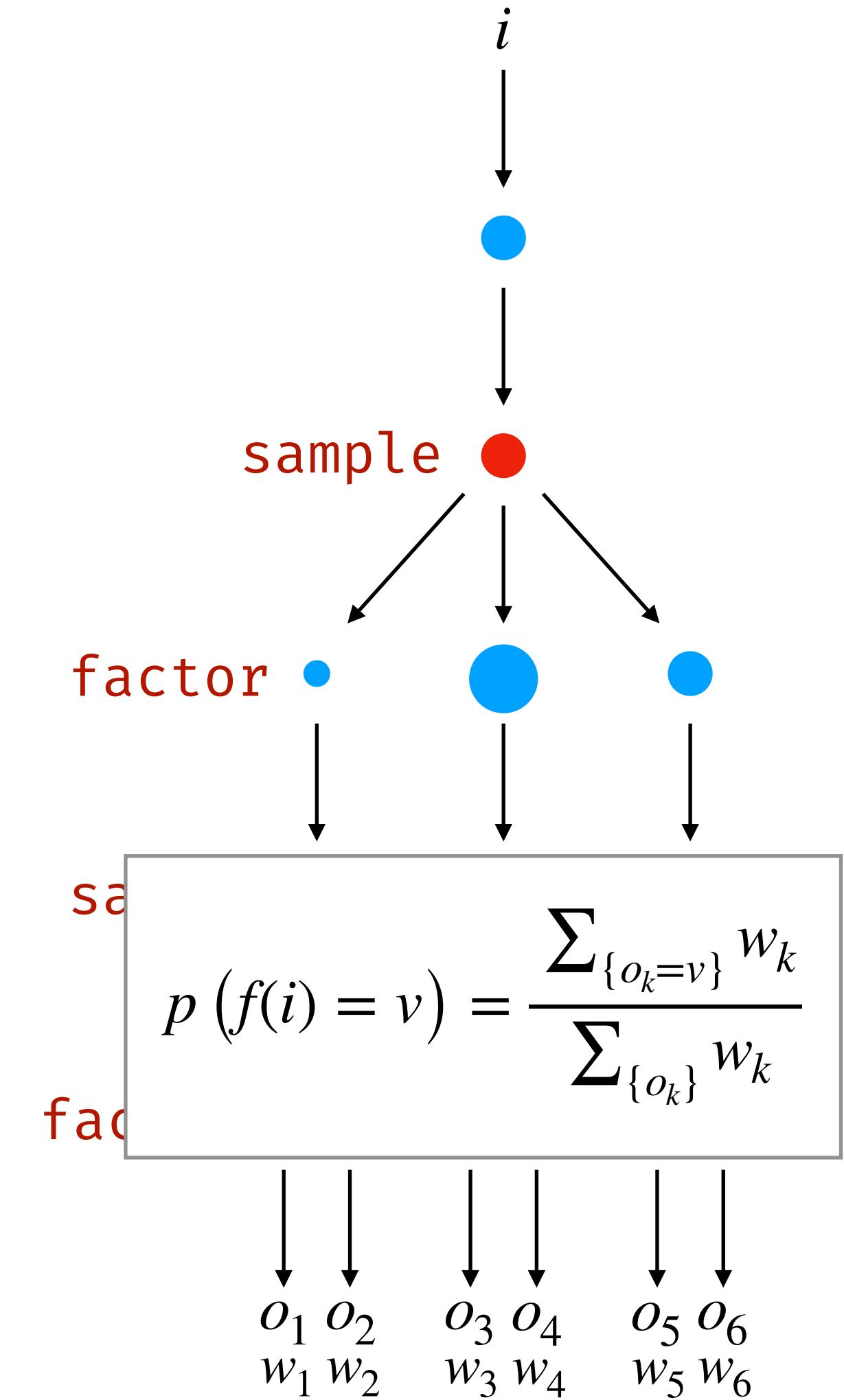
sample



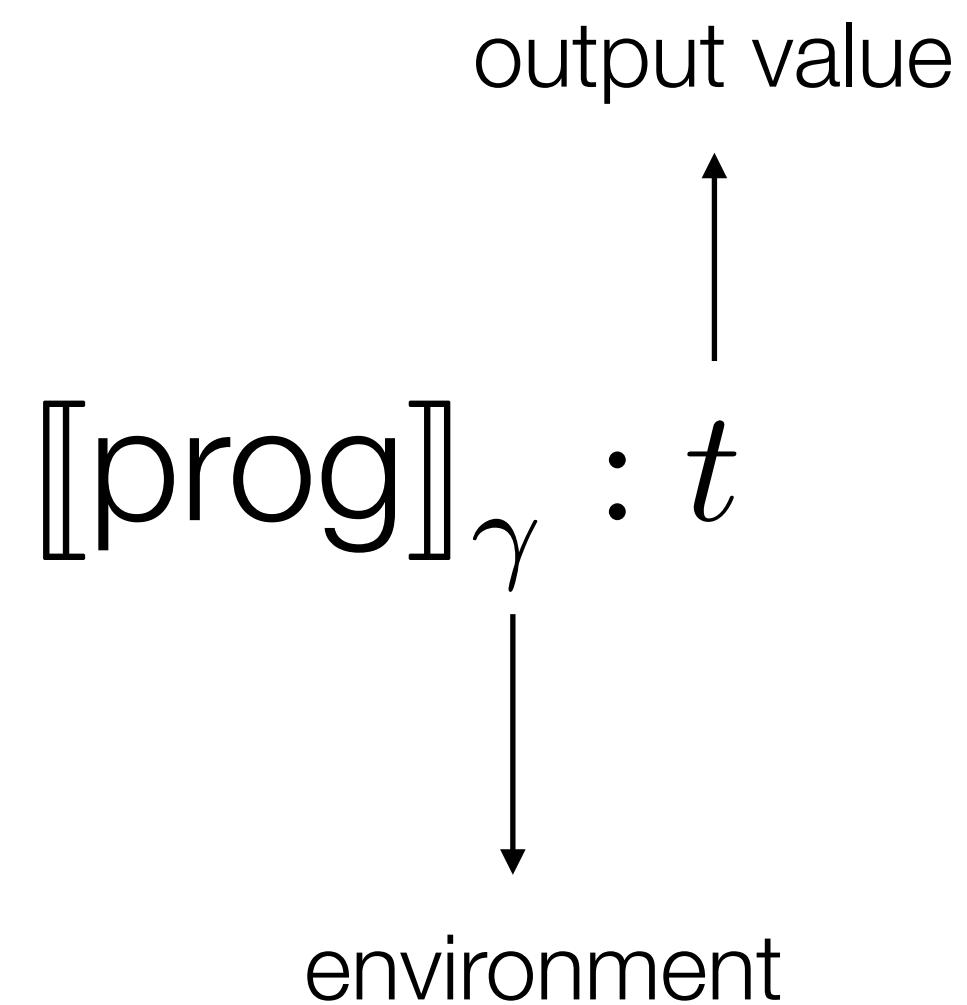
assume



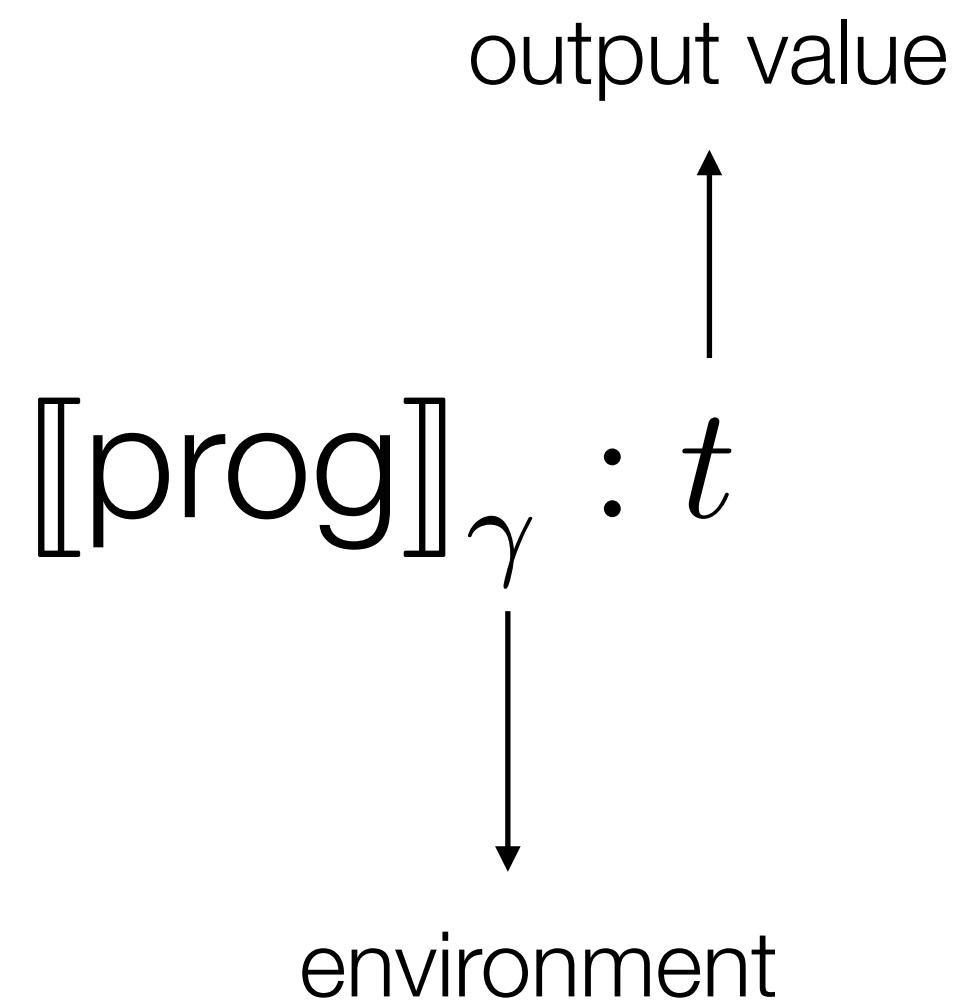
factor



Deterministic vs. probabilistic semantics



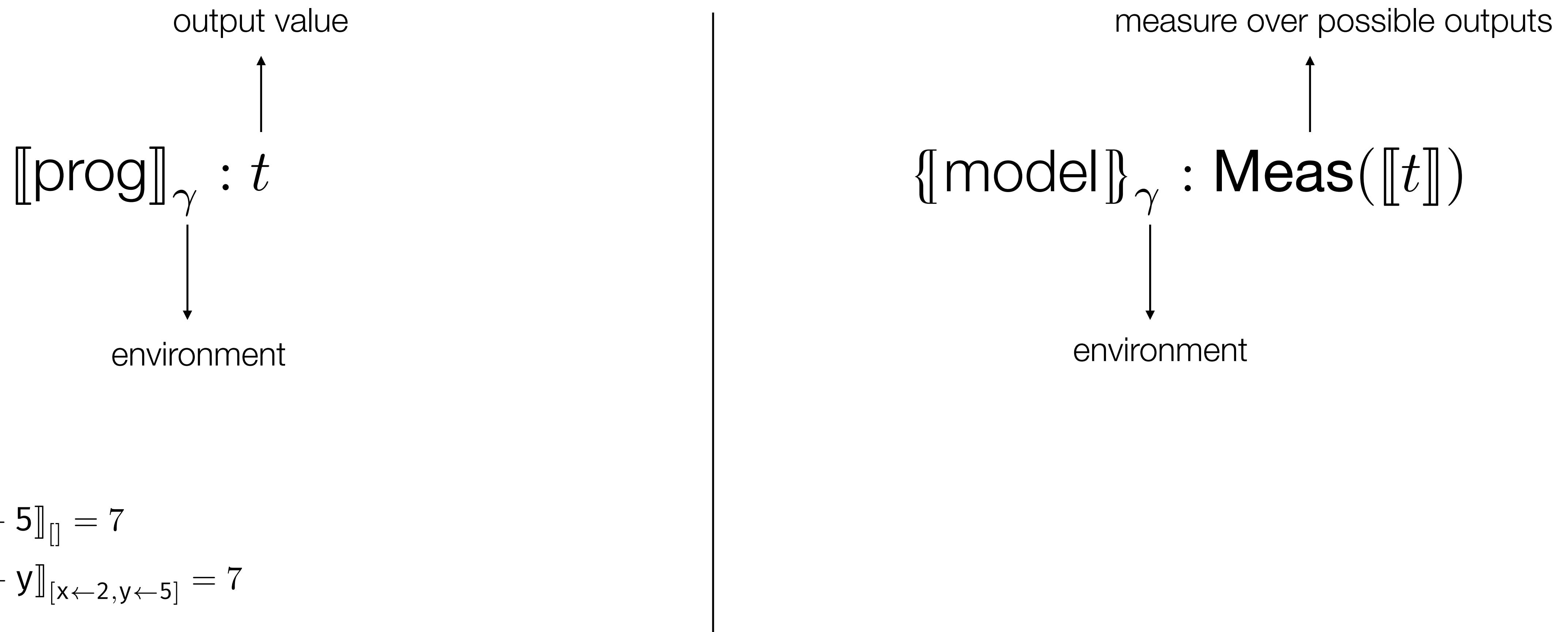
Deterministic vs. probabilistic semantics



$$\llbracket 2 + 5 \rrbracket_{[]} = 7$$

$$\llbracket x + y \rrbracket_{[x \leftarrow 2, y \leftarrow 5]} = 7$$

Deterministic vs. probabilistic semantics



Deterministic vs. probabilistic semantics

$$\begin{array}{c} \text{output value} \\ \uparrow \\ \llbracket \text{prog} \rrbracket_{\gamma} : t \\ \downarrow \\ \text{environment} \\ \llbracket 2 + 5 \rrbracket_{[]} = 7 \\ \llbracket x + y \rrbracket_{[x \leftarrow 2, y \leftarrow 5]} = 7 \end{array}$$

Measures

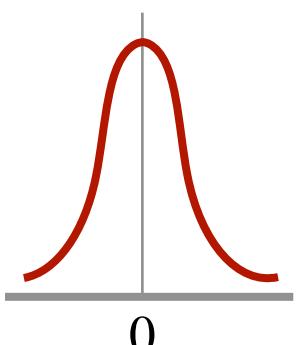
Maps a set of possible outcomes to a positive score: $\mu : \Sigma_X \rightarrow [0, \infty]$

Probability distributions are normalized measures, i.e., $\mu(\top) = 1$

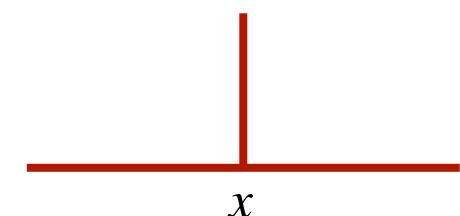
Bernoulli distribution (discrete)

$$\mathcal{B}(0.3)(\{1\}) = 0.3$$
$$\mathcal{B}(0.3)(\{0,1\}) = 1$$

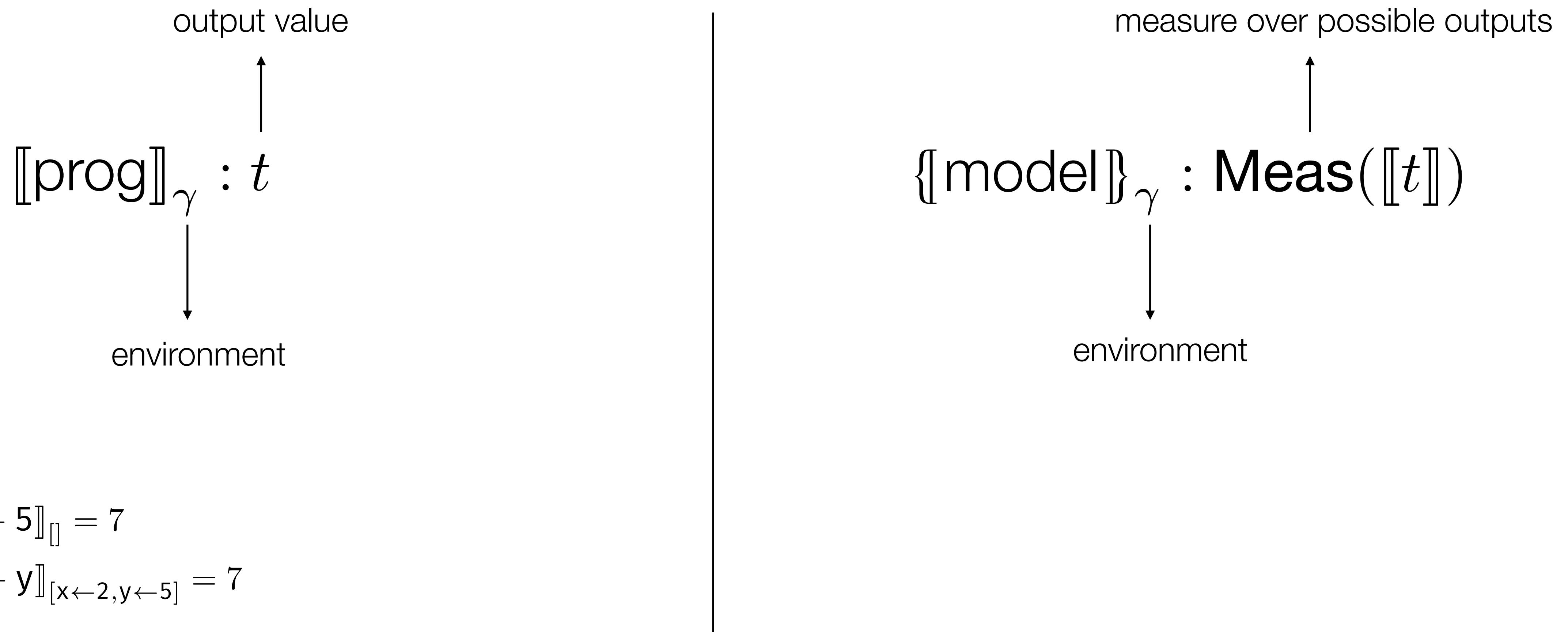

Normal distribution (continuous)

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


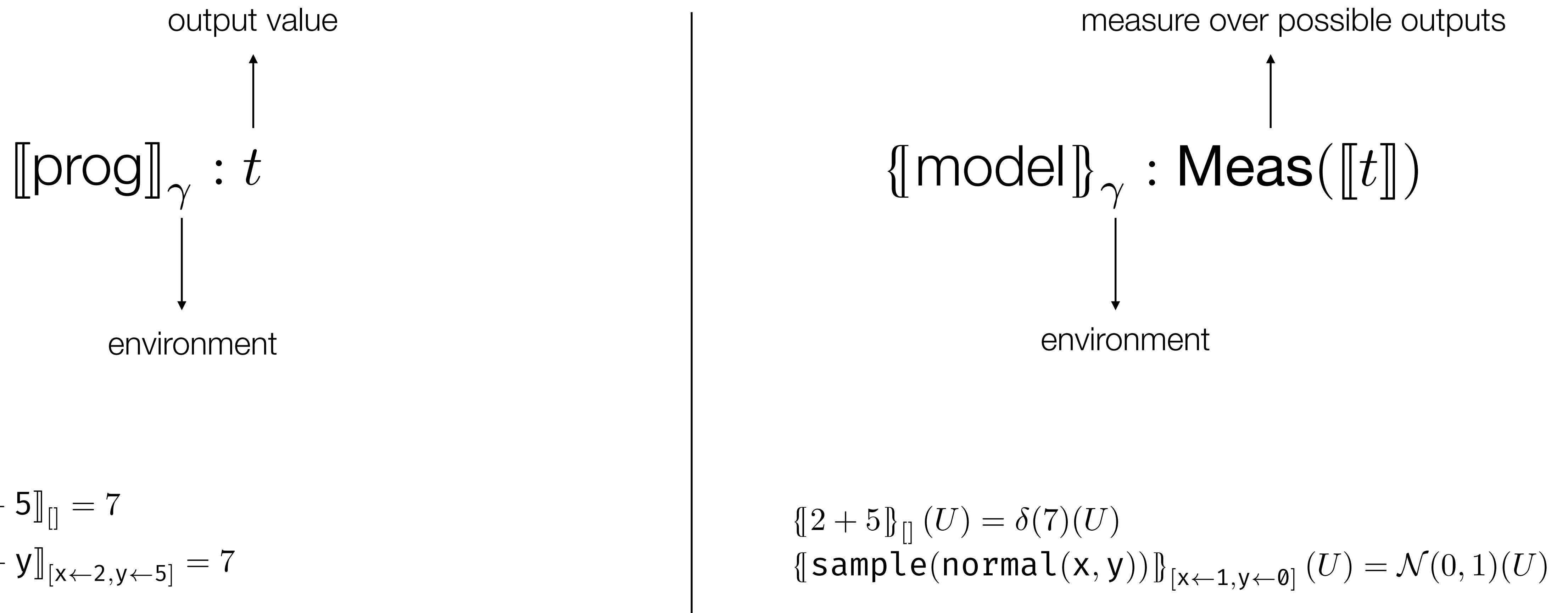
Dirac delta measure

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


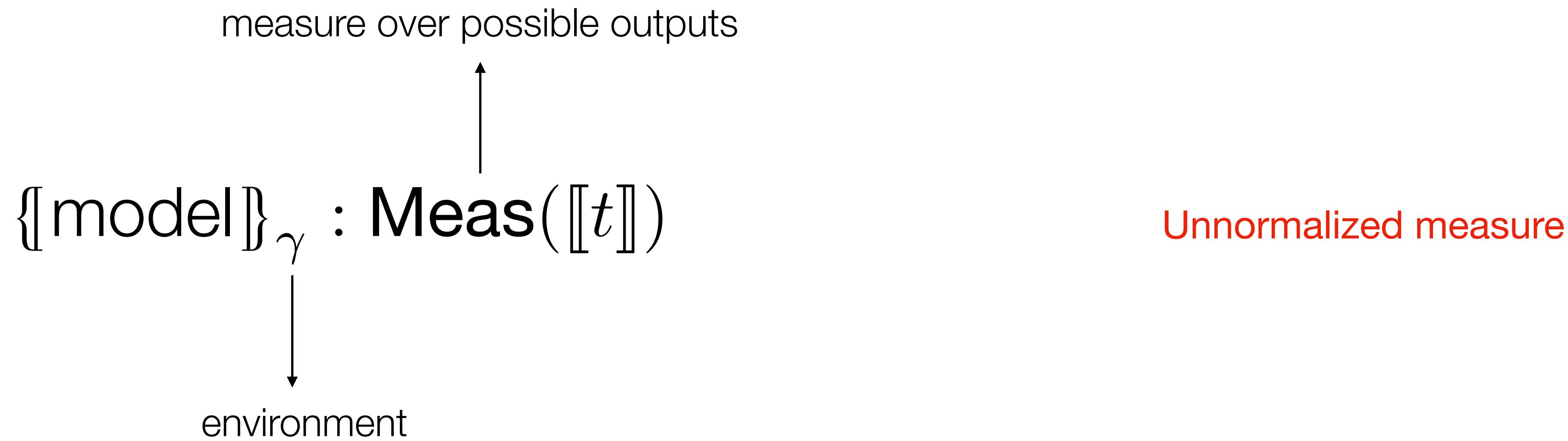
Deterministic vs. probabilistic semantics



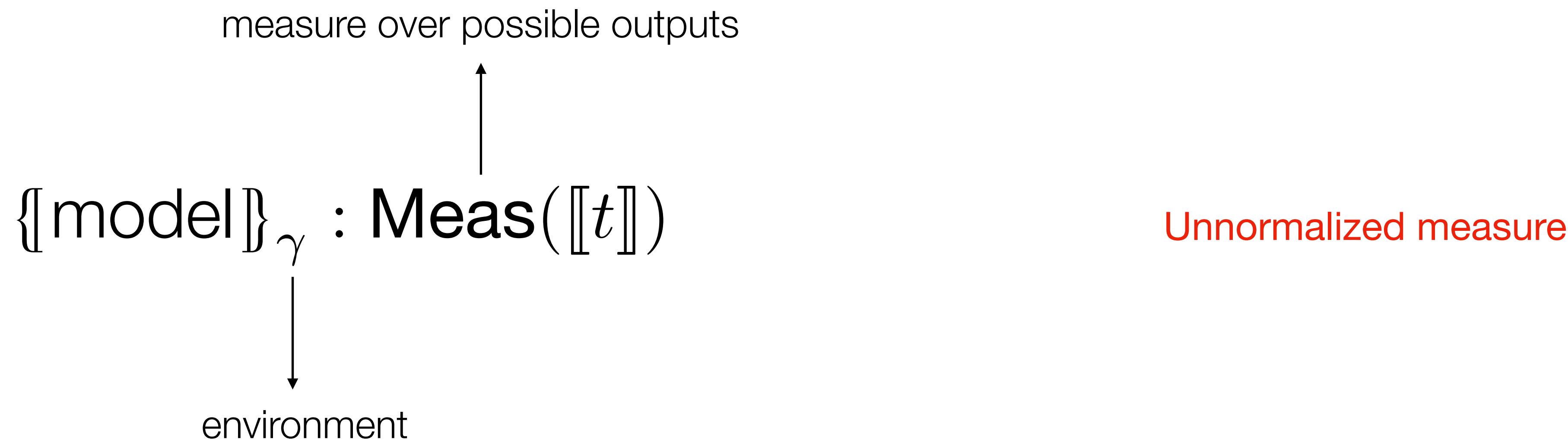
Deterministic vs. probabilistic semantics



(Un)normalized measures



(Un)normalized measures



$$\llbracket \text{infer}(\text{model}) \rrbracket_\gamma = \frac{\{\text{model}\}_\gamma}{\{\text{model}\}_\gamma (\llbracket \text{typeOf}(\text{model}) \rrbracket)}$$

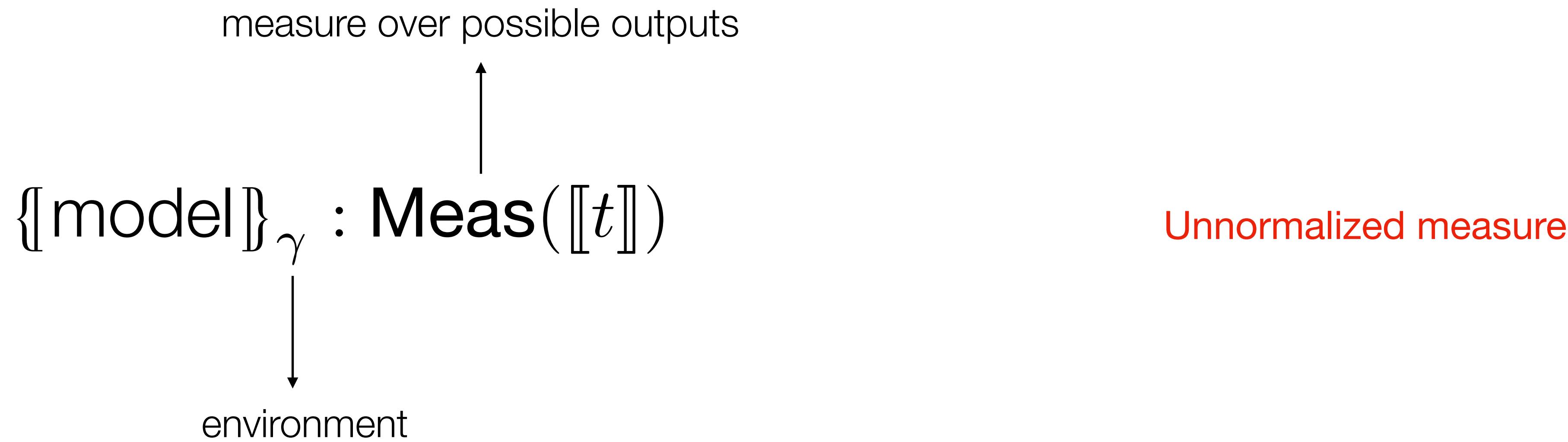
normalize over all possible values

↓

Distribution

The equation shows the calculation of a distribution from an unnormalized measure. It consists of a fraction where the numerator is the unnormalized measure $\{\text{model}\}_\gamma$ and the denominator is the sum of the unnormalized measures for all possible types, represented by $\{\text{model}\}_\gamma (\llbracket \text{typeOf}(\text{model}) \rrbracket)$. A downward-pointing arrow connects the denominator to the text "normalize over all possible values". To the right of the equation, the text "Distribution" is written in red.

(Un)normalized measures



$$\llbracket \text{infer}(\text{model}) \rrbracket_\gamma = \frac{\{\text{model}\}_\gamma}{\{\text{model}\}_\gamma (\llbracket \text{typeOf}(\text{model}) \rrbracket)}$$



Distribution

normalize over all possible values

Approximate inference

Probabilistic Programming Languages

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

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

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

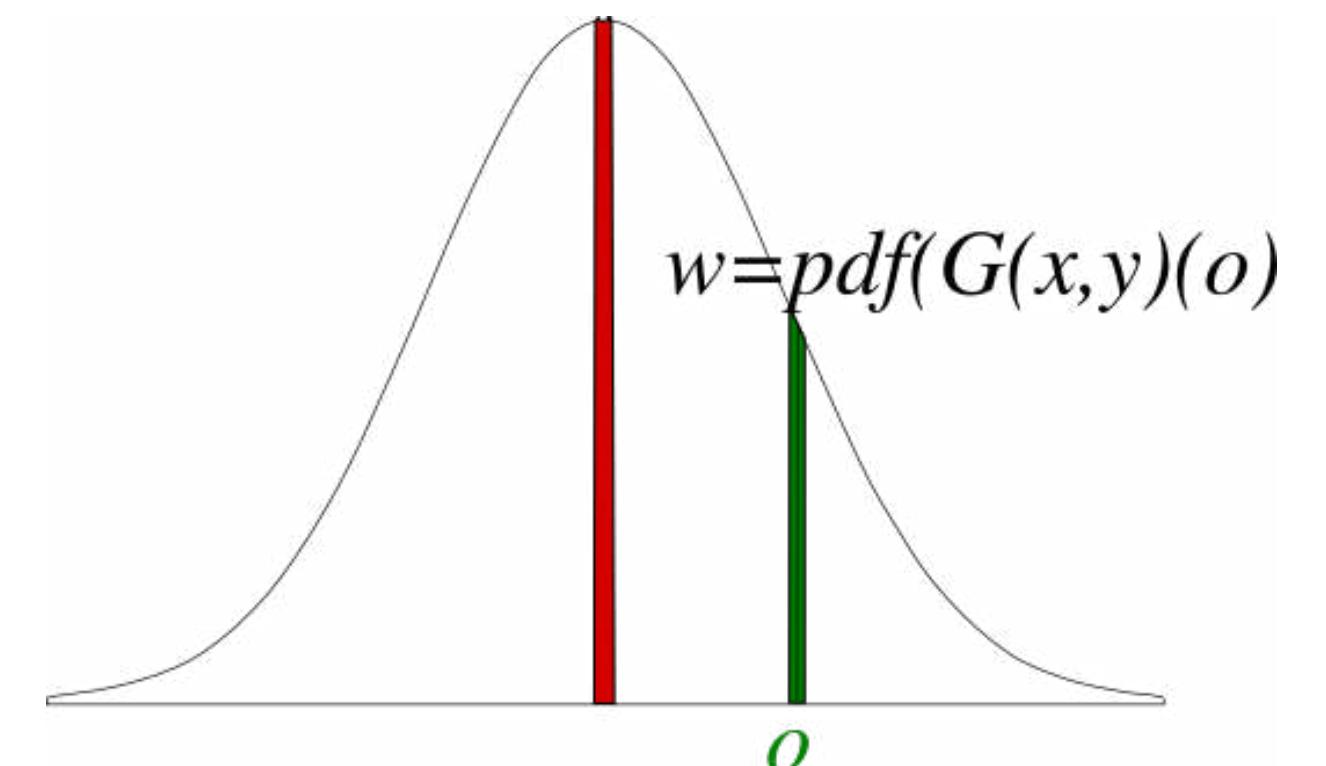
Inference algorithm

- 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

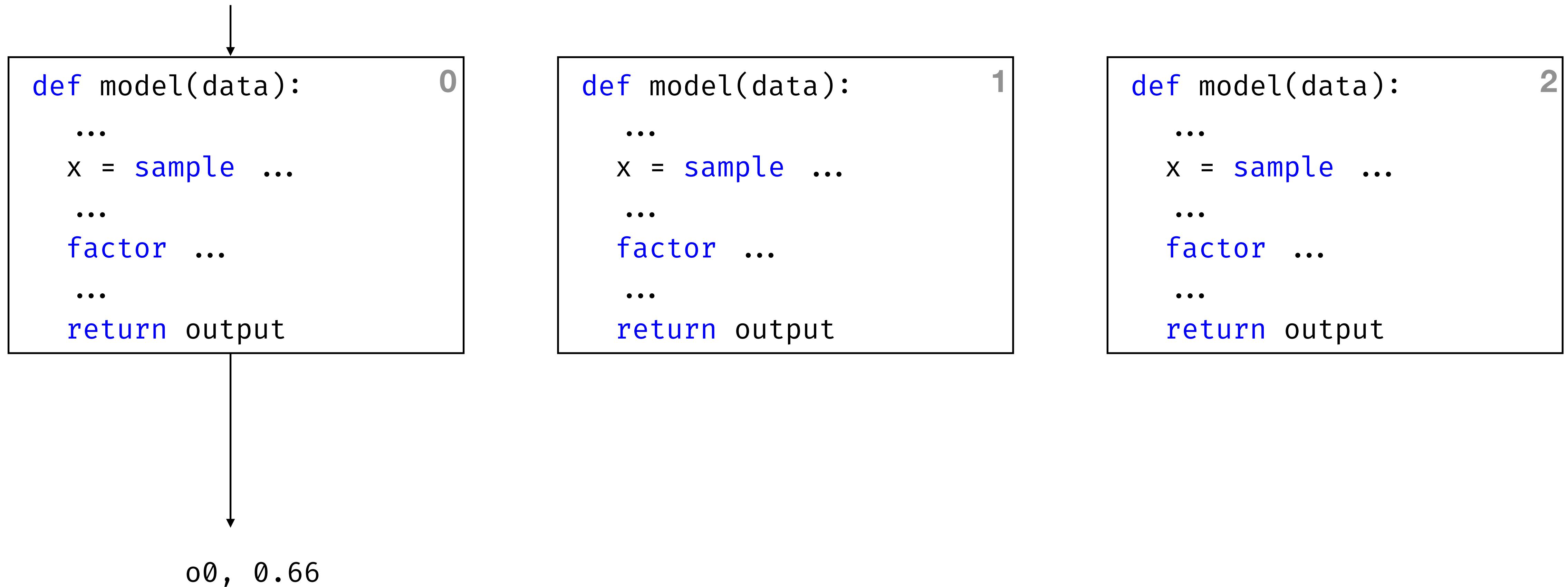


```
def model(data): 0  
    ...  
    x = sample ...  
    ...  
    factor ...  
    ...  
    return output
```

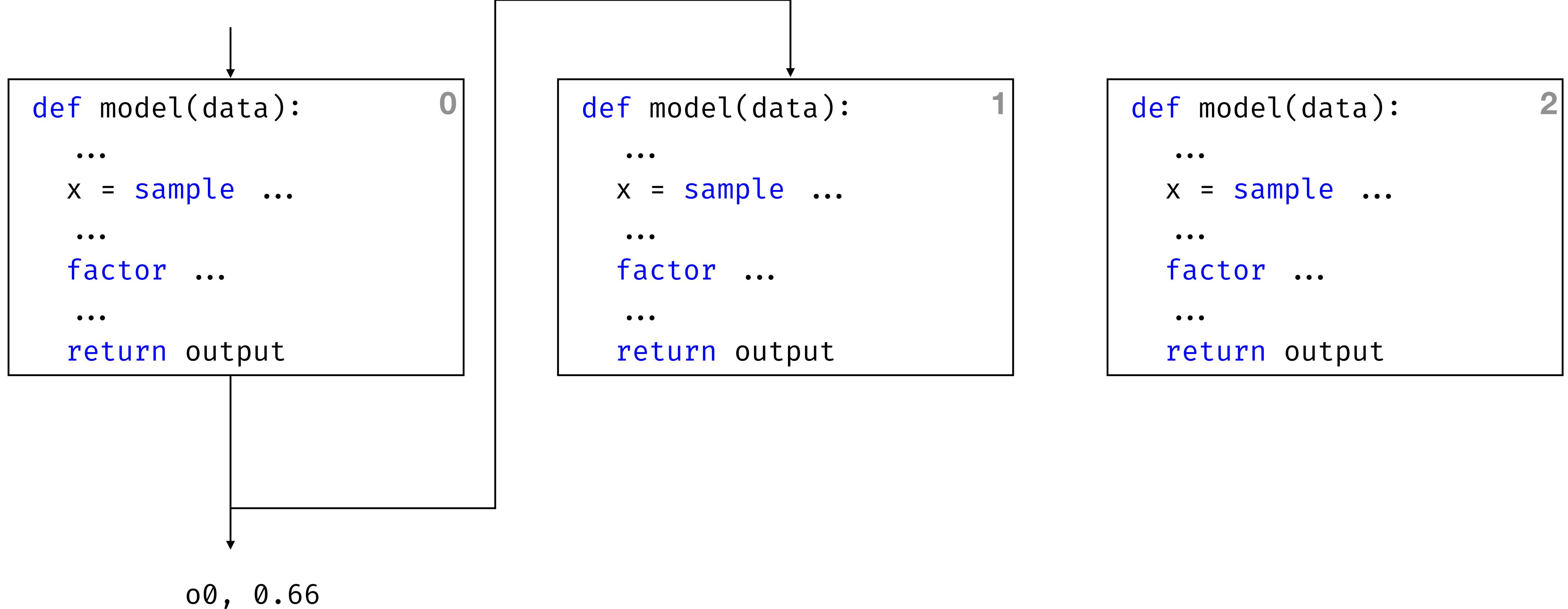
```
def model(data): 1  
    ...  
    x = sample ...  
    ...  
    factor ...  
    ...  
    return output
```

