Reactive Probabilistic Programming

Guillaume Baudart

Inria

L. Mandel

IBM Research

E. Atkinson
B. Sherman
C. Yuan
M. Carbin

MIT

M. Pouzet

ENS

Uncertainty in Embedded Systems

Uncertainty in Embedded Systems

Synchronous languages

- High-level specification language
- Generate correct-by-construction embedded code
- Industrial tool: ANSYS Scade

Challenges

- Noisy environment, perceived through noisy sensors
- Interaction with other autonomous entities

Uncertainty in Embedded Systems

Synchronous languages

- High-level specification language
- Generate correct-by-construction embedded code
- Industrial tool: ANSYS Scade

Challenges

- Noisy environment, perceived through noisy sensors
- Interaction with other autonomous entities

Uncertainty in Embedded Systems

Synchronous languages

- High-level specification language
- Generate correct-by-construction embedded code
- Industrial tool: ANSYS Scade

Challenges

- Noisy environment, perceived through noisy sensors
- Interaction with other autonomous entities

Uncertainty in Embedded Systems

Synchronous languages

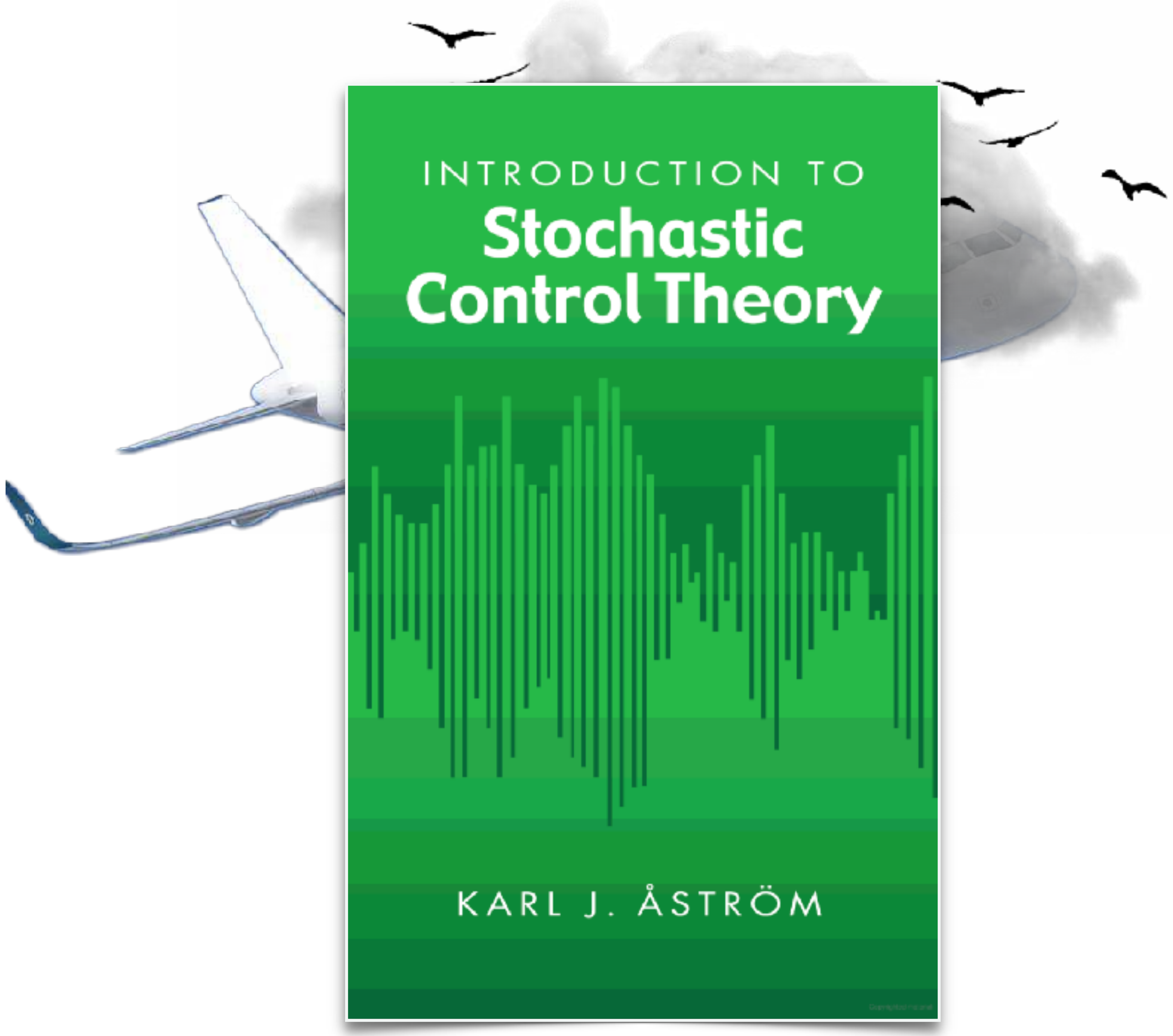
- High-level specification language
- Generate correct-by-construction embedded code
- Industrial tool: ANSYS Scade

Challenges

- Noisy environment, perceived through noisy sensors
- Interaction with other autonomous entities

Existing approaches

- Manually implement stochastic controller: *Can be error prone*
- Offline statistical tests: *Requires up-to-date offline data*



Uncertainty in Embedded Systems

Synchronous languages

- High-level specification language
- Generate correct-by-construction embedded code
- Industrial tool: ANSYS Scade

Challenges

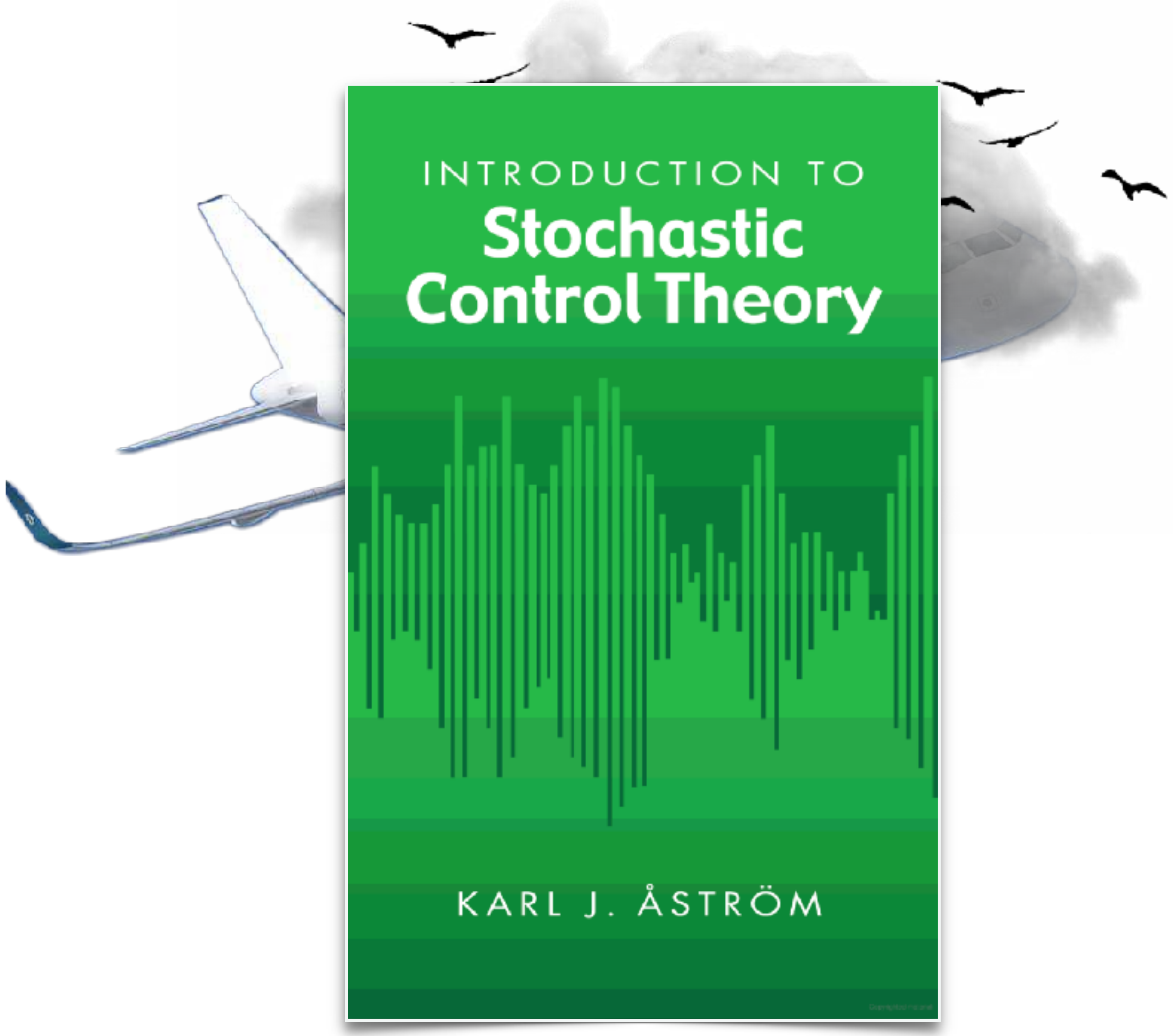
- Noisy environment, perceived through noisy sensors
- Interaction with other autonomous entities

Existing approaches

- Manually implement stochastic controller: *Can be error prone*
- Offline statistical tests: *Requires up-to-date offline data*

Reactive Probabilistic Programming

- Synchronous languages with probabilistic constructs
- Make the probabilistic model explicit
- Automatically learn posterior distributions from observations

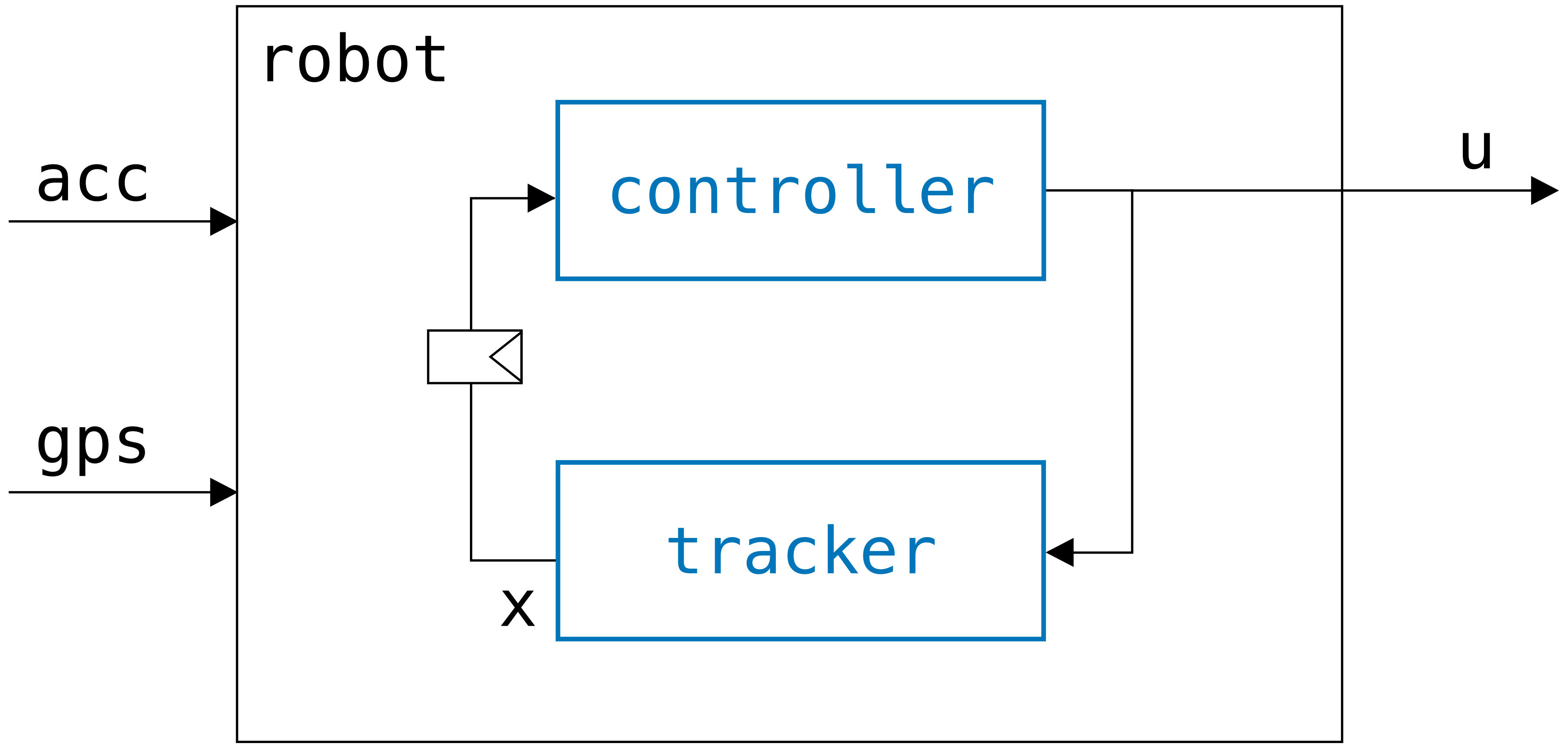


Reactive Systems

Synchronous data-flow languages and block diagrams

- Signal: stream of values
- System: stream processor

State: `x`: $(position \times velocity \times acceleration)$

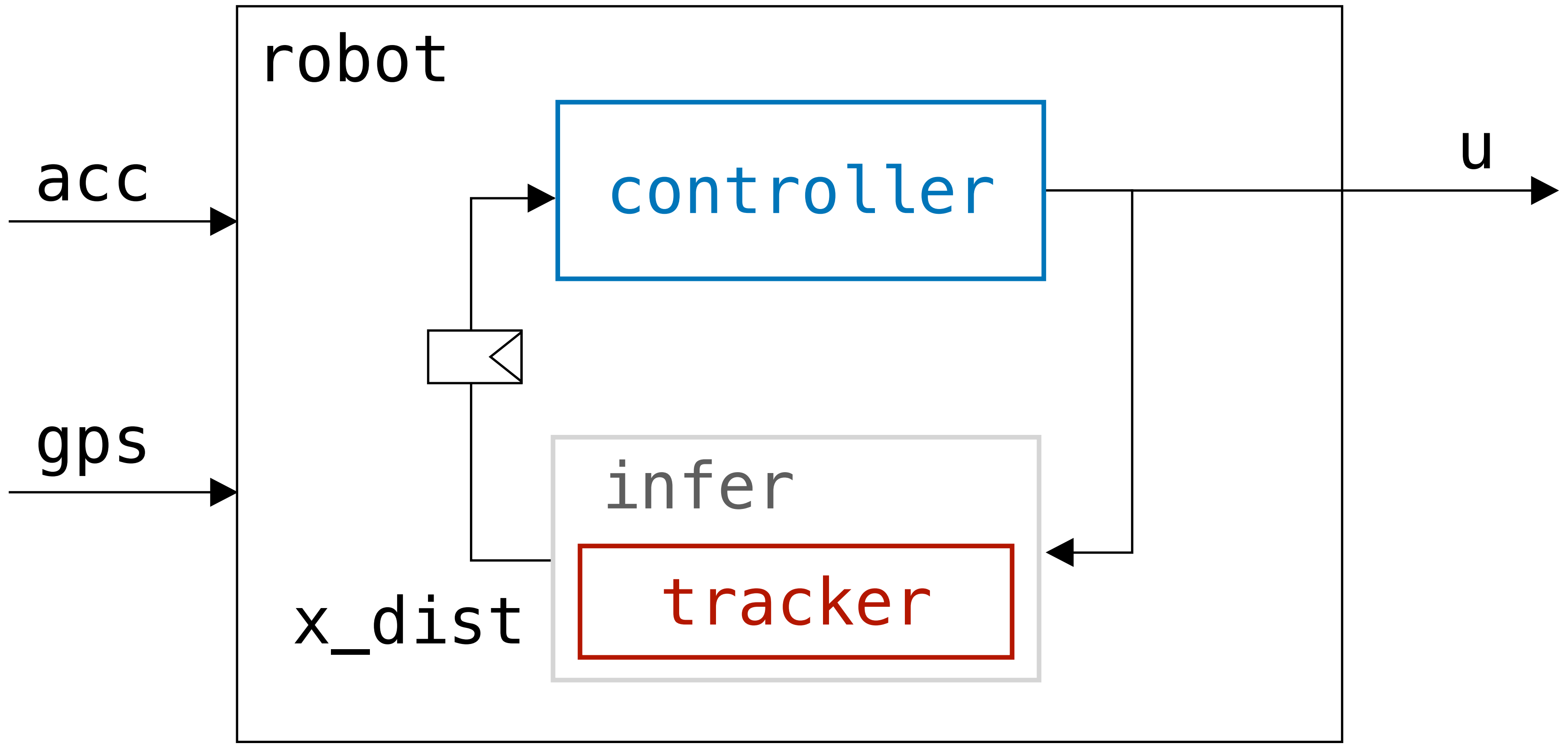


Reactive Probabilistic Systems

Synchronous data-flow languages and block diagrams

- Signal: stream of values
- System: stream processor

State: `x_dist`: $(\mathit{position} \times \mathit{velocity} \times \mathit{acceleration})$ `dist`



ProbZelus: add support to deal with uncertainty

- Extend a synchronous language
- Parallel composition: deterministic/probabilistic
- Inference-in-the-loop
- Streaming inference

Synchronous Programming

Reactive Probabilistic Programming

`Lustre → Lucid Synchrone → Zelus → ProbZelus`

Dataflow synchronous programming

- Set of stream equations
- Discrete logical time steps
- At each step, compute the current value given inputs and previous values

Stream operations

- Constant are lifted to stream: `1 = 1, 1, 1, ....`
- Temporal operators: `->`, `pre`, `fby`
- Control structures: `reset/every`, `present`, `automaton`

Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

- Set of stream equations
- Discrete logical time steps
- At each step, compute the current value given input and previous values

```
node nat v = cpt where
  rec cpt = v → pre cpt + 1
```

$$cpt_n = \mathit{if}\ (n = 0)\ \mathit{then}\ v_0\ \mathit{else}\ cpt_{n-1} + 1$$

Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

- Set of stream equations
- Discrete logical time steps
- At each step, compute the current value given input and previous values

```
node nat v = cpt where
  rec cpt = v → pre cpt + 1
```

$$cpt_n = if\ (n = 0)\ then\ v_0\ else\ cpt_{n-1} + 1$$

```
nat
```

$t = 0$

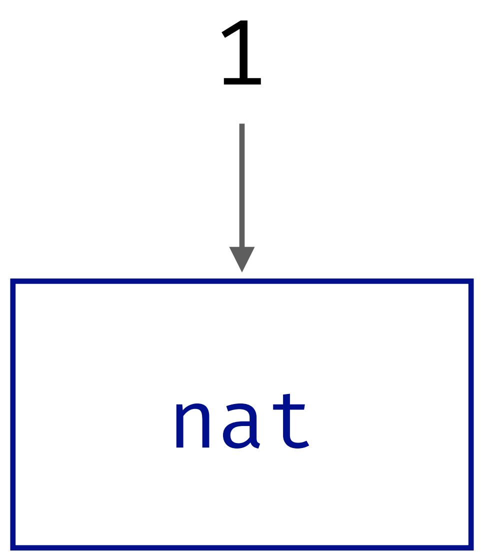
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

- Set of stream equations
- Discrete logical time steps
- At each step, compute the current value given input and previous values

```
node nat v = cpt where
  rec cpt = v → pre cpt + 1
```

$$cpt_n = if\ (n = 0)\ then\ v_0\ else\ cpt_{n-1} + 1$$



$t = 0$

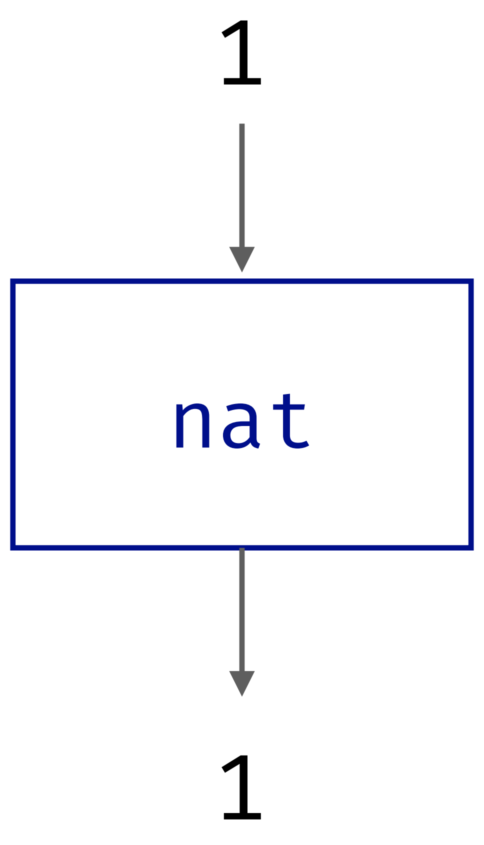
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

- Set of stream equations
- Discrete logical time steps
- At each step, compute the current value given input and previous values

```
node nat v = cpt where
  rec cpt = v -> pre cpt + 1
```

$$cpt_n = if\ (n = 0)\ then\ v_0\ else\ cpt_{n-1} + 1$$



$t = 0$

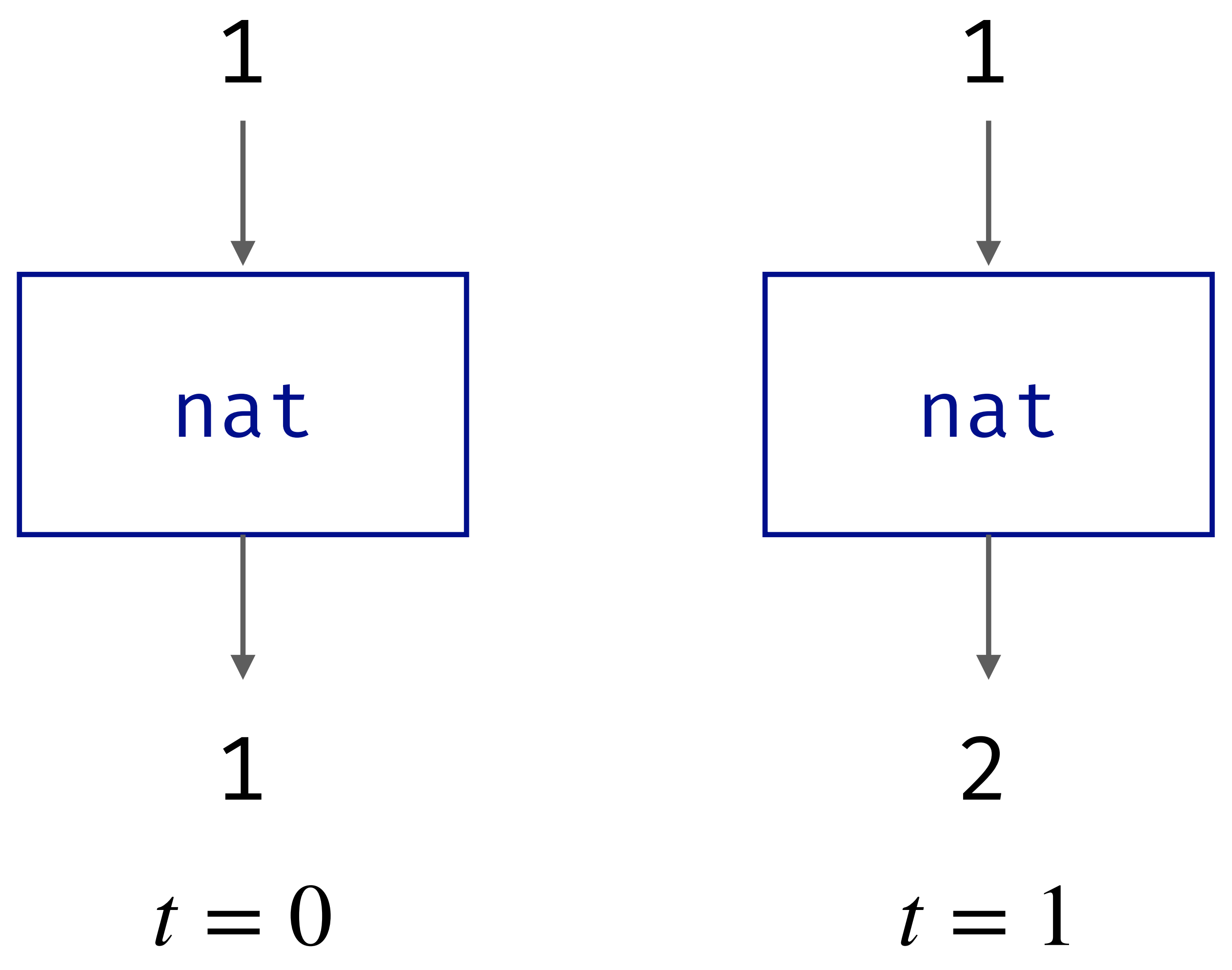
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

- Set of stream equations
- Discrete logical time steps
- At each step, compute the current value given input and previous values

```
node nat v = cpt where
  rec cpt = v → pre cpt + 1
```

$$cpt_n = if\ (n = 0)\ then\ v_0\ else\ cpt_{n-1} + 1$$



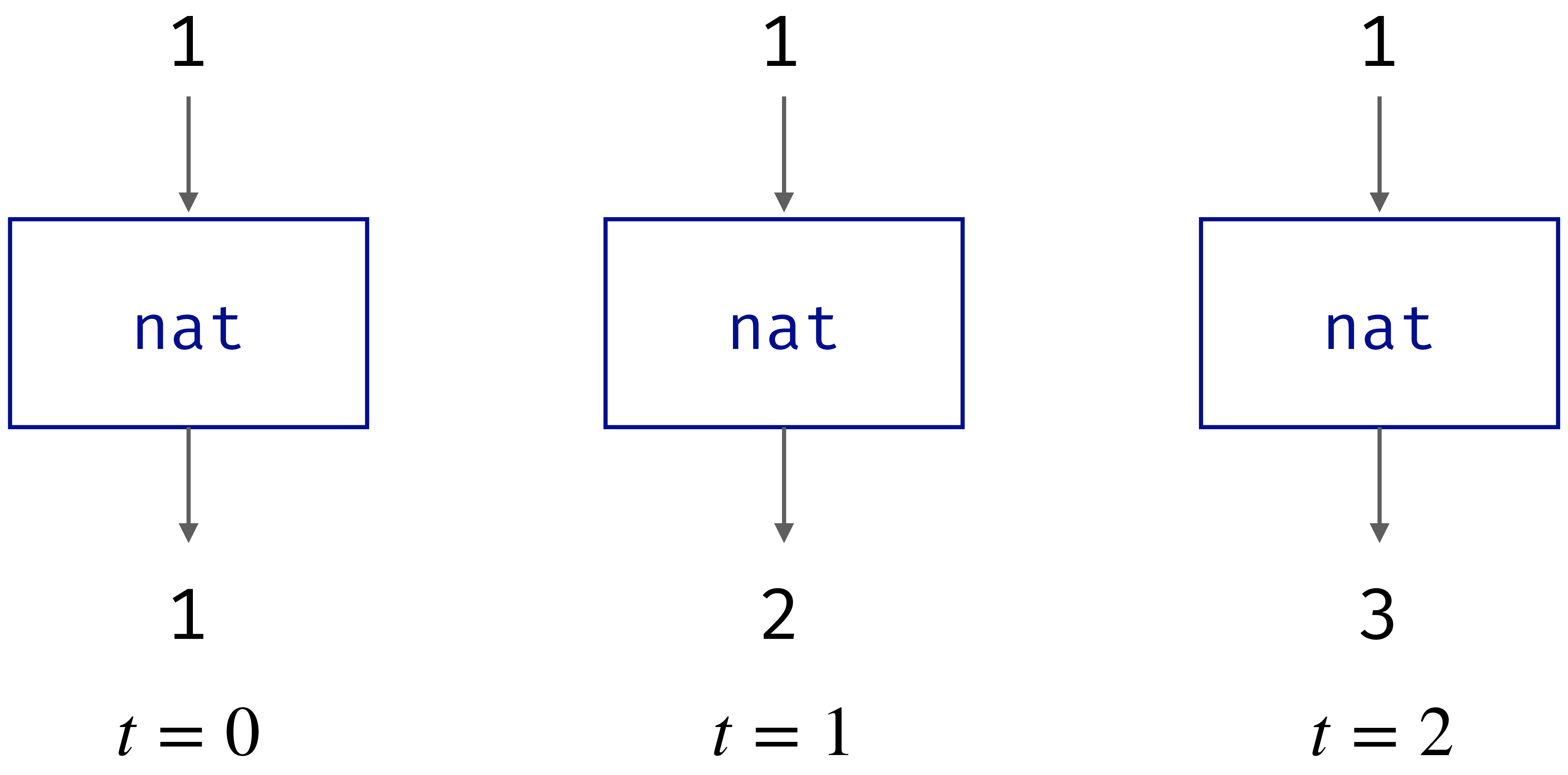
`Lustre → Lucid Synchrone → Zelus → ProbZelus`

Dataflow synchronous programming

- Set of stream equations
- Discrete logical time steps
- At each step, compute the current value given input and previous values

```
node nat v = cpt where
  rec cpt = v → pre cpt + 1
```

$$cpt_n = if\ (n = 0)\ then\ v_0\ else\ cpt_{n-1} + 1$$



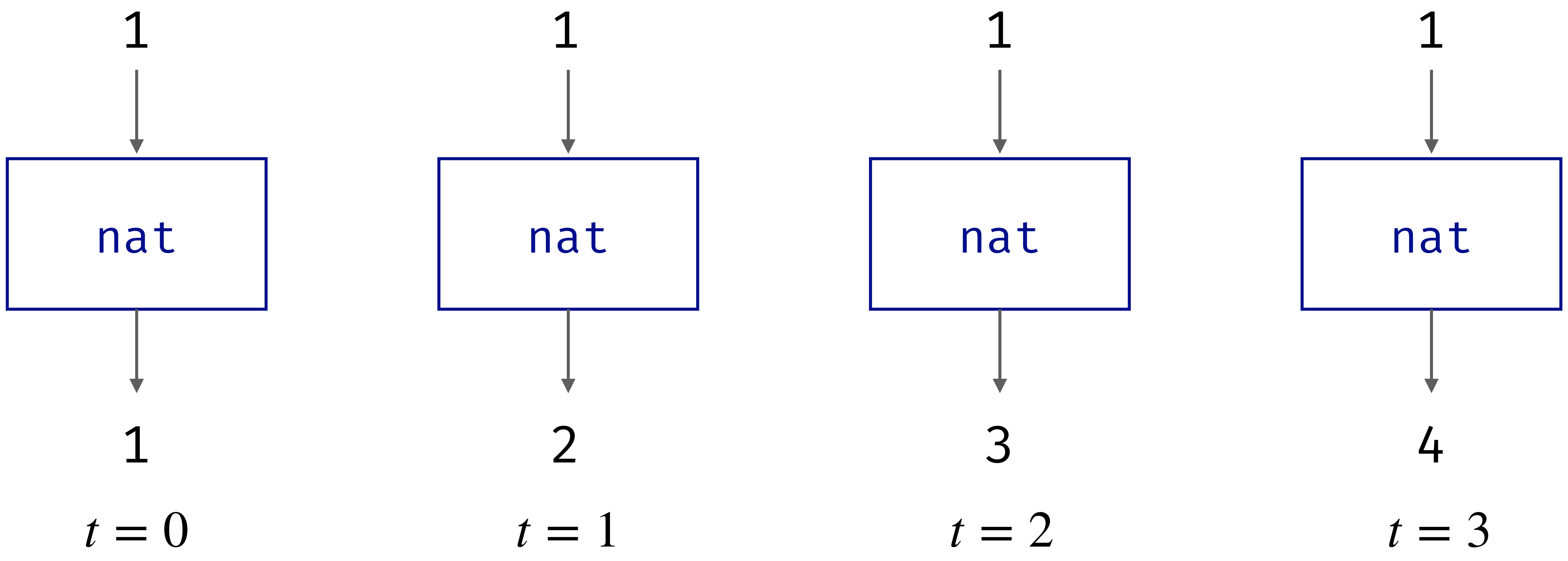
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

- Set of stream equations
- Discrete logical time steps
- At each step, compute the current value given input and previous values

```
node nat v = cpt where
  rec cpt = v → pre cpt + 1
```

$$cpt_n = if\ (n = 0)\ then\ v_0\ else\ cpt_{n-1} + 1$$



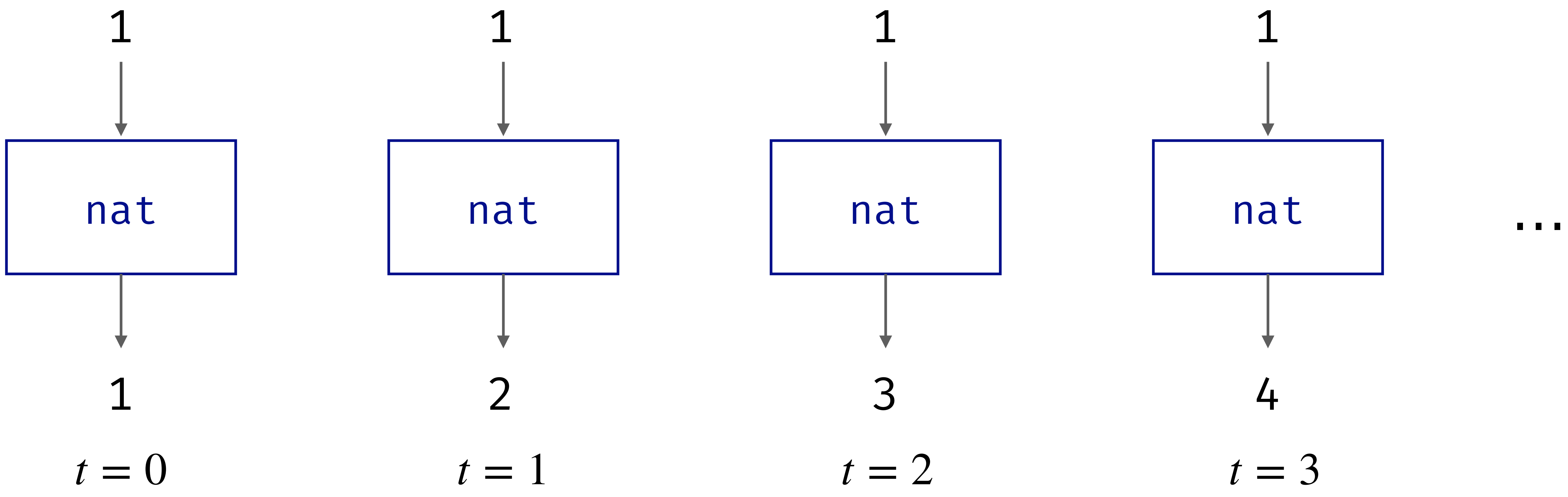
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

- Set of stream equations
- Discrete logical time steps
- At each step, compute the current value given input and previous values

```
node nat v = cpt where
  rec cpt = v → pre cpt + 1
```

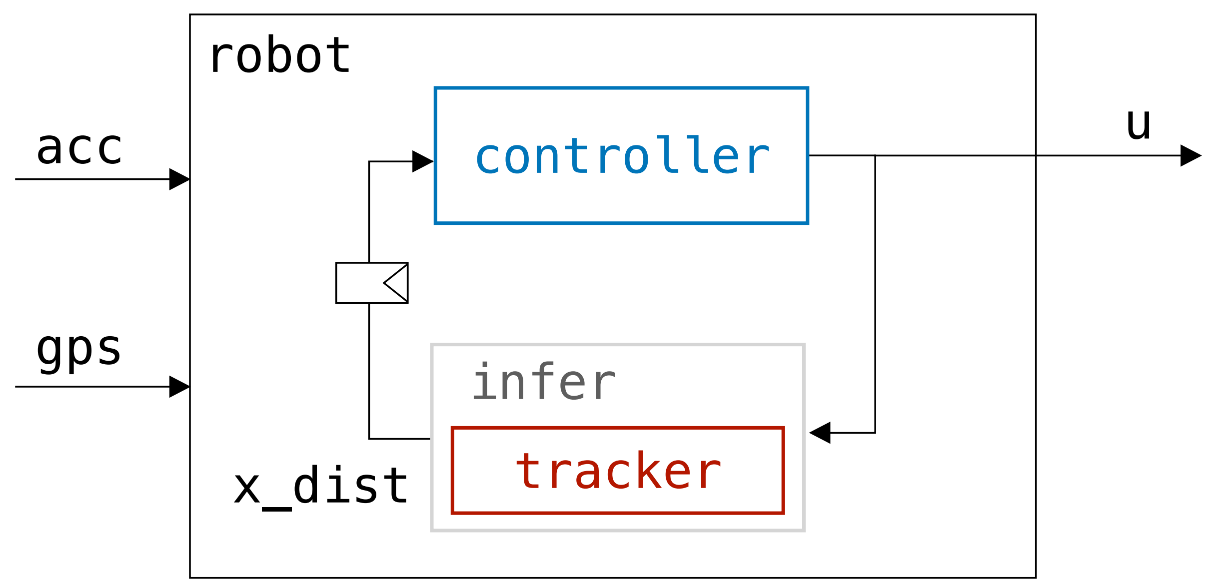
$$cpt_n = if\ (n = 0)\ then\ v_0\ else\ cpt_{n-1} + 1$$



