Probabilistic Programming Languages

Reactive Probabilistic Programming

Guillaume Baudart

MPRI 2023-2024

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

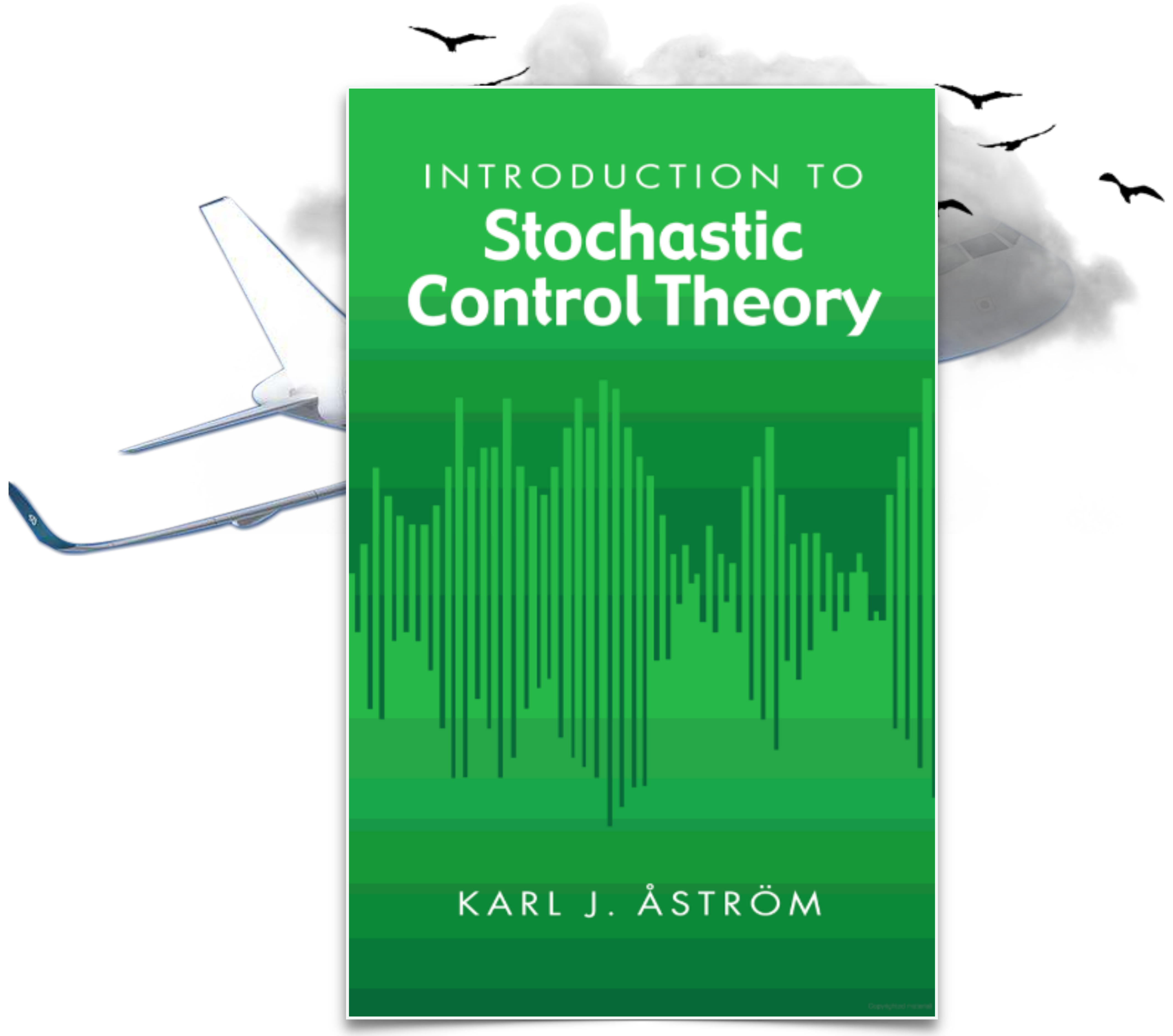
- High-level specification language
- Generate correct-by-construction embedded code
- Industrial tool: ANSYS Scade

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

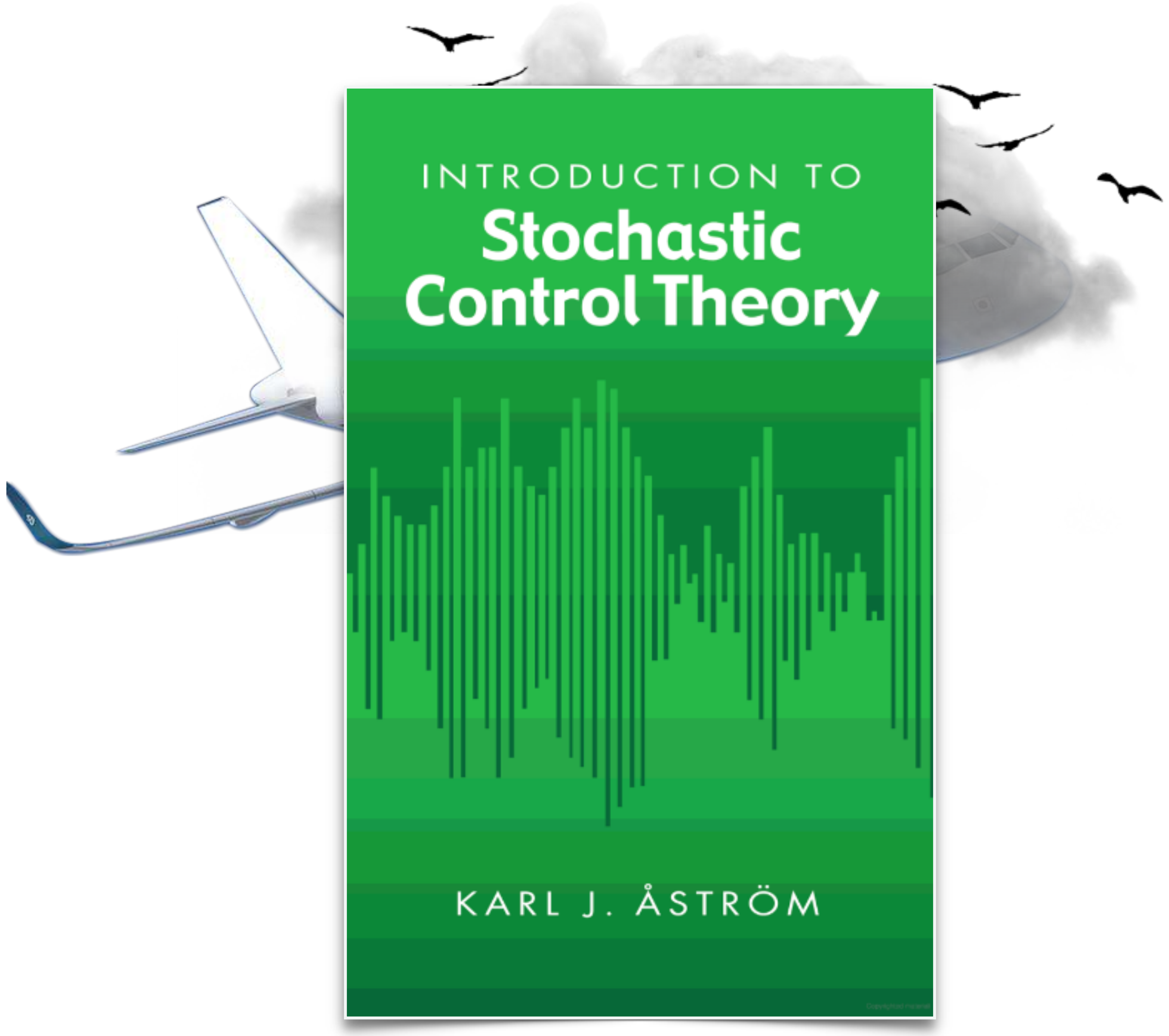
- Noisy environment, perceived through noisy sensors
- Interaction with other autonomous entities

Existing approaches

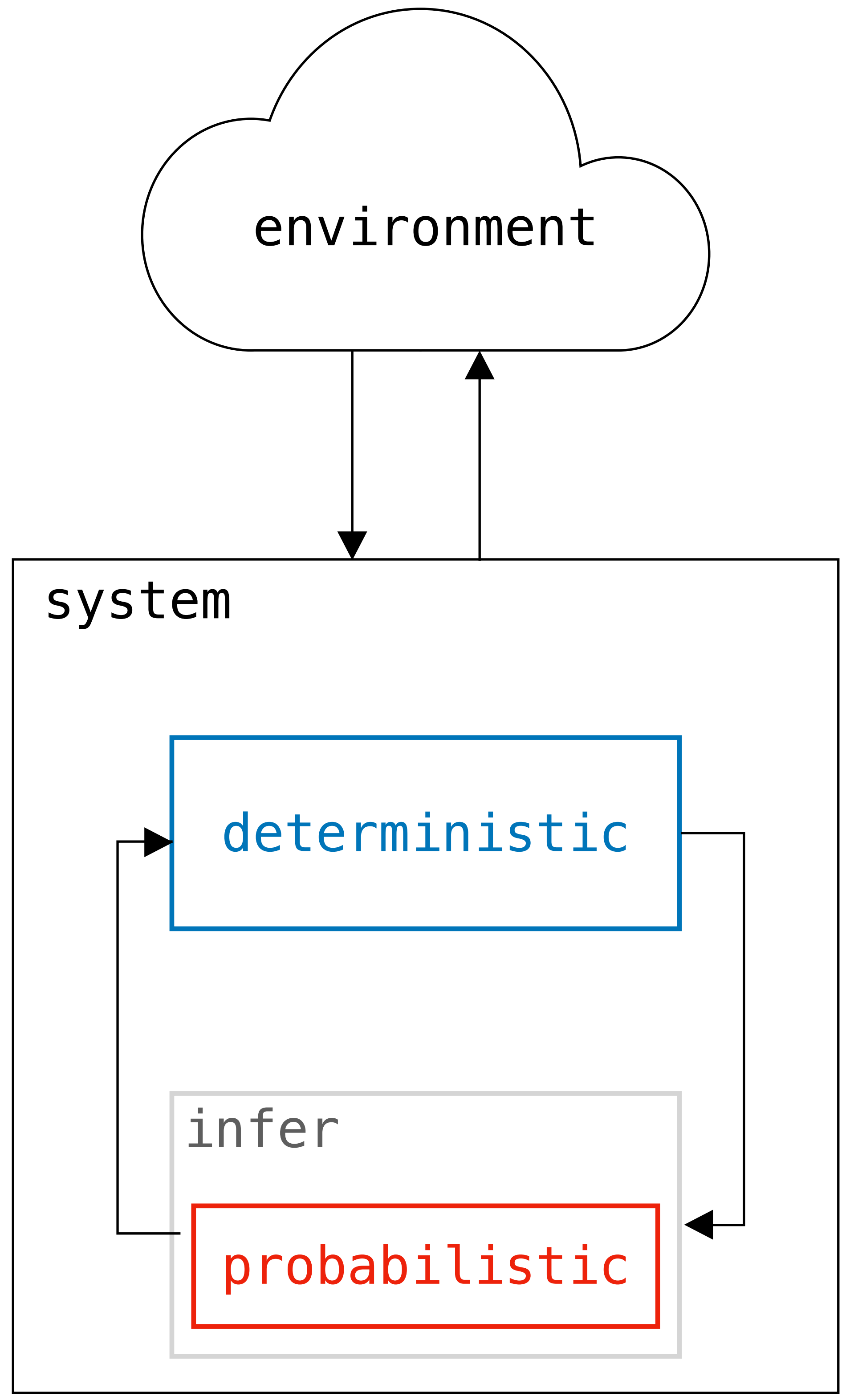
- Manually implement stochastic controller: *Can be error prone*
- Offline statistical tests: *Requires up-to-date offline data*

Reactive Probabilistic Programming

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

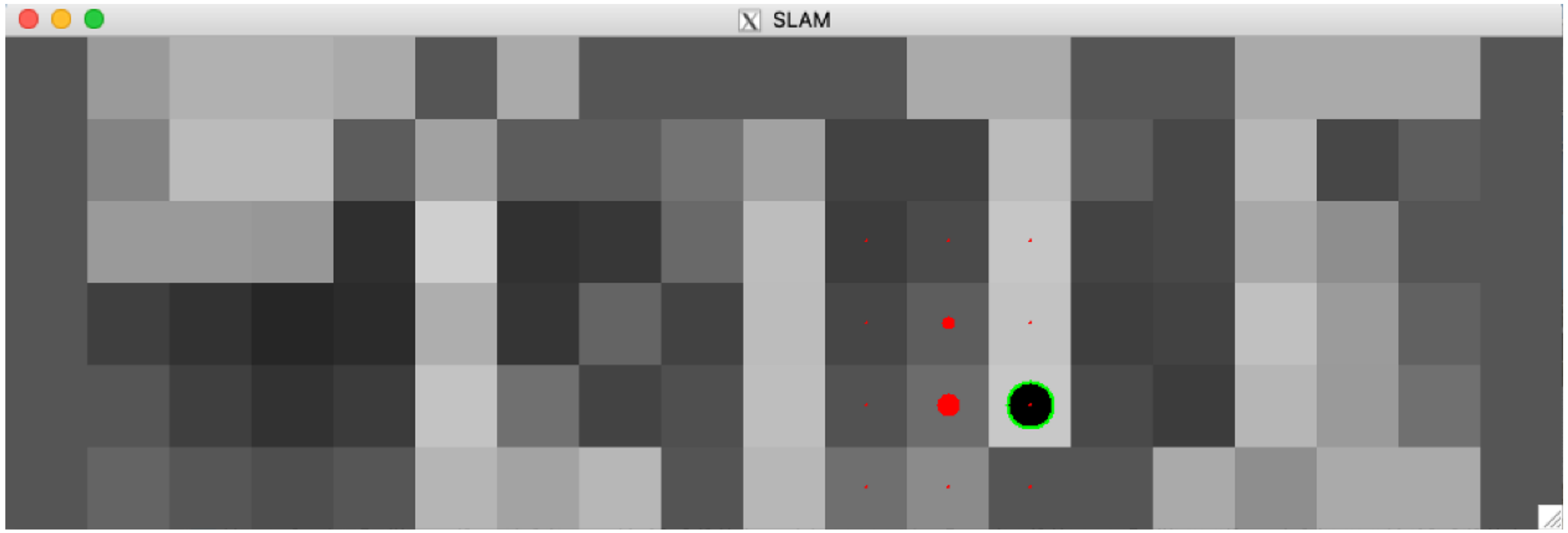


Reactive Probabilistic Programming (Demo)



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:

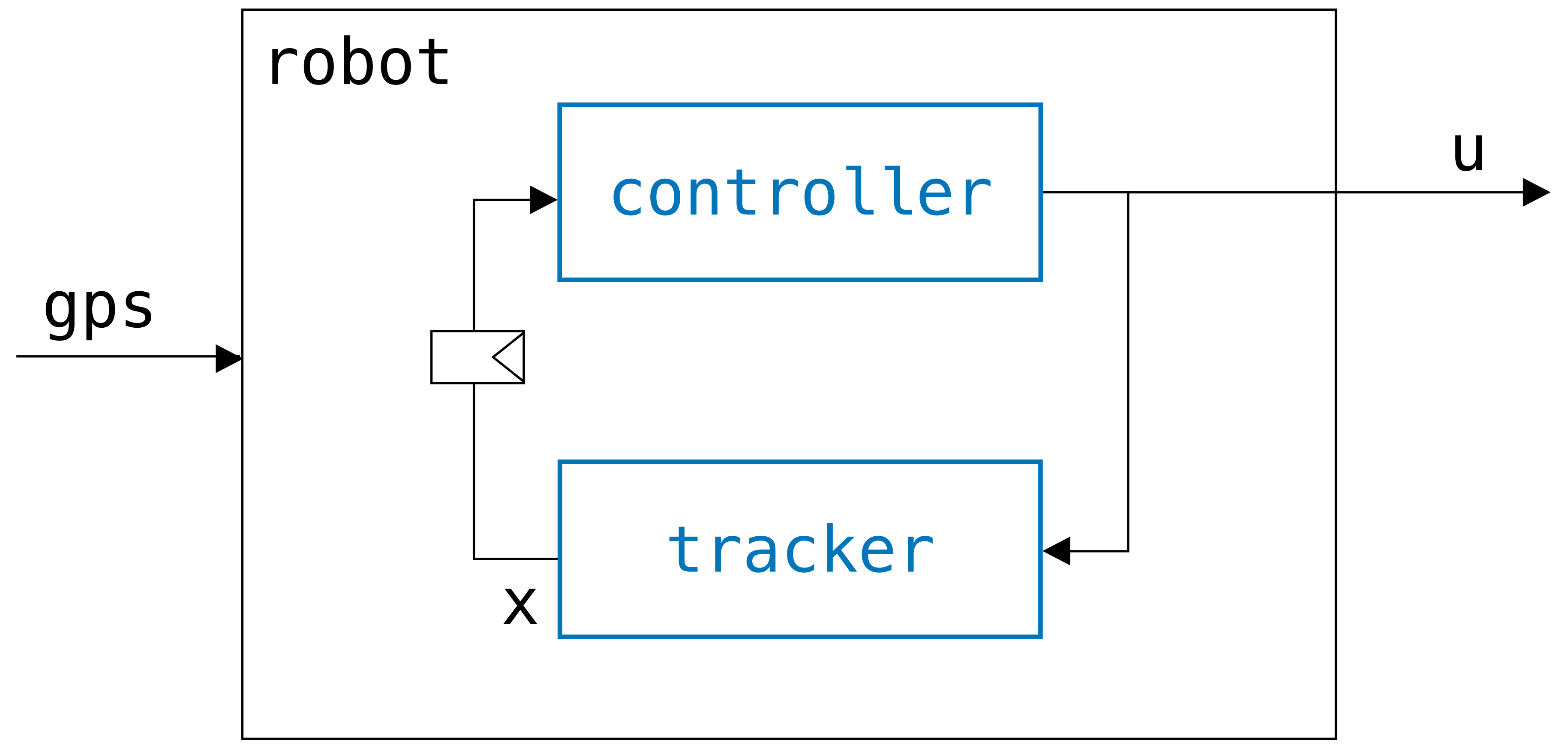
- Move to the next position
- Observe the color of the ground
- Use inferred position to compute next command

●	`exact position + color sensor`
■	`estimated color of a map cell`
🔴	`estimated position`

Reactive systems

Synchronous data-flow languages and block diagrams

- Signal: stream of values
- System: stream processor



```
let node robot (gps) = u where
  rec u = controller (x0 → pre x)
  and x = tracker (u, gps)
```

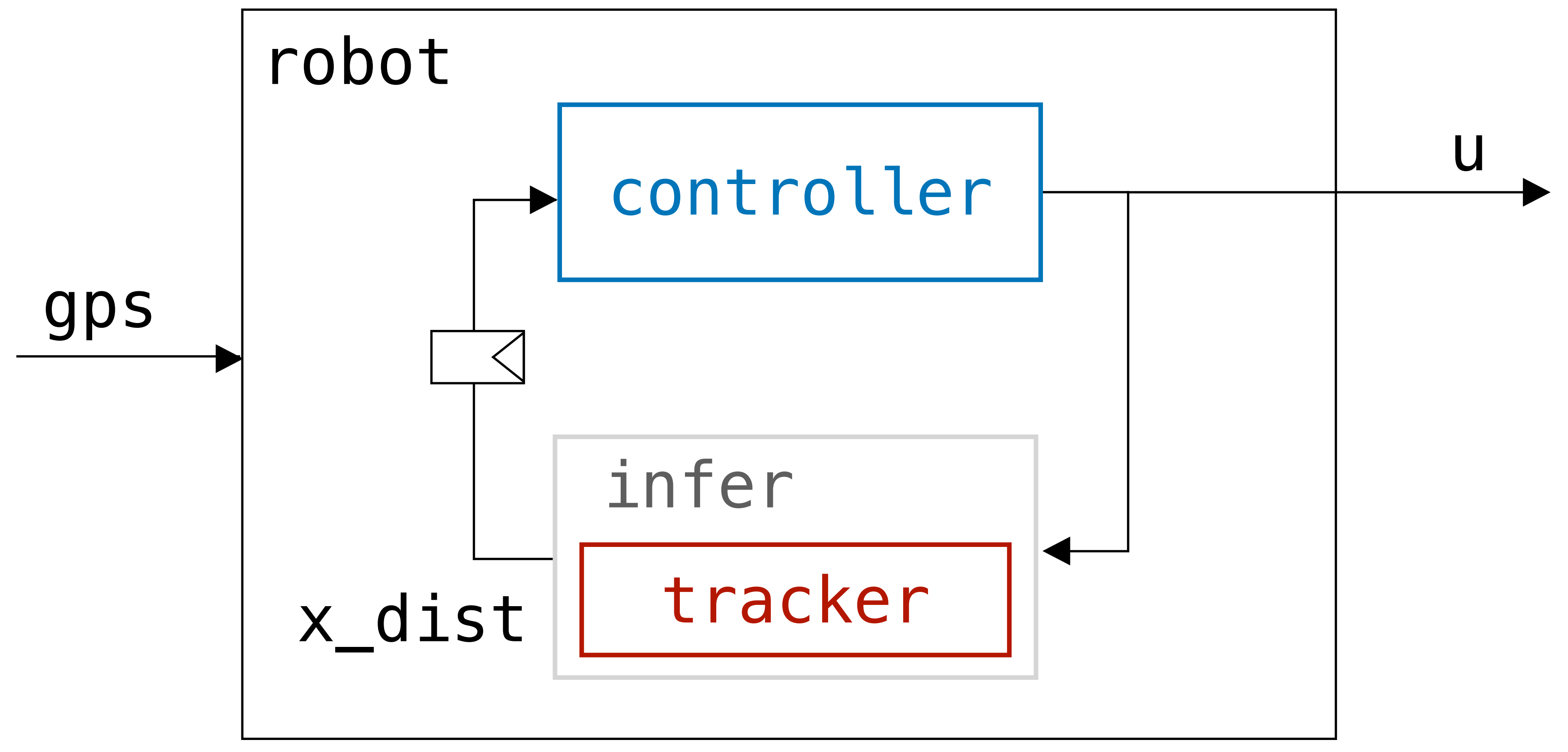
Reactive probabilistic systems

Synchronous data-flow languages and block diagrams

- Signal: stream of values
- System: stream processor

ProbZelus: add support to deal with uncertainty

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



```
let proba robot (gps) = u where
  rec u = controller (x0_dist → pre x_dist)
  and x_dist = infer tracker (u, gps)
```

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`

Bourke, Pouzet 2013

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

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

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

Bourke, Pouzet 2013

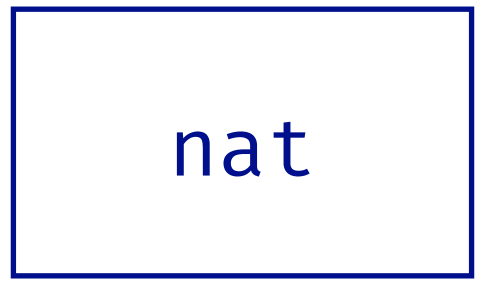
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

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

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

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



$t = 0$

Bourke, Pouzet 2013

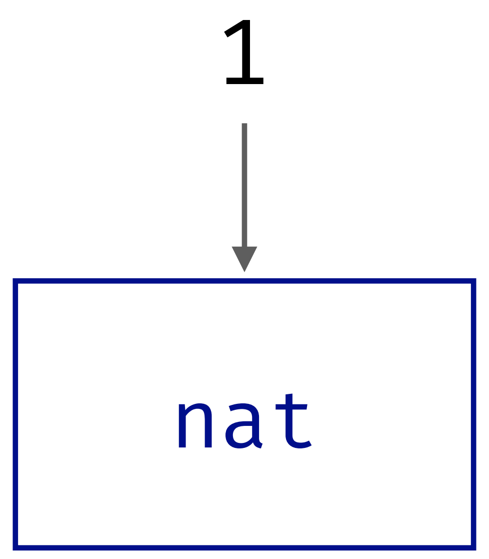
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

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

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

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



$t = 0$

Bourke, Pouzet 2013

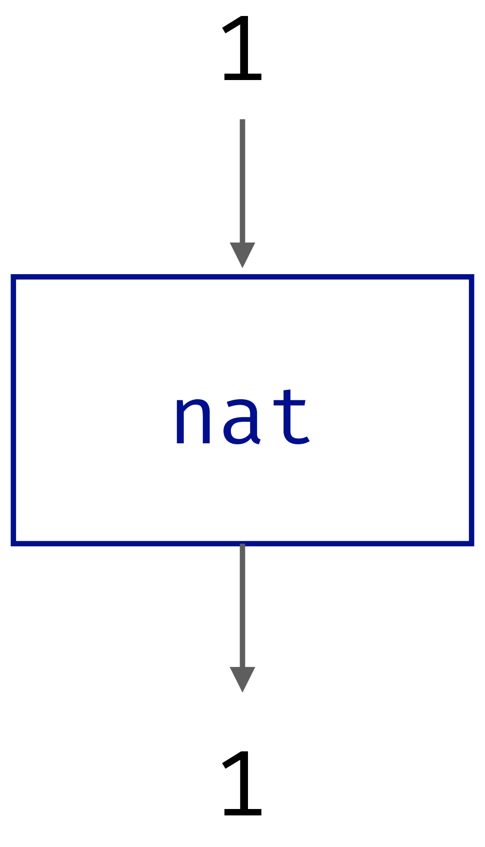
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