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

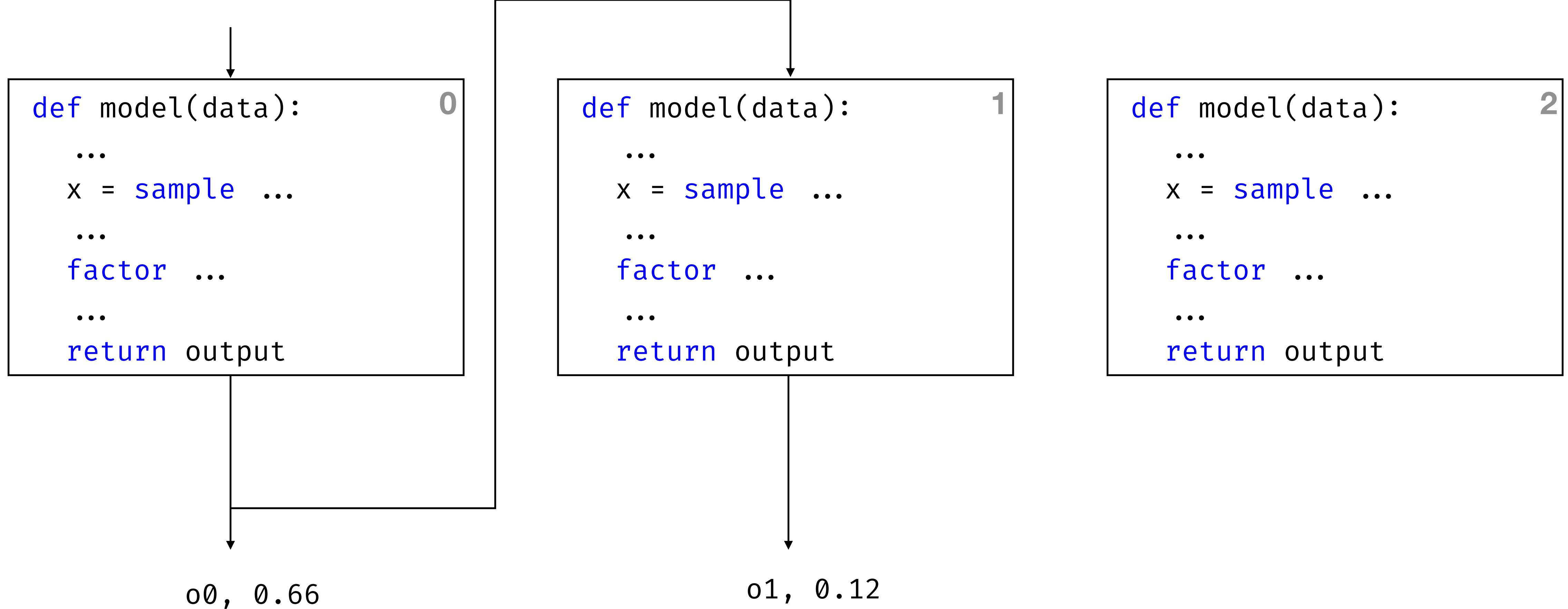
Importance sampling



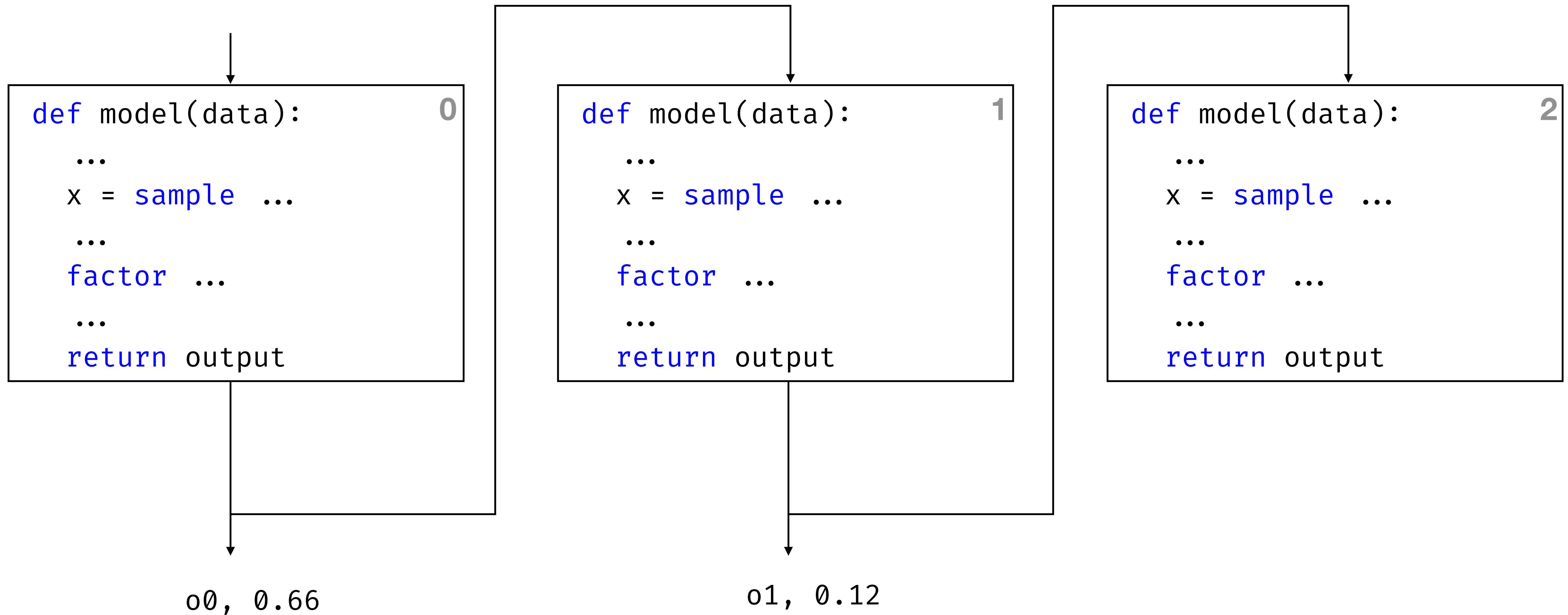
Importance sampling



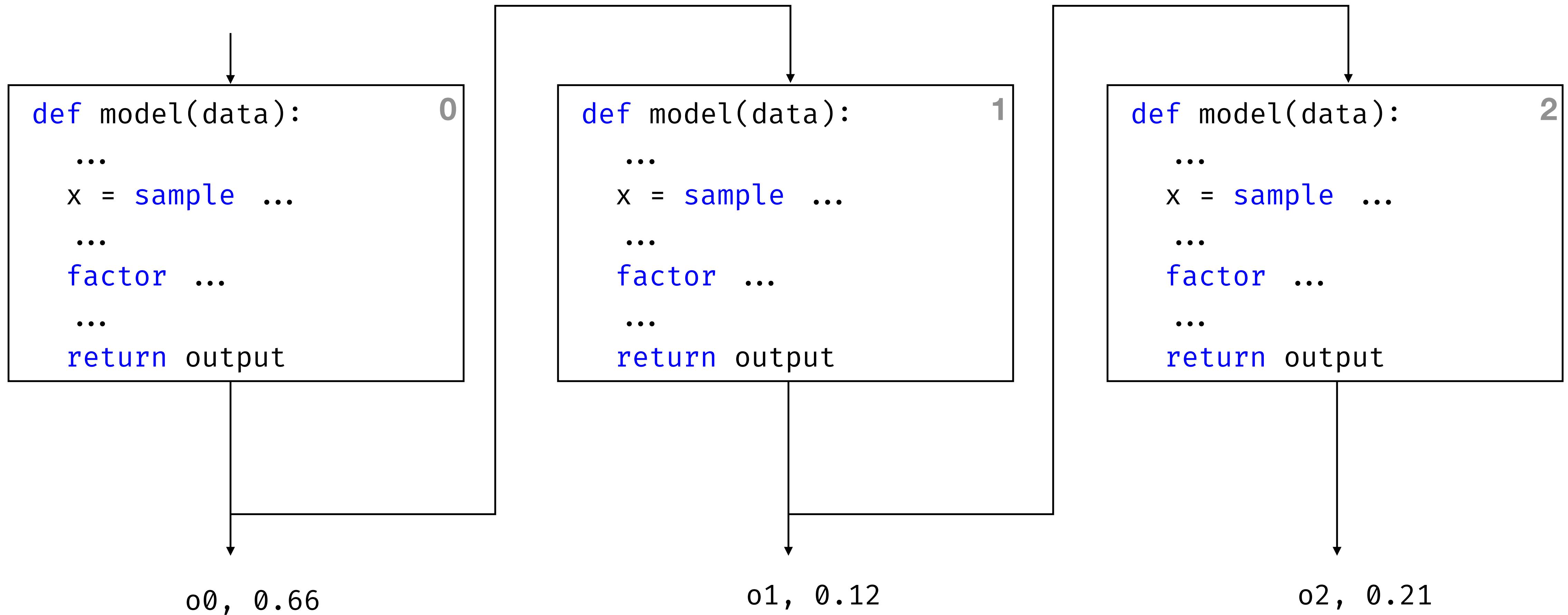
Importance sampling



Importance sampling



Importance sampling



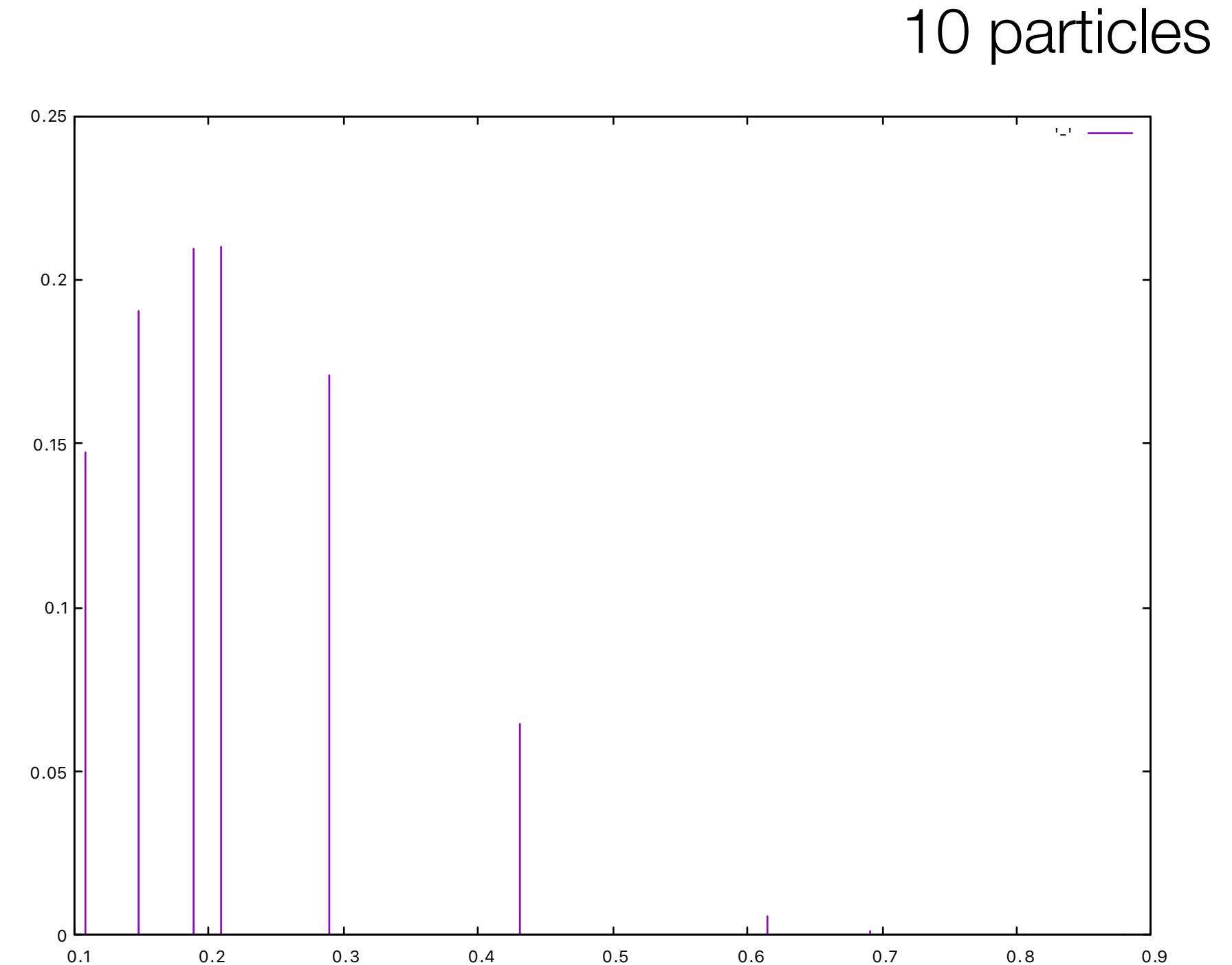


Example: coin

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



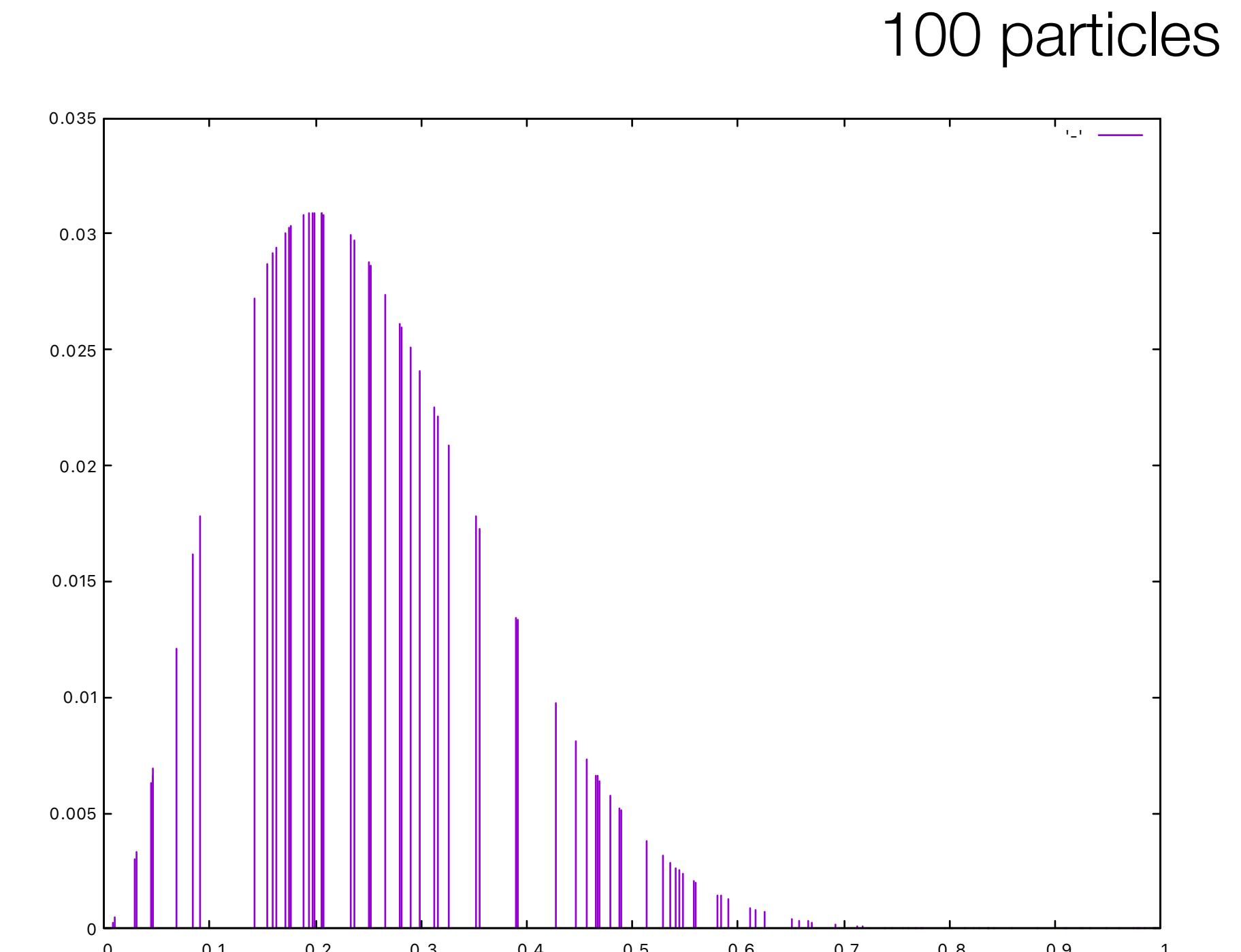


Example: coin

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



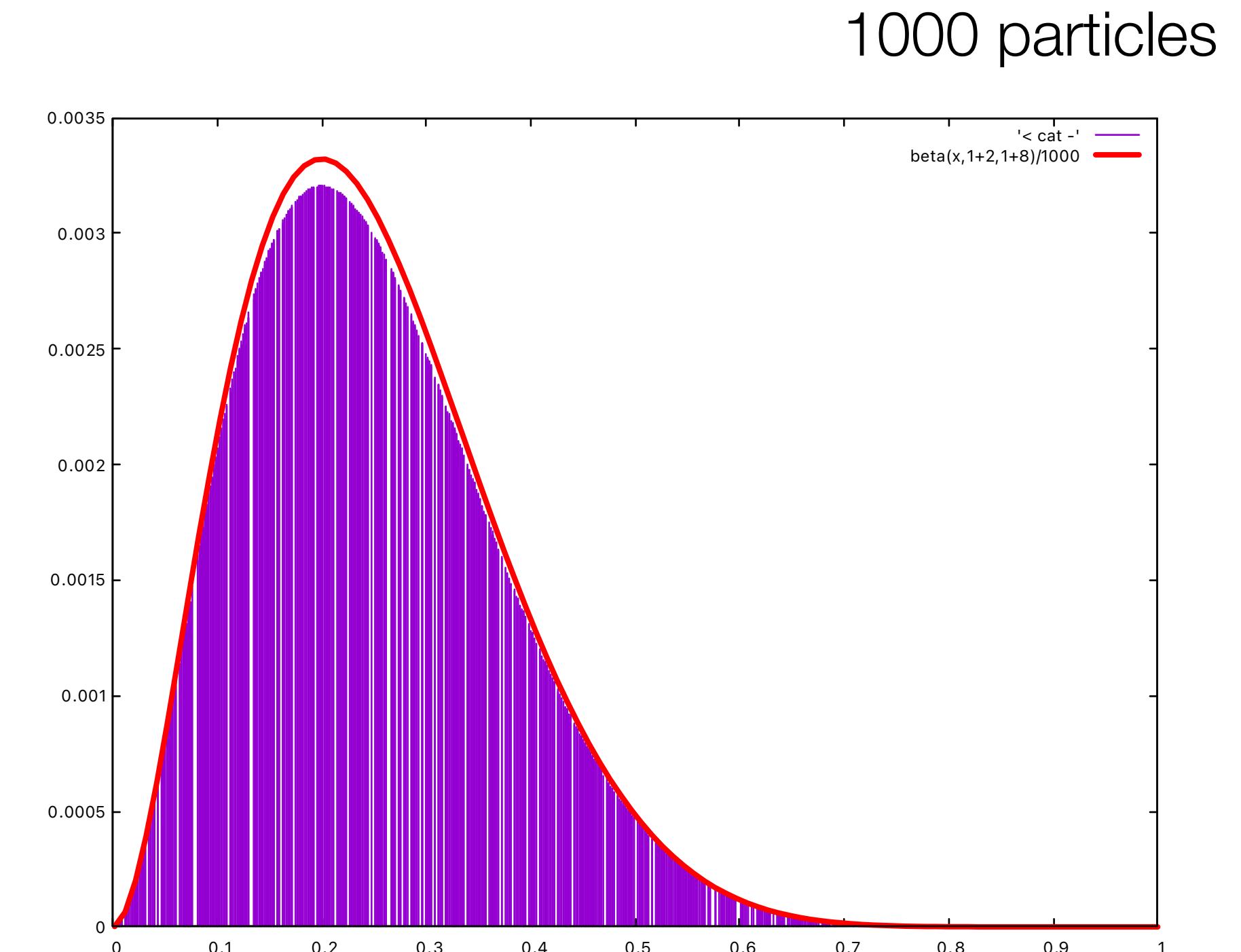


Example: coin

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





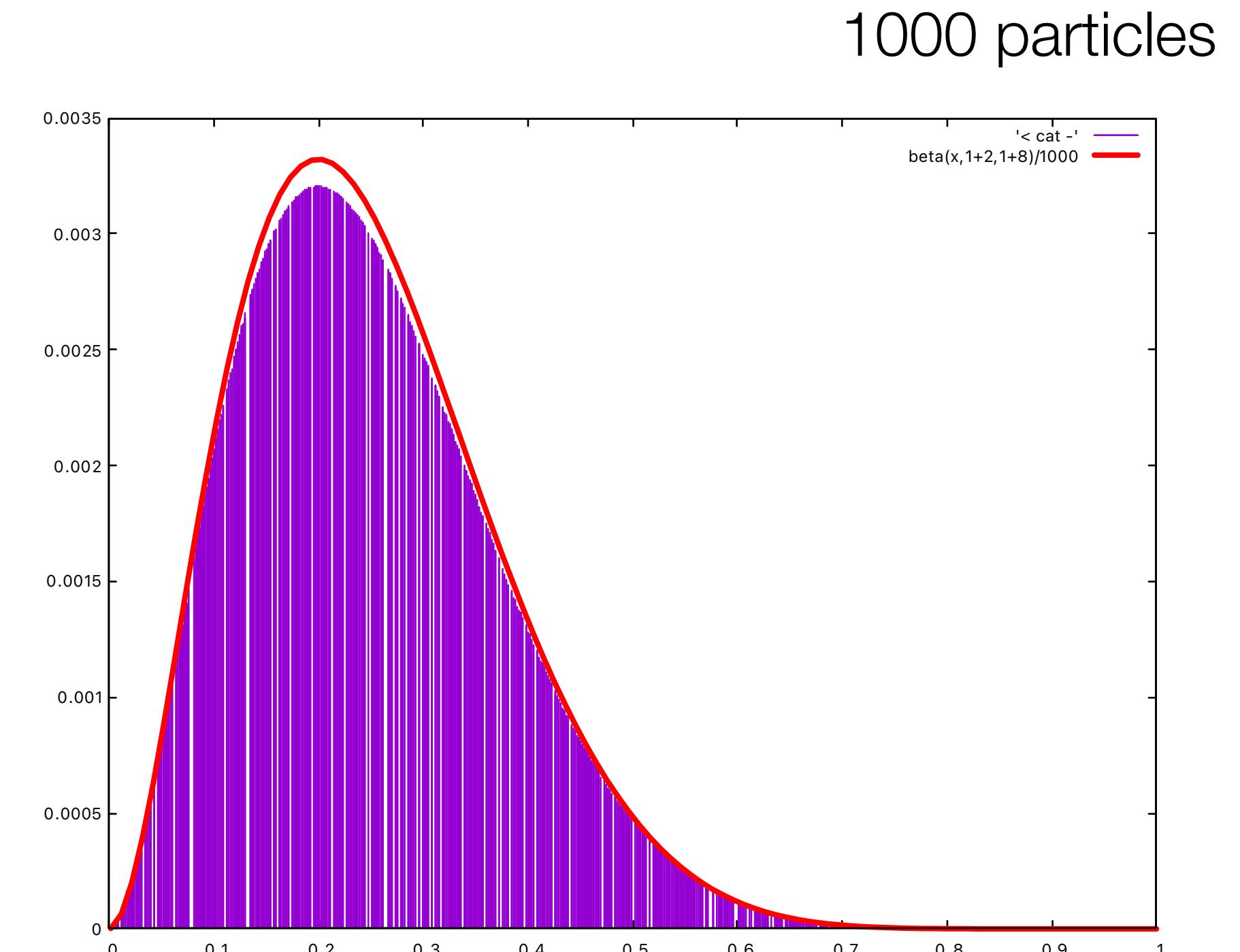
Example: coin

```
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

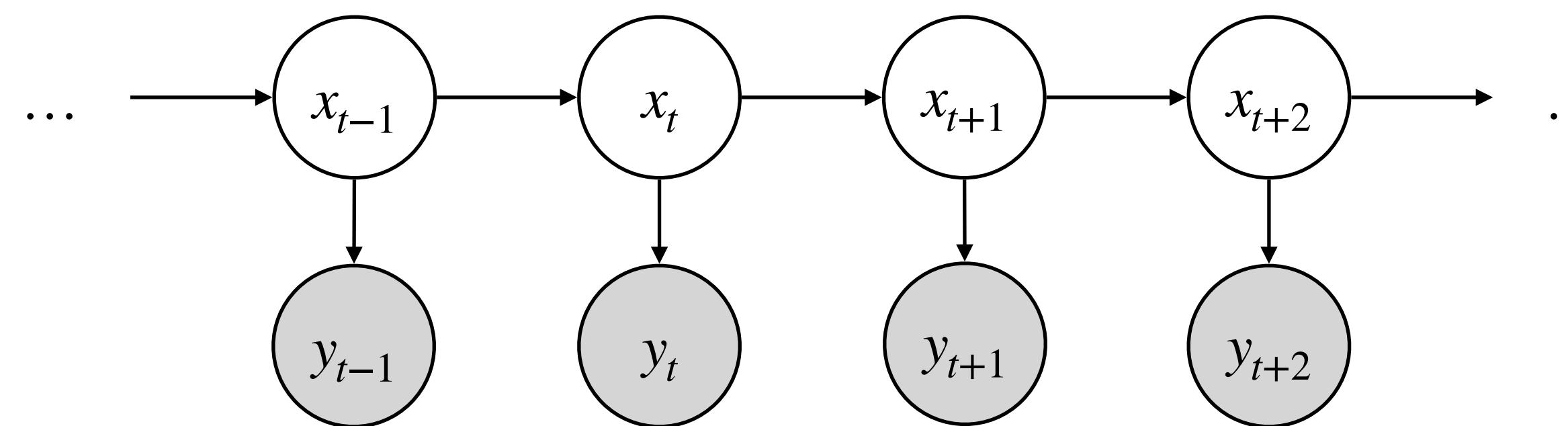
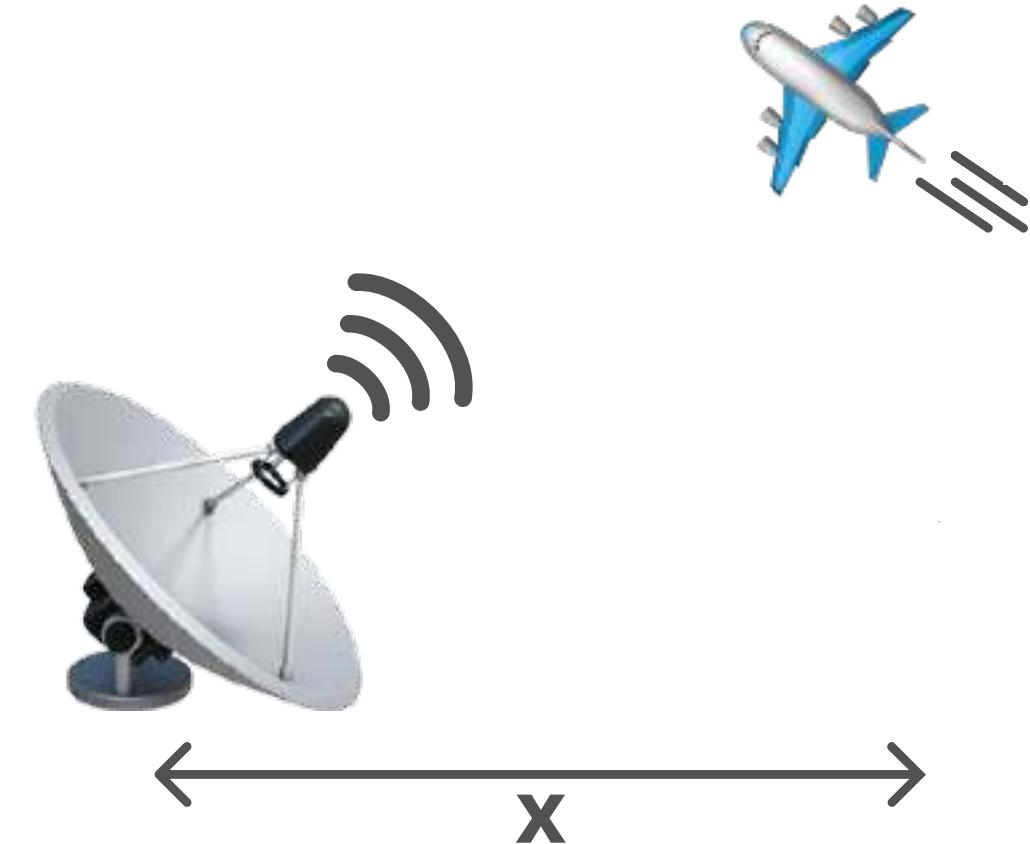
with ImportanceSampling(num_particles=1000):
    dist = infer(coin, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1])
viz(dist)
```