ProbZelus

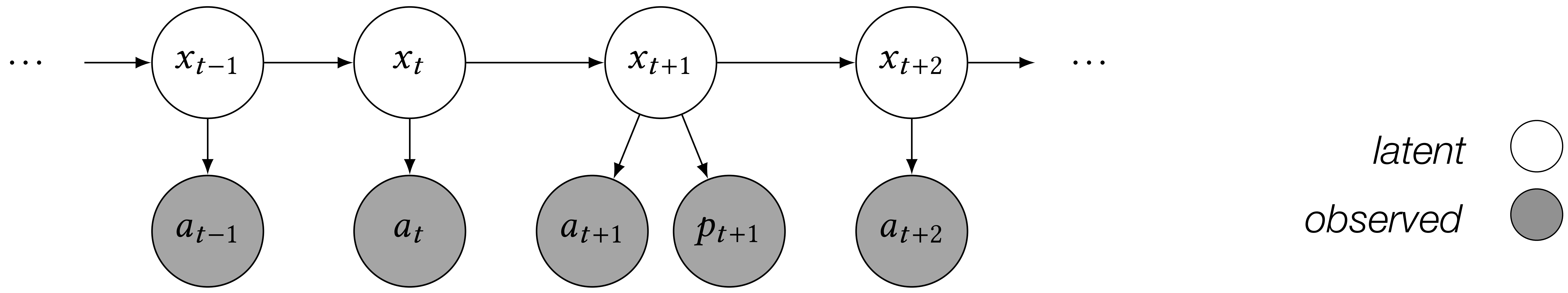
Reactive Probabilistic Programming

Reactive Probabilistic Programming



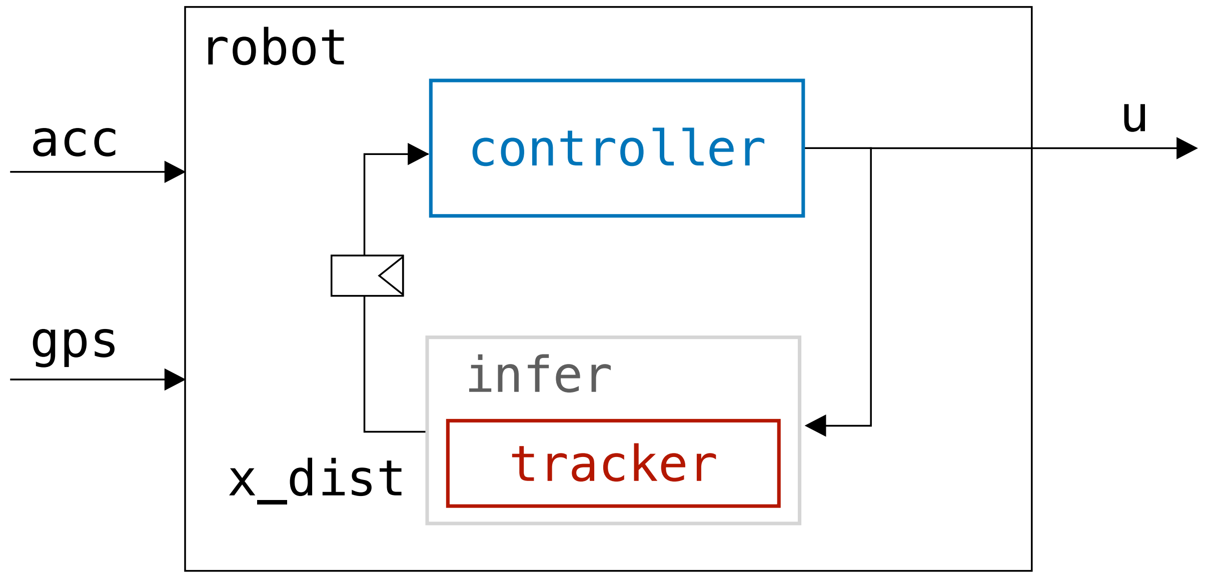
Extend Zelus with probabilistic constructs

- `x = sample(d)`: introduce a random variable `x` of distribution `d`
- `observe(d, y)`: condition on the fact that `y` was sampled from `d`
- `infer m obs`: compute the distribution of output of the model `m` with respect to `obs`



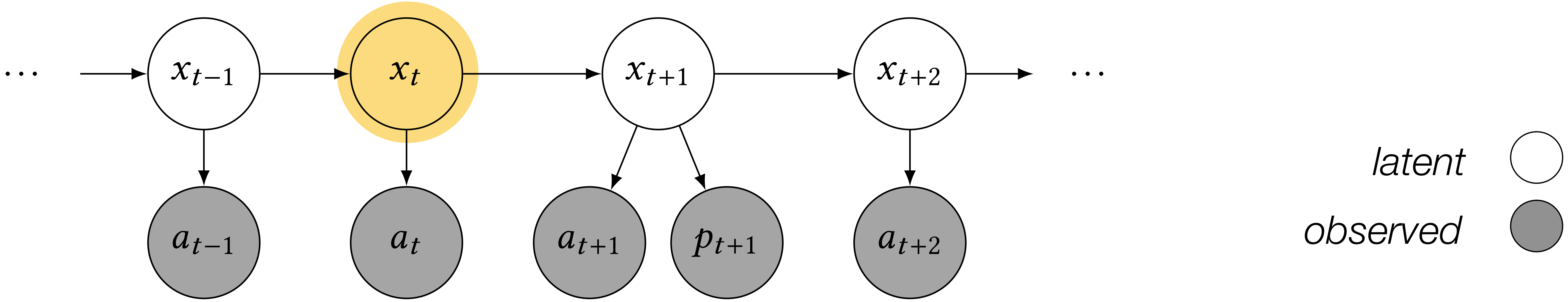

```
proba tracker (u, acc, gps) = x where
  rec  x = sample (mv_gaussian (motion (u, x0 → pre x), noise))
  and () = observe (gaussian (get_acc x, 1.0), acc)
  and present gps (pos) →
          do () = observe (gaussian (get_pos x, 0.01), pos) done
```

Reactive Probabilistic Programming



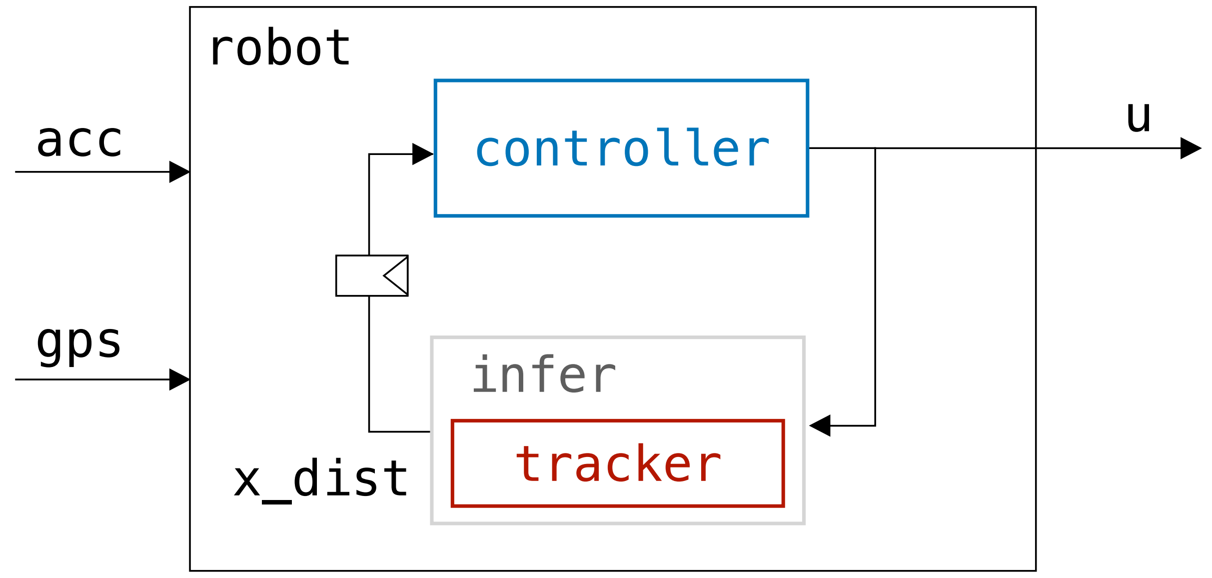
Extend Zelus with probabilistic constructs

- `x = sample(d)`: introduce a random variable `x` of distribution `d`
- `observe(d, y)`: condition on the fact that `y` was sampled from `d`
- `infer m obs`: compute the distribution of output of the model `m` with respect to `obs`



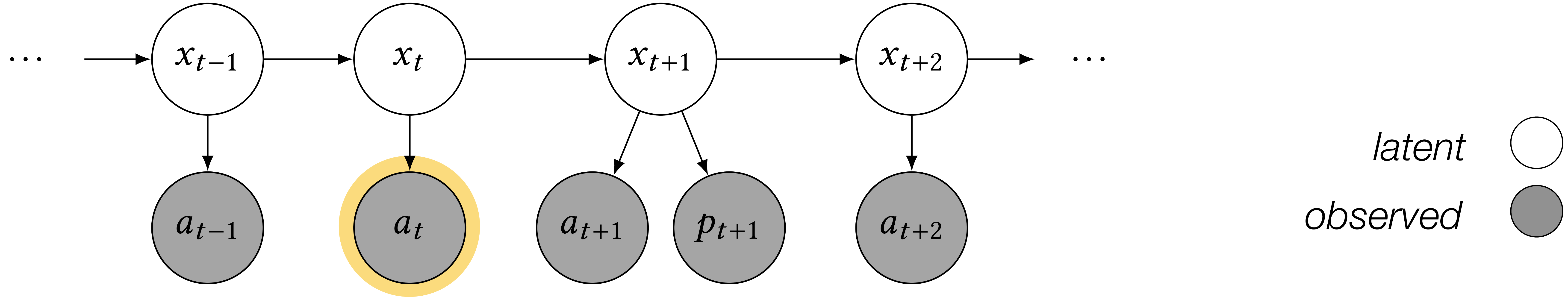

```
proba tracker (u, acc, gps) = x where
  rec  x = sample (mv_gaussian (motion (u, x0 → pre x), noise))
  and () = observe (gaussian (get_acc x, 1.0), acc)
  and present gps (pos) →
          do () = observe (gaussian (get_pos x, 0.01), pos) done
```

Reactive Probabilistic Programming



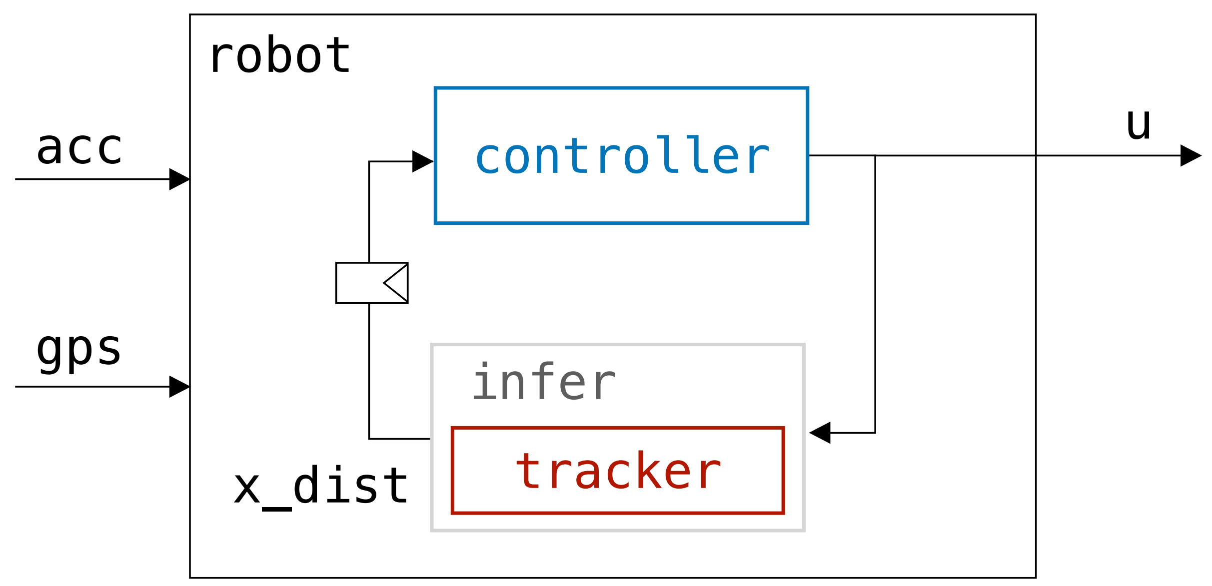
Extend Zelus with probabilistic constructs

- `x = sample(d)`: introduce a random variable `x` of distribution `d`
- `observe(d, y)`: condition on the fact that `y` was sampled from `d`
- `infer m obs`: compute the distribution of output of the model `m` with respect to `obs`



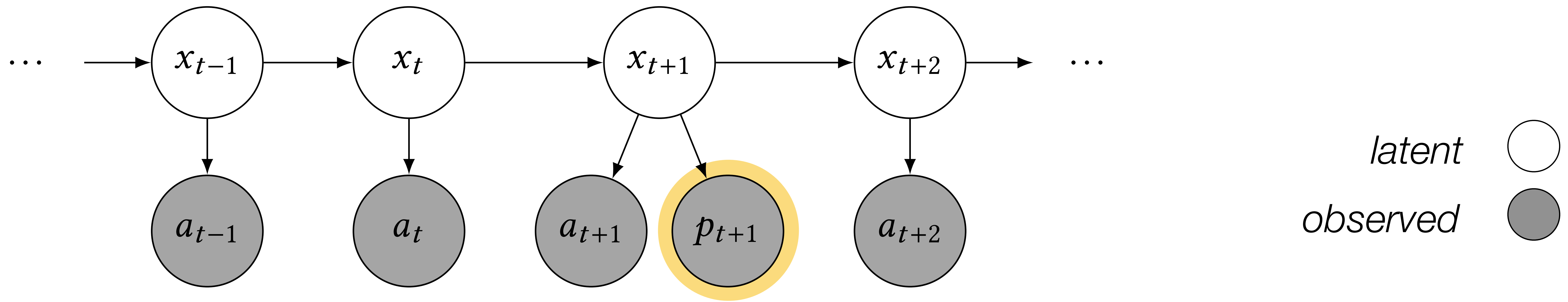

```
proba tracker (u, acc, gps) = x where
  rec  x = sample (mv_gaussian (motion (u, x0 → pre x), noise))
  and () = observe (gaussian (get_acc x, 1.0), acc)
  and present gps (pos) →
          do () = observe (gaussian (get_pos x, 0.01), pos) done
```

Reactive Probabilistic Programming



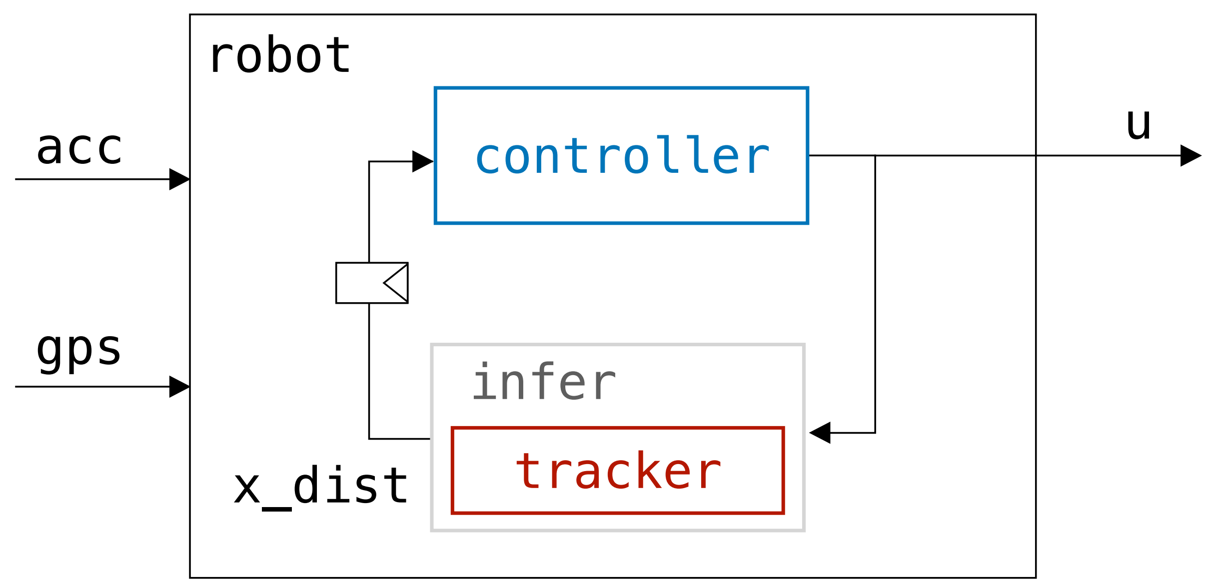
Extend Zelus with probabilistic constructs

- `x = sample(d)`: introduce a random variable `x` of distribution `d`
- `observe(d, y)`: condition on the fact that `y` was sampled from `d`
- `infer m obs`: compute the distribution of output of the model `m` with respect to `obs`



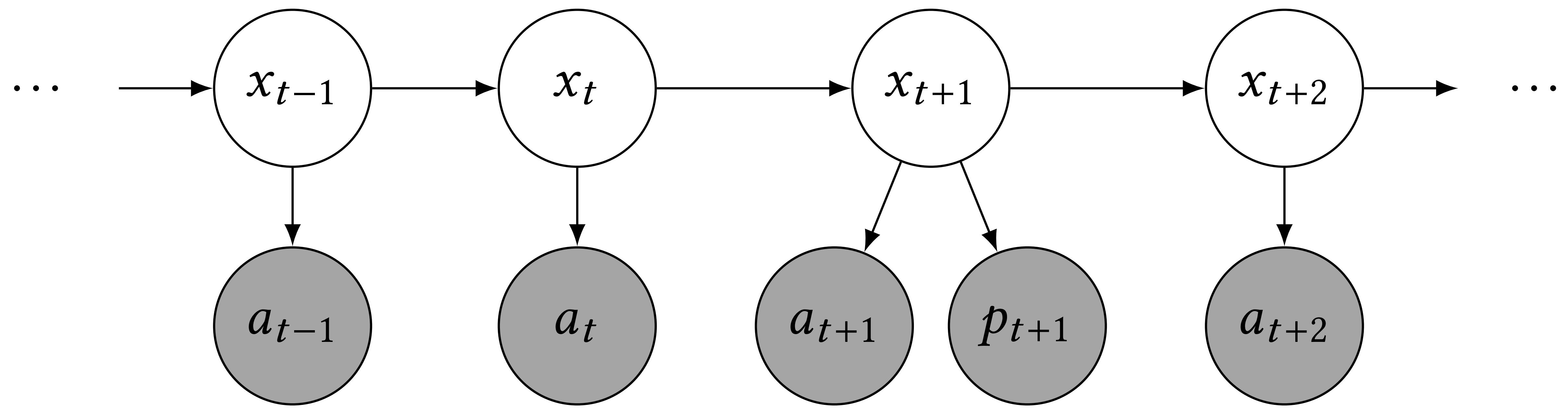

```
proba tracker (u, acc, gps) = x where
  rec  x = sample (mv_gaussian (motion (u, x0 → pre x), noise))
  and () = observe (gaussian (get_acc x, 1.0), acc)
  and present gps (pos) →
          do () = observe (gaussian (get_pos x, 0.01), pos) done
```

Reactive Probabilistic Programming



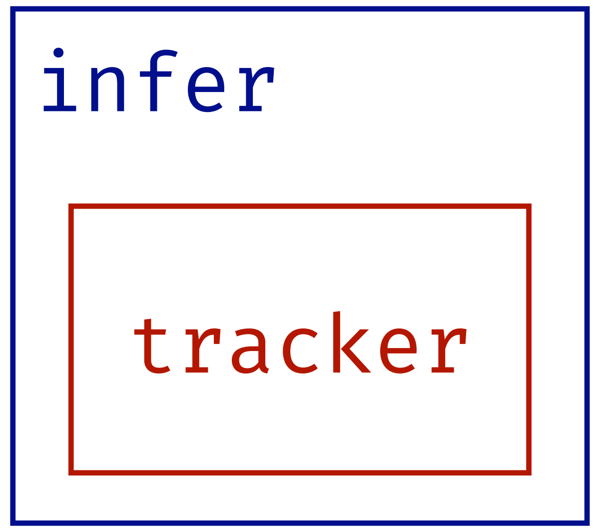
Extend Zelus with probabilistic constructs

- `x = sample(d)`: introduce a random variable `x` of distribution `d`
- `observe(d, y)`: condition on the fact that `y` was sampled from `d`
- `infer m obs`: compute the distribution of output of the model `m` with respect to `obs`



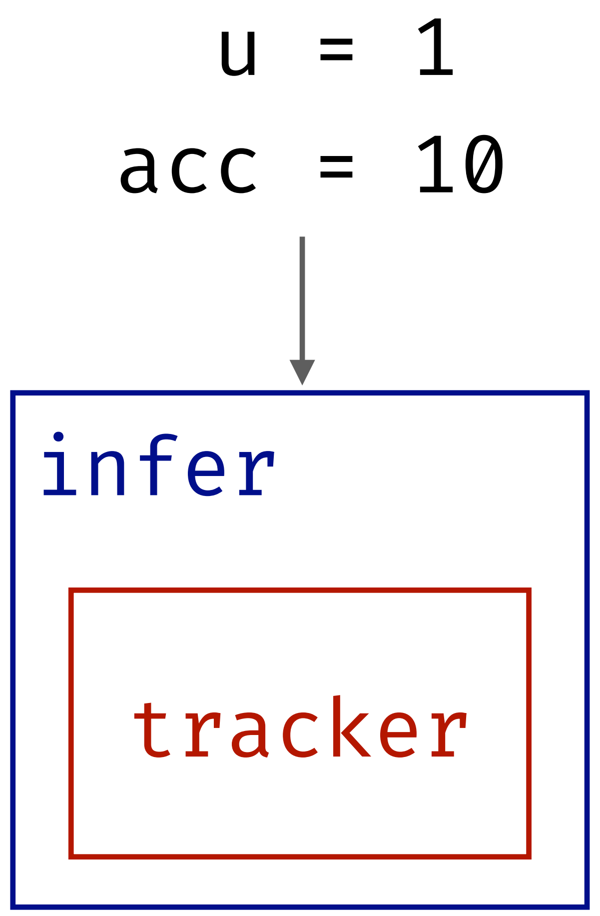

```
proba tracker (u, acc, gps) = x where
  rec  x = sample (mv_gaussian (motion (u, x0 → pre x), noise))
  and () = observe (gaussian (get_acc x, 1.0), acc)
  and present gps (pos) →
          do () = observe (gaussian (get_pos x, 0.01), pos) done
```

Reactive Probabilistic Programming



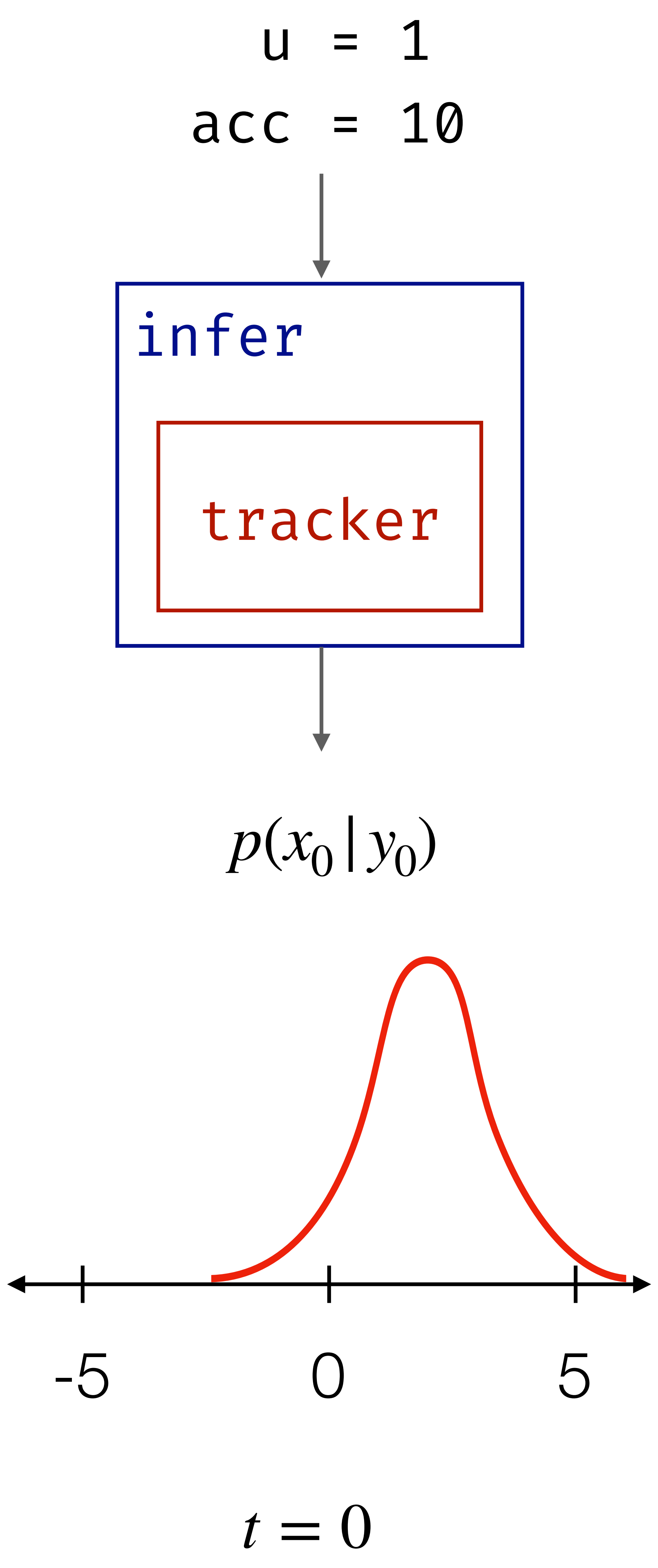
$t = 0$

Reactive Probabilistic Programming

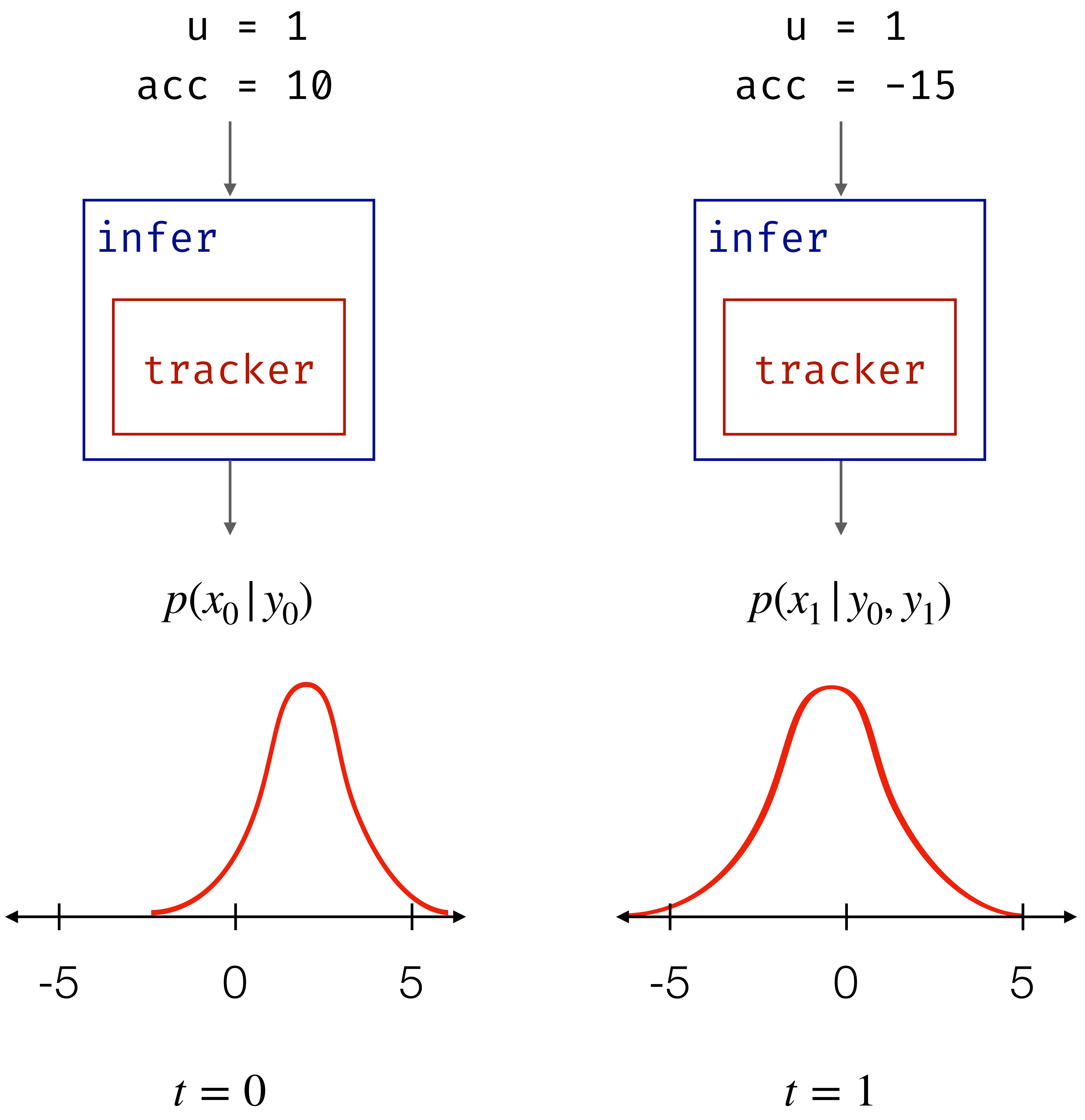


$t = 0$

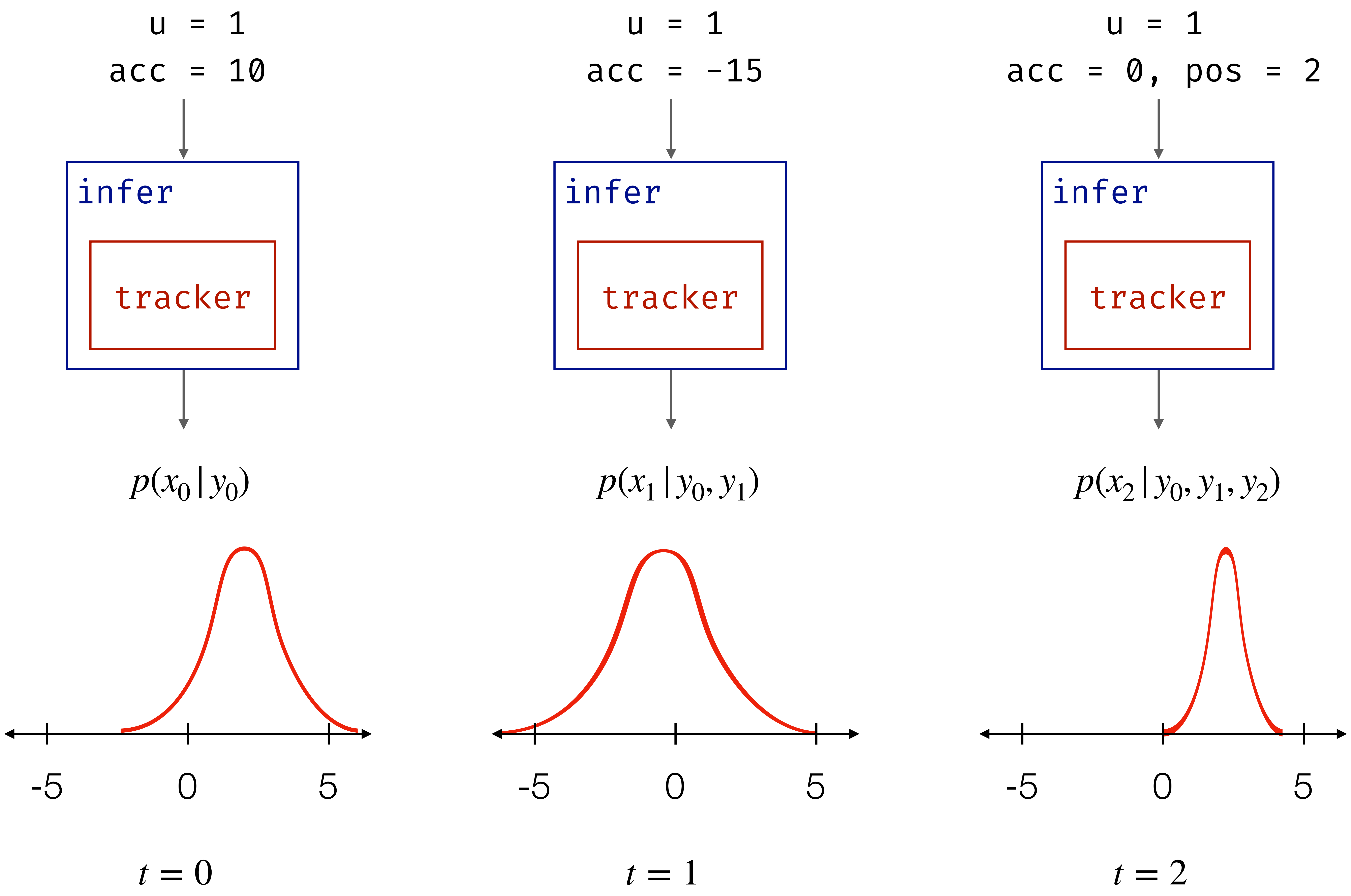
Reactive Probabilistic Programming



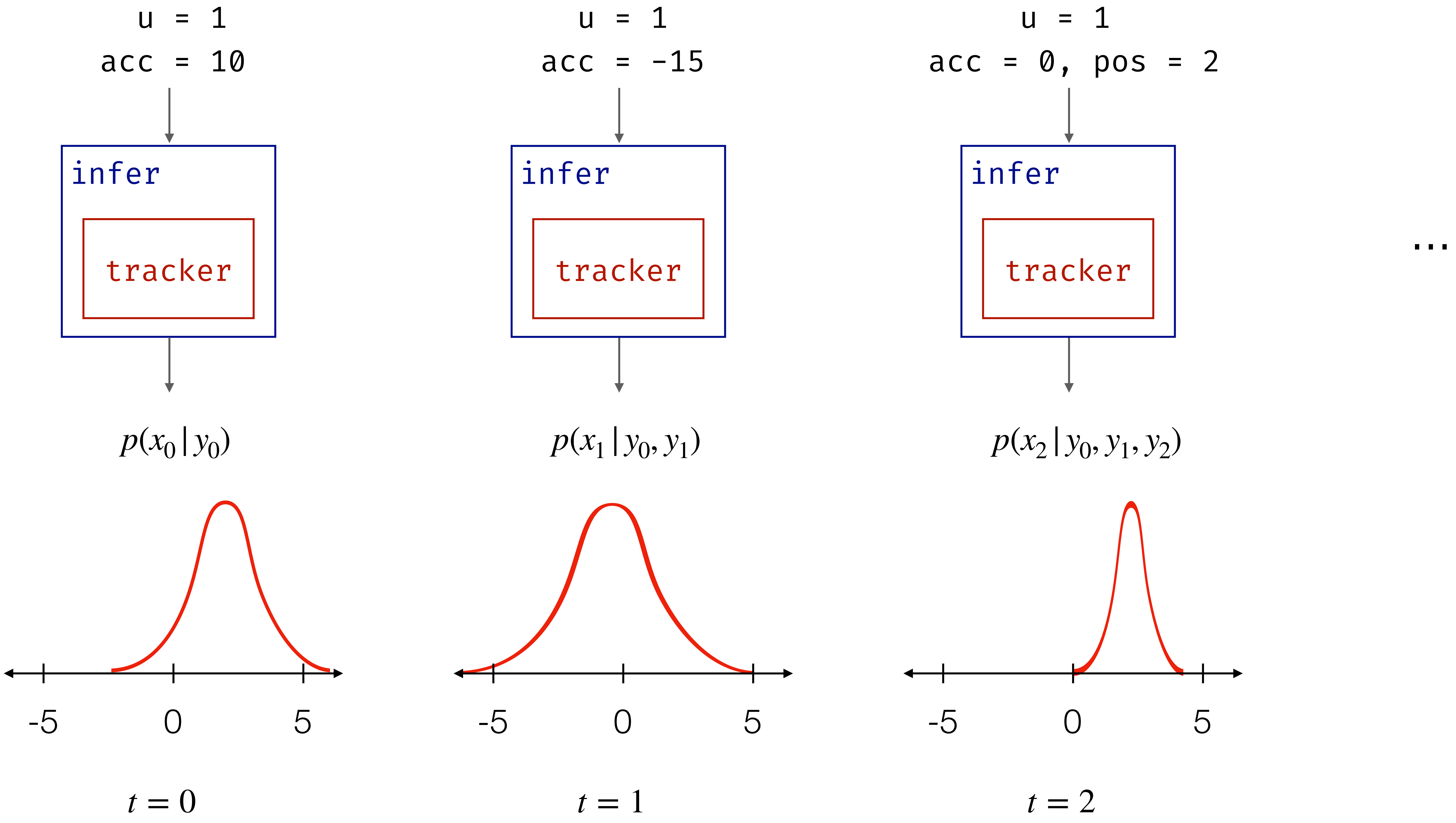
Reactive Probabilistic Programming



Reactive Probabilistic Programming



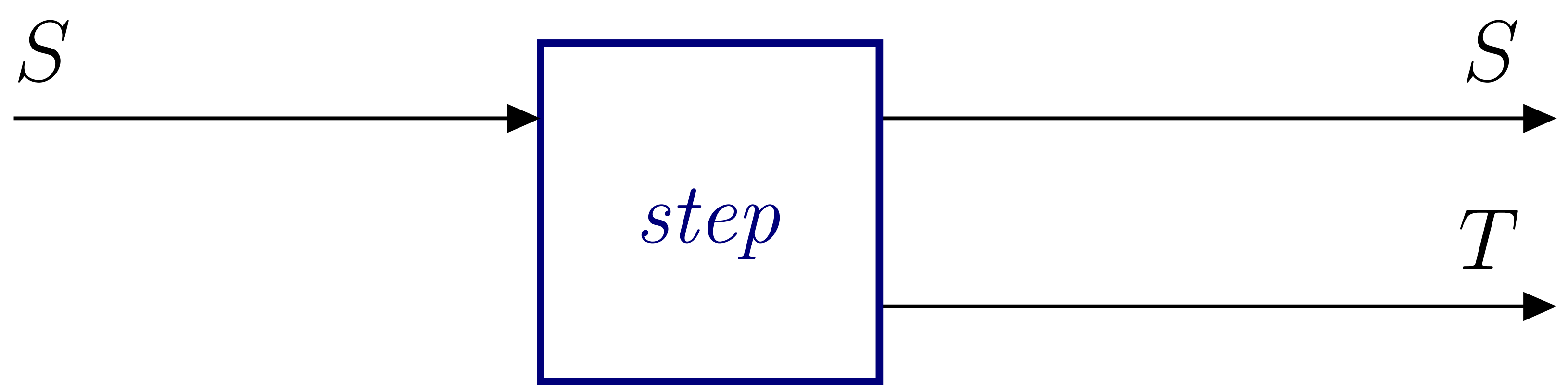
Reactive Probabilistic Programming



Deterministic Semantics

Initial state, transition function

$CoStream(T, S) = S \times (S \to S \times T)$

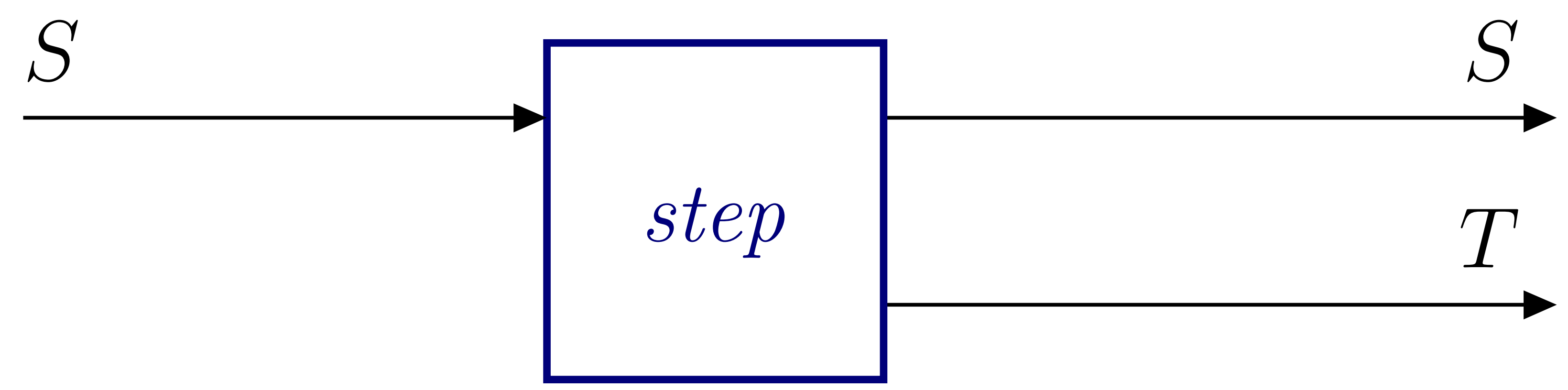


Caspi & Pouzet. CMCS 1998

Deterministic Semantics

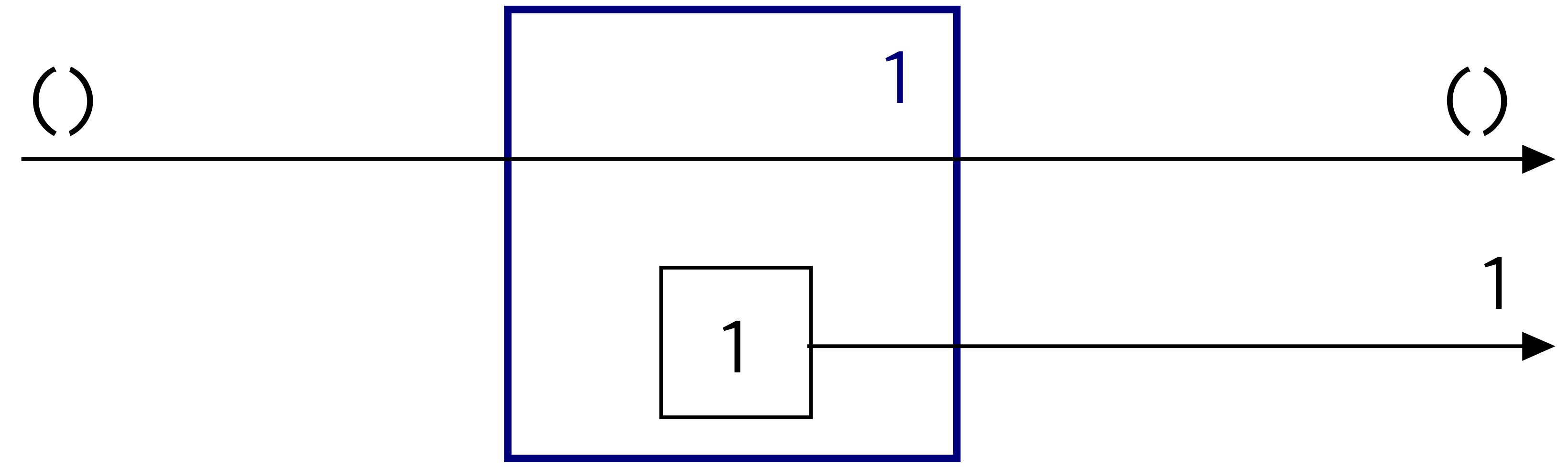
Initial state, transition function

$CoStream(T, S) = S \times (S \to S \times T)$



Constant 1: $\texttt{unit} \times (\texttt{unit} \to \texttt{unit} \times \texttt{int})$