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

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

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



$t = 0$

Bourke, Pouzet 2013

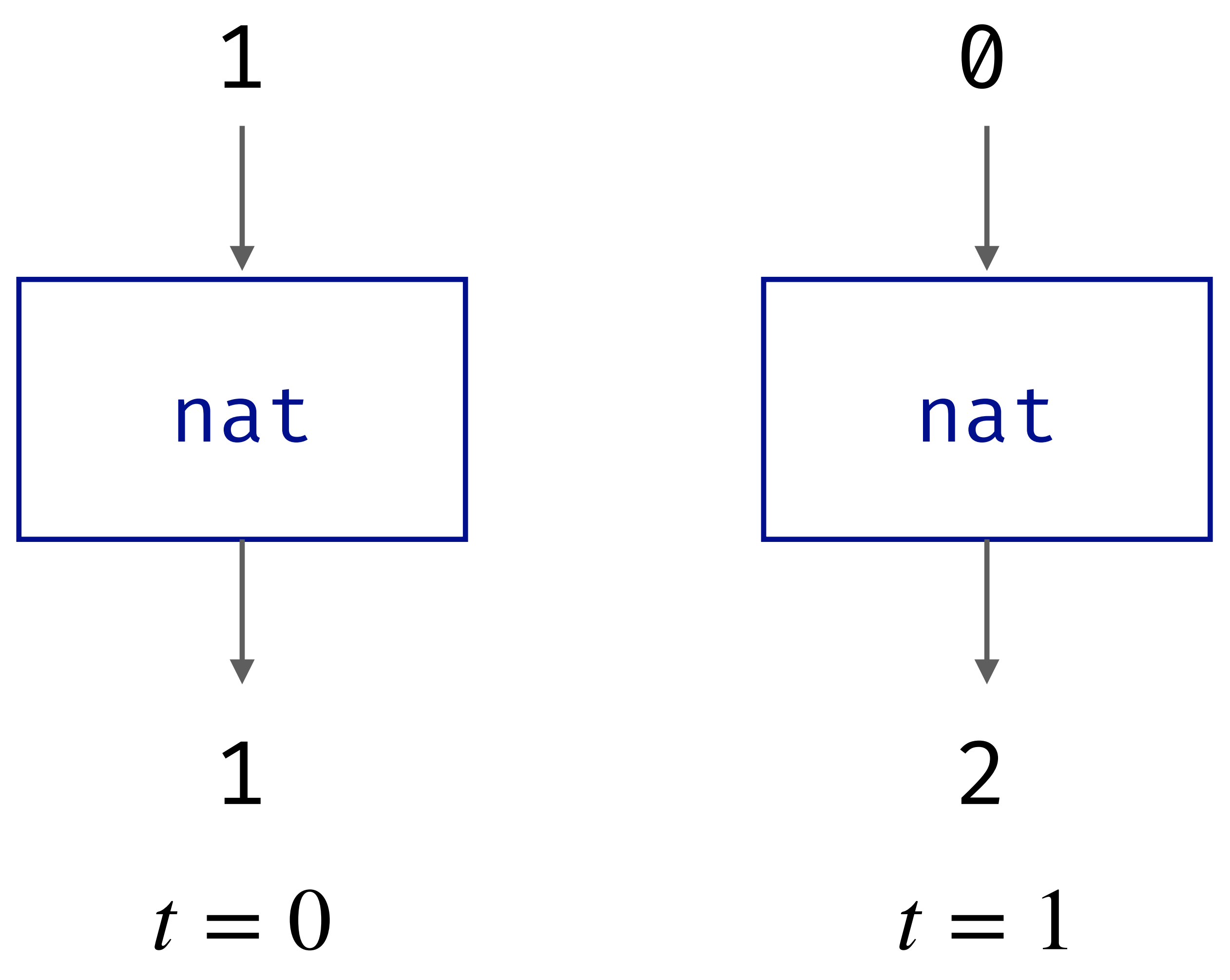
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

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

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

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



Bourke, Pouzet 2013

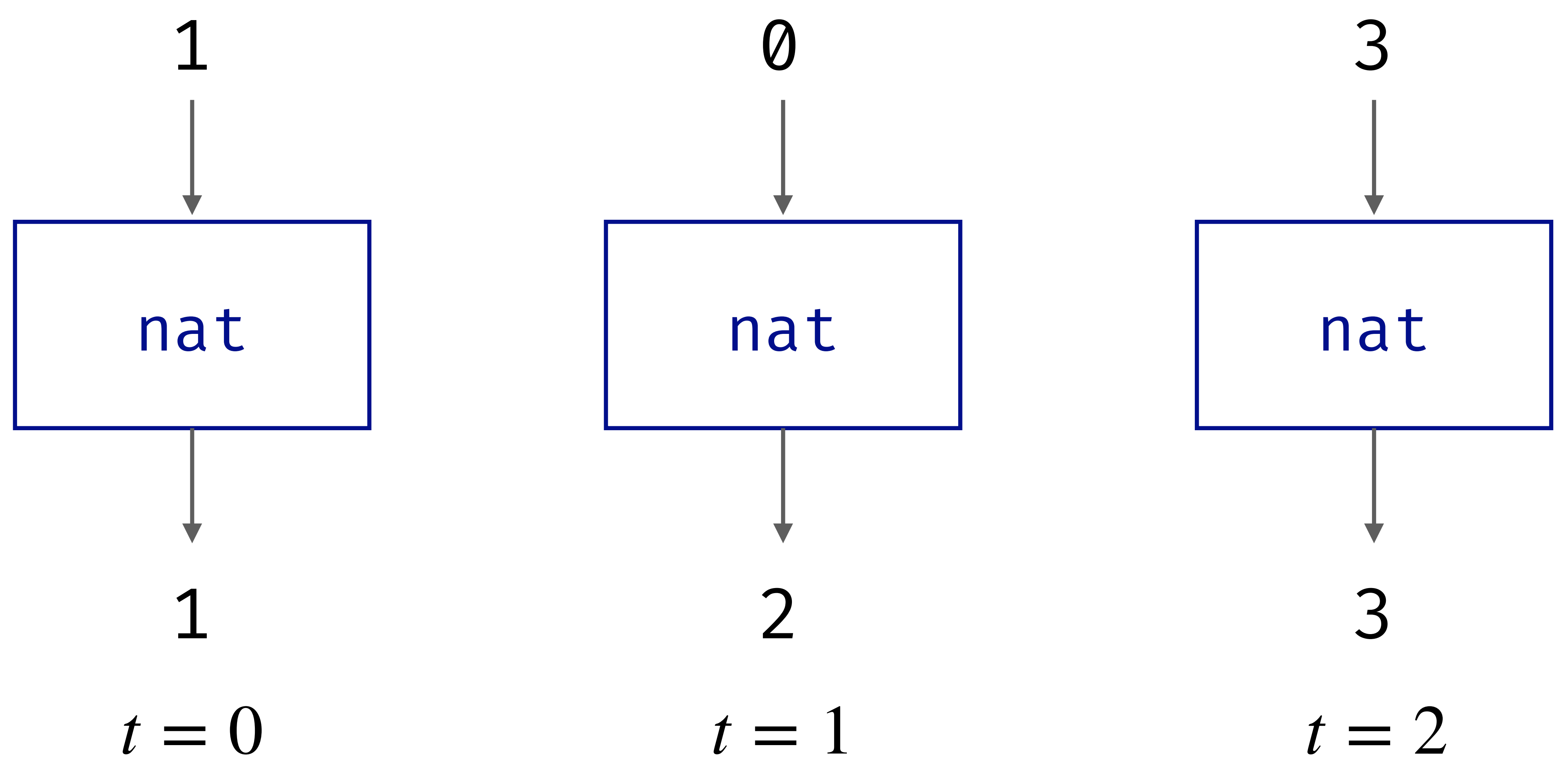
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

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

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

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



Bourke, Pouzet 2013

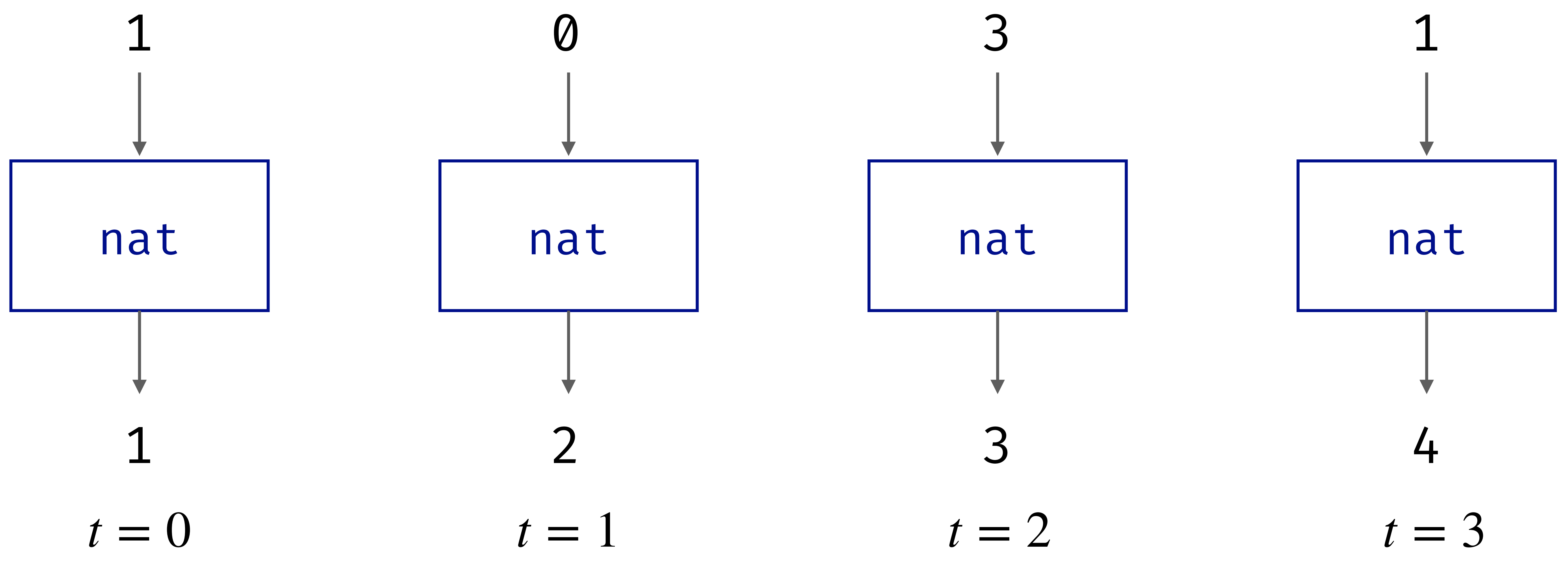
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

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

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

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



Bourke, Pouzet 2013

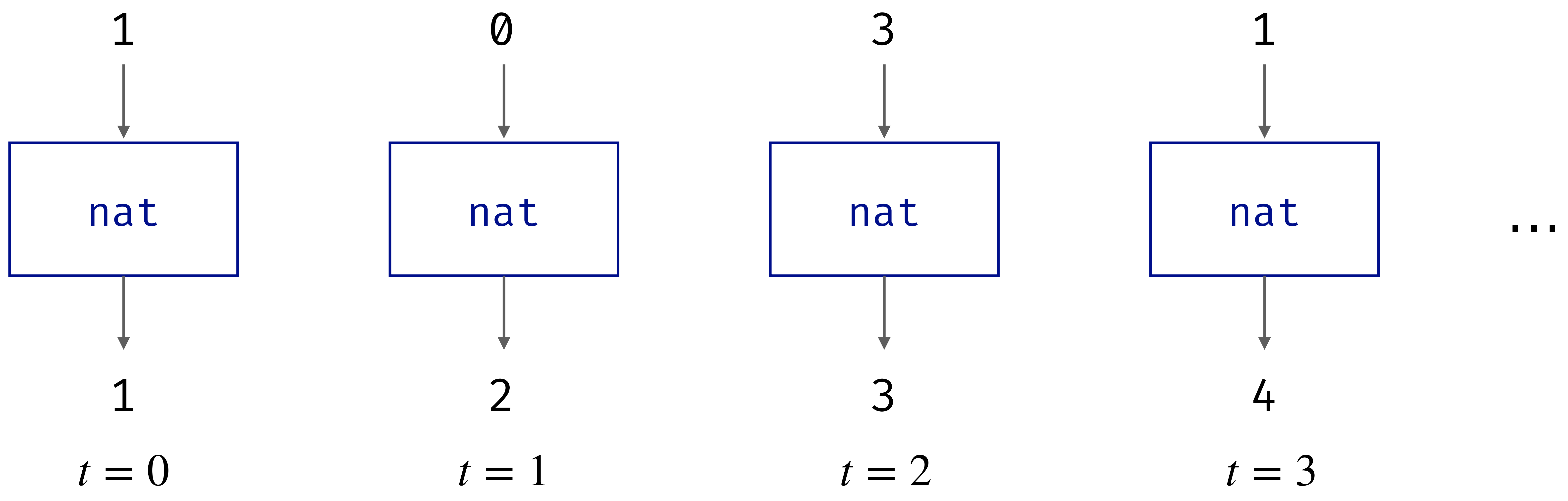
Lustre → Lucid Synchrone → Zelus → ProbZelus

Dataflow synchronous programming

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

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

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



Bourke, Pouzet 2013

Examples

```
let node n1 () = (o1, o2) where
  rec reset o1 = nat 0 every (0 fby o2 = 3)
  and reset o2 = nat 0 every (0 fby o1 = 2)
```

Examples

```
let node n1 () = (o1, o2) where
  rec reset o1 = nat 0 every (0 fby o2 = 3)
  and reset o2 = nat 0 every (0 fby o1 = 2)
```

```
o1 | 0 1 2 3 4 5 6 0 1 2  ...
o2 | 0 1 2 0 1 2 3 4 5 6  ...
```

Examples

```
let node n1 () = (o1, o2) where
  rec reset o1 = nat 0 every (0 fby o2 = 3)
  and reset o2 = nat 0 every (0 fby o1 = 2)
```

```
o1 | 0 1 2 3 4 5 6 0 1 2  ...
o2 | 0 1 2 0 1 2 3 4 5 6  ...
```

```
let node n2 () = o where
  rec o1, o2 = n1 ()
  and o = present (o2 = 0) → o1 else 1
```

Examples

```
let node n1 () = (o1, o2) where
  rec reset o1 = nat 0 every (0 fby o2 = 3)
  and reset o2 = nat 0 every (0 fby o1 = 2)
```

```
o1 | 0 1 2 3 4 5 6 0 1 2  ...
o2 | 0 1 2 0 1 2 3 4 5 6  ...
```

```
let node n2 () = o where
  rec o1, o2 = n1 ()
  and o = present (o2 = 0) → o1 else 1
```

```
o1 | 0 1 2 3 4 5 6 0 1 2  ...
o2 | 0 1 2 0 1 2 3 4 5 6  ...
 o | 1 1 1 3 1 1 1 1 1 1  ...
```

Examples

```
let node n1 () = (o1, o2) where
  rec reset o1 = nat 0 every (0 fby o2 = 3)
  and reset o2 = nat 0 every (0 fby o1 = 2)
```

```
o1 | 0 1 2 3 4 5 6 0 1 2  ...
o2 | 0 1 2 0 1 2 3 4 5 6  ...
```

```
let node n2 () = o where
  rec o1, o2 = n1 ()
  and o = present (o2 = 0) → o1 else 1
```

```
o1 | 0 1 2 3 4 5 6 0 1 2  ...
o2 | 0 1 2 0 1 2 3 4 5 6  ...
 o | 1 1 1 3 1 1 1 1 1 1  ...
```

```
let node n3 () = o where
  rec o1, o2 = n1 ()
  and o = present (o2 = 0) → last o + o1 init 1
```

Examples

```
let node n1 () = (o1, o2) where
  rec reset o1 = nat 0 every (0 fby o2 = 3)
  and reset o2 = nat 0 every (0 fby o1 = 2)
```

```
o1 | 0 1 2 3 4 5 6 0 1 2 ...
o2 | 0 1 2 0 1 2 3 4 5 6 ...
```

```
let node n2 () = o where
  rec o1, o2 = n1 ()
  and o = present (o2 = 0) → o1 else 1
```

```
o1 | 0 1 2 3 4 5 6 0 1 2 ...
o2 | 0 1 2 0 1 2 3 4 5 6 ...
 o | 1 1 1 3 1 1 1 1 1 1 ...
```

```
let node n3 () = o where
  rec o1, o2 = n1 ()
  and o = present (o2 = 0) → last o + o1 init 1
```

```
o1 | 0 1 2 3 4 5 6 0 1 2 ...
o2 | 0 1 2 0 1 2 3 4 5 6 ...
 o | 1 1 1 4 4 4 4 4 4 4 ...
```

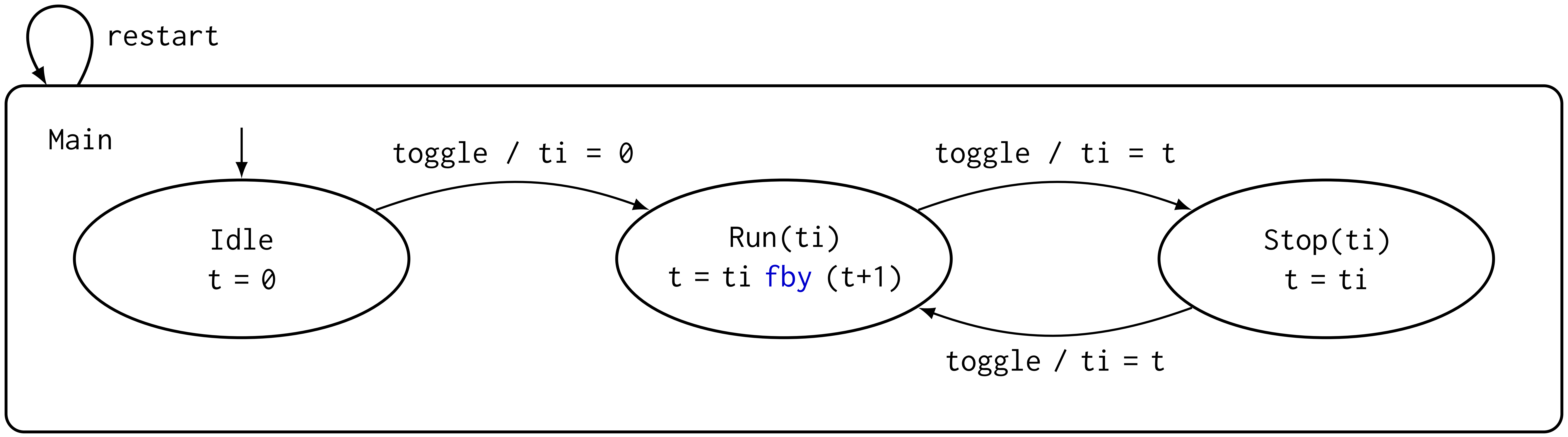
Exercises

pid.zls

```
let node integr (y, dt) = o where
  rec ???

let node deriv (y, dt) = o where
  rec ???

let node pid ((p, i, d), r, y, dt) = o where
  rec ???
```

Hierarchical automata

```
let node stopwatch (toggle, restart) = t where
  rec automaton
    | Main →
      do automaton
        | Idle     → do t = 0                until toggle then Run(0)
        | Run(ti)  → do t = ti fby (t + 1) until toggle then Stop(t)
        | Stop(ti) → do t = ti               until toggle then Run(t)
      end
      until restart then Main
```



Demo

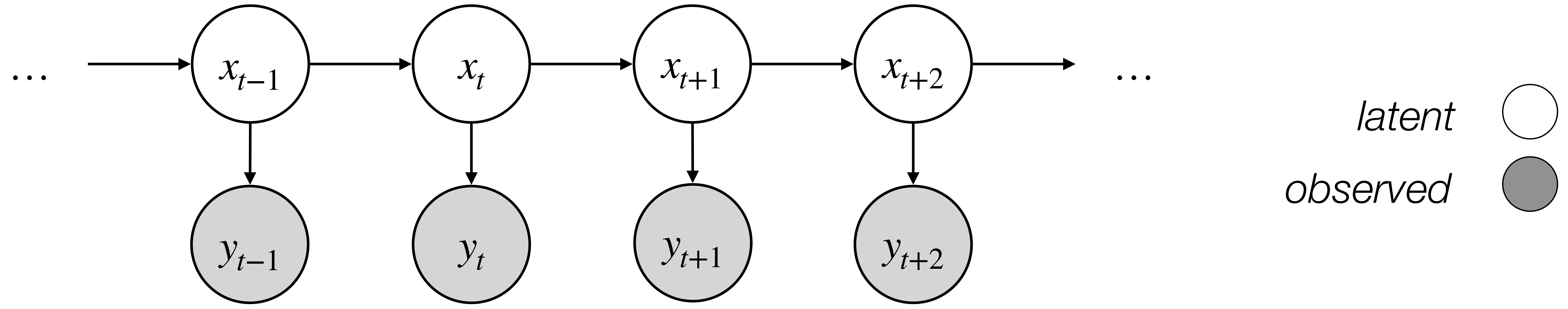
ProbZelus

Reactive Probabilistic Programming

Reactive probabilistic programming

Probabilistic constructs

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

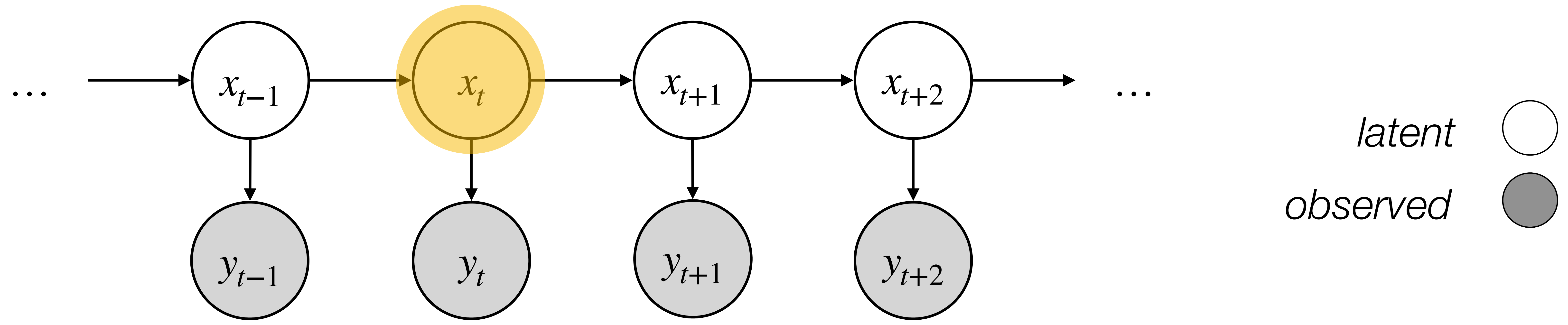


```
let proba tracker (y) = x where
    rec x = x0 → sample(mv_gaussian(f *@ (pre x), q))
    and () = observe(mv_gaussian(h *@ x, r), y)
```

Reactive probabilistic programming

Probabilistic constructs

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

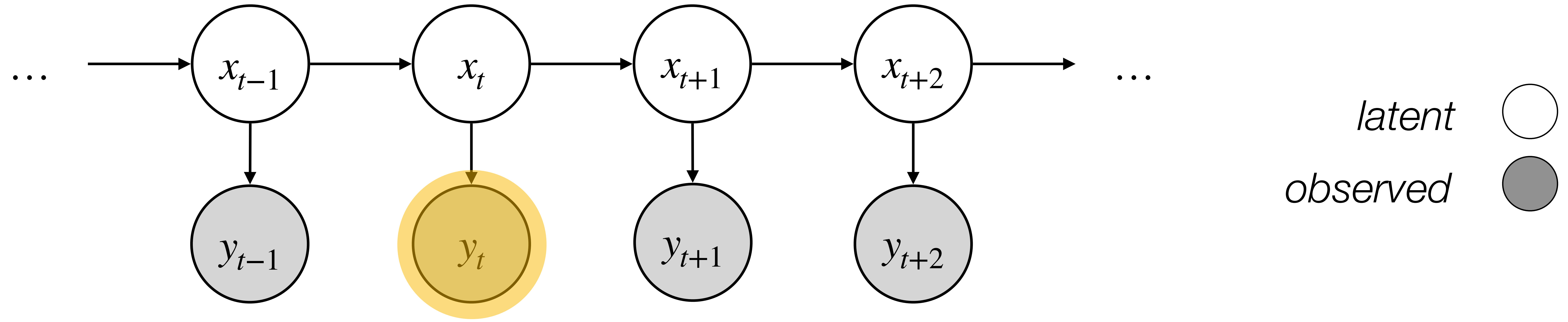


```
let proba tracker (y) = x where
  rec x = x0 → sample(mv_gaussian(f *@ (pre x), q))
  and () = observe(mv_gaussian(h *@ x, r), y)
```

Reactive probabilistic programming

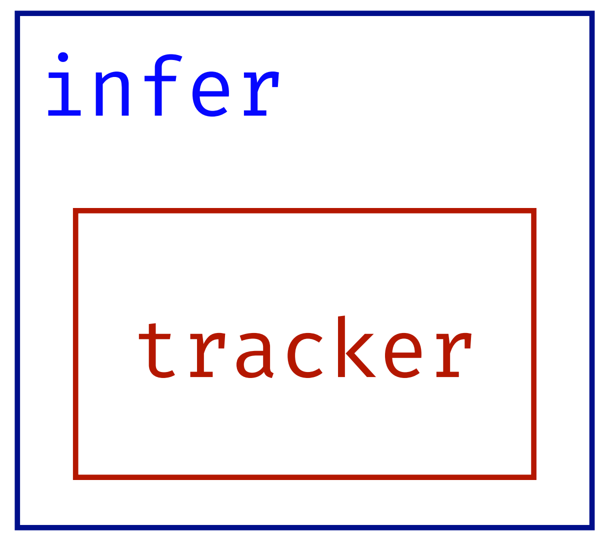
Probabilistic constructs

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



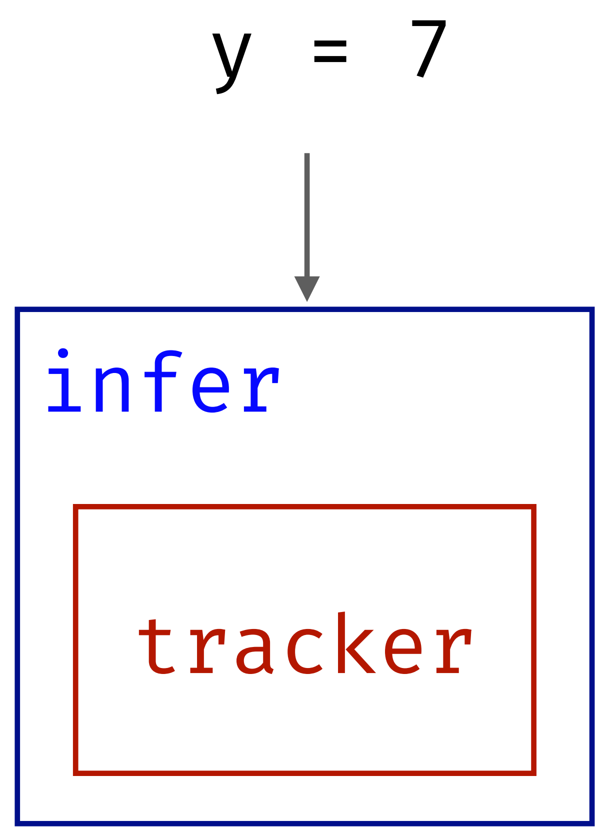
```
let proba tracker (y) = x where
    rec x = x0 → sample(mv_gaussian(f *@ (pre x), q))
    and () = observe(mv_gaussian(h *@ x, r), y)
```