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



Example: tracking

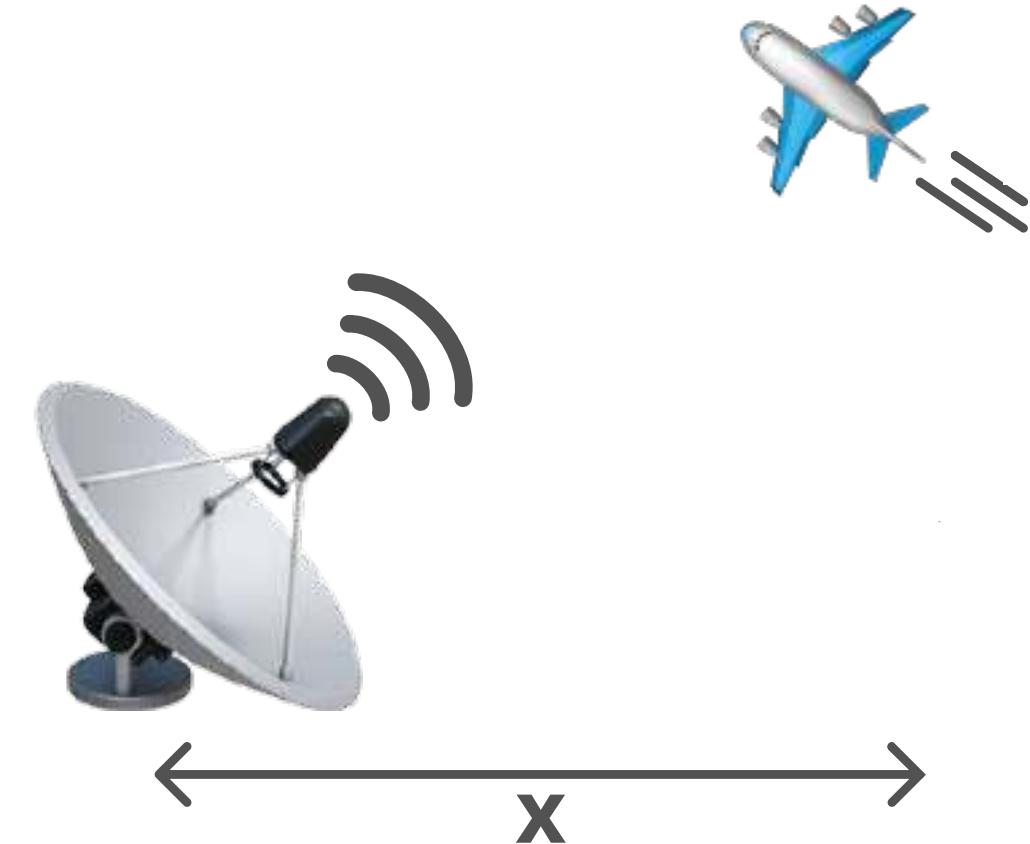
```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



latent observed

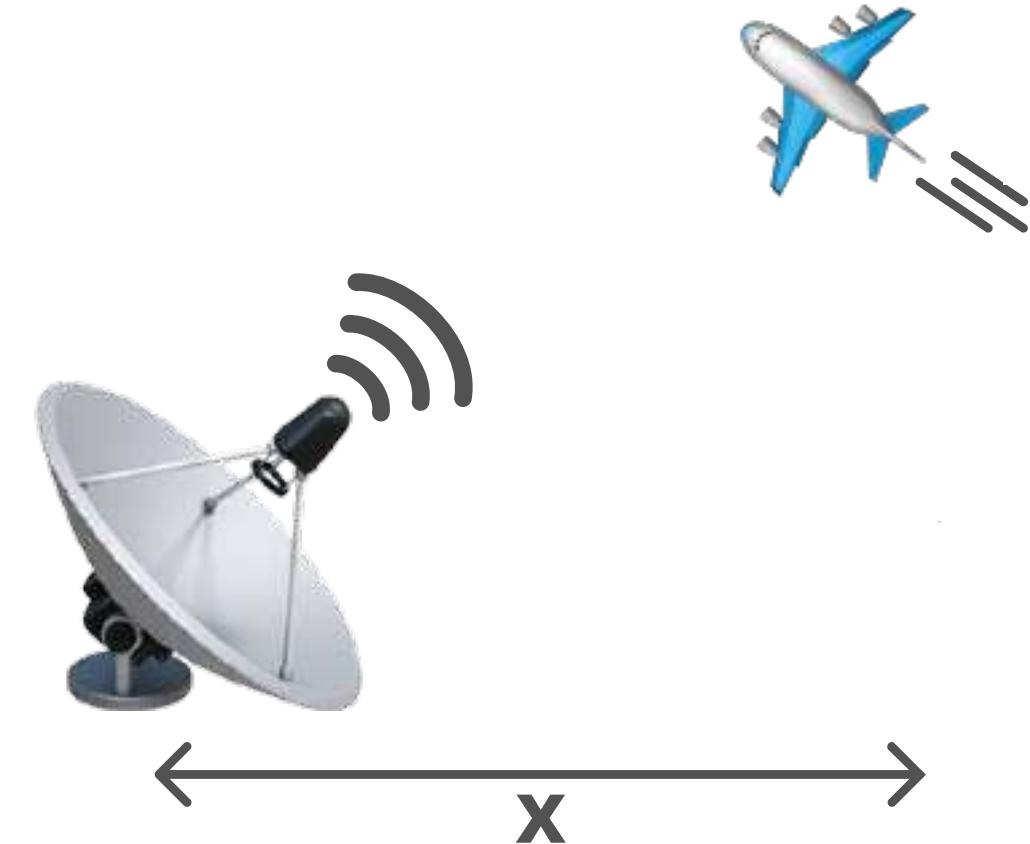
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



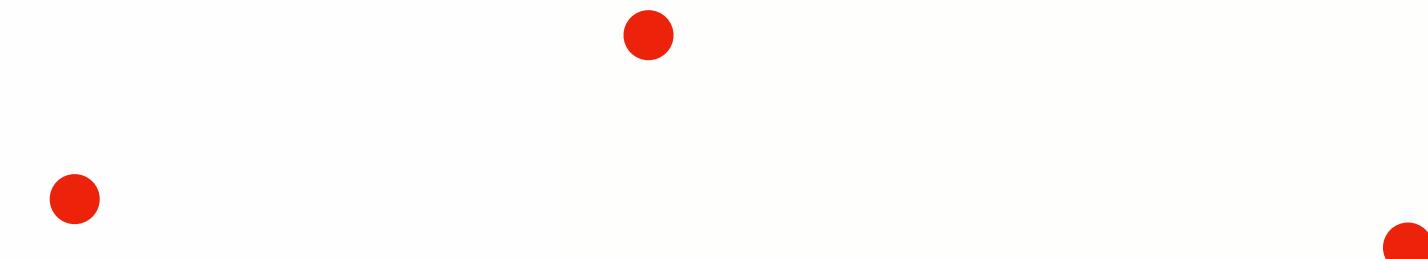
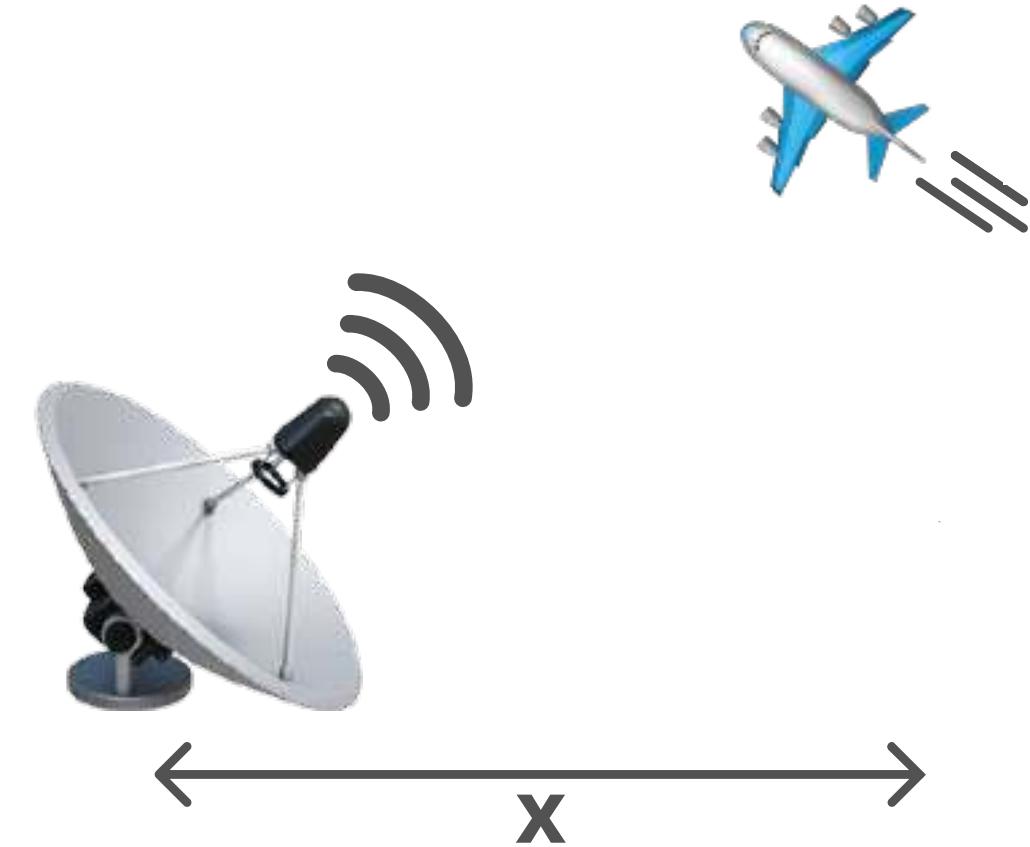
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



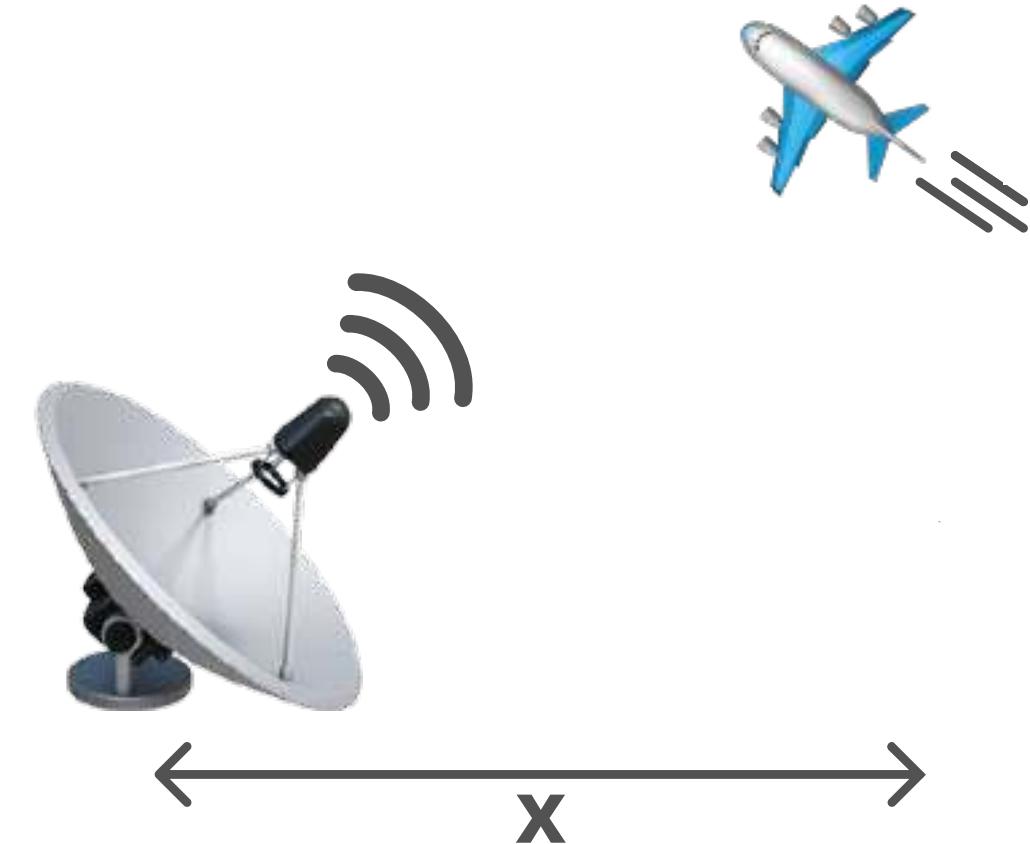
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



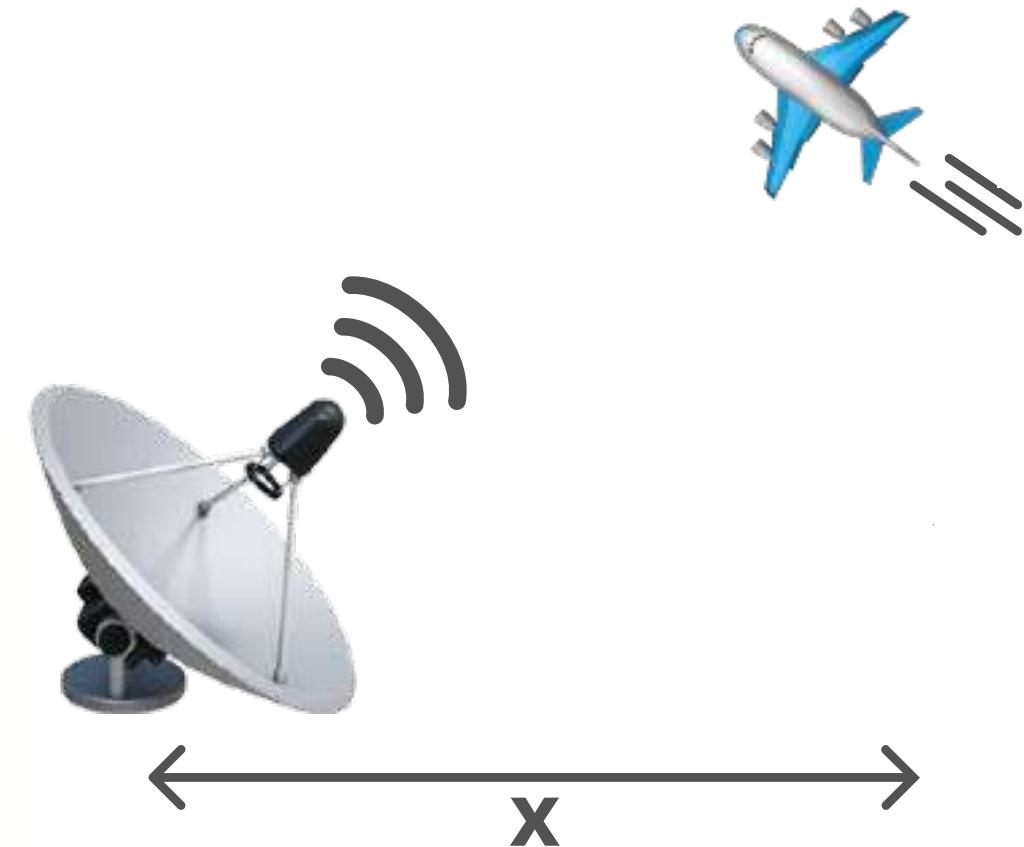
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



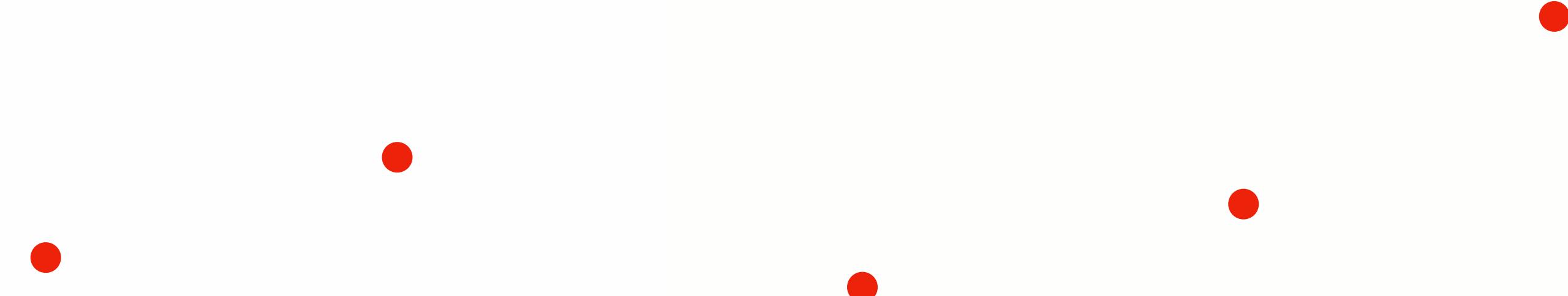
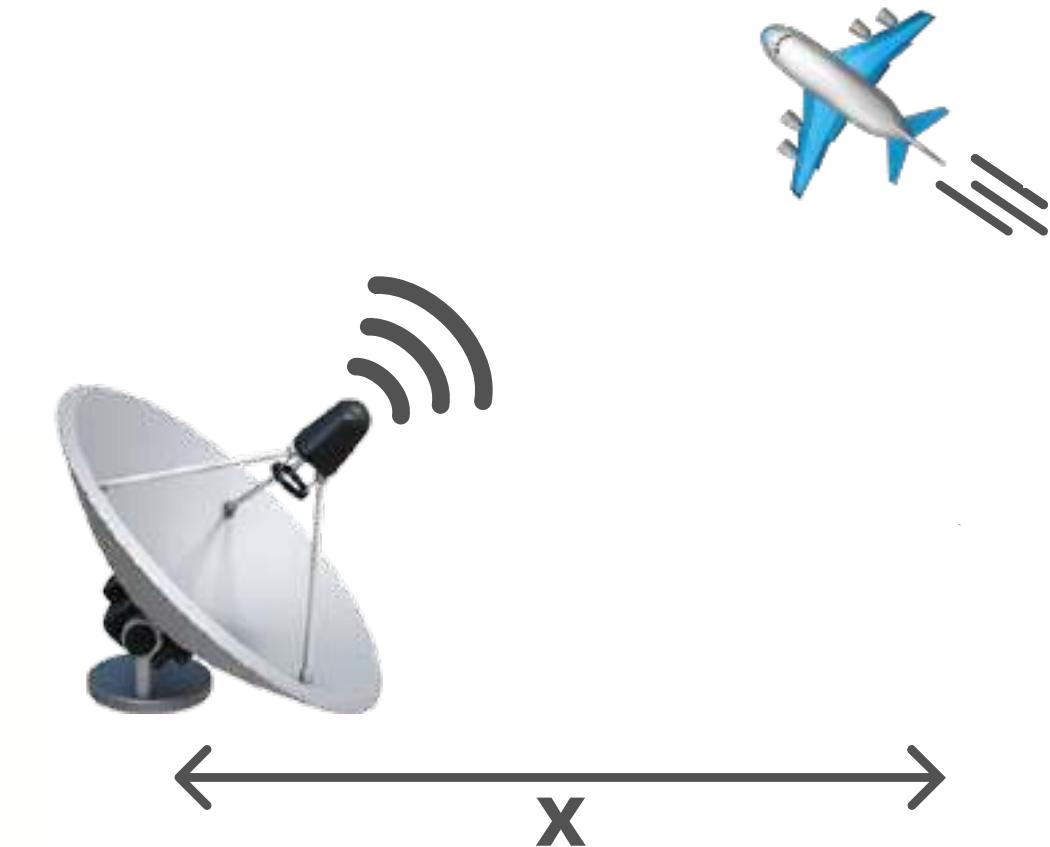
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



Example: tracking

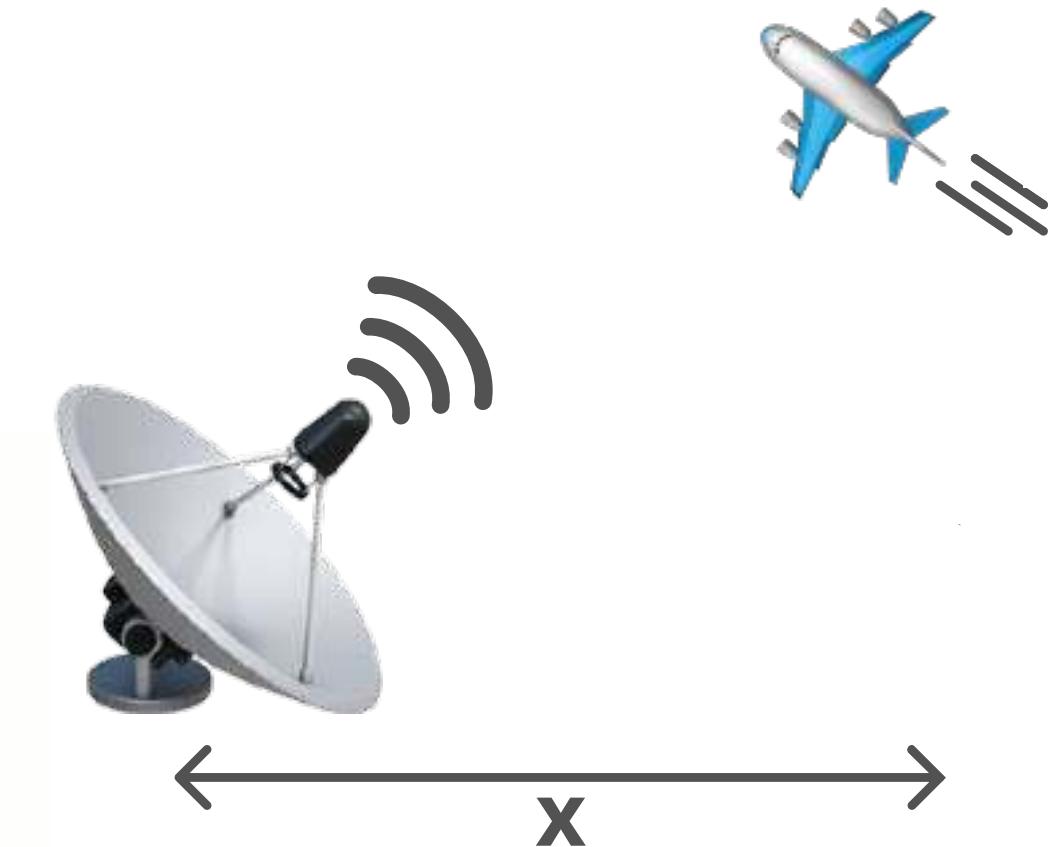
```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)         # condition on observation  
        acc.append(x)  
    return acc
```



Bad estimation

Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



Problem:

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



Bad estimation

The curse of dimensionality



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



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

17h45mn



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

17h45mn

How to mitigate this problem?

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



Particle filter

Particle filter

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)

Particle filter

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)

Add a resampling step

- Checkpoint: stop the particles in the middle of the execution
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

Particle filter



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

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

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

Particle filter



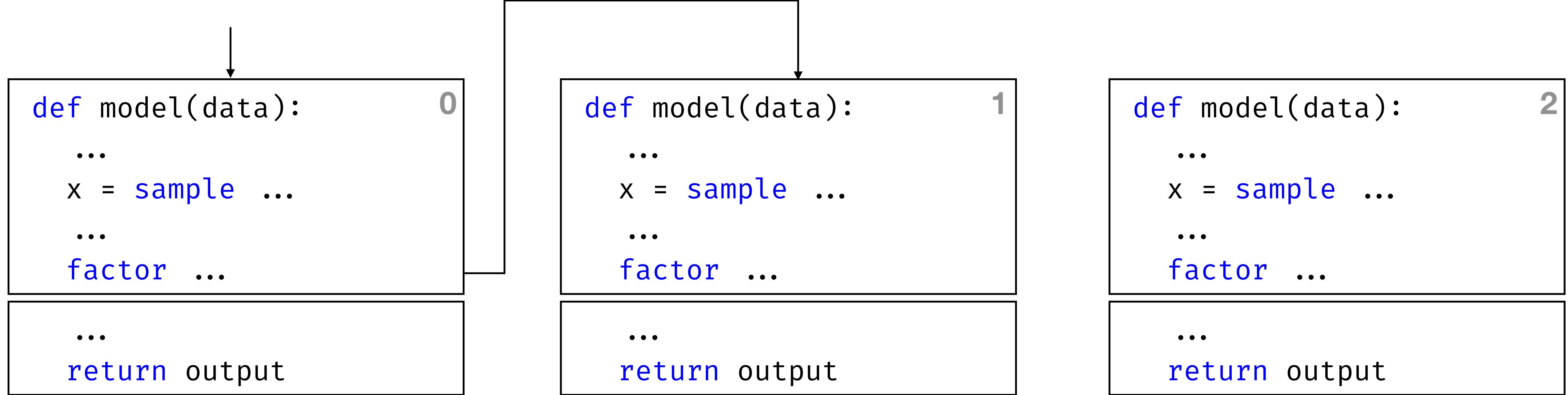
```
def model(data): 0  
    ...  
    x = sample ...  
    ...  
    factor ...  
    ...  
    return output
```

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

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

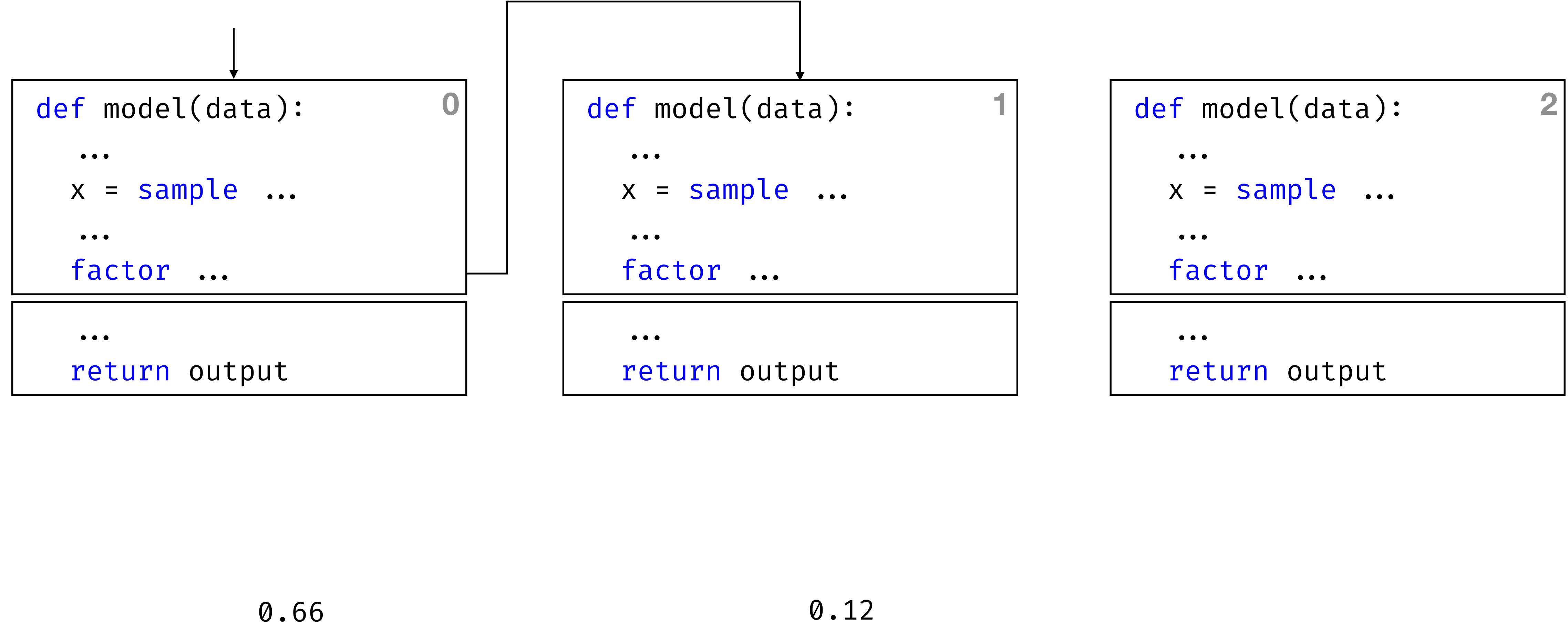
0.66

Particle filter

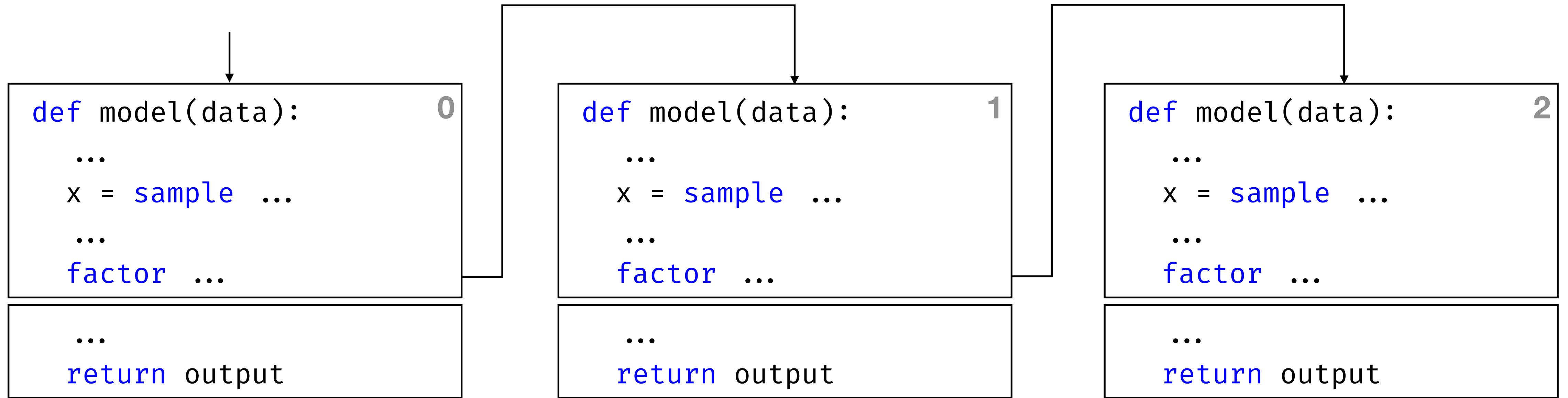


0.66

Particle filter



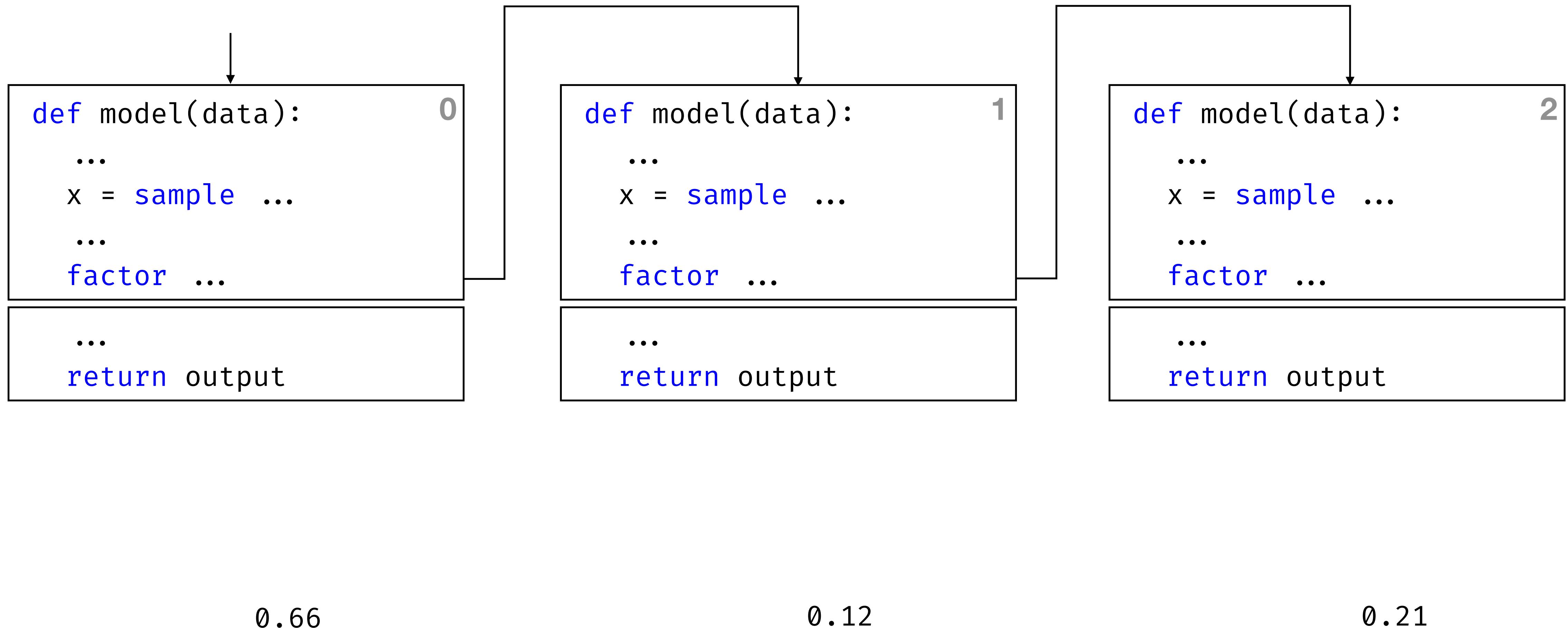
Particle filter



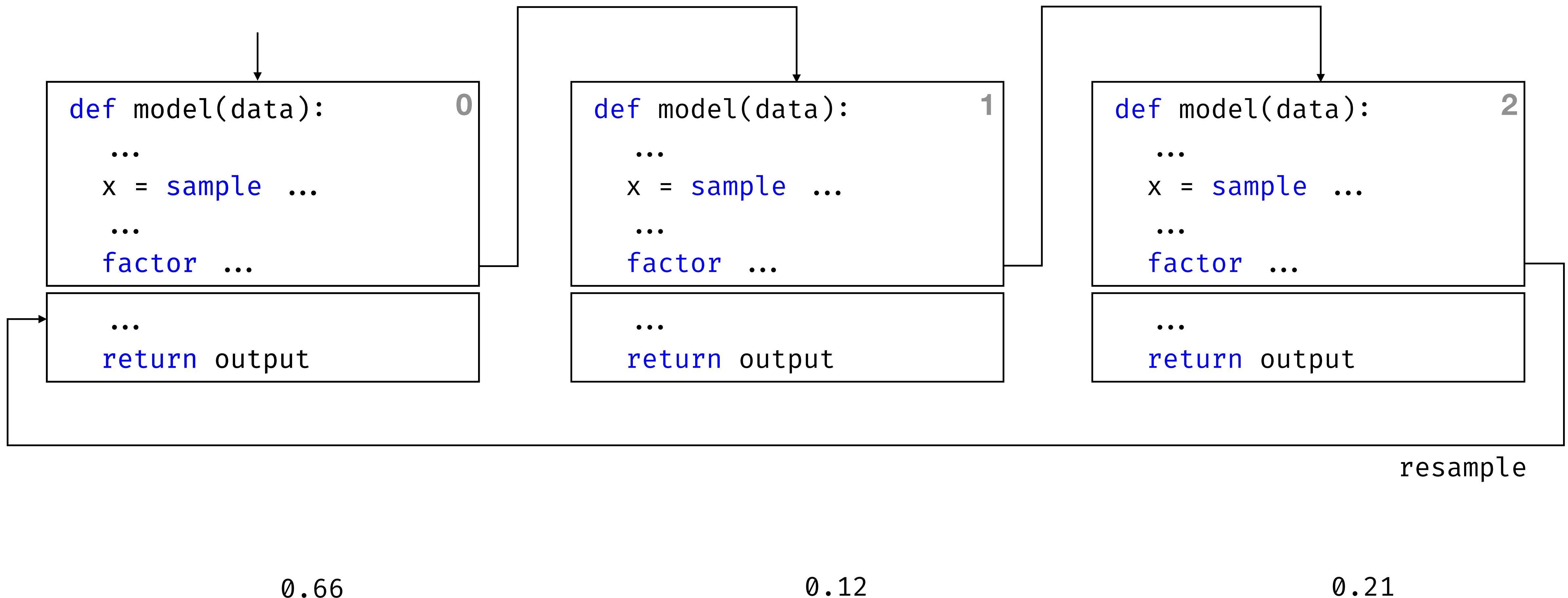
0.66

0.12

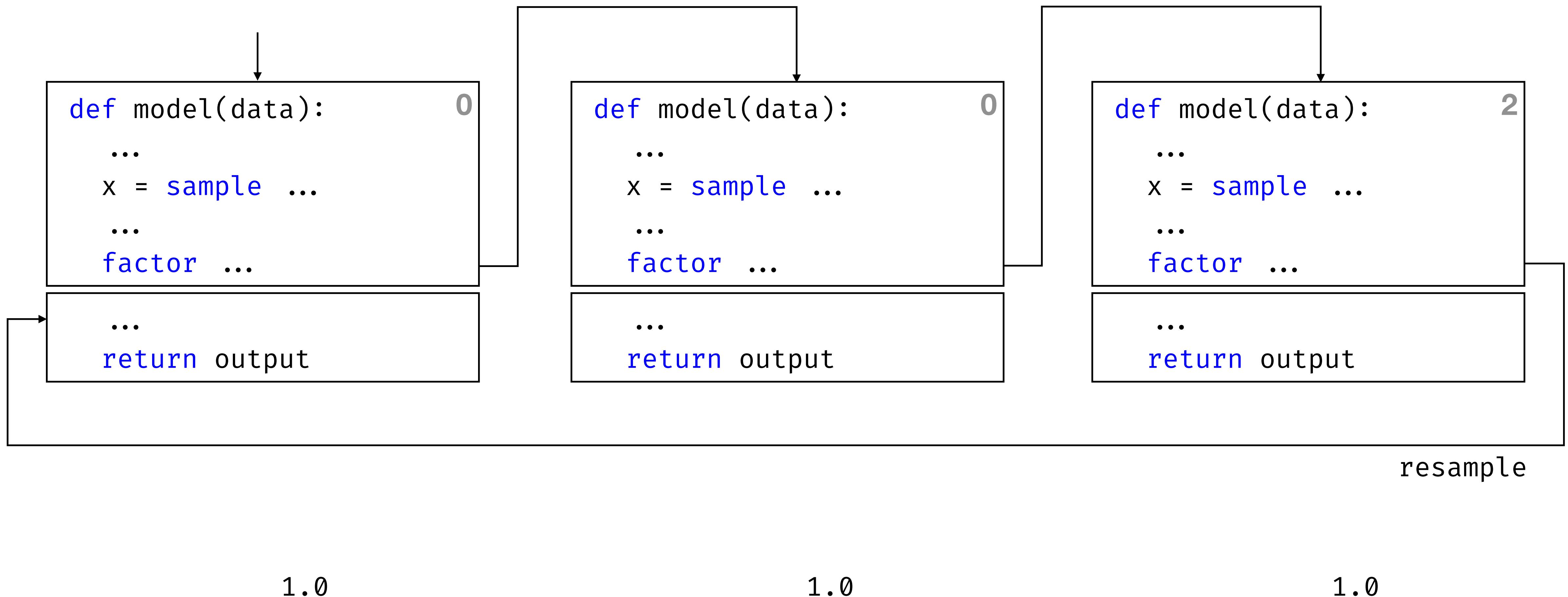
Particle filter



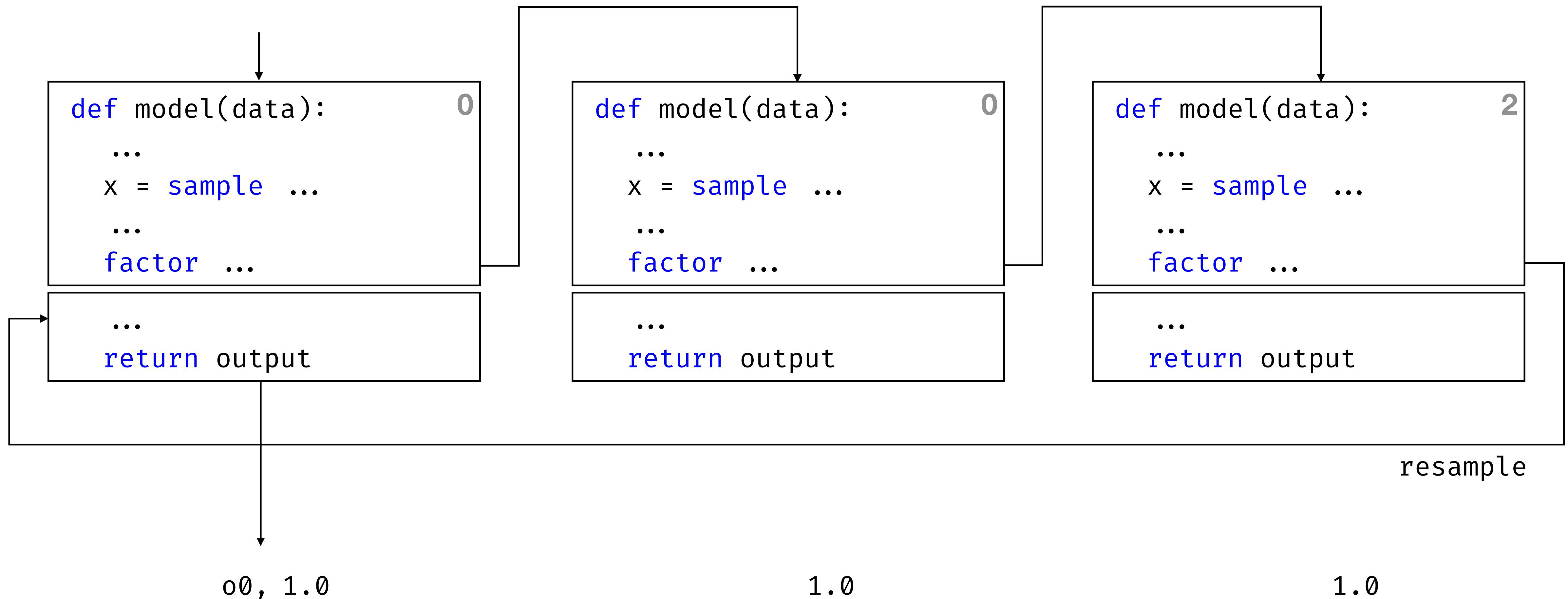
Particle filter



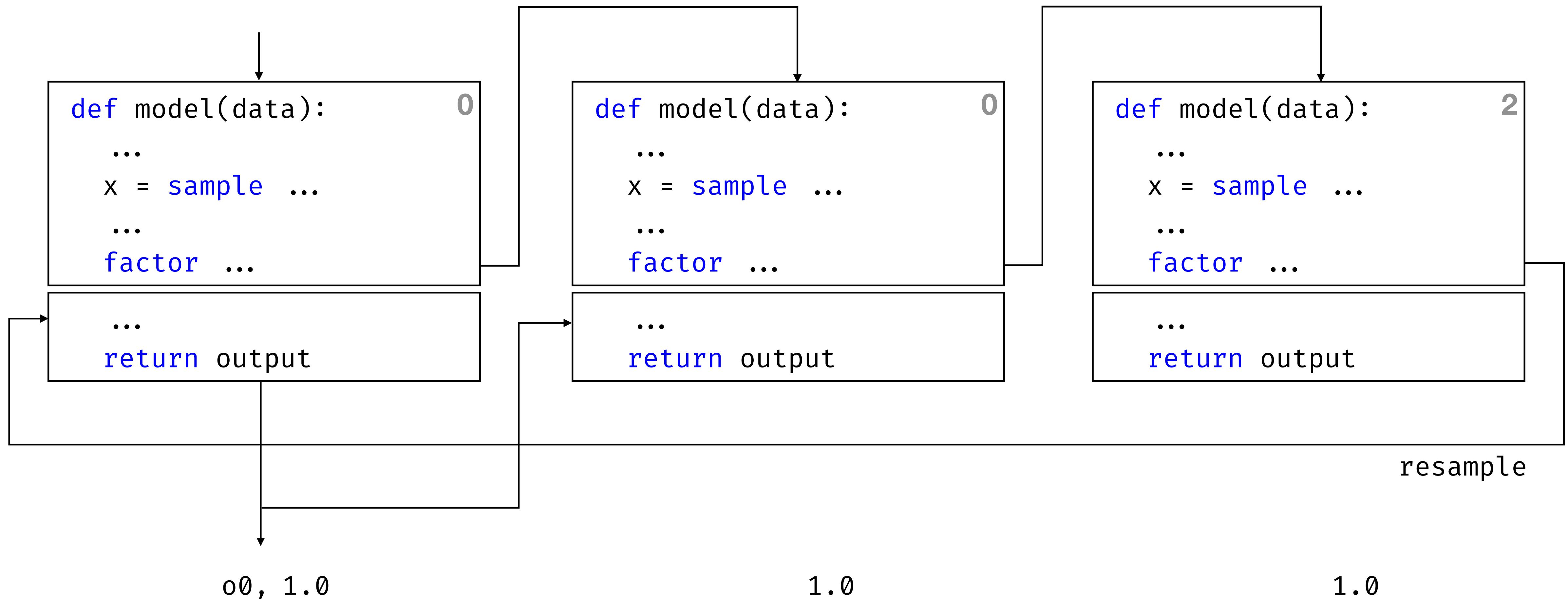
Particle filter



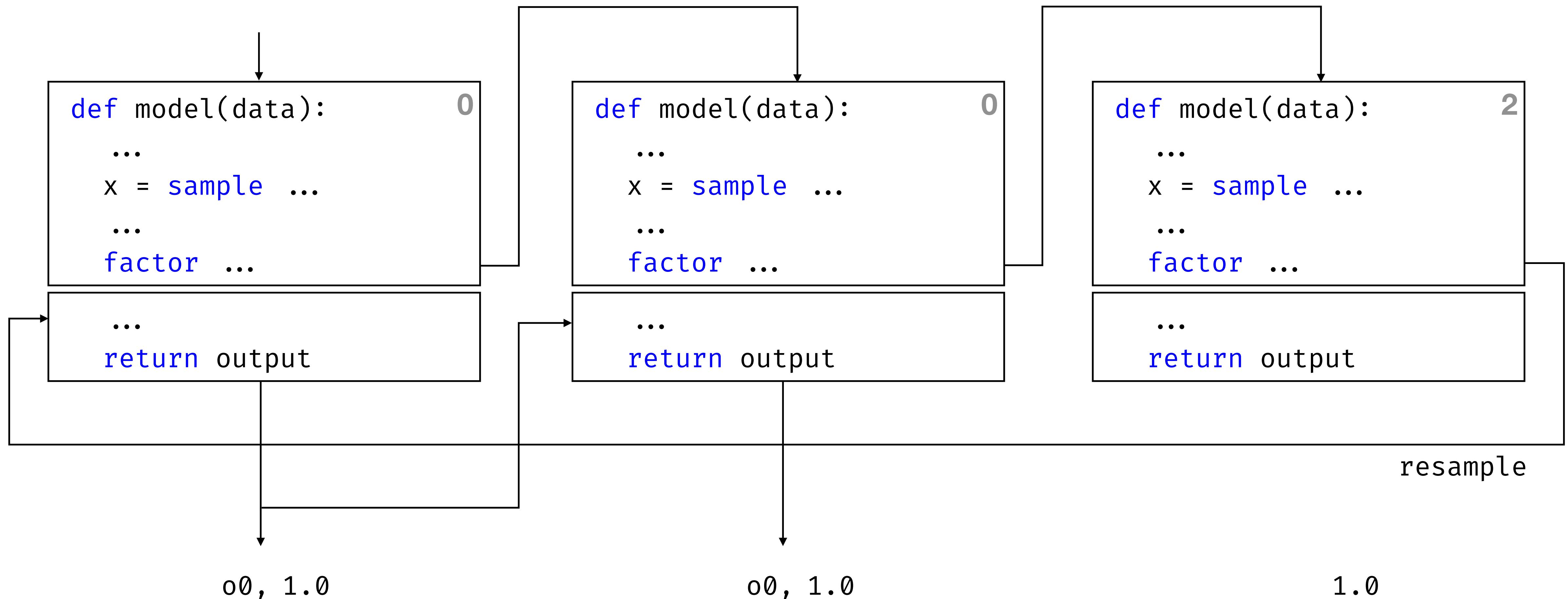
Particle filter



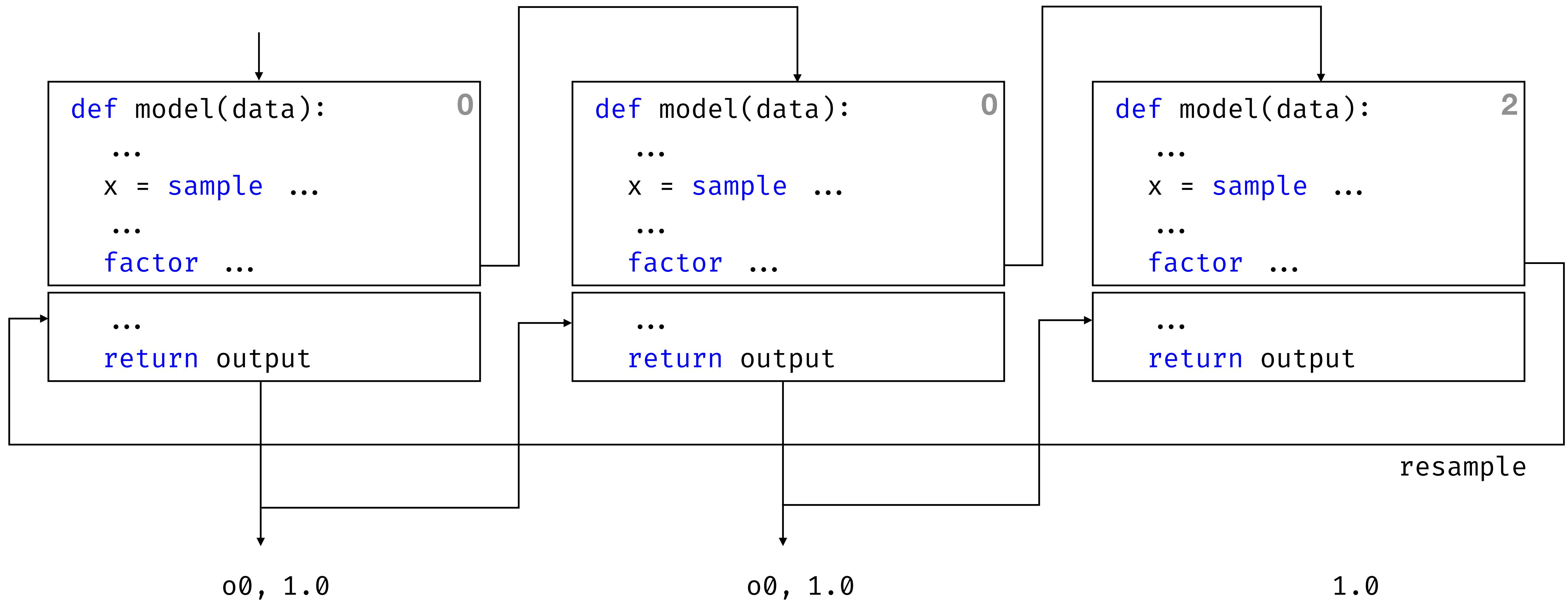
Particle filter



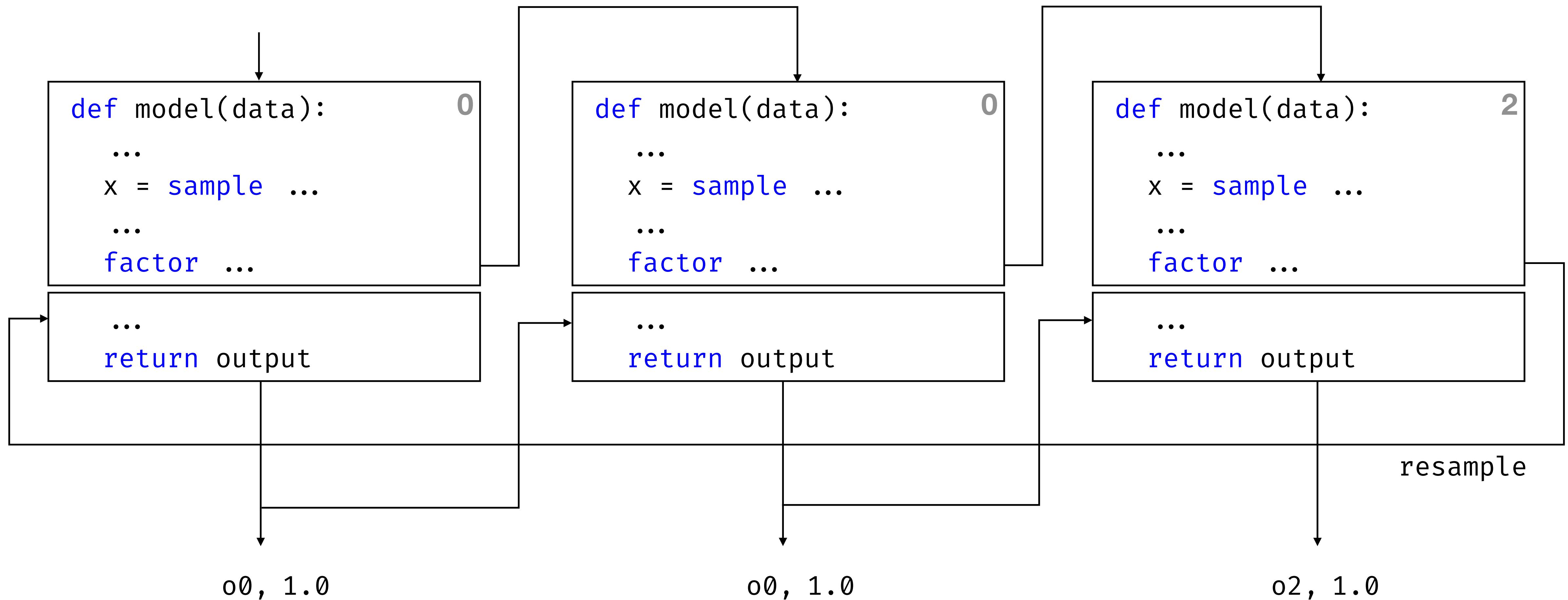
Particle filter



Particle filter

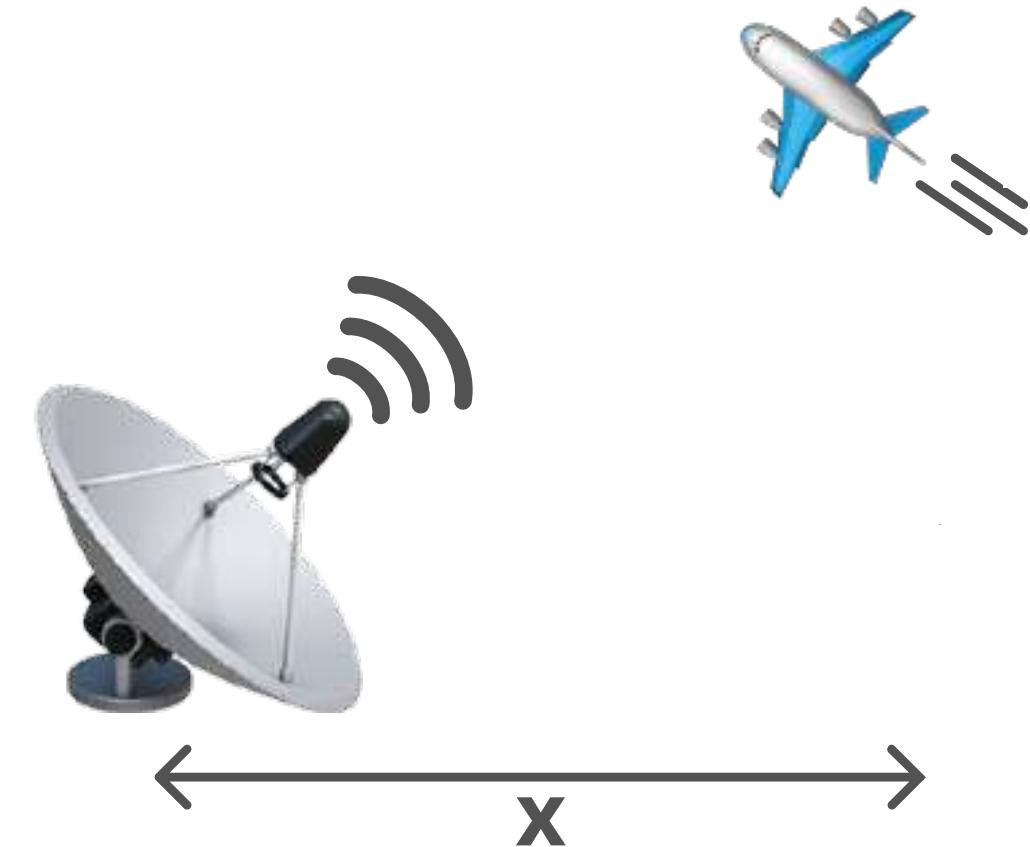


Particle filter



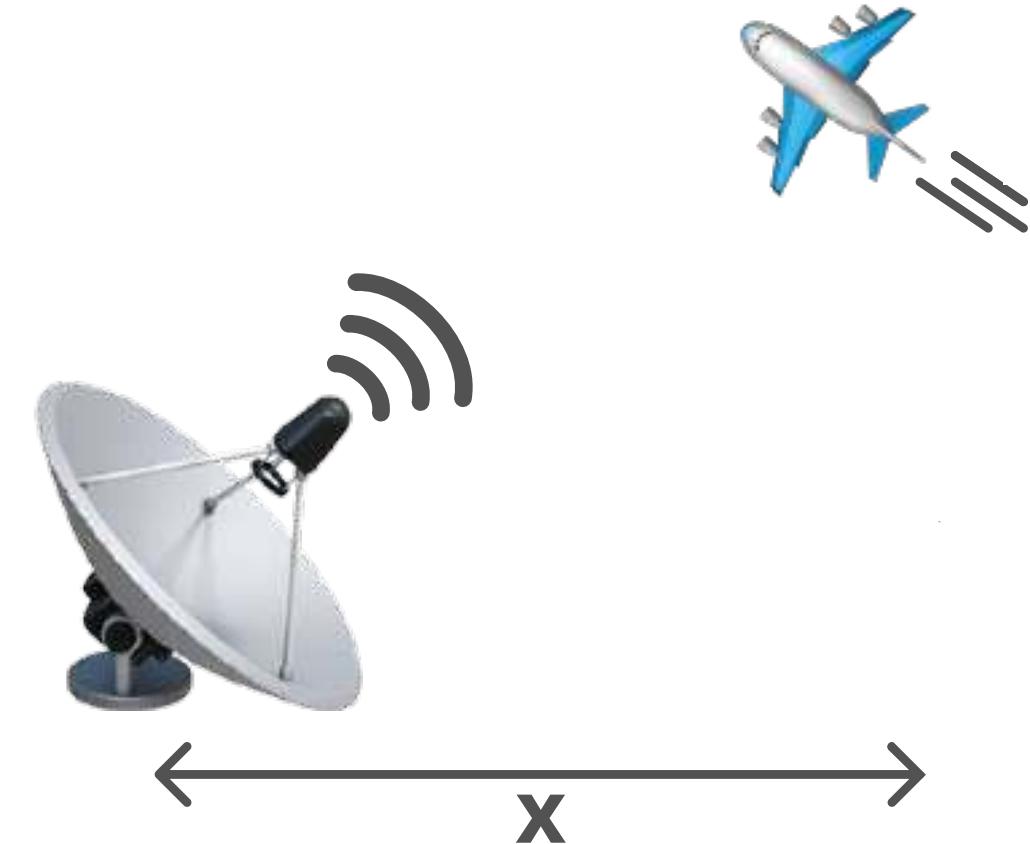
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



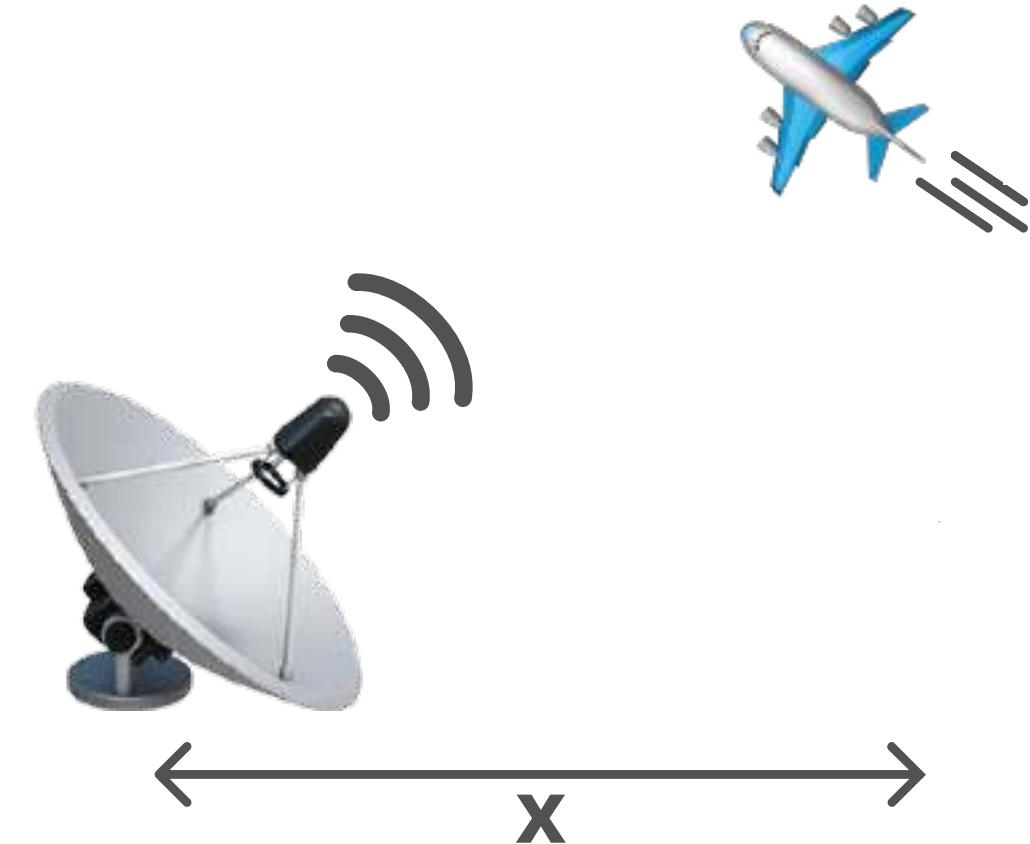
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



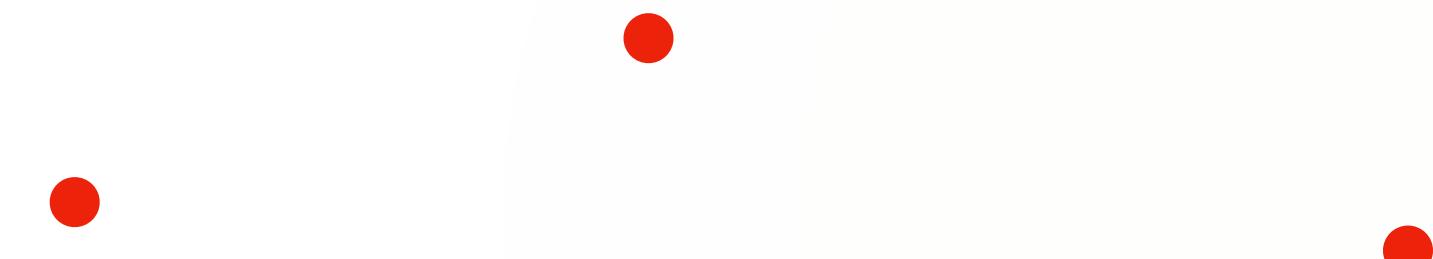
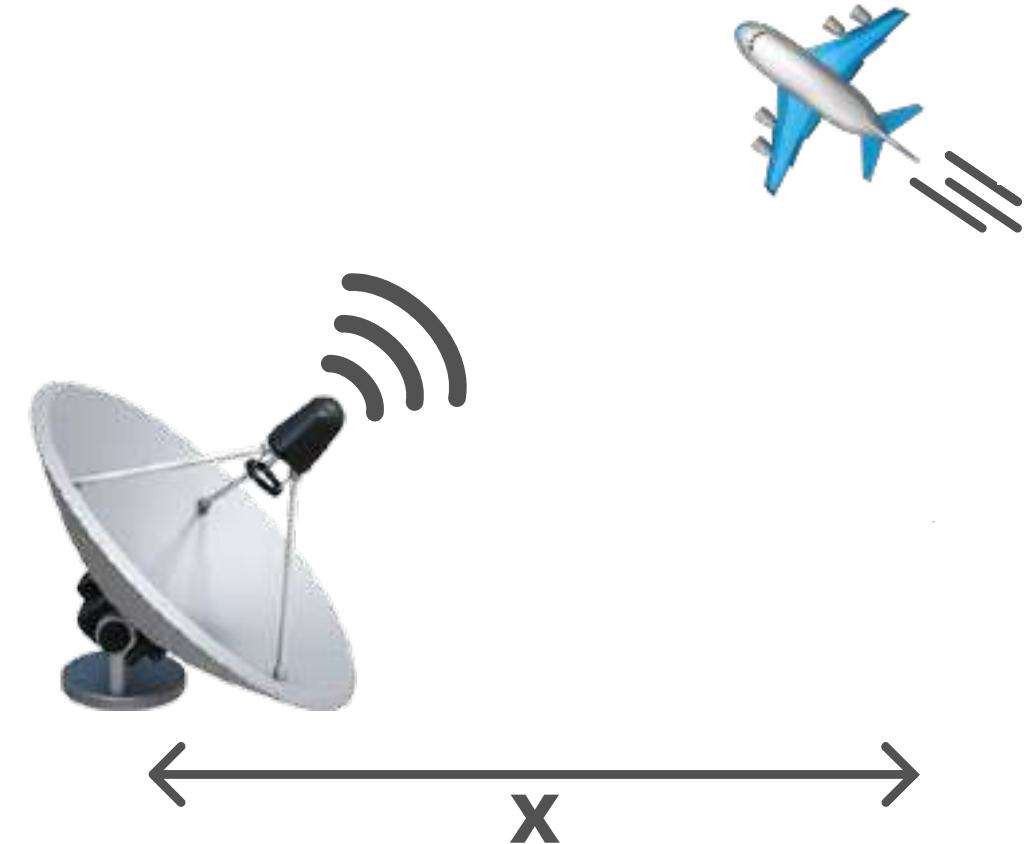
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



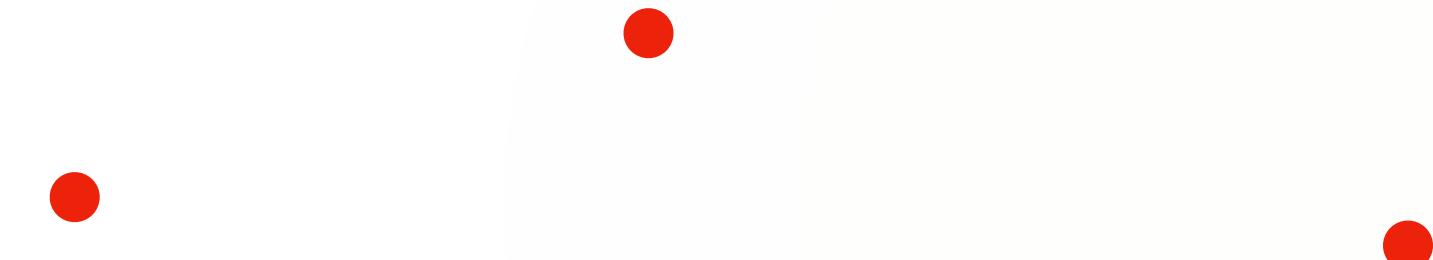
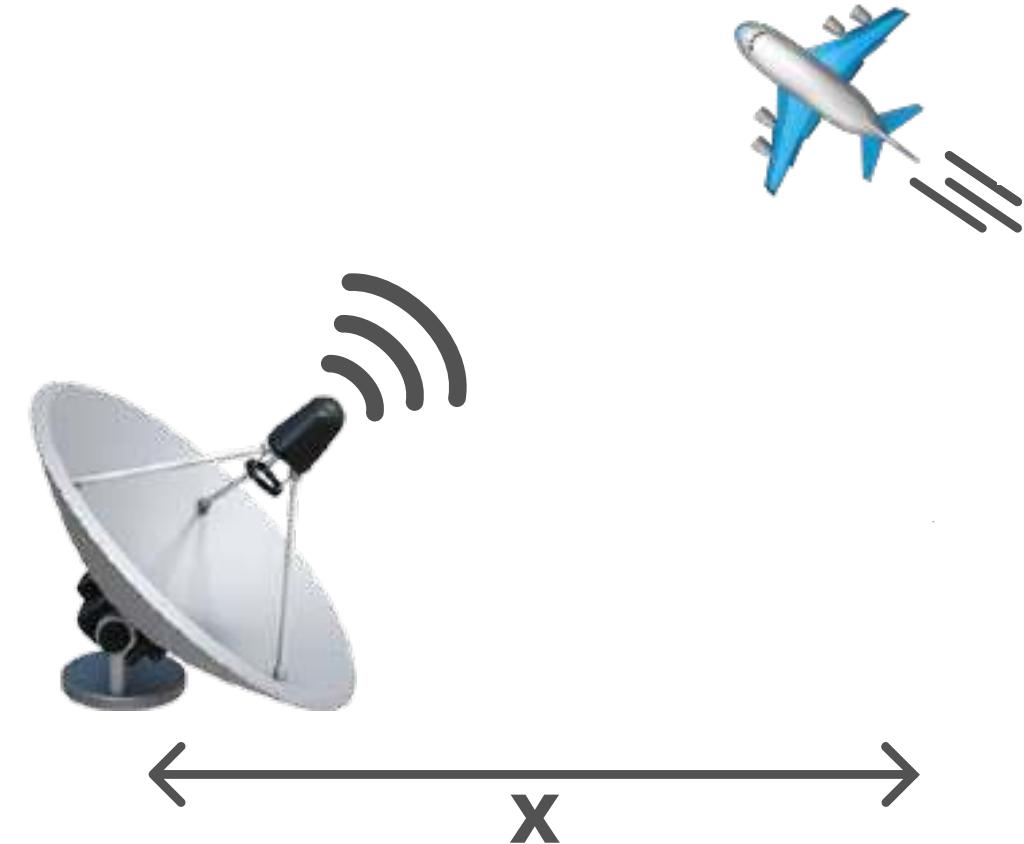
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



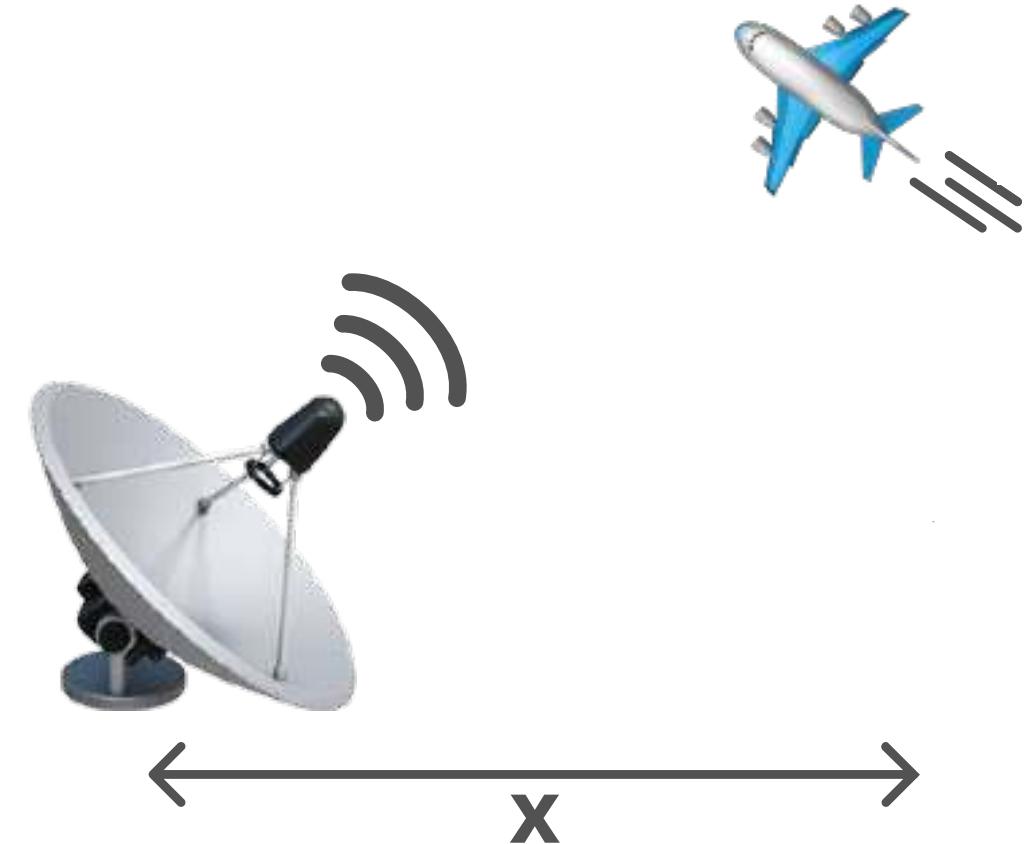
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



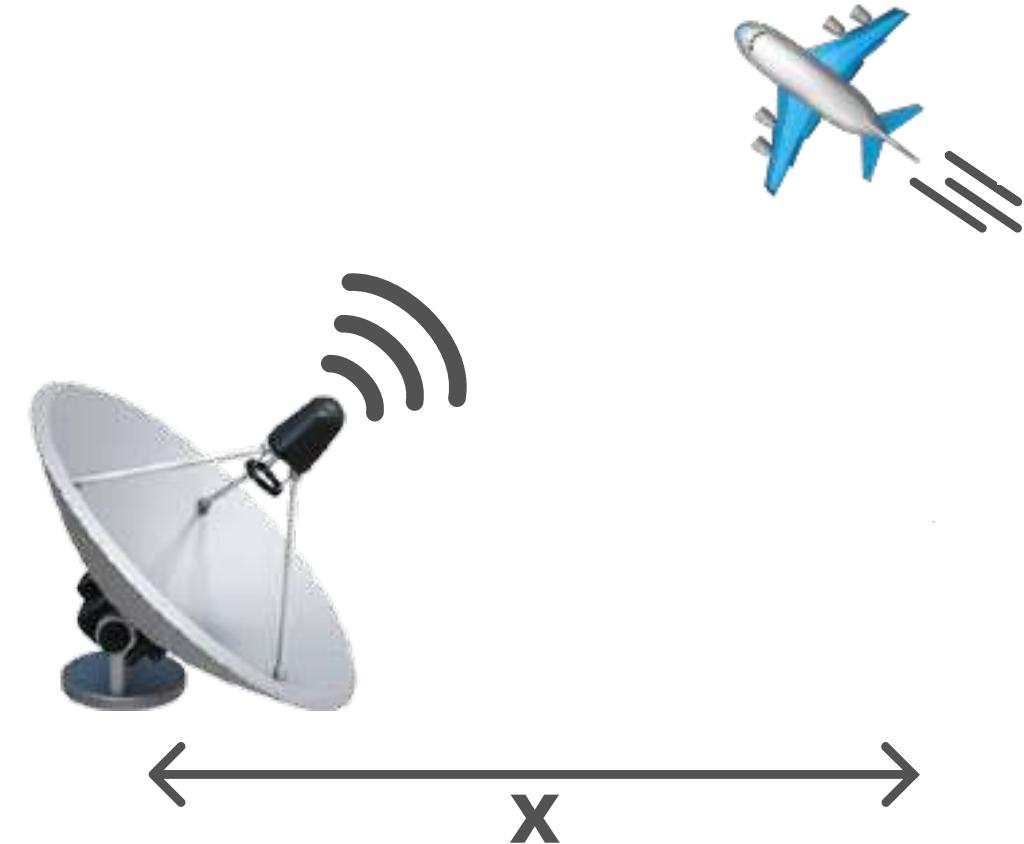
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



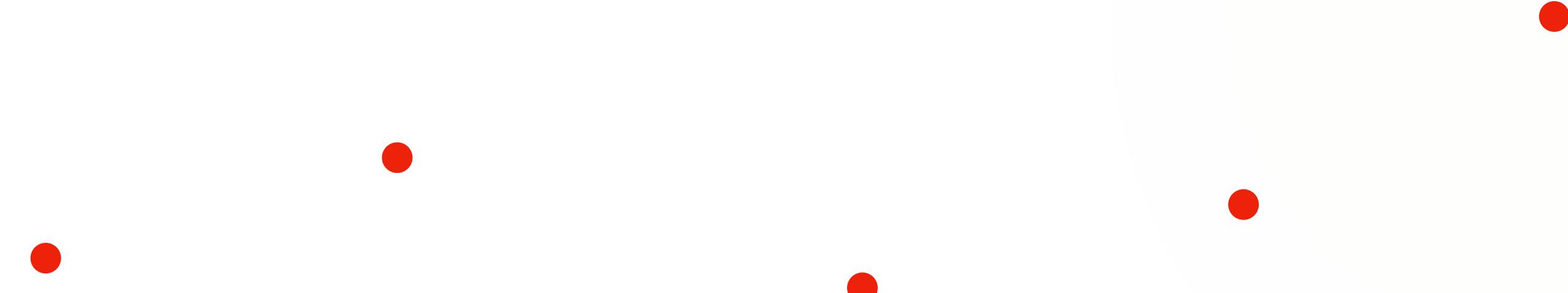
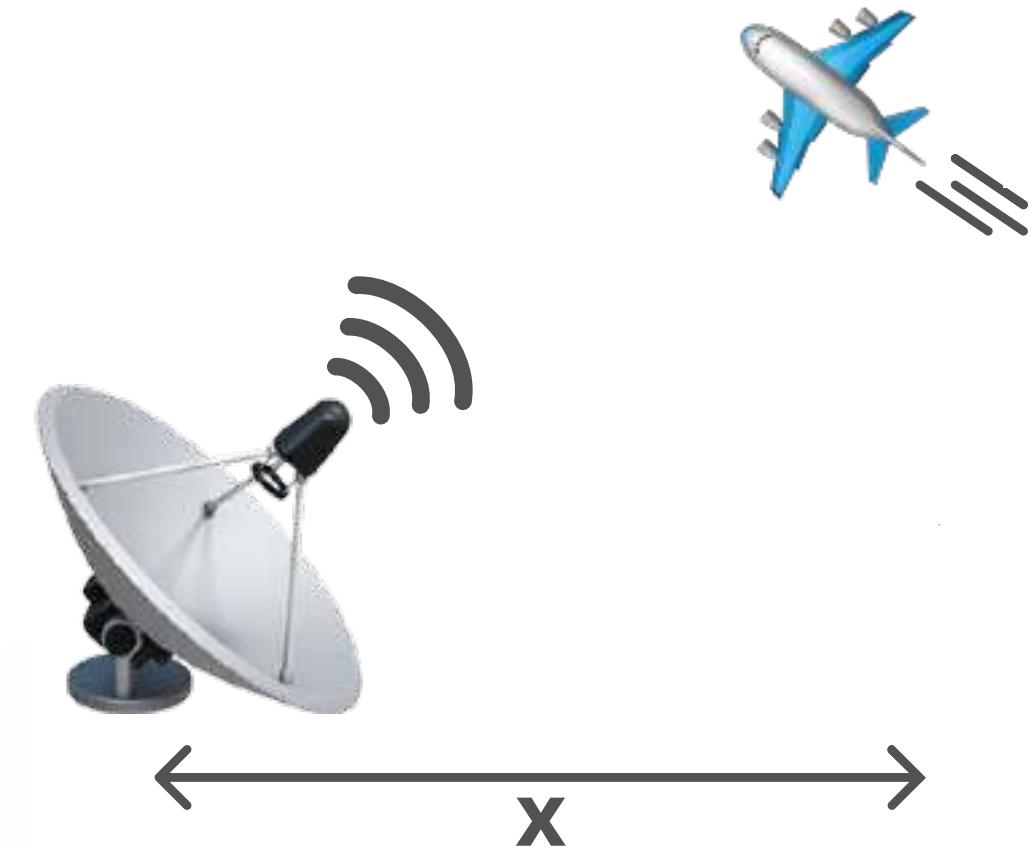
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



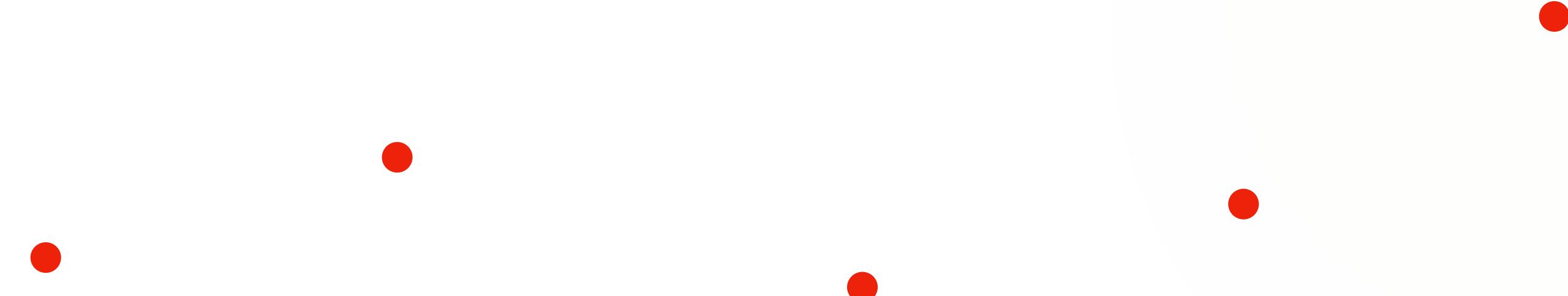
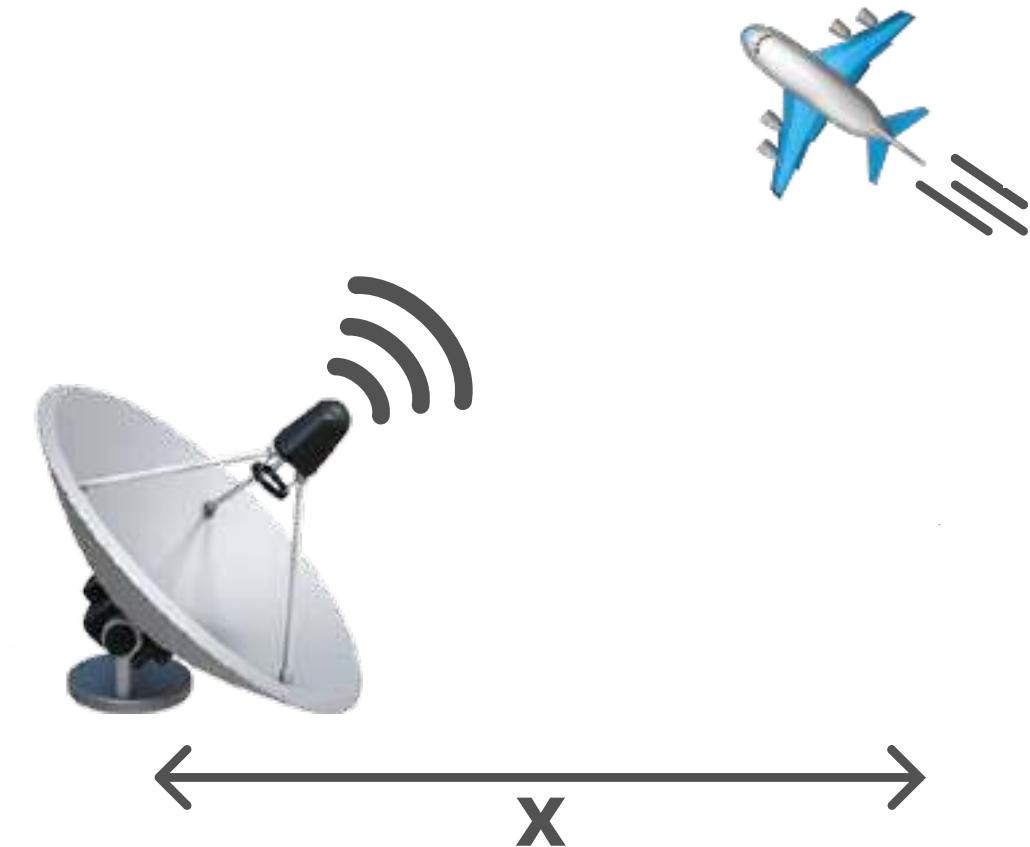
Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)        # condition on observation  
        acc.append(x)  
    return acc
```