- Initial state: the value of type unit
- Step function: return the empty state and the constant 1

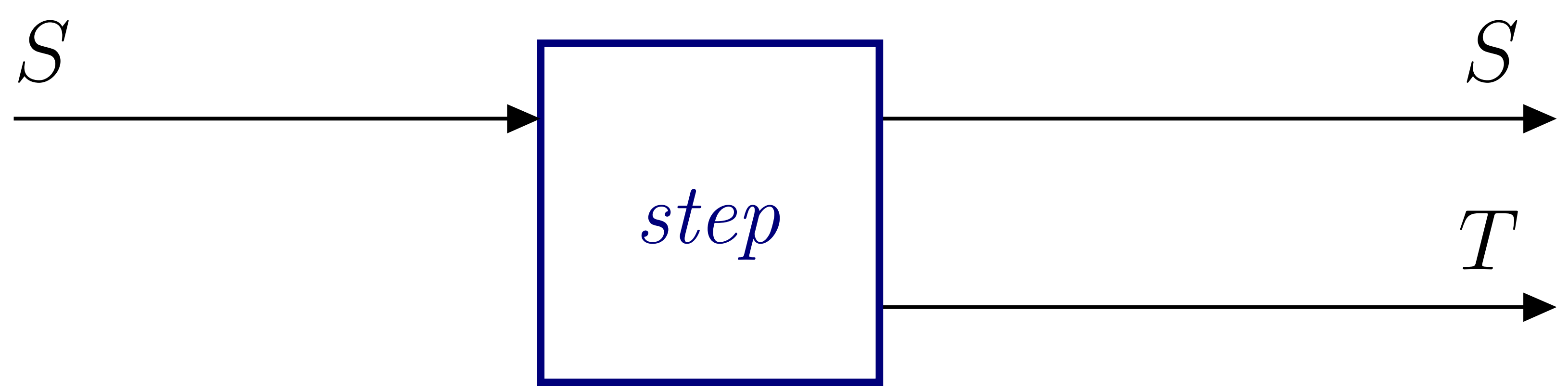


Caspi & Pouzet. CMCS 1998

Deterministic Semantics

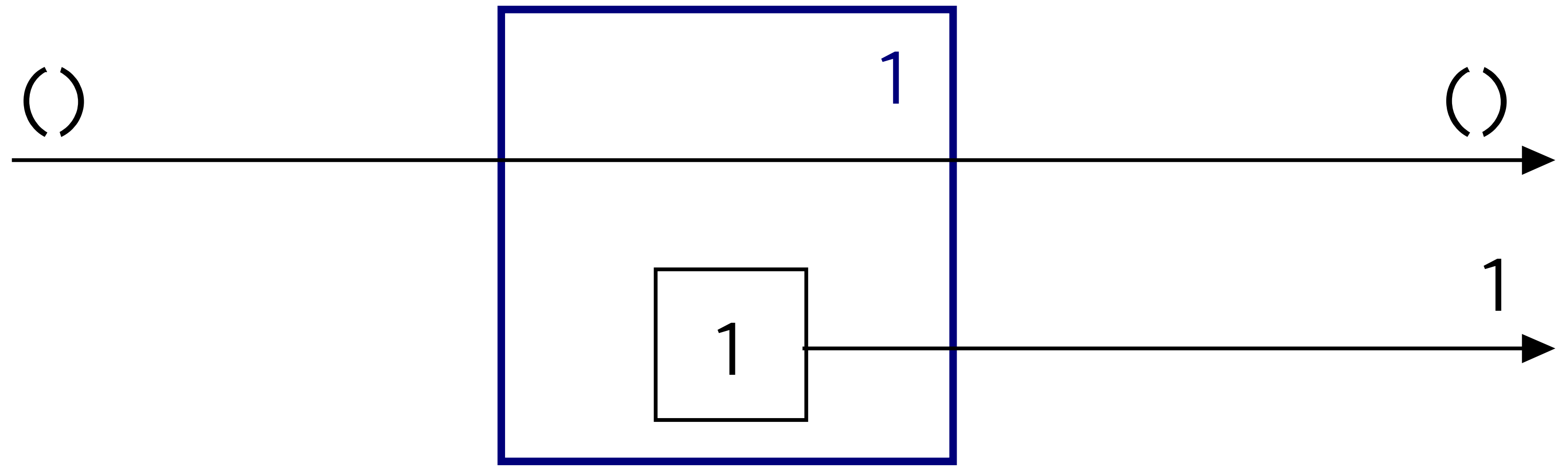
Initial state, transition function

$CoStream(T, S) = S \times (S \rightarrow S \times T)$



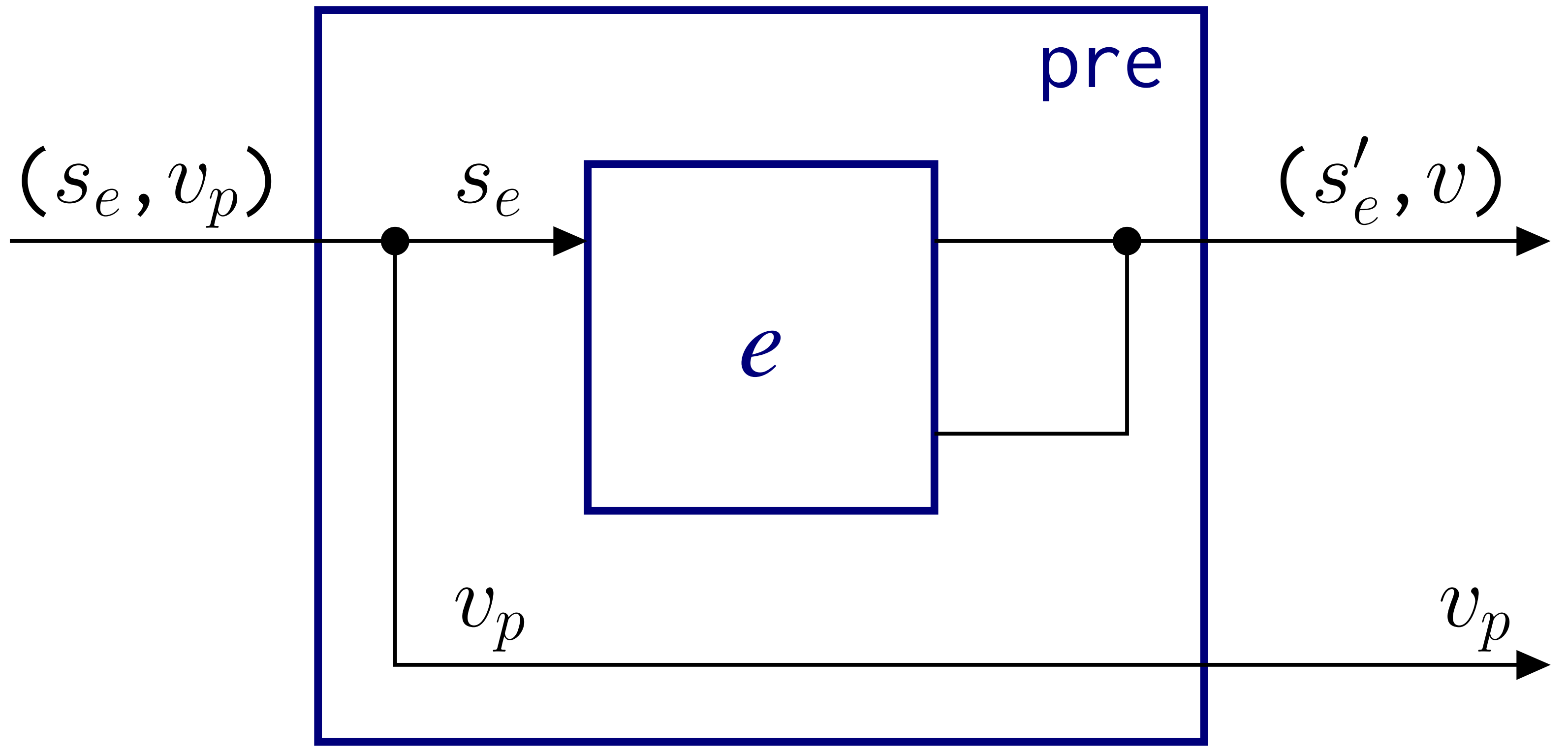
Constant 1: $\texttt{unit} \times (\texttt{unit} \rightarrow \texttt{unit} \times \texttt{int})$

- Initial state: the value of type unit
- Step function: return the empty state and the constant 1



Unit delay `pre` e: $(S \times T) \times (S \times T \rightarrow (S \times T) \times T)$

- Initial state: the initial state of e and default value
- Step function: the result of e is stored in the state and returned at the next iteration

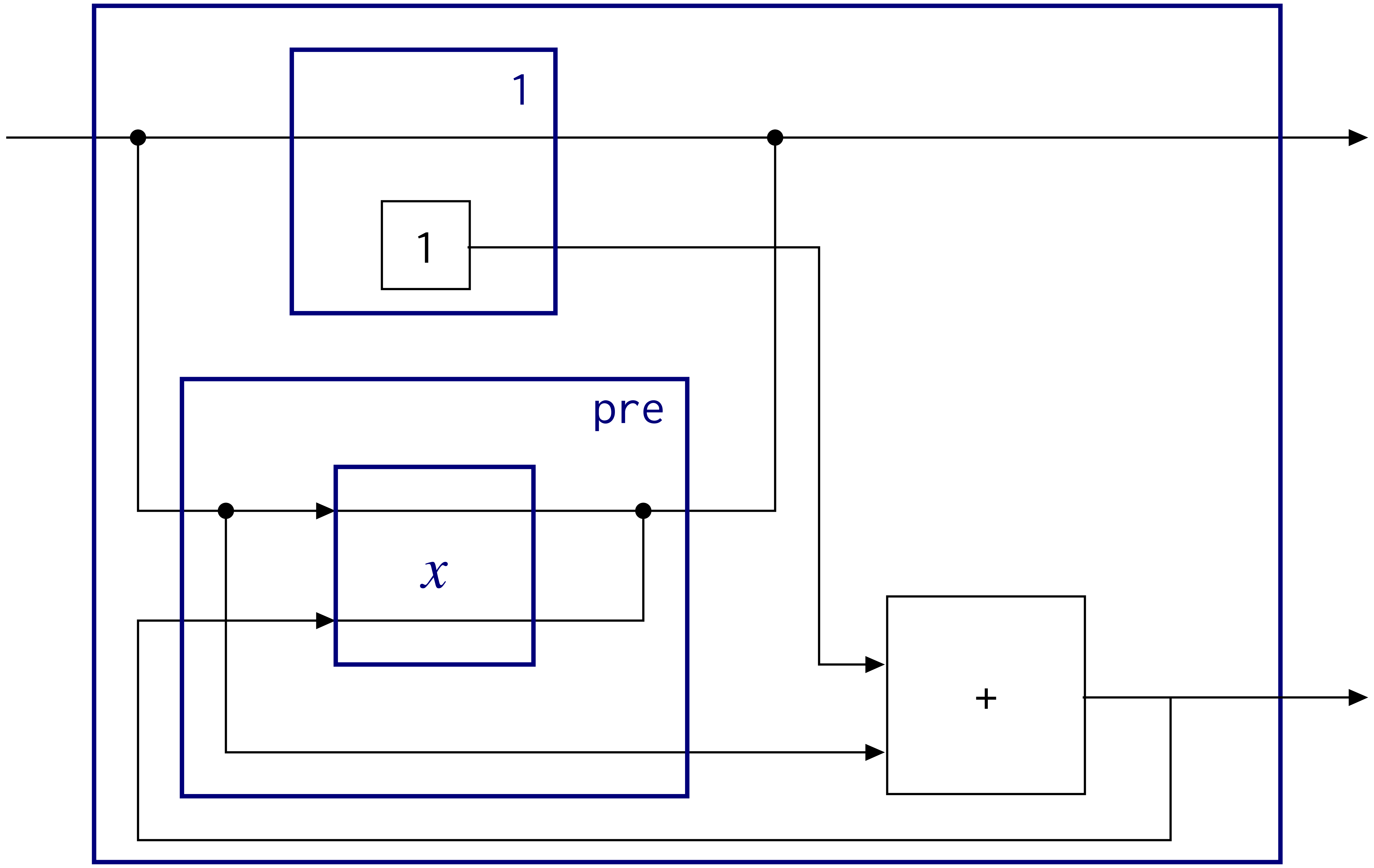


Caspi & Pouzet. CMCS 1998

Deterministic Semantics

```
rec x = 1 + pre x
```

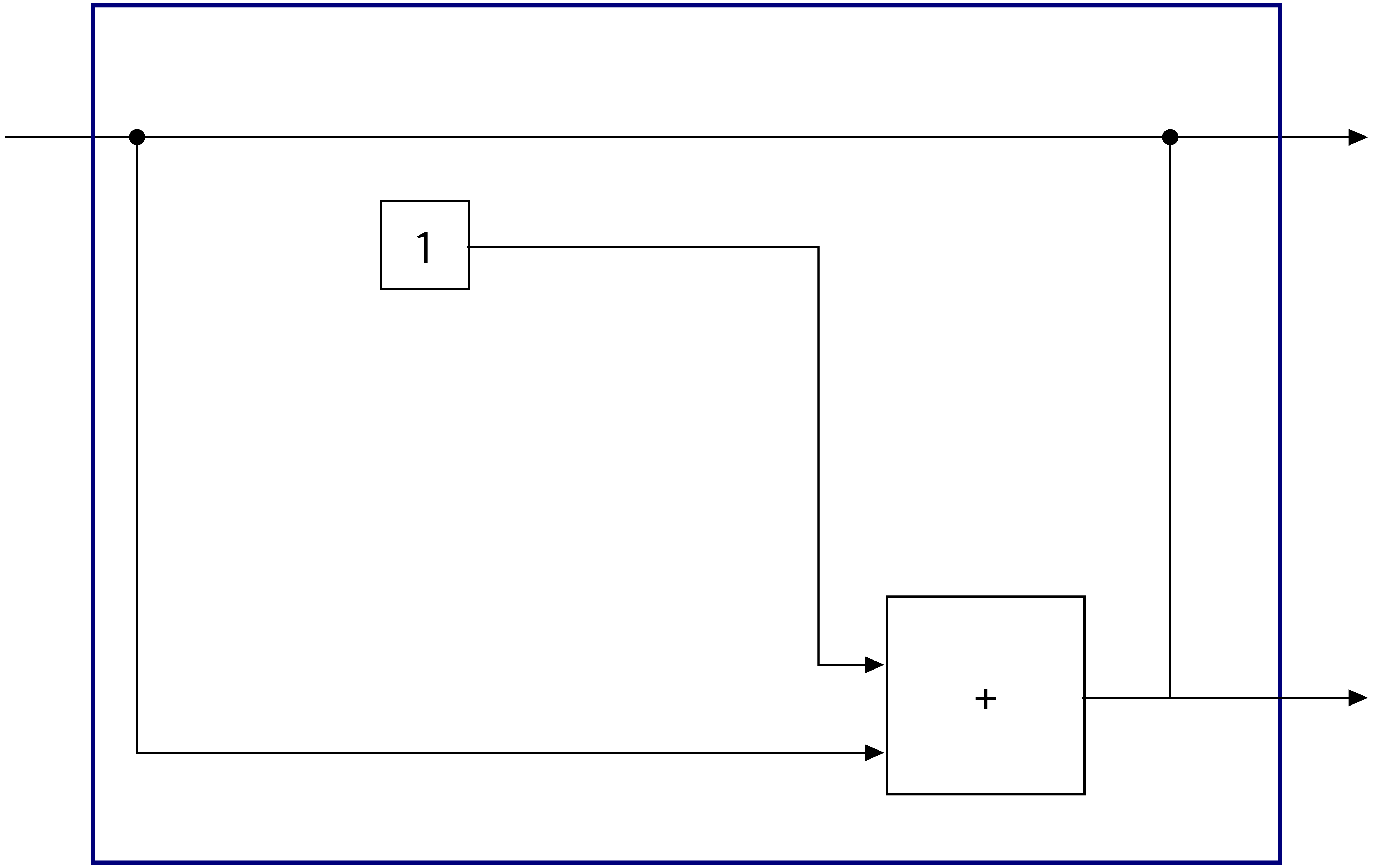
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

`rec x = 1 + pre x`

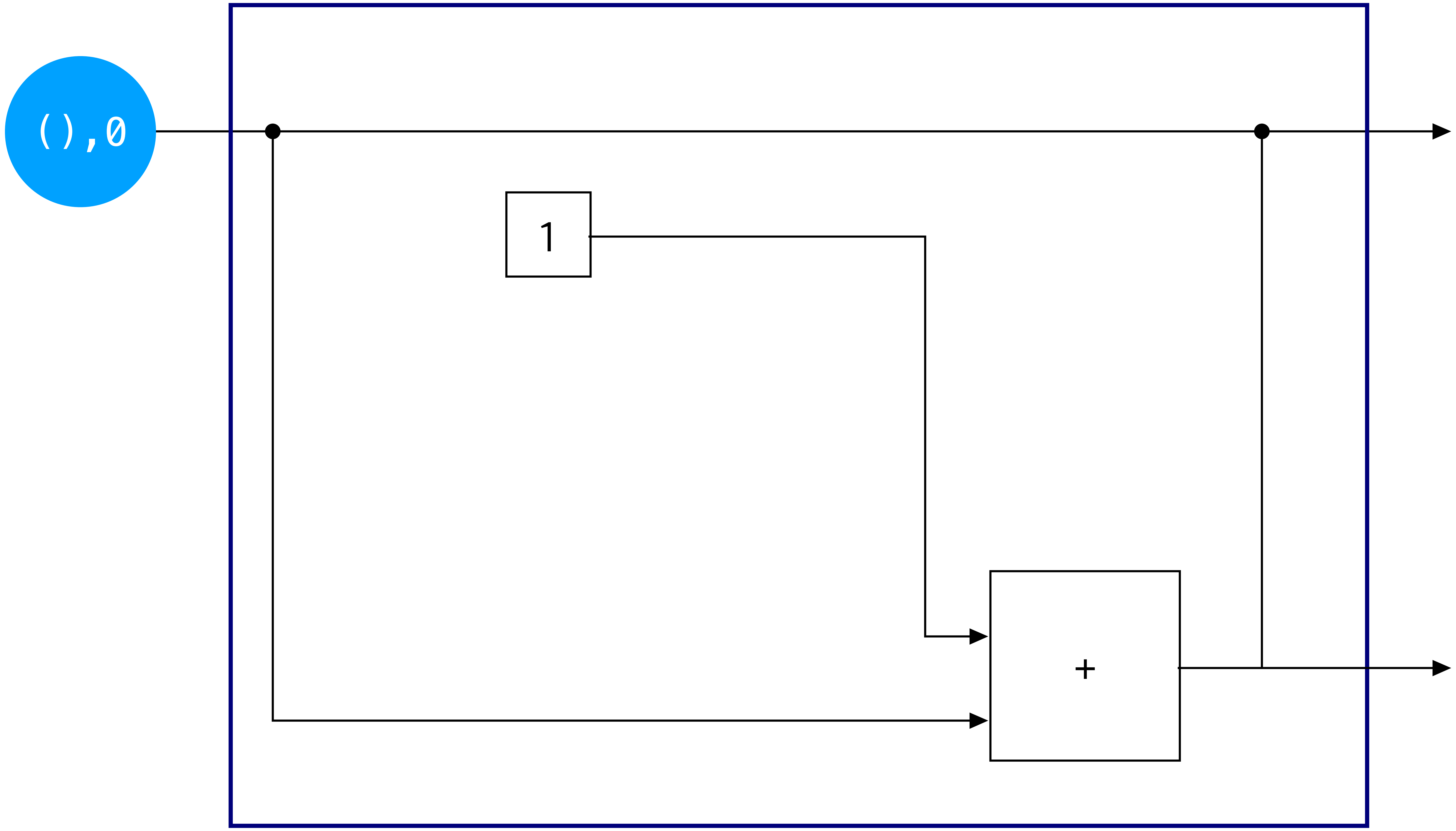
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

```
rec x = 1 + pre x
```

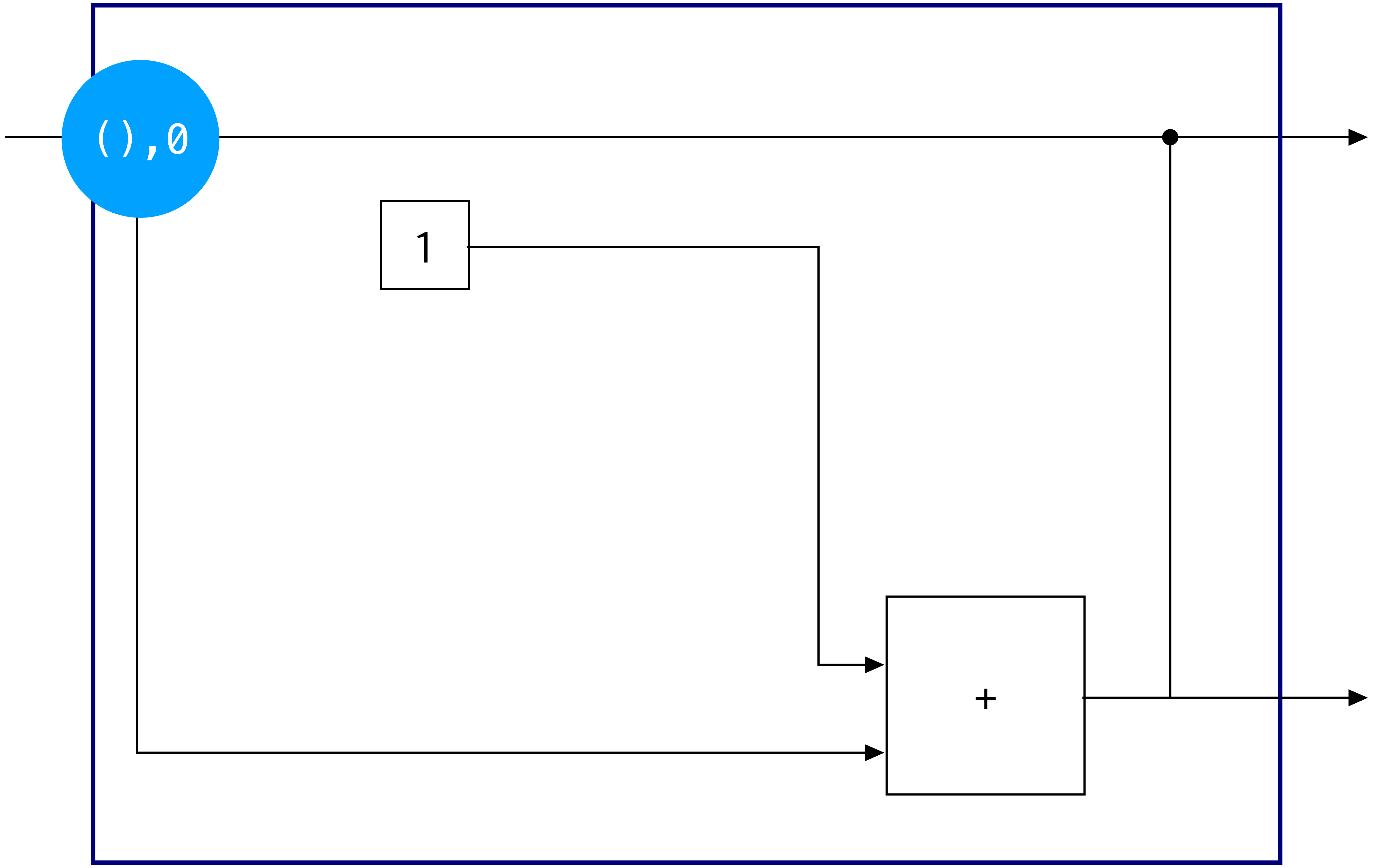
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

```
rec x = 1 + pre x
```

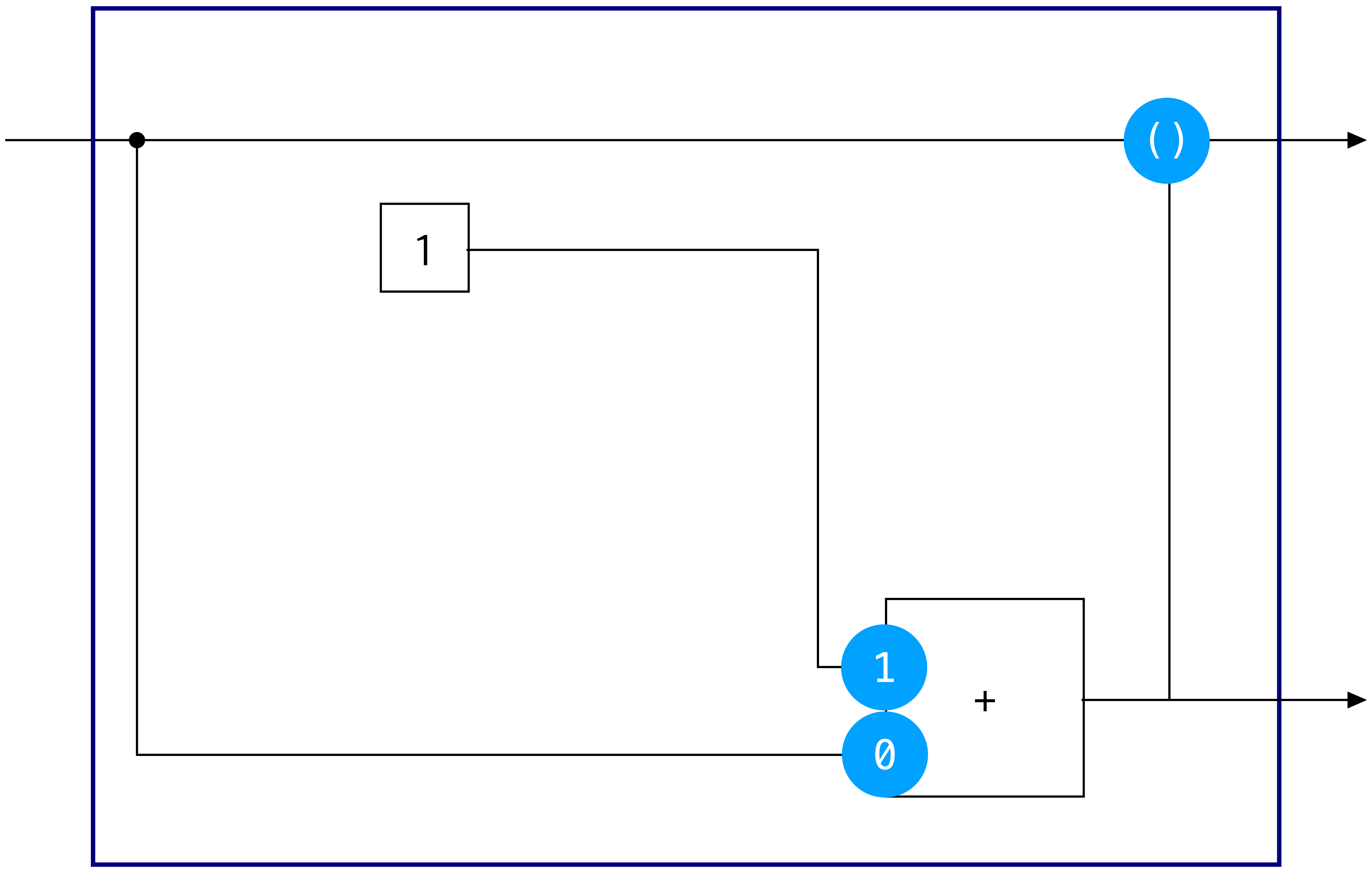
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

```
rec x = 1 + pre x
```

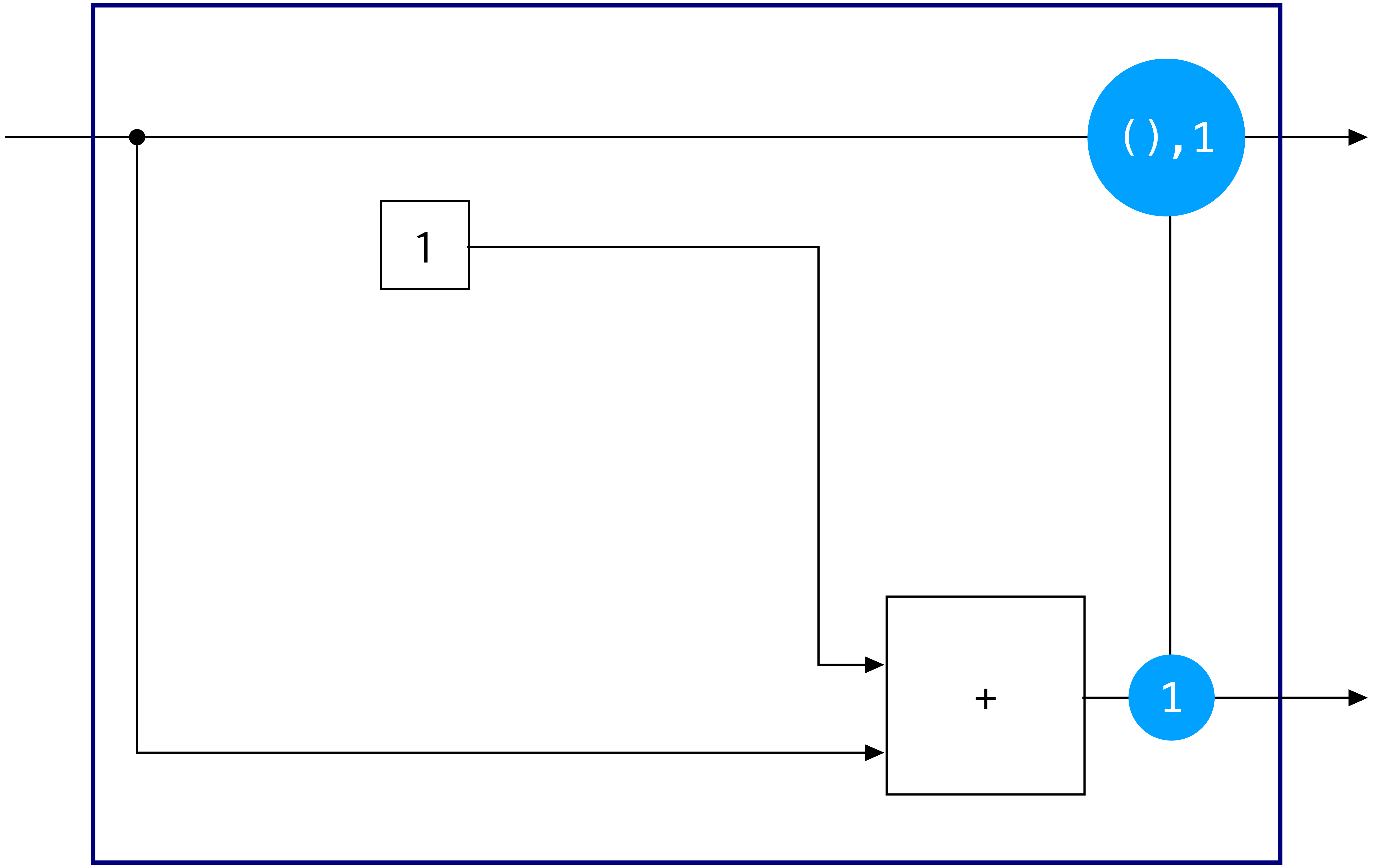
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

```
rec x = 1 + pre x
```

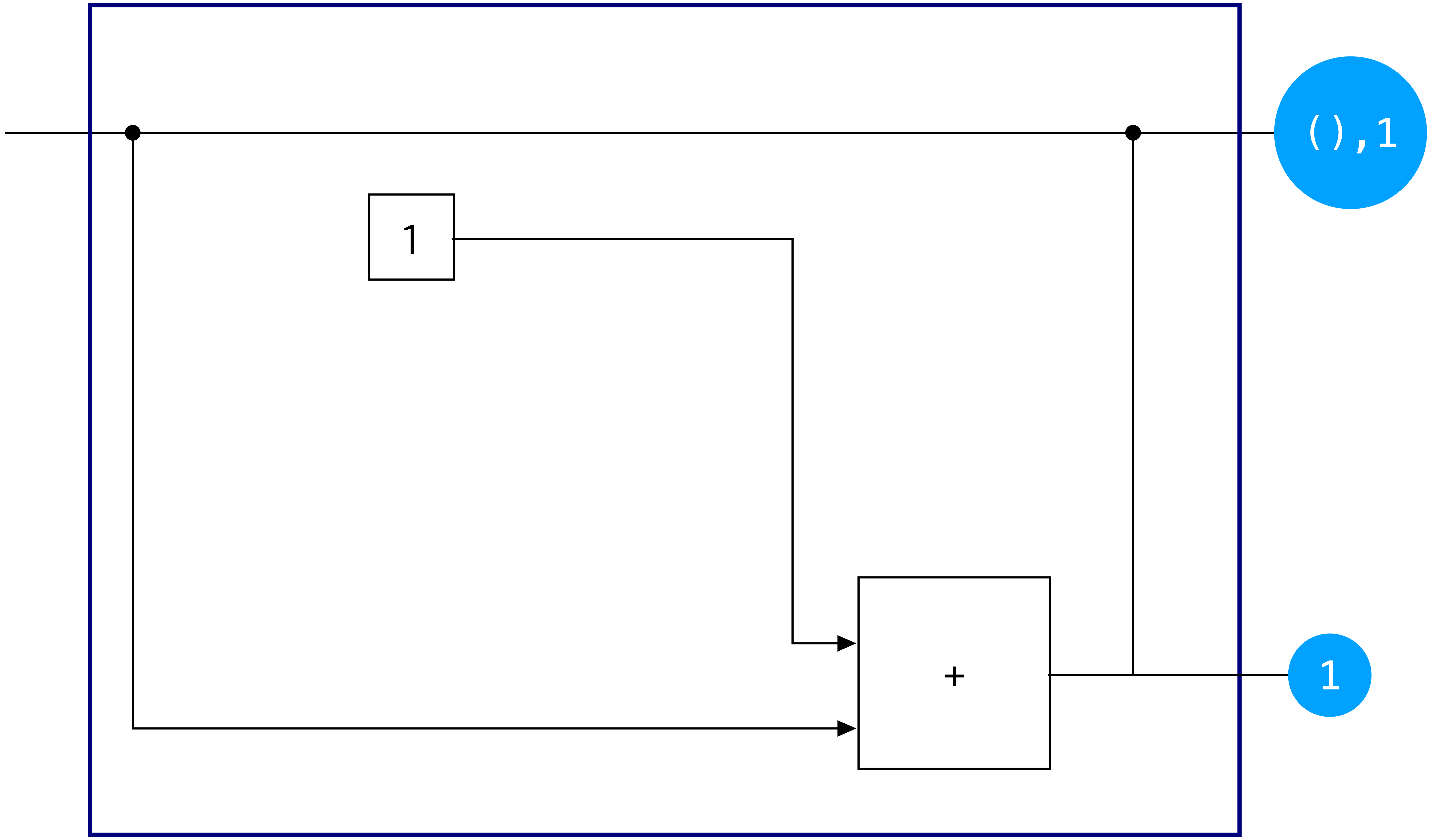
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

```
rec x = 1 + pre x
```