Reactive probabilistic programming



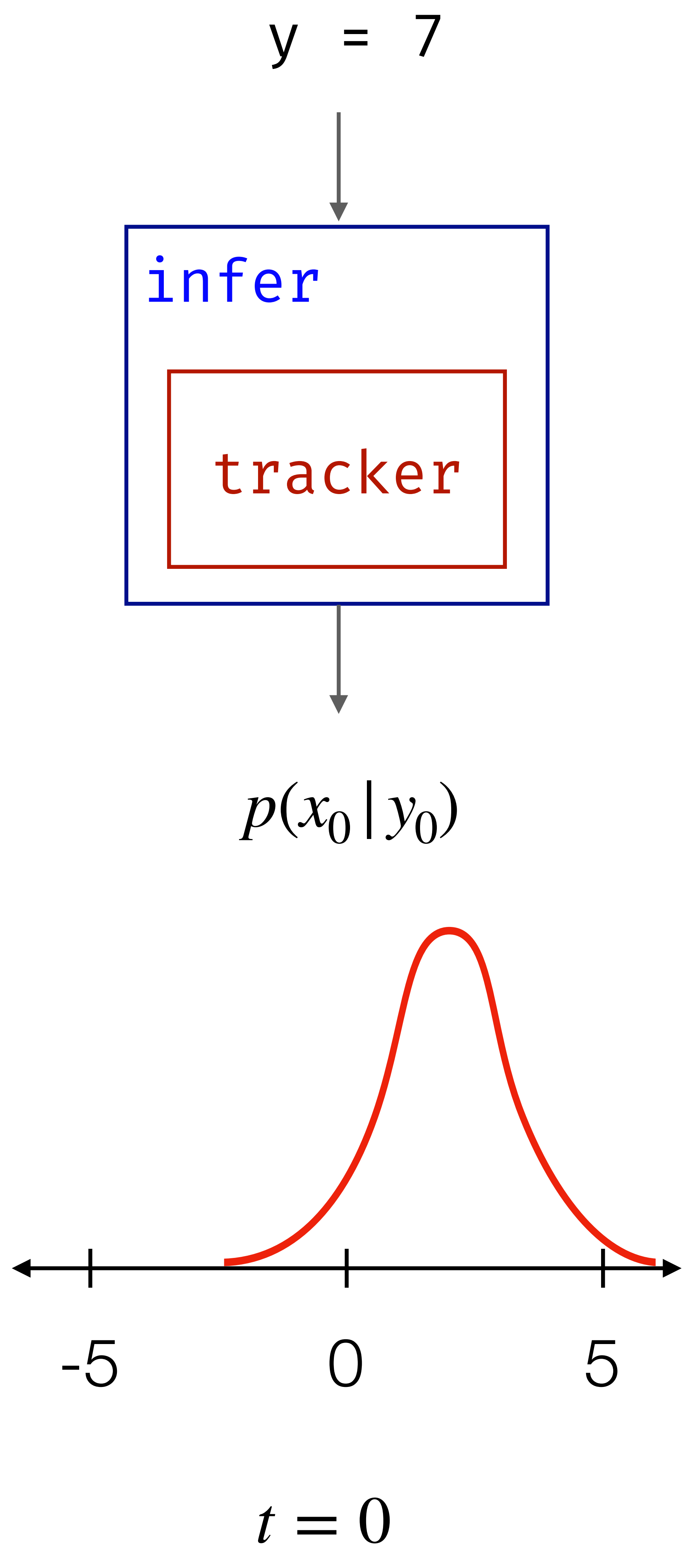
$t = 0$

Reactive probabilistic programming

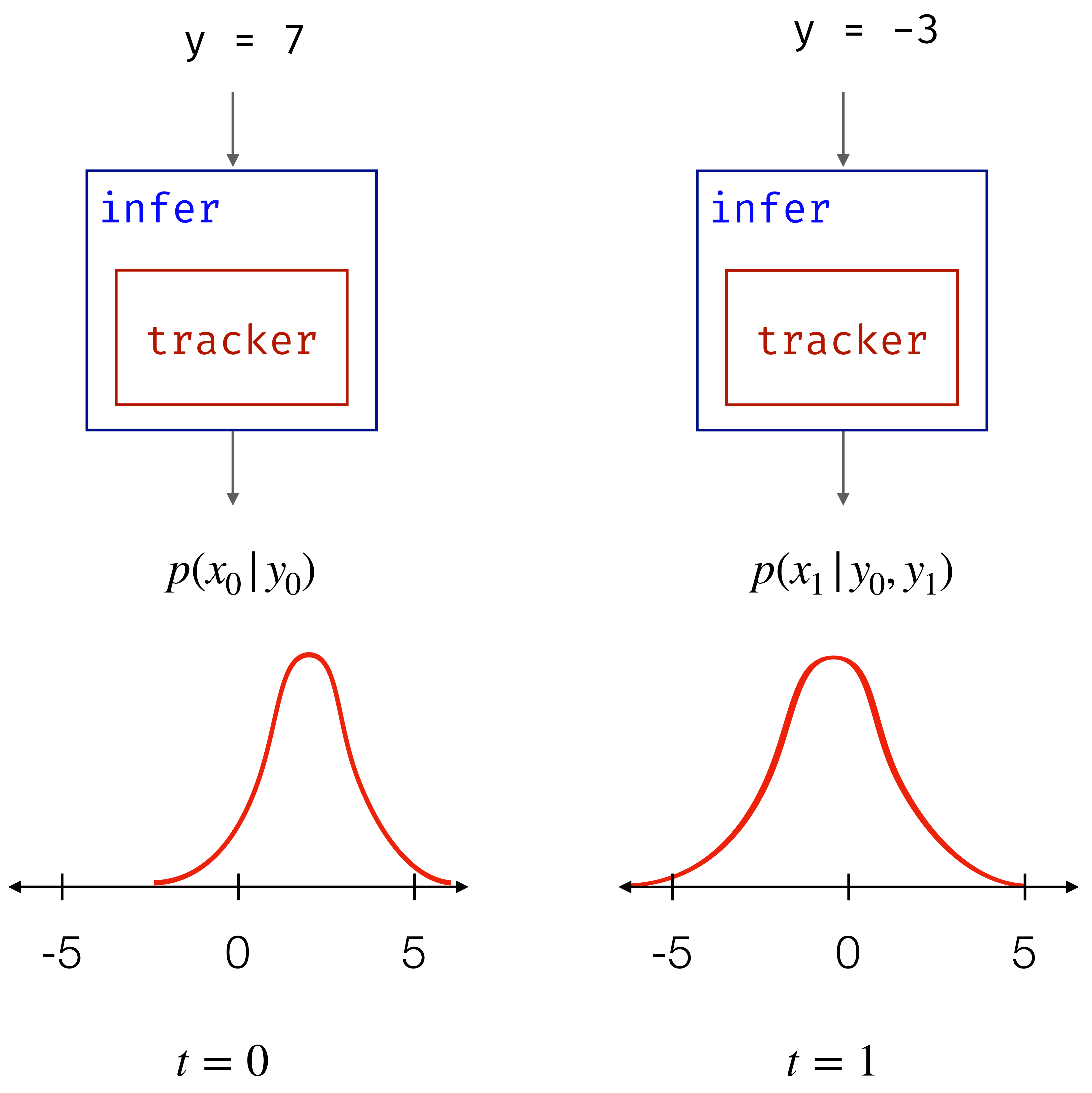


$t = 0$

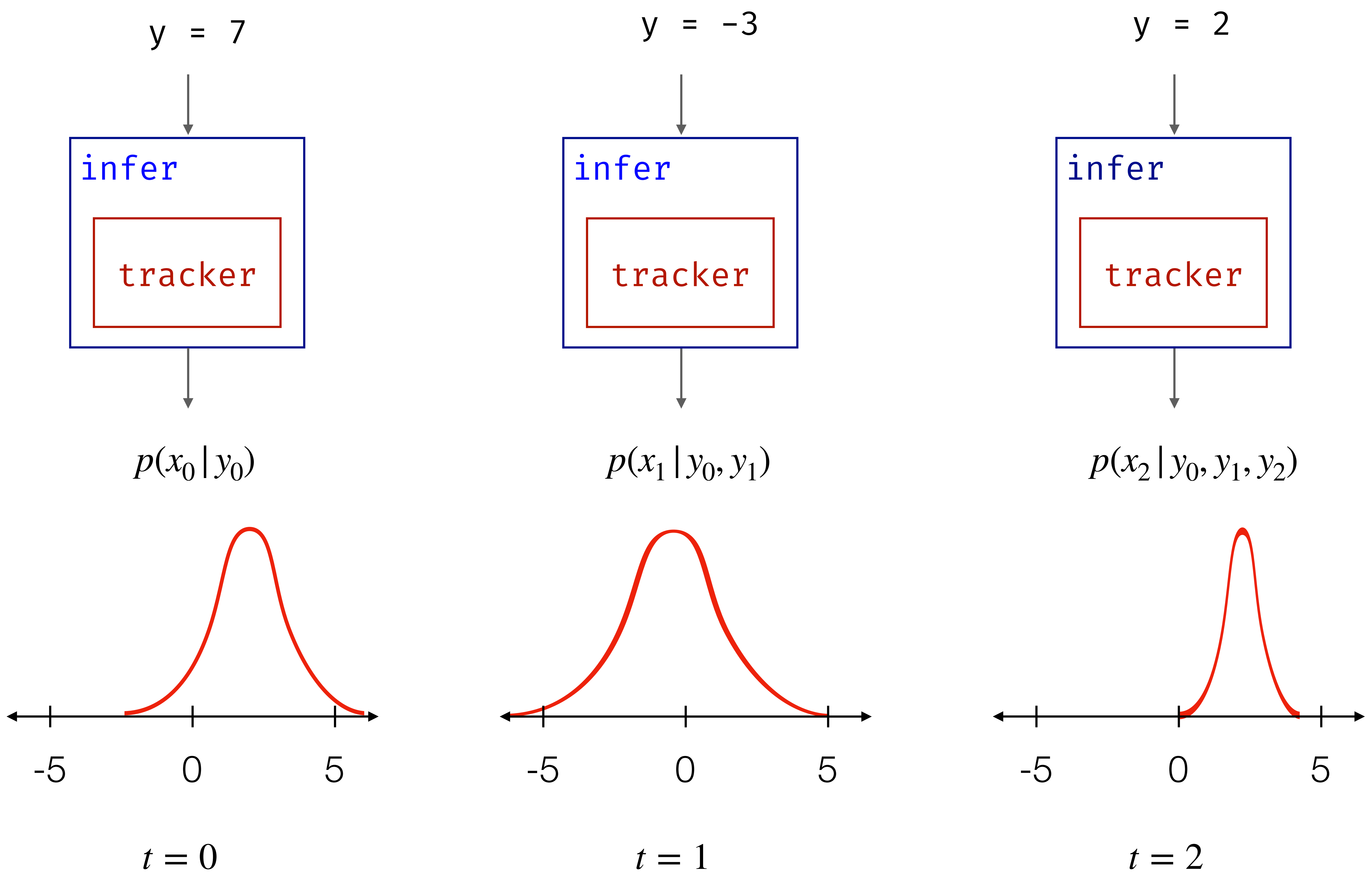
Reactive probabilistic programming



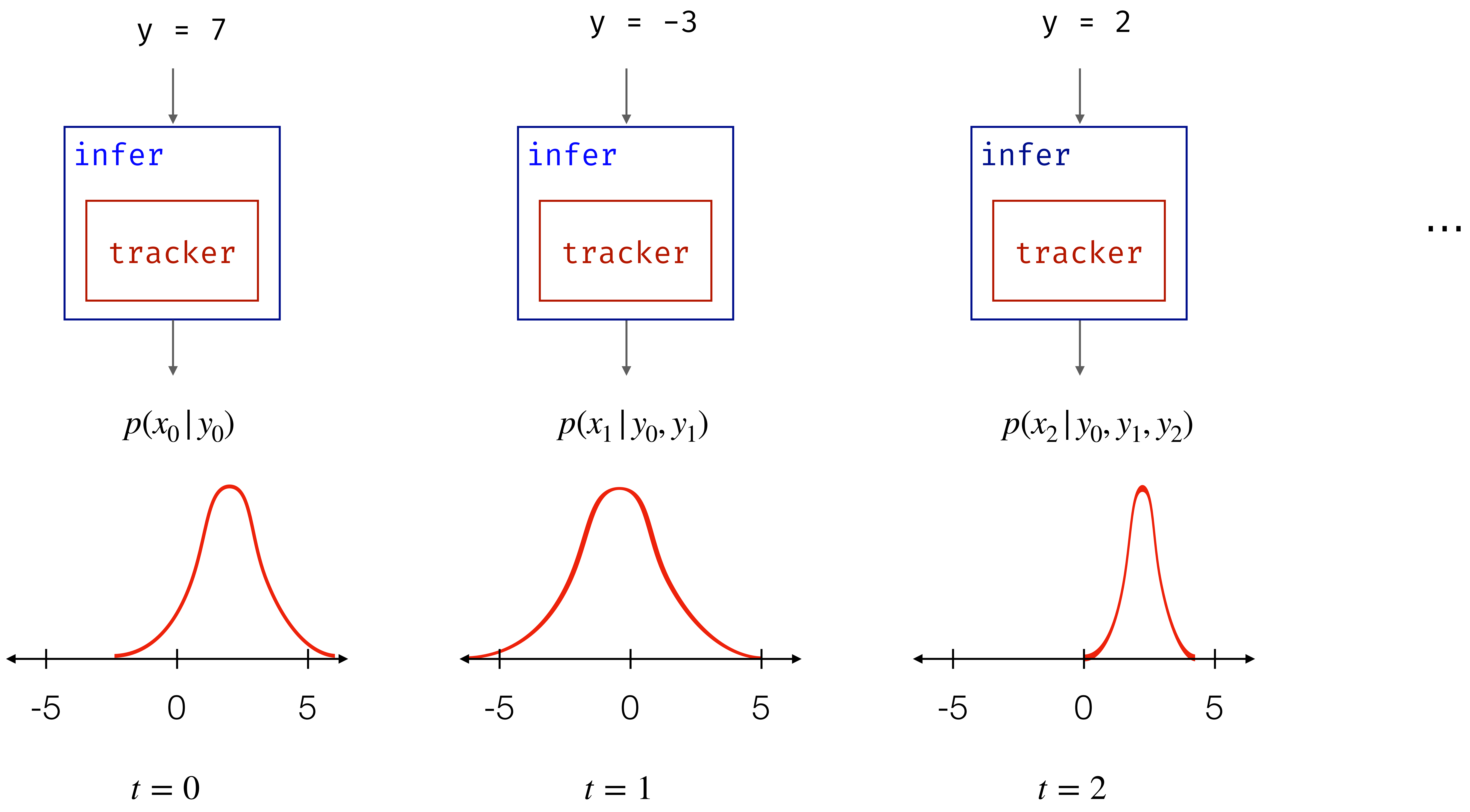
Reactive probabilistic programming



Reactive probabilistic programming



Reactive probabilistic programming



Demo

Co-iterative semantics

Schedule agnostic semantics

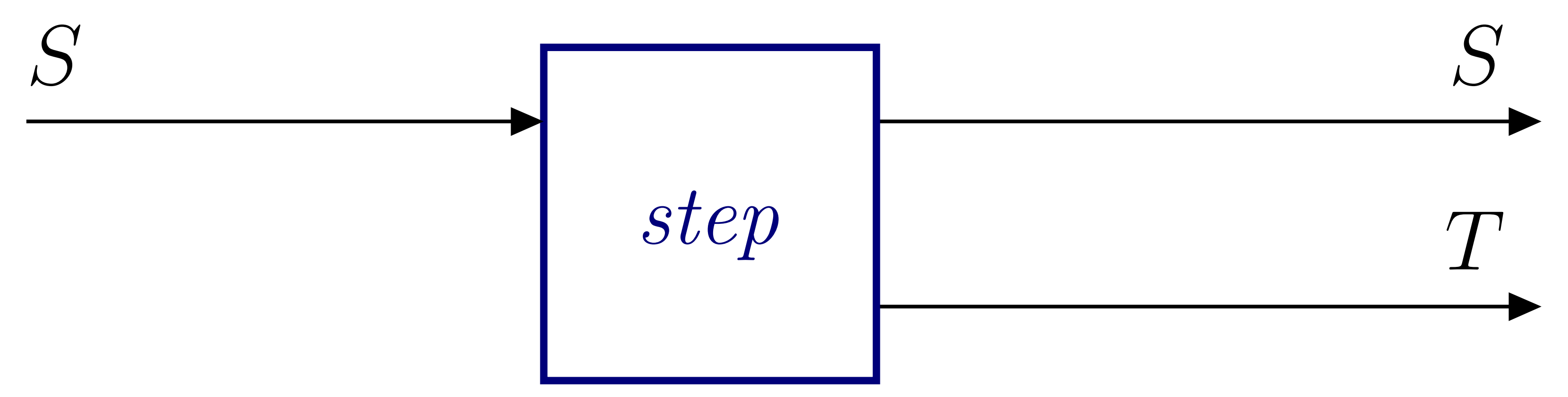
Deterministic semantics

allocation: S

transition: $S \to S \times T$

$$[\![e]\!] = [\![e]\!]^{\text{init}}_{\gamma}\ ,\ [\![e]\!]^{\text{step}}_{\gamma}$$

environment



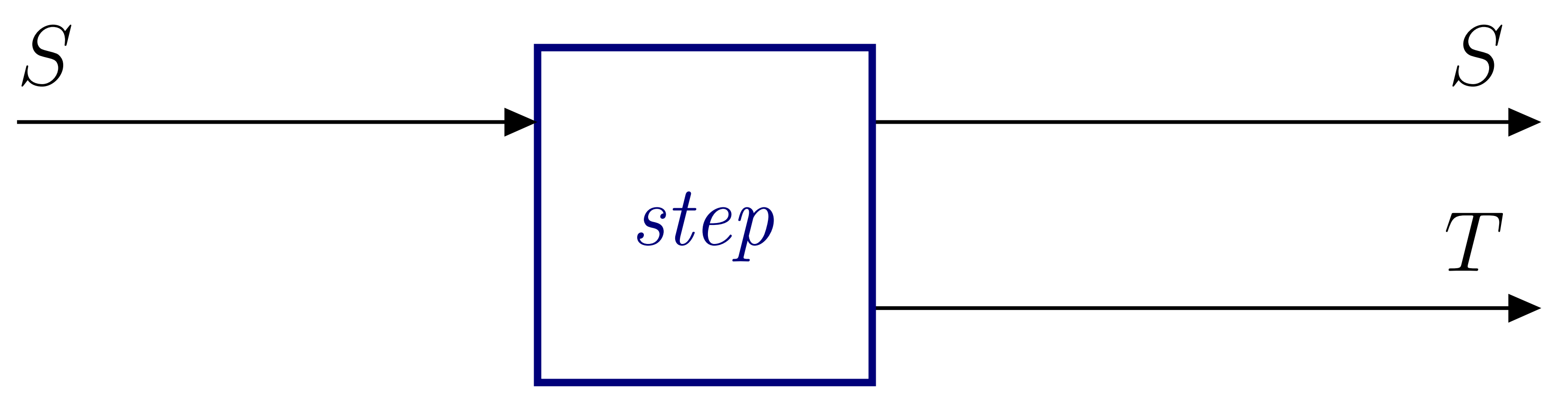
$$\begin{aligned}
[\![c]\!]^{\text{init}}_{\gamma} &= (\,) \\
[\![c]\!]^{\text{step}}_{\gamma}\,((\,)) &= (\,), c \\
\\
[\![x]\!]^{\text{init}}_{\gamma} &= (\,) \\
[\![x]\!]^{\text{step}}_{\gamma}\,((\,)) &= (\,), \gamma(x) \\
\\
[\![i \to \texttt{pre}\ e]\!]^{\text{init}}_{\gamma} &= (i, [\![e]\!]^{\text{init}}_{\gamma}) \\
[\![i \to \texttt{pre}\ e]\!]^{\text{step}}_{\gamma}\,(p, s) &= \textit{let } s', v = [\![e]\!]^{\text{step}}_{\gamma}\,(s) \textit{ in } (v, s'), p
\end{aligned}$$

Deterministic semantics

allocation: S transition: $S \to S \times T$

$$[\![e]\!] = [\![e]\!]^{\text{init}}_{\gamma}\,, [\![e]\!]^{\text{step}}_{\gamma}$$

environment



$[\![c]\!]^{\text{init}}_{\gamma}$

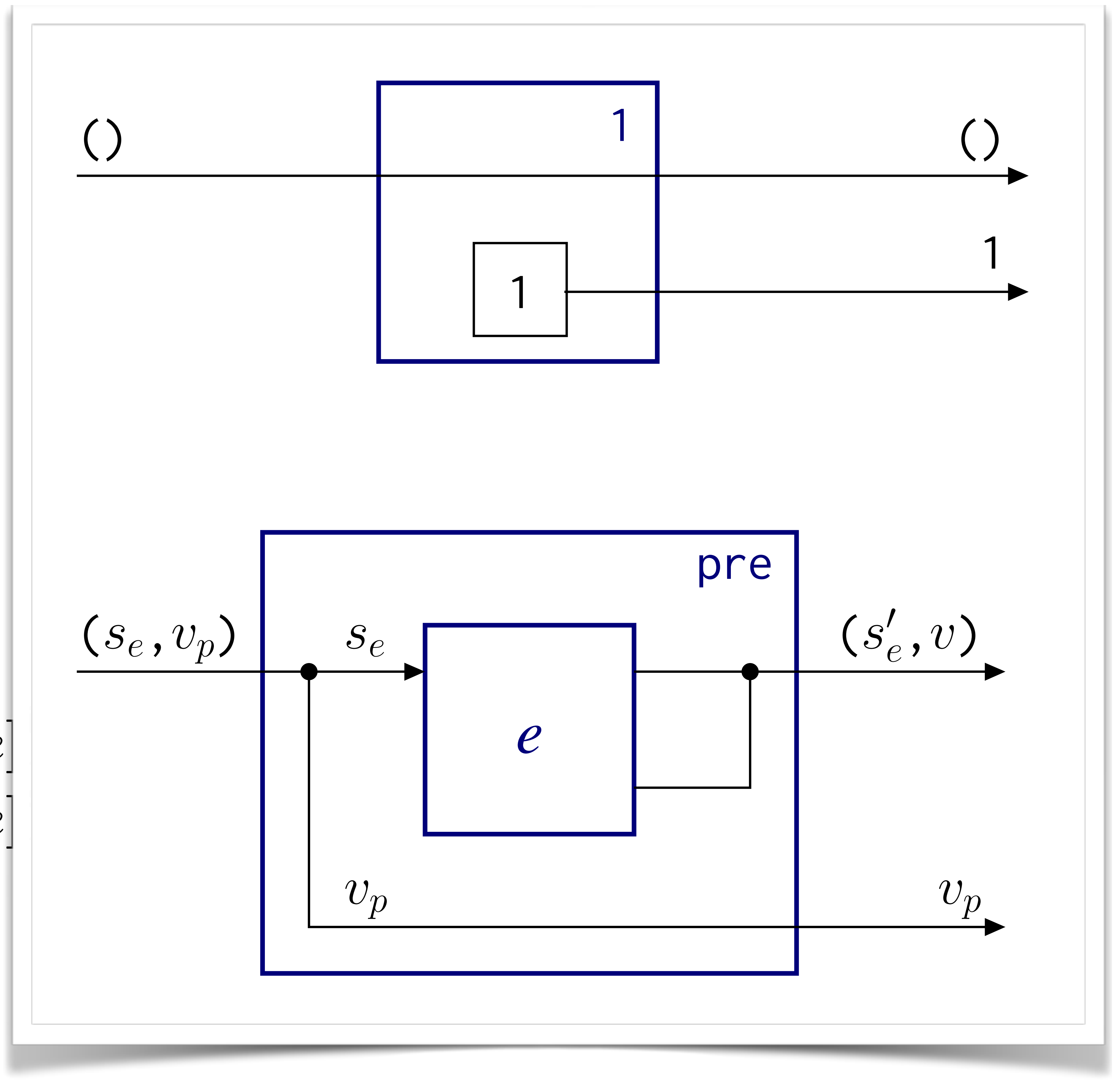
$[\![c]\!]^{\text{step}}_{\gamma}(\,(\,)\,)$

$[\![x]\!]^{\text{init}}_{\gamma}$

$[\![x]\!]^{\text{step}}_{\gamma}(\,(\,)\,)$

$[\![i \to$ `pre` $e]$

$[\![i \to$ `pre` $e]$



Deterministic equations

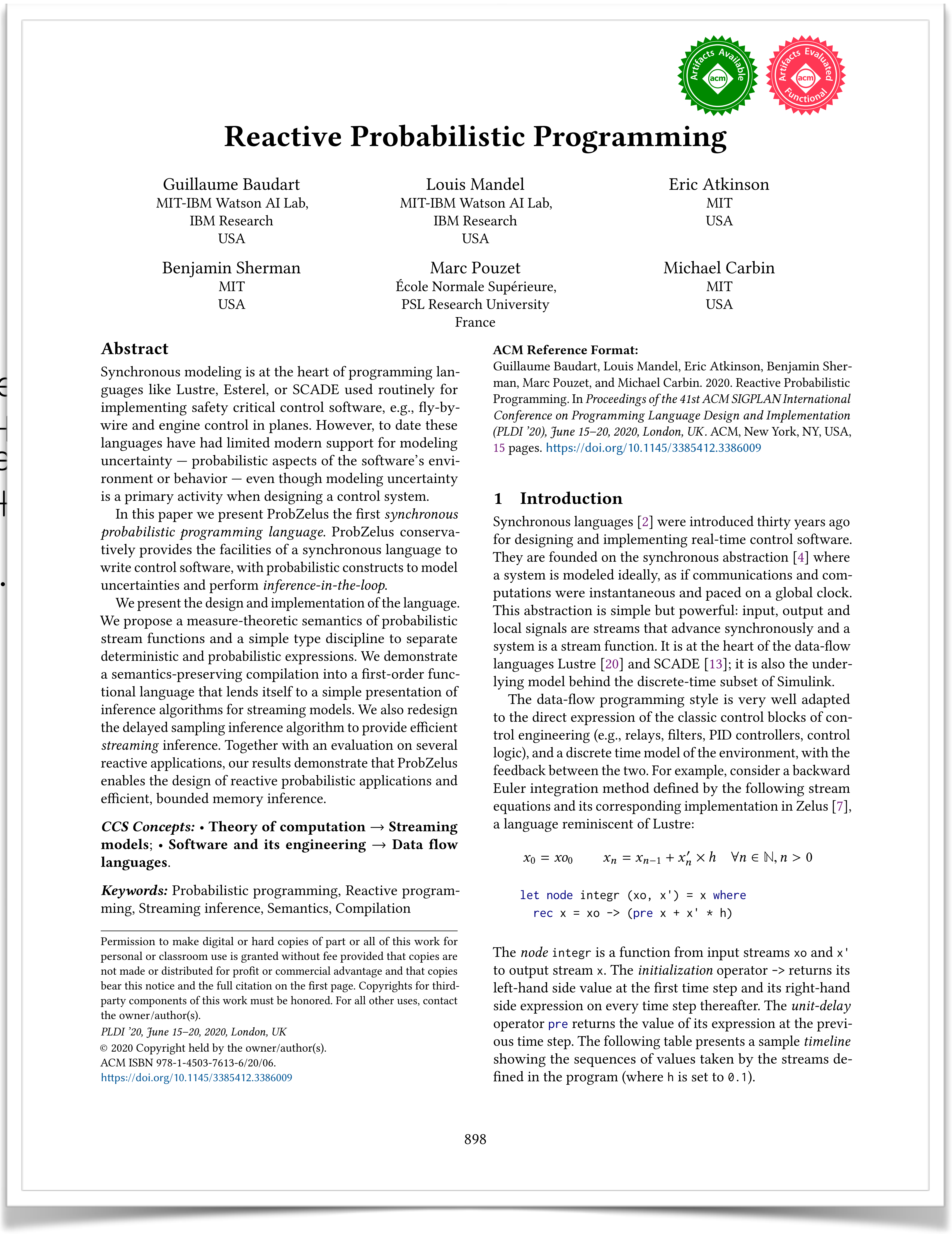
$$\left[\!\!\left[\begin{array}{l} e\ \texttt{where rec init}\ x = c \\ \qquad\qquad \texttt{and}\ x = e_x \\ \qquad\qquad \texttt{and}\ y = e_y \end{array}\right]\!\!\right]^{\text{init}}_{\gamma} = c, \left([\![e]\!]^{\text{init}}_{\gamma}, [\![e_x]\!]^{\text{init}}_{\gamma}, [\![e_y]\!]^{\text{init}}_{\gamma}\right)$$

$$\left[\!\!\left[\begin{array}{l} e\ \texttt{where rec init}\ x = c \\ \qquad\qquad \texttt{and}\ x = e_x \\ \qquad\qquad \texttt{and}\ y = e_y \end{array}\right]\!\!\right]^{\text{step}}_{\gamma} (p_x, (s, s_x, s_y)) = \begin{array}{l} \textit{let}\ s'_x, v_x = [\![e_x]\!]^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x]}(s_x)\ \textit{in} \\ \textit{let}\ s'_y, v_y = [\![e_y]\!]^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x, x\leftarrow v_x]}(s_y)\ \textit{in} \\ \textit{let}\ s', v = [\![e]\!]^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x, x\leftarrow v_x, y\leftarrow v_y]}(s)\ \textit{in} \\ (v_x, (s', s'_x, s'_y)) \end{array}$$

Deterministic equations

$$\left[\!\!\left[\begin{array}{l} e\ \texttt{where}\ \texttt{rec}\ \texttt{init}\ x = c \\ \qquad\quad \texttt{and}\ x = e_x \\ \qquad\quad \texttt{and}\ y = e_y \end{array}\right]\!\!\right]^{\text{init}}_{\gamma} = c, \left([\![e]\!]^{\text{init}}_{\gamma}, [\![e_x]\!]^{\text{init}}_{\gamma}, \right.$$

$$\left[\!\!\left[\begin{array}{l} e\ \texttt{where}\ \texttt{rec}\ \texttt{init}\ x = c \\ \qquad\quad \texttt{and}\ x = e_x \\ \qquad\quad \texttt{and}\ y = e_y \end{array}\right]\!\!\right]^{\text{step}}_{\gamma} (p_x, (s, s_x, s_y)) = \begin{array}{l} \textit{let } s'_x, v_x = [\![e_x]\!]^{\text{ste}}_{\gamma+} \\ \textit{let } s'_y, v_y = [\![e_y]\!]^{\text{ste}}_{\gamma+} \\ \textit{let } s', v = [\![e]\!]^{\text{step}}_{\gamma+[x.} \\ (v_x, (s', s'_x, s'_y)) \end{array}$$



Reactive Probabilistic Programming

Guillaume Baudart
MIT-IBM Watson AI Lab,
IBM Research
USA

Louis Mandel
MIT-IBM Watson AI Lab,
IBM Research
USA

Eric Atkinson
MIT
USA

Benjamin Sherman
MIT
USA

Marc Pouzet
École Normale Supérieure,
PSL Research University
France

Michael Carbin
MIT
USA

Abstract

Synchronous modeling is at the heart of programming languages like Lustre, Esterel, or SCADE used routinely for implementing safety critical control software, e.g., fly-by-wire and engine control in planes. However, to date these languages have had limited modern support for modeling uncertainty — probabilistic aspects of the software's environment or behavior — even though modeling uncertainty is a primary activity when designing a control system.