Example: tracking

```
def tracker(data: list[float]) → list[float]:  
    acc = [0]  
    for y in data:                      # at each time step  
        pre_x = acc[-1]  
        x = sample(Gaussian(pre_x, 1))   # random walk  
        observe(Gaussian(x, 1), y)         # condition on observation  
        acc.append(x)  
    return acc
```



Better estimation

Digression: Kalman filter

Probabilistic Programming Languages

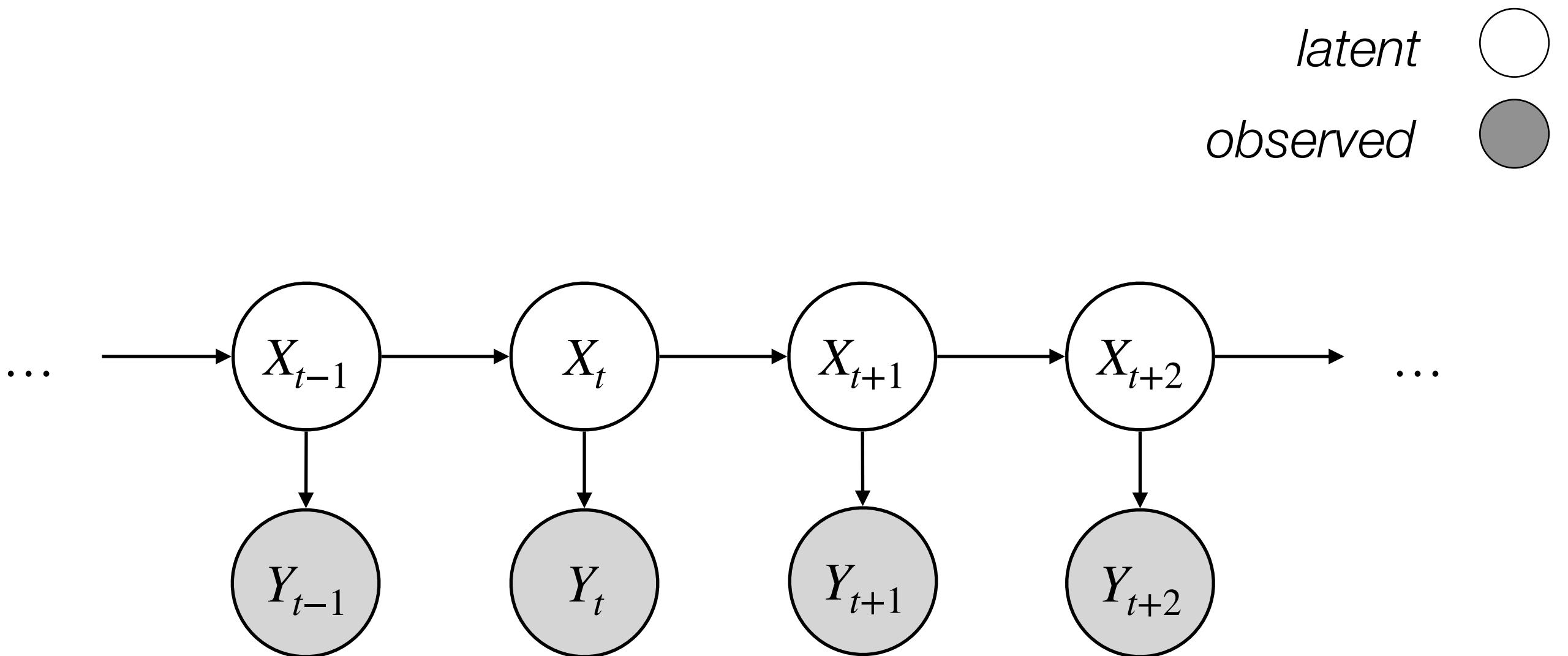
Example: tracker

Model

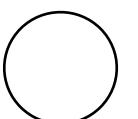
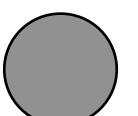
- Linear motion: $X_k \sim \mathcal{N}(F X_{k-1}, Q)$
- Observation: $Y_k \sim \mathcal{N}(H X_k, R)$

E.g., with Q and R constant noise matrices

- $X_k = \begin{pmatrix} p_k \\ v_k \end{pmatrix}$ (position, velocity)
- $F = \begin{pmatrix} 1 & dt \\ 0 & 1 \end{pmatrix}$ (discrete integration)
- $H = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ (projection)



Example: tracker

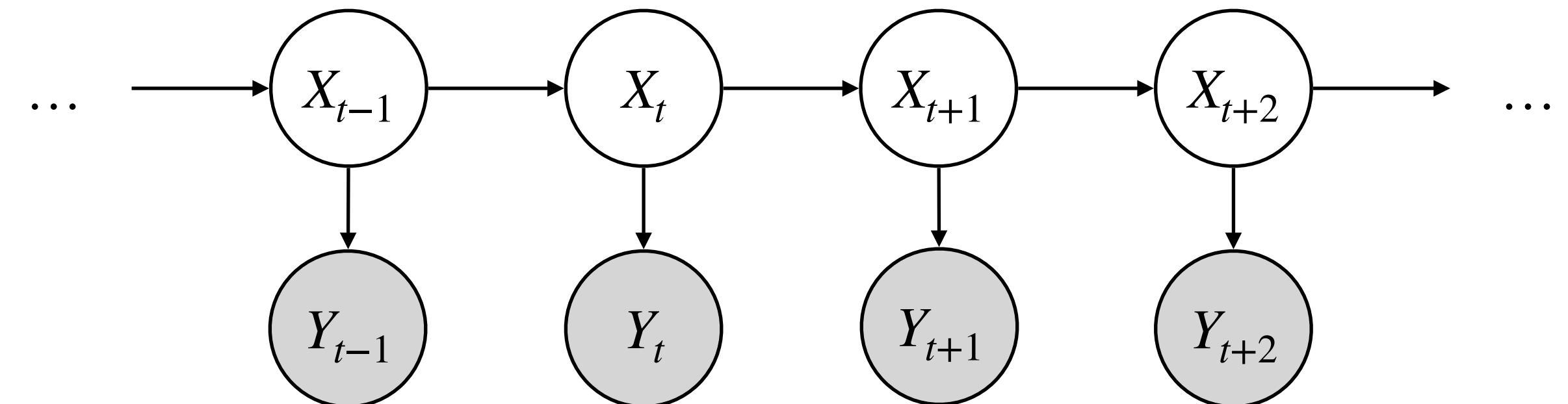
latent 
observed 

Model

- Linear motion: $X_k \sim \mathcal{N}(FX_{k-1}, Q)$
- Observation: $Y_k \sim \mathcal{N}(HX_k, R)$

E.g., with Q and R constant noise matrices

- $X_k = \begin{pmatrix} p_k \\ v_k \end{pmatrix}$ (position, velocity)
- $F = \begin{pmatrix} 1 & dt \\ 0 & 1 \end{pmatrix}$ (discrete integration)
- $H = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ (projection)



X_k^{pred}	$=$	FX_{k-1}^{est}
X_k^{est}	$=$	$X_k^{\text{pred}} + K_k(Y_k - HX_k^{\text{pred}})$
S_k	$=$	$(R + HP_k^{\text{pred}}H^T)^{-1}$
K_k	$=$	$P_k^{\text{pred}}H^TS_k$
P_k^{pred}	$=$	$Q + FP_{k-1}^{\text{est}}F^T$
P_k^{est}	$=$	$P_k^{\text{pred}} - K_kHP_k^{\text{pred}}$

Solution: Kalman filter

Demo

Semi-symbolic inference

Probabilistic Programming Languages

Semi-symbolic inference

Simple particles filters can be impractical

- Require lot of computing power
- Poor approximation

Exact inference is often possible

Rao-Blackwellized particle filters

- Each particle maintains symbolic distributions
- Perform as much exact computation as possible
- Fall back to a particle filter when symbolic computation fails

Main idea

- Keep track of conjugacy relationships
- Incorporate observations analytically
- Sample only when necessary

Semi-symbolic inference

Simple particles filters can be impractical

- Require lot of computing power
- Poor approximation

Exact inference is often possible

Rao-Blackwellized particle filters

- Each particle maintains symbolic distributions
- Perform as much exact computation as possible
- Fall back to a particle filter when symbolic computation fails

Main idea

- Keep track of conjugacy relationships
- Incorporate observations analytically
- Sample only when necessary

Example: Conjugate Gaussians

$$x \sim \mathcal{N}(\mu_0, \sigma_0^2)$$

$$y \sim \mathcal{N}(x, \sigma^2)$$

$$x | (y = v) \sim \mathcal{N}(\mu_1, \sigma_1^2)$$

$$\mu_1 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2} \right)^{-1} \left(\frac{\mu_0}{\sigma_0^2} + \frac{v}{\sigma^2} \right)$$

$$\sigma_1^2 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2} \right)^{-1}$$