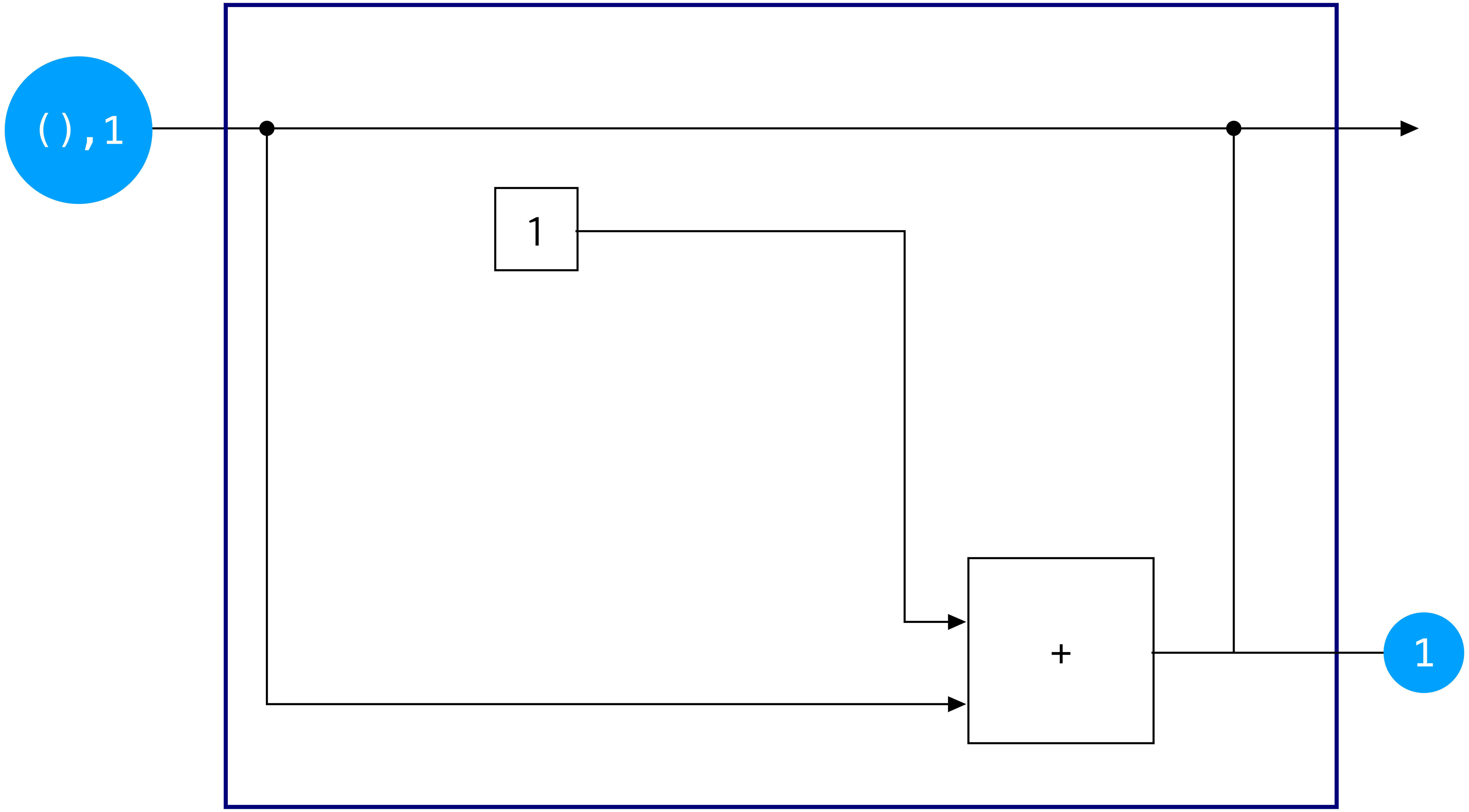
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

```
rec x = 1 + pre x
```

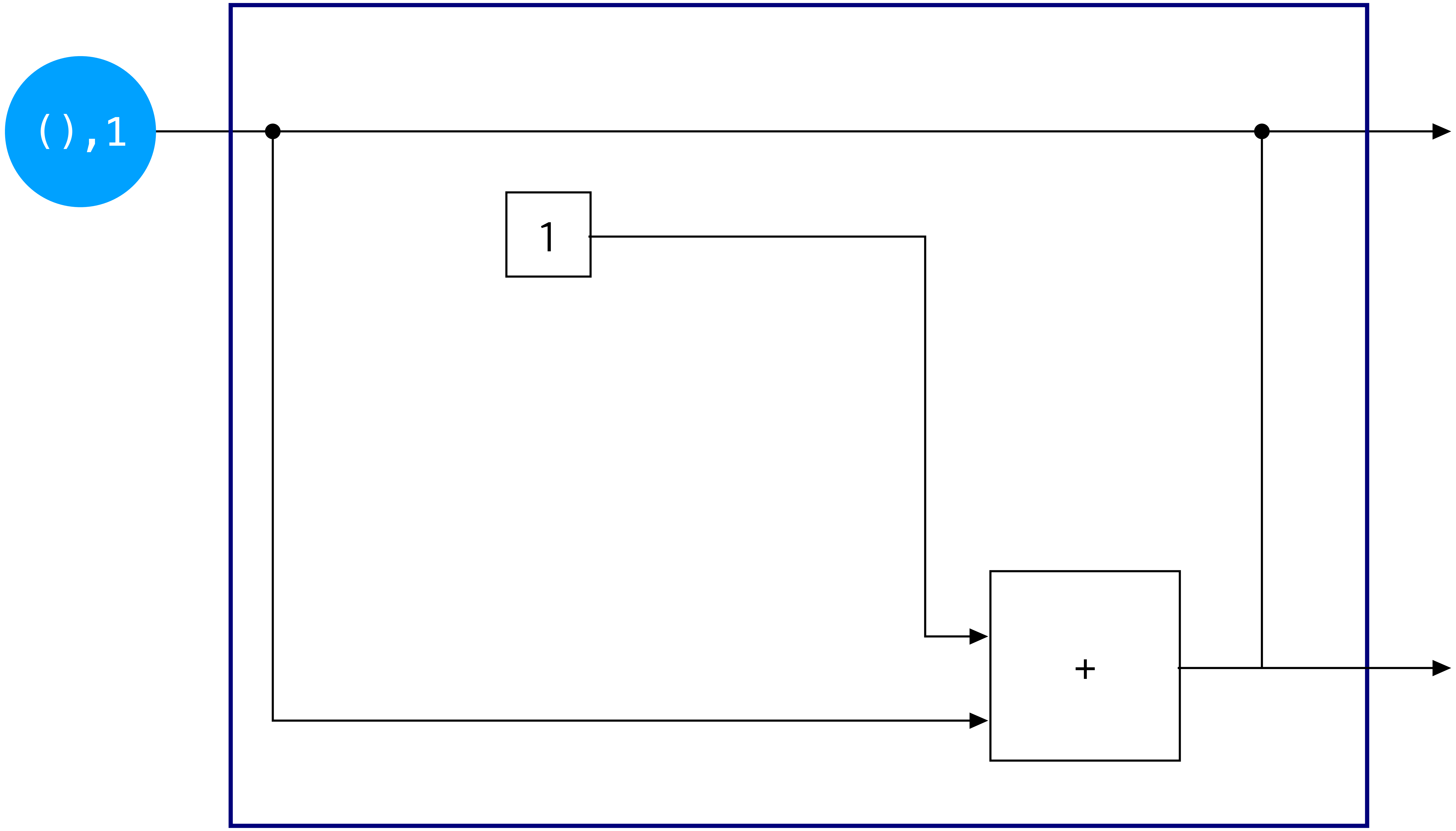
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

```
rec x = 1 + pre x
```

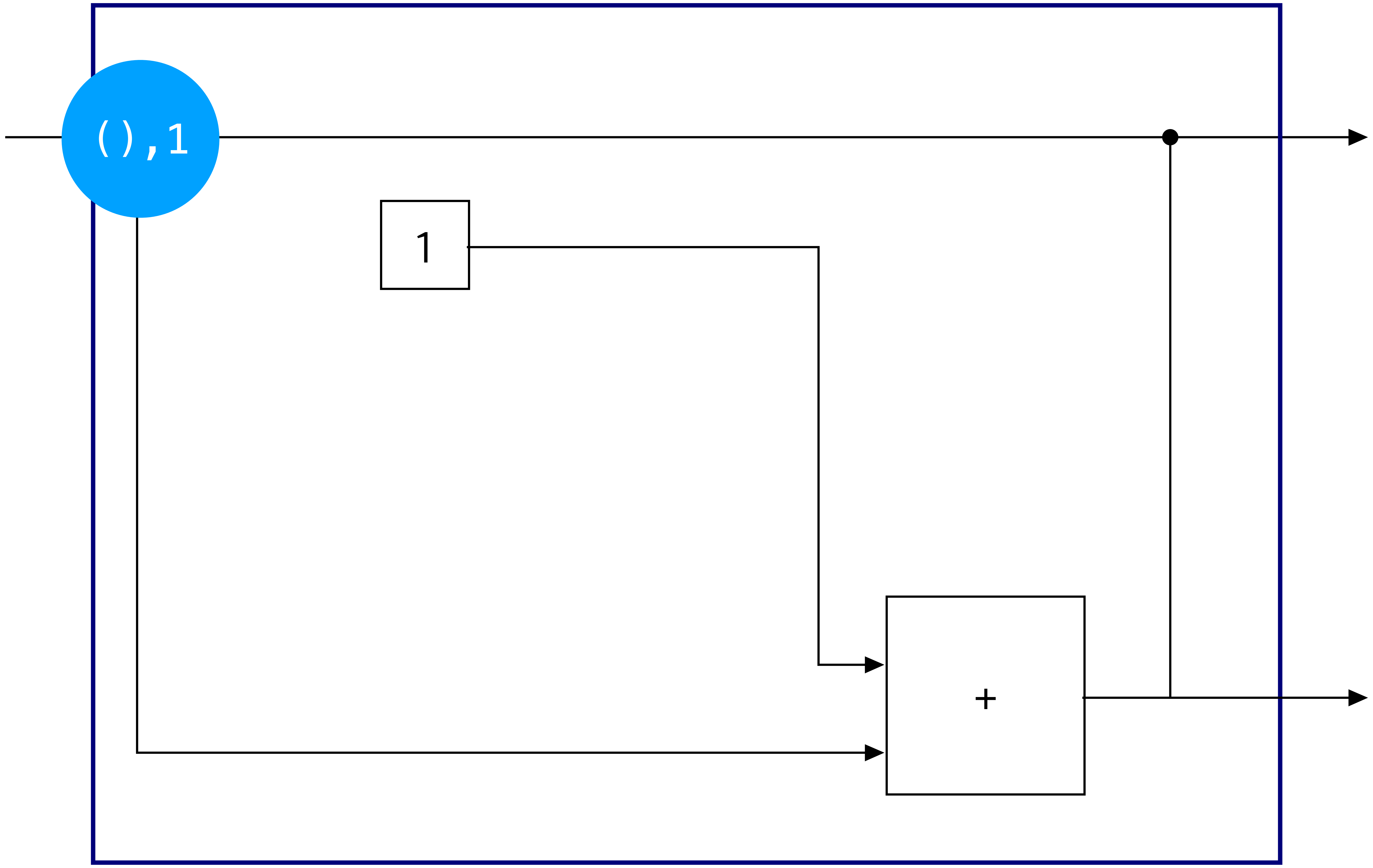
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

`rec x = 1 + pre x`

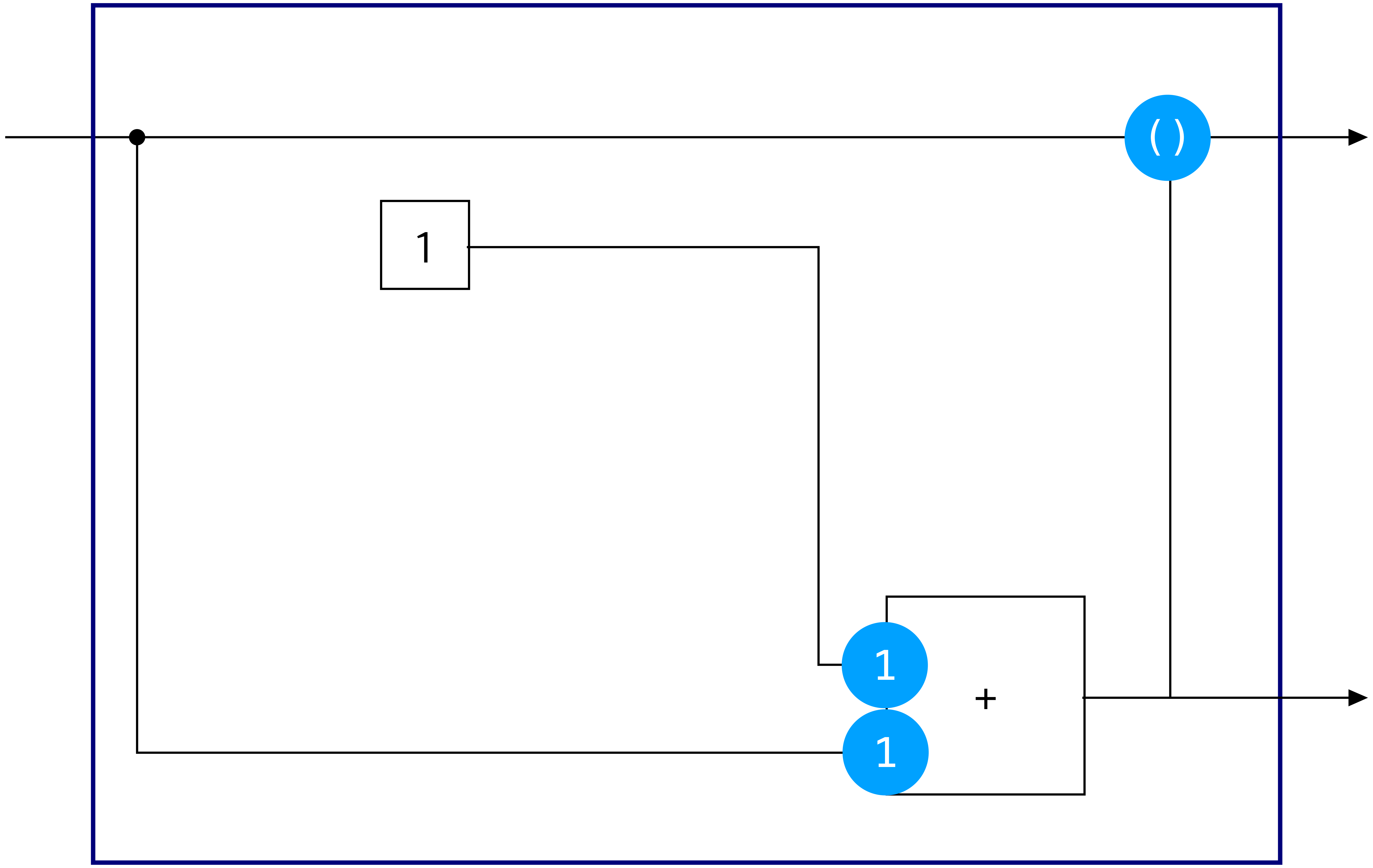
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

```
rec x = 1 + pre x
```

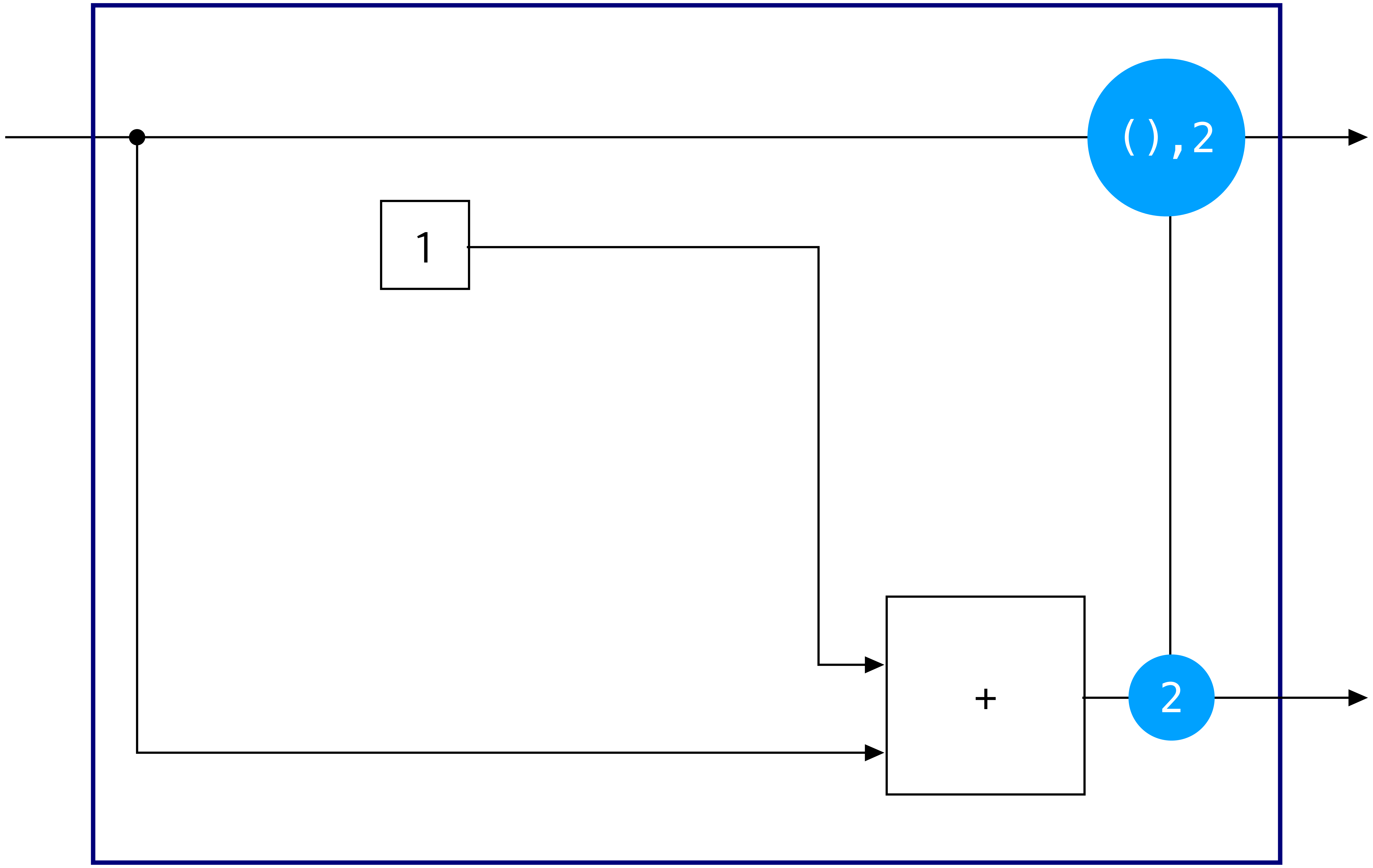
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

`rec x = 1 + pre x`

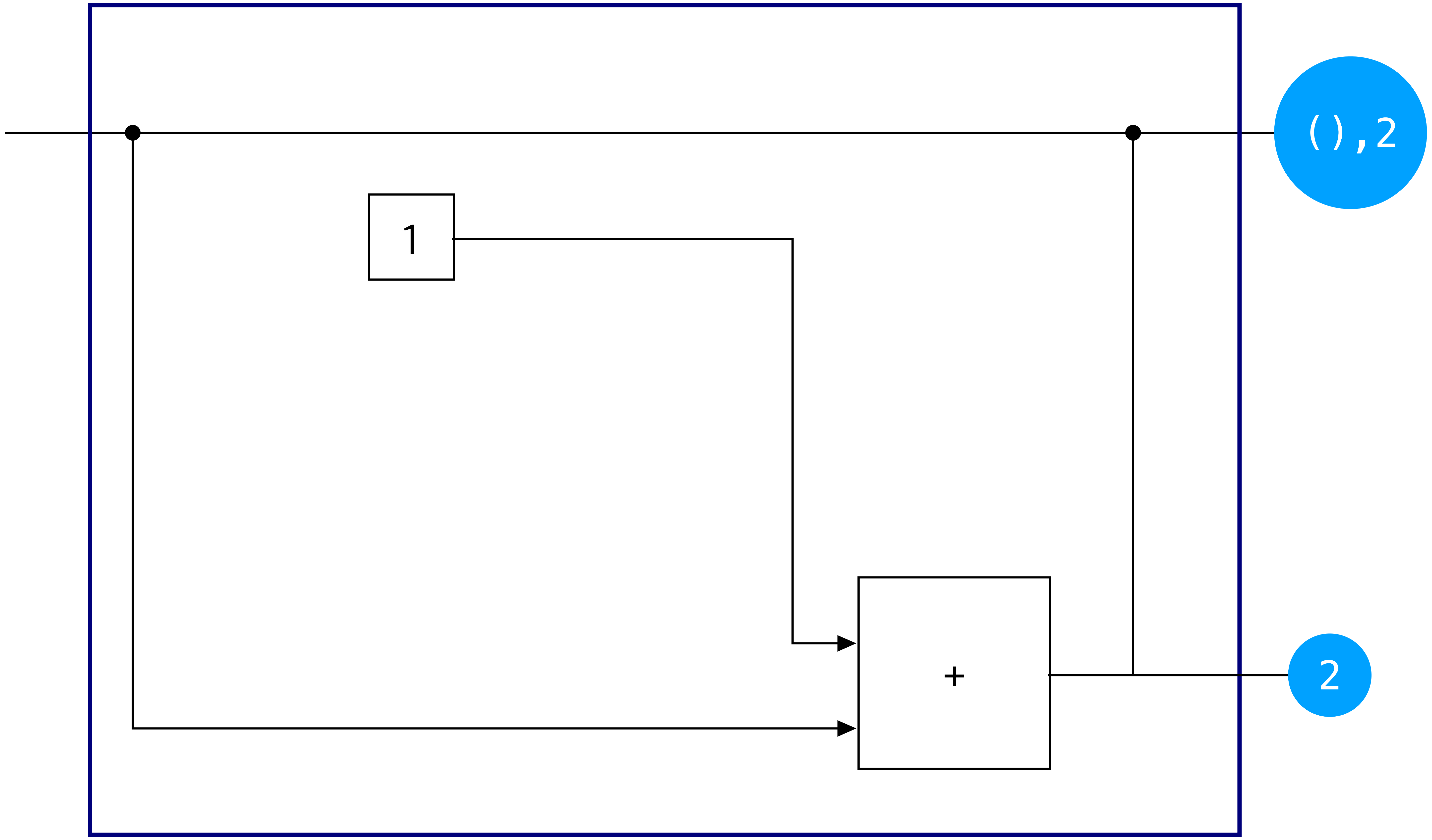
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

```
rec x = 1 + pre x
```

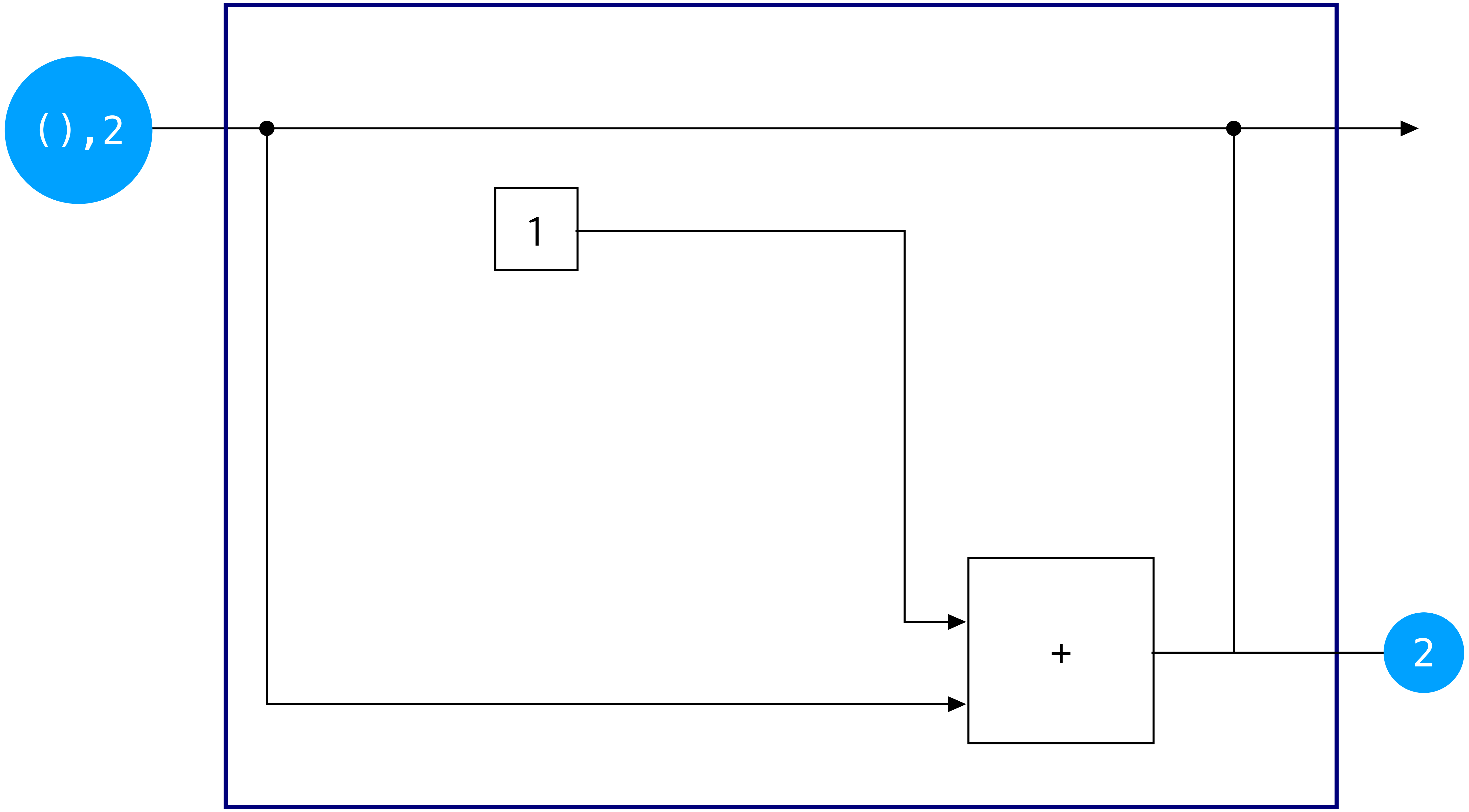
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic Semantics

```
rec x = 1 + pre x
```

- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`

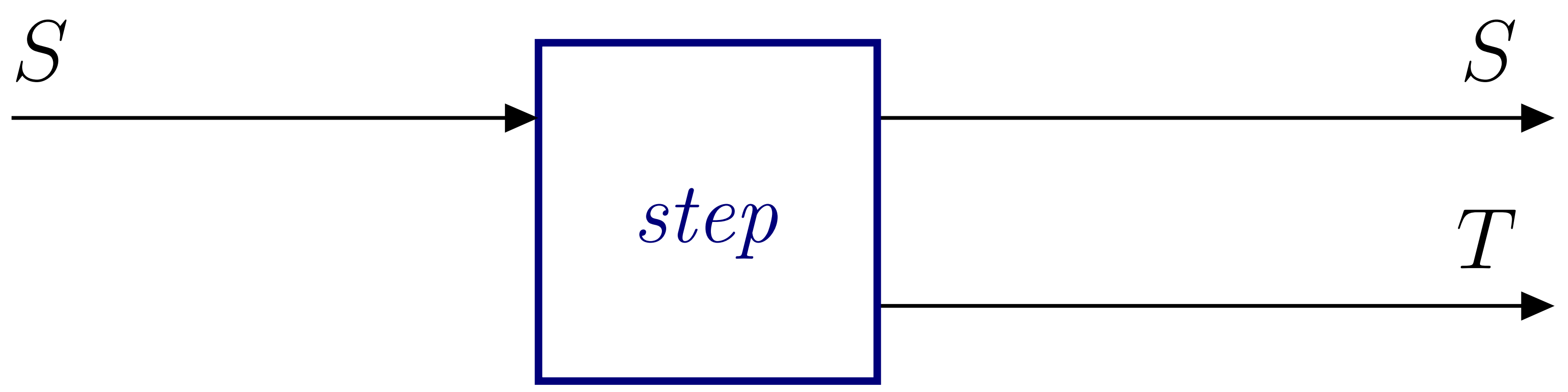


Deterministic vs. Probabilistic

Deterministic Streams

Transition function returns a pair of state and value

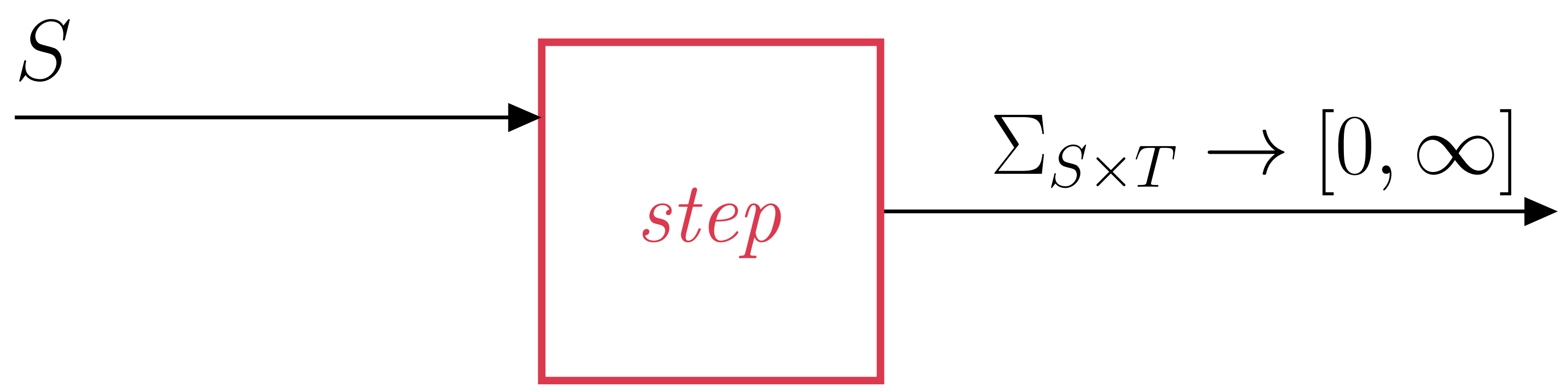
$CoStream(T, S) = S \times (S \to S \times T)$



Probabilistic Streams

Transition function returns a **measure** over (state, value)

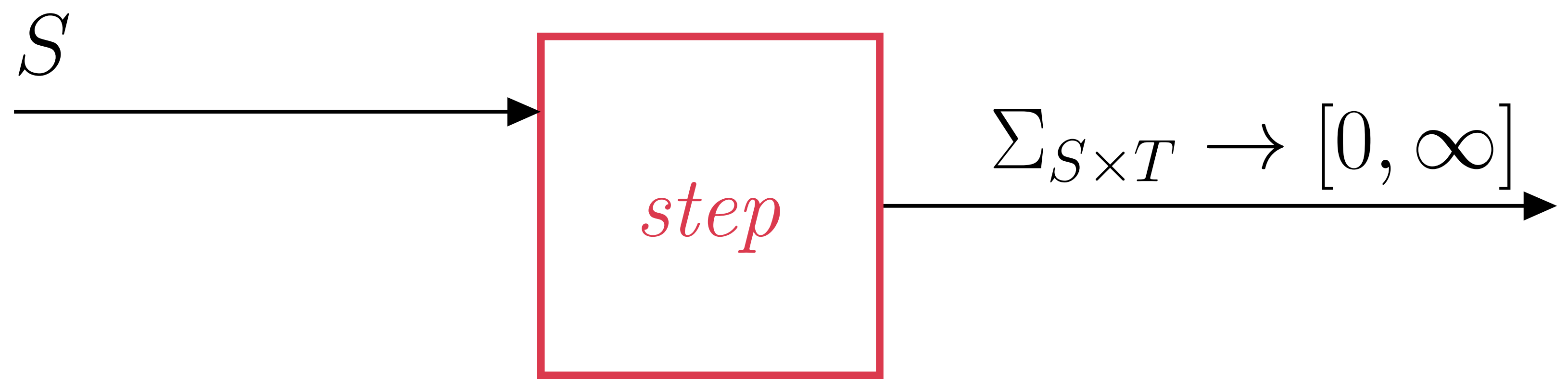
$CoPStream(T, S) = S \times (S \to \Sigma_{S \times T} \to [0,\infty])$



Probabilistic Semantics

Transition function returns a *measure*

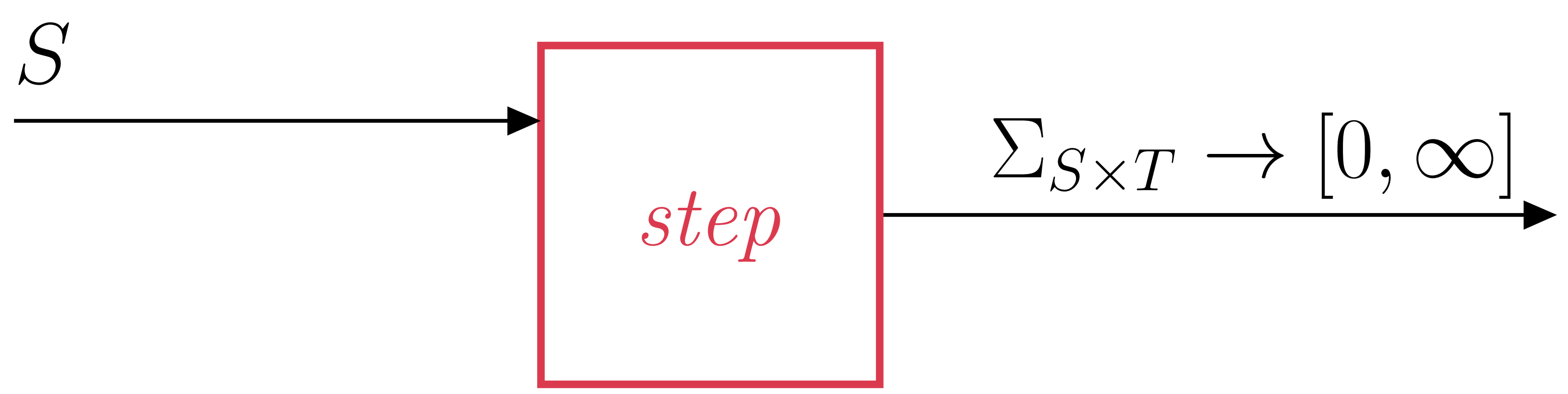
$CoPStream(T, S) = S \times (S \to \Sigma_{S \times T} \to [0,\infty])$



Probabilistic Semantics

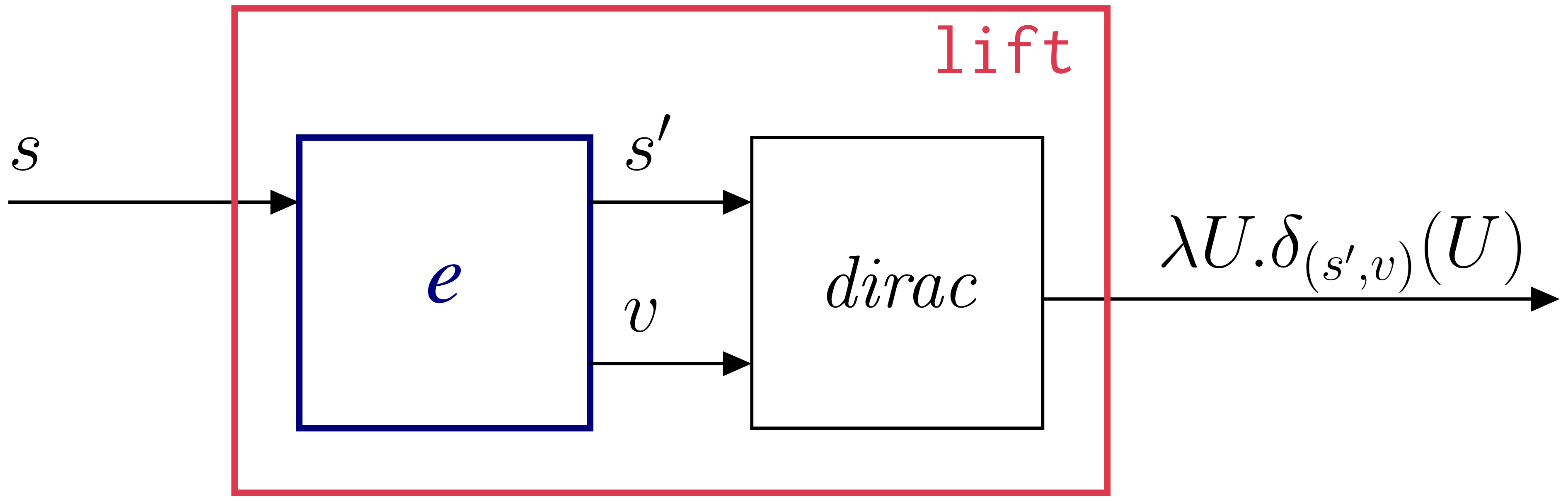
Transition function returns a *measure*

$CoPStream(T, S) = S \times (S \to \Sigma_{S \times T} \to [0,\infty])$



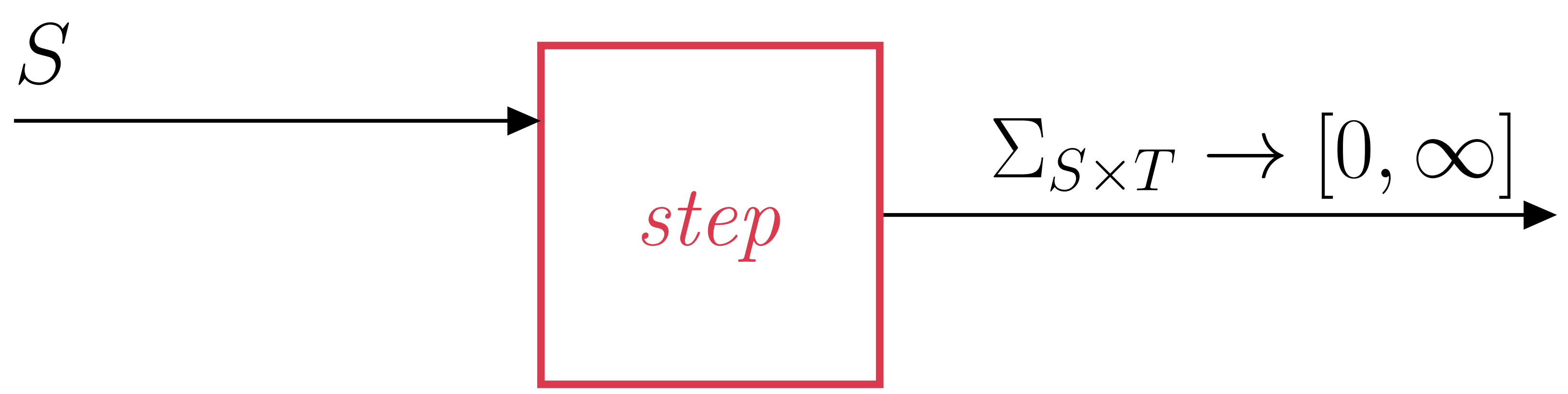
`lift` turns a deterministic expression into a probabilistic one

`lift`: $S \times (S \to S \times T) \to S \times (S \to \Sigma_{S \times T} \to [0,\infty])$