In this paper we present ProbZelus the first *synchronous probabilistic programming language*. ProbZelus conservatively provides the facilities of a synchronous language to write control software, with probabilistic constructs to model uncertainties and perform *inference-in-the-loop*.

We present the design and implementation of the language. We propose a measure-theoretic semantics of probabilistic stream functions and a simple type discipline to separate deterministic and probabilistic expressions. We demonstrate a semantics-preserving compilation into a first-order functional language that lends itself to a simple presentation of inference algorithms for streaming models. We also redesign the delayed sampling inference algorithm to provide efficient *streaming* inference. Together with an evaluation on several reactive applications, our results demonstrate that ProbZelus enables the design of reactive probabilistic applications and efficient, bounded memory inference.

CCS Concepts: • **Theory of computation** → **Streaming models**; • **Software and its engineering** → **Data flow languages**.

Keywords: Probabilistic programming, Reactive programming, Streaming inference, Semantics, Compilation

PLDI '20, June 15–20, 2020, London, UK
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3386009

ACM Reference Format:
Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive Probabilistic Programming. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3386009

1 Introduction

Synchronous languages [2] were introduced thirty years ago for designing and implementing real-time control software. They are founded on the synchronous abstraction [4] where a system is modeled ideally, as if communications and computations were instantaneous and paced on a global clock. This abstraction is simple but powerful: input, output and local signals are streams that advance synchronously and a system is a stream function. It is at the heart of the data-flow languages Lustre [20] and SCADE [13]; it is also the underlying model behind the discrete-time subset of Simulink.

The data-flow programming style is very well adapted to the direct expression of the classic control blocks of control engineering (e.g., relays, filters, PID controllers, control logic), and a discrete time model of the environment, with the feedback between the two. For example, consider a backward Euler integration method defined by the following stream equations and its corresponding implementation in Zelus [7], a language reminiscent of Lustre:

$$x_0 = xo_0 \qquad x_n = x_{n-1} + x'_n \times h \quad \forall n \in \mathbb{N}, n > 0$$

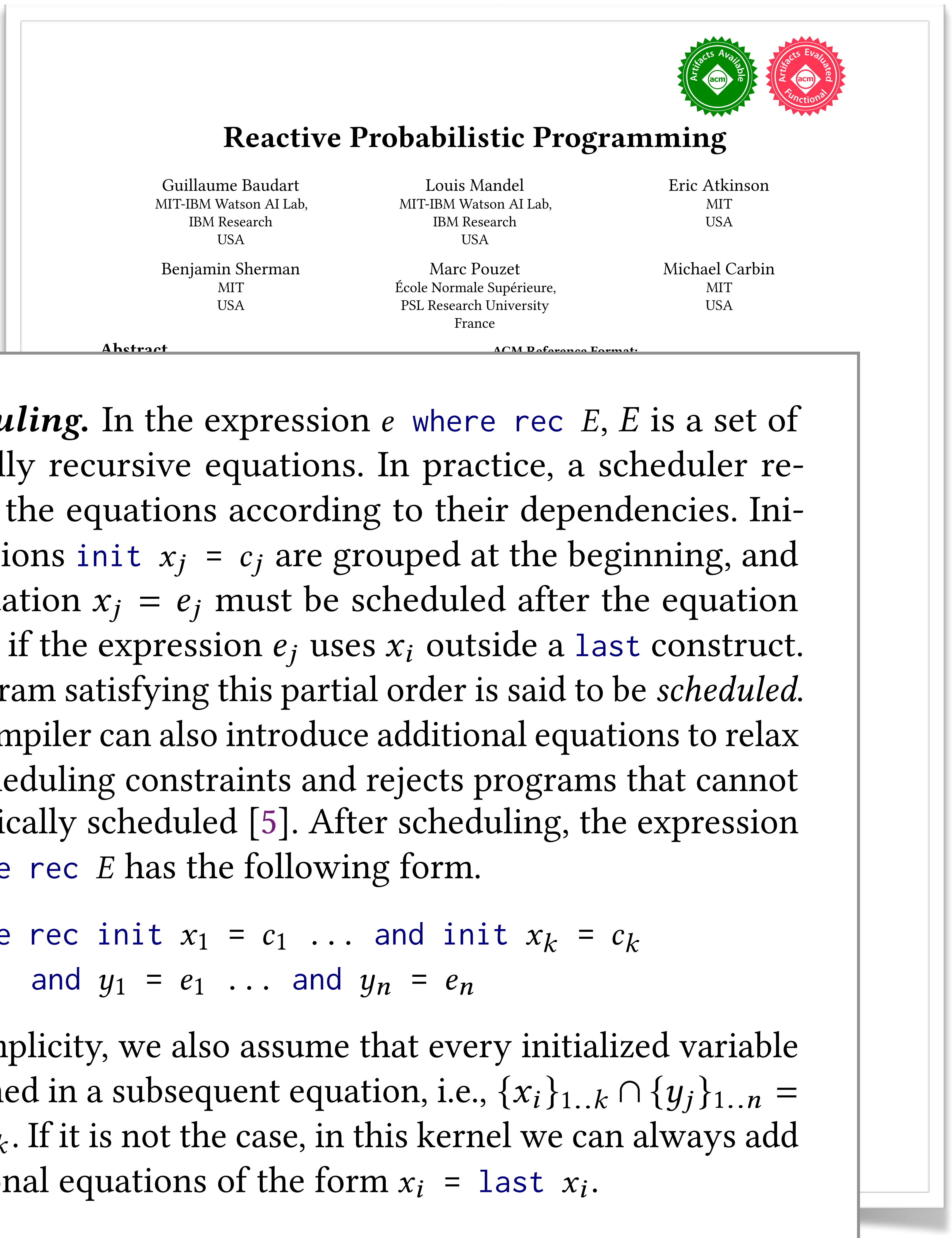
```
let node integr (xo, x') = x where
  rec x = xo -> (pre x + x' * h)
```

The *node* `integr` is a function from input streams `xo` and `x'` to output stream `x`. The *initialization* operator `->` returns its left-hand side value at the first time step and its right-hand side expression on every time step thereafter. The *unit-delay* operator `pre` returns the value of its expression at the previous time step. The following table presents a sample *timeline* showing the sequences of values taken by the streams defined in the program (where `h` is set to `0.1`).

Deterministic equations

$$\left\llbracket \begin{array}{l} e \texttt{ where rec init } x = c \\ \qquad\qquad \texttt{and } x = e_x \\ \qquad\qquad \texttt{and } y = e_y \end{array} \right\rrbracket^{\text{init}}_{\gamma} = c, \Big(\llbracket e \rrbracket^{\text{init}}_{\gamma}, \llbracket e_x \rrbracket^{\text{init}}_{\gamma},$$

$$\left\llbracket \begin{array}{l} e \texttt{ where rec init } x = c \\ \qquad\qquad \texttt{and } x = e_x \\ \qquad\qquad \texttt{and } y = e_y \end{array} \right\rrbracket^{\text{step}}_{\gamma} (p_x, (s, s_x, s_y)) = \begin{array}{l} \textit{let } s'_x, v_x \\ \textit{let } s'_y, v_y \\ \textit{let } s', v = \\ (v_x, (s', s' \end{array}$$



Reactive Probabilistic Programming

Guillaume Baudart
MIT-IBM Watson AI Lab, IBM Research
USA

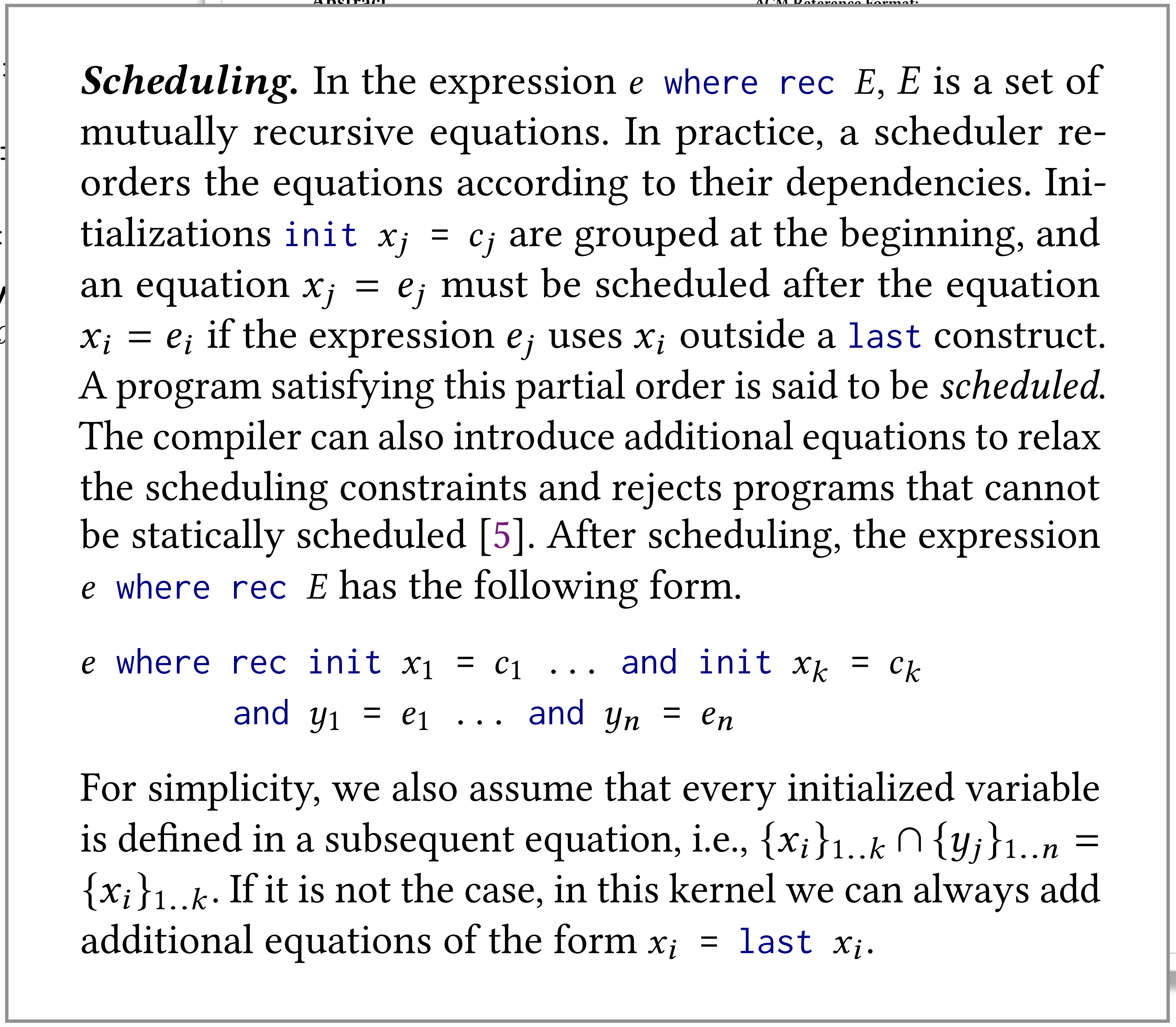
Louis Mandel
MIT-IBM Watson AI Lab, IBM Research
USA

Eric Atkinson
MIT
USA

Benjamin Sherman
MIT
USA

Marc Pouzet
École Normale Supérieure, PSL Research University
France

Michael Carbin
MIT
USA



Scheduling. In the expression e `where rec` E, E is a set of mutually recursive equations. In practice, a scheduler reorders the equations according to their dependencies. Initializations `init` x_j = c_j are grouped at the beginning, and an equation $x_j = e_j$ must be scheduled after the equation $x_i = e_i$ if the expression e_j uses x_i outside a `last` construct. A program satisfying this partial order is said to be *scheduled*. The compiler can also introduce additional equations to relax the scheduling constraints and rejects programs that cannot be statically scheduled [5]. After scheduling, the expression e `where rec` E has the following form.

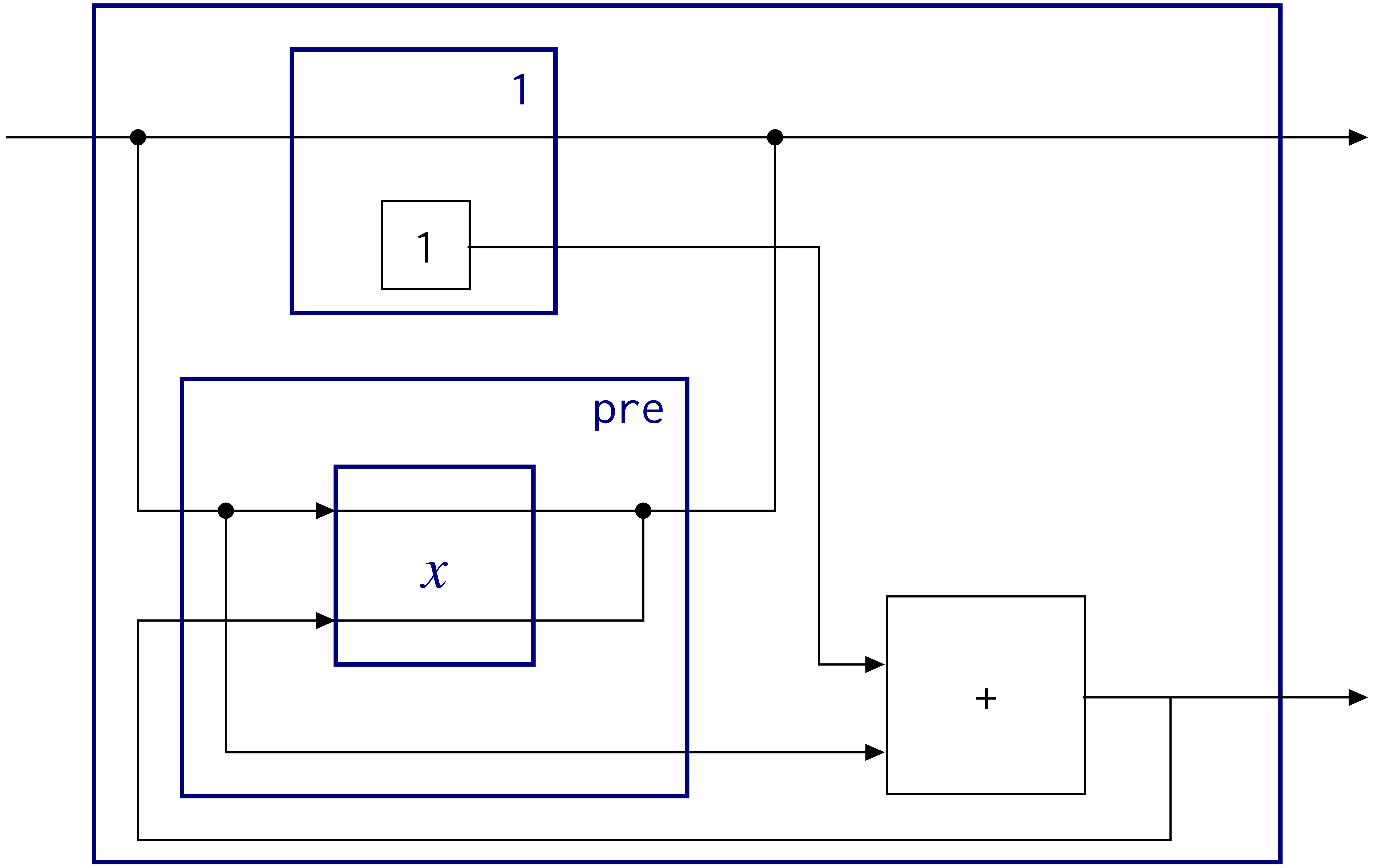
```
e where rec init x₁ = c₁ ... and init xₖ = cₖ
        and y₁ = e₁ ... and yₙ = eₙ
```

For simplicity, we also assume that every initialized variable is defined in a subsequent equation, i.e., $\{x_i\}_{1..k} \cap \{y_j\}_{1..n} = \{x_i\}_{1..k}$. If it is not the case, in this kernel we can always add additional equations of the form x_i = `last` x_i.

Example

`rec x = 1 + pre x`

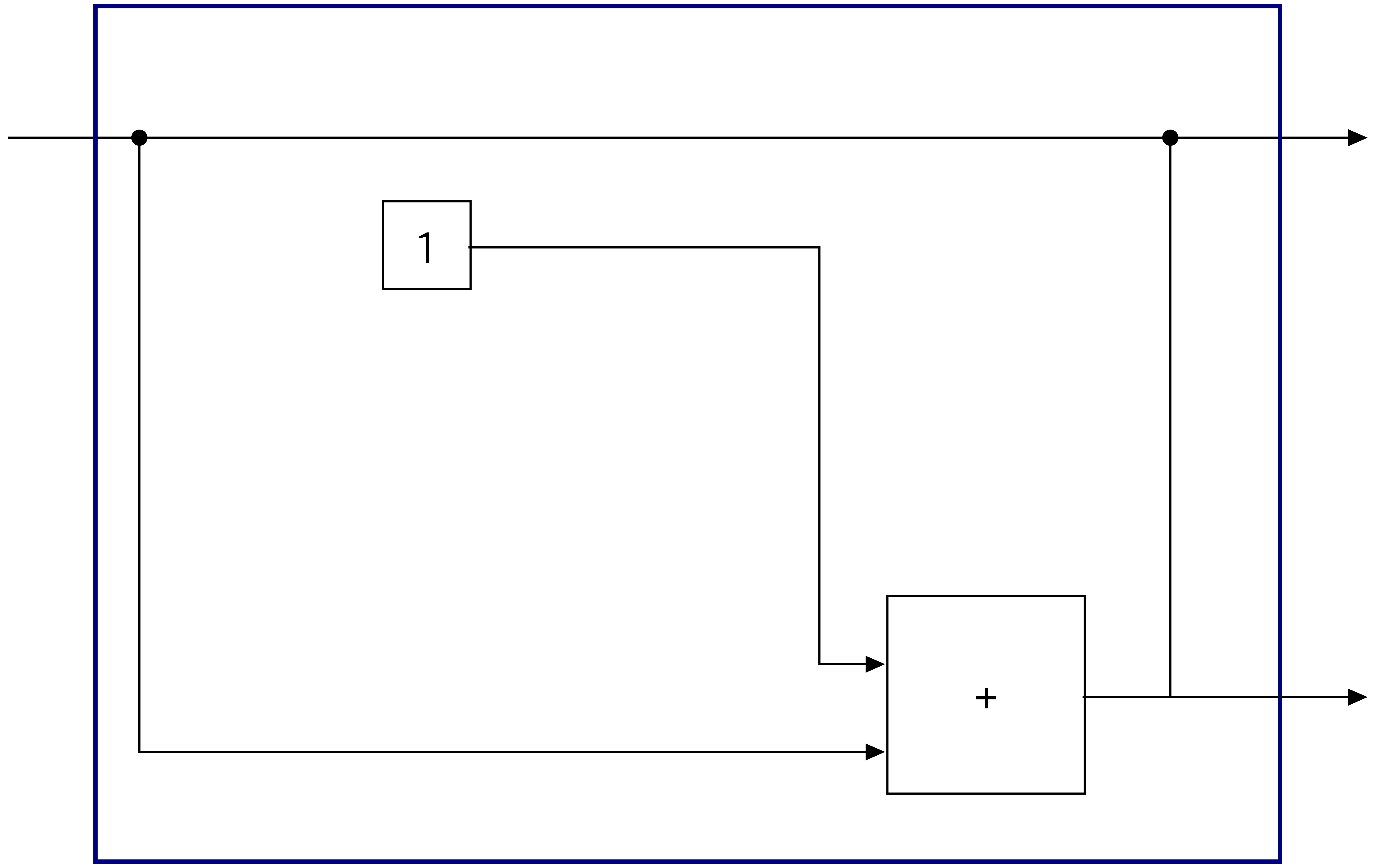
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

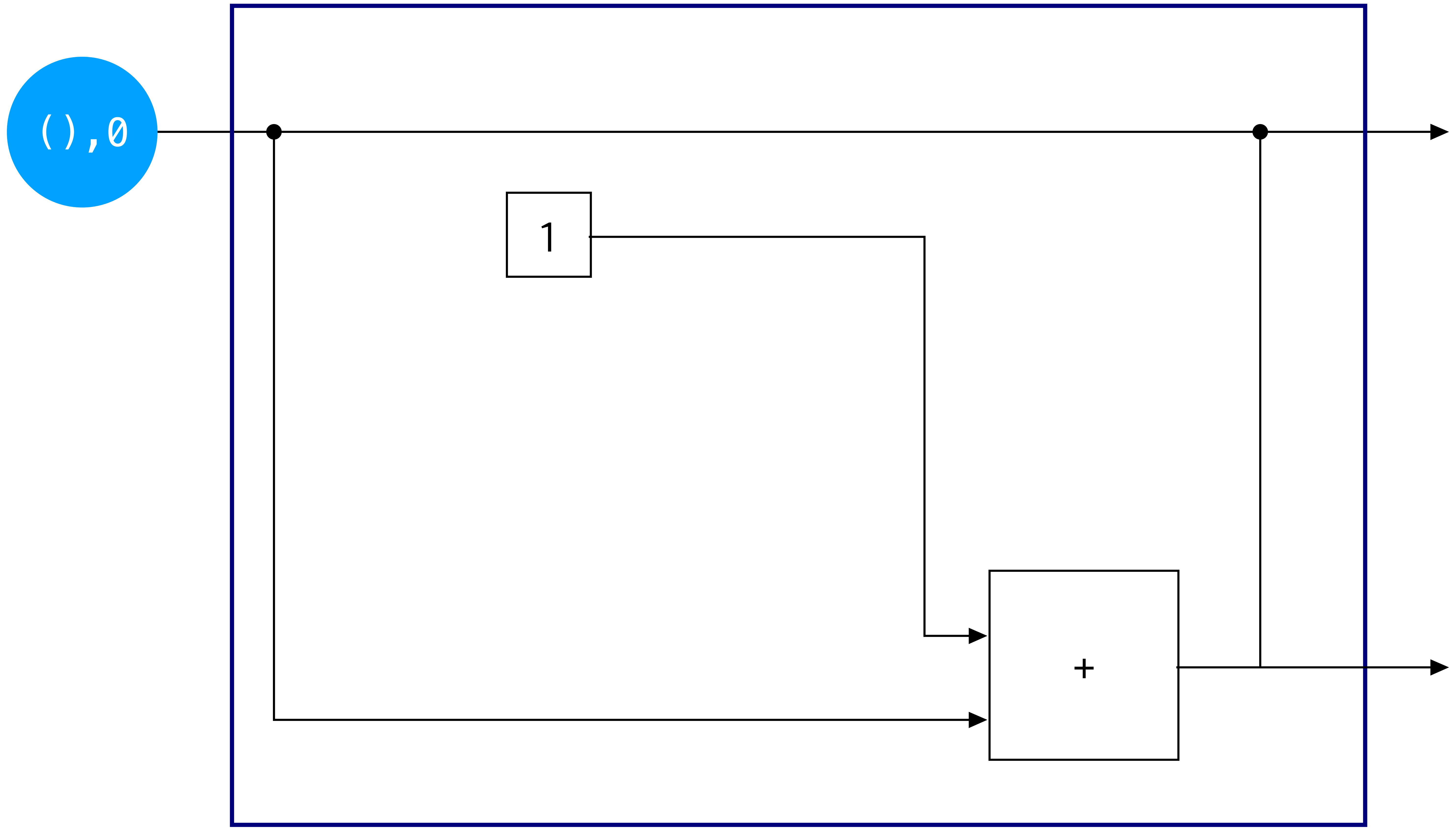
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