Semi-symbolic inference

```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

Semi-symbolic inference

$t = 0$

```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

Semi-symbolic inference

$t = 0$

```
sample (gaussian (0, 1))
```

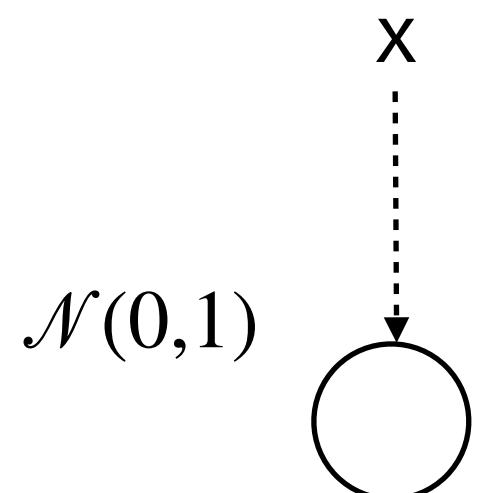
```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

Semi-symbolic inference

```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))
```

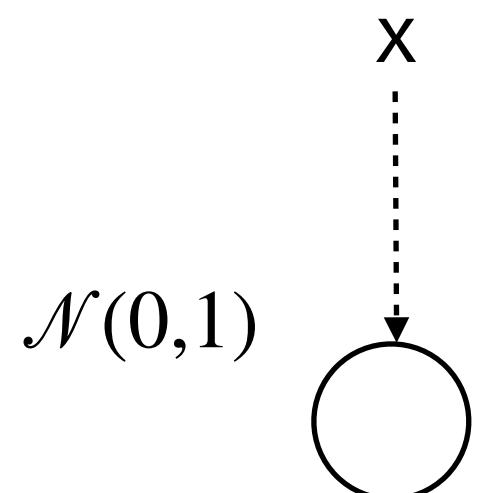


Semi-symbolic inference

```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))  
observe (gaussian (x, 1), 3)
```

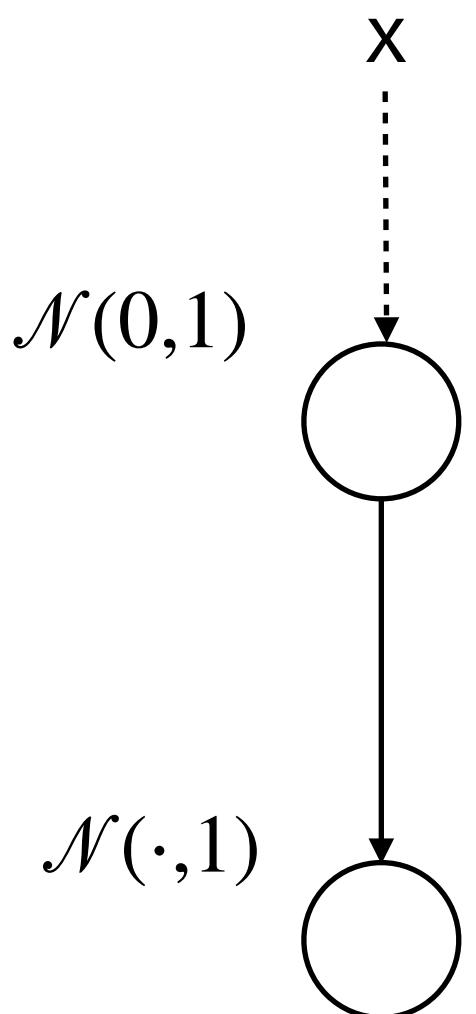


Semi-symbolic inference

```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))  
observe (gaussian (x, 1), 3)
```

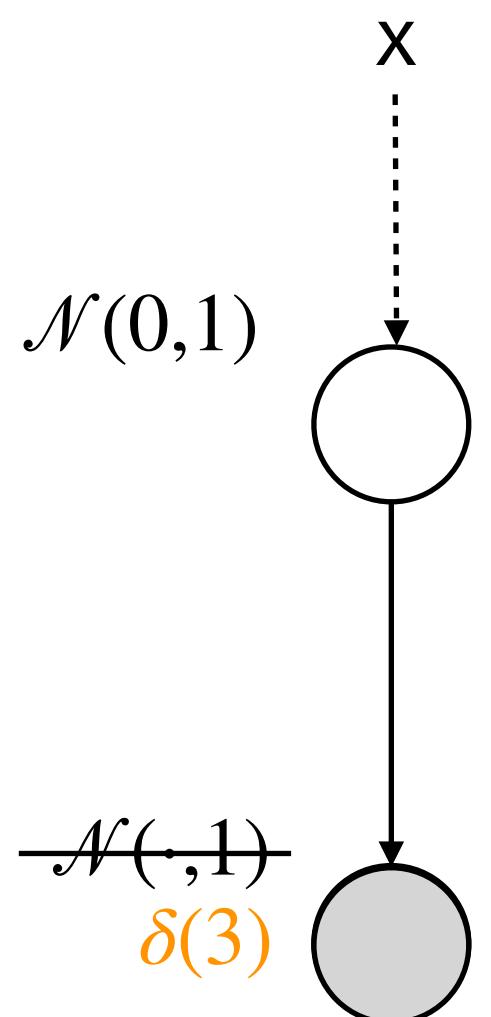


Semi-symbolic inference

```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))  
observe (gaussian (x, 1), 3)
```

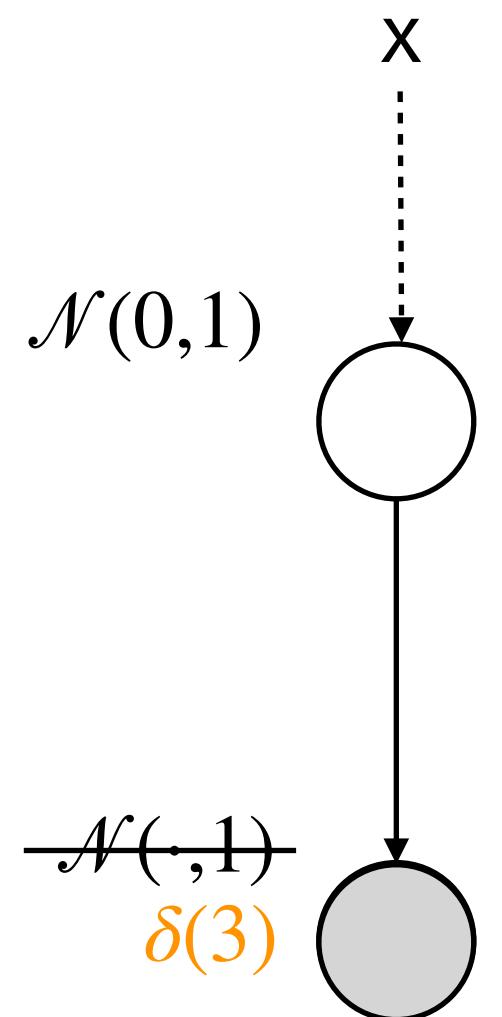


Semi-symbolic inference

```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))  
observe (gaussian (x, 1), 3)
```



Example: Conjugate Gaussians

$$x \sim \mathcal{N}(\mu_0, \sigma_0^2)$$

$$y \sim \mathcal{N}(x, \sigma^2)$$

$$x | (y = v) \sim \mathcal{N}(\mu_1, \sigma_1^2)$$

$$\mu_1 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2} \right)^{-1} \left(\frac{\mu_0}{\sigma_0^2} + \frac{v}{\sigma^2} \right)$$

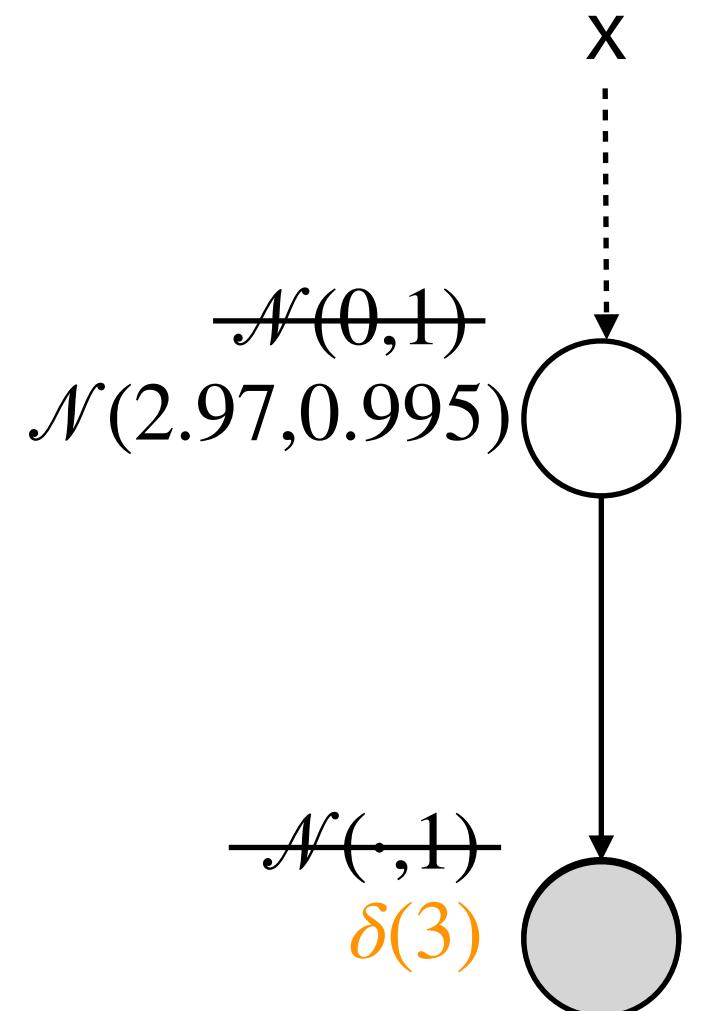
$$\sigma_1^2 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2} \right)^{-1}$$

Semi-symbolic inference

```
for y in data:
    pre_x = acc[-1]
    x = sample(gaussian(pre_x, 1))
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))
observe (gaussian (x, 1), 3)
```



Example: Conjugate Gaussians

$$x \sim \mathcal{N}(\mu_0, \sigma_0^2)$$

$$y \sim \mathcal{N}(x, \sigma^2)$$

$$x | (y = v) \sim \mathcal{N}(\mu_1, \sigma_1^2)$$

$$\mu_1 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2} \right)^{-1} \left(\frac{\mu_0}{\sigma_0^2} + \frac{v}{\sigma^2} \right)$$

$$\sigma_1^2 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2} \right)^{-1}$$

Semi-symbolic inference

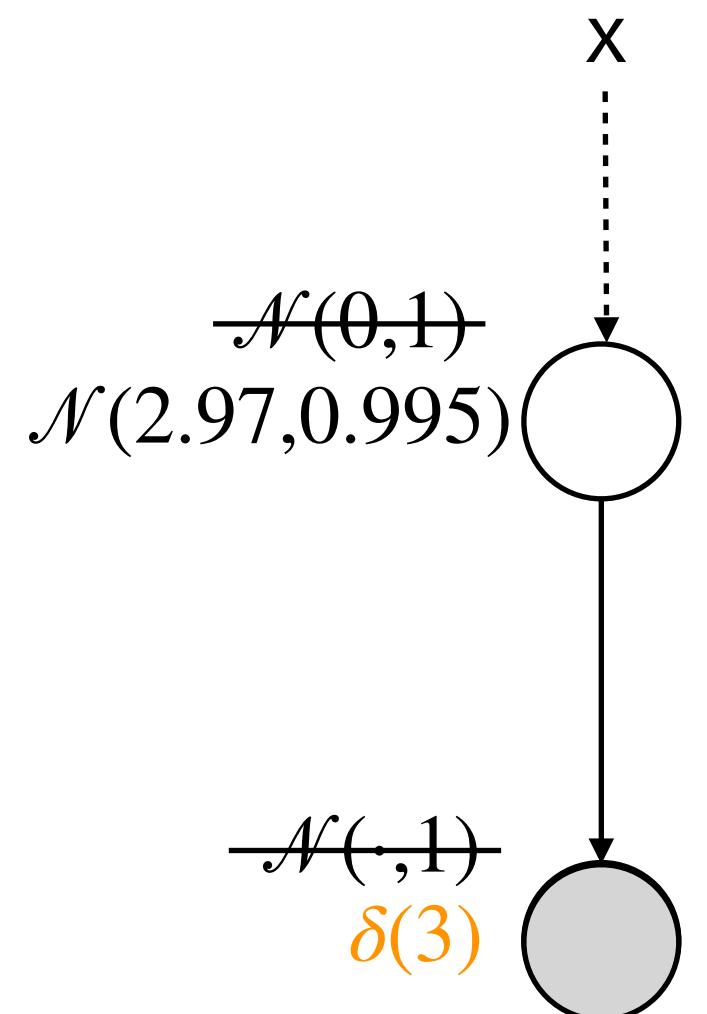
```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))  
observe (gaussian (x, 1), 3)
```

$t = 1$

```
sample (gaussian (pre_x, 1))
```



Semi-symbolic inference

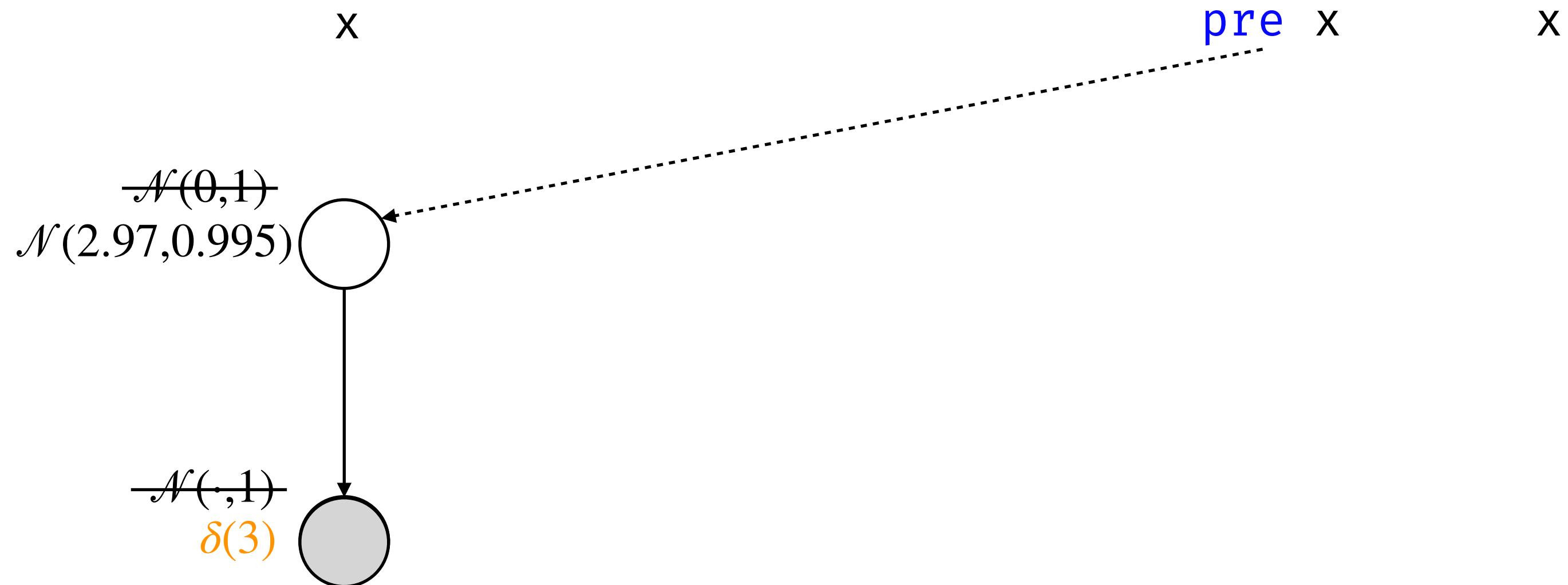
```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))  
observe (gaussian (x, 1), 3)
```

$t = 1$

```
sample (gaussian (pre_x, 1))
```



Semi-symbolic inference

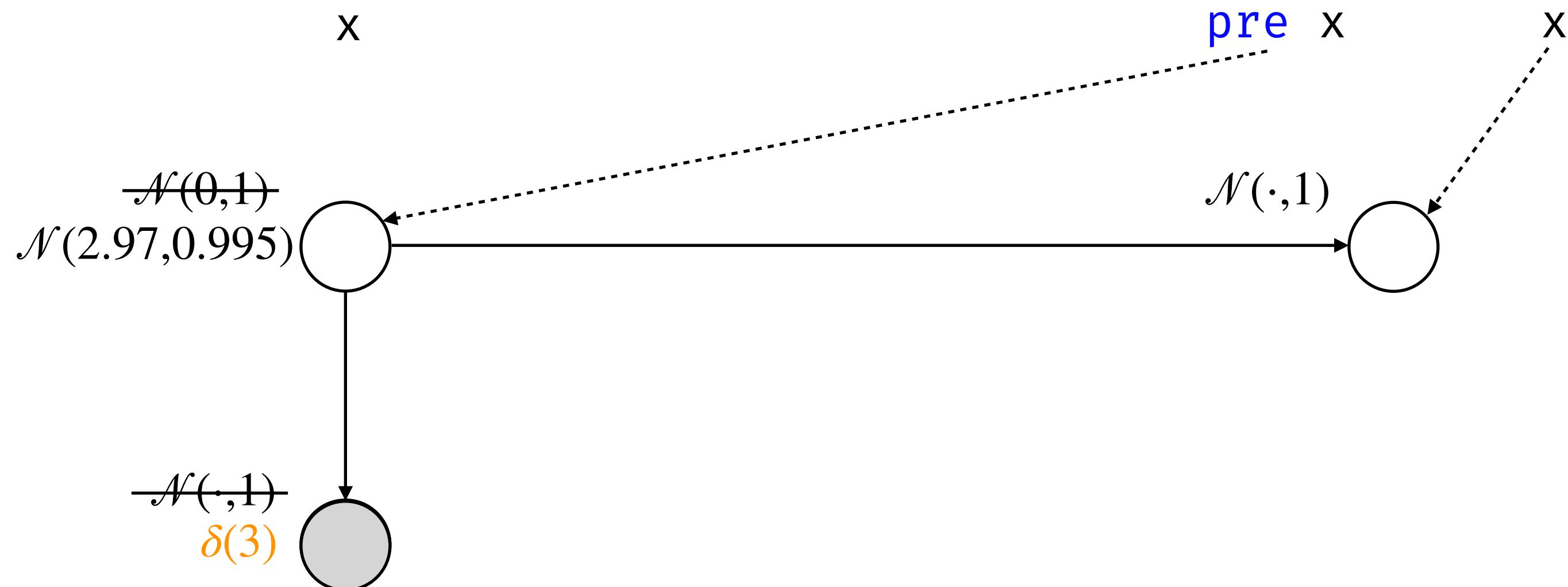
```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))  
observe (gaussian (x, 1), 3)
```

$t = 1$

```
sample (gaussian (pre_x, 1))
```



Semi-symbolic inference

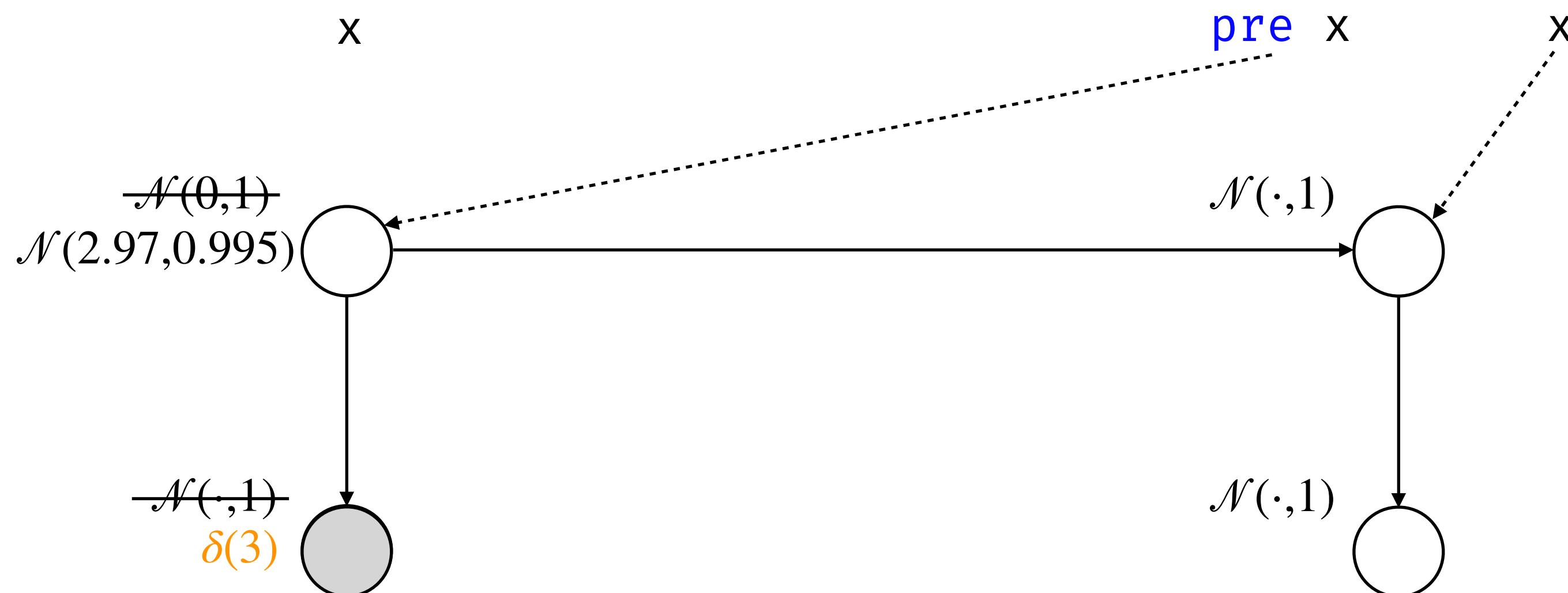
```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))  
observe (gaussian (x, 1), 3)
```

$t = 1$

```
sample (gaussian (pre_x, 1))  
observe (gaussian (x, 1), 5)
```



Semi-symbolic inference

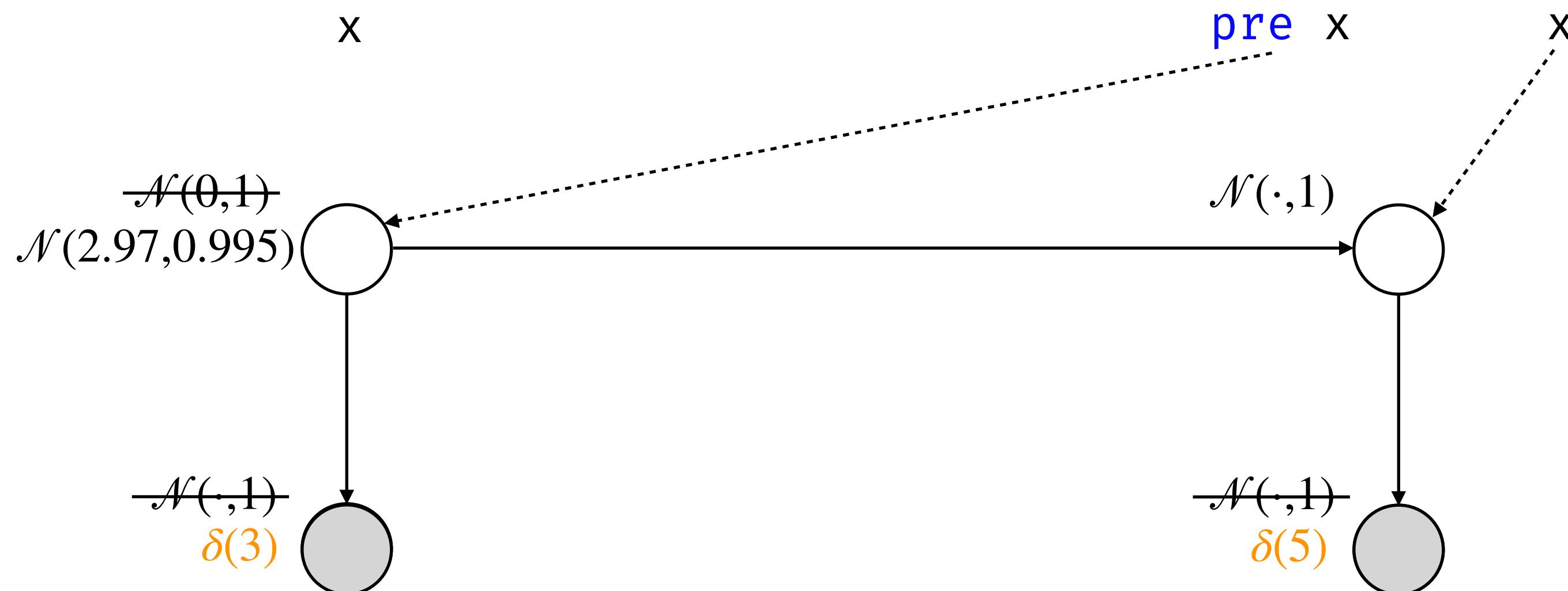
```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))  
observe (gaussian (x, 1), 3)
```

$t = 1$

```
sample (gaussian (pre_x, 1))  
observe (gaussian (x, 1), 5)
```



Semi-symbolic inference

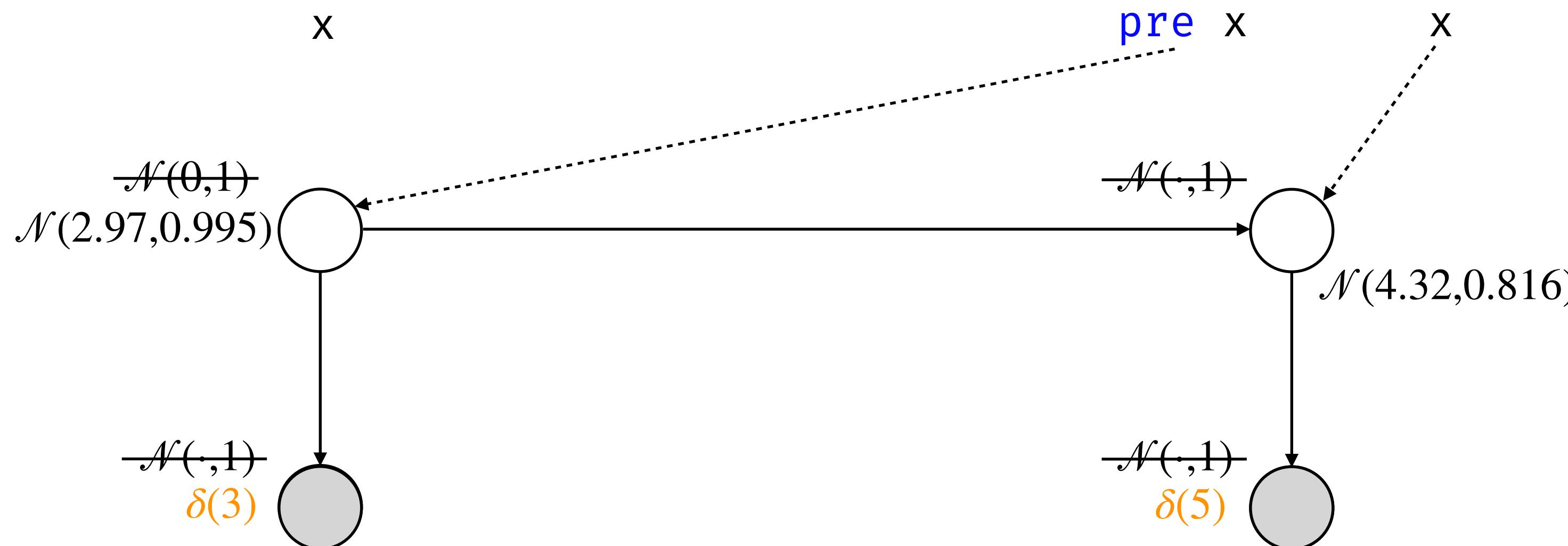
```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 1))  
observe (gaussian (x, 1), 3)
```

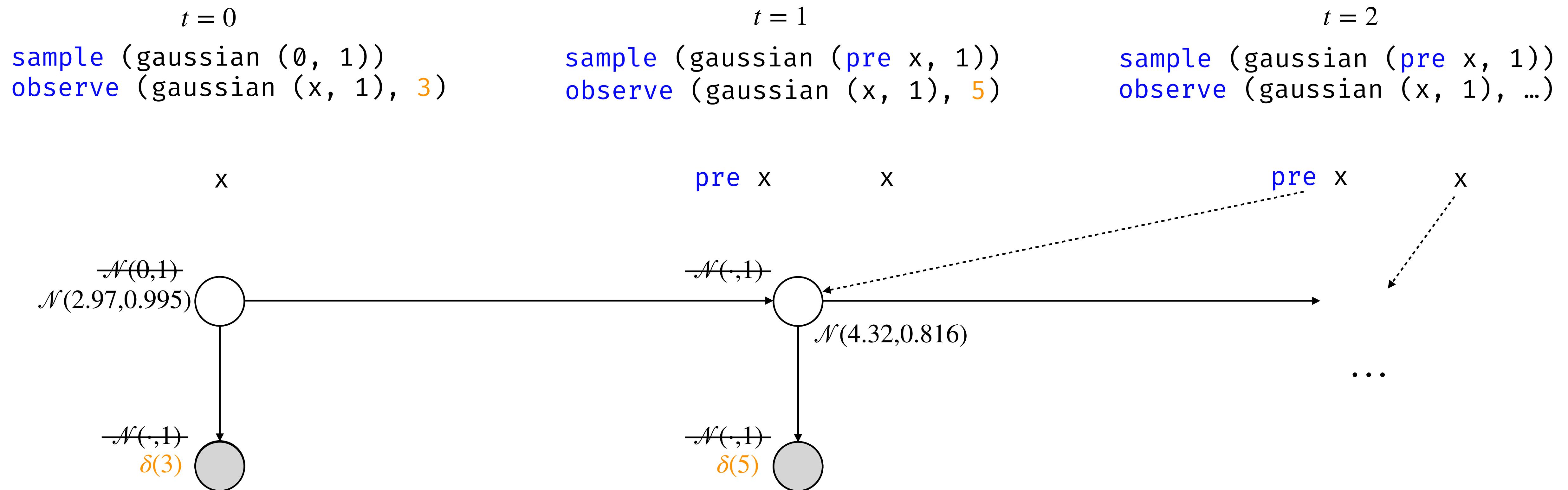
$t = 1$

```
sample (gaussian (pre_x, 1))  
observe (gaussian (x, 1), 5)
```



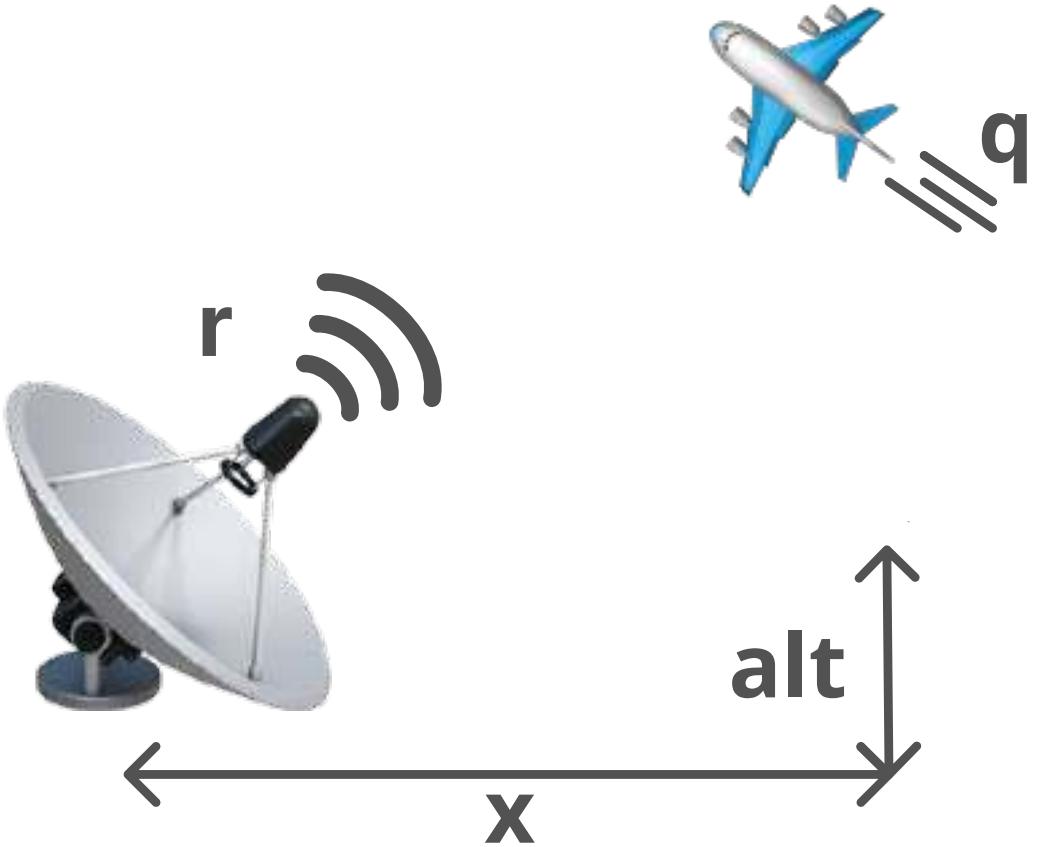
Semi-symbolic inference

```
for y in data:  
    pre_x = acc[-1]  
    x = sample(gaussian(pre_x, 1))  
    observe(gaussian(x, 1), y)
```



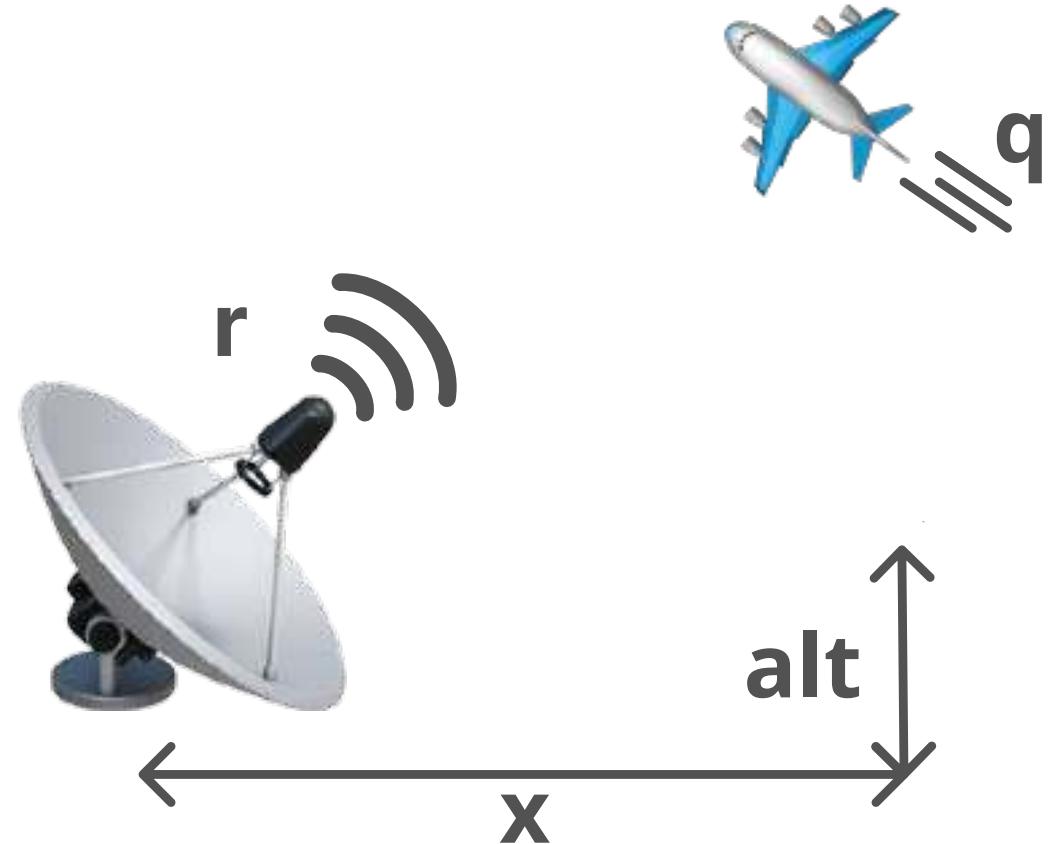
Example: tracker++

```
def tracker(data: list[X * A]) → list[X * A]:  
  
    # q and r are constants  
  
    acc = [(0,0)]  
    for (xo, ao) in data:                      # at each time step  
        pre_x, pre_a = acc[-1]  
        x = sample(Gaussian(pre_x, q))          # random walk  
        a = sample(Gaussian(pre_a, q))  
  
        observe(Gaussian(x, r), xo)               # condition on observations  
        observe(Gaussian(a, r), ao)  
        acc.append((x, a))  
  
    return acc
```

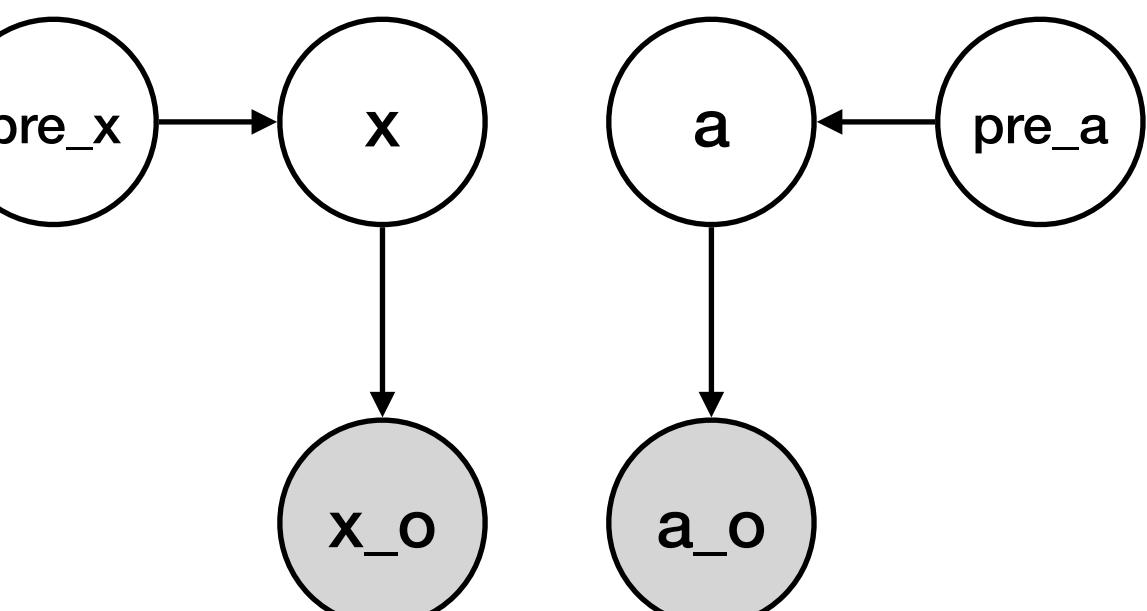


Example: tracker++

```
def tracker(data: list[X * A]) → list[X * A]:  
  