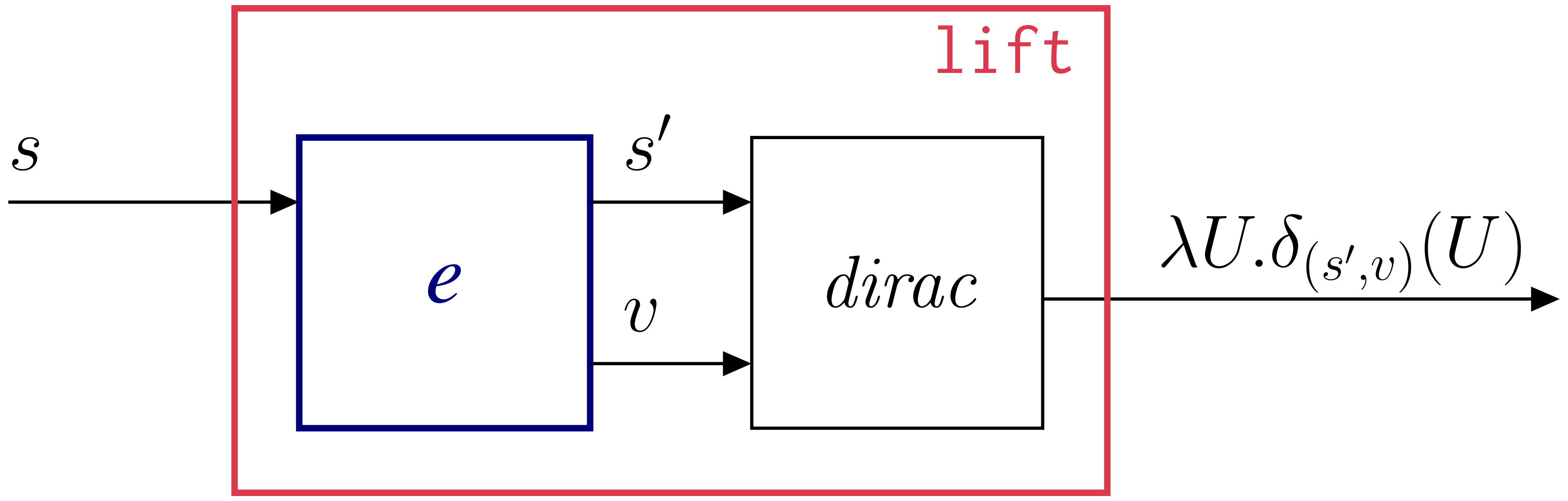


Probabilistic Semantics

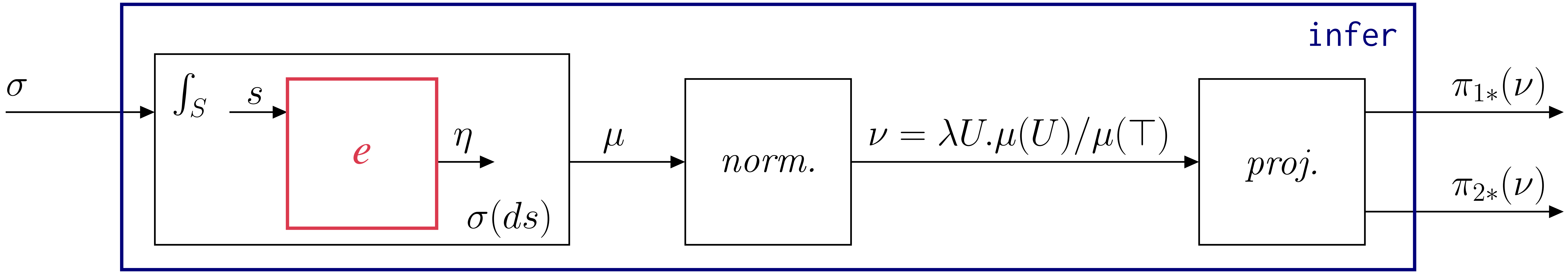
Transition function returns a *measure*
$CoPStream(T, S) = S \times (S \to \Sigma_{S \times T} \to [0,\infty])$



`lift` turns a deterministic expression into a probabilistic one
`lift`: $S \times (S \to S \times T) \to S \times (S \to \Sigma_{S \times T} \to [0,\infty])$



`infer` turns probabilistic expressions to a pair of distributions
`infer`: $S \times (S \to \Sigma_{S \times T} \to [0,\infty]) \to S$ `dist` $\times (S$ `dist` $\to S$ `dist` $\times\, T$ `dist`$)$



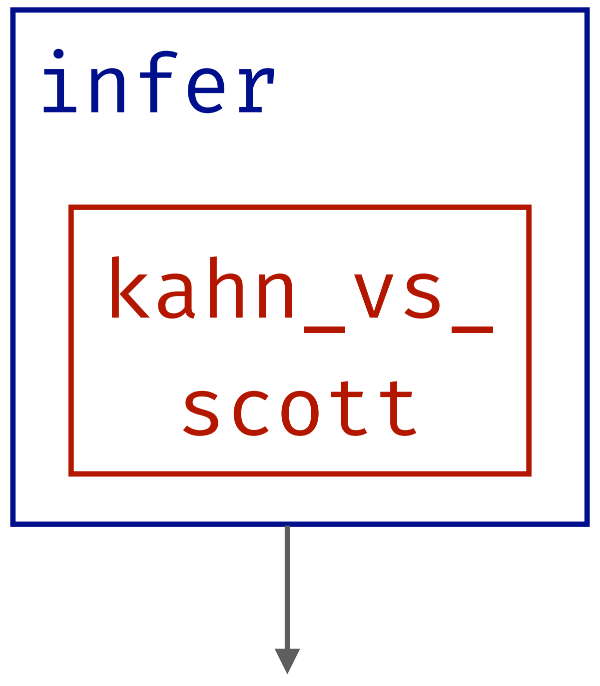
Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```

Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```

Kahn



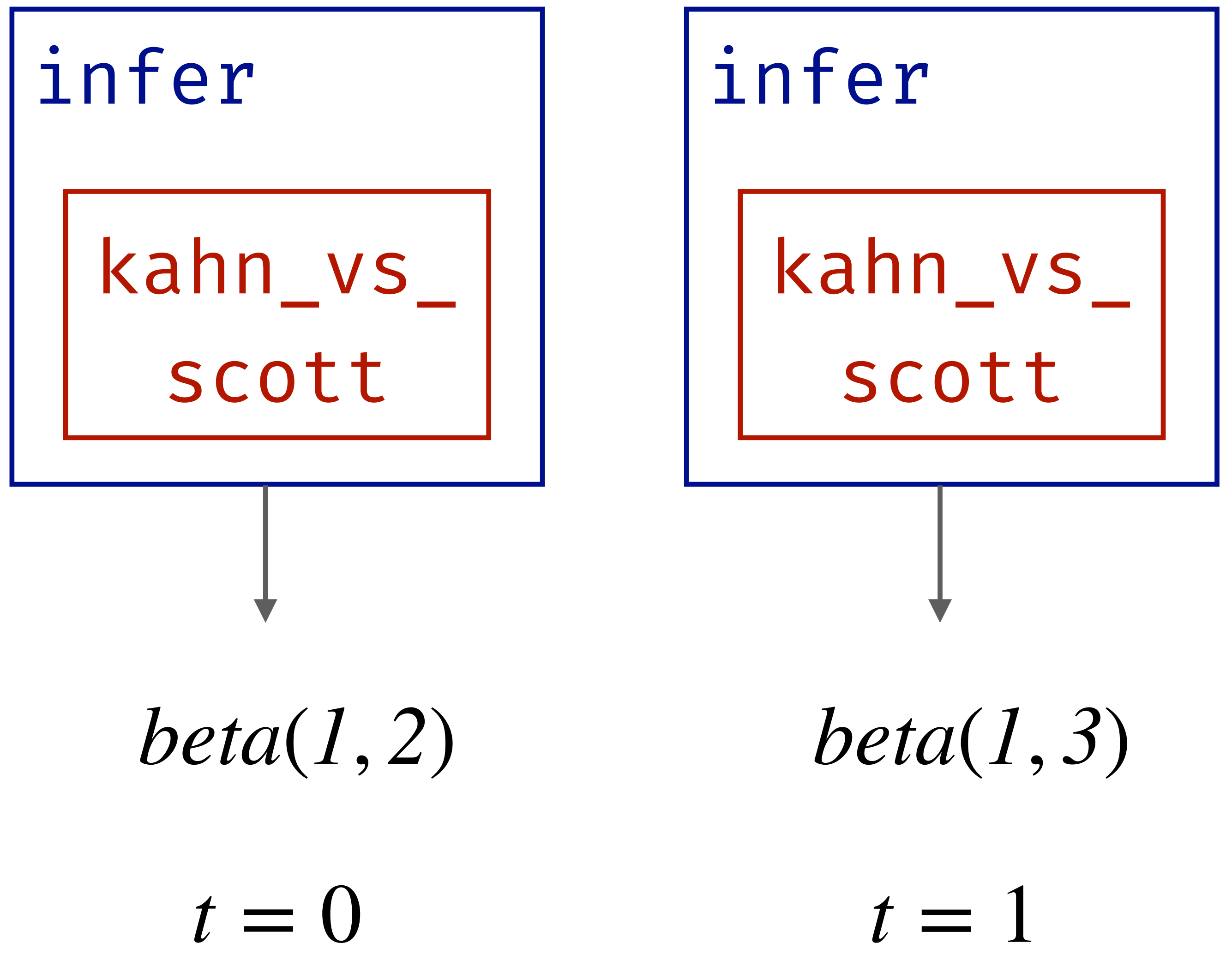
$$beta(1, 2)$$

$$t = 0$$

Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```

Kahn



Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```

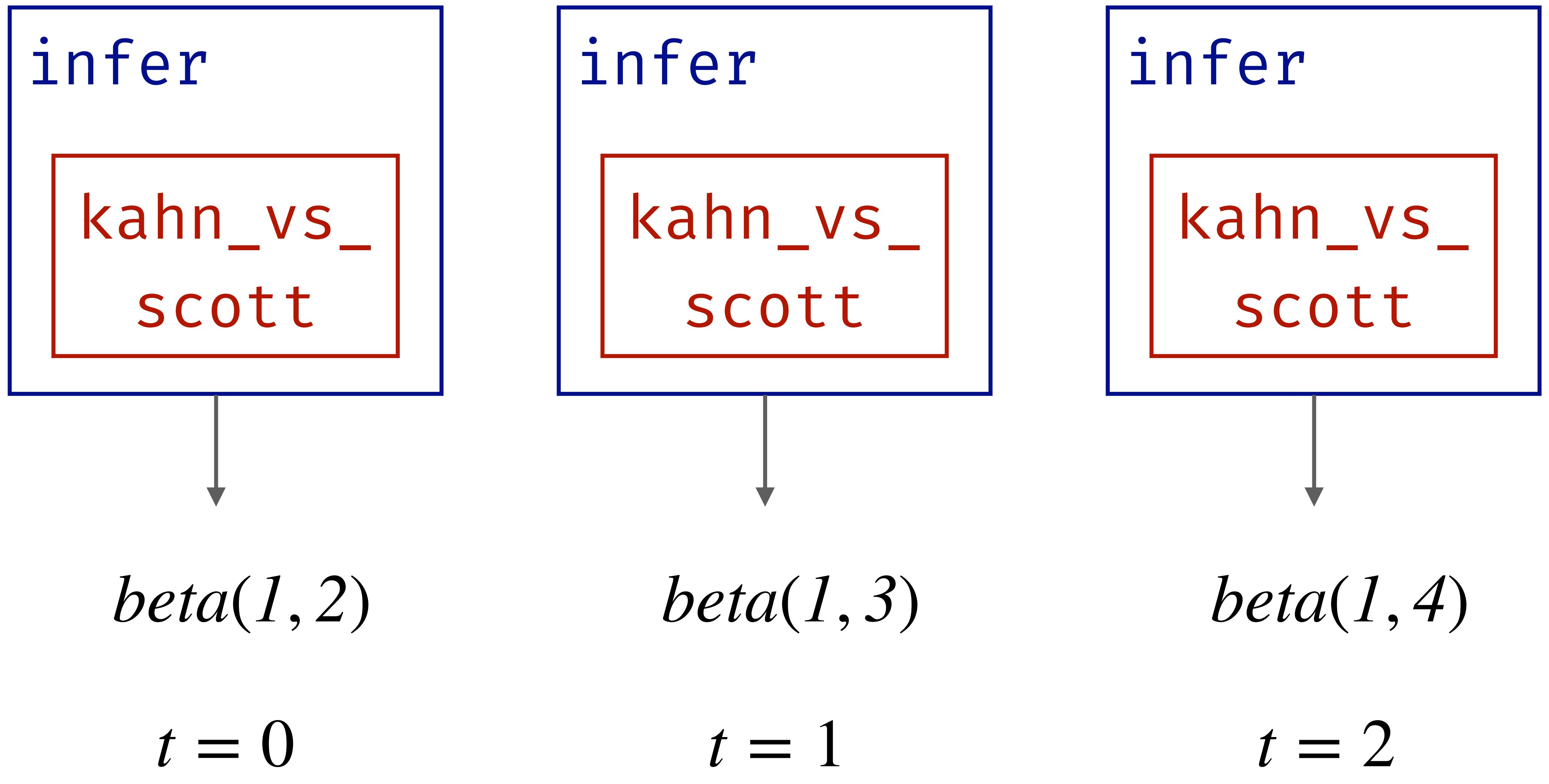
Kahn

infer kahn_vs_scott	infer kahn_vs_scott	infer kahn_vs_scott
$beta(1,2)$	$beta(1,3)$	$beta(1,4)$
$t = 0$	$t = 1$	$t = 2$

Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```

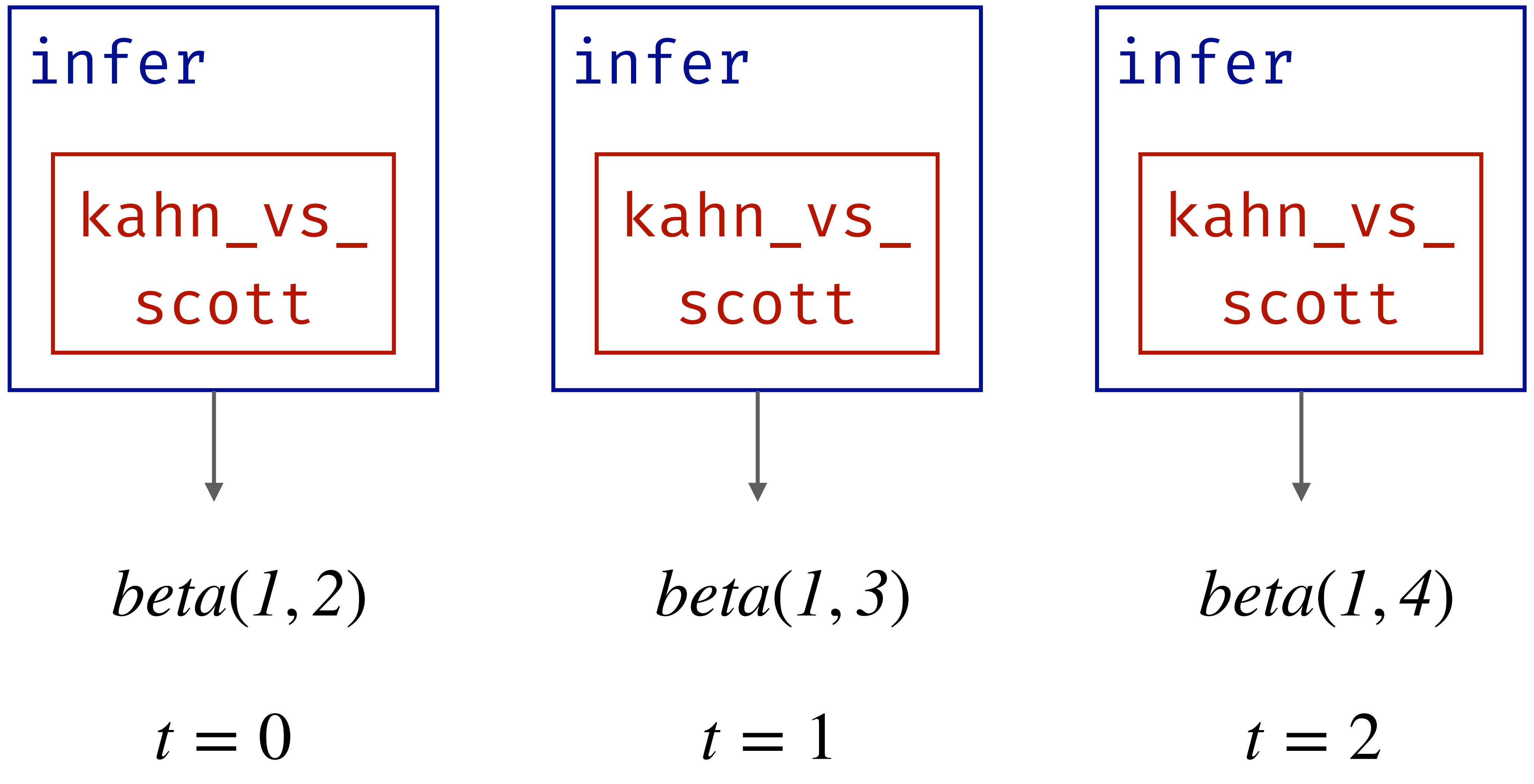
Kahn



Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```

Kahn



Scott

Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```

Kahn

infer kahn_vs_scott → $beta(1,2)$ — $t = 0$

infer kahn_vs_scott → $beta(1,3)$ — $t = 1$

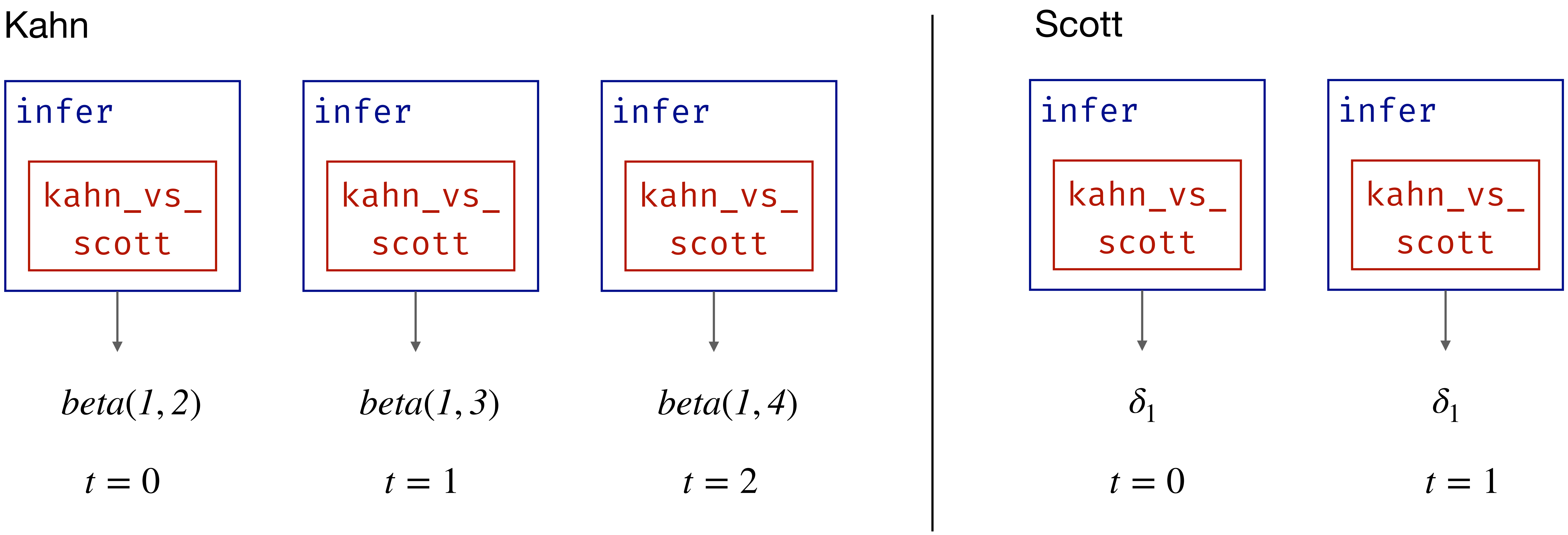
infer kahn_vs_scott → $beta(1,4)$ — $t = 2$

Scott

infer kahn_vs_scott → δ_1 — $t = 0$

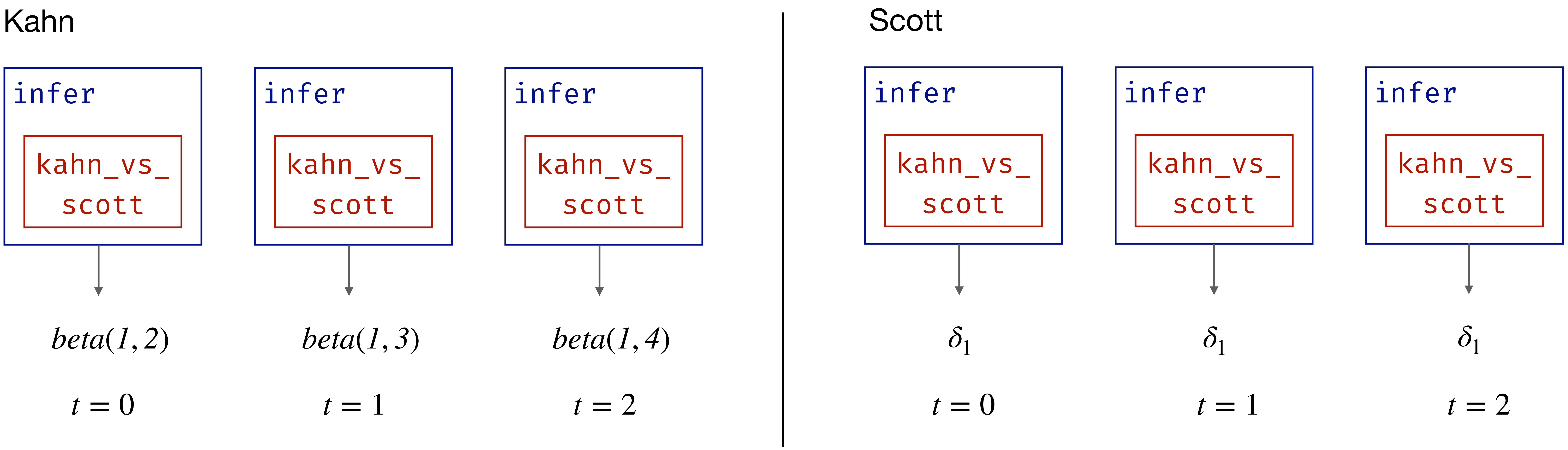
Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```



Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```

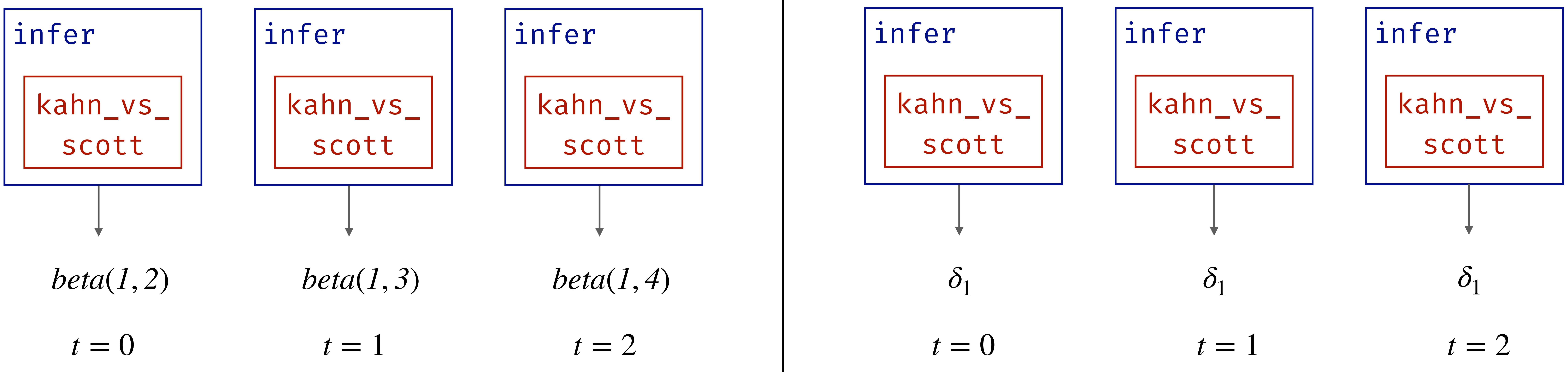


Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```

Kahn

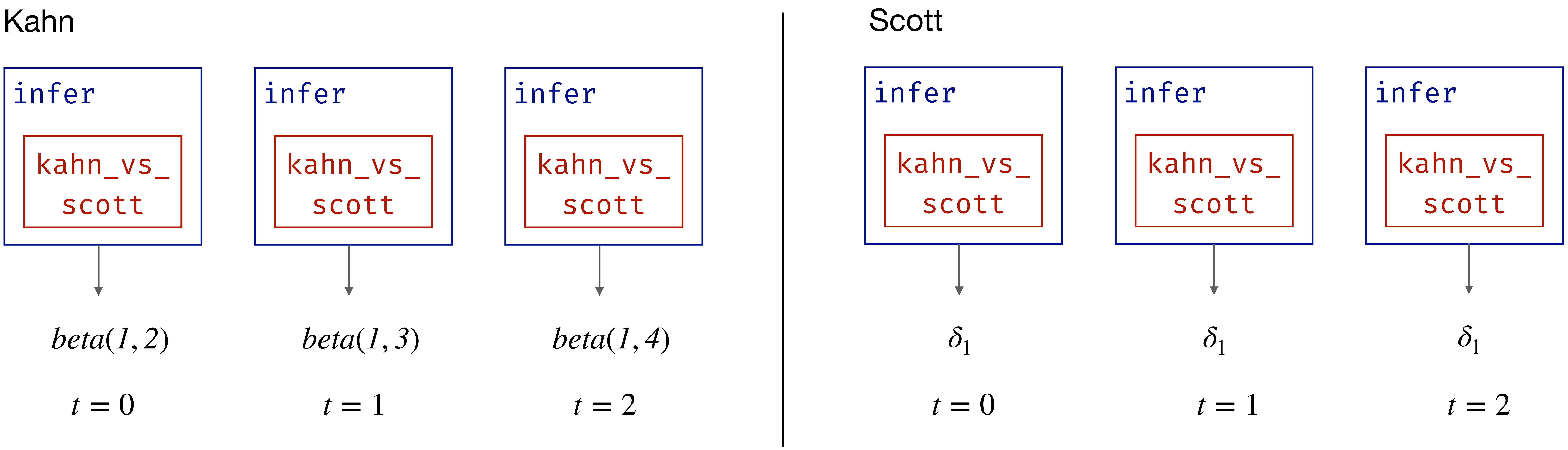
Scott



Moving constants...

Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```



Moving constants...

Depends on the future

Kahn vs. Scott Semantics

```
proba kahn_vs_scott () = z where
  rec init z = sample(uniform(0, 1))
  and () = observe(bernoulli(z), true)
```

could be an input

Kahn

infer kahn_vs_scott	infer kahn_vs_scott	infer kahn_vs_scott
$beta(1, 2)$	$beta(1, 3)$	$beta(1, 4)$
$t = 0$	$t = 1$	$t = 2$

Moving constants...

Scott

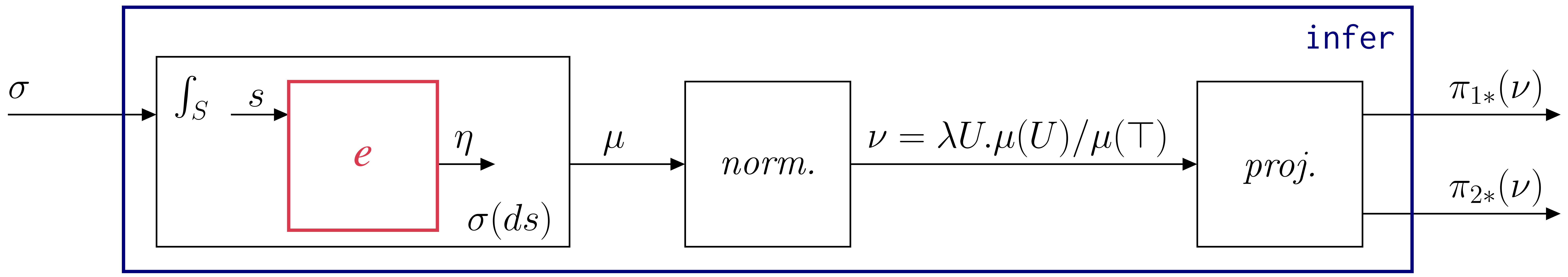
infer kahn_vs_scott	infer kahn_vs_scott	infer kahn_vs_scott
δ_1	δ_1	δ_1
$t = 0$	$t = 1$	$t = 2$

Depends on the future

Streaming Inference

Reactive Probabilistic Programming

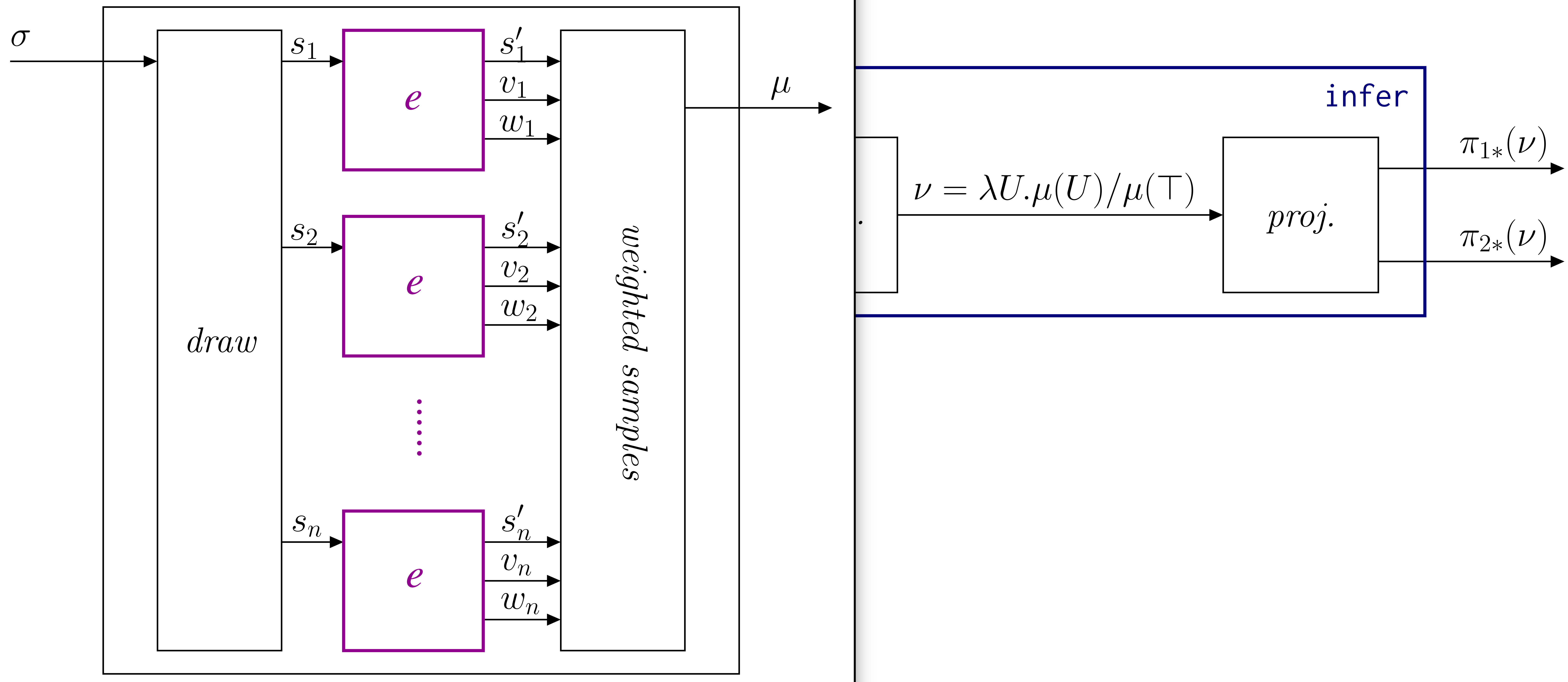
Particle Filtering



Particle Filtering

Intractable integrals

Approximation: weighted sum from multiple particles



Delayed Sampling

Simple Particles Filters can be impractical

- Require lot of computing power
- Poor approximation

Exact inference is often possible

Semi-Symbolic inference

- 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

Murray et al. 2018

Delayed Sampling

Simple Particles Filters can be impractical

- Require lot of computing power
- Poor approximation

Exact inference is often possible

Semi-Symbolic inference

- 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

$$x \sim \mathcal{N}(\mu_0, \sigma_0)$$
$$y \sim \mathcal{N}(x, \sigma)$$

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

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

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

Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```


Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) -> gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x   = sample (gaussian (0, 10)) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
```

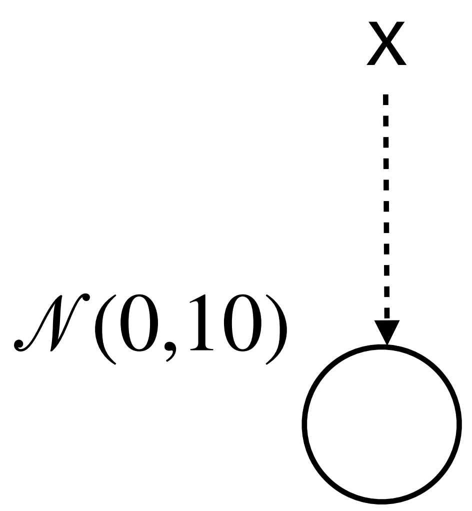

Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10)) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
```



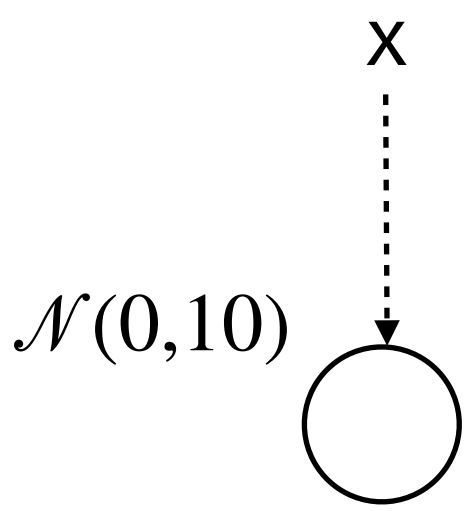
Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```



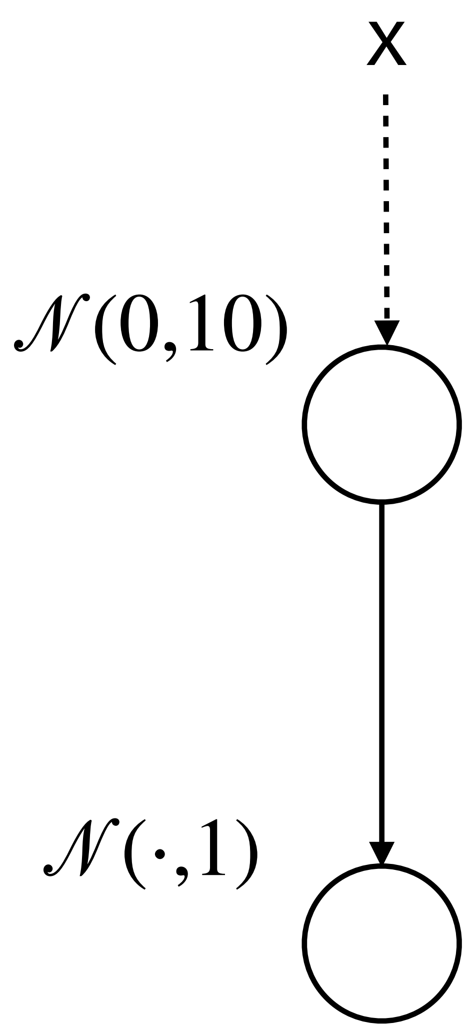
Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) -> gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```



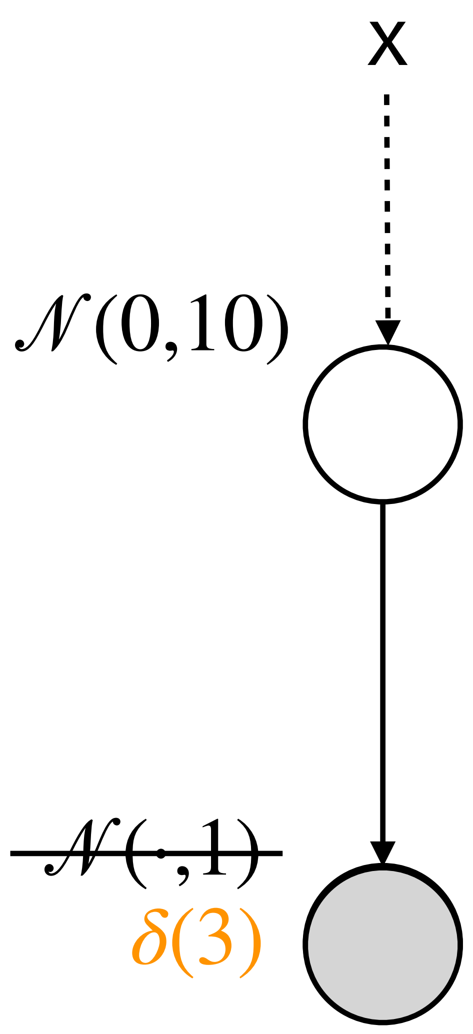
Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) -> gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```



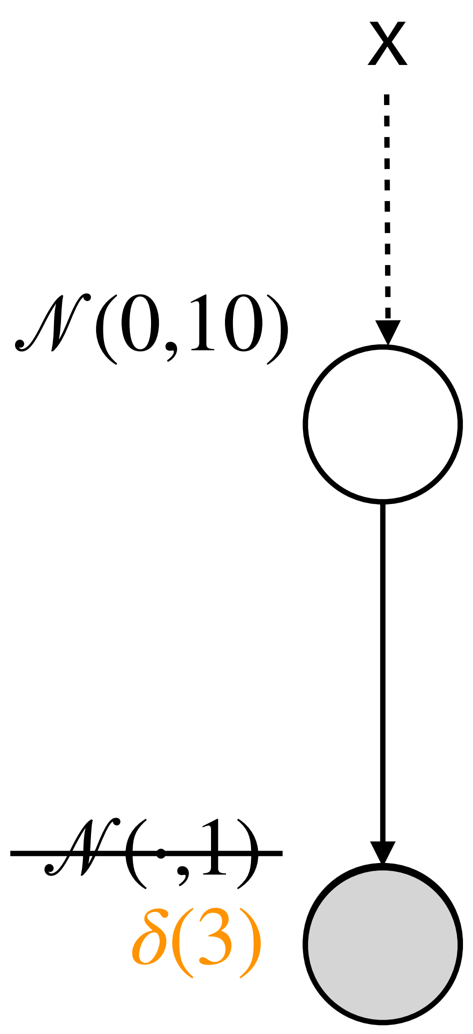
Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) -> gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```