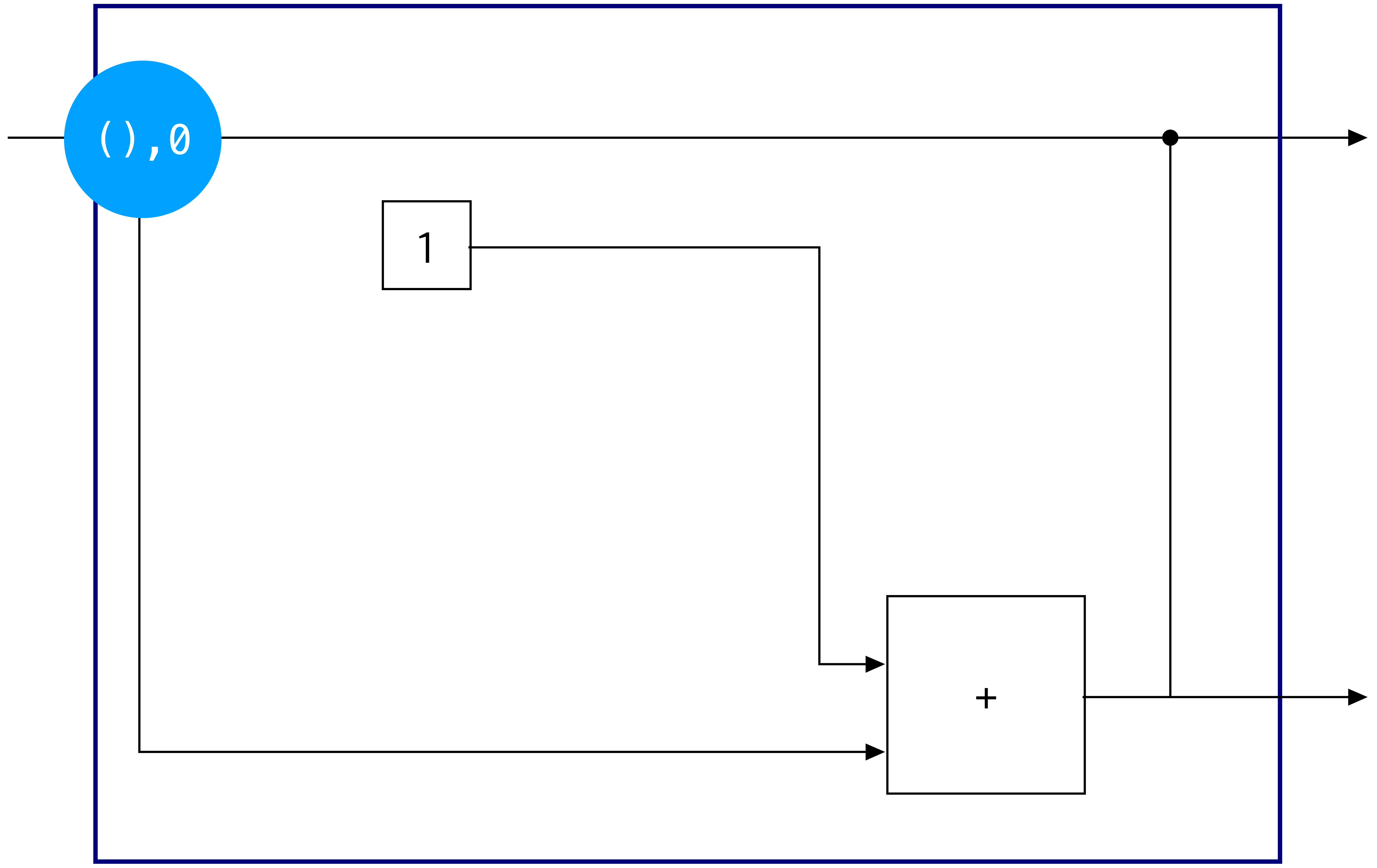
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

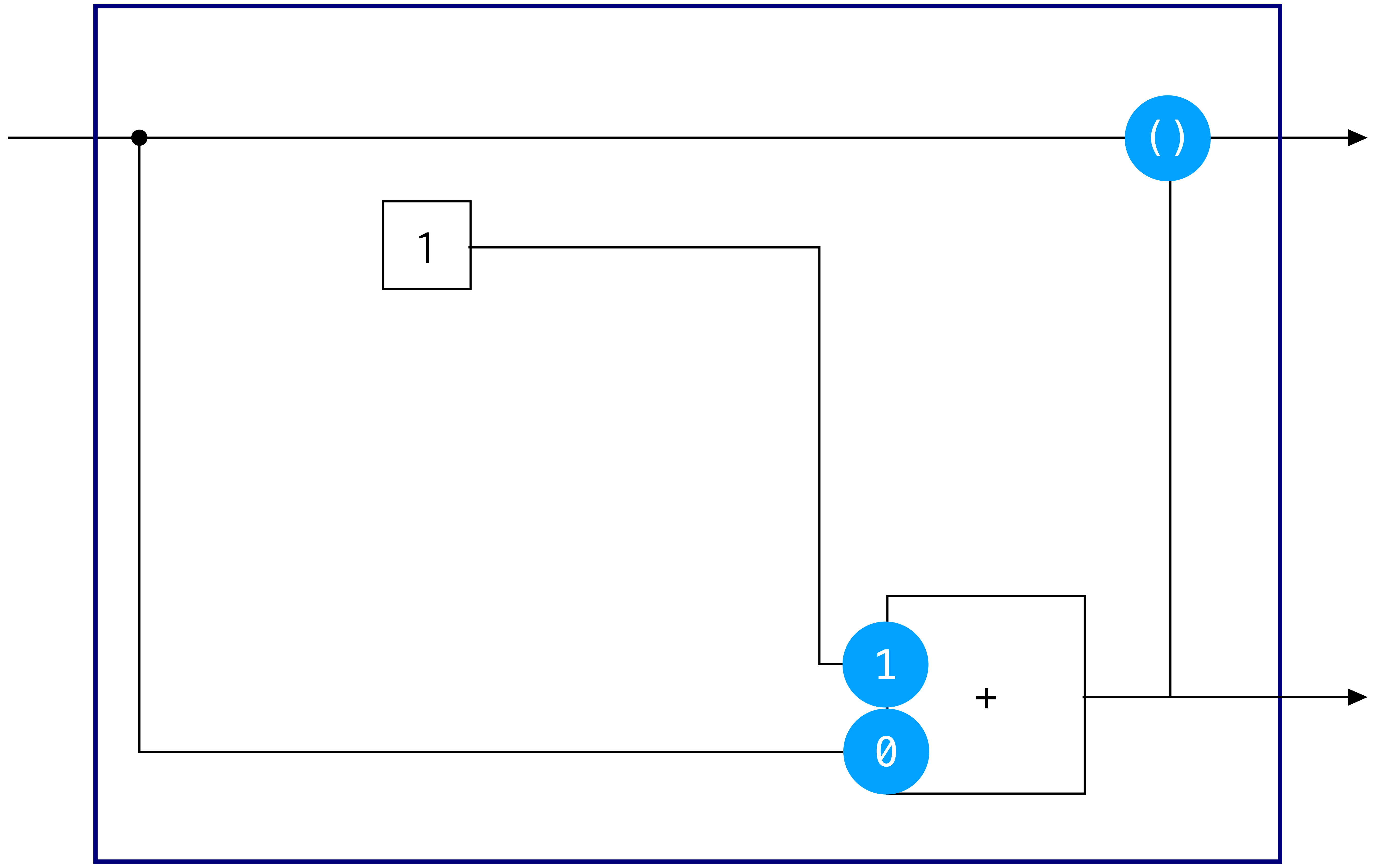
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

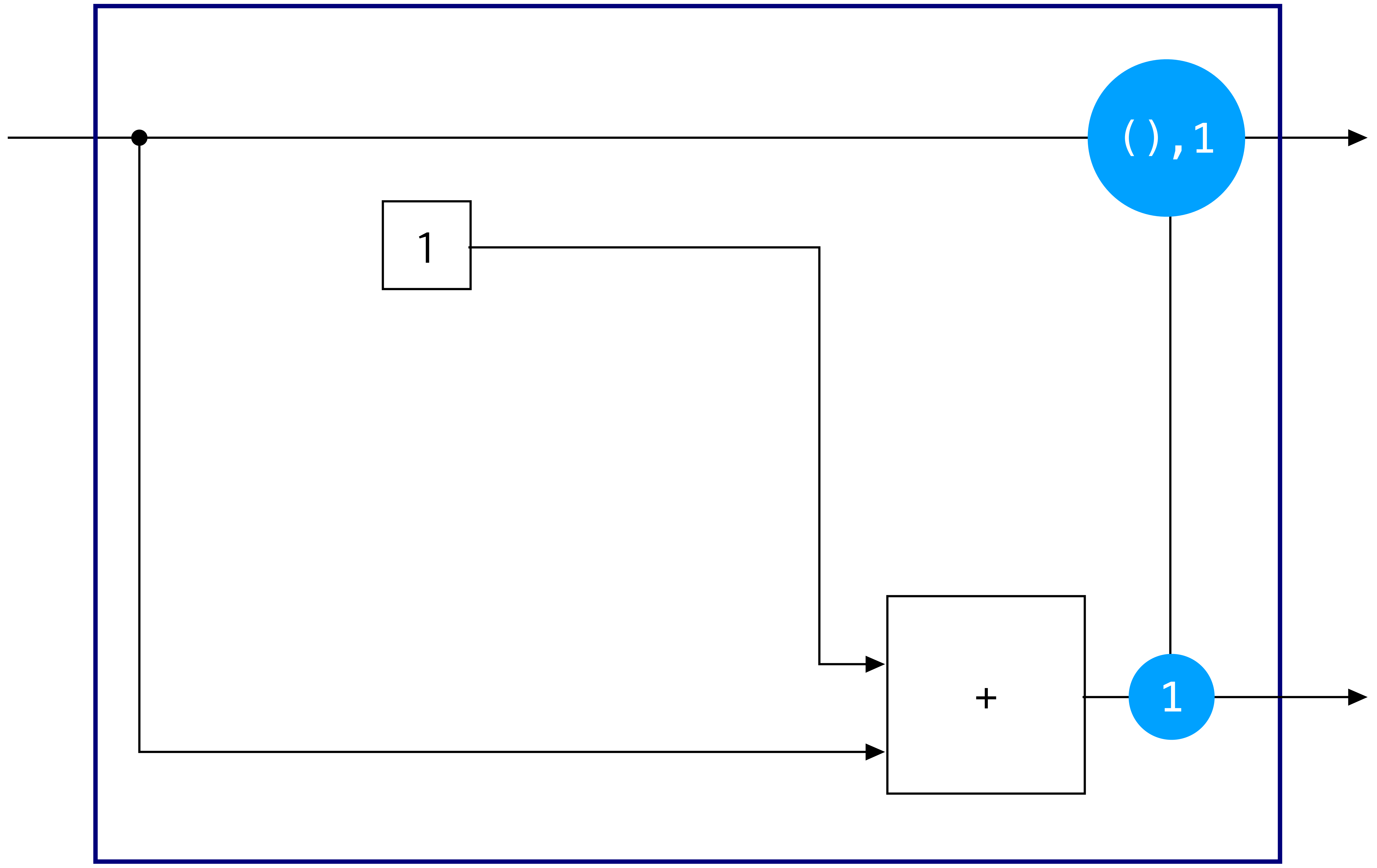
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

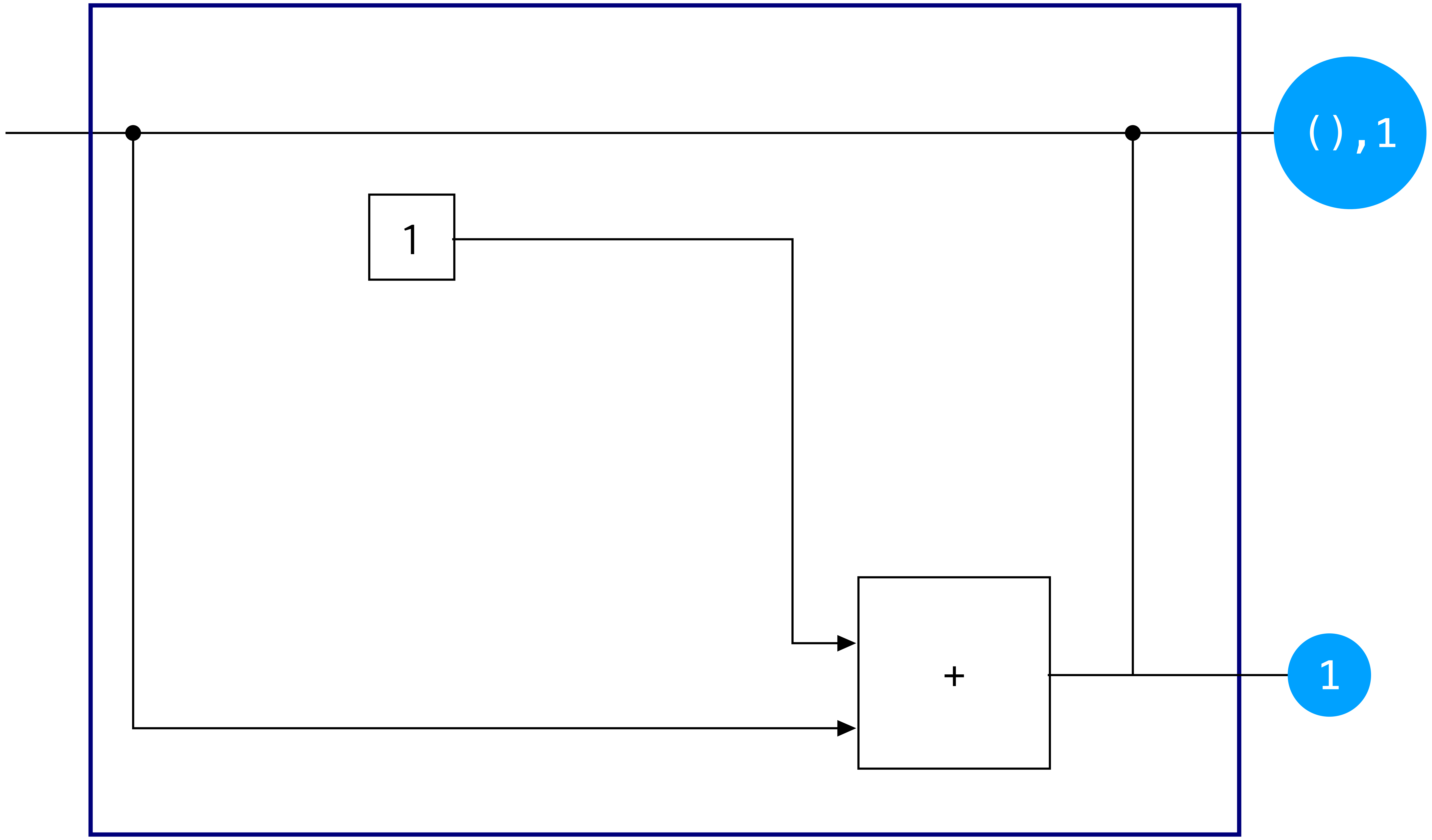
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

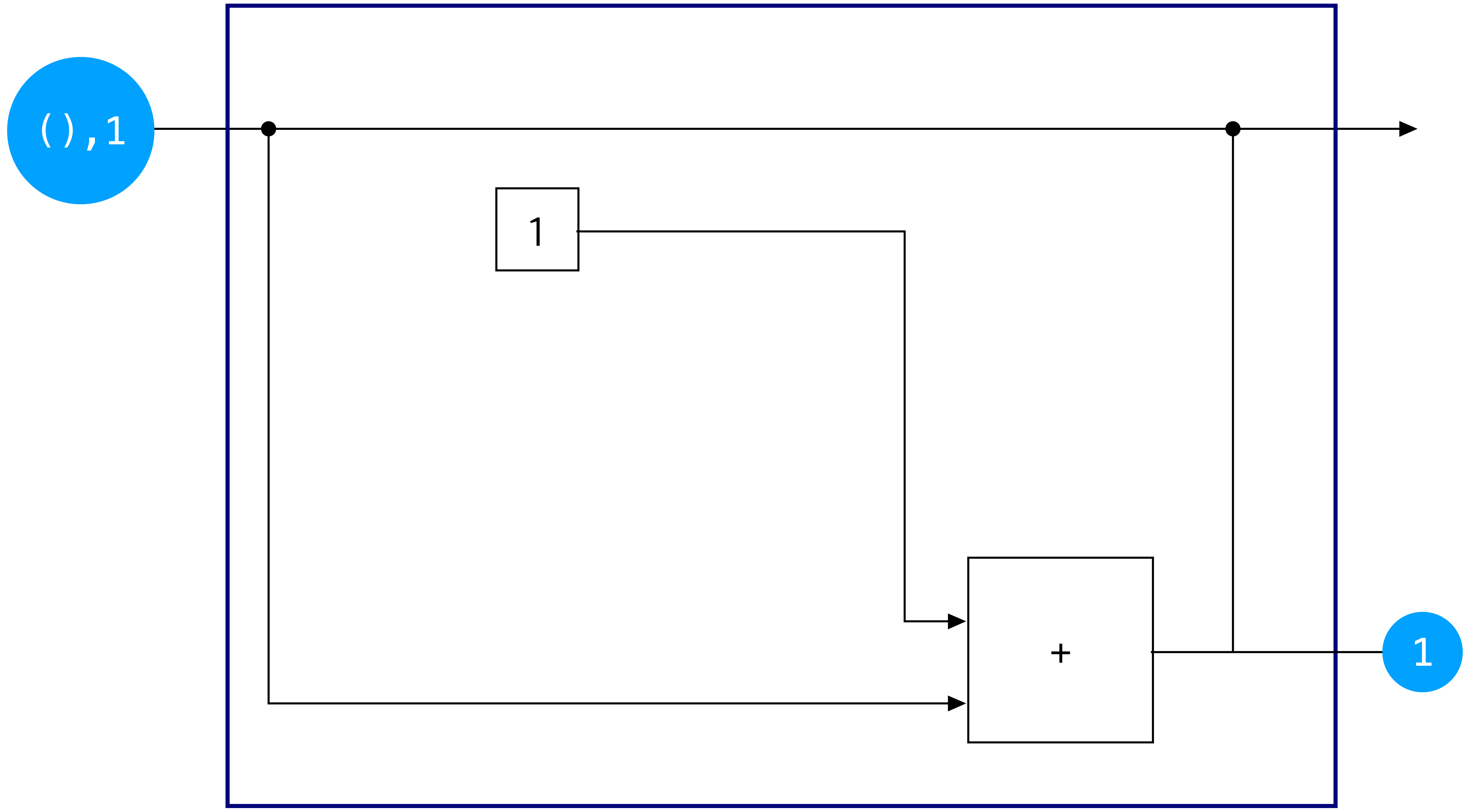
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

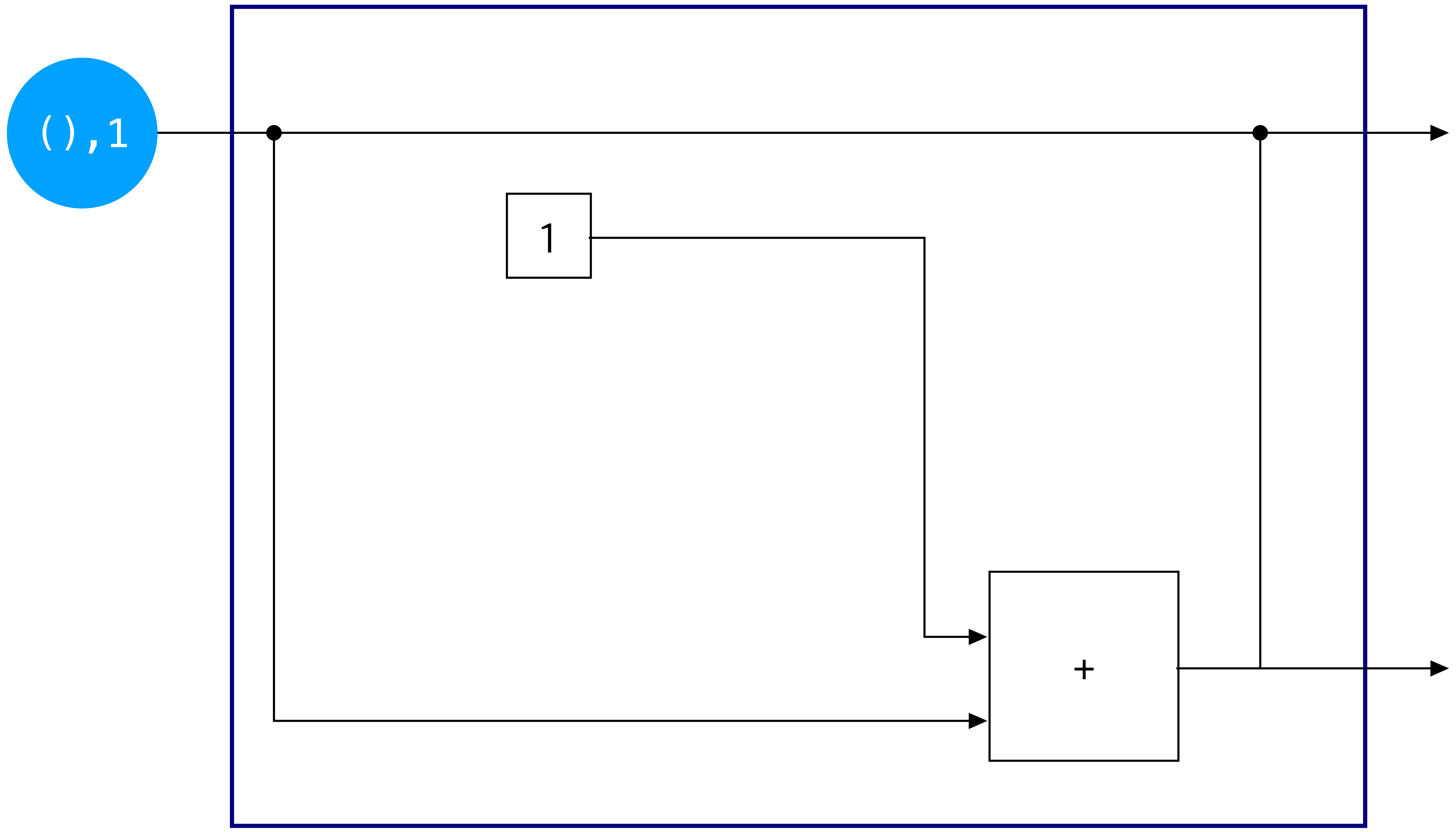
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

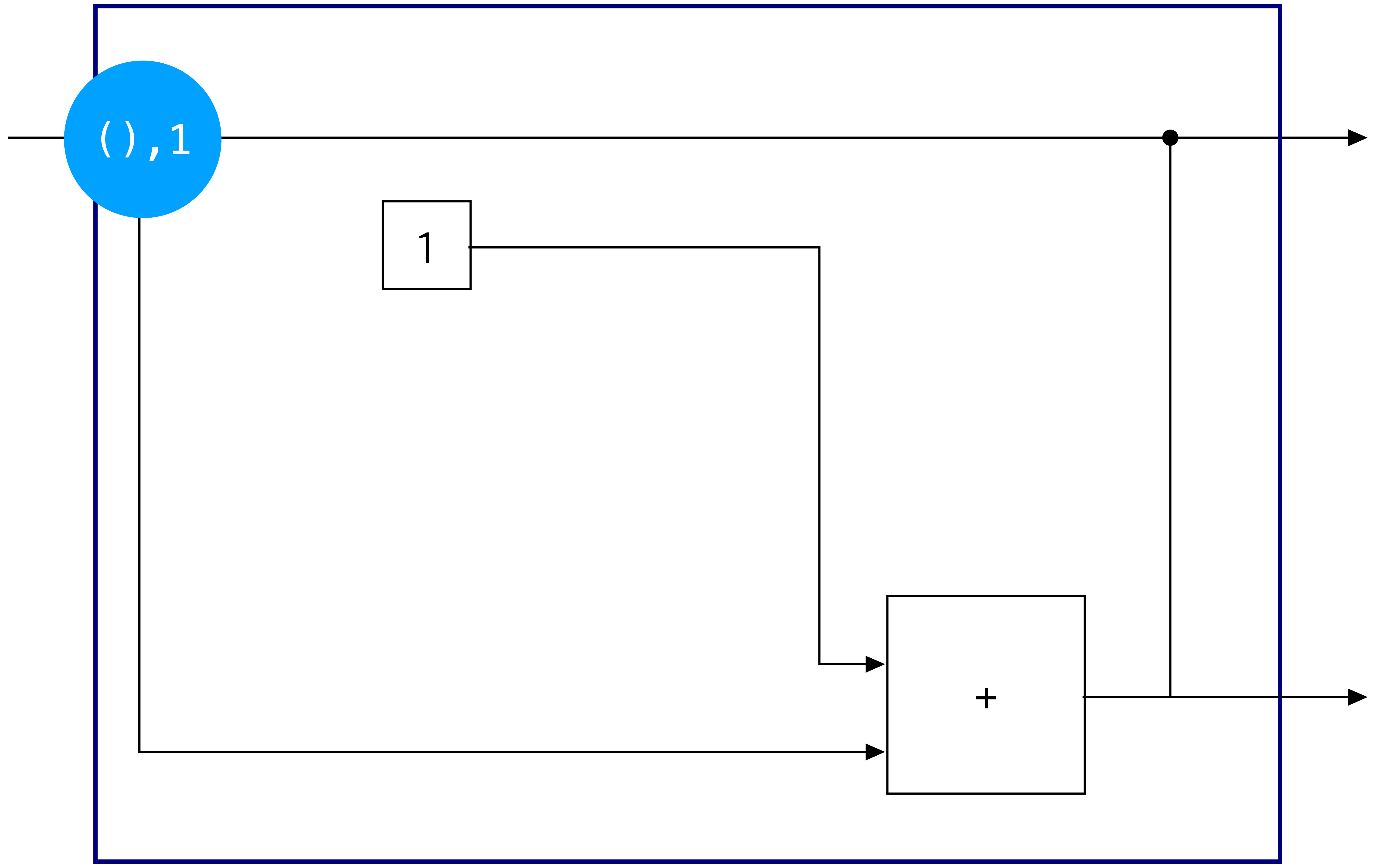
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