    # q and r are constants  
  
    acc = [(0,0)]  
    for (xo, ao) in data:                      # at each time step  
        pre_x, pre_a = acc[-1]  
        x = sample(Gaussian(pre_x, q))          # random walk  
        a = sample(Gaussian(pre_a, q))  
  
        observe(Gaussian(x, r), xo)               # condition on obse  
        observe(Gaussian(a, r), ao)  
        acc.append((x, a))  
  
    return acc
```

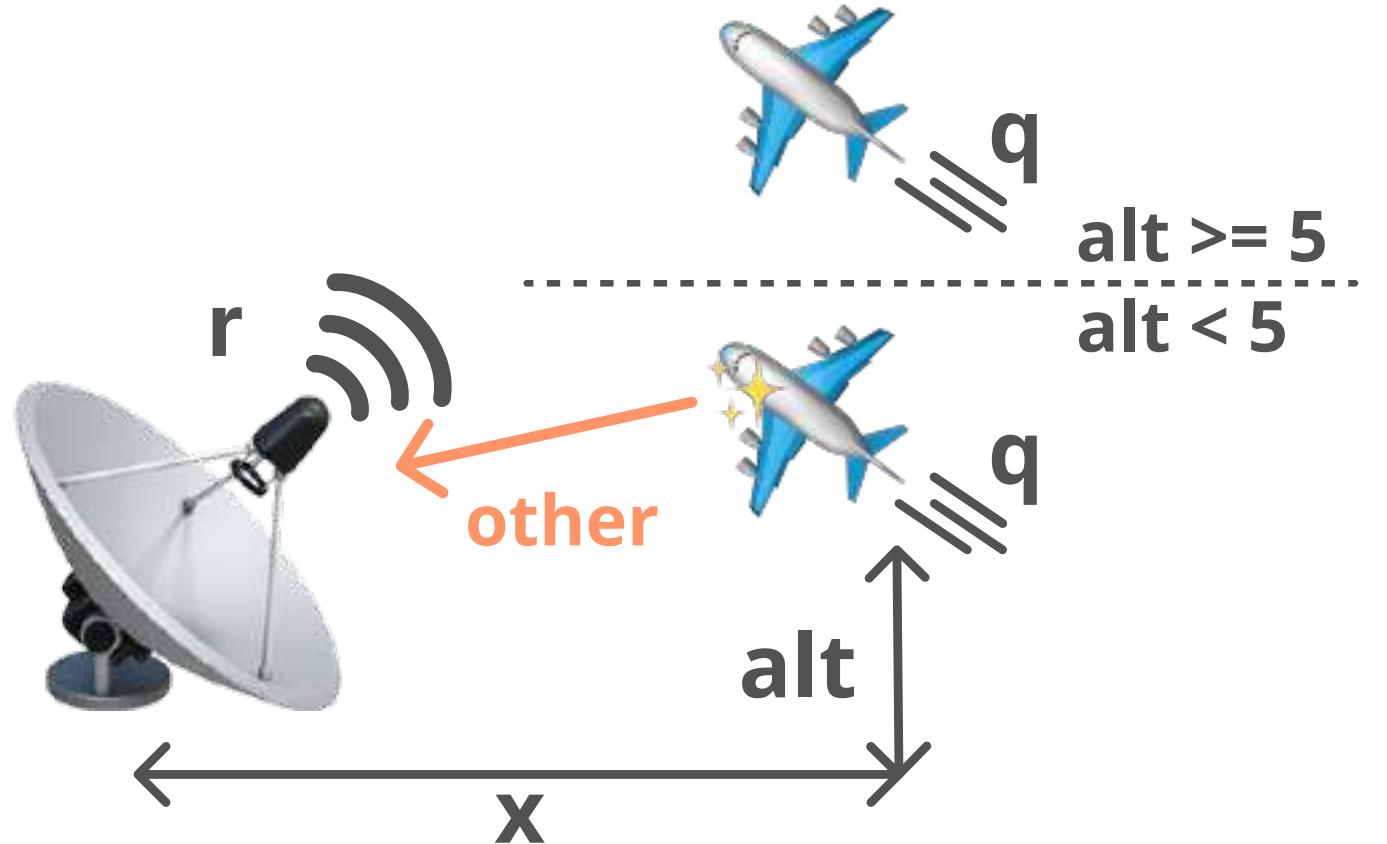


Exact solution



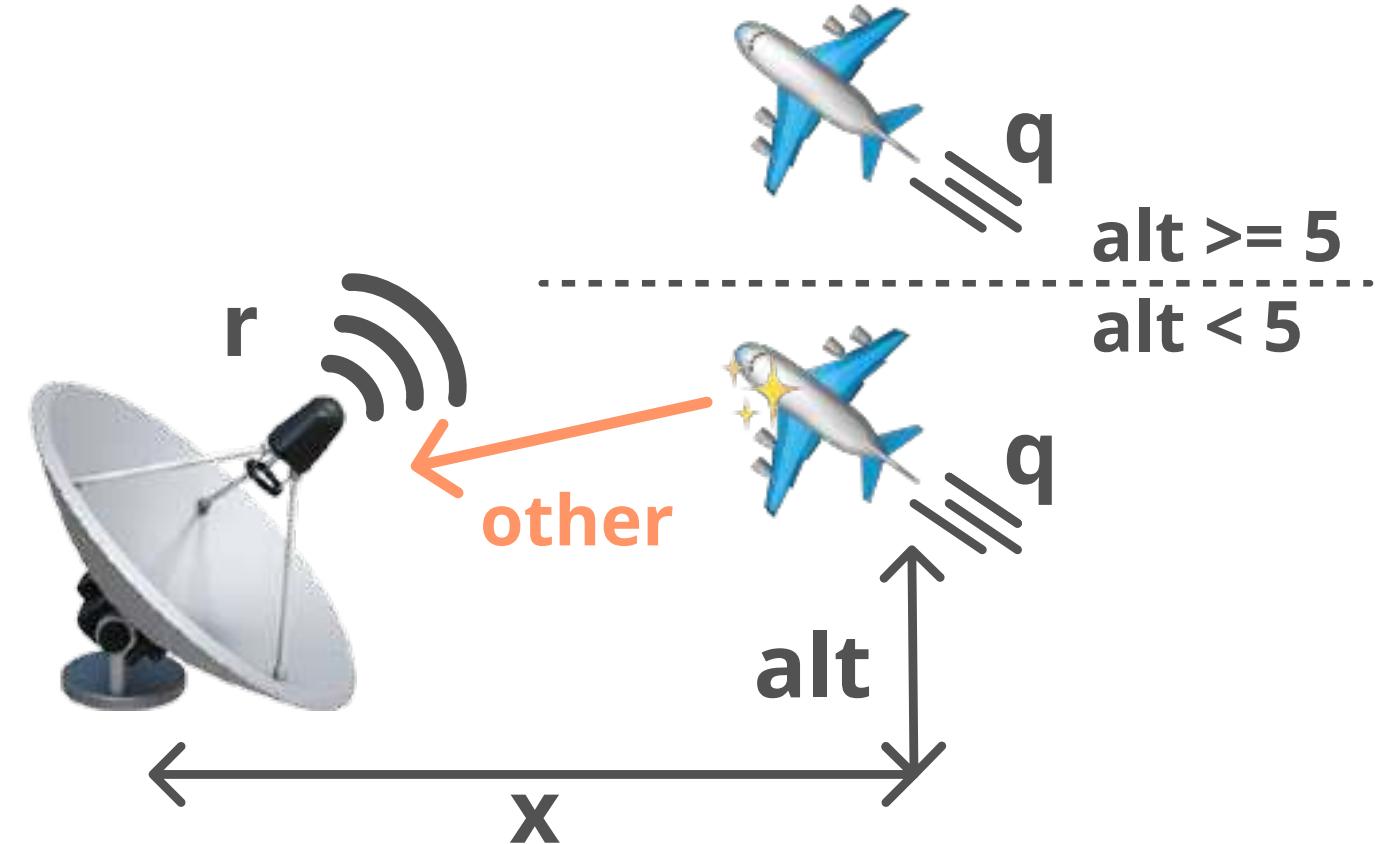
Example: tracker++

```
def tracker(data: list[X * A]) → list[X * A]:  
  
    # q and v are constants  
  
    acc = [(0,0)]  
    for (xo, ao) in data:                      # at each time step  
        pre_x, pre_a = acc[-1]  
        x = sample(Gaussian(pre_x, q))          # random walk  
        a = sample(Gaussian(pre_a, q))  
        other = sample(InvGamma(1., 1.)) # spiking noise at low alt  
        v = r + other if a < 5 else r  
        observe(Gaussian(x, v), xo)             # condition on observations  
        observe(Gaussian(a, v), ao)  
        acc.append((x, a))  
  
    return acc
```



Example: tracker++

```
def tracker(data: list[X * A]) → list[X * A] * float * float:  
    q = sample(InvGamma(1., 1.)) # move noise  
    r = sample(InvGamma(1., 1.)) # observation noise  
  
    acc = [(0,0)]  
    for (xo, ao) in data: # at each time step  
        pre_x, pre_a = acc[-1]  
        x = sample(Gaussian(pre_x, q)) # random walk  
        a = sample(Gaussian(pre_a, q))  
        other = sample(InvGamma(1., 1.)) # spiking noise at low alt  
        v = r + other if a < 5 else r  
        observe(Gaussian(x, v), xo) # condition on observations  
        observe(Gaussian(a, v), ao)  
        acc.append((x, a))  
  
    return acc, q, r
```



Example: tracker++

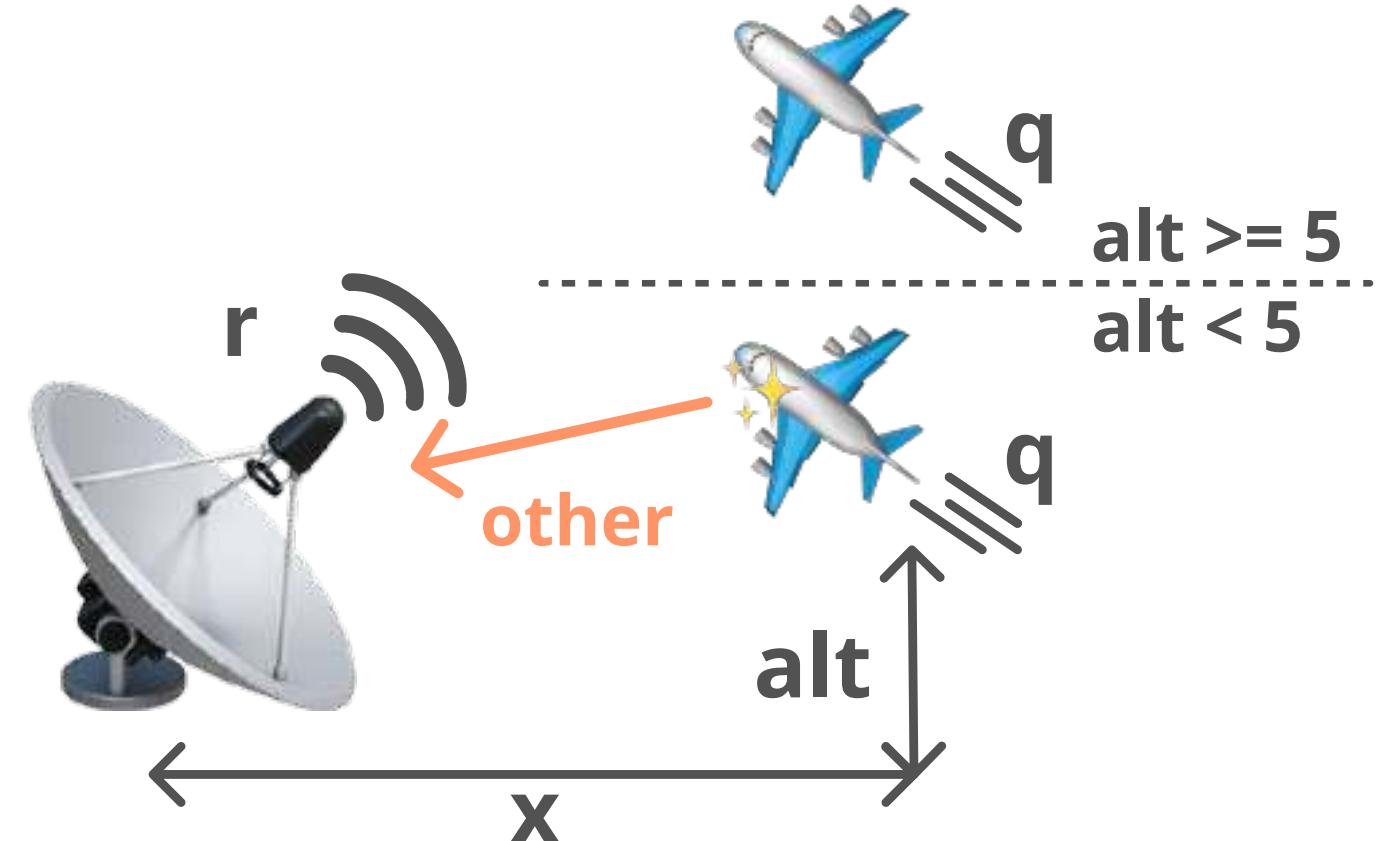
```

def tracker(data: list[X * A]) → list[X * A] * float * float:
    q = sample(InvGamma(1., 1.)) # move noise
    r = sample(InvGamma(1., 1.)) # observation noise

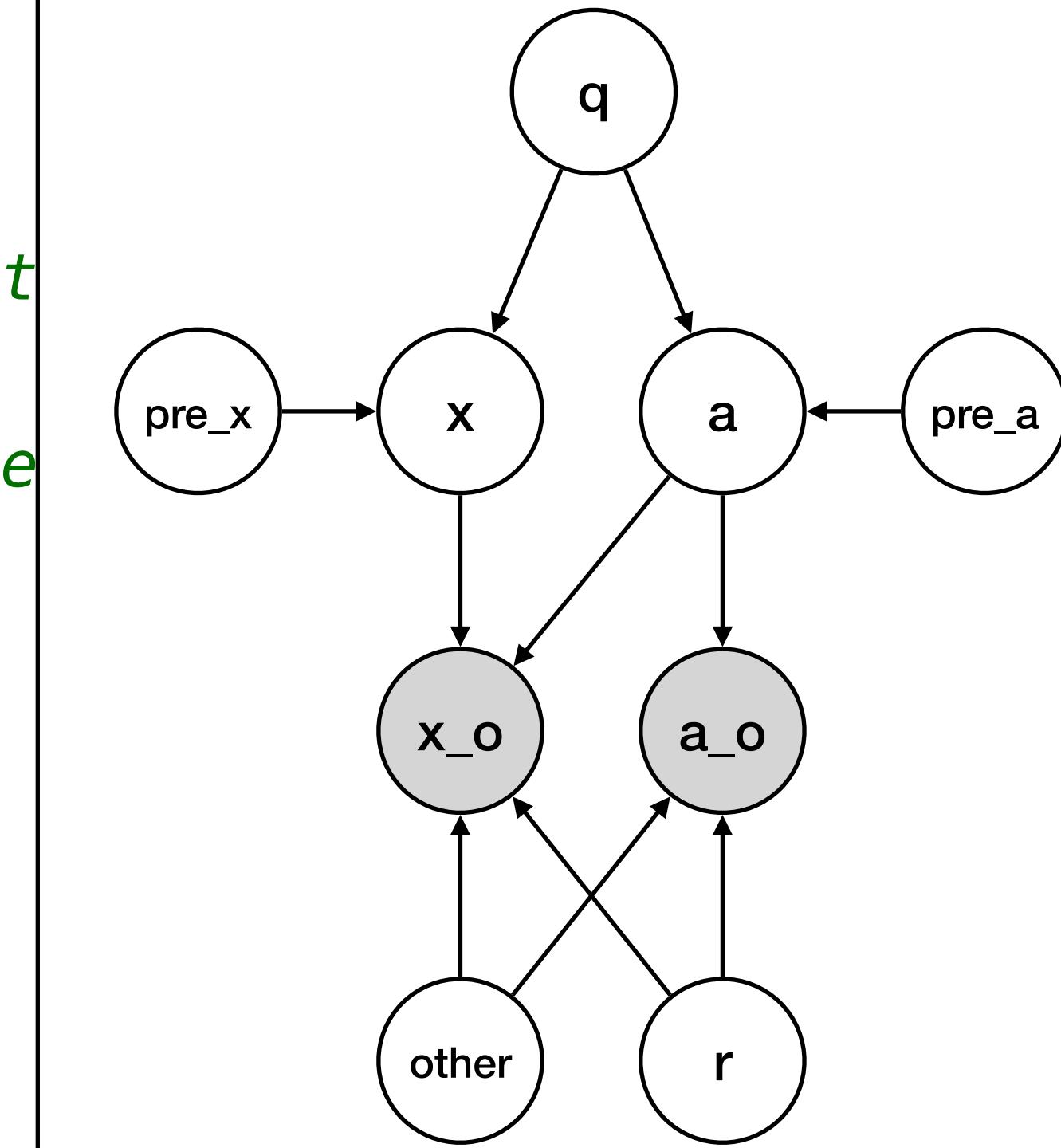
    acc = [(0,0)]
    for (xo, ao) in data:           # at each time step
        pre_x, pre_a = acc[-1]
        x = sample(Gaussian(pre_x, q)) # random walk
        a = sample(Gaussian(pre_a, q))
        other = sample(InvGamma(1., 1.)) # spiking noise at
        v = r + other if a < 5 else r
        observe(Gaussian(x, v), xo)   # condition on obse
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r

```

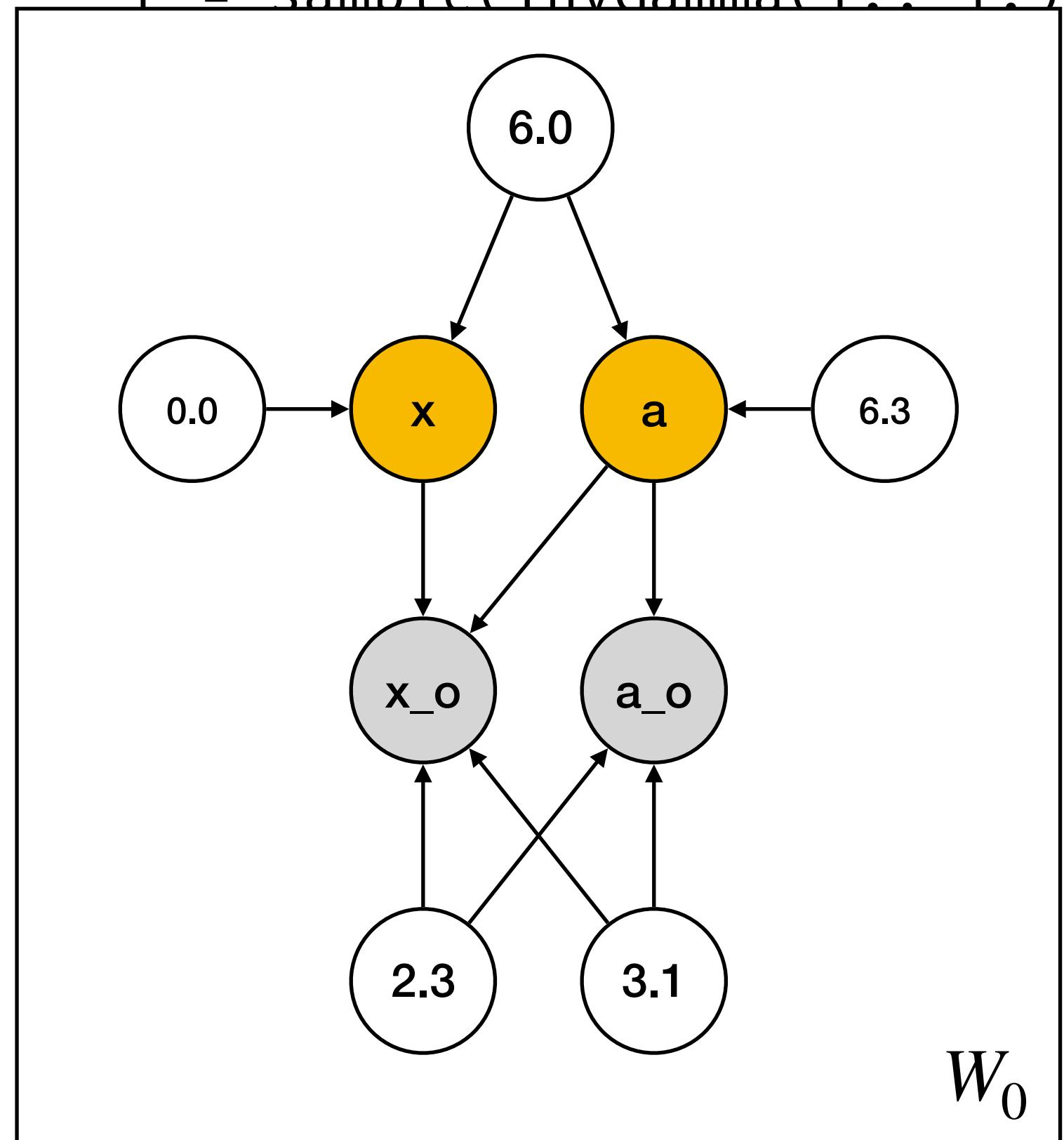


at each time step **No analytical solution**

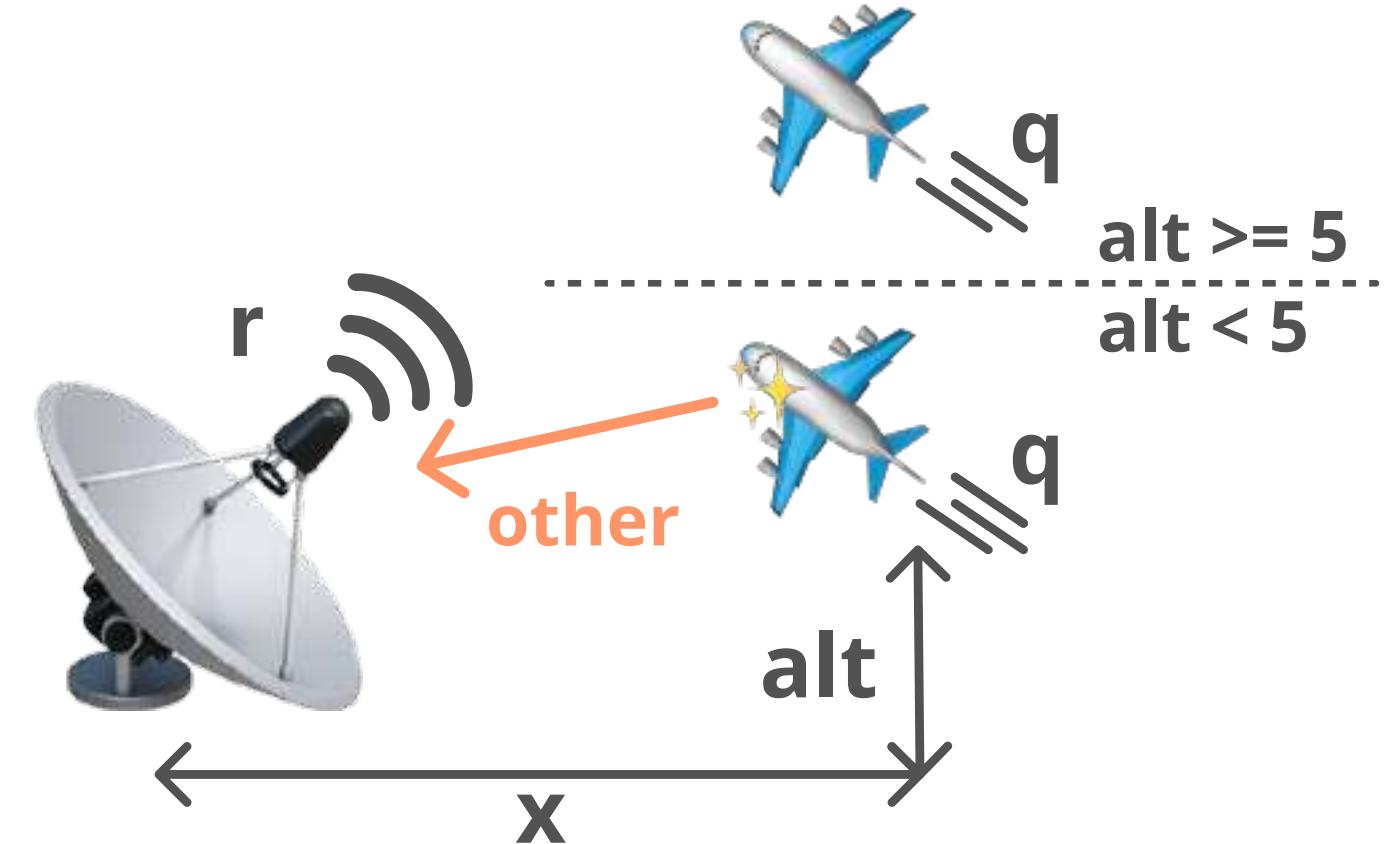


Semi-symbolic inference

```
def tracker(data: list[X * A]) → list[X * A] * float * float:  
    q = sample(InvGamma(1., 1.)) # move noise  
    r = sample(TinvGamma(1.. 1.)) # observation noise
```

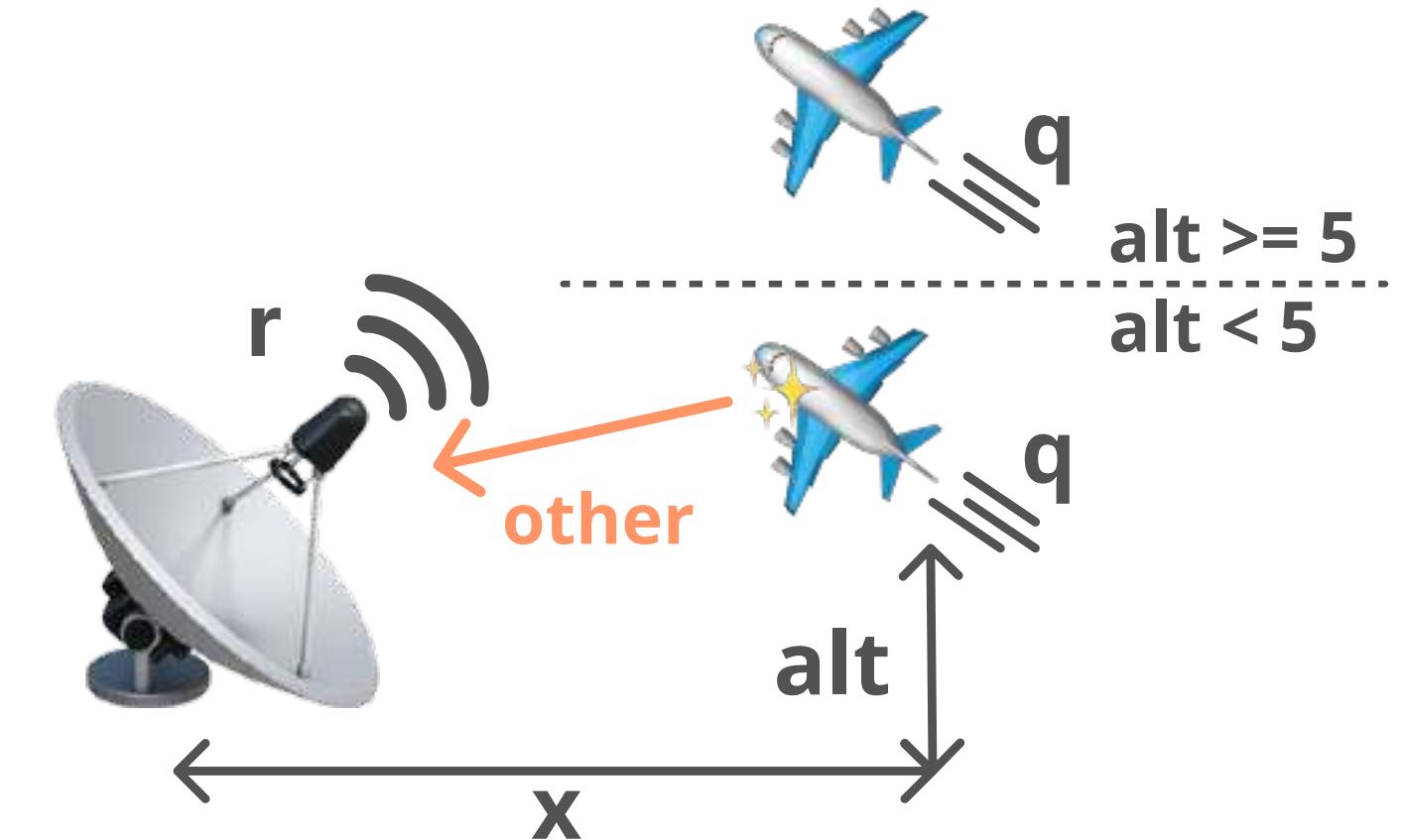
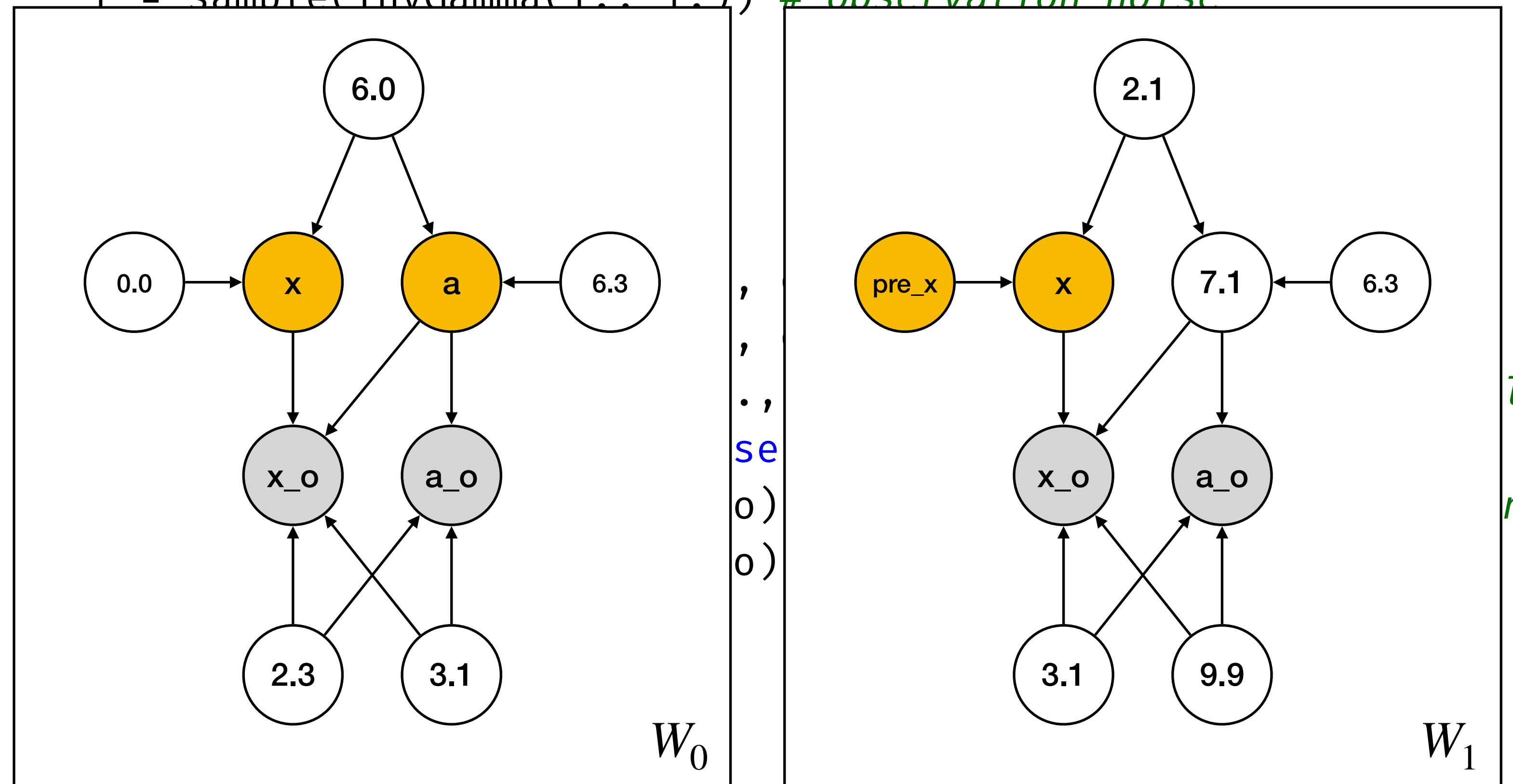


```
# at each time step  
, q)) # random walk  
, q))  
. , 1.)) # spkiking noise at low alt  
se r  
o)  
o)  
# condition on observations
```



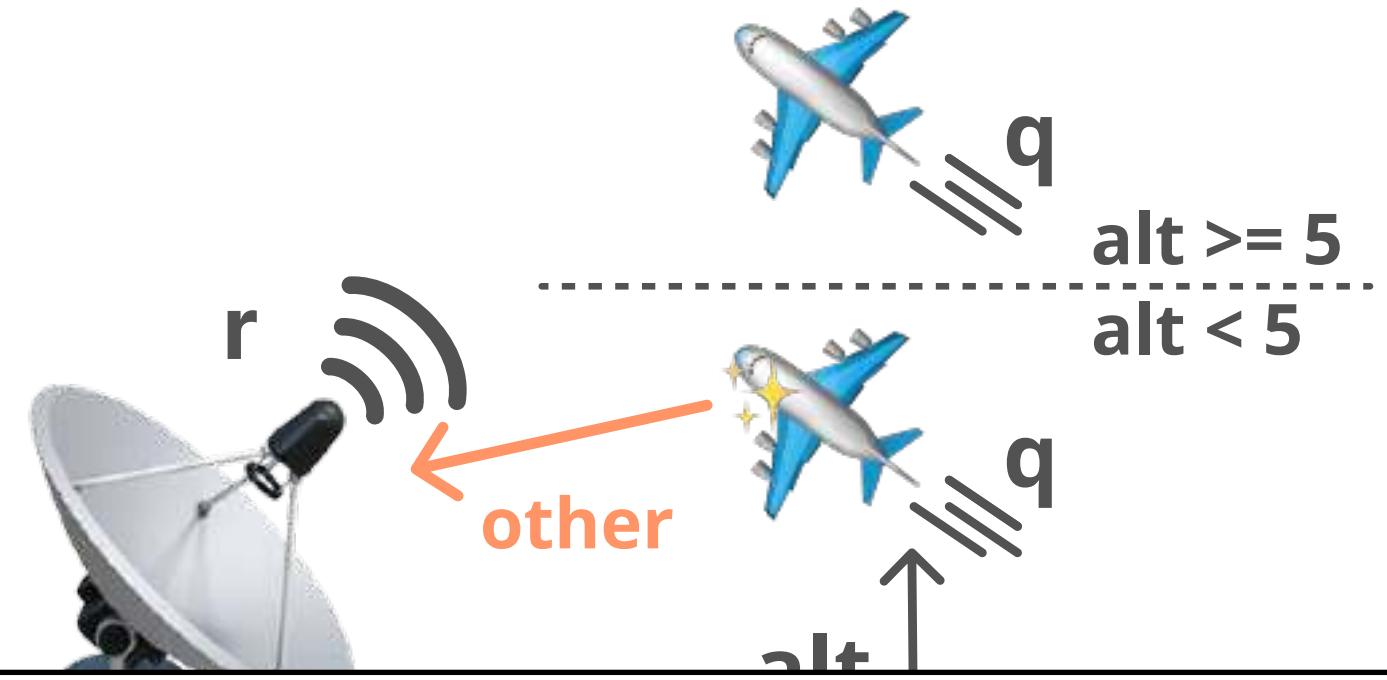
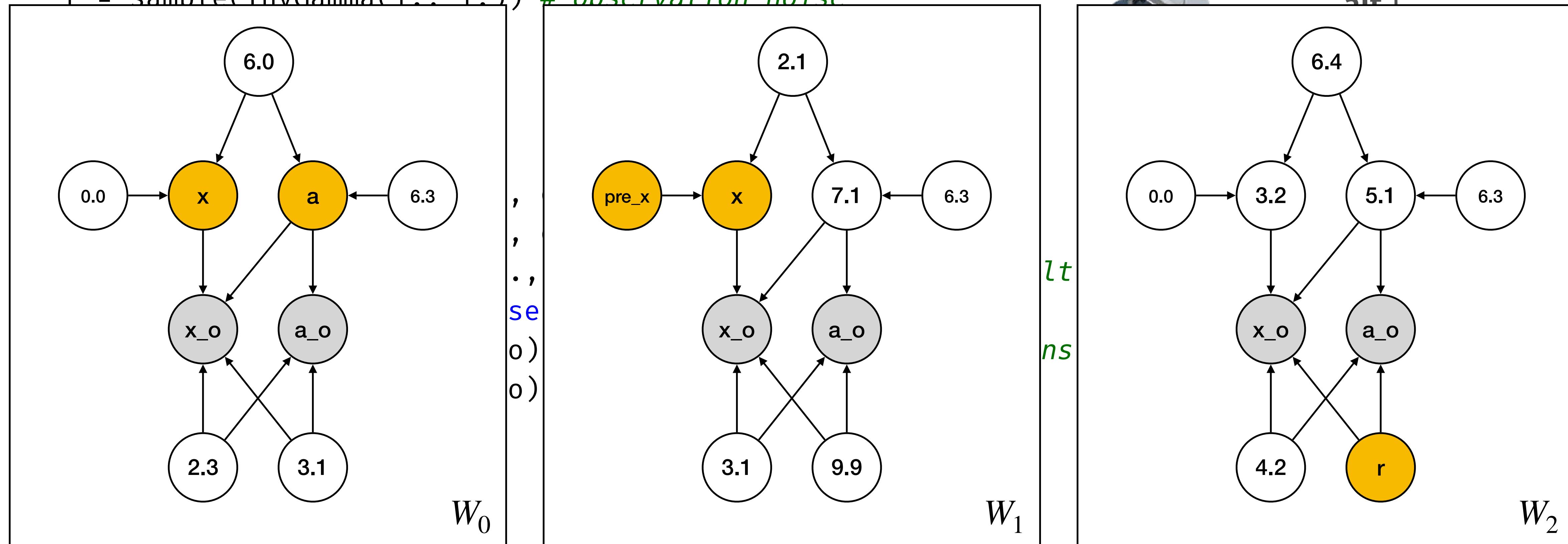
Semi-symbolic inference