Example: 2 Gaussians

$$x \sim \mathcal{N}(\mu_0, \sigma_0)$$
$$y \sim \mathcal{N}(x, \sigma)$$

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

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

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

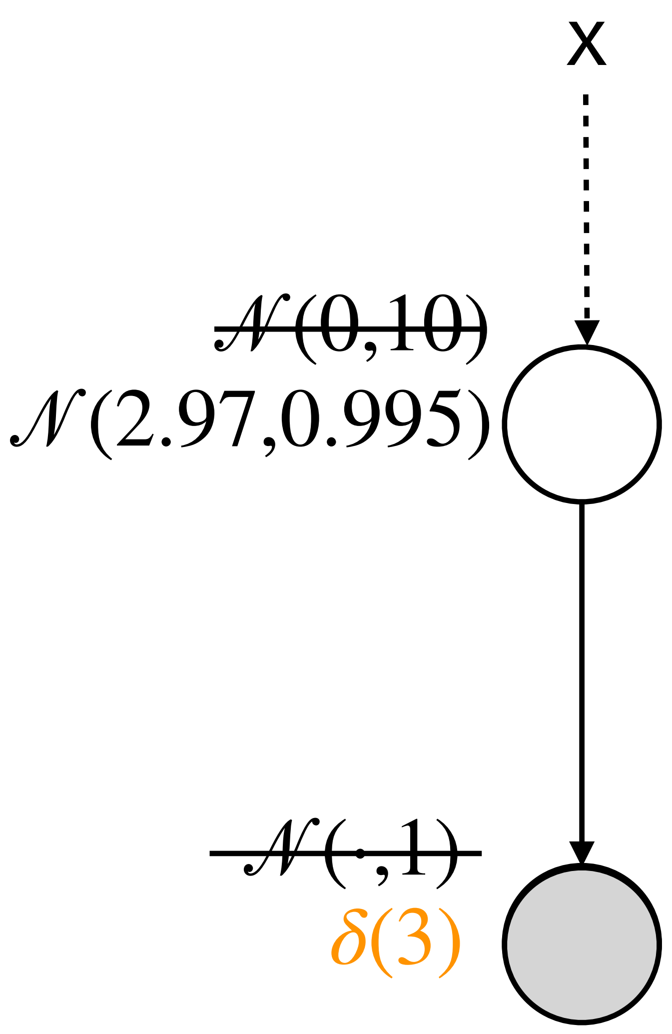
Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) -> gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```



Example: 2 Gaussians

$$x \sim \mathcal{N}(\mu_0, \sigma_0)$$
$$y \sim \mathcal{N}(x, \sigma)$$

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

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

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

Murray et al. 2018

Delayed Sampling

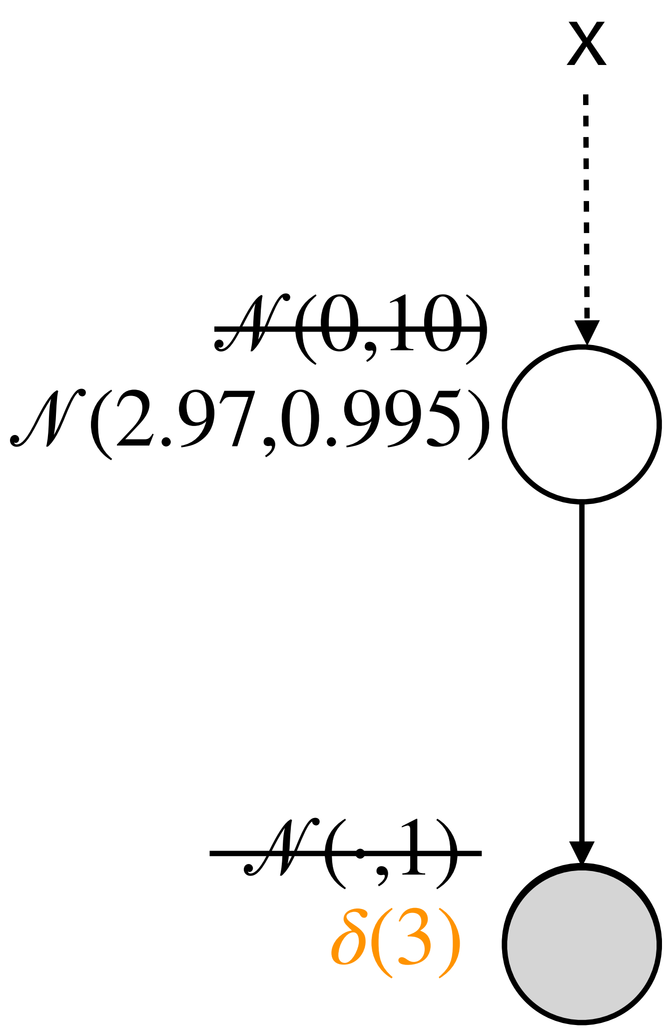
```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

```
sample (gaussian (pre x, 1))
```



Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

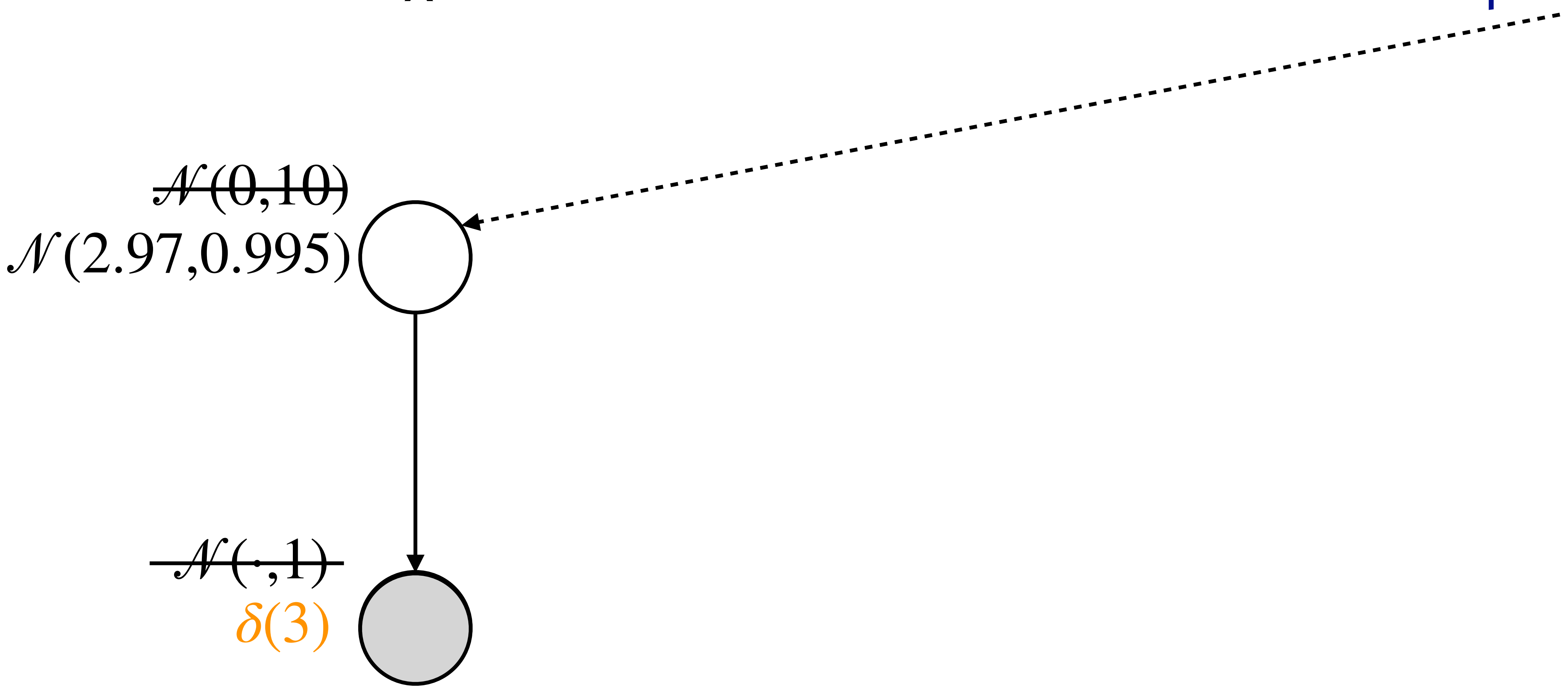
$t = 1$

```
sample (gaussian (pre x, 1))
```

x

pre x

x



Murray et al. 2018

Delayed Sampling

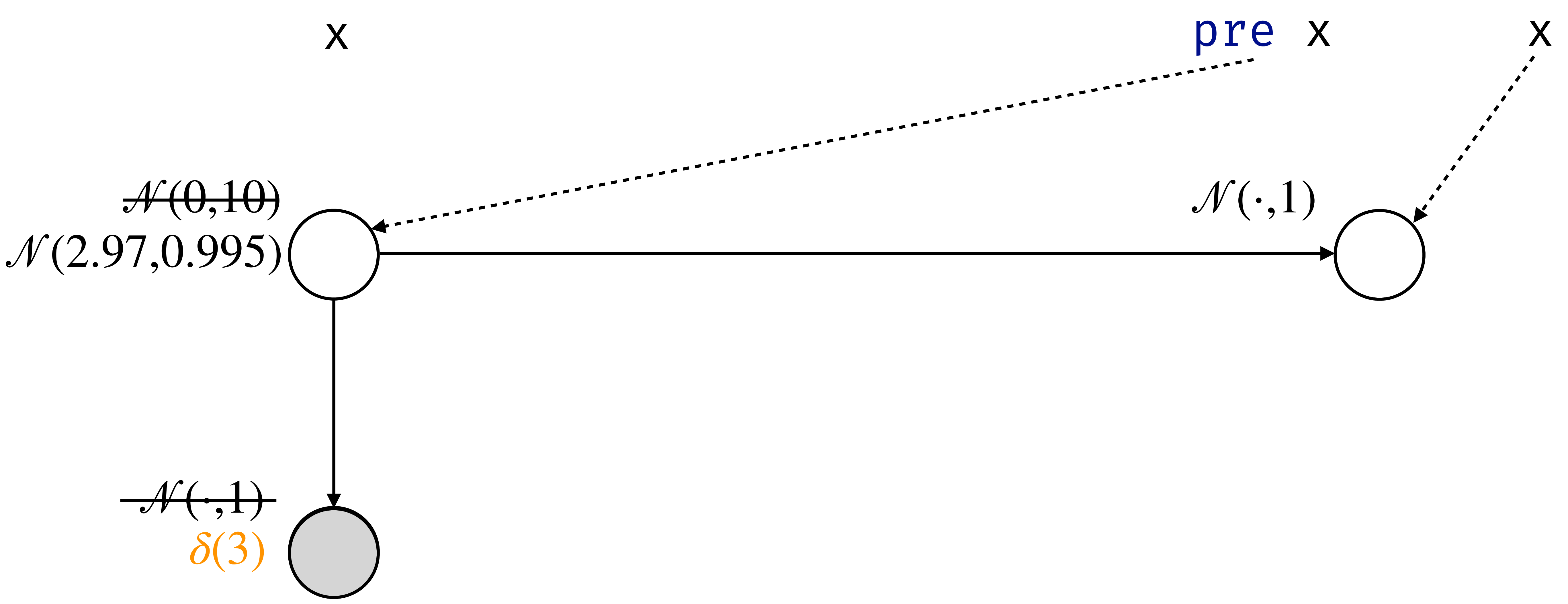
```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

```
sample (gaussian (pre x, 1))
```



Murray et al. 2018

Delayed Sampling

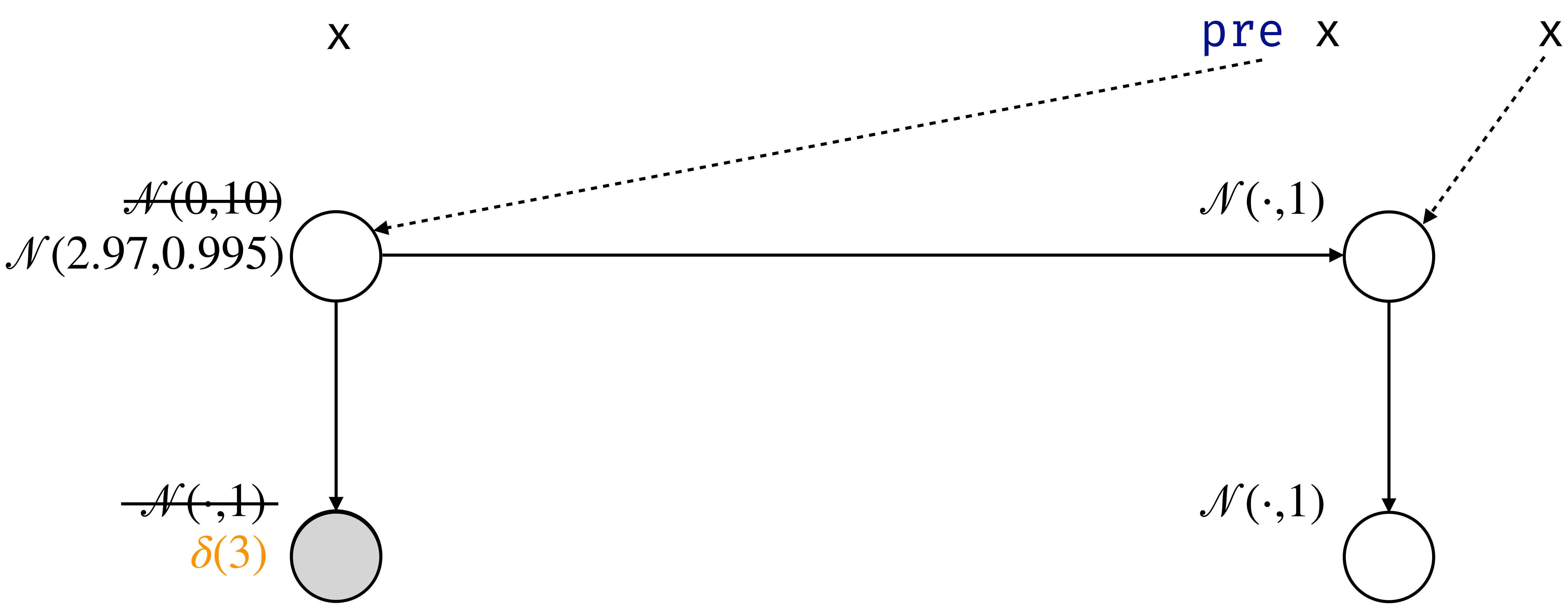
```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) -> gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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



Murray et al. 2018

Delayed Sampling

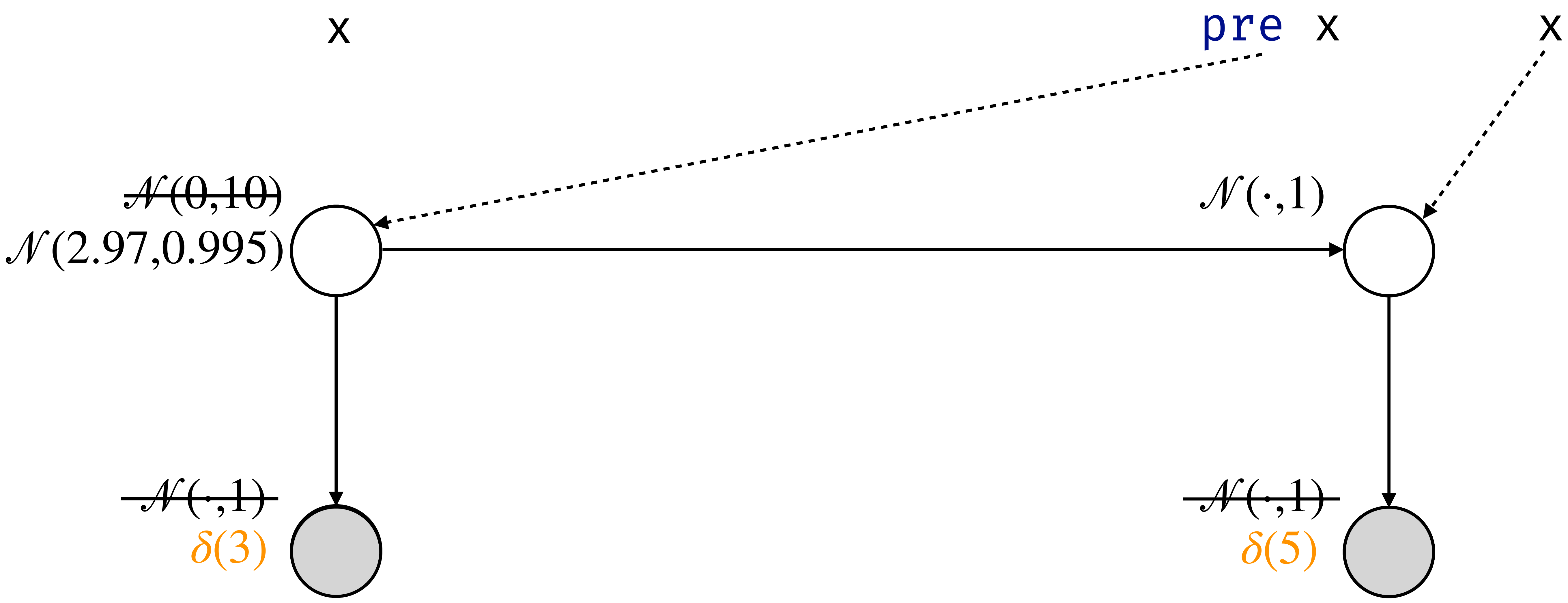
```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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



Murray et al. 2018

Delayed Sampling

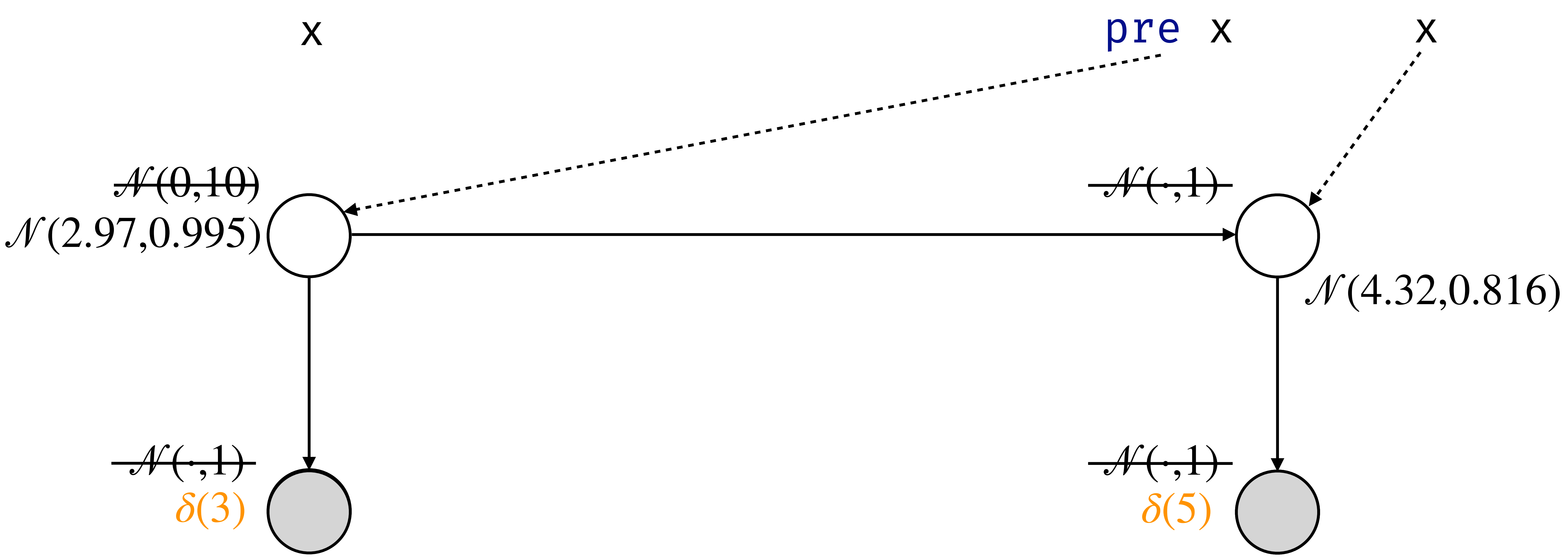
```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) -> gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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



Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

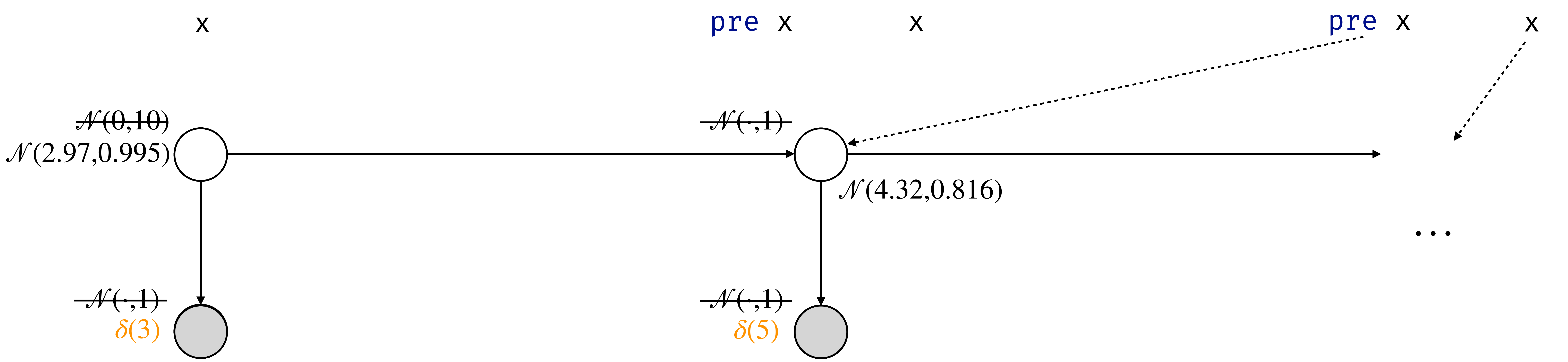
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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

$t = 2$

```
sample (gaussian (pre x, 1))
observe (gaussian (x, 1), …)
```



Murray et al. 2018

Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

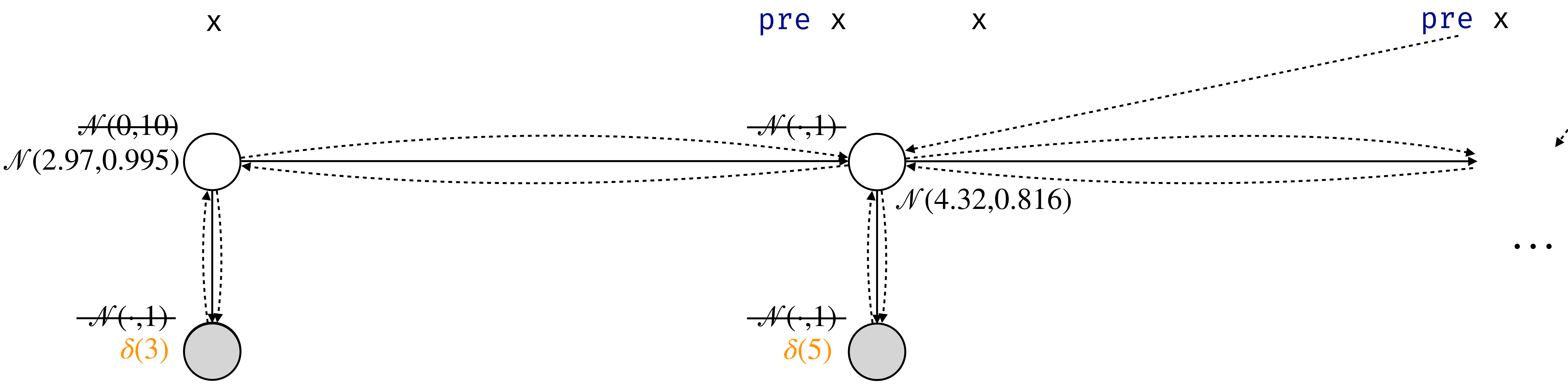
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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

$t = 2$

```
sample (gaussian (pre x, 1))
observe (gaussian (x, 1), …)
```



Murray et al. 2018

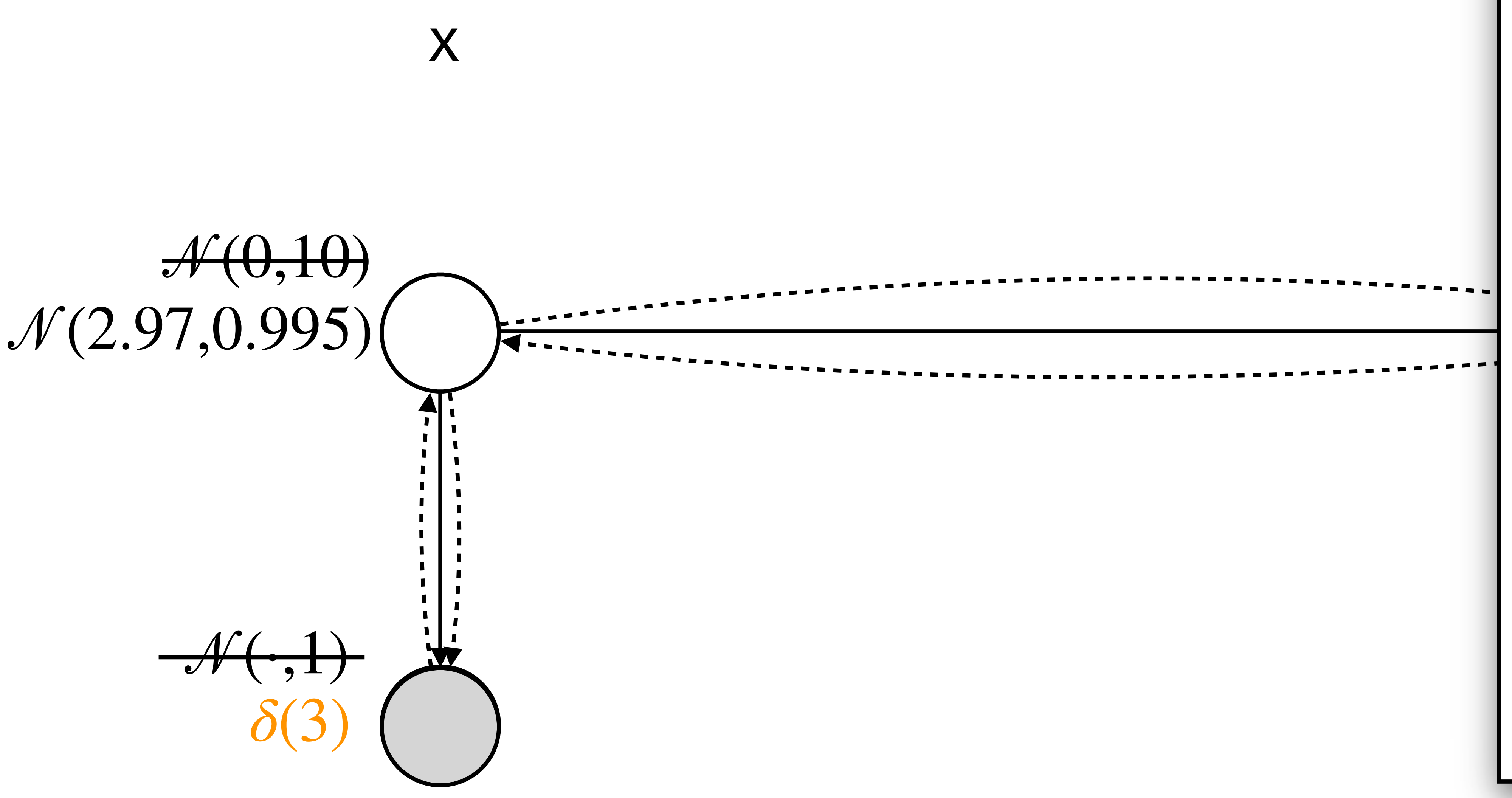
Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) -> gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

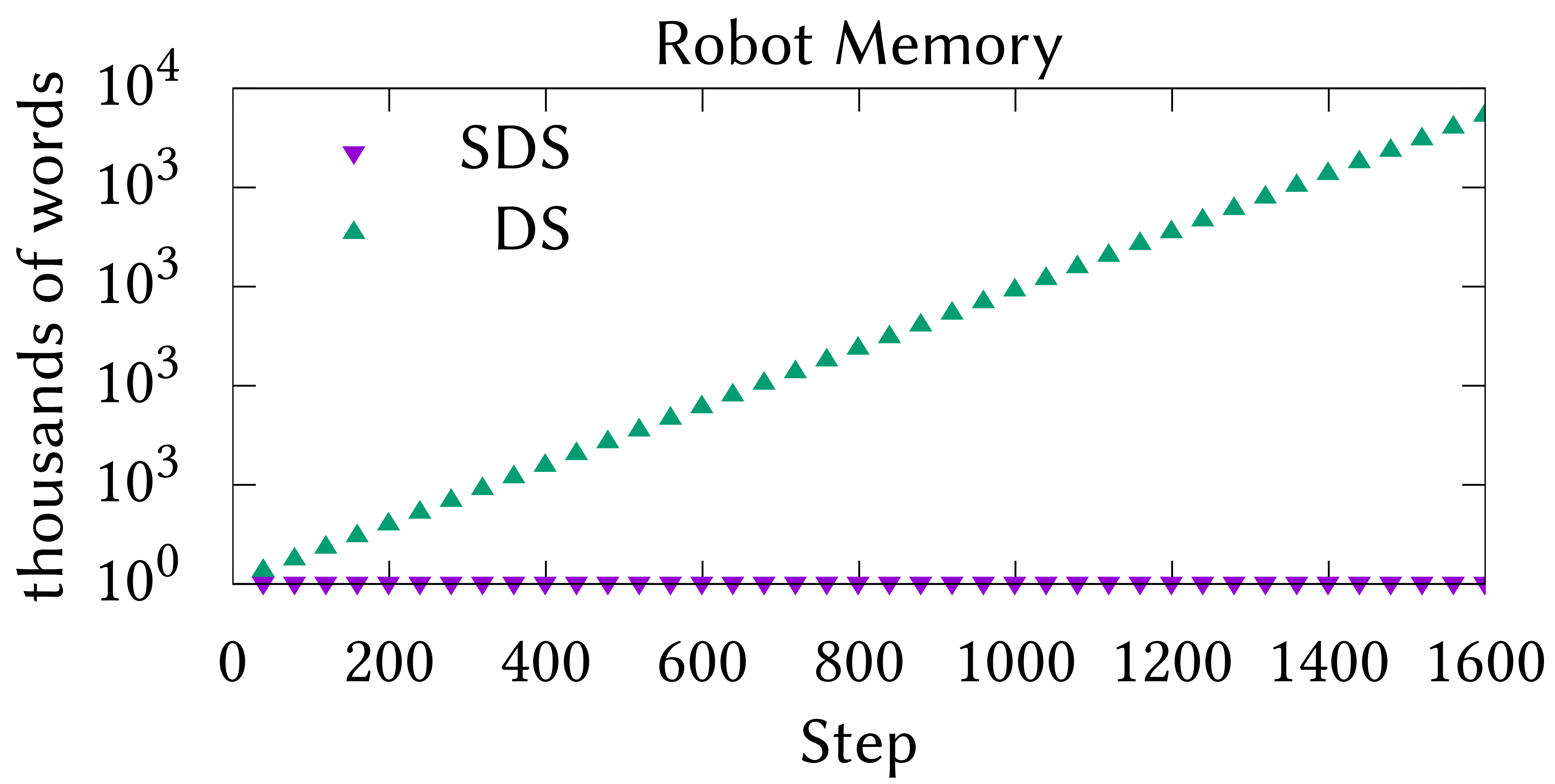
$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

```
sample                                                   , 1))
observe                                                  ), …)
```



Unbounded resources



Murray et al. 2018

Streaming Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

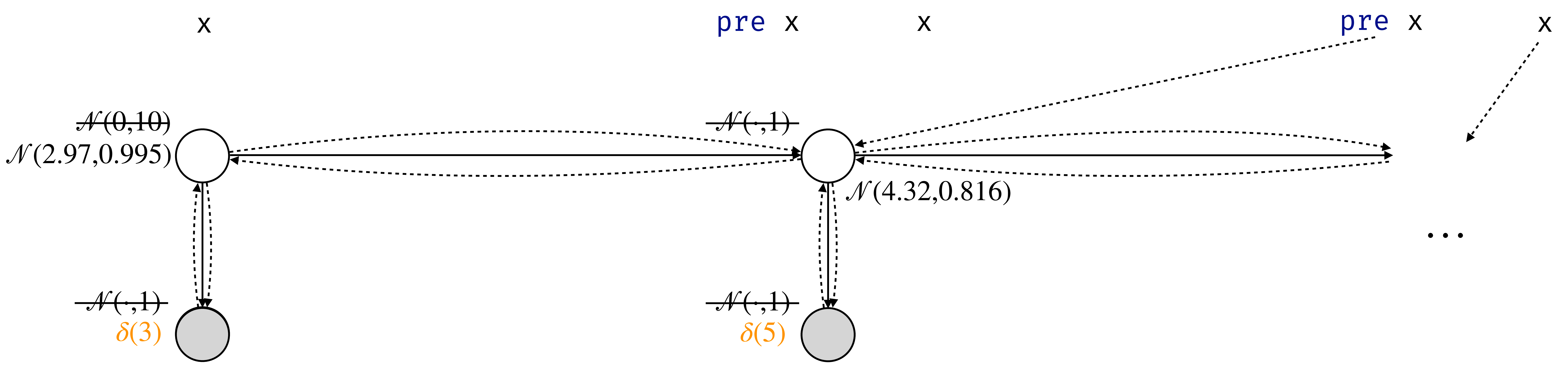
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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

$t = 2$

```
sample (gaussian (pre x, 1))
observe (gaussian (x, 1), …)
```



Streaming Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

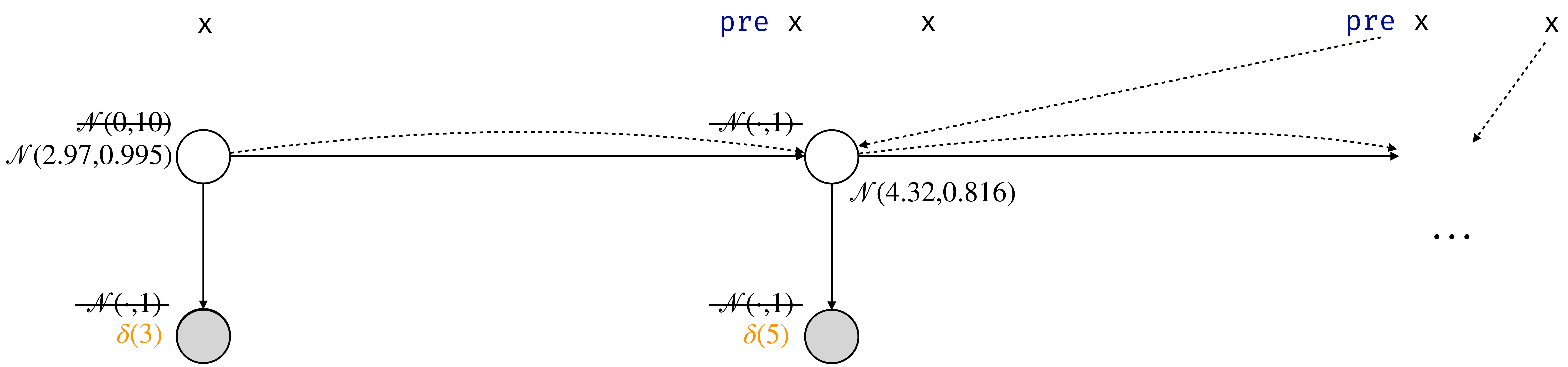
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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

$t = 2$

```
sample (gaussian (pre x, 1))
observe (gaussian (x, 1), …)
```



Streaming Delayed Sampling

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10)) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

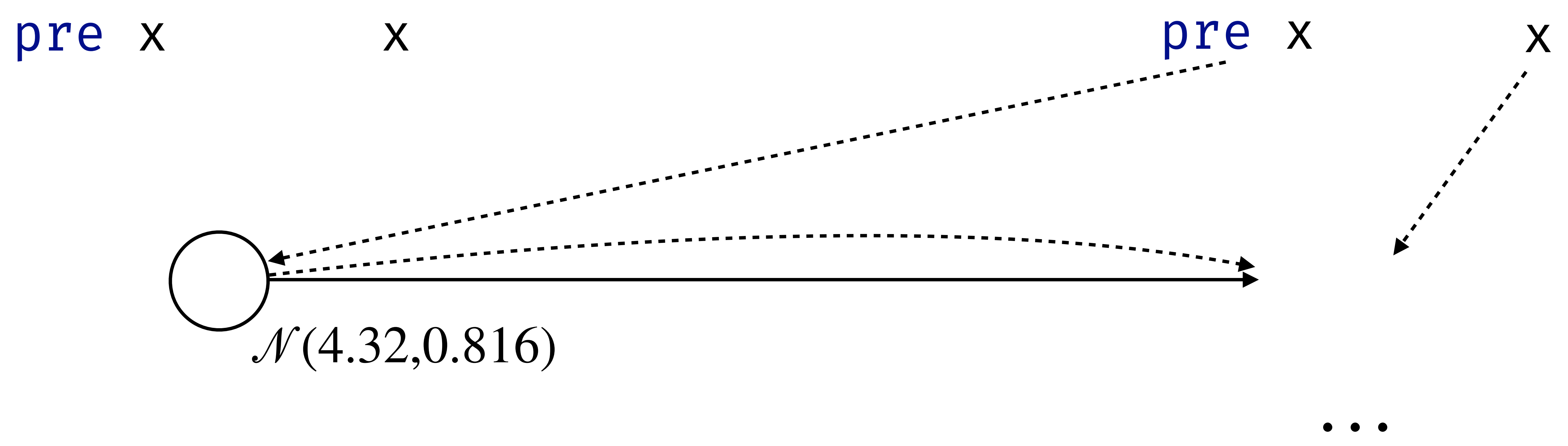
x

$t = 1$

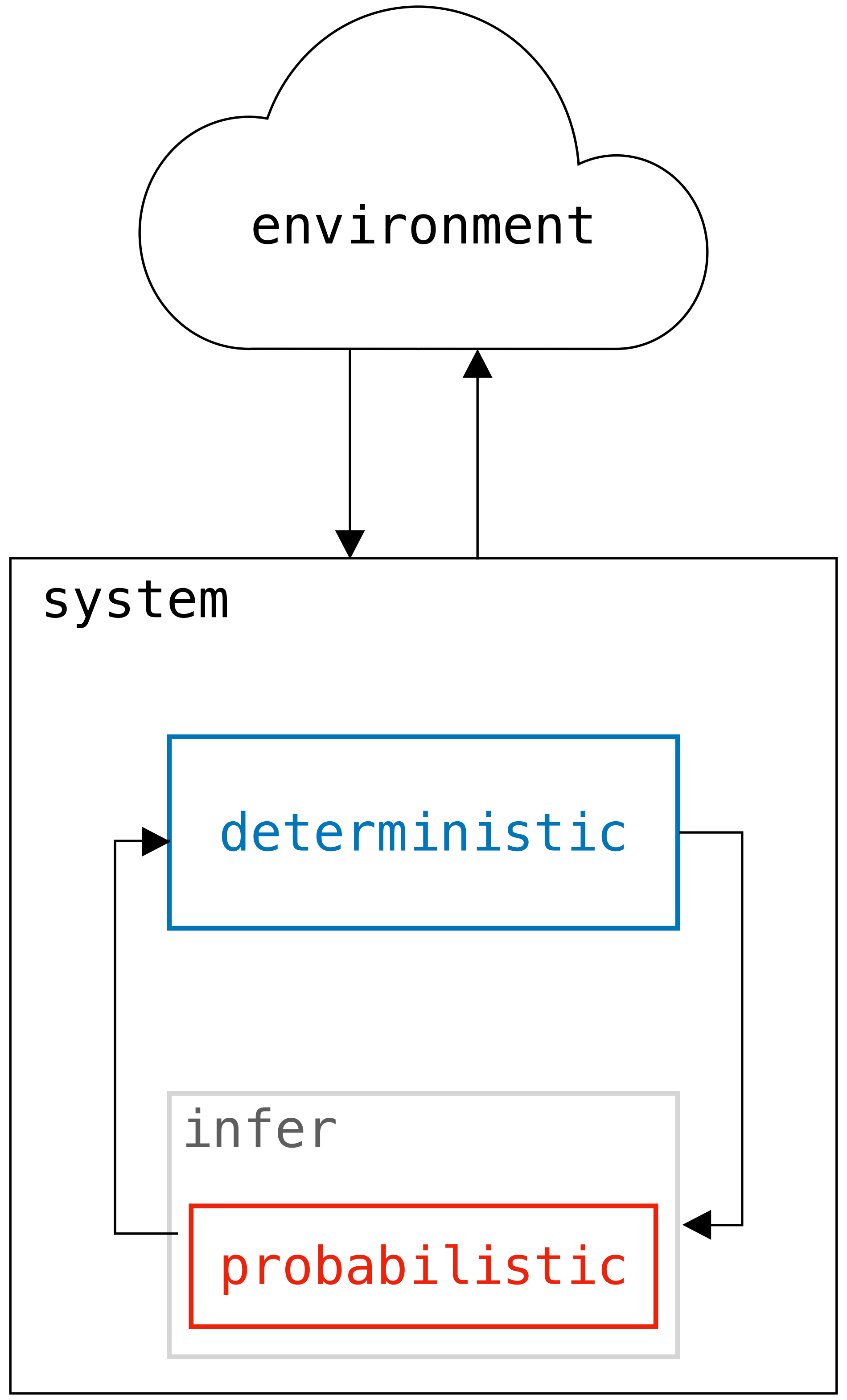
```
sample (gaussian (pre x, 1))
observe (gaussian (x, 1), 5)
```

$t = 2$