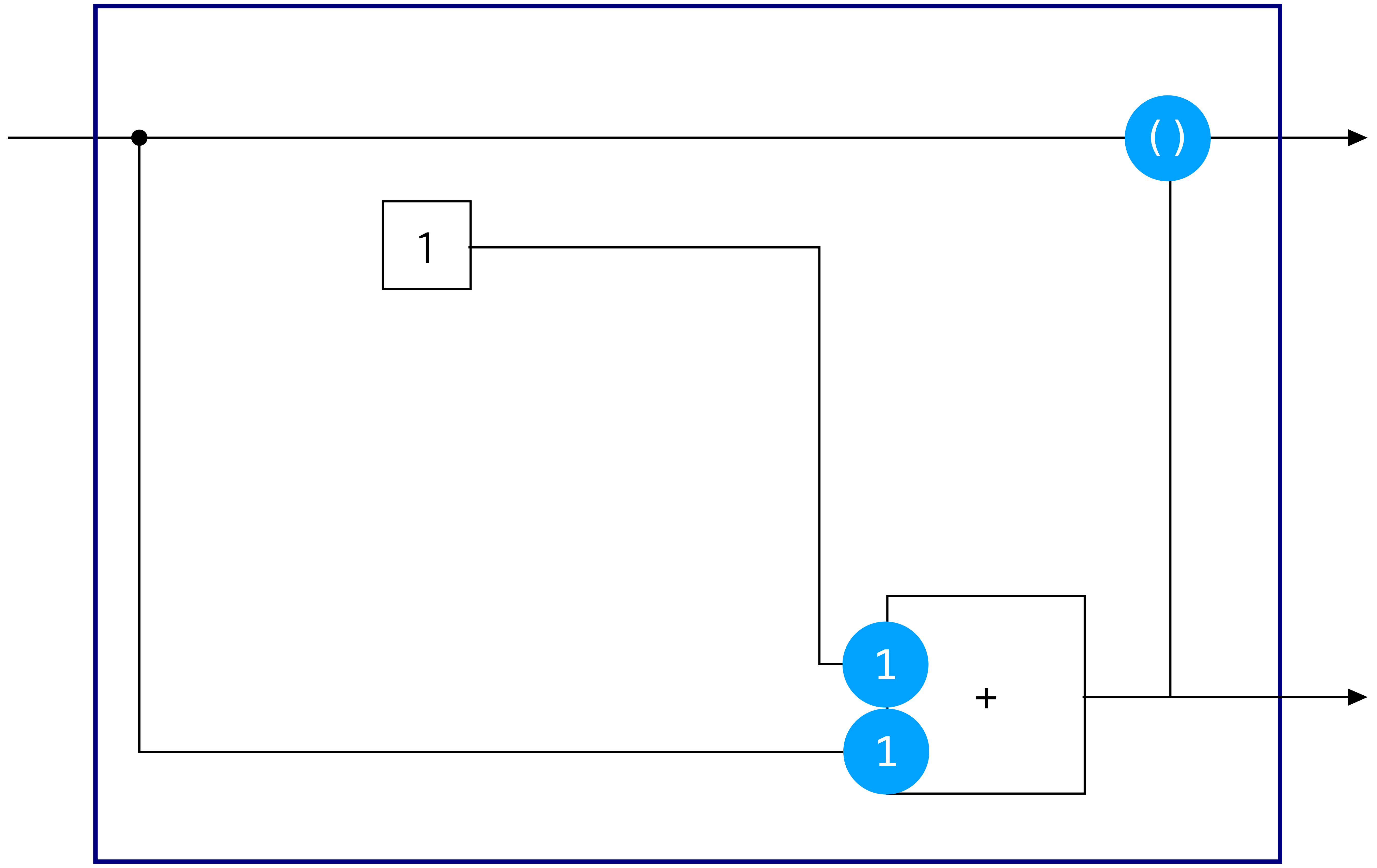
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

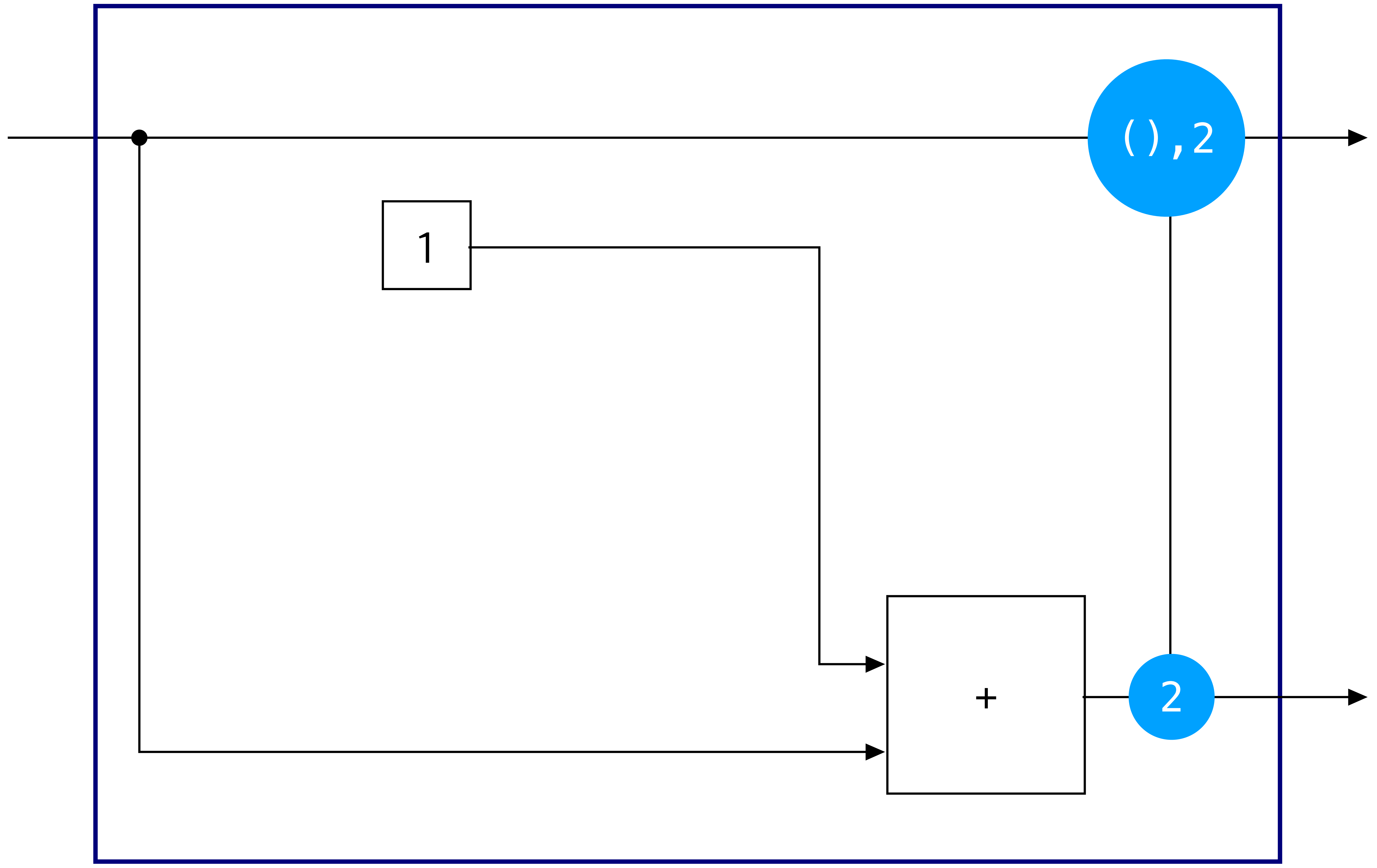
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

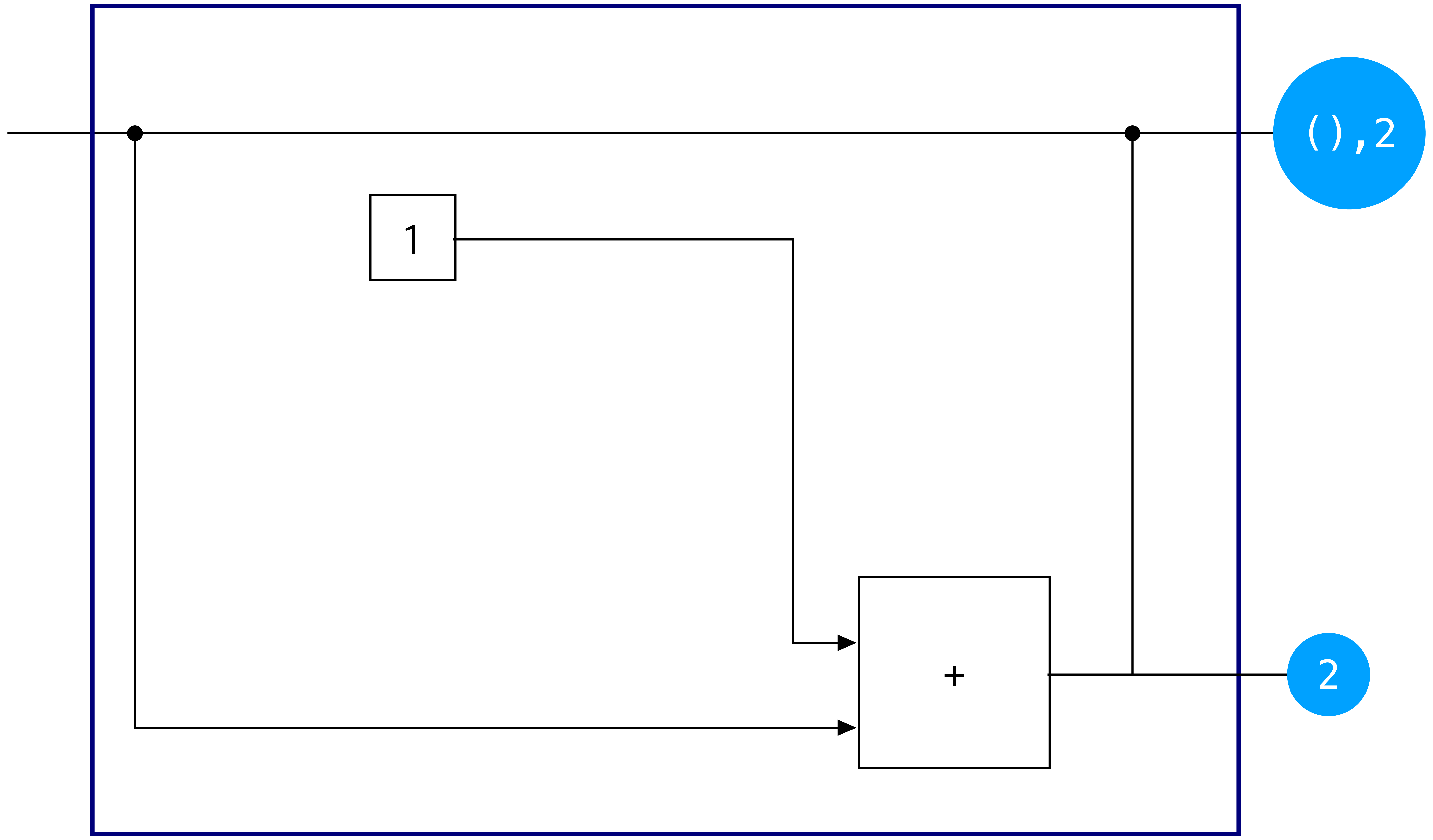
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

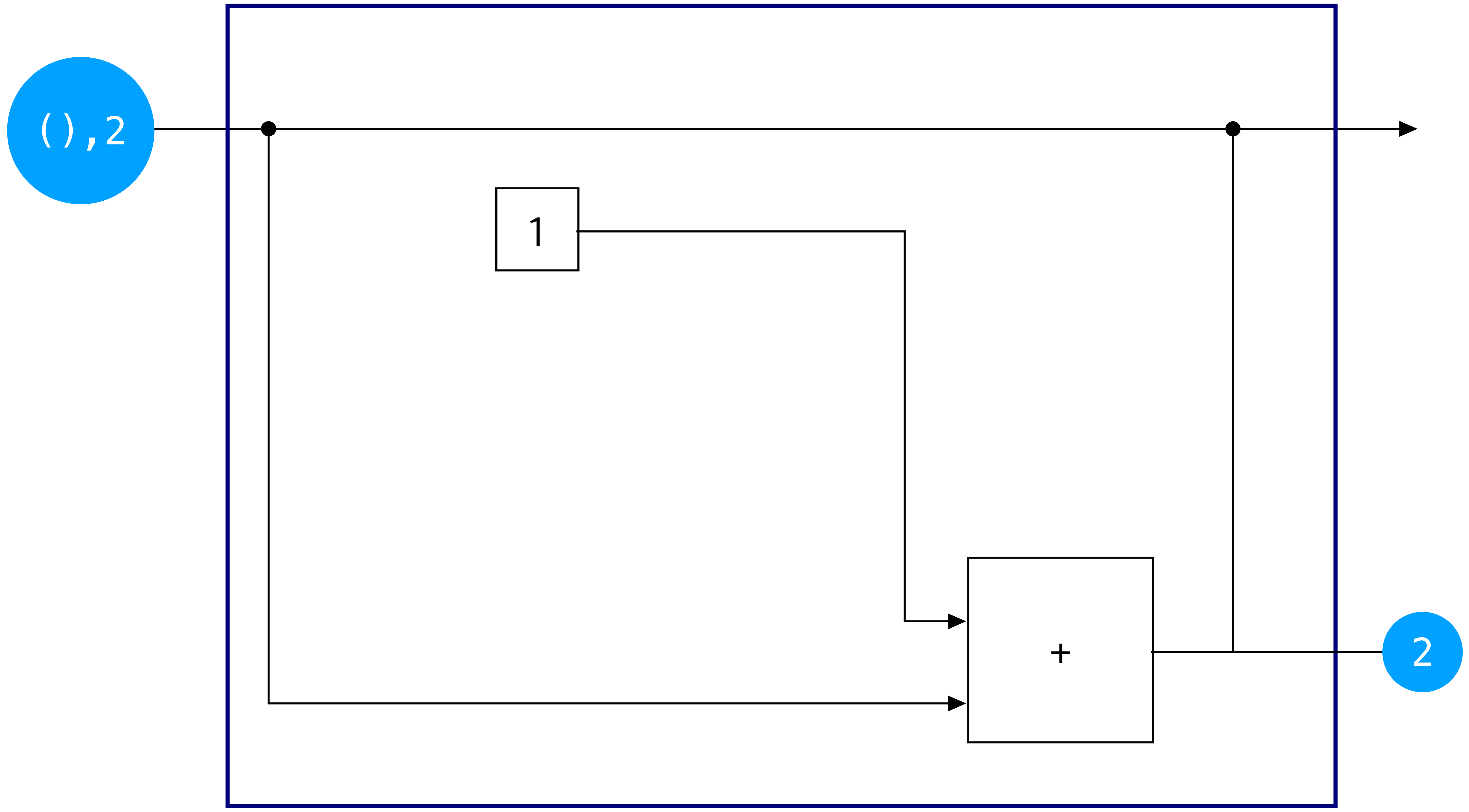
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Example

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

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

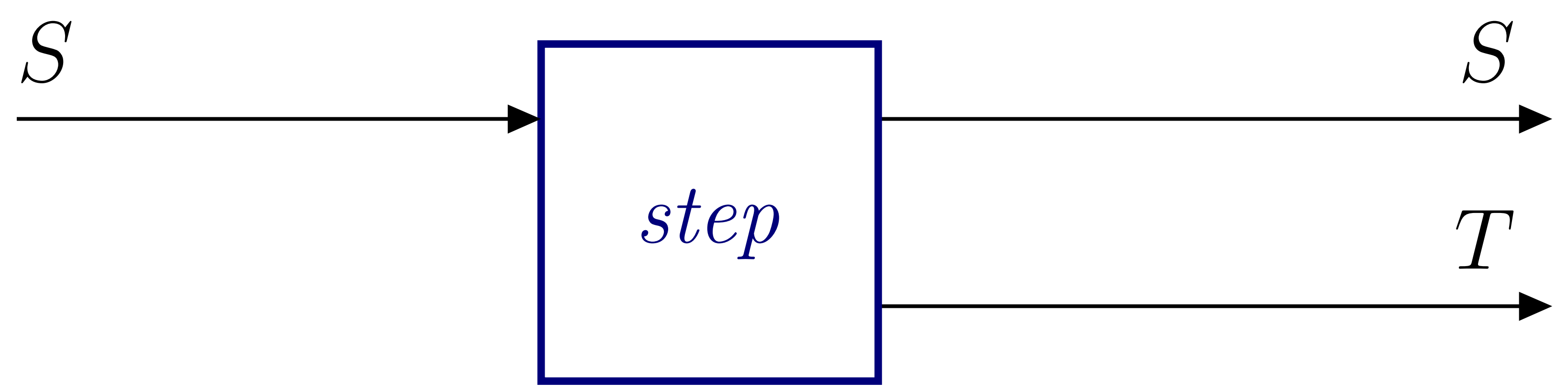


Deterministic vs. probabilistic

Deterministic streams

Transition function returns a pair of state and value

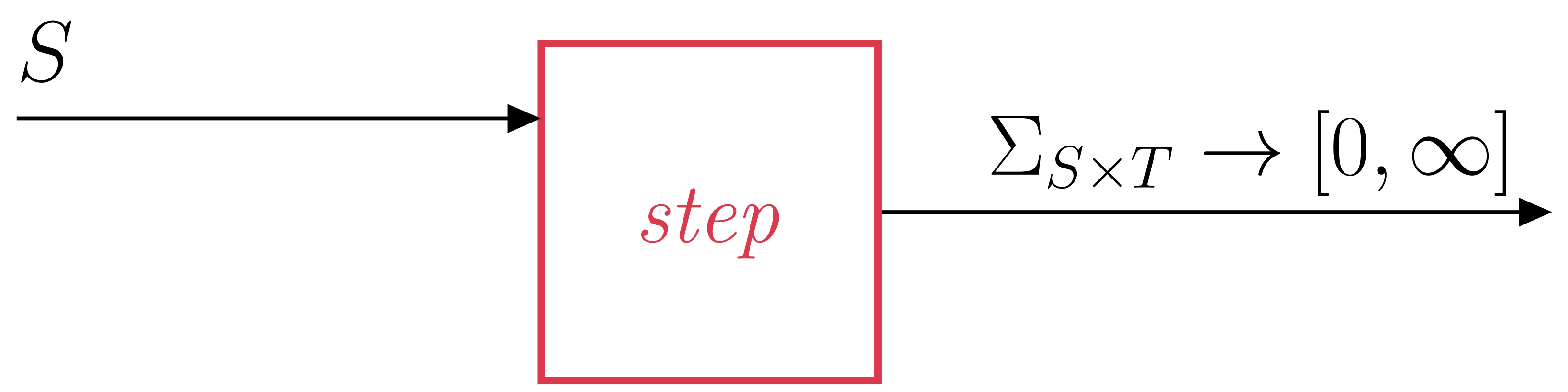
$$\textit{CoStream}(T, S) = S \times (S \to S \times T)$$



Probabilistic streams

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

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



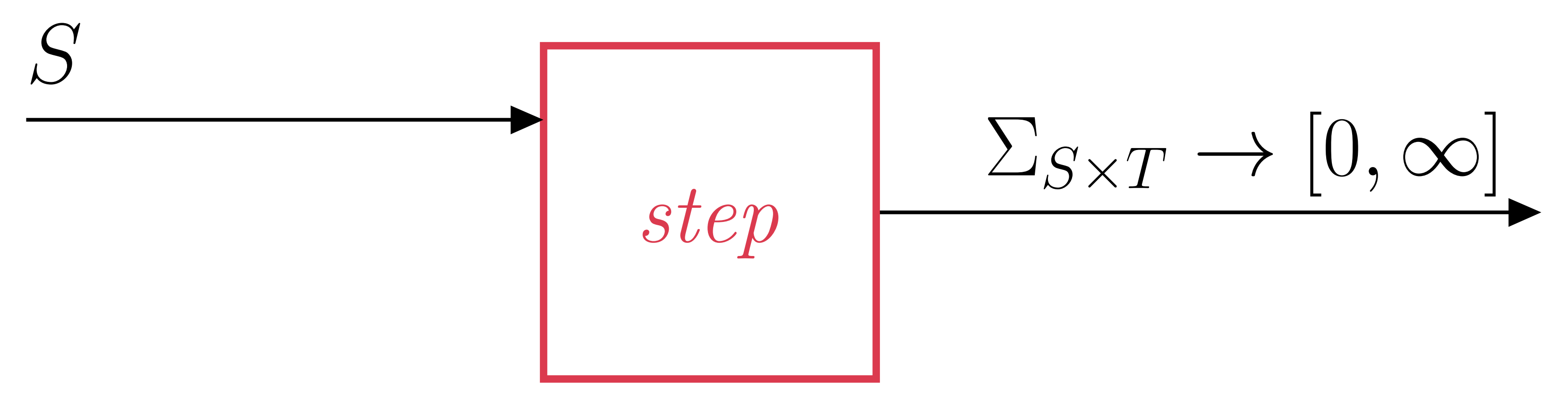
Probabilistic semantics

allocation: S

transition: $S \to \Sigma_{S \times T} \to [0, \infty]$

$$\{\!\![e]\!\!\} = \{\!\![e]\!\!\}^{\text{init}}_{\gamma}\,,\ \{\!\![e]\!\!\}^{\text{step}}_{\gamma}$$

environment

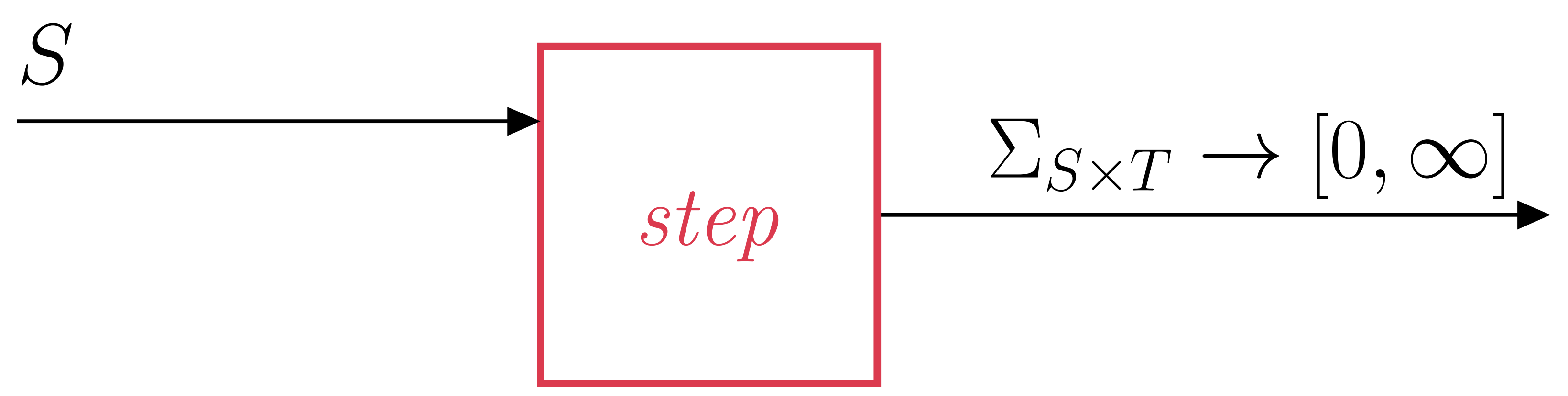


Probabilistic semantics

allocation: S transition: $S \to \Sigma_{S\times T} \to [0,\infty]$

$$\{\!\![e]\!\!\} = \{\!\![e]\!\!\}_{\gamma}^{\text{init}} , \{\!\![e]\!\!\}_{\gamma}^{\text{step}}$$

environment



if e is deterministic

$$
\begin{aligned}
\{\!\![e]\!\!\}_{\gamma}^{\text{init}} &= [\![e]\!]_{\gamma}^{\text{init}}\\
\{\!\![e]\!\!\}_{\gamma}^{\text{step}}(s) &= \textit{let } s', v = [\![e]\!]_{\gamma}^{\text{step}}(s) \textit{ in } \delta_{s',v}\\[1em]
\{\!\![\texttt{sample}(e)]\!\!\}_{\gamma}^{\text{init}} &= [\![e]\!]_{\gamma}^{\text{init}}\\
\{\!\![\texttt{sample}(e)]\!\!\}_{\gamma}^{\text{step}}(s) &= \textit{let } s', \mu = [\![e]\!]_{\gamma}^{\text{step}}(s) \textit{ in } \int \mu(dv)\, \delta_{s',v}\\[1em]
\{\!\![\texttt{observe}(e_1, e_2)]\!\!\}_{\gamma}^{\text{init}} &= [\![e_1]\!]_{\gamma}^{\text{init}}, [\![e_2]\!]_{\gamma}^{\text{init}}\\
\{\!\![\texttt{observe}(e_1, e_2)]\!\!\}_{\gamma}^{\text{step}}(s_1, s_2) &= \textit{let } s_1', \mu = [\![e_1]\!]_{\gamma}^{\text{step}}(s_1) \textit{ in}\\
&\quad\ \textit{let } s_2', v = [\![e_2]\!]_{\gamma}^{\text{step}}(s_2) \textit{ in}\\
&\quad\ \mu_{\text{pdf}}(v) * \delta_{(s_1', s_2'),()}
\end{aligned}
$$

Probabilistic equations

$$\left\{\!\!\left[\begin{array}{l} e\ \texttt{where rec init}\ x = c \\ \qquad\qquad \texttt{and}\ x = e_x \\ \qquad\qquad \texttt{and}\ y = e_y \end{array}\right]\!\!\right\}^{\mathsf{init}}_{\gamma} \quad = \quad c, \left(\{\![e]\!\}^{\mathsf{init}}_{\gamma}, \{\![e_x]\!\}^{\mathsf{init}}_{\gamma}, \{\![e_y]\!\}^{\mathsf{init}}_{\gamma}\right)$$

$$\left\{\!\!\left[\begin{array}{l} e\ \texttt{where rec init}\ x = c \\ \qquad\qquad \texttt{and}\ x = e_x \\ \qquad\qquad \texttt{and}\ y = e_y \end{array}\right]\!\!\right\}^{\mathsf{step}}_{\gamma} (p_x, (m, m_x, m_y)) \quad = \quad \int \{\![e_x]\!\}^{\mathsf{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x]} (m_x)(dm'_x, dv_x) \int \{\![e_y]\!\}^{\mathsf{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x, x\leftarrow v_x]} (m_y)(dm'_y, dv_y) \int \{\![e]\!\}^{\mathsf{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x, x\leftarrow v_x, y\leftarrow v_y]} (m)(dm', d_v)\ \delta_{(v_x, (m', m'_x, m'_y)), v}$$

Probabilistic equations

$$\left\{\!\!\left[\begin{array}{l} e\ \texttt{where rec init}\ x = c \\ \qquad\quad \texttt{and}\ x = e_x \\ \qquad\quad \texttt{and}\ y = e_y \end{array}\right]\!\!\right\}^{\text{init}}_{\gamma} = c, \left(\{\![e]\!\}^{\text{init}}_{\gamma}, \{\![e_x]\!\}^{\text{init}}_{\gamma}, \{\![e_y]\!\}^{\text{init}}_{\gamma}\right)$$

$$\left\{\!\!\left[\begin{array}{l} e\ \texttt{where rec init}\ x = c \\ \qquad\quad \texttt{and}\ x = e_x \\ \qquad\quad \texttt{and}\ y = e_y \end{array}\right]\!\!\right\}^{\text{step}}_{\gamma} (p_x, (m, m_x, m_y)) = \int \{\![e_x]\!\}^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x]} (m_x)(dm'_x, dv_x)$$