```
def tracker(data: list[X * A]) → list[X * A] * float * float:
    q = sample(InvGamma(1., 1.)) # move noise
    r = sample(TinvGamma(1.. 1.)) # observation noise
```



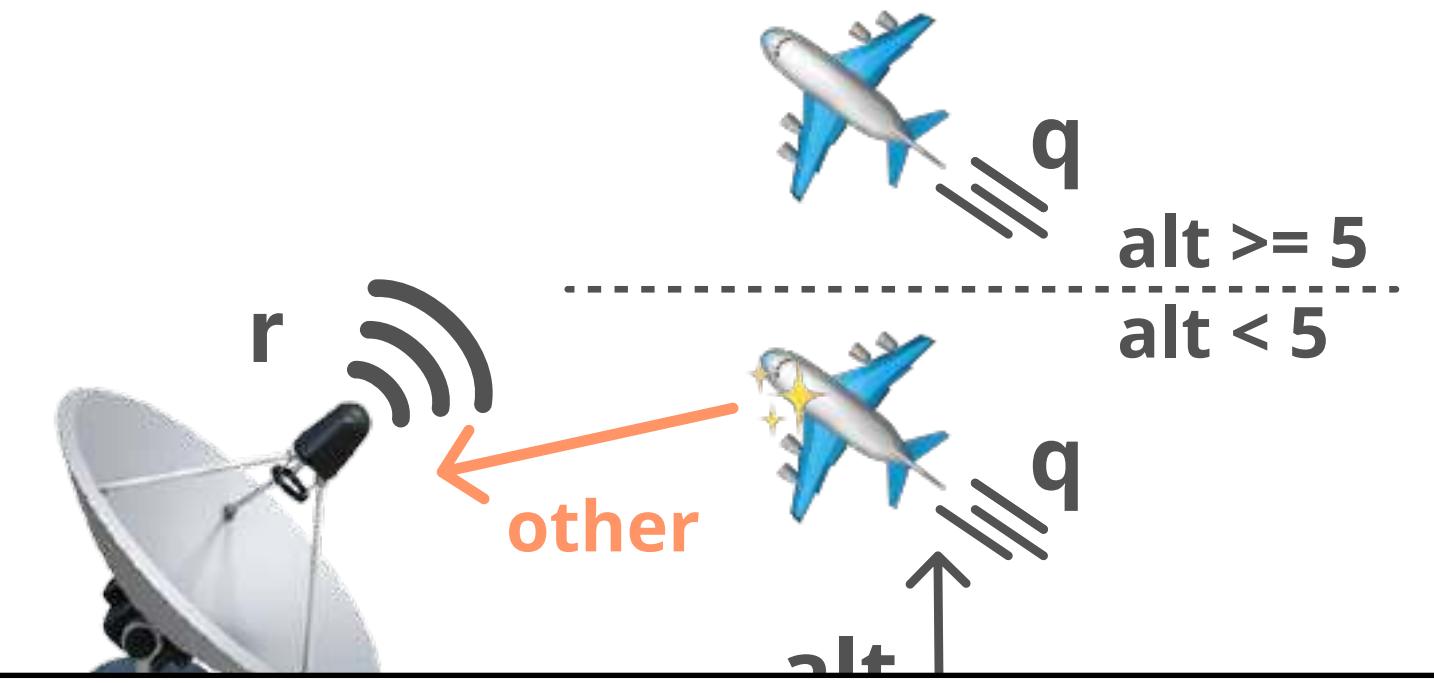
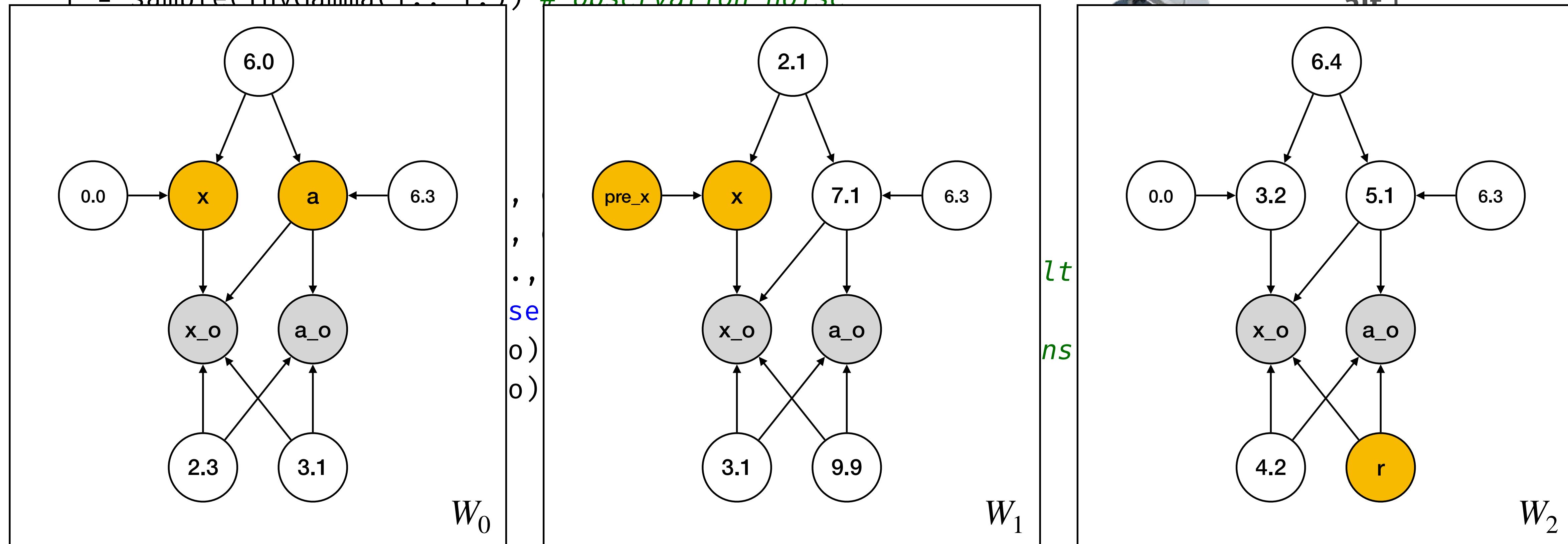
Semi-symbolic inference

```
def tracker(data: list[X * A]) → list[X * A] * float * float:
    q = sample(InvGamma(1., 1.)) # move noise
    r = sample(TinvGamma(1.. 1.)) # observation noise
```



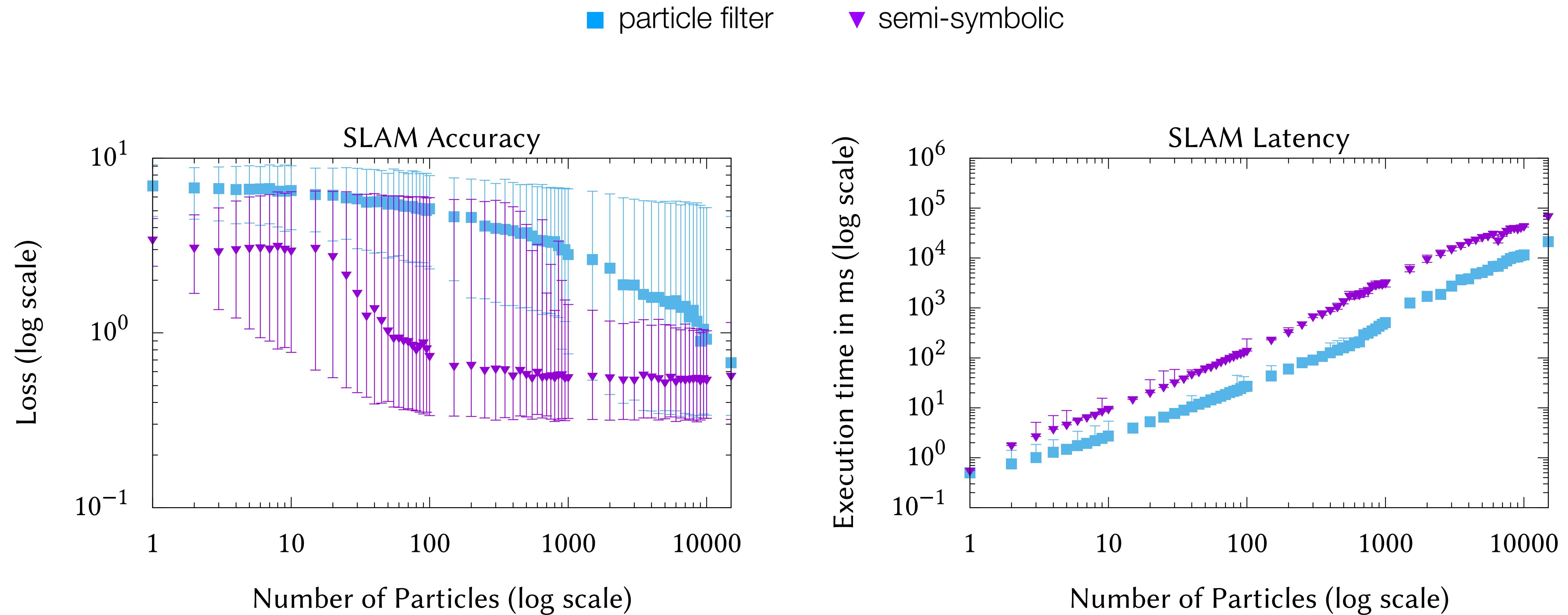
Semi-symbolic inference

```
def tracker(data: list[X * A]) → list[X * A] * float * float:
    q = sample(InvGamma(1., 1.)) # move noise
    r = sample(TinvGamma(1.. 1.)) # observation noise
```



Resample: particle filtering

Speed vs. accuracy



Benchmarks

Time to reach a target accuracy

Baseline:
SDS with 1,000 particles

SSI computations cost more,
but can outperform PF

MODEL	PF		DS		SSI	
	# PART.	TIME (MS)	# PART.	TIME (MS)	# PART.	TIME (MS)
Beta-Bernoulli	200	23.05 (22.54-23.90)	✓ 1	0.28 (0.27-0.28)	✓ 1	0.65 (0.65-0.66)
Gaussian-Gaussian	3000	877.44 (689.04-890.68)	150	62.99 (61.69-65.86)	150	182.57 (181.40-183.54)
Kalman-1D	15	3.27 (3.26-3.28)	✓ 1	0.33 (0.33-0.34)	✓ 1	1.15 (1.14-1.15)
Outlier	700	222.27 (220.76-223.84)	65	43.81 (43.45-46.48)	65	125.88 (125.27-128.08)
Robot	85	771.32 (767.98-775.51)	✓ 1	91.44 (90.94-92.07)	✓ 1	96.40 (96.21-97.53)
SLAM			✗	2812.55 (2755.99-2853.89)	800	5649.30 (5619.59-5675.81)
MTT			✗	2889.11 (2615.76-3244.30)	60	4457.79 (4068.35-4996.20)
Tree	150	35.55 (35.41-35.68)	90	58.83 (58.55-59.74)	✓ 1	2.67 (2.66-2.70)
Wheels	550	246.48 (245.06-248.75)	550	699.12 (672.25-713.64)	✓ 1	8.04 (8.00-8.10)
Delayed GPS	150	1221.00 (1218.76-1230.67)	9	304.73 (303.17-306.31)	✓ 1	108.55 (108.02-109.07)

Inference plan

Probabilistic Programming Languages

Probabilistic inference by program transformation in Hakaru (system description)*

Praveen Narayanan¹, Jacques Carette², Wren Romano¹, Chung-chieh Shan¹,
and Robert Zinkov¹

¹ Indiana University {pravnar,wrengr,ccshan,zinkov}@indiana.edu

² McMaster University carette@mcmaster.ca

Semi-symbolic Inference for Efficient Streaming Probabilistic Programming

ERIC ATKINSON, MIT, USA

CHARLES YUAN, MIT, USA

GUILLAUME BAUDART, ENS – PSL University – CNRS – Inria, France

LOUIS MANDEL, IBM Research, USA

MICHAEL CARBIN, MIT, USA

Delayed Sampling and Automatic Rao–Blackwellization of Probabilistic Programs

Lawrence M. Murray
Uppsala University

Daniel Lundén
KTH Royal Institute of Technology

Jan Kudlicka
Uppsala University

David Broman
KTH Royal Institute of Technology

Thomas B. Schön
Uppsala University

Automatic Rao-Blackwellization for Sequential Monte Carlo with Belief Propagation

Waïss Azizian¹ Guillaume Baudart² Marc Lelarge³

Automatically Marginalized MCMC in Probabilistic Programming

Jinlin Lai¹ Javier Burroni¹ Hui Guan¹ Daniel Sheldon¹

Inference plan

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:sample = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:symbolic = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q))
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r
```

Symbolic x plan

Inference plan

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:sample = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:symbolic = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q))
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r
```

Symbolic x plan

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:symbolic = sample(InvGamma(1., 1.))

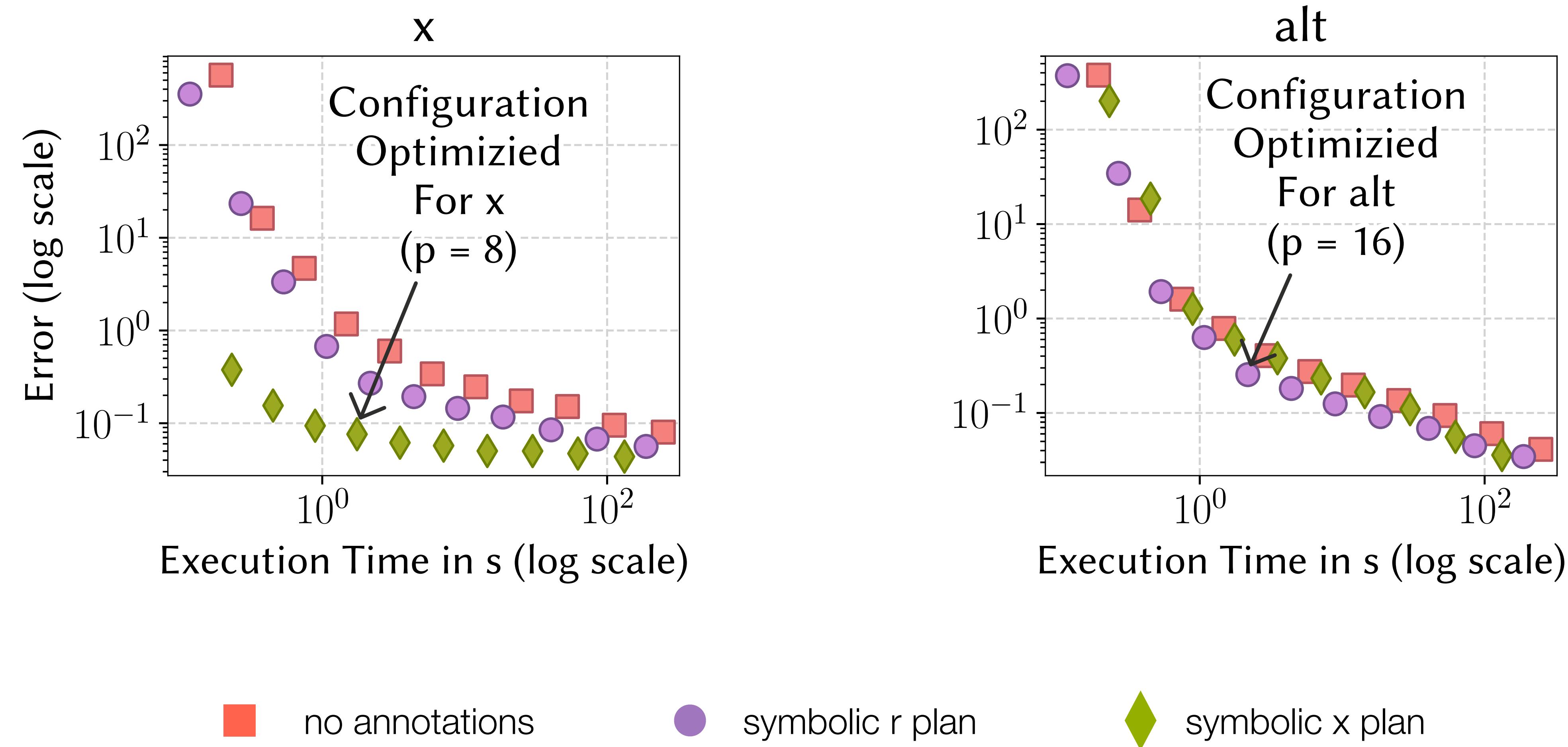
    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:sample = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q))
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r
```

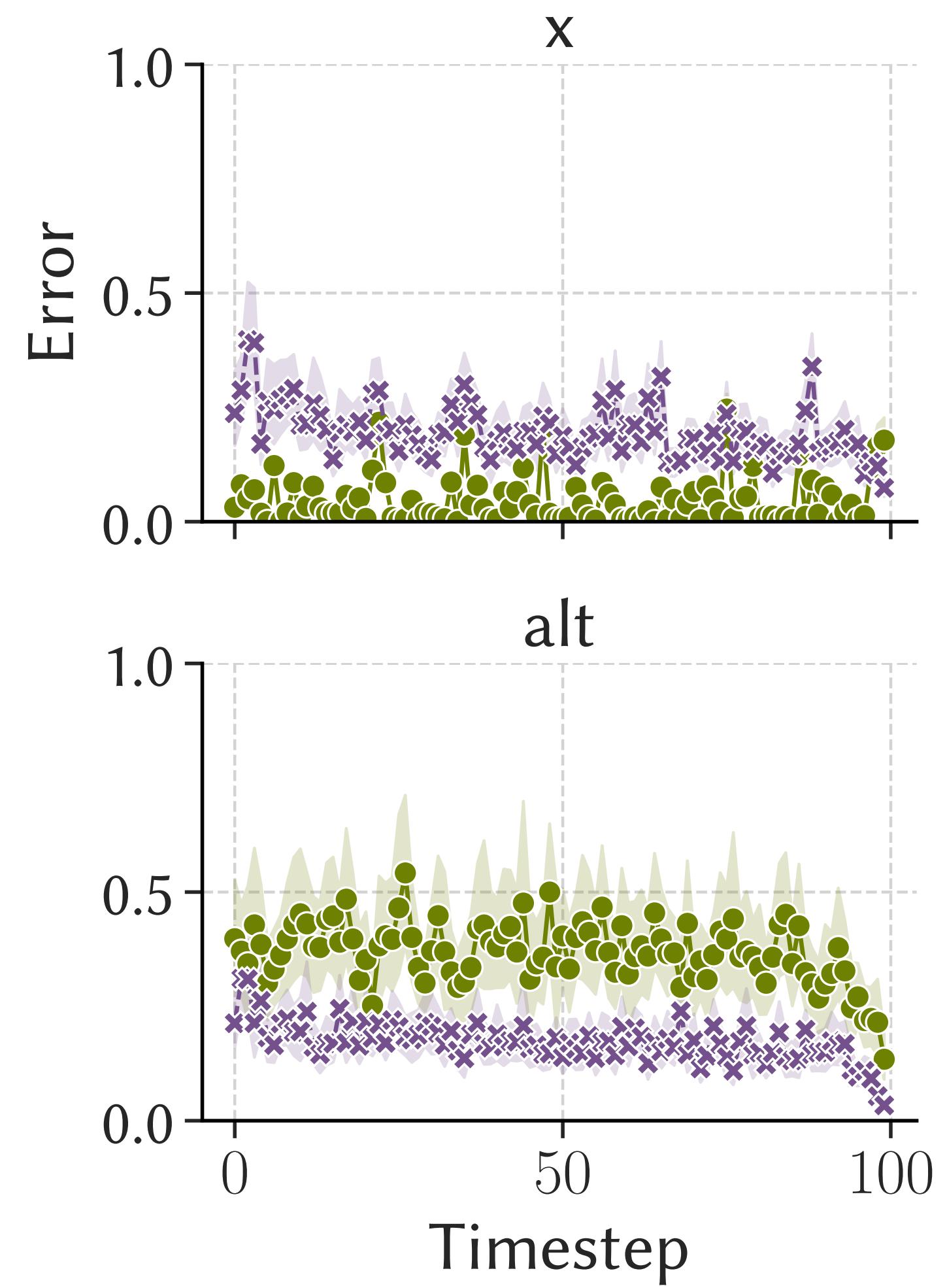
Symbolic r plan

Trade-off accuracy / speed

Max runtime: 3-seconds

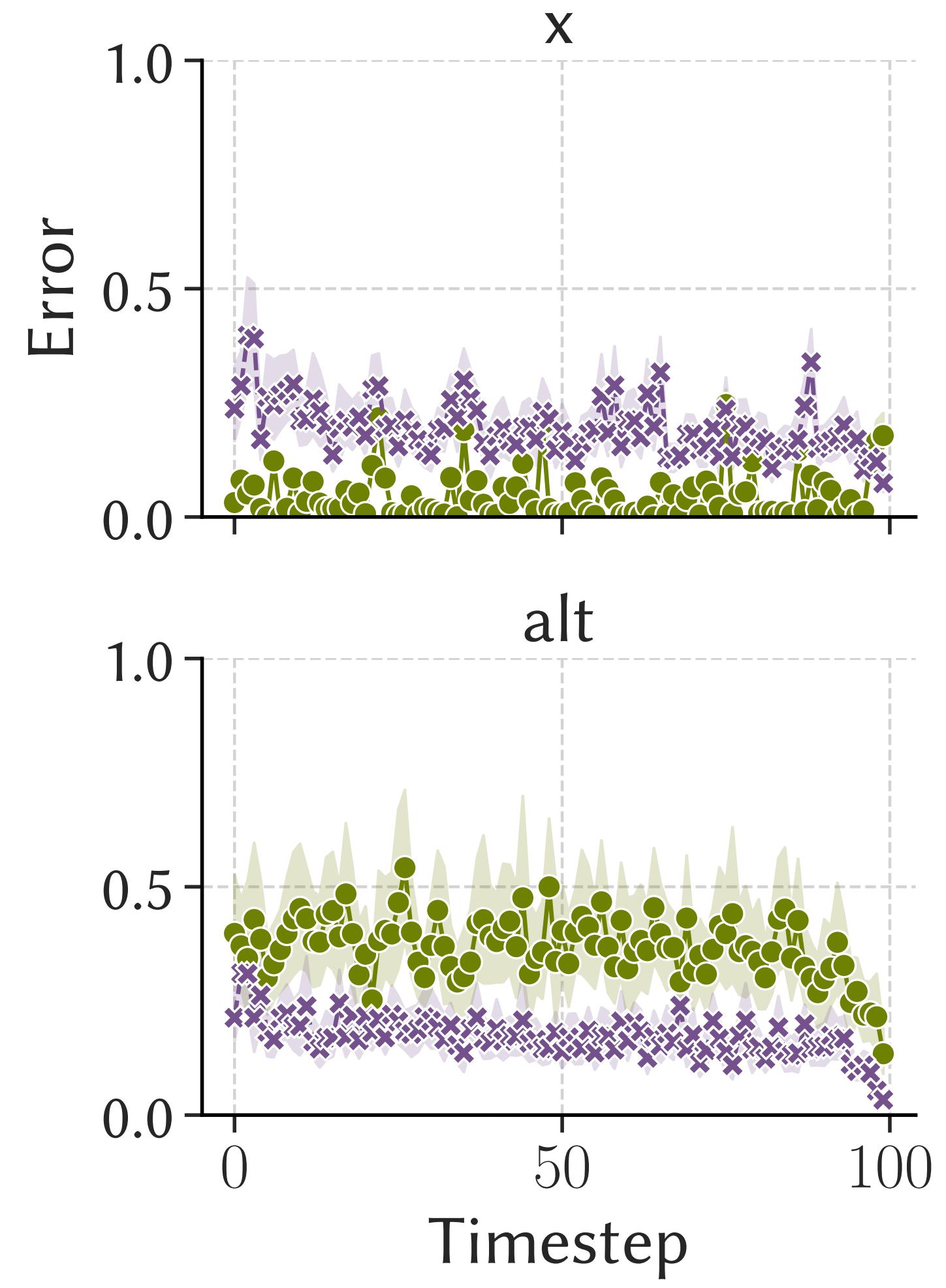


Runtime



Runtime

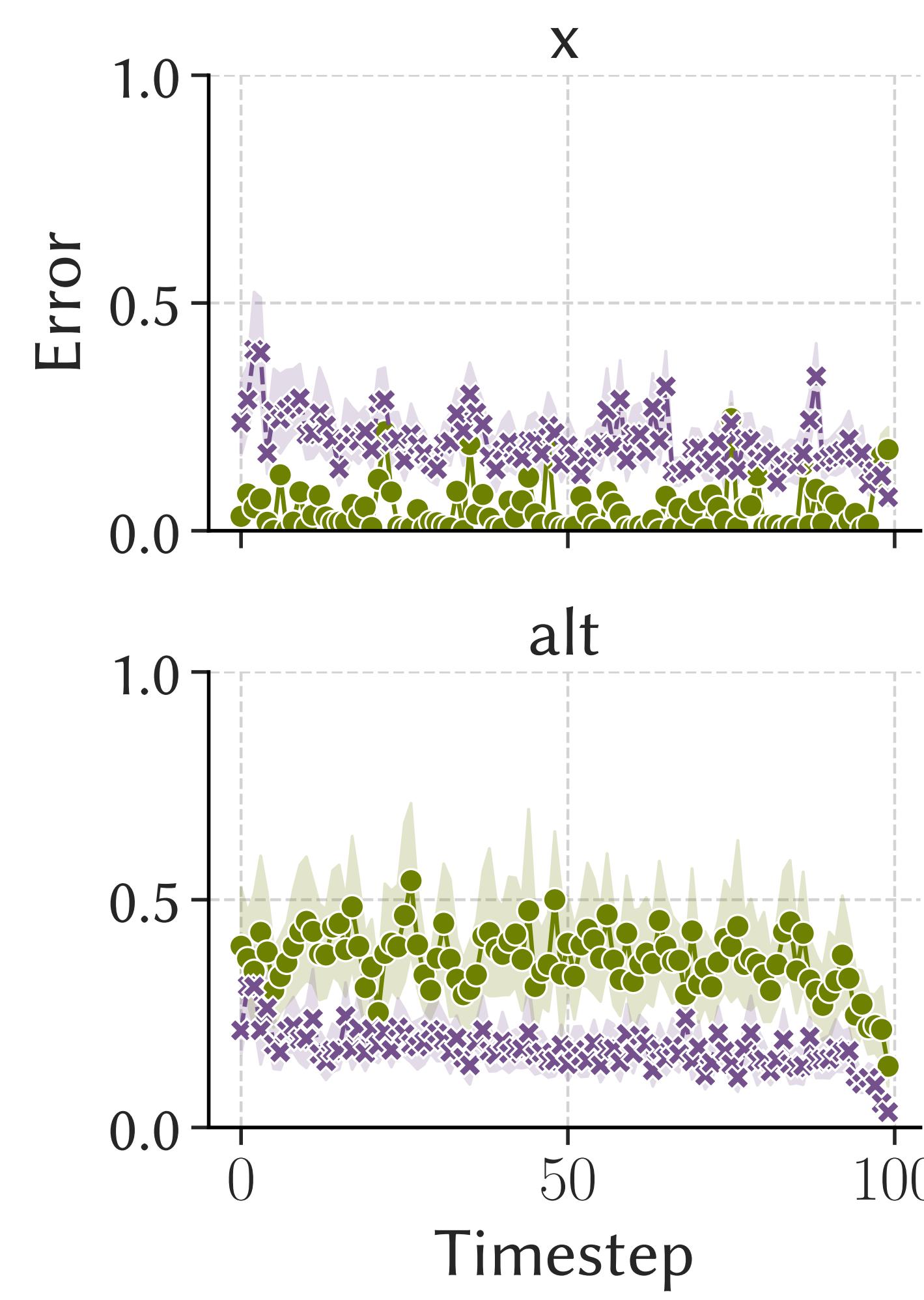
—●— Symbolic x Plan ($p = 8$) -·-·- Symbolic r Plan ($p = 16$)



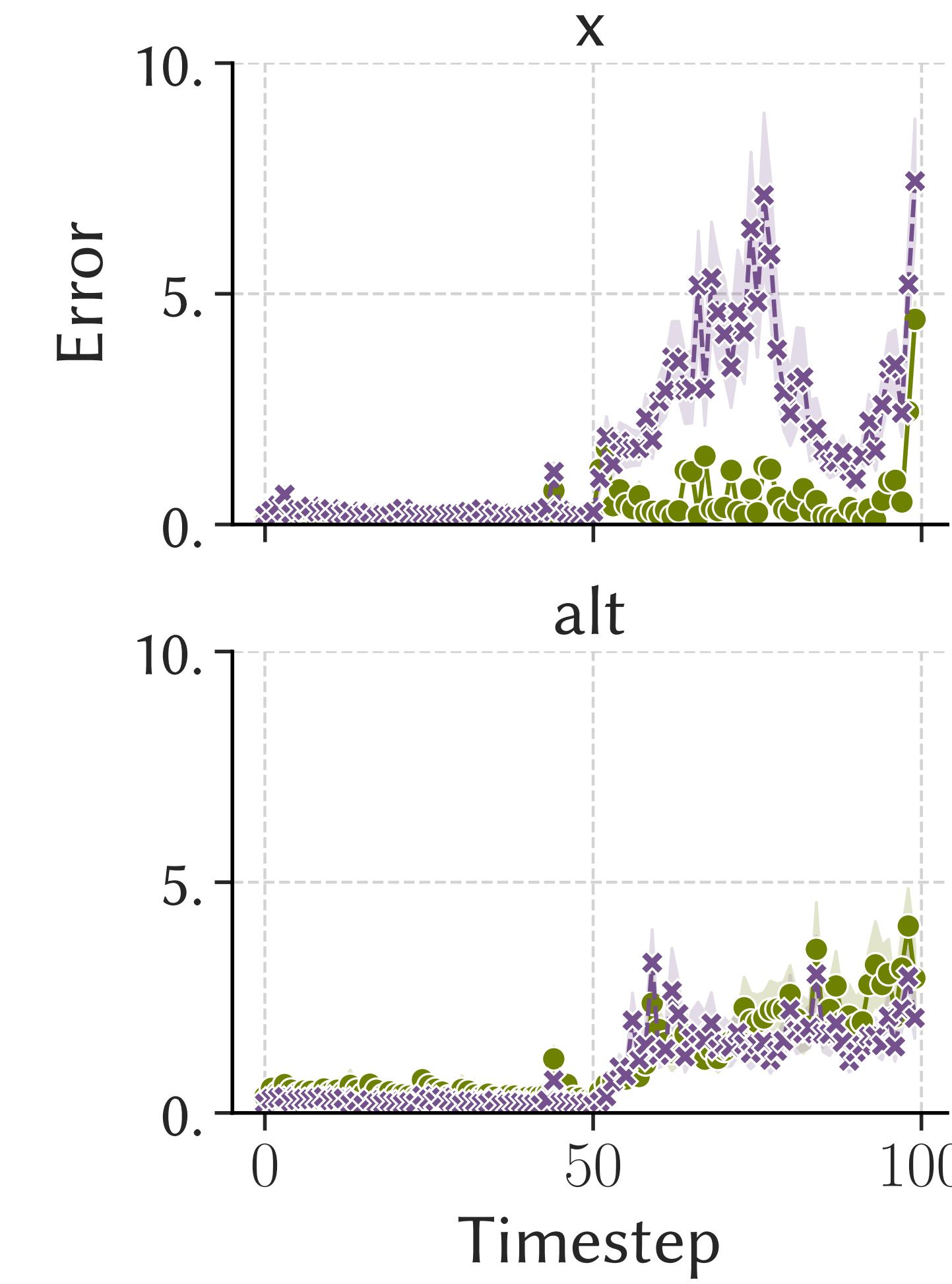
alt > 10

Runtime

—●— Symbolic x Plan ($p = 8$) -·-·- Symbolic r Plan ($p = 16$)

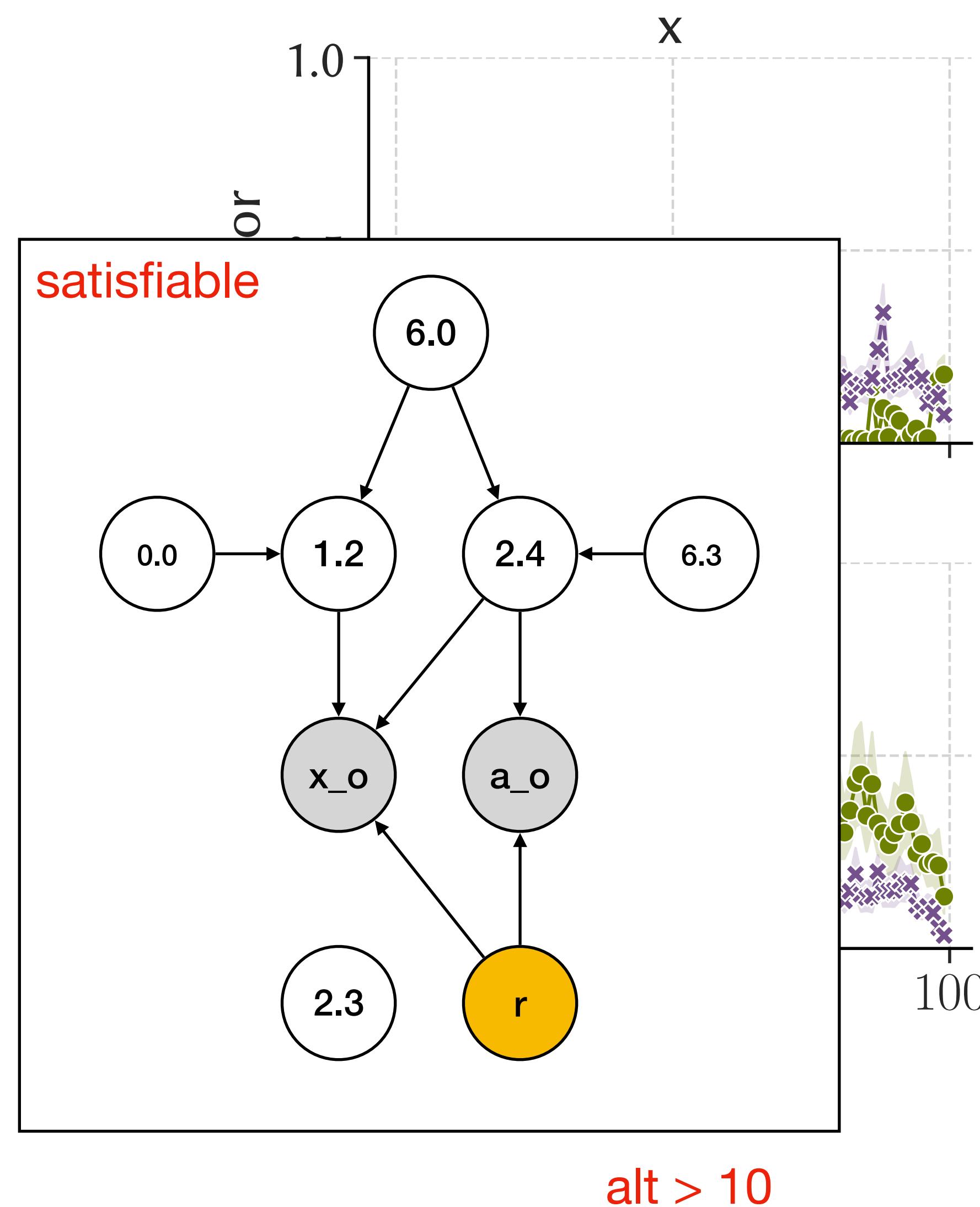


alt > 10

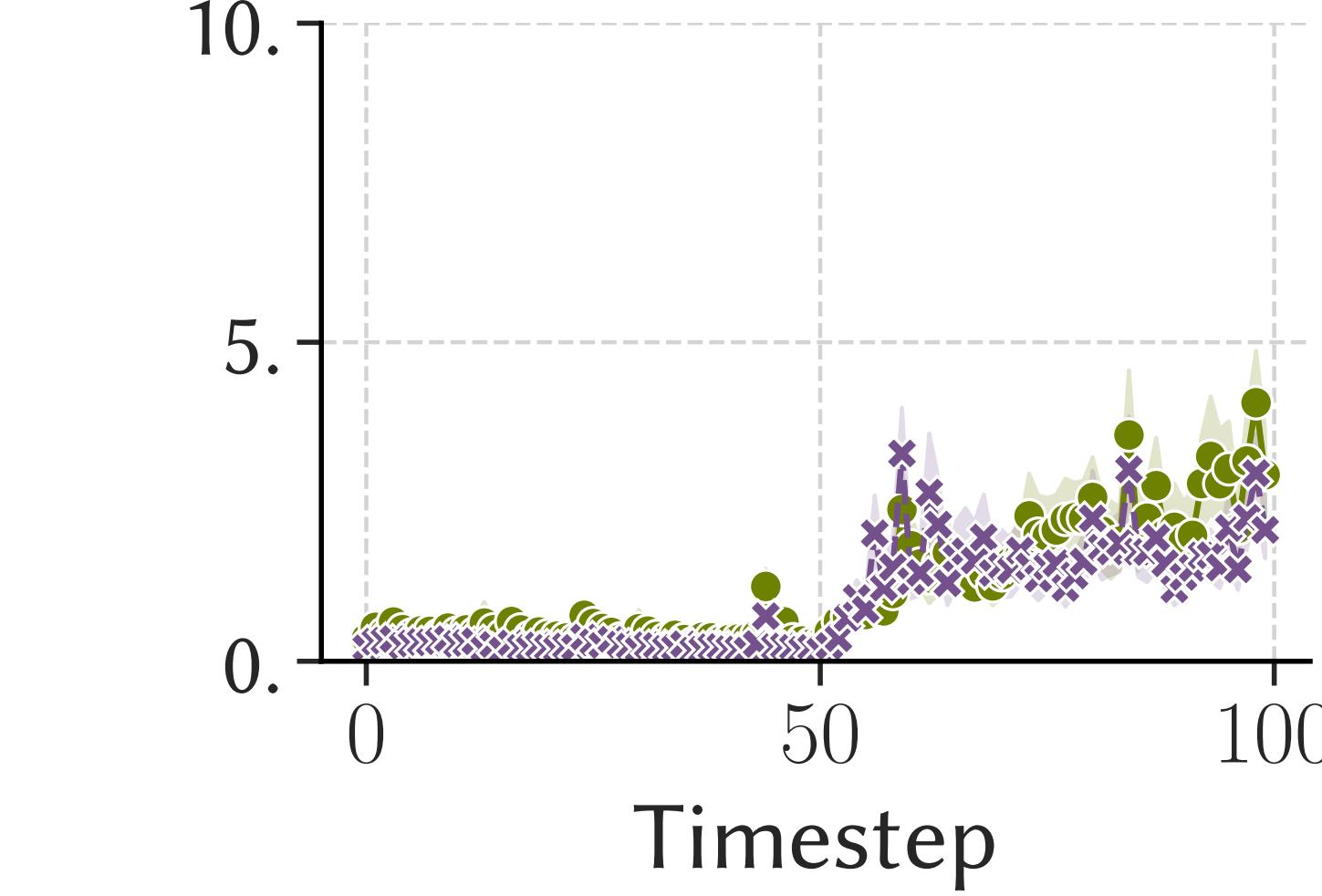
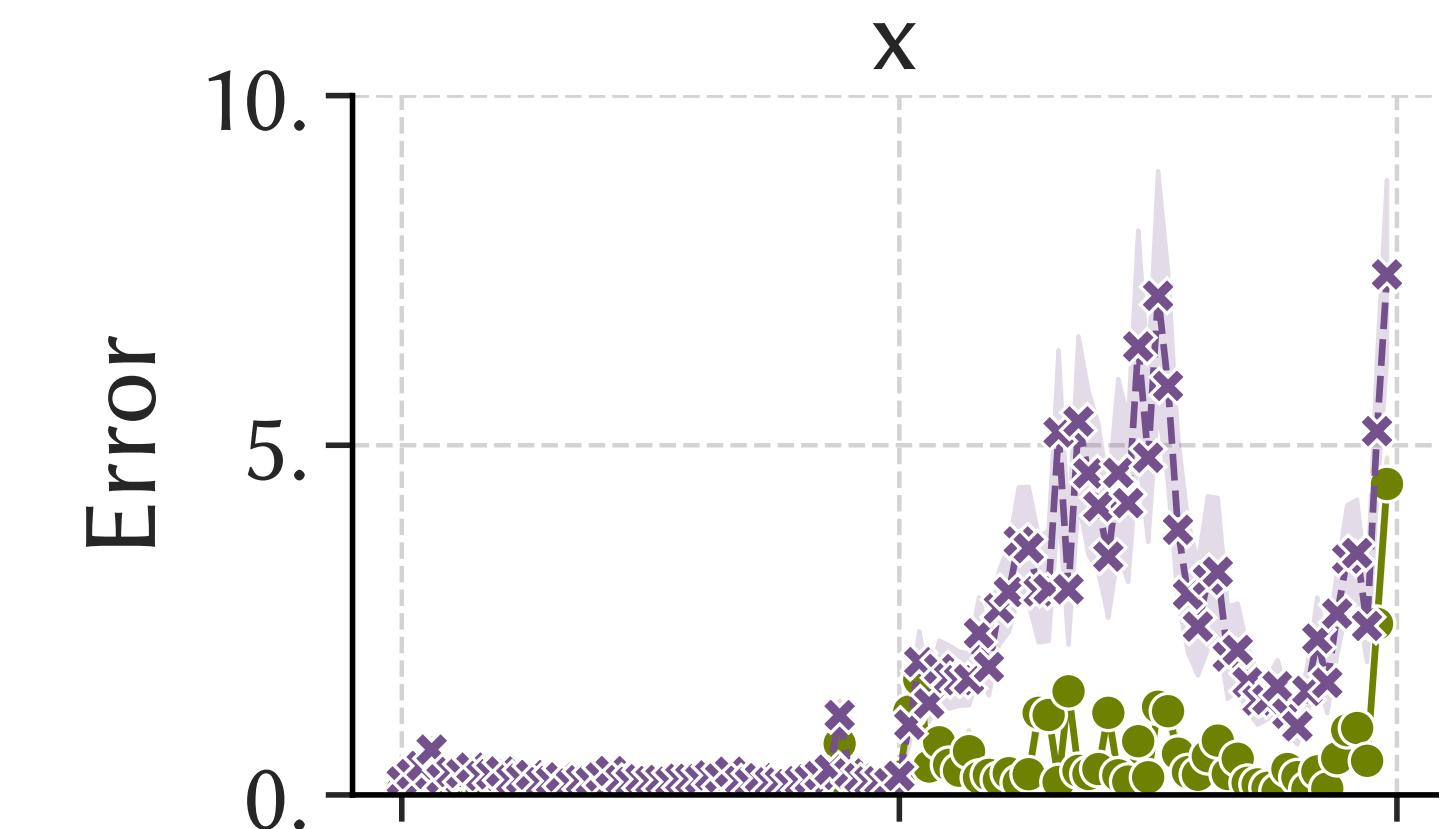


alt < 10

Runtime



—●— Symbolic x Plan ($p = 8$) -·-·-·- Symbolic r Plan ($p = 16$)