```
sample (gaussian (pre x, 1))
observe (gaussian (x, 1), …)
```

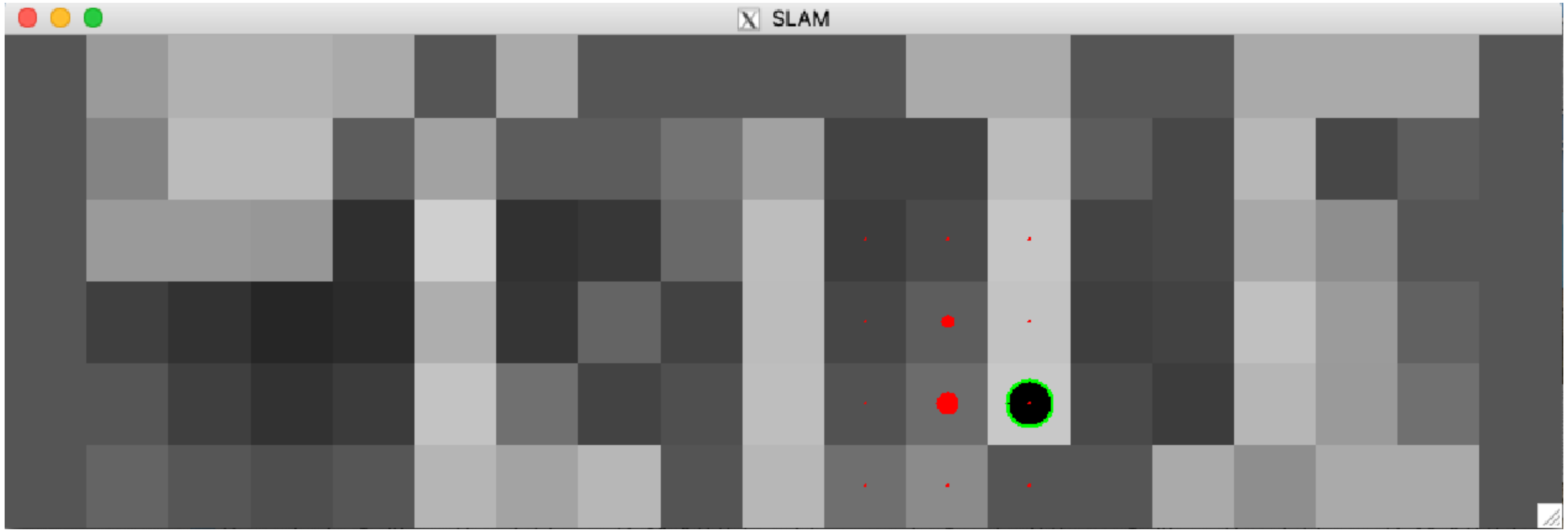


Reactive Probabilistic Programming (Demo)



Simultaneous **L**ocalization **A**nd **M**apping

- Environment: slippery wheels and noisy color sensor
- System: infer current position and map, output command (left/right/up/down)



At each step:

- Move to the next position
- Observe the color of the ground
- Use inferred position to compute next command

`exact position + color sensor`

`estimated color of a map cell`

`estimated position`

Evaluation

Language features

- Moving parameters
- Fixed parameters
- Inference-in-the-loop

Algorithms comparison

- ■ PF Particle Filtering
- ▼ SDS Streaming Delayed Sampling

Evaluation

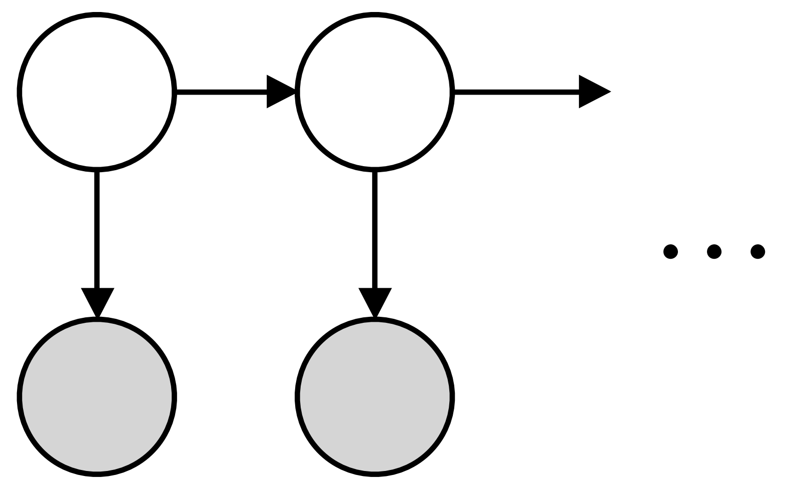
Language features

- Moving parameters
- Fixed parameters
- Inference-in-the-loop

Algorithms comparison

- PF Particle Filtering
- SDS Streaming Delayed Sampling

Moving parameters



Evaluation

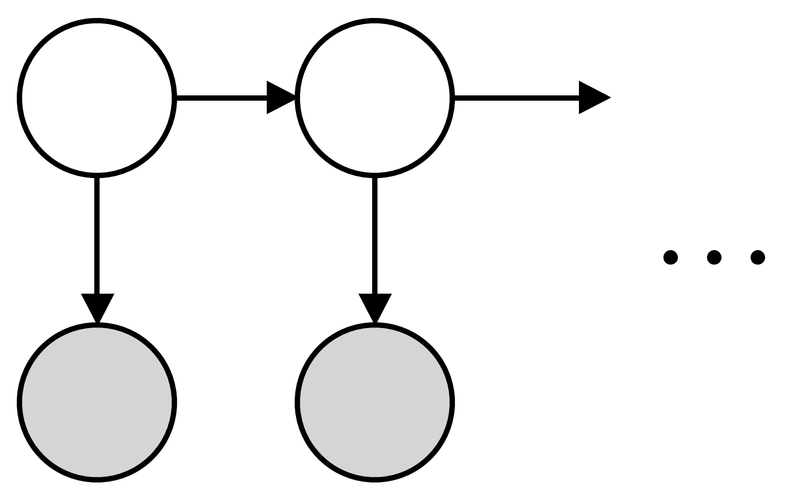
Language features

- Moving parameters
- Fixed parameters
- Inference-in-the-loop

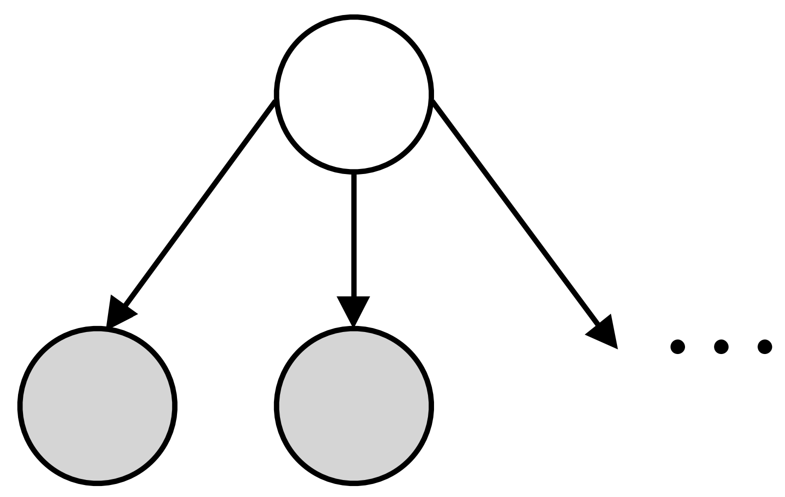
Algorithms comparison

- PF Particle Filtering
- SDS Streaming Delayed Sampling

Moving parameters



Fixed parameters

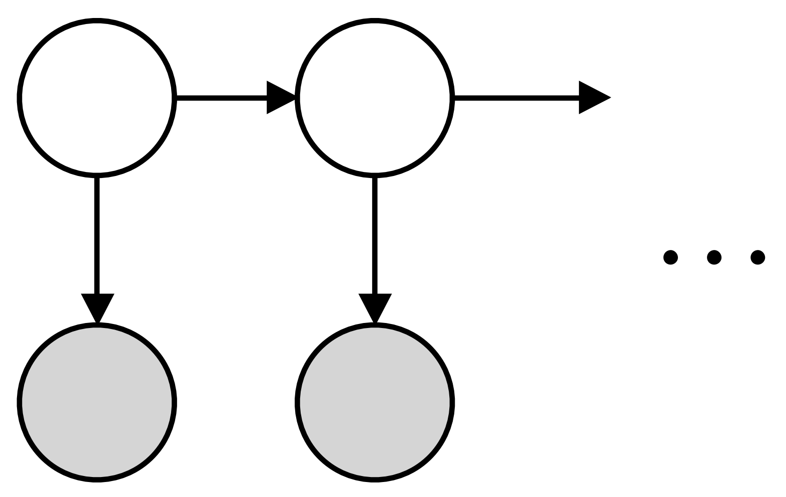


Evaluation

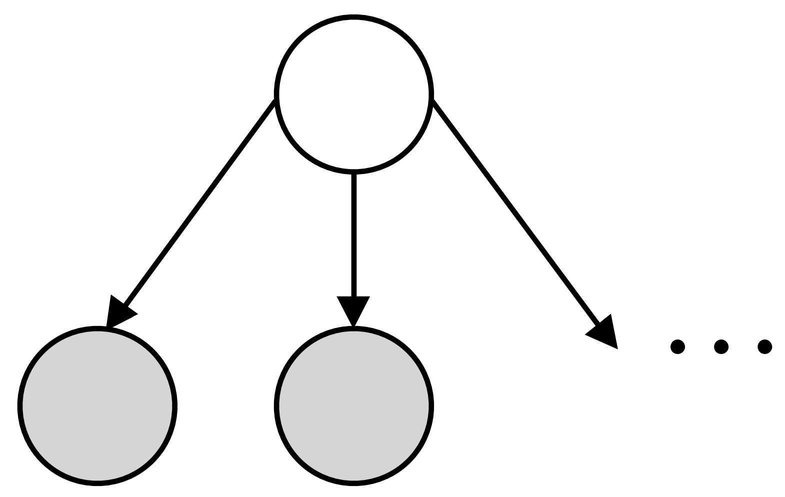
Language features

- Moving parameters
- Fixed parameters
- Inference-in-the-loop

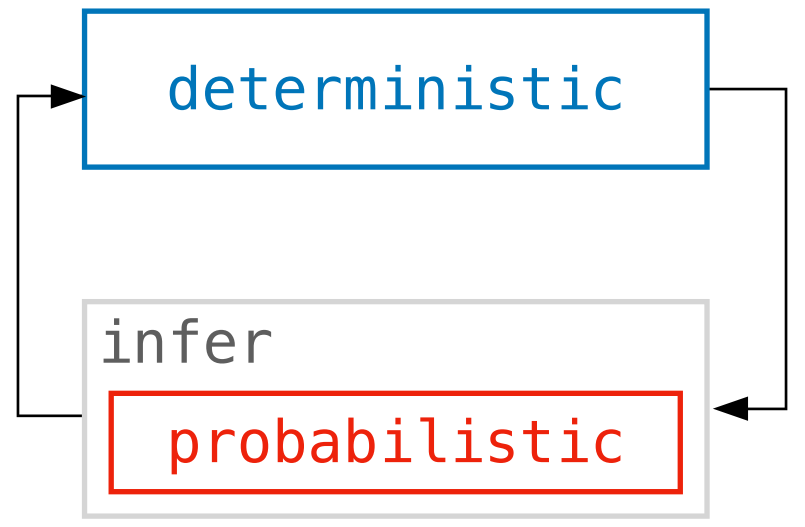
Moving parameters



Fixed parameters



Inference-in-the-loop



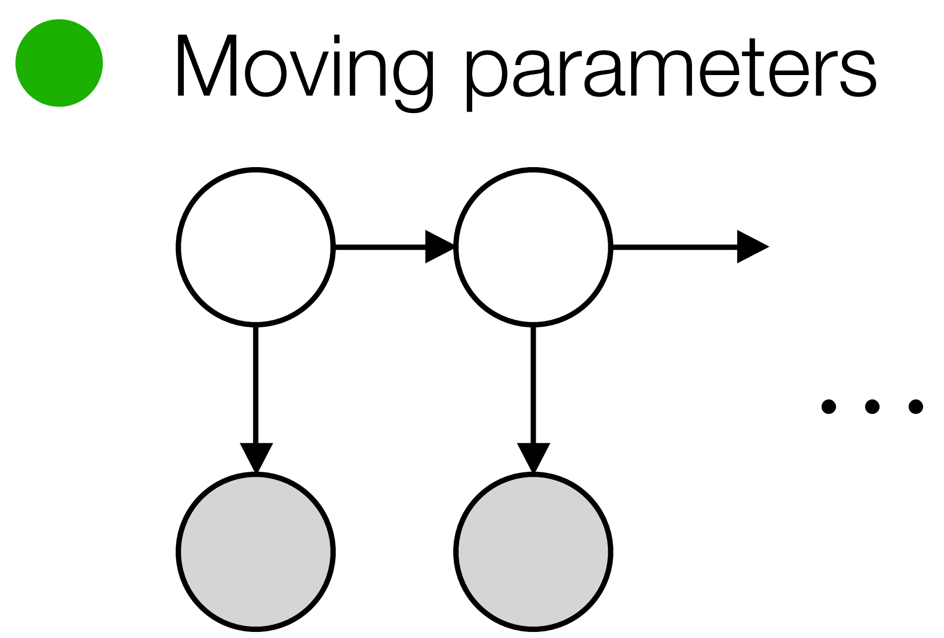
Algorithms comparison

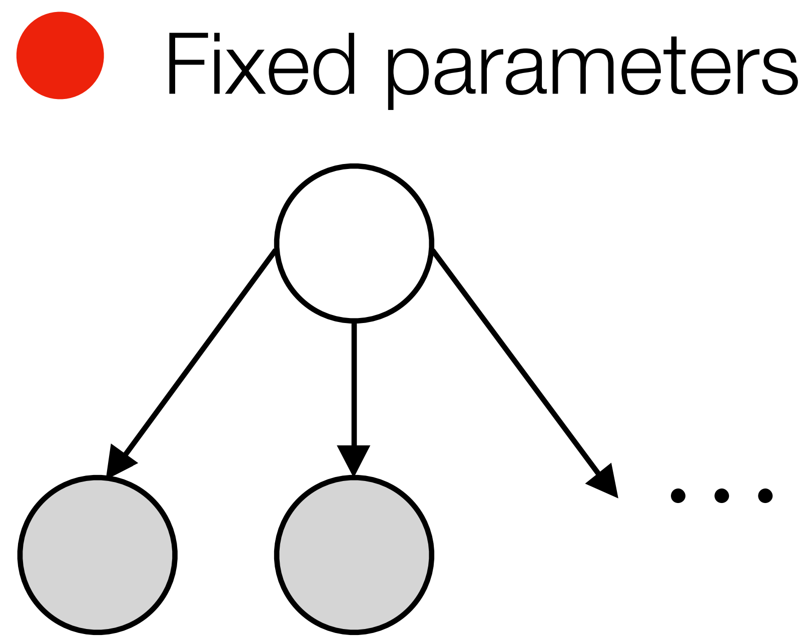
- PF Particle Filtering
- SDS Streaming Delayed Sampling

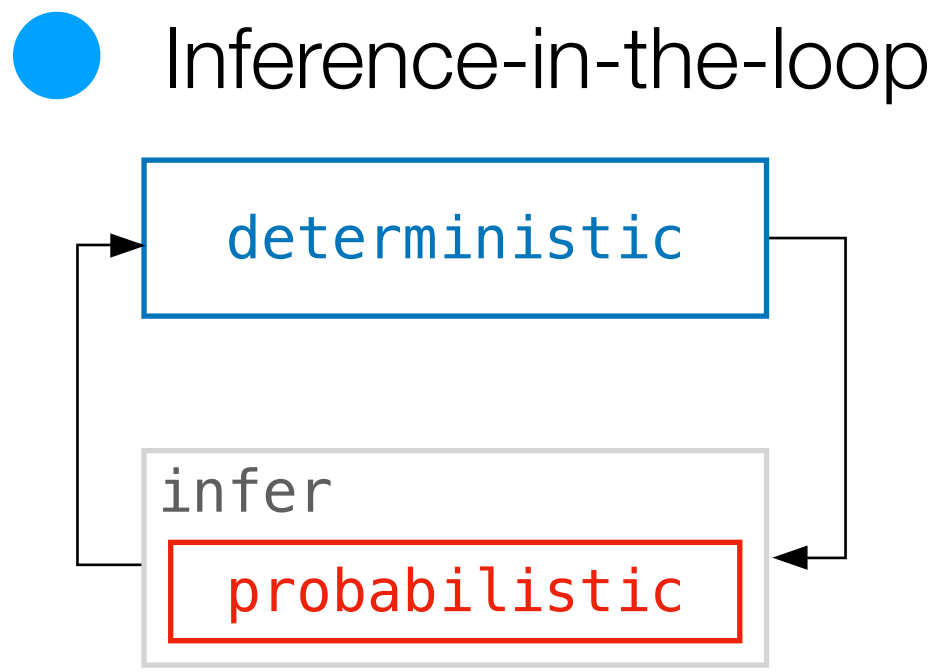
Evaluation

Language features

- Moving parameters
- Fixed parameters
- Inference-in-the-loop

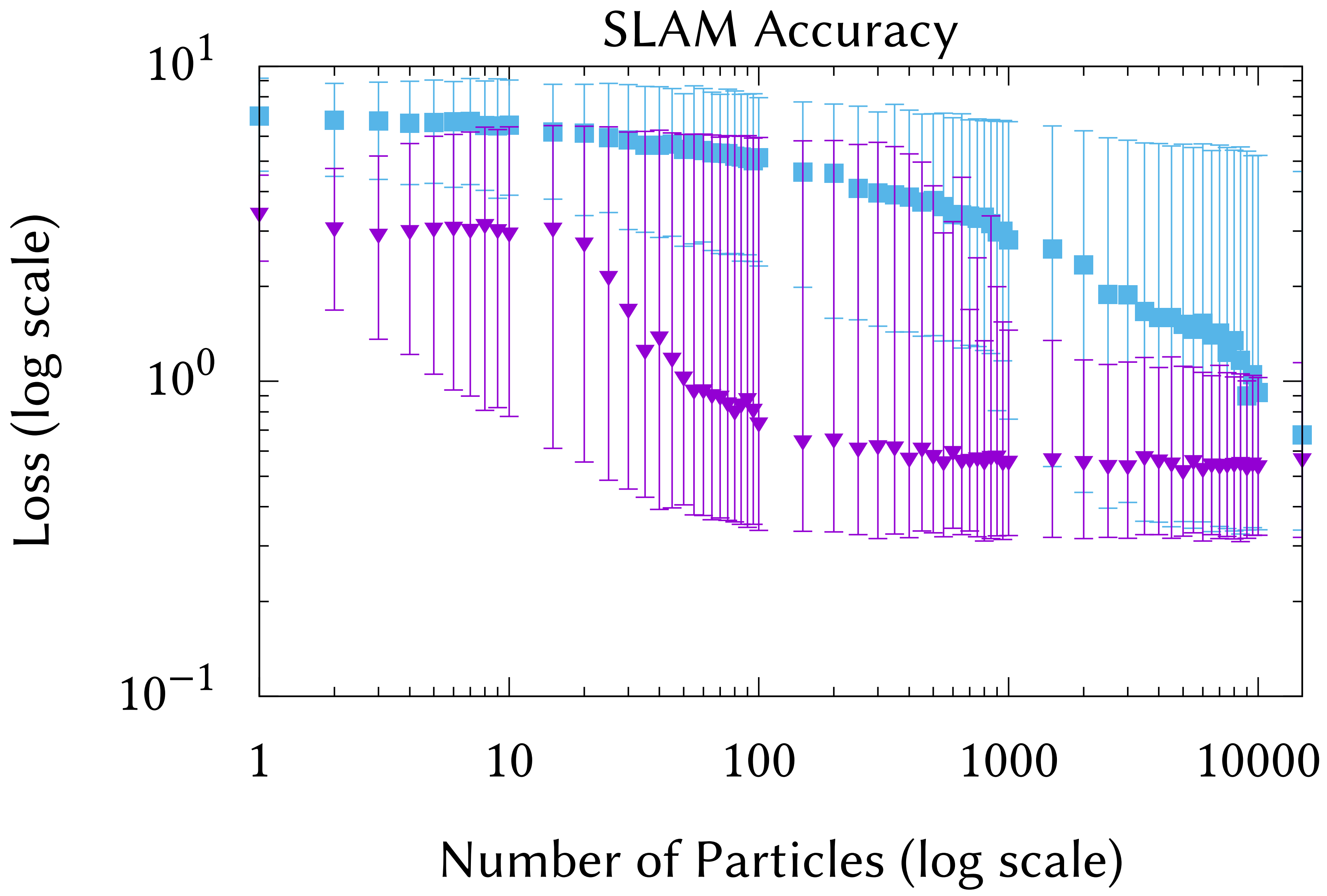


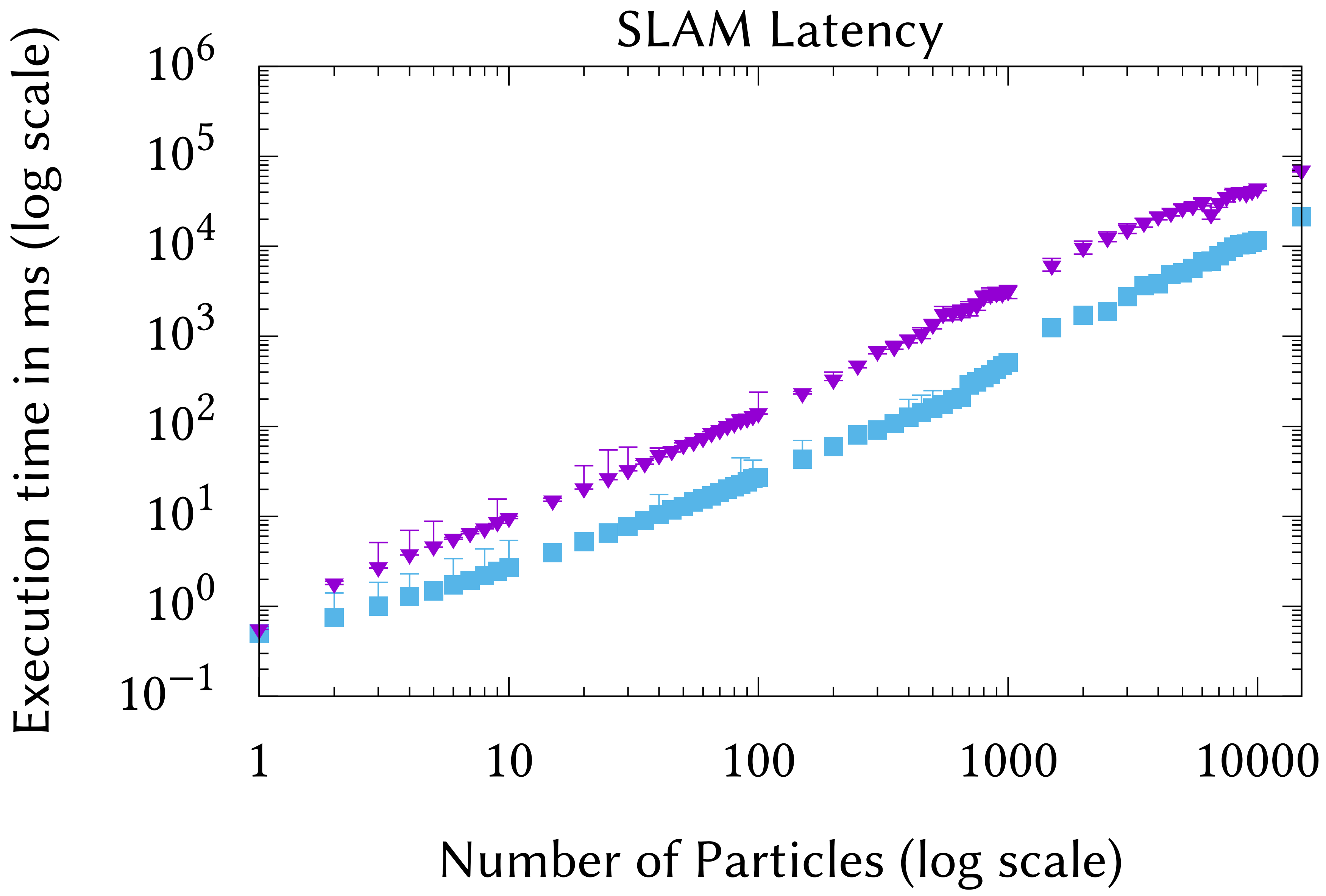




Algorithms comparison

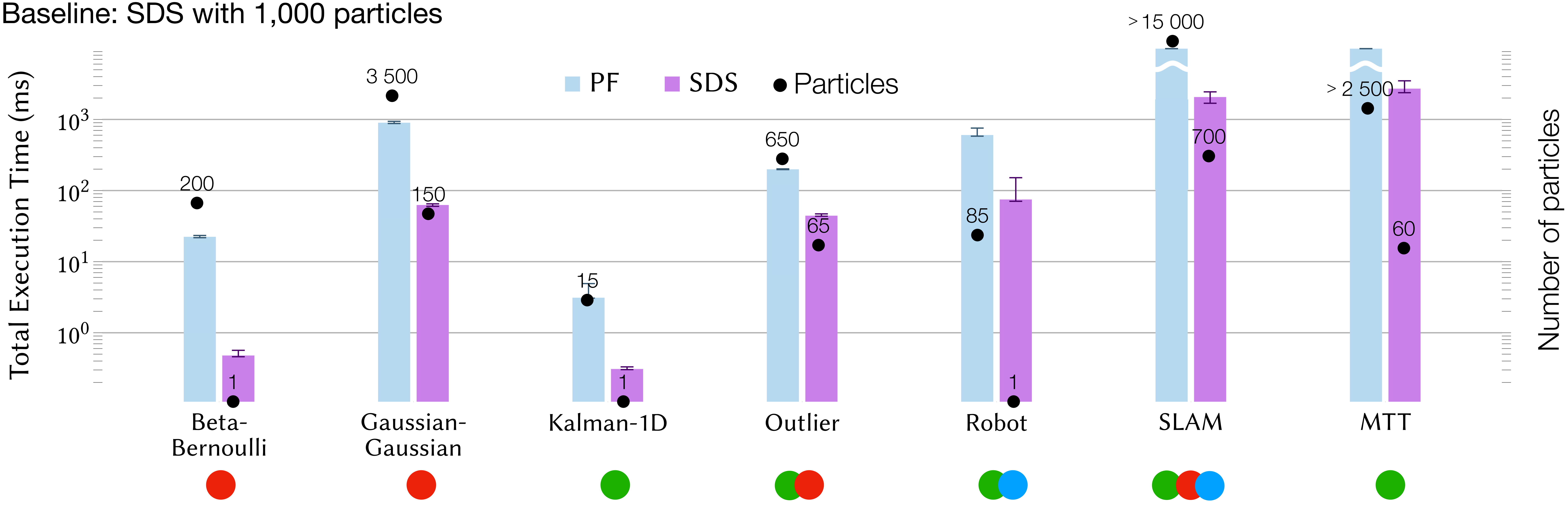
- PF Particle Filtering
- SDS Streaming Delayed Sampling





Benchmarks

Baseline: SDS with 1,000 particles



Moving parameters
Fixed parameters
Inference-in-the-loop

Conclusions

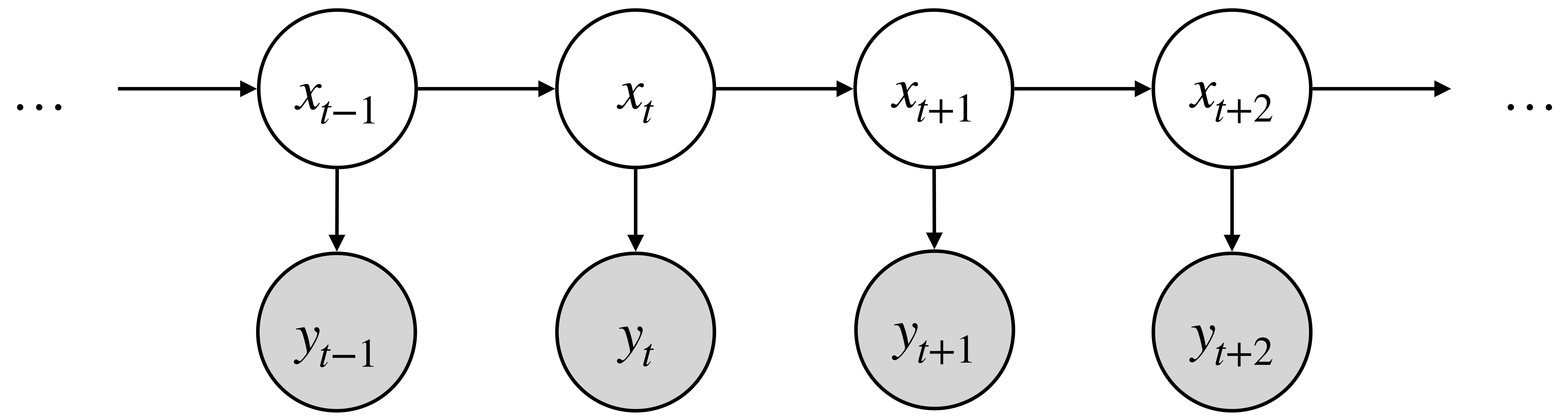
- SDS is always faster to match accuracy
- Reduction in particle count outweighs symbolic overhead
- SDS can be exact (1 particle)
- PF is impractical for advanced examples

Static Analysis

Reactive Probabilistic Programming

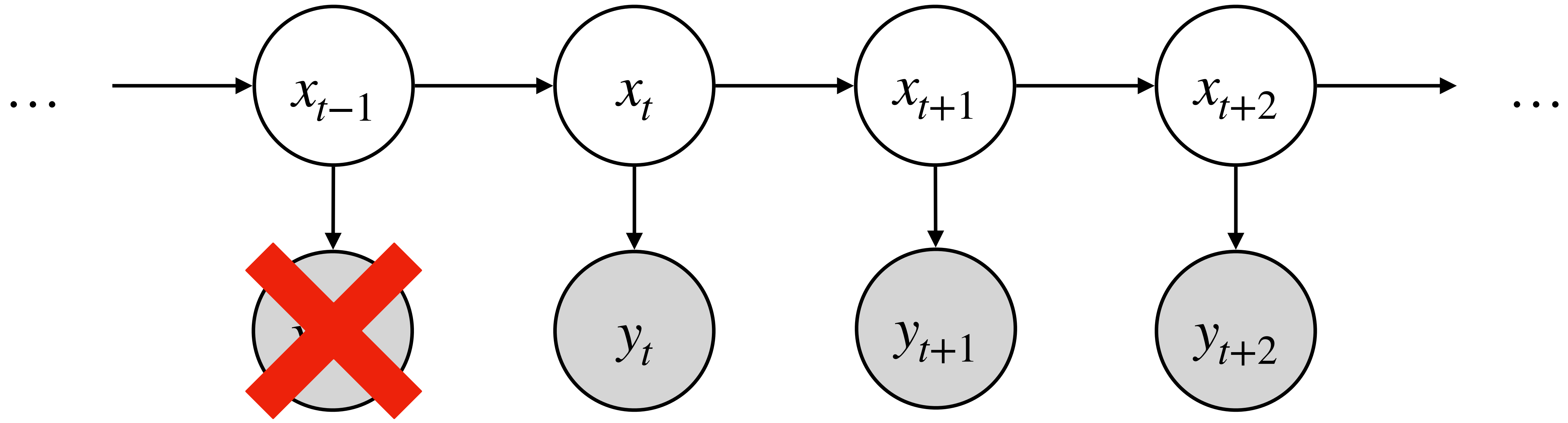
Bounded Memory Delayed Sampling?

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



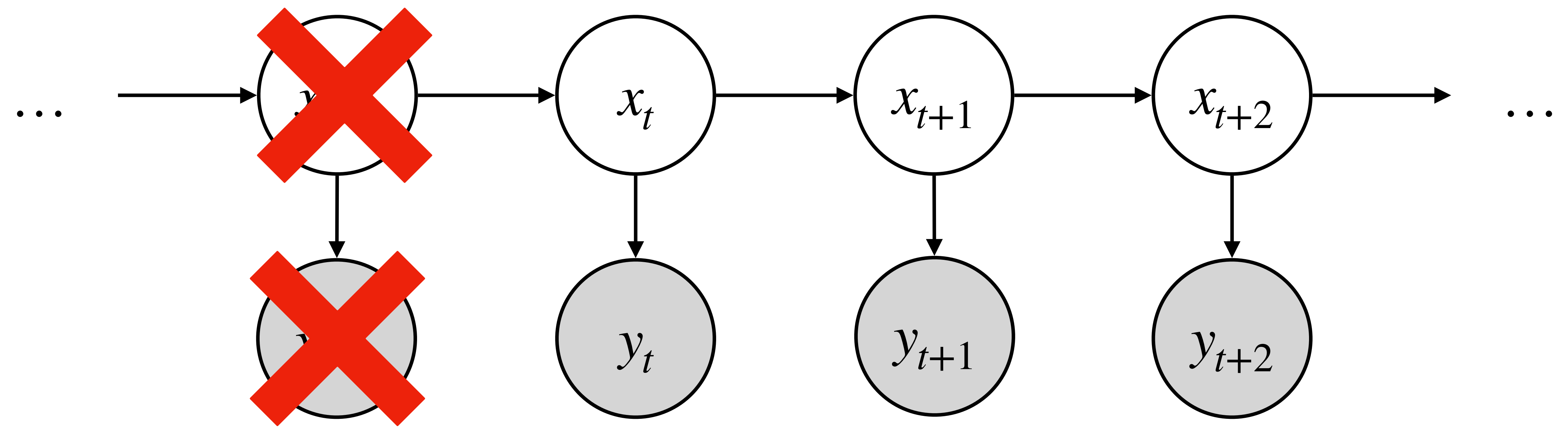
Bounded Memory Delayed Sampling?

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



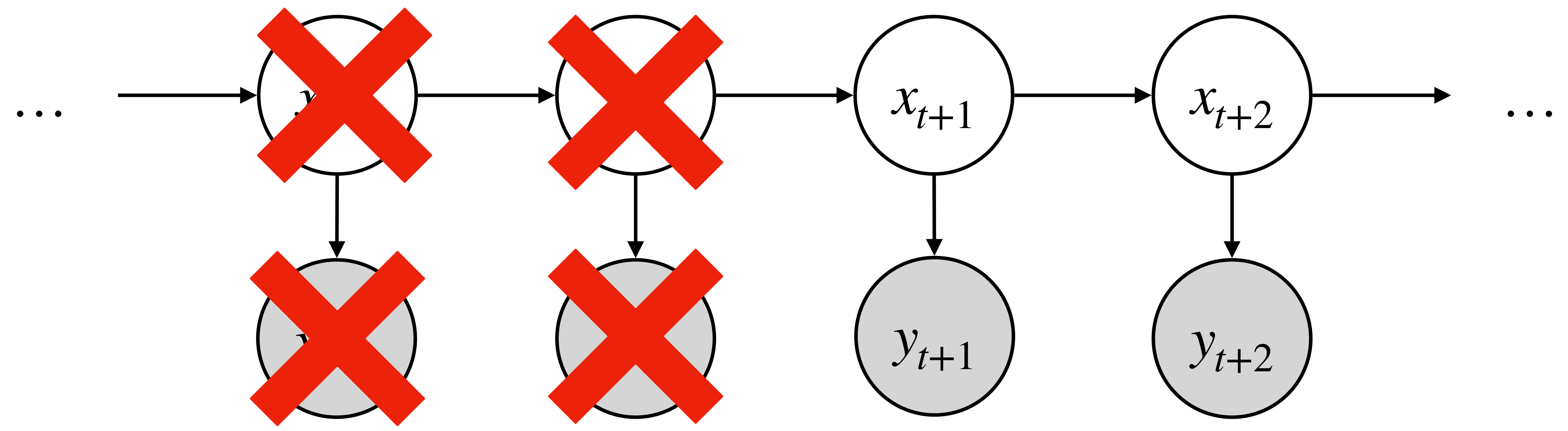
Bounded Memory Delayed Sampling?

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



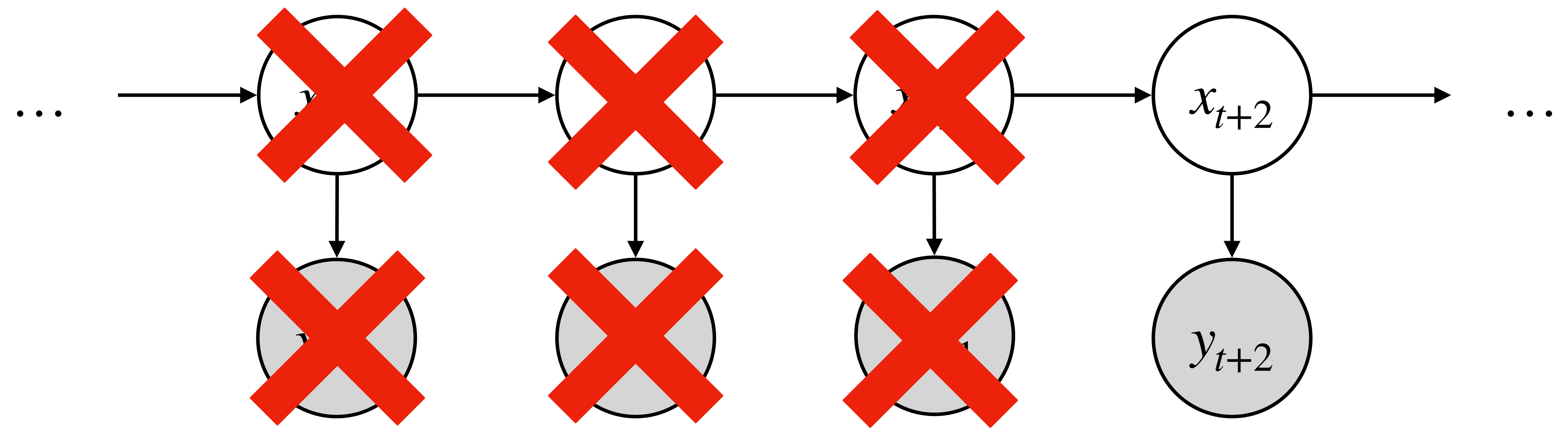
Bounded Memory Delayed Sampling?

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



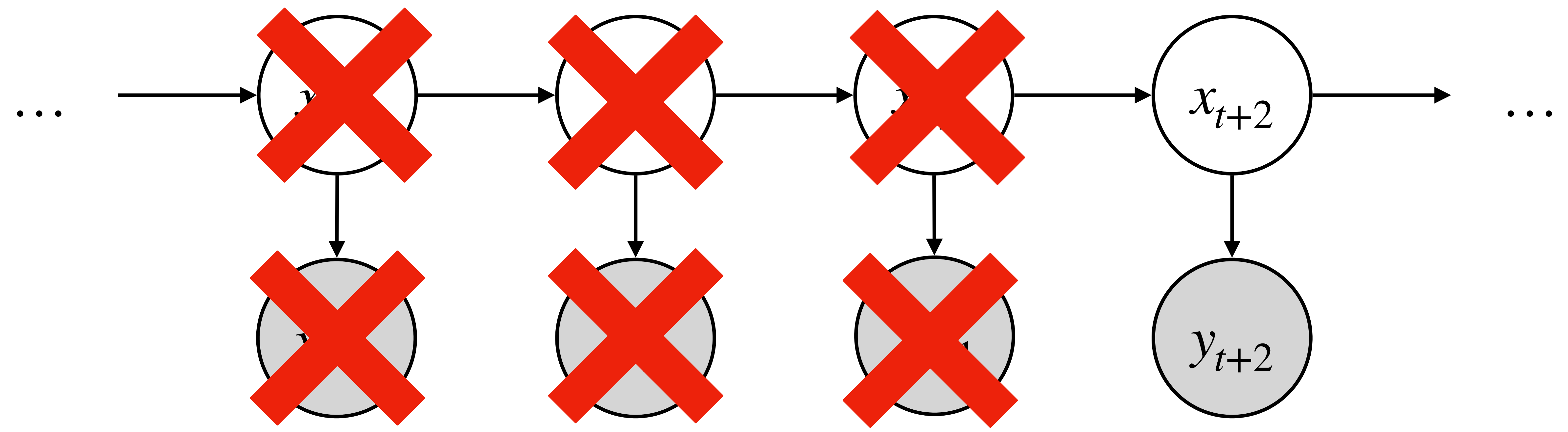
Bounded Memory Delayed Sampling?

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



Bounded Memory Delayed Sampling?

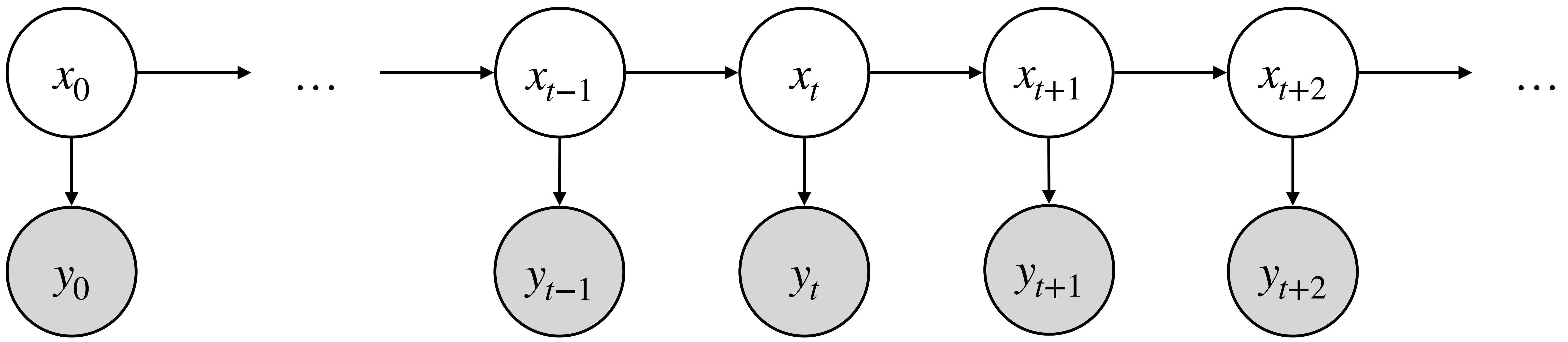
```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