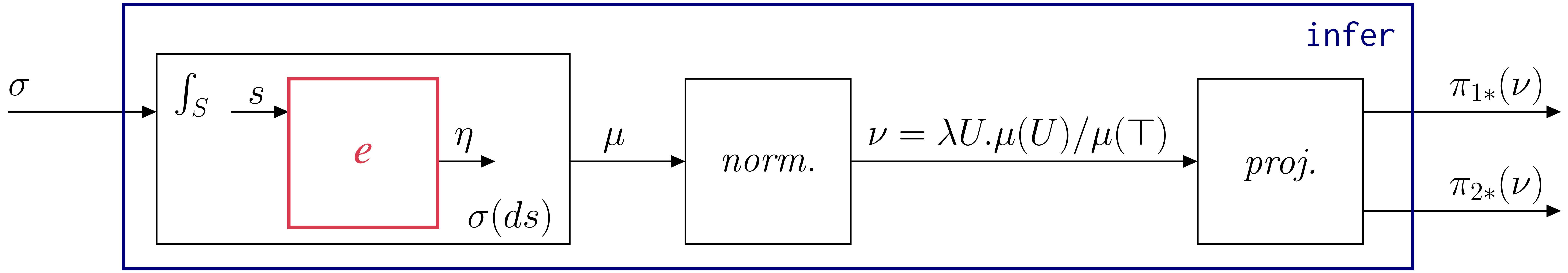
$$\int \{\![e_y]\!\}^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x, x\leftarrow v_x]} (m_y)(dm'_y, dv_y)$$

$$\int \{\![e]\!\}^{\text{step}}_{\gamma+[x.\texttt{last}\leftarrow p_x, x\leftarrow v_x, y\leftarrow v_y]} (m)(dm', d_v)$$

$$\delta_{(v_x, (m', m'_x, m'_y)), v}$$

Nested integrals require a fixed schedule

Semantics of `infer`



$$\begin{aligned}
[\![\texttt{infer}(e)]\!]^{\text{init}}_{\gamma} &= \delta_{[\![e]\!]^{\text{init}}_{\gamma}} \\
[\![\texttt{infer}(e)]\!]^{\text{step}}_{\gamma}(\sigma) &= \textit{let } \nu = \int \sigma(dm)\, \{\![e]\!\}^{\text{step}}_{\gamma}(m) \textit{ in let } \overline{\nu} = \nu/\nu(\top) \textit{ in} \\
&\quad (\pi_{1*}(\overline{\nu}), \pi_{2*}(\overline{\nu}))
\end{aligned}$$

Compilation

Reactive Probabilistic Programming

Target

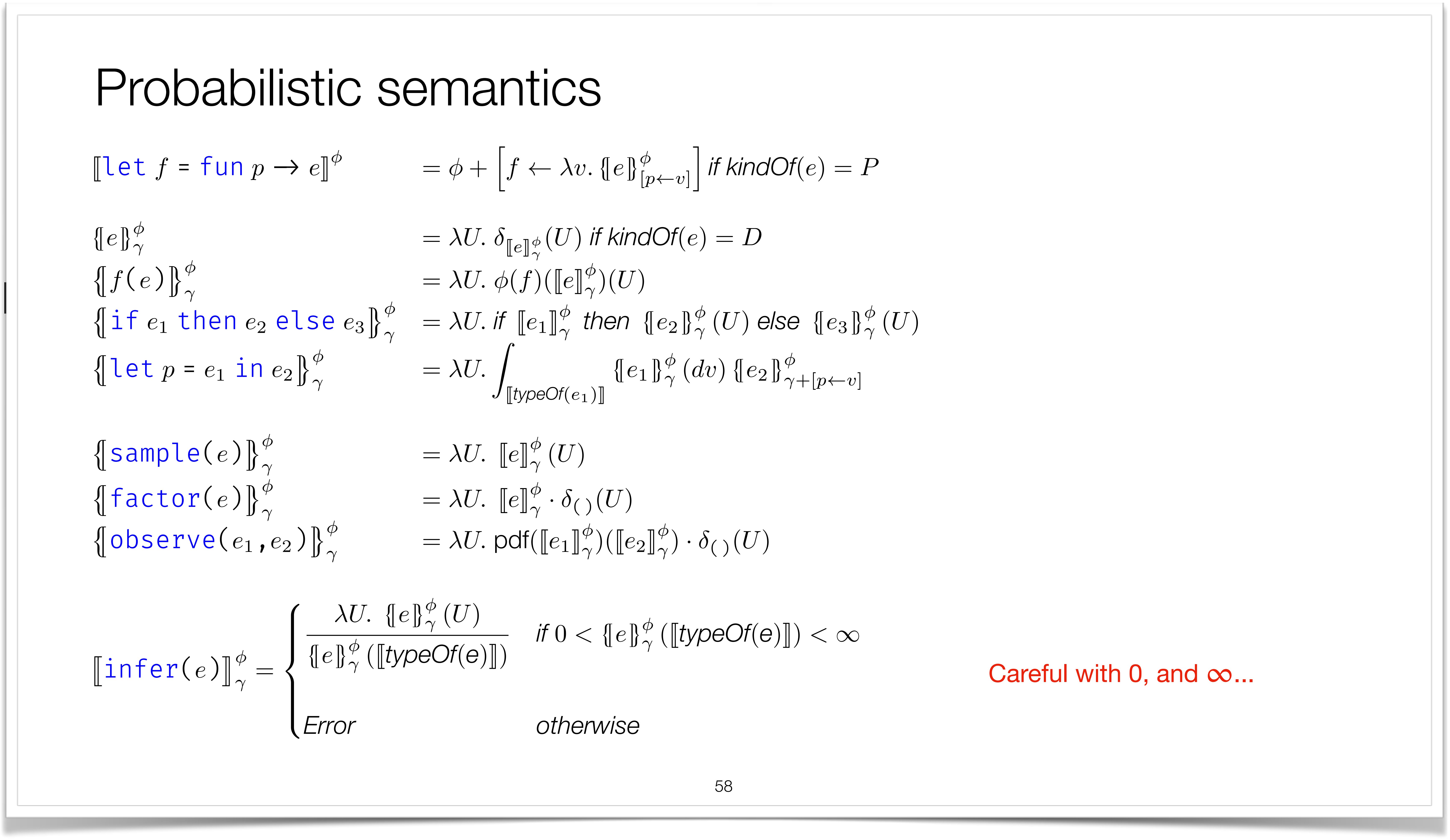
Simplified syntax

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

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

Target

Simplified syntax

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

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



Probabilistic semantics

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

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

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

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

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

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

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

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

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

Careful with 0, and ∞...

58

Compilation = Allocation + Transition

Synchronous languages

- Static analyses (typing, causality, initialization)
- Normalization, scheduling (`rec x = y + 1 and y = 0 -> pre x`)
- Compilation

Memory can be statically allocated

$$
\begin{tikzcd}
 & \text{transition: } S \to S \times T \\
e \mapsto \mathcal{A}(e), \mathcal{C}(e) \arrow[ur] \arrow[d] & \\
\text{allocation: } S &
\end{tikzcd}
$$

Allocation

$$\begin{aligned}
\mathcal{A}(x) &= (\,) \\
\mathcal{A}((e_1, e_2)) &= (\mathcal{A}(e_1), \mathcal{A}(e_2)) \\
\mathcal{A}(\texttt{present}\ e \rightarrow e_1\ \texttt{else}\ e_2) &= (\mathcal{A}(e), \mathcal{A}(e_1), \mathcal{A}(e_2)) \\
\mathcal{A}(\mathit{op}(e)) &= \mathcal{A}(e) \\
\mathcal{A}(f(e)) &= (f\texttt{\_init}, \mathcal{A}(e)) \\
\mathcal{A}(\texttt{sample}(e)) &= \mathcal{A}(e) \\
\mathcal{A}(\texttt{factor}(e)) &= \mathcal{A}(e) \\
\mathcal{A}(\texttt{observe}(e_1, e_2)) &= (\mathcal{A}(e_1), \mathcal{A}(e_2)) \\
\mathcal{A}(\texttt{infer}(e)) &= \mathcal{A}(e)
\end{aligned}$$

Allocation

$$
\begin{array}{lcl}
\mathcal{A}(x) & = & (\,) \\
\mathcal{A}((e_1, e_2)) & = & (\mathcal{A}(e_1), \mathcal{A}(e_2)) \\
\mathcal{A}(\texttt{present}\ e \rightarrow e_1\ \texttt{else}\ e_2) & = & (\mathcal{A}(e), \mathcal{A}(e_1), \mathcal{A}(e_2)) \\
\mathcal{A}(op(e)) & = & \mathcal{A}(e) \\
\mathcal{A}(f(e)) & = & (f\texttt{\_init}, \mathcal{A}(e)) \\
\mathcal{A}(\texttt{sample}(e)) & = & \mathcal{A}(e) \\
\mathcal{A}(\texttt{factor}(e)) & = & \mathcal{A}(e) \\
\mathcal{A}(\texttt{observe}(e_1, e_2)) & = & (\mathcal{A}(e_1), \mathcal{A}(e_2)) \\
\mathcal{A}(\texttt{infer}(e)) & = & \mathcal{A}(e)
\end{array}
$$

$$
\mathcal{A}\left(\begin{array}{l}
e\ \texttt{where} \\
\quad \texttt{rec init}\ x = c_x \\
\quad \texttt{and init}\ y = c_y \\
\quad \texttt{and}\ x = e_x \\
\quad \texttt{and}\ y = e_y
\end{array}\right)
=
\left(\begin{array}{c}
(c_x, c_y), \\
(\mathcal{A}(e_1), \mathcal{A}(e_2)), \\
\mathcal{A}(e)
\end{array}\right)
$$

Transition

```
C(c) = fun s -> (c, s)
C(x) = fun s -> (x, s)
C(last x) = fun s -> (x_last, s)

C((e1, e2)) = fun (s1,s2) ->
  let v1,s1' = C(e1)(s1) in
  let v2,s2' = C(e2)(s2) in
  ((v1,v2), (s1',s2'))

C(op(e)) = fun s ->
  let v,s' = C(e)(s) in
  (op(v), s')

C(f(e)) = fun (s1,s2) ->
  let v1,s1' = C(e)(s1) in
  let v2,s2' = f_step(s2,v) in
  (v2, (s1',s2'))

C(present e -> e1 else e2) =
fun (s,s1,s2) ->
  let v, s' = C(e)(s) in
  if v then let v1,s1' = C(e1)(s1) in
    (v1, (s',s1',s2))
  else let v2,s2' = C(e2)(s2) in
    (v2, (s',s1,s2'))
```

```
C(sample(e)) = fun s ->
  let mu,s' = C(e)(s) in
  let v = sample(mu) in (v, s')

C(observe(e1, e2)) = fun (s1,s2) ->
  let v1,s1' = C(e1)(s1) in
  let v2,s2' = C(e2)(s2) in
  let _ = observe(v1,v2) in
  ((), (s1',s2'))

C(factor(e)) = fun s ->
  let v,s' = C(e)(s) in
  let _ = factor(v) in ((), s')

C(infer(e)) = fun sigma ->
  let mu,sigma' = infer(C(e), sigma) in
  (mu, sigma')
```

Transition

```
C(c) = fun s -> (c, s)
C(x) = fun s -> (x, s)
C(last x) = fun s -> (x_last, s)

C((e1, e2)) = fun (s1,s2) ->
  let v1,s1' = C(e1)(s1) in
  let
  ((v

C(op(
  let
  (op

C(f(e
  let
  let
  (v2
```

```
C(present e -> e1 else e2) =
fun (s,s1,s2) ->
  let v, s' = C(e)(s) in
  if v then let v1,s1' = C(e1)(s1) in
    (v1, (s',s1',s2))
  else let v2,s2' = C(e2)(s2) in
    (v2, (s',s1,s2'))
```

```
C(present e -> e1 else e2) =
fun (s,s1,s2) ->
  let v, s' = C(e)(s) in
  if v then let v1,s1' = C(e1)(s1) in
    (v1, (s',s1',s2))
  else let v2,s2' = C(e2)(s2) in
    (v2, (s',s1,s2'))
```

```
C(sample(e)) = fun s ->
  let mu,s' = C(e)(s) in
  let v = sample(mu) in (v, s')

C(observe(e1, e2)) = fun (s1,s2) ->
  let v1,s1' = C(e1)(s1) in
  let v2,s2' = C(e2)(s2) in
  let _ = observe(v1,v2) in
  ((), (s1',s2'))

factor(e)) = fun s ->
  let v,s' = C(e)(s) in
  let _ = factor(v) in ((), s')

infer(e)) = fun sigma ->
  let mu,sigma' = infer(C(e), sigma) in
  (mu, sigma')
```

Transition

```
C(c) = fun s -> (c, s)
C(x) = fun s -> (x, s)
C(last x) = fun s -> (x_last, s)

C((e1, e2)) = fun (s1,s2) ->
  let v1,s1' = C(e1)(s1) in
  let
  ((v

C(op(
  let
  (op

C(f(e
  let
  let
  (v2
```

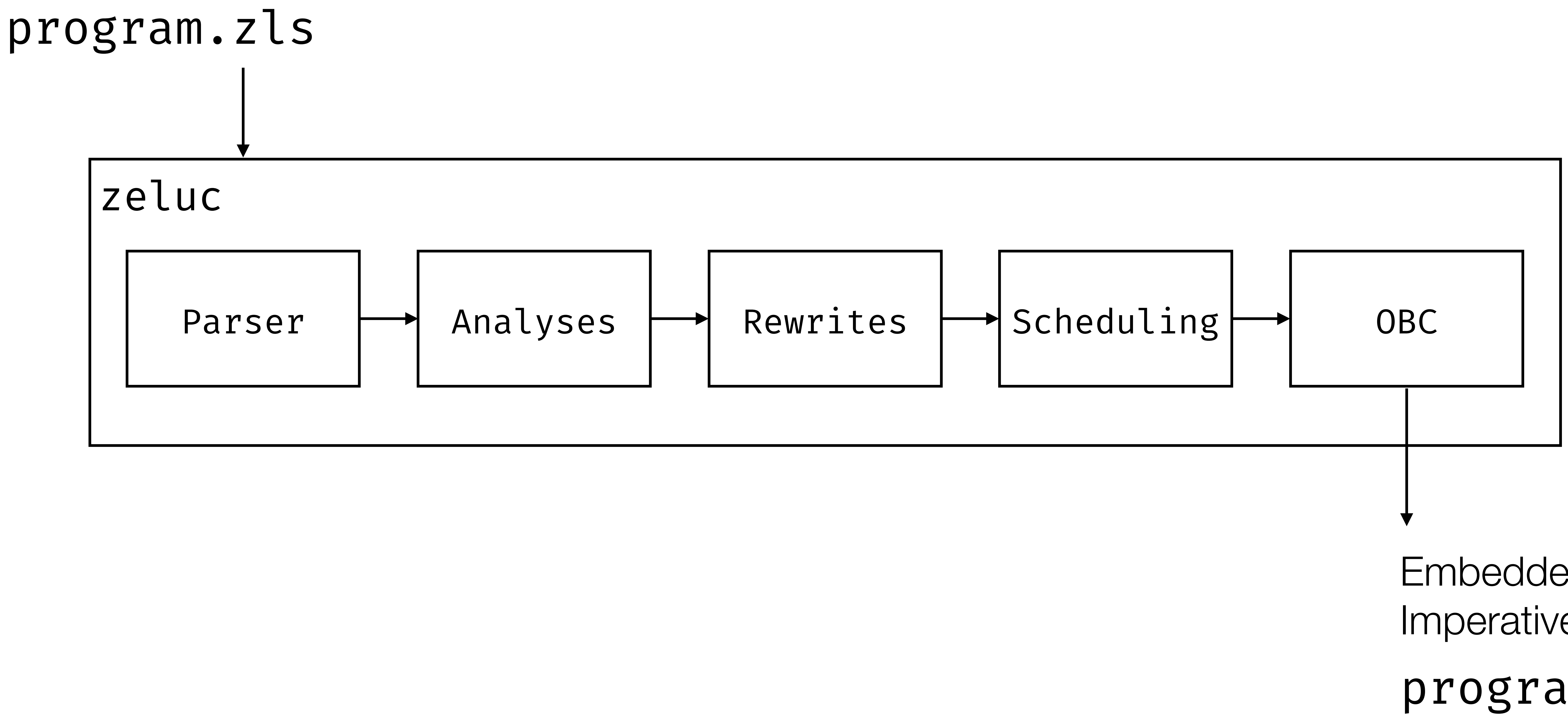
```
C(present e -> e1 else e2) =
fun (s,s1,s2) ->
  let v, s' = C(e)(s) in
  if v then let v1,s1' = C(e1)(s1) in
    (v1, (s',s1',s2))
  else let v2,s2' = C(e2)(s2) in
    (v2, (s',s1,s2'))
```

```
C(sample(e)
  let mu,s'
  let v = s

C(observe(e
  let v1,s1
  let v2,s2
  let _ = c
  ((), (s1'

factor(e)
  let v,s'
  let _ = f

infer(e))
  let mu,si
  (mu, sigm
```

```
C(sample(e)) = fun s ->
  let mu,s' = C(e)(s) in
  let v = sample(mu) in (v, s')

C(observe(e1, e2)) = fun (s1,s2) ->
  let v1,s1' = C(e1)(s1) in
  let v2,s2' = C(e2)(s2) in
  let _ = observe(v1,v2) in
  ((), (s1',s2'))

C(factor(e)) = fun s ->
  let v,s' = C(e)(s) in
  let _ = factor(v) in ((), s')

C(infer(e)) = fun sigma ->
  let mu,sigma' = infer(C(e), sigma) in
  (mu, sigma')
```

```
C(present e -> e1 else e2) =
fun (s,s1,s2) ->
  let v, s' = C(e)(s) in
  if v then let v1,s1' = C(e1)(s1) in
    (v1, (s',s1',s2))
  else let v2,s2' = C(e2)(s2) in
    (v2, (s',s1,s2'))
```

The Zelus compiler



Bourke, Pouzet 2013

Generated code

```
(* a synchronous stream function with type 'a -D→ 'b *)
(* is represented by an OCaml value of type ('a, 'b) node *)
type ('a, 'b) node =
    Node:
      { alloc : unit → 's; (* allocate the state *)
        step : 's → 'a → 'b; (* compute a step *)
        reset : 's → unit; (* reset/initialize the state *)
      } → ('a, 'b) cnode

(*
  let m = alloc () in
  reset m;
  while true do
    let o = step m i in ...
  done
*)
```

https://github.com/INRIA/zelus/blob/main/lib/std/ztypes.ml

Streaming inference

Reactive Probabilistic Programming

Particle filter

Approximate inference algorithm : importance sampling, but...

- Add a resampling step at each `observe`
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

How can we duplicate a particle during execution?

- Continuation Passing Style (CPS)?
- Clone the memory state?

Particle filter

Approximate inference algorithm : importance sampling, but...

- Add a resampling step at each `observe`
- Compute the score of the particles to compute a distribution
- Re-sample a new set of particles from this distribution

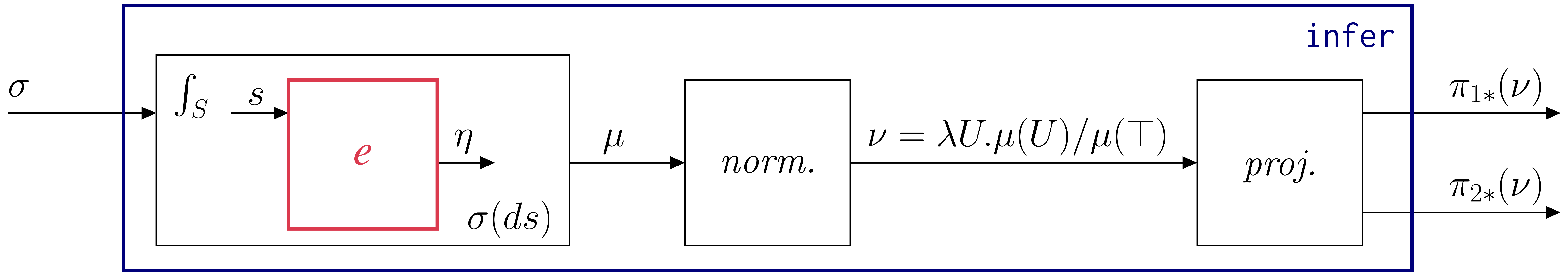
How can we duplicate a particle during execution?

- Continuation Passing Style (CPS)?
- Clone the memory state?

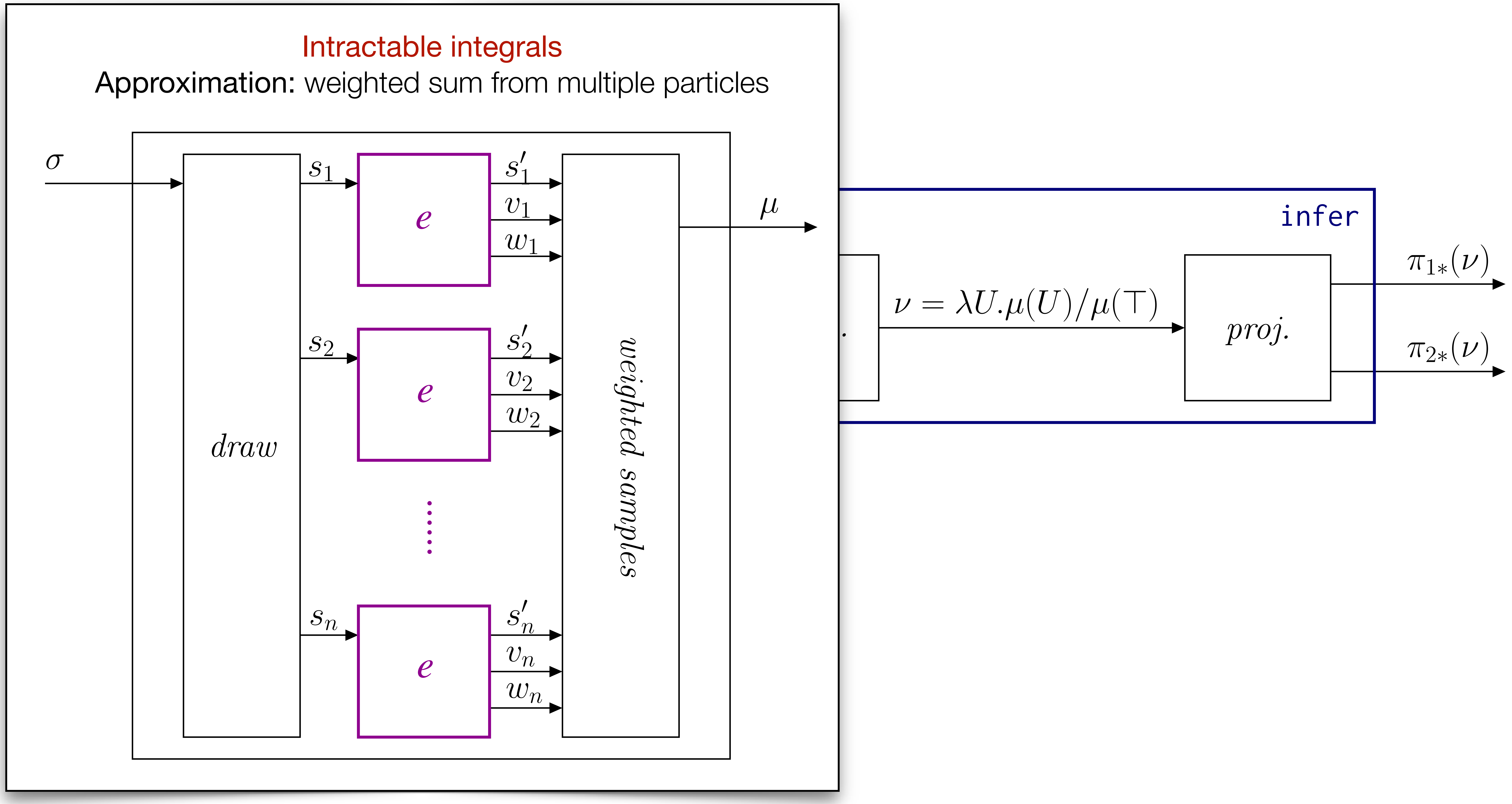
```
(* the same with a method copy *)
type ('a, 'b) cnode =
    Cnode:
      { alloc : unit → 's; (* allocate the state *)
        copy : 's → 's → unit; (* copy the source into the destination *)
        step : 's → 'a → 'b; (* compute a step *)
        reset : 's → unit; (* reset/initialize the state *)
      } → ('a, 'b) cnode
```

https://github.com/INRIA/zelus/blob/main/lib/std/ztypes.ml

Particle filter



Particle filter



Particle filter

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

Particle filter

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

$t = 0$

Particle filter

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

$t = 0$

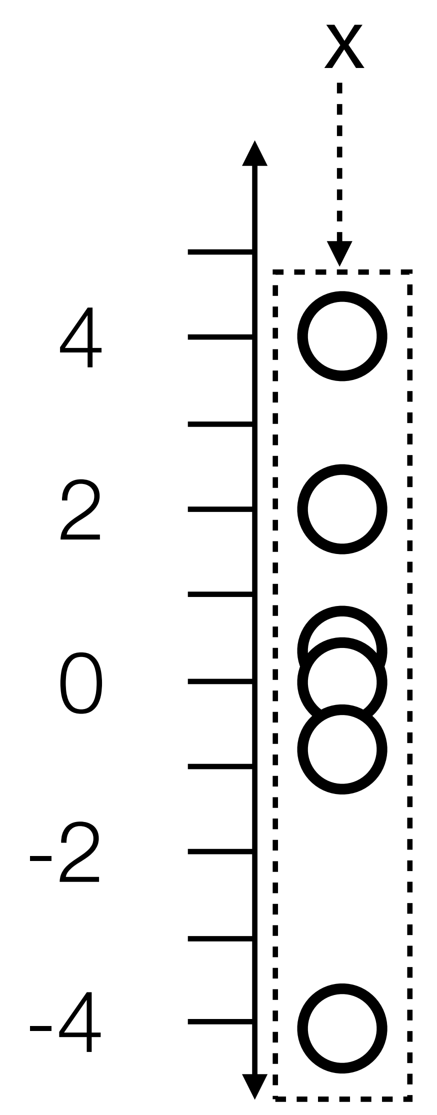
```
sample (gaussian (0, 10))
```

Particle filter

```
let 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))
```

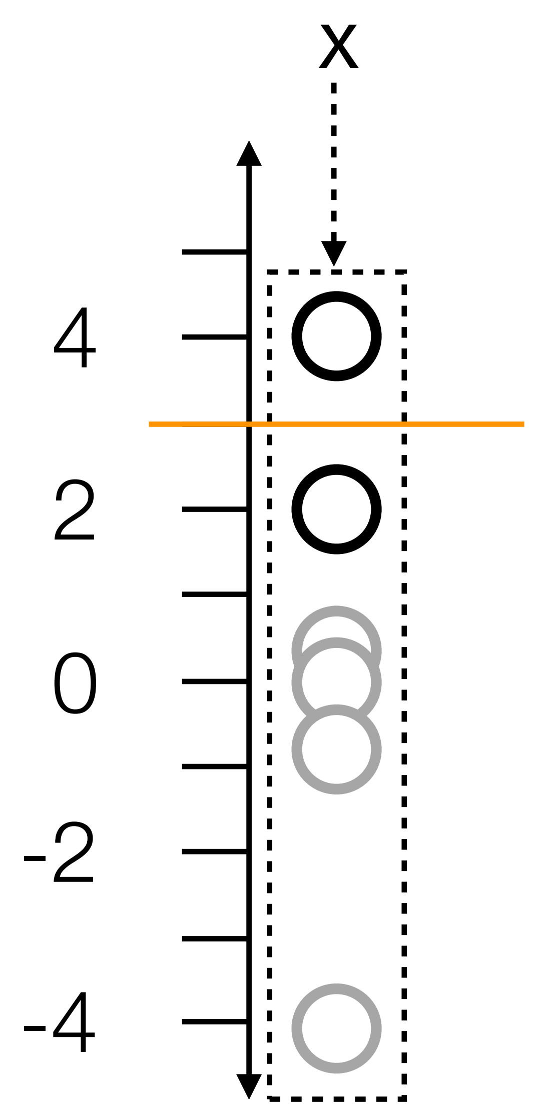


Particle filter

```
let 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)
```