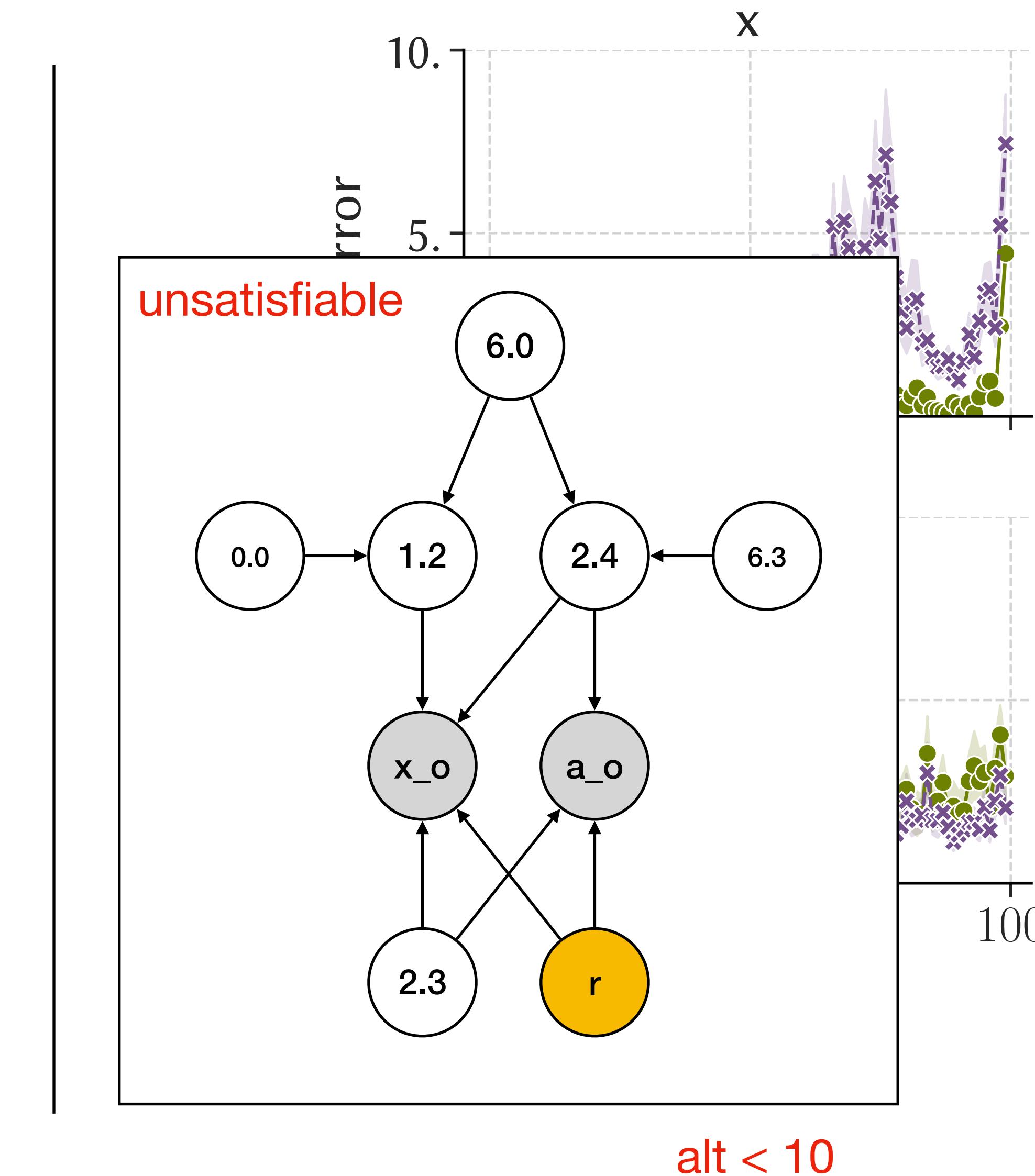
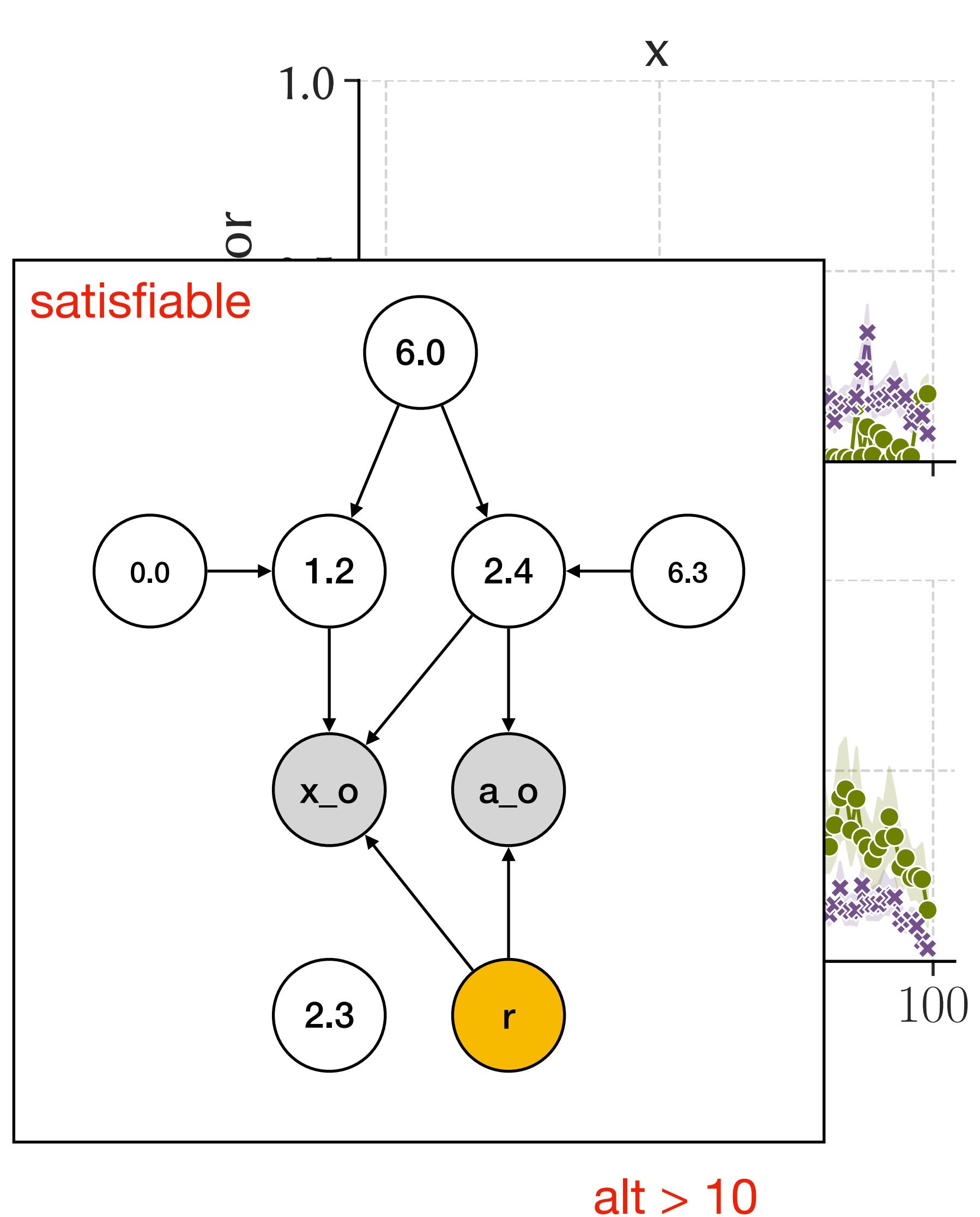


$alt < 10$

Runtime

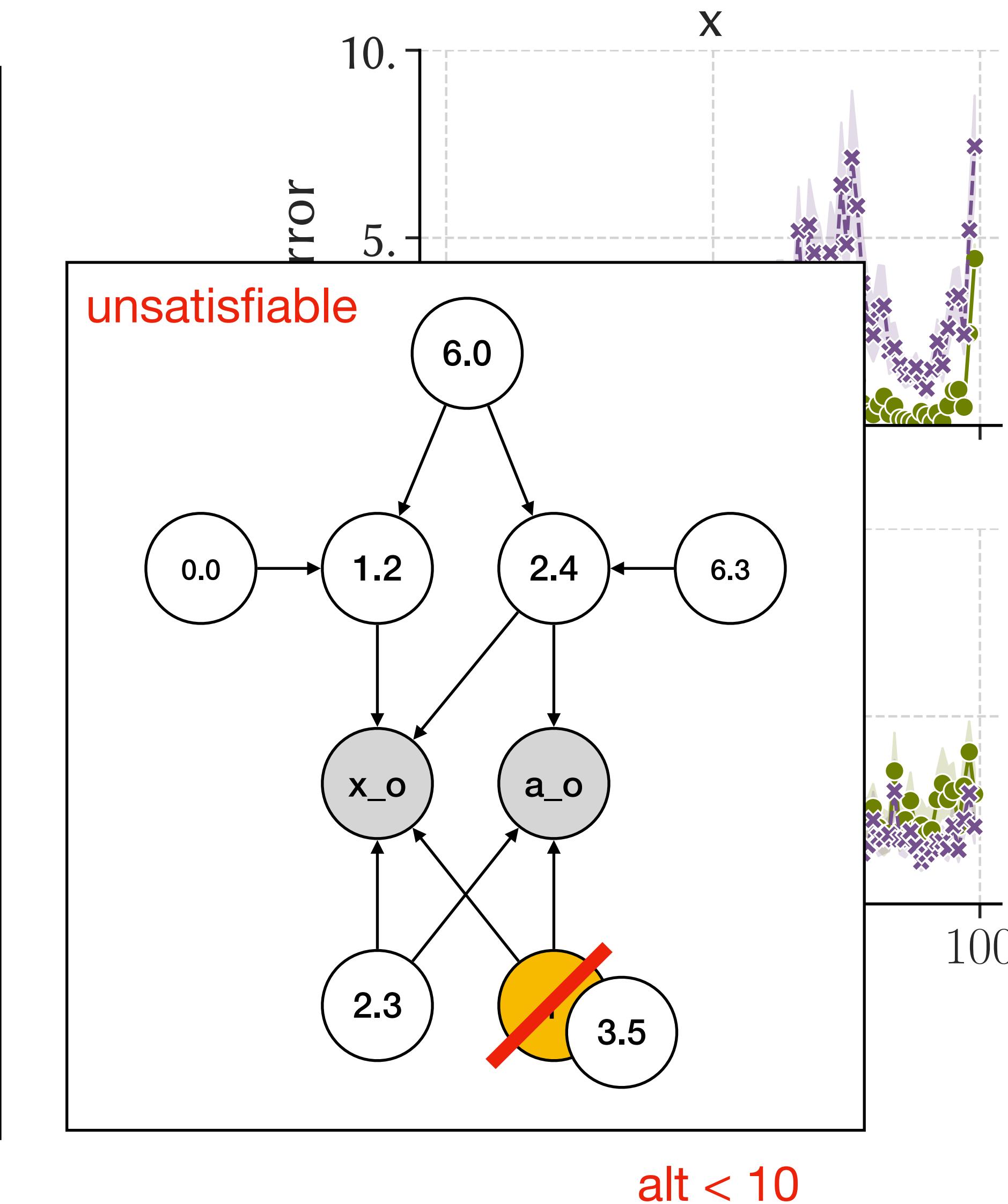
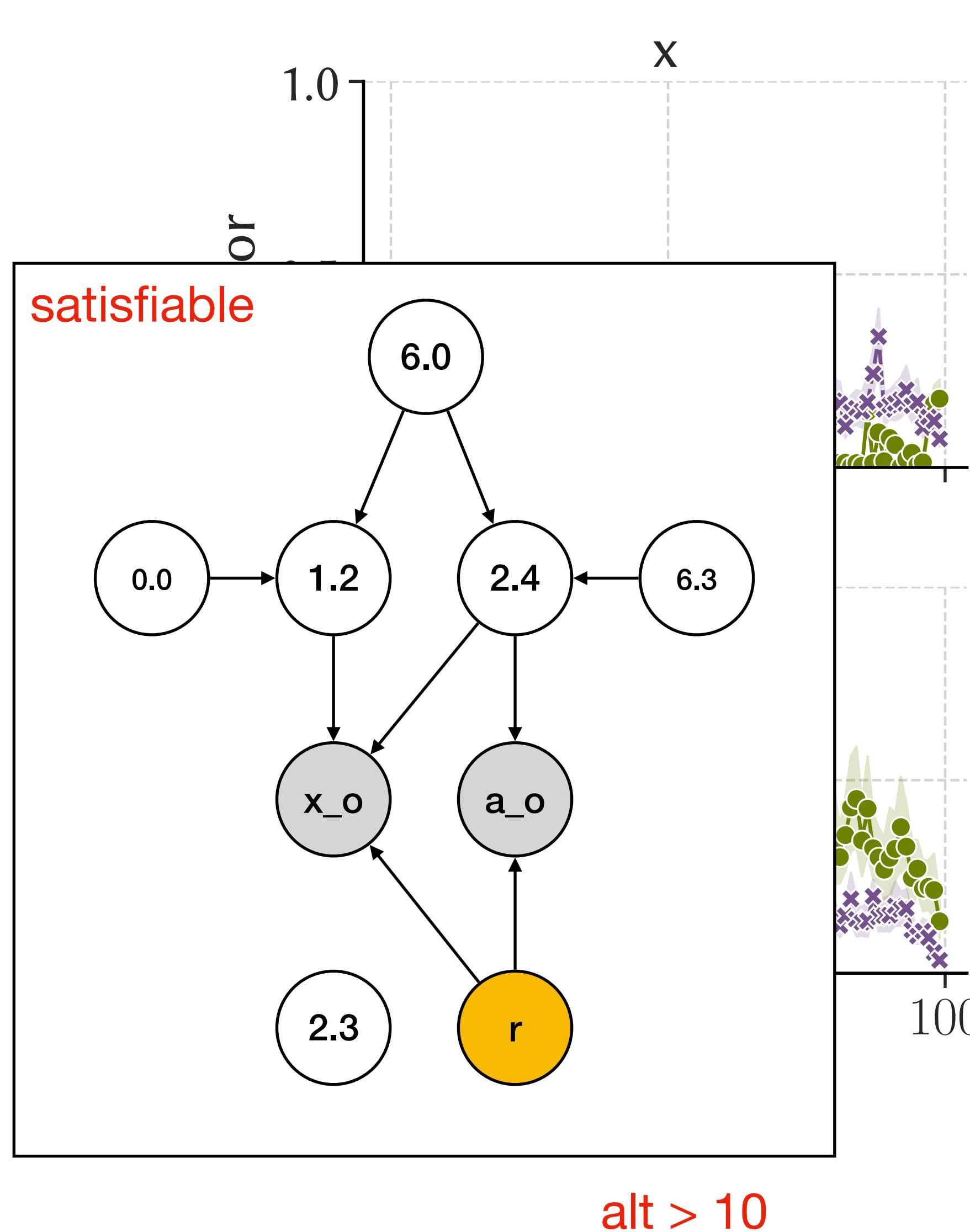
—●— Symbolic x Plan ($p = 8$) -•- Symbolic

-----*----- Symbolic r Plan (p = 16)



Runtime

—●— Symbolic x Plan (p = 8) -·-·- Symbolic r Plan (p = 16)



Plan satisfiability analysis

Statically determine if a plan is satisfiable

- Check all possible executions of a program
- Formalized using abstract interpretation

Abstract symbolic expressions

$$\begin{aligned} D ::= & \ N(E, E) \mid \text{Bernoulli}(E) \mid \text{InvGamma}(E, E) \mid \Delta(E) \mid \Delta_S(E) \\ & \mid \text{UnkD}(S) \mid \text{TopD} \end{aligned}$$
$$\begin{aligned} E ::= & \ r \mid c \mid x \mid E + E \mid E - E \mid E * E \mid E / E \\ & \mid \text{UnkC} \mid \text{UnkE}(S) \mid \text{TopE} \end{aligned}$$

Soundness: If the analysis says that the plan is satisfiable, all possible executions are valid

Plan satisfiability analysis

Statically determine if a plan is satisfiable

- Check all possible executions of a program
- Formalized using abstract interpretation

Abstract symbolic expressions

$$\begin{aligned} D ::= & \ N(E, E) \mid \text{Bernoulli}(E) \mid \text{InvGamma}(E, E) \mid \Delta(E) \mid \Delta_S(E) \\ & \mid \text{UnkD}(S) \mid \text{TopD} \end{aligned}$$
$$\begin{aligned} E ::= & \ r \mid c \mid x \mid E + E \mid E - E \mid E * E \mid E / E \\ & \mid \text{UnkC} \mid \text{UnkE}(S) \mid \text{TopE} \end{aligned}$$

Soundness: If the analysis says that the plan is satisfiable, all possible executions are valid

Possible false alarms

Plan satisfiability analysis

```
def tracker(data: list[X * A]) → list[X * A] * float * float:
    q:sample = sample(InvGamma(1., 1.))
    r:symbolic = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:sample = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q))
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r
```

Plan satisfiability analysis

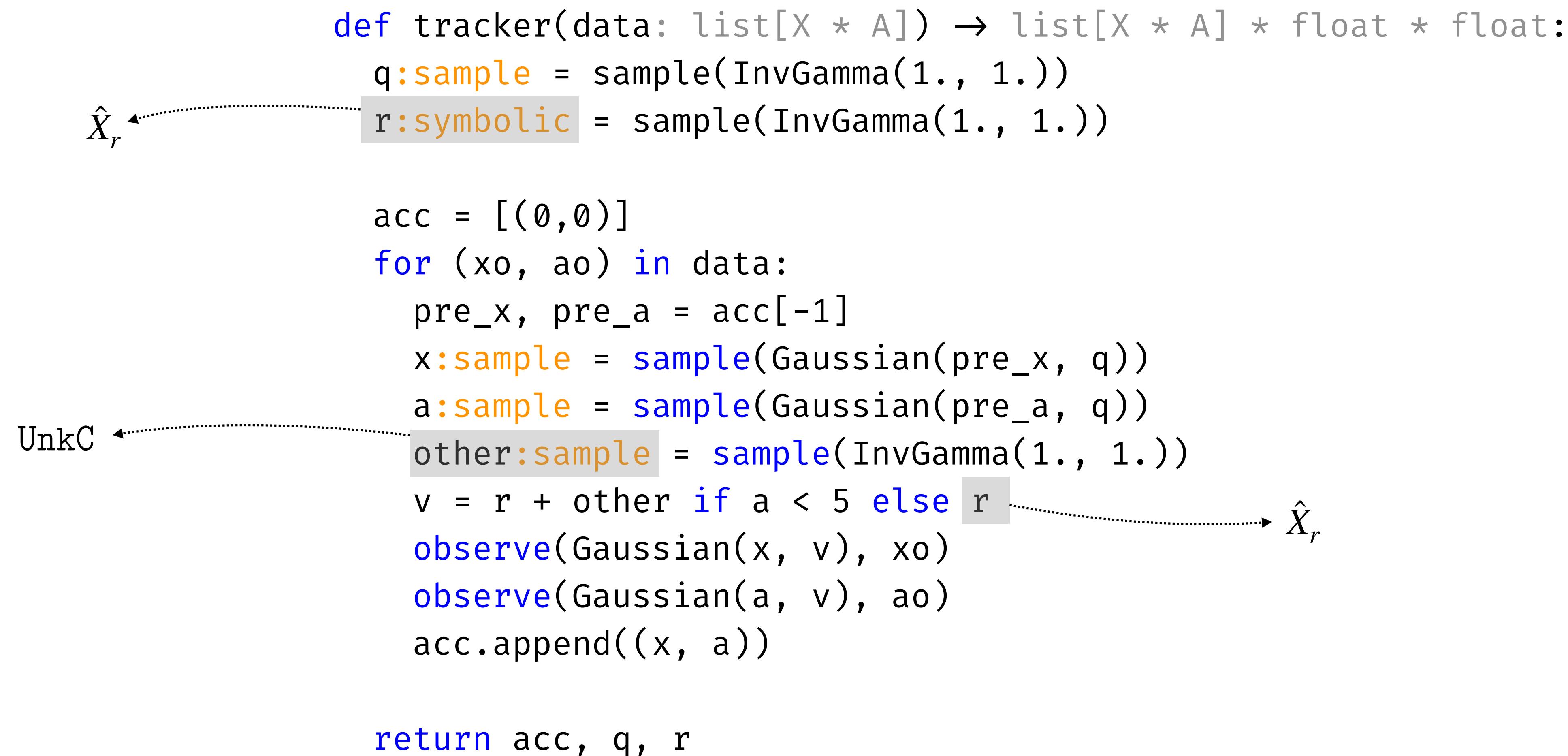
```
def tracker(data: list[X * A]) → list[X * A] * float * float:  
    q:sample = sample(InvGamma(1., 1.))  
    r:symbolic = sample(InvGamma(1., 1.))  
  
    acc = [(0,0)]  
    for (xo, ao) in data:  
        pre_x, pre_a = acc[-1]  
        x:sample = sample(Gaussian(pre_x, q))  
        a:sample = sample(Gaussian(pre_a, q))  
        other:sample = sample(InvGamma(1., 1.))  
        v = r + other if a < 5 else r  
        observe(Gaussian(x, v), xo)  
        observe(Gaussian(a, v), ao)  
        acc.append((x, a))  
  
    return acc, q, r
```

Plan satisfiability analysis

```
def tracker(data: list[X * A]) → list[X * A] * float * float:  
    q:sample = sample(InvGamma(1., 1.))  
    r:symbolic = sample(InvGamma(1., 1.))  
  
    acc = [(0,0)]  
    for (xo, ao) in data:  
        pre_x, pre_a = acc[-1]  
        x:sample = sample(Gaussian(pre_x, q))  
        a:sample = sample(Gaussian(pre_a, q))  
        other:sample = sample(InvGamma(1., 1.))  
        v = r + other if a < 5 else r  
        observe(Gaussian(x, v), xo)  
        observe(Gaussian(a, v), ao)  
        acc.append((x, a))  
  
    return acc, q, r
```

Plan satisfiability analysis

```
def tracker(data: list[X * A]) → list[X * A] * float * float:  
    q:sample = sample(InvGamma(1., 1.))  
    r:symbolic = sample(InvGamma(1., 1.))  
  
    acc = [(0,0)]  
    for (xo, ao) in data:  
        pre_x, pre_a = acc[-1]  
        x:sample = sample(Gaussian(pre_x, q))  
        a:sample = sample(Gaussian(pre_a, q))  
        other:sample = sample(InvGamma(1., 1.))  
        v = r + other if a < 5 else r  
        observe(Gaussian(x, v), xo)  
        observe(Gaussian(a, v), ao)  
        acc.append((x, a))  
  
    return acc, q, r
```



The diagram illustrates the flow of information in the `tracker` function. It shows variables being assigned values and how these values depend on each other. Specifically, the variable `v` is assigned the value of `r` plus `other` if `a` is less than 5, or `r` otherwise. The variable `r` is also used in the assignment of `x` and `a`. Additionally, `r` is passed to the `X_r` calculation. The variable `UnkC` is shown pointing to `r`, which suggests it might be a placeholder or a variable being tracked.

Plan satisfiability analysis

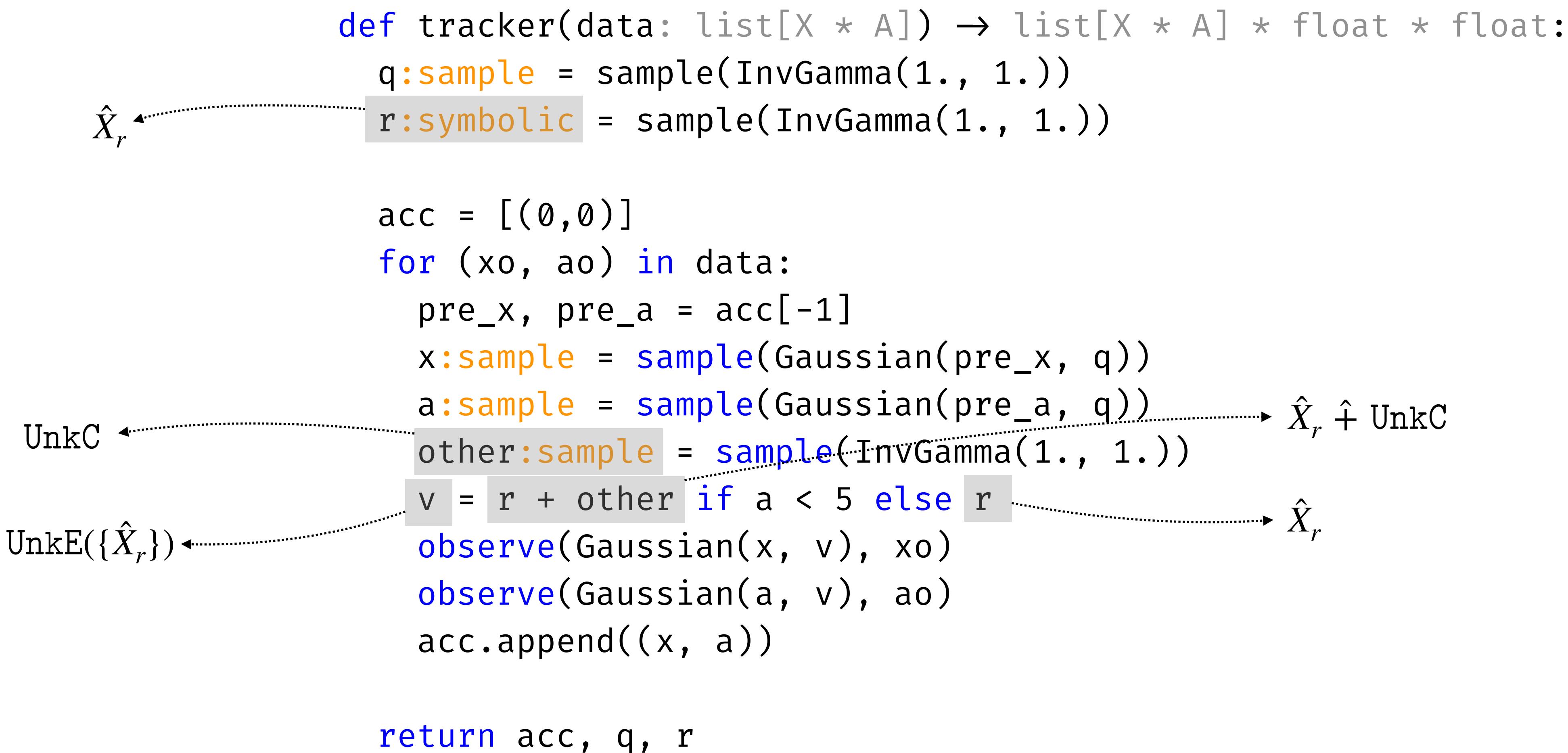
```
def tracker(data: list[X * A]) → list[X * A] * float * float:  
    q:sample = sample(InvGamma(1., 1.))  
    r:symbolic = sample(InvGamma(1., 1.))  
  
    acc = [(0,0)]  
    for (xo, ao) in data:  
        pre_x, pre_a = acc[-1]  
        x:sample = sample(Gaussian(pre_x, q))  
        a:sample = sample(Gaussian(pre_a, q))  
        other:sample = sample(InvGamma(1., 1.))  
        v = r + other if a < 5 else r  
        observe(Gaussian(x, v), xo)  
        observe(Gaussian(a, v), ao)  
        acc.append((x, a))  
  
    return acc, q, r
```

The diagram illustrates variable dependencies in the `tracker` function. It uses dotted arrows to show how variables are updated or derived from previous values.

- A dotted arrow points from \hat{X}_r to the assignment of `r:symbolic`.
- A dotted arrow points from UnkC to the assignment of `other:sample`.
- A dotted arrow points from $\hat{X}_r + \text{UnkC}$ to the assignment of `v`.
- A dotted arrow points from \hat{X}_r to the assignment of `v`.

Plan satisfiability analysis

```
def tracker(data: list[X * A]) → list[X * A] * float * float:  
    q:sample = sample(InvGamma(1., 1.))  
    r:symbolic = sample(InvGamma(1., 1.))  
  
    acc = [(0,0)]  
    for (xo, ao) in data:  
        pre_x, pre_a = acc[-1]  
        x:sample = sample(Gaussian(pre_x, q))  
        a:sample = sample(Gaussian(pre_a, q))  
        other:sample = sample(InvGamma(1., 1.))  
        v = r + other if a < 5 else r  
        observe(Gaussian(x, v), xo)  
        observe(Gaussian(a, v), ao)  
        acc.append((x, a))  
  
    return acc, q, r
```



The diagram illustrates the flow of variables in the `tracker` function. It shows the initial inputs `UnkC` and `UnkE({X_r})` entering the function. Inside the loop, symbolic values `r` and `other` are sampled from `InvGamma` distributions. These values are then used to calculate `v` using an `if` condition. The variable `X_r` is updated by adding `r` and `other`. Finally, the function returns `acc, q, r`.

Plan satisfiability analysis

```
def tracker(data: list[X * A]) → list[X * A] * float * float:  
    q:sample = sample(InvGamma(1., 1.))  
    r:symbolic = sample(InvGamma(1., 1.))  
  
    acc = [(0,0)]  
    for (xo, ao) in data:  
        pre_x, pre_a = acc[-1]  
        x:sample = sample(Gaussian(pre_x, q))  
        a:sample = sample(Gaussian(pre_a, q))  
        other:sample = sample(InvGamma(1., 1.))  
        v = r + other if a < 5 else r  
        observe(Gaussian(x, v), xo)  
        observe(Gaussian(a, v), ao)  
        acc.append((x, a))  
  
    return acc, q, r
```

The diagram illustrates the flow of information through the code. It shows various variable definitions and their relationships. Labels indicate specific states or constraints:

- \hat{X}_r : A label pointing to the variable `r`.
- UnkC : A label pointing to the variable `other`.
- $\text{UnkE}(\{\hat{X}_r\})$: A label pointing to the expression `v = r + other if a < 5 else r`.
- $\hat{X}_r \dagger \text{UnkC}$: A label pointing to the expression `x:sample = sample(Gaussian(pre_x, q))`.
- \hat{X}_r : A label pointing to the expression `a:sample = sample(Gaussian(pre_a, q))`.
- $\mathcal{N}(\text{UnkC}, \text{UnkE}(\{\hat{X}_r\}))$: A label pointing to the final return statement.

Plan satisfiability analysis

```
def tracker(data: list[X * A]) → list[X * A] * float * float:  
    q:sample = sample(InvGamma(1., 1.))  
    r:symbolic = sample(InvGamma(1., 1.))  
  
    acc = [(0,0)]  
    for (xo, ao) in data:  
        pre_x, pre_a = acc[-1]  
        x:sample = sample(Gaussian(pre_x, q))  
        a:sample = sample(Gaussian(pre_a, q))  
        other:sample = sample(InvGamma(1., 1.))  
        v = r + other if a < 5 else r  
        observe(Gaussian(x, v), xo)  
        observe(Gaussian(a, v), ao)  
        acc.append((x, a))  
  
    return acc, q, r
```

UnkC ← $\hat{X}_r \uparrow \text{UnkC}$

UnkE($\{\hat{X}_r\}$) ← \hat{X}_r

$\hat{X}_r \uparrow \text{UnkC}$

$\mathcal{N}(\text{UnkC}, \text{UnkE}(\{\hat{X}_r\}))$

unsatisfiable

Plan satisfiability analysis

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:sample = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:symbolic = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q))
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

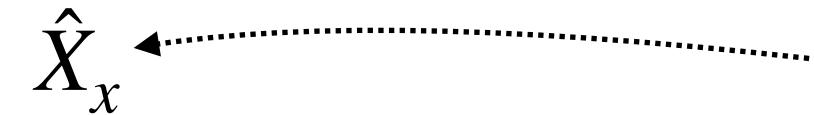
    return acc, q, r
```

Plan satisfiability analysis

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:sample = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:symbolic = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q))
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r
```



Plan satisfiability analysis

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:sample = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:symbolic = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q))
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r
```

$\hat{X}_x \leftarrow$

$\text{UnkC} \leftarrow$

Plan satisfiability analysis

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:sample = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:symbolic = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q))
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r
```

The diagram illustrates variable dependencies in the `tracker` function. It shows three main components: \hat{X}_x , UnkC , and $\text{UnkC} \hat{+} \text{UnkC}$. Dotted arrows indicate dependencies from UnkC to \hat{X}_x and UnkC , and from UnkC to $\text{UnkC} \hat{+} \text{UnkC}$. The code itself contains several annotations with gray boxes:

- A box around `x:symbolic = sample(Gaussian(pre_x, q))` corresponds to \hat{X}_x .
- A box around `a:sample = sample(Gaussian(pre_a, q))` corresponds to UnkC .
- A box around `other:sample = sample(InvGamma(1., 1.))` corresponds to $\text{UnkC} \hat{+} \text{UnkC}$.
- A box around `v = r + other if a < 5 else r` also corresponds to $\text{UnkC} \hat{+} \text{UnkC}$.

Plan satisfiability analysis

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:sample = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:symbolic = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q)) → UnkC ⋂ UnkC
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r → UnkC
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r
```

The diagram illustrates the dependencies between variables in the `tracker` function. It shows three types of annotations:

- \hat{X}_x : A label pointing to the assignment of `x:symbolic`.
- UnkC : A label pointing to the assignment of `a:sample`, the assignment of `v`, and the final return statement.
- $\text{UnkC} \hat{+} \text{UnkC}$: A label pointing to the assignment of `a:sample`.

Plan satisfiability analysis

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:sample = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:symbolic = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q)) → UnkC ⋂ UnkC
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r → UnkC
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r
```

The diagram illustrates the flow of variables and their types. It starts with a variable \hat{X}_x which points to the assignment of `x:symbolic`. This variable `x:symbolic` is then used in the assignment of `a:sample`. Both `x:symbolic` and `a:sample` point to a common label $\text{UnkC} \hat{+} \text{UnkC}$. The variable `other:sample` also points to this same label. The assignment of `v` (from `v = r + other if a < 5 else r`) points to a label UnkC . Finally, the variable `acc` points back to its initial assignment in the loop.

Plan satisfiability analysis

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:sample = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:symbolic = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q)) → UnkC ⋂ UnkC
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r → UnkC
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a)) → N(X̂_x, UnkC)

    return acc, q, r
```

Plan satisfiability analysis

```
def tracker(data):
    q:sample = sample(InvGamma(1., 1.))
    r:sample = sample(InvGamma(1., 1.))

    acc = [(0,0)]
    for (xo, ao) in data:
        pre_x, pre_a = acc[-1]
        x:symbolic = sample(Gaussian(pre_x, q))
        a:sample = sample(Gaussian(pre_a, q))
        other:sample = sample(InvGamma(1., 1.))
        v = r + other if a < 5 else r
        observe(Gaussian(x, v), xo)
        observe(Gaussian(a, v), ao)
        acc.append((x, a))

    return acc, q, r
```

The diagram illustrates the flow of variables and their types during the execution of the `tracker` function. It uses dotted arrows to show the assignment of values from one part of the code to another. Labels like \hat{X}_x , UnkC , and $\mathcal{N}(\hat{X}_x, \text{UnkC})$ are placed near specific parts of the code to indicate the state of variables at different points.

- Initial State:** \hat{X}_x is labeled above the first assignment of `x`.
- Assignment 1:** `x:symbolic = sample(Gaussian(pre_x, q))` is highlighted with a gray box. A dotted arrow points from this assignment to the label UnkC .
- Assignment 2:** `a:sample = sample(Gaussian(pre_a, q))` is highlighted with a gray box. A dotted arrow points from this assignment to the label $\text{UnkC} \dagger \text{UnkC}$.
- Assignment 3:** `other:sample = sample(InvGamma(1., 1.))` is highlighted with a gray box. A dotted arrow points from this assignment to the label UnkC .
- Assignment 4:** `v = r + other if a < 5 else r` is highlighted with a gray box. A dotted arrow points from this assignment to the label UnkC .
- Assignment 5:** `observe(Gaussian(x, v), xo)` is highlighted with a gray box. A dotted arrow points from this assignment to the label UnkC .
- Assignment 6:** `observe(Gaussian(a, v), ao)` is highlighted with a gray box. A dotted arrow points from this assignment to the label $\mathcal{N}(\hat{X}_x, \text{UnkC})$.
- Assignment 7:** `acc.append((x, a))` is highlighted with a gray box. A dotted arrow points from this assignment to the label $\mathcal{N}(\hat{X}_x, \text{UnkC})$.
- Final State:** The label **satisfiable** is placed to the right of the final `return` statement.

Takeaway

I - Probabilistic programming

- A program describes a distribution
- We can sample from distributions and condition on data
- Inference computes the distribution: Intractable problem in general

EJCIM'24

Introduction à la programmation probabiliste

Guillaume Baudart et Christine Tasson

La programmation probabiliste est un paradigme qui a connu un essor important ces dernières années. Les langages probabilistes manipulent l'incertitude de manière explicite. Ils reposent sur la méthode bayésienne qui permet d'apprendre la distribution des paramètres d'un modèle à partir d'observations statistiques. De nombreux langages de programmation reposant sur ce paradigme ont été développés : WebPPL, Venture, Anglican, Stan, Gen, Pyro, Turing.jl... Ces langages sont utilisés dans des domaines qui vont de la vision par ordinateur (génération d'images) et la robotique (planification), à la santé (épidémiologie) et les sciences sociales (sondages).

Ces notes présentent les concepts fondamentaux de la programmation probabiliste :
1 Introduction à la modélisation bayésienne

II - Approximate inference

- Importance sampling: weighted sampler returns pairs (value, score)
- Particle filtering: add checkpoints and resample particles

III - Semi-symbolic inference

- Each particle performs symbolic computations
- Fall back to a particle filter when symbolic computation fails

IV - Inference plan

- Control symbolic vs. approximate inference with annotations
- Static analysis checks if a plan is satisfiable

OOPSLA'22



Semi-symbolic Inference for Efficient Streaming Probabilistic Programming

ERIC ATKINSON, MIT, USA

CHARLES YUAN, MIT, USA

GUILLAUME BAUDART, ENS – PSL University – CNRS – Inria, France

LOUIS MANDEL, IBM Research, USA

MICHAEL CARBIN, MIT, USA

A streaming probabilistic program receives a stream of observations and produces a stream of distributions that are conditioned on these observations. Efficient inference is often possible in a streaming context using Rao-Blackwellized particle filters (RBPFs), which exactly solve inference problems when possible and fall back to a particle filter when symbolic computation fails.

POPL '25



Inference Plans for Hybrid Particle Filtering

ELLIE Y. CHENG, MIT CSAIL, USA

ERIC ATKINSON, Binghamton University, USA

GUILLAUME BAUDART, Université Paris Cité – CNRS – Inria – IRIF, France

LOUIS MANDEL, IBM Research, USA

MICHAEL CARBIN, MIT CSAIL, USA

Advanced probabilistic programming languages (PPLs) using *hybrid particle filtering* combine symbolic exact inference and Monte Carlo methods to improve inference performance. These systems use heuristics to partition random variables within the program into variables that are encoded symbolically and variables that are encoded with sampled values, and the heuristics are not necessarily aligned with the developer's performance evaluation metrics. In this work, we present *inference plans*, a programming interface that enables developers to control the partitioning of random variables during hybrid particle filtering. We further present