Yes!

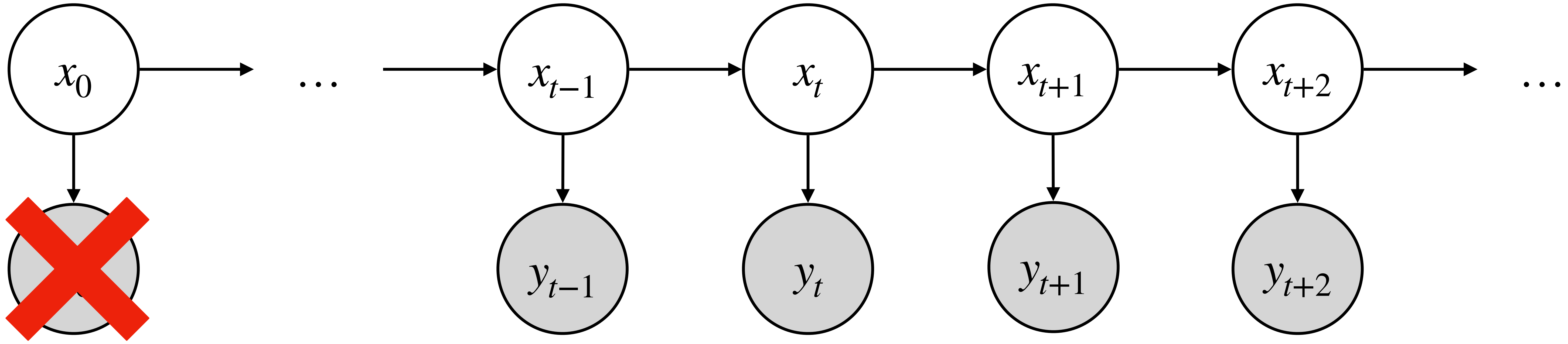
Bounded Memory Delayed Sampling?

```
proba tracker (y) = x, x0 where
  rec init x0 = sample (gaussian (0, 10))
  and x   = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



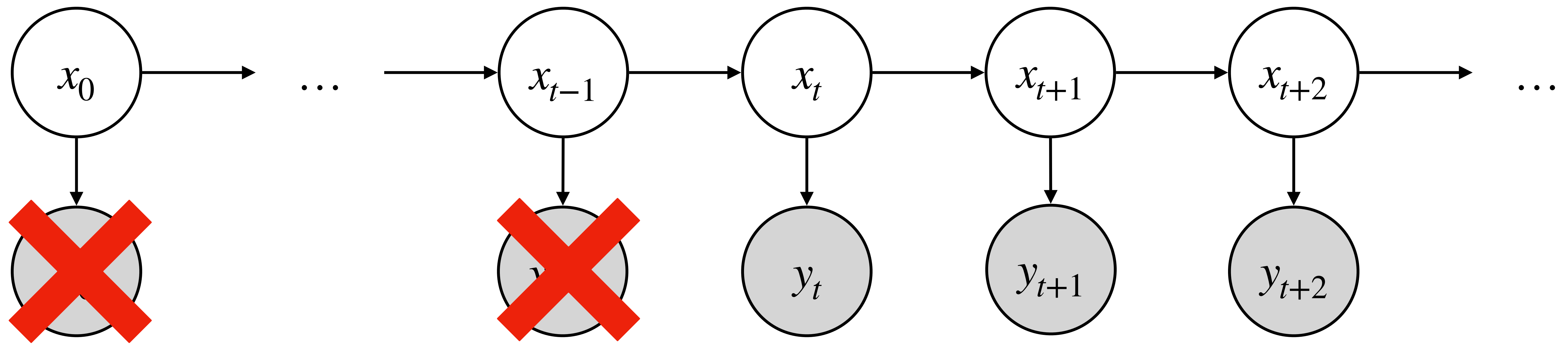
Bounded Memory Delayed Sampling?

```
proba tracker (y) = x, x0 where
  rec init x0 = sample (gaussian (0, 10))
  and x   = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



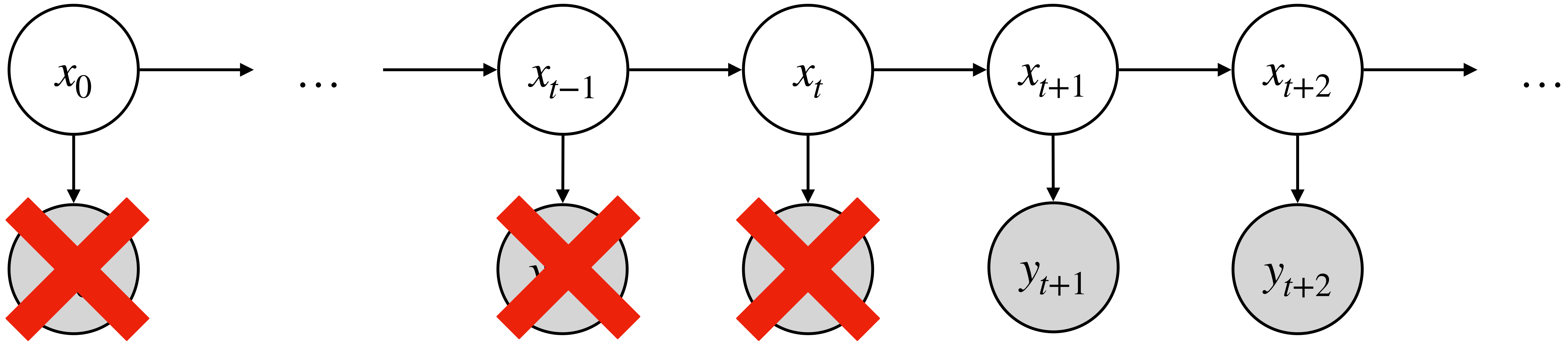
Bounded Memory Delayed Sampling?

```
proba tracker (y) = x, x0 where
  rec init x0 = sample (gaussian (0, 10))
  and x   = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



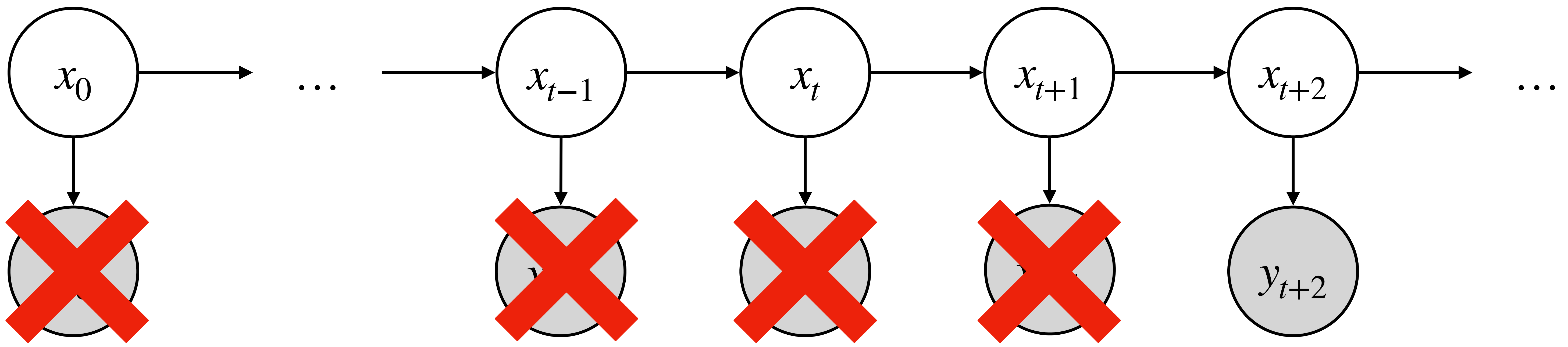
Bounded Memory Delayed Sampling?

```
proba tracker (y) = x, x0 where
  rec init x0 = sample (gaussian (0, 10))
  and x   = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



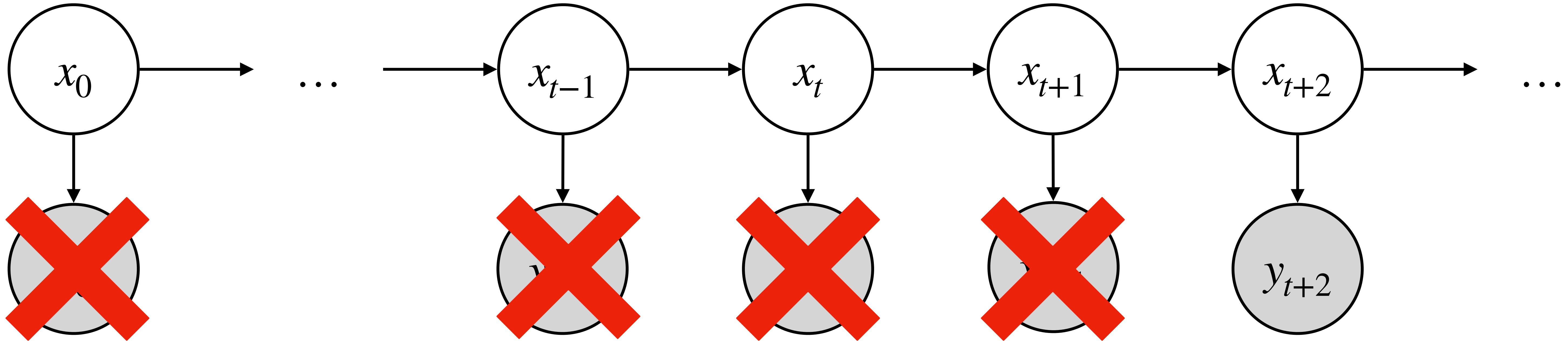
Bounded Memory Delayed Sampling?

```
proba tracker (y) = x, x0 where
  rec init x0 = sample (gaussian (0, 10))
  and x   = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



Bounded Memory Delayed Sampling?

```
proba tracker (y) = x, x0 where
  rec init x0 = sample (gaussian (0, 10))
  and x  = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

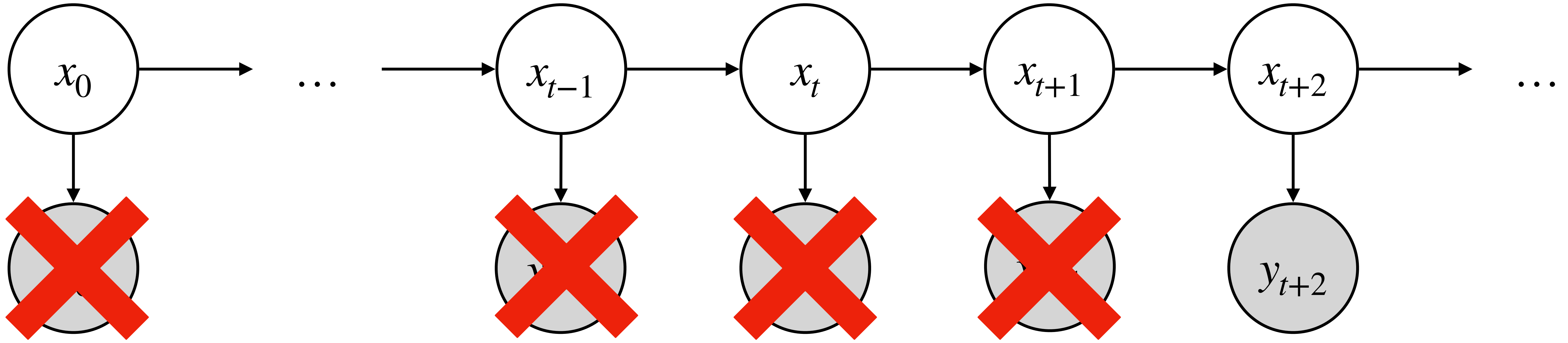


No!

Bounded Memory Delayed Sampling?

```
proba tracker (y) = x, x0 where
  rec init x0 = sample (gaussian (0, 10))
  and x  = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

Can we determine if a given program will run in bounded memory?



No!

Trace: Abstract Execution

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

trace	state	time
$x_0 \leftarrow \perp$ :: $y_0 \leftarrow x_0$ :: observe y_0 ::	x = x_0	$t = 0$
$x_1 \leftarrow x_0$ :: $y_1 \leftarrow x_1$ :: observe y_1 ::	x = x_1, pre x = x_0	$t = 1$
$x_2 \leftarrow x_1$ :: $y_2 \leftarrow x_2$ :: …	x = x_2, pre x = x_1	$t = 2$

Trace: Abstract Execution

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

	`trace`	`state`	`time`
random variable →	$x_0 \leftarrow \perp$ ::	`x = ` x_0	$t = 0$
	$y_0 \leftarrow x_0$ ::		
	observe y_0 ::		
	$x_1 \leftarrow x_0$ ::	`x = ` x_1`, pre x = ` x_0	$t = 1$
	$y_1 \leftarrow x_1$ ::		
	observe y_1 ::		
	$x_2 \leftarrow x_1$ ::	`x = ` x_2`, pre x = ` x_1	$t = 2$
	$y_2 \leftarrow x_2$ ::		
	$\ldots$		

Trace: Abstract Execution

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

	trace	state	time
random variable →	$x_0 \leftarrow \perp$ ::	`x` = x_0	$t = 0$
	$y_0 \leftarrow x_0$ ::		
observation →	observe y_0 ::		
	$x_1 \leftarrow x_0$ ::	`x` = x_1, `pre x` = x_0	$t = 1$
	$y_1 \leftarrow x_1$ ::		
	observe y_1 ::		
	$x_2 \leftarrow x_1$ ::	`x` = x_2, `pre x` = x_1	$t = 2$
	$y_2 \leftarrow x_2$ ::		
	…		

Static Analysis for Delayed Sampling

Semantic properties

m-consumed property
Chains of variables before an observe are bounded

unseparated paths property
Chains of variables referenced in the state are bounded

Theorem: *The program satisfies these two properties iff it executes in bounded memory*

Static Analysis for Delayed Sampling

Semantic properties

m-consumed property
Chains of variables before an observe are bounded

unseparated paths property
Chains of variables referenced in the state are bounded

Theorem: *The program satisfies these two properties iff it executes in bounded memory*

Static analysis

Track variables introduced but not used yet

Track maximal path between pairs of variable in the state

Theorem: *The program pass the analysis if it executes in bounded memory*

m-consumed Property

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

trace	state	time
$x_0 \leftarrow \perp$ ::	`x = ` x_0	$t = 0$
$y_0 \leftarrow x_0$ ::		
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x = ` x_1`, pre x = ` x_0	$t = 1$
$y_1 \leftarrow x_1$ ::		
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x = ` x_2`, pre x = ` x_1	$t = 2$
$y_2 \leftarrow x_2$ ::		
$\ldots$		

m-consumed Property

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gauss
  and () = observe (gaussian (x, 1), y)
```

Chains of variables before an observe are bounded

trace	state	time
$x_0 \leftarrow \bot$ ::	`x = ` x_0	$t = 0$
$y_0 \leftarrow x_0$ ::		
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x = ` x_1`, pre x = ` x_0	$t = 1$
$y_1 \leftarrow x_1$ ::		
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x = ` x_2`, pre x = ` x_1	$t = 2$
$y_2 \leftarrow x_2$ ::		
…		

m-consumed Property

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gauss
  and () = observe (gaussian (x, 1), y)
```

Chains of variables before an observe are bounded

y_0 is 0-consumed

trace	state	time
$x_0 \leftarrow \bot$ ::	`x = ` x_0	$t = 0$
$y_0 \leftarrow x_0$ ::		
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x = ` x_1`, pre x = ` x_0	$t = 1$
$y_1 \leftarrow x_1$ ::		
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x = ` x_2`, pre x = ` x_1	$t = 2$
$y_2 \leftarrow x_2$ ::		
…		

m-consumed Property

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gauss
  and () = observe (gaussian (x, 1), y)
```

Chains of variables before an observe are bounded

x_0 is 1-consumed

y_0 is 0-consumed

trace	state	time
$x_0 \leftarrow \bot$ ::	`x = ` x_0	$t = 0$
$y_0 \leftarrow x_0$ ::		
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x = ` x_1`, pre x = ` x_0	$t = 1$
$y_1 \leftarrow x_1$ ::		
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x = ` x_2`, pre x = ` x_1	$t = 2$
$y_2 \leftarrow x_2$ ::		
…		

m-consumed Property

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gauss
  and () = observe (gaussian (x, 1), y)
```

Chains of variables before an observe are bounded

	trace	state	time
	$x_0 \leftarrow \bot$ ::	`x` = x_0	$t = 0$
x_0 is 1-consumed →	$y_0 \leftarrow x_0$ ::		
y_0 is 0-consumed →	observe y_0 ::		
	$x_1 \leftarrow x_0$ ::	`x` = x_1, `pre x` = x_0	$t = 1$
x_1 is 1-consumed →	$y_1 \leftarrow x_1$ ::		
y_1 is 0-consumed →	observe y_1 ::		
	$x_2 \leftarrow x_1$ ::	`x` = x_2, `pre x` = x_1	$t = 2$
	$y_2 \leftarrow x_2$ ::		
	…		

m-consumed Property

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gauss
  and () = observe (gaussian (x, 1), y)
```

Chains of variables before an observe are bounded

	trace	state	time
	$x_0 \leftarrow \bot$ ::	`x = ` x_0	$t = 0$
x_0 is 1-consumed	$y_0 \leftarrow x_0$ ::		
y_0 is 0-consumed	observe y_0 ::		
	$x_1 \leftarrow x_0$ ::	`x = ` x_1, `pre x = ` x_0	$t = 1$
x_1 is 1-consumed	$y_1 \leftarrow x_1$ ::		
y_1 is 0-consumed	observe y_1 ::		
	$x_2 \leftarrow x_1$ ::	`x = ` x_2, `pre x = ` x_1	$t = 2$
	$y_2 \leftarrow x_2$ ::		
	…		

Yes!

Unseparated Paths Property

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

trace	state	time
$x_0 \leftarrow \bot$ ::	`x = ` x_0	$t = 0$
$y_0 \leftarrow x_0$ ::		
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x = ` x_1`, pre x = ` x_0	$t = 1$
$y_1 \leftarrow x_1$ ::		
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x = ` x_2`, pre x = ` x_1	$t = 2$
$y_2 \leftarrow x_2$ ::		
…		

Unseparated Paths Property

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gauss
  and () = observe (gaussian (x, 1), y)
```

Chains of variables referenced in the state are bounded

trace	state	time
$x_0 \leftarrow \bot$ ::	`x = ` x_0	$t = 0$
$y_0 \leftarrow x_0$ ::		
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x = ` x_1`, pre x = ` x_0	$t = 1$
$y_1 \leftarrow x_1$ ::		
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x = ` x_2`, pre x = ` x_1	$t = 2$
$y_2 \leftarrow x_2$ ::		
$\ldots$		

Unseparated Paths Property

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10)) → gauss
  and () = observe (gaussian (x, 1), y)
```

Chains of variables referenced in the state are bounded

trace	state	time
$x_0 \leftarrow \perp$ ::	`x` = x_0	$t = 0$
$y_0 \leftarrow x_0$ ::		
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x` = x_1, `pre x` = x_0	$t = 1$
$y_1 \leftarrow x_1$ ::		
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x` = x_2, `pre x` = x_1	$t = 2$
$y_2 \leftarrow x_2$ ::		
…		

Unseparated Paths Property

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gauss
  and () = observe (gaussian (x, 1), y)
```

Chains of variables referenced in the state are bounded

trace	state	time
$x_0 \leftarrow \perp$ ::	`x` = x_0	$t = 0$
$y_0 \leftarrow x_0$ ::		
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x` = x_1, `pre x` = x_0	$t = 1$
$y_1 \leftarrow x_1$ ::		
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x` = x_2, `pre x` = x_1	$t = 2$
$y_2 \leftarrow x_2$ ::		
…		

Yes!

Unseparated Paths Property

```
proba tracker (obs) = x where
  rec init x0 = sample (gaussian (0, 10))
  and x  = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

Chains of variables referenced in the state are bounded

trace	state	time
$x_0 \leftarrow \perp$ ::	`x` = x_0	$t = 0$
$y_0 \leftarrow x_0$ ::	`x0` = x_0	
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x` = x_1, `pre x` = x_0	$t = 1$
$y_1 \leftarrow x_1$ ::	`x0` = x_0	
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x` = x_2, `pre x` = x_1	$t = 2$
$y_2 \leftarrow x_2$ ::	`x0` = x_0	
…		

Unseparated Paths Property

```
proba tracker (obs) = x where
  rec init x0 = sample (gaussian (0, 10))
  and x   = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

Chains of variables referenced in the state are bounded

trace	state	time
$x_0 \leftarrow \bot$ ::	`x` = x_0	$t = 0$
$y_0 \leftarrow x_0$ ::	`x0` = x_0	
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x` = x_1, `pre x` = x_0	$t = 1$
$y_1 \leftarrow x_1$ ::	`x0` = x_0	
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x` = x_2, `pre x` = x_1	$t = 2$
$y_2 \leftarrow x_2$ ::	`x0` = x_0	
…		

Unseparated Paths Property

```
proba tracker (obs) = x where
  rec init x0 = sample (gaussian (0, 10))
  and x   = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

Chains of variables referenced in the state are bounded

trace	state	time
$x_0 \leftarrow \perp$ ::	`x = ` x_0	$t = 0$
$y_0 \leftarrow x_0$ ::	`x0 = ` x_0	
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	`x = ` x_1`, pre x = ` x_0	$t = 1$
$y_1 \leftarrow x_1$ ::	`x0 = ` x_0	
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	`x = ` x_2`, pre x = ` x_1	$t = 2$
$y_2 \leftarrow x_2$ ::	`x0 = ` x_0	
…		

No!

Evaluation

	m-consumed		unsep. paths		bounded mem.	
	output	actual	output	actual	output	actual
Kalman	✓	✓	✓	✓	✓	✓
Kalman Hold-First	✓	✓	✗	✗	✗	✗
Gaussian Random Walk	✗	✗	✓	✓	✗	✗
Robot	✓	✓	✓	✓	✓	✓
Coin	✓	✓	✓	✓	✓	✓
Gaussian-Gaussian	✓	✓	✓	✓	✓	✓
Outlier	✗	✗	✓	✓	✗	✗
MTT	✗	✗	✓	✓	✗	✗
SLAM	✗	✓	✓	✓	✗	✓

Evaluation

memory is probabilistically bounded

	m-consumed		unsep. paths		bounded mem.	
	output	actual	output	actual	output	actual
Kalman	✓	✓	✓	✓	✓	✓
Kalman Hold-First	✓	✓	✗	✗	✗	✗
Gaussian Random Walk	✗	✗	✓	✓	✗	✗
Robot	✓	✓	✓	✓	✓	✓
Coin	✓	✓	✓	✓	✓	✓
Gaussian-Gaussian	✓	✓	✓	✓	✓	✓
Outlier	✗	✗	✓	✓	✗	✗
MTT	✗	✗	✓	✓	✗	✗
SLAM	✗	✓	✓	✓	✗	✓

Evaluation

	m-consumed		unsep. paths		bounded mem.	
	output	actual	output	actual	output	actual
Kalman	✓	✓	✓	✓	✓	✓
Kalman Hold-First	✓	✓	✗	✗	✗	✗
Gaussian Random Walk	✗	✗	✓	✓	✗	✗
Robot	✓	✓	✓	✓	✓	✓
Coin	✓	✓	✓	✓	✓	✓
Gaussian-Gaussian	✓	✓	✓	✓	✓	✓
Outlier	✗	✗	✓	✓	✗	✗
MTT	✗	✗	✓	✓	✗	✗
SLAM	✗	✓	✓	✓	✗	✓

memory is probabilistically bounded

memory is always bounded

Take Away

Language design and implementation

- Parallel composition, control structures, and inference in the loop
- Measure-based semantics

Inference with bounded resources

- Semi-symbolic inference on streaming models based on Delayed Sampling

Static analysis

- Can delayed sampling run in bounded memory?

Ongoing and future work

- JAX based parallel inference (L. Mandel, R. Tekin)
- Reactive probabilistic programming in Julia (W. Azizian, M. Lelarge)
- Hybrid probabilistic programming (L. Mandel, M. Pouzet, C. Tasson)

Take Away

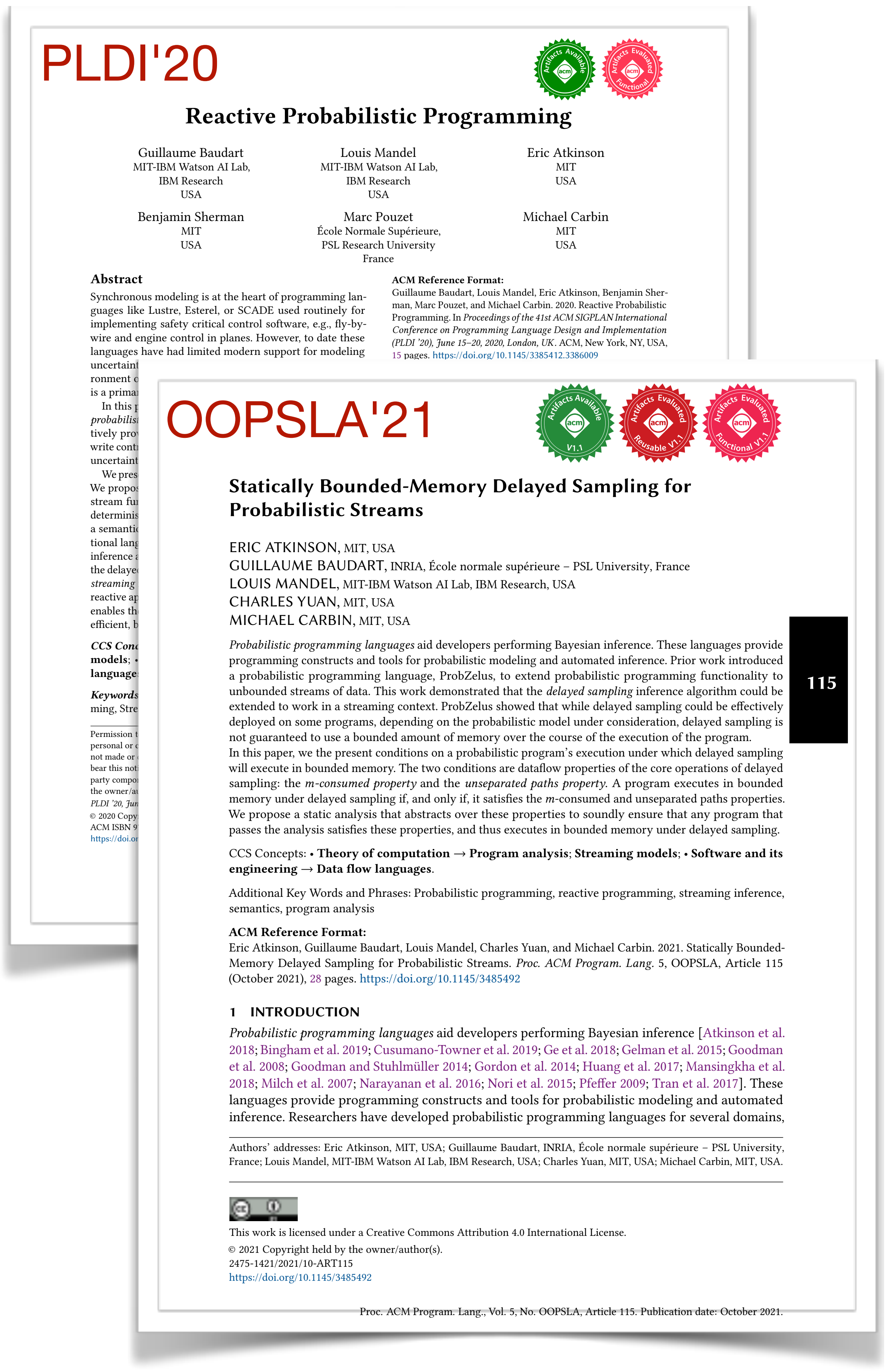
https://github.com/IBM/probzelus

Language design and implementation
- Parallel composition, control structures, and inference in the loop
- Measure-based semantics

Inference with bounded resources
- Semi-symbolic inference on streaming models based on Delayed Sampling

Static analysis
- Can delayed sampling run in bounded memory?

Ongoing and future work
- JAX based parallel inference (L. Mandel, R. Tekin)
- Reactive probabilistic programming in Julia (W. Azizian, M. Lelarge)
- Hybrid probabilistic programming (L. Mandel, M. Pouzet, C. Tasson)

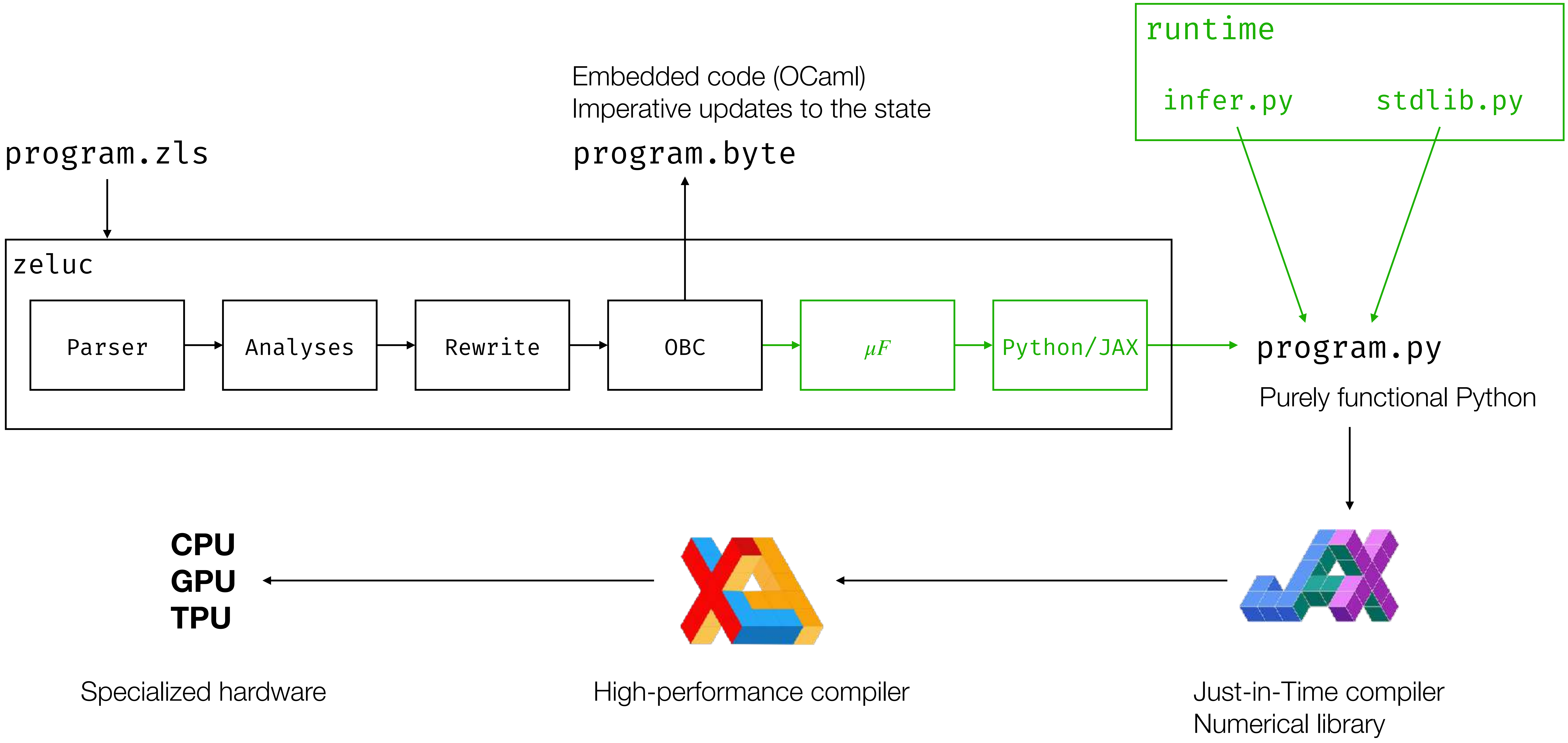


Ongoing & Future Work

Reactive Probabilistic Programming

JAX-Based Parallel Inference

with L. Mandel and R. Tekin



Reactive Probabilistic Programming in Julia

with W. Azizian and M. Lelarge

```
@node function model()
    @init x = rand(Normal(0.0, 1000.0))   # x_0 ~ N(0, 1000)
    x = rand(Normal(@prev(x), speed))     # x_t ~ N(x_{t-1}, speed)
    y = rand(Normal(x, noise))            # y_t ~ N(x_t, noise)
    return x, y
end


@node function hmm(obs)
    x, y = @nodecall model() # apply model to get x, y
    @observe(y, obs)         # assume y_t is observed with value obs_t
    return x
end
```

Hybrid Probabilistic Programming with ODEs

with L. Mandel, M. Pouzet, and C. Tasson

```
let hybrid ball g = h where
  rec der v = -. g init 0. reset up (-. h) → -. phi *. (last v)
  and der h = v init h0

let hybrid proba ball_pos obs = g where
  rec init g = sample (gaussian (5., 5.))
  and h = ball g
  and present obs(x) →
    do () = observe (gaussian (h, 0.1), x) done
```