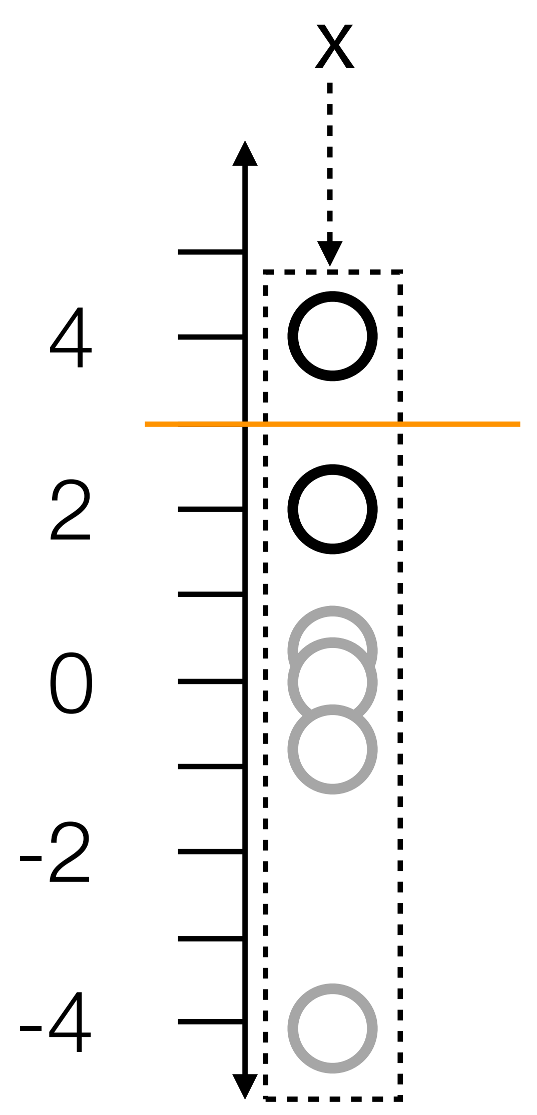
Particle filter

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

$t = 0$

$t = 1$

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



Particle filter

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

$t = 0$

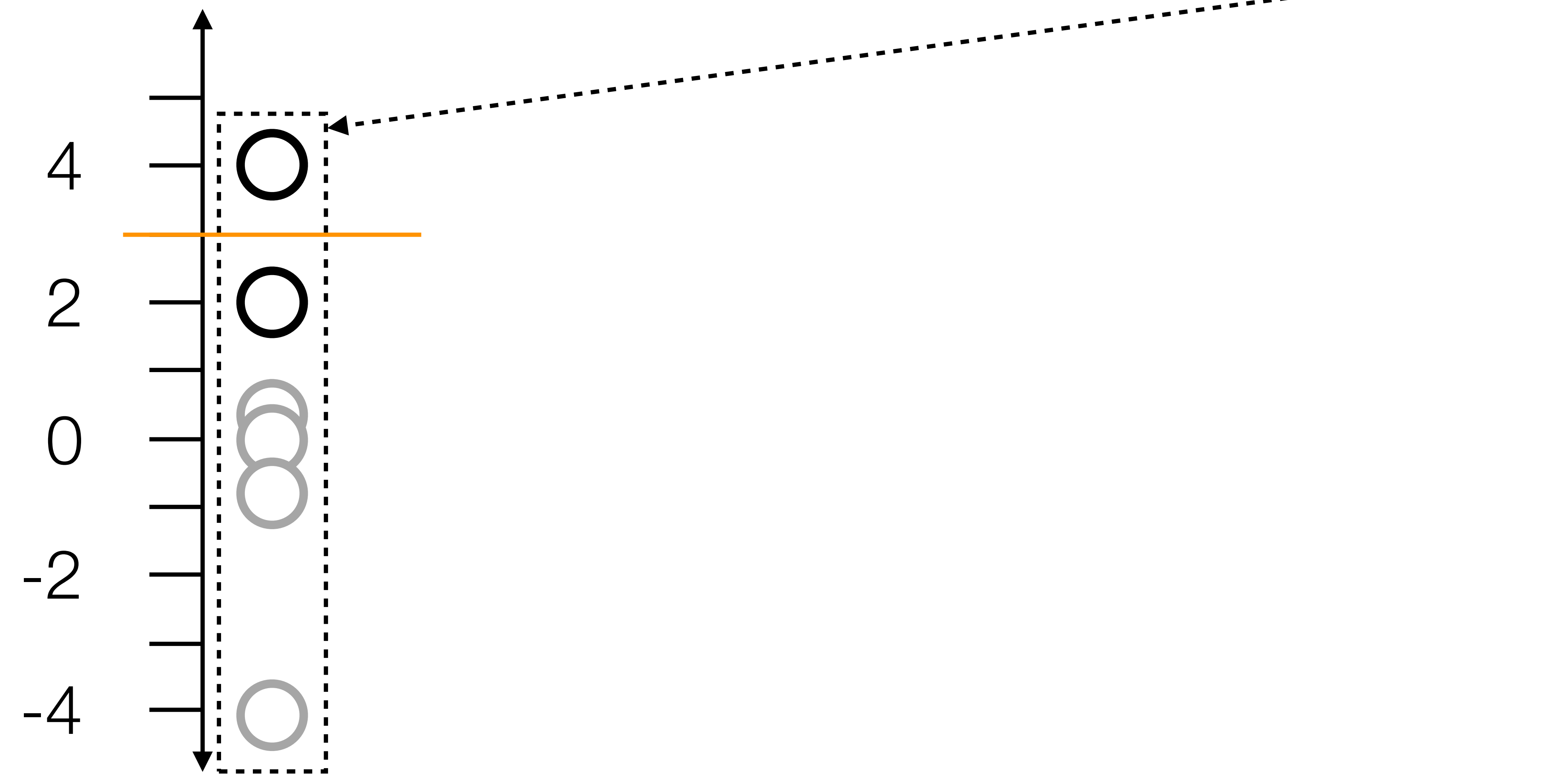
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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

x

pre x   x



Particle filter

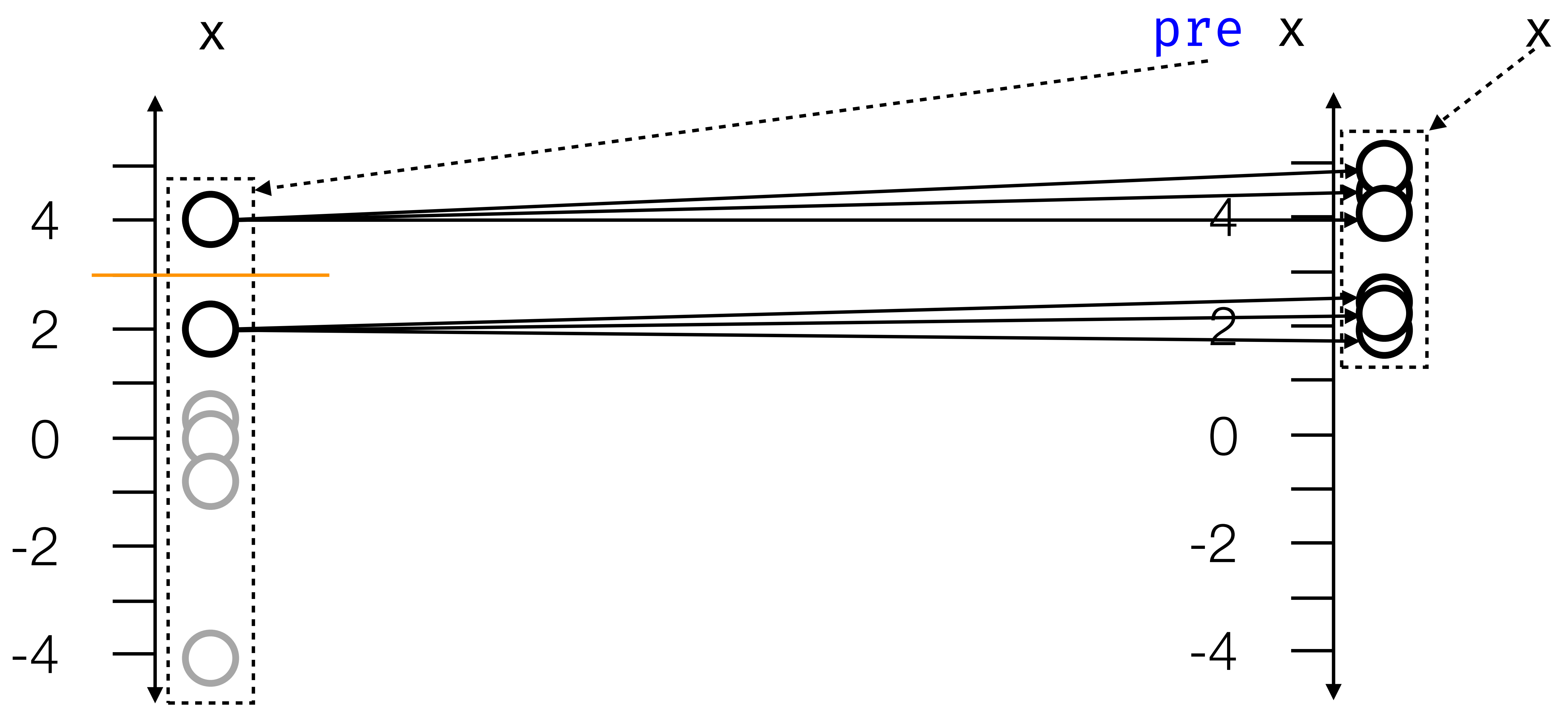
```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

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

$t = 1$

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



Particle filter

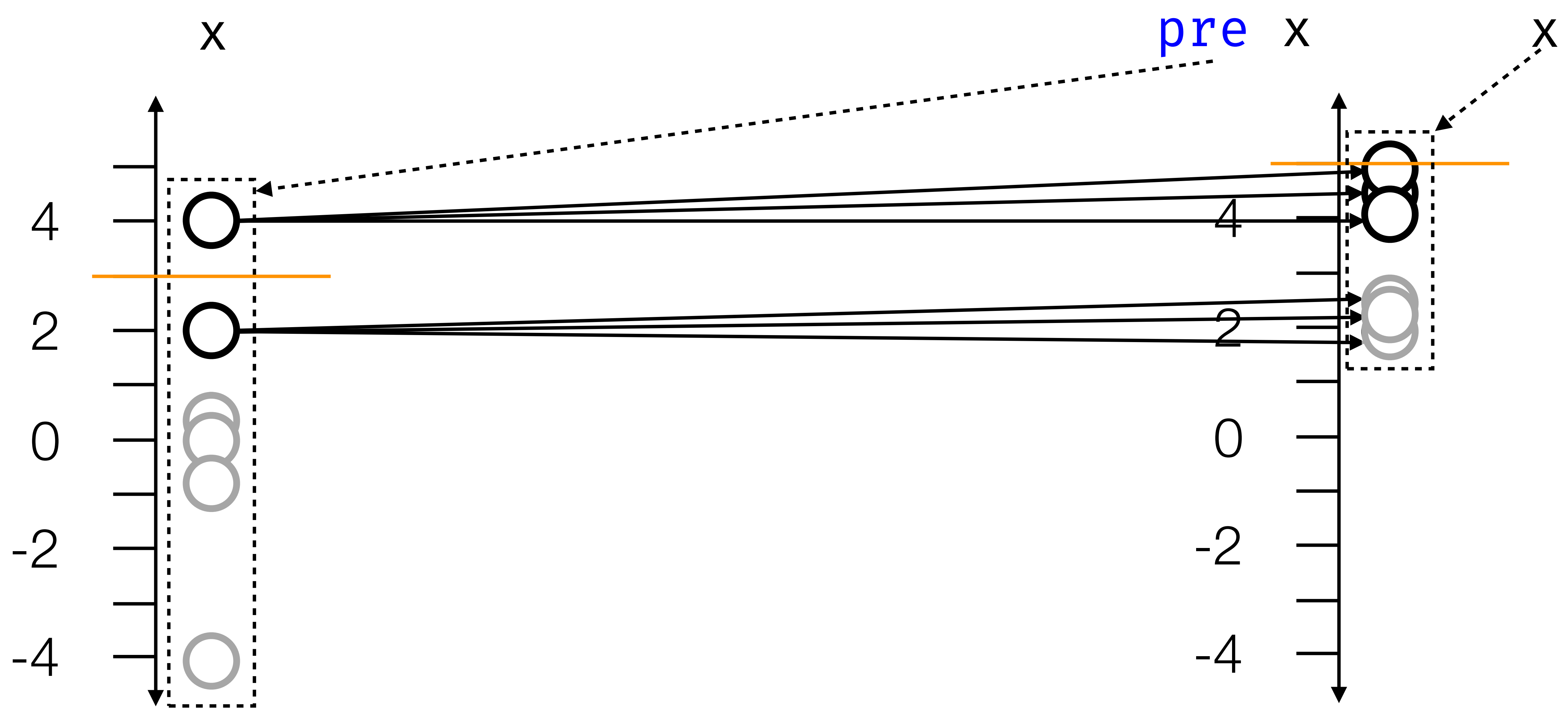
```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

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

$t = 1$

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



Particle filter

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

$t = 0$

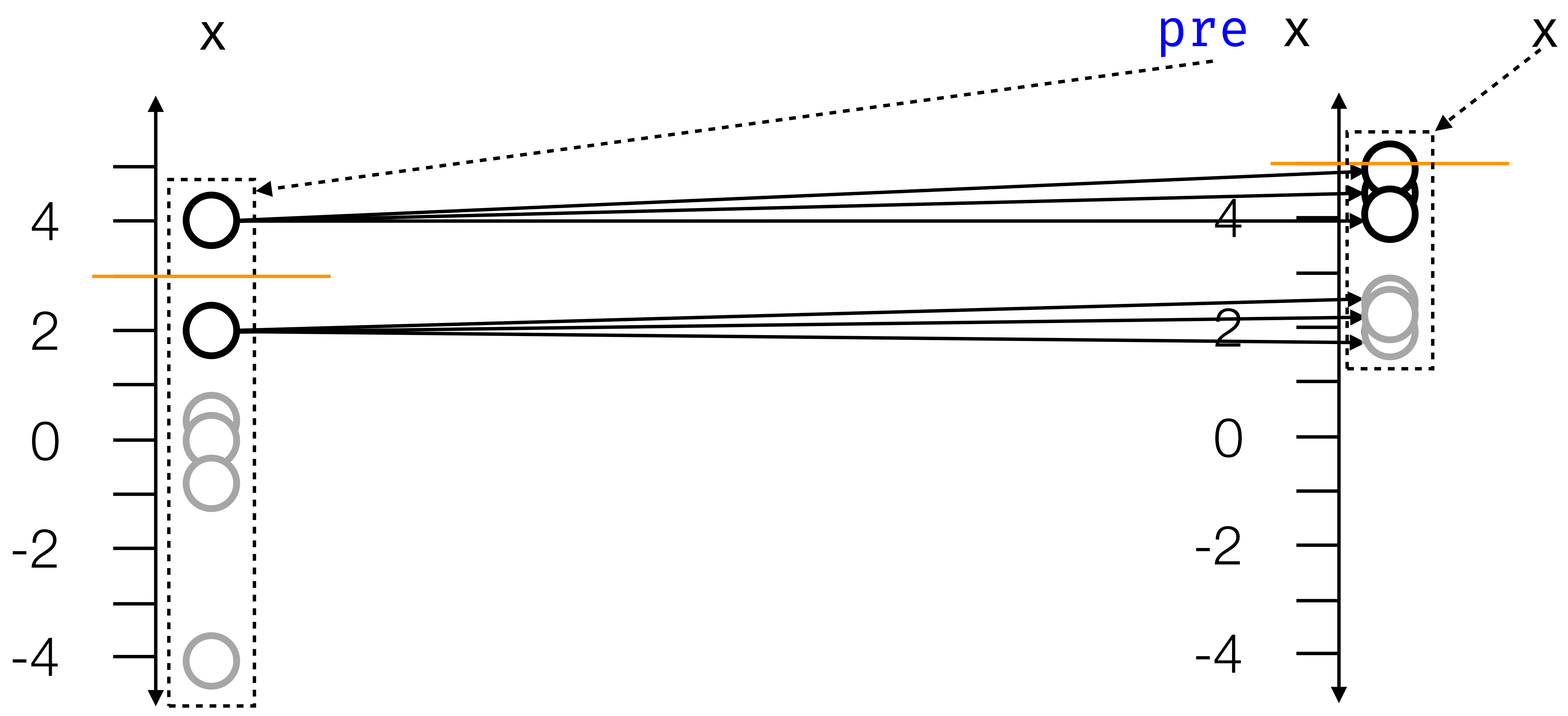
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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

$t = 2$

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



…

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

```
let 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

```
let 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

```
let 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))
```

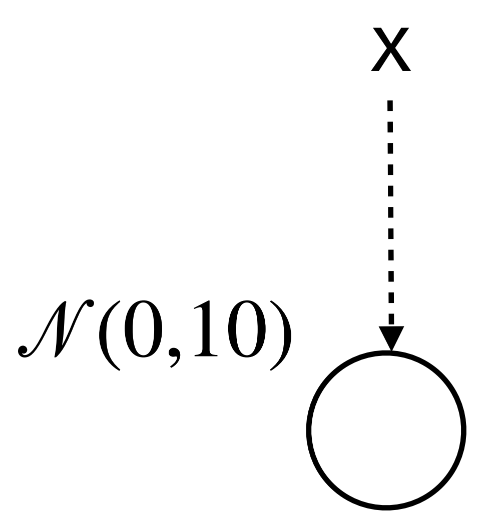
Murray et al. 2018

Delayed sampling

```
let 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))
```



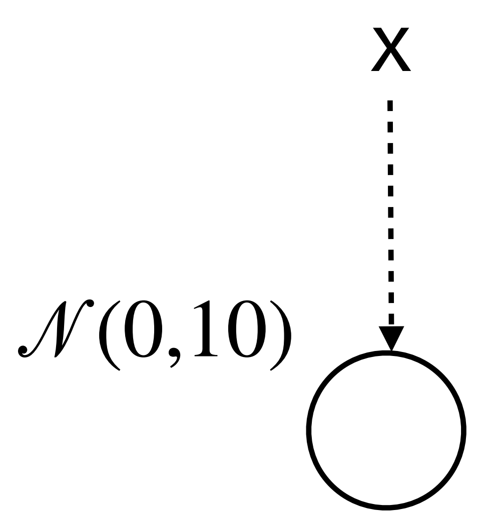
Murray et al. 2018

Delayed sampling

```
let 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)
```



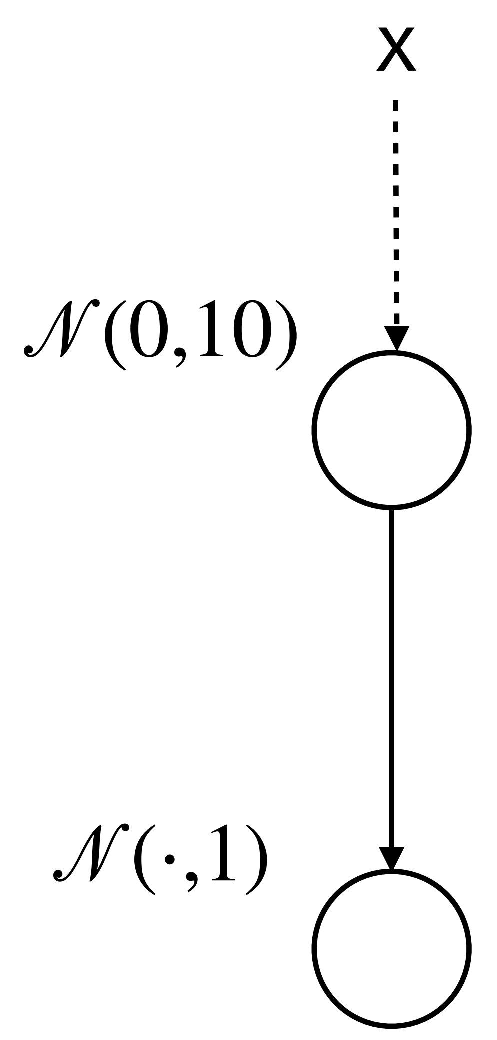
Murray et al. 2018

Delayed sampling

```
let 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)
```



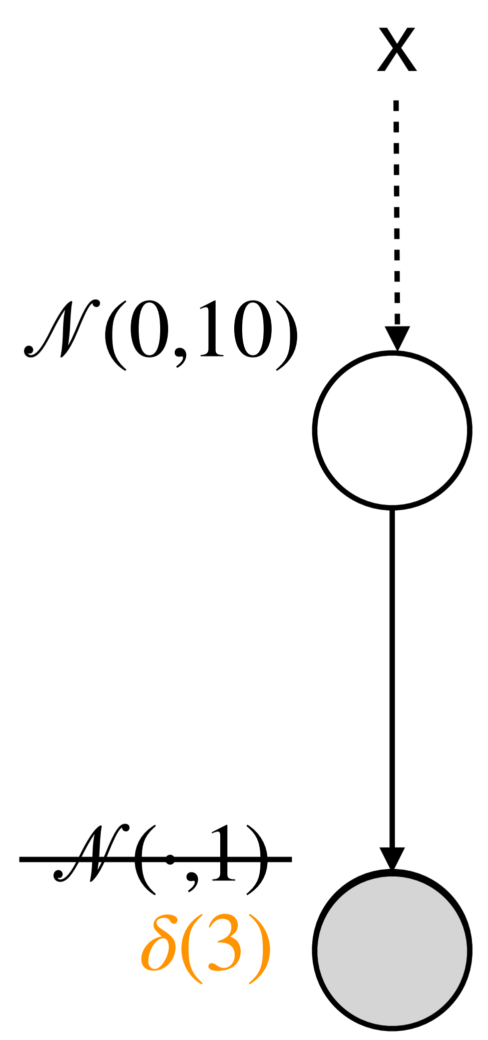
Murray et al. 2018

Delayed sampling

```
let 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)
```



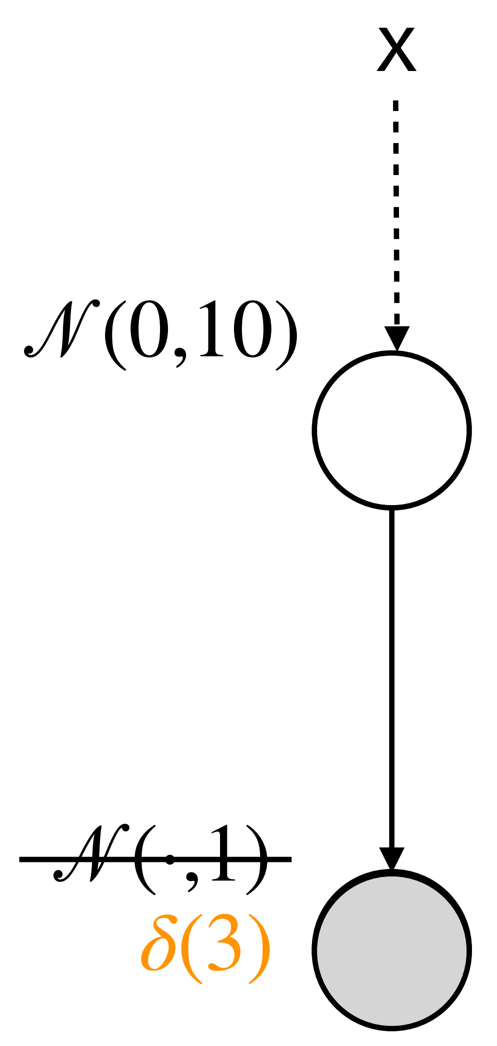
Murray et al. 2018

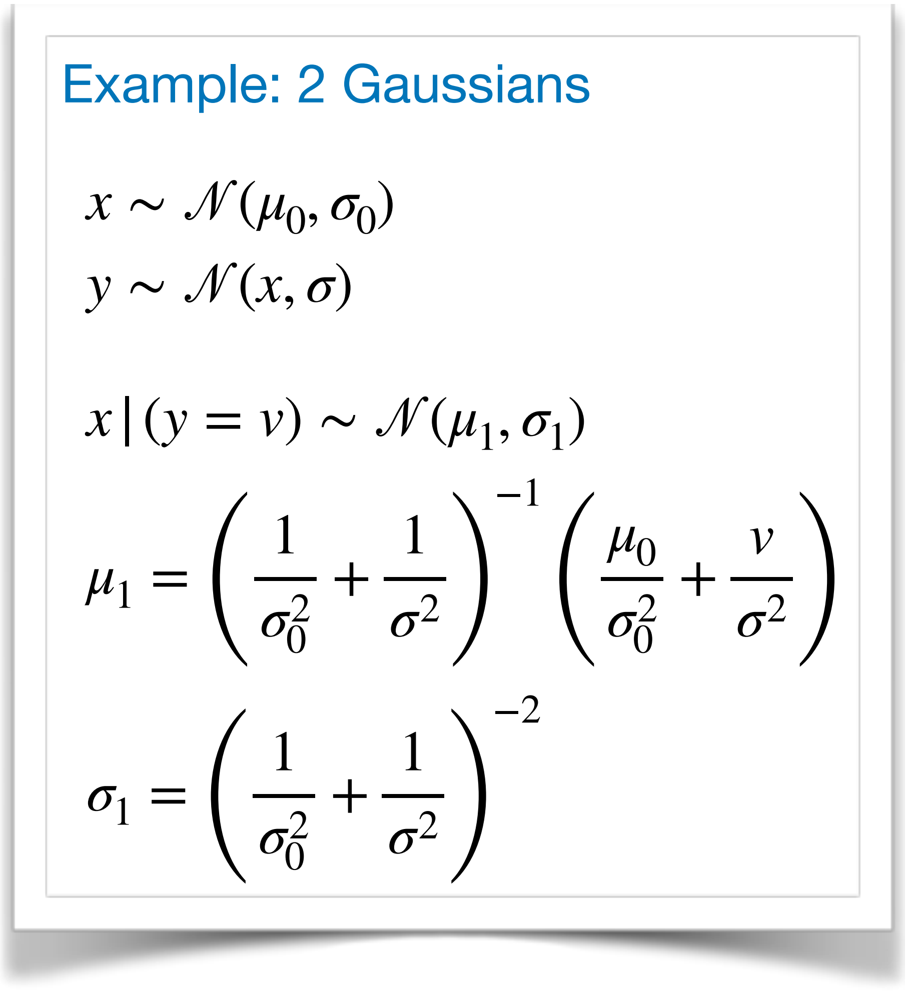
Delayed sampling

```
let 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

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

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

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

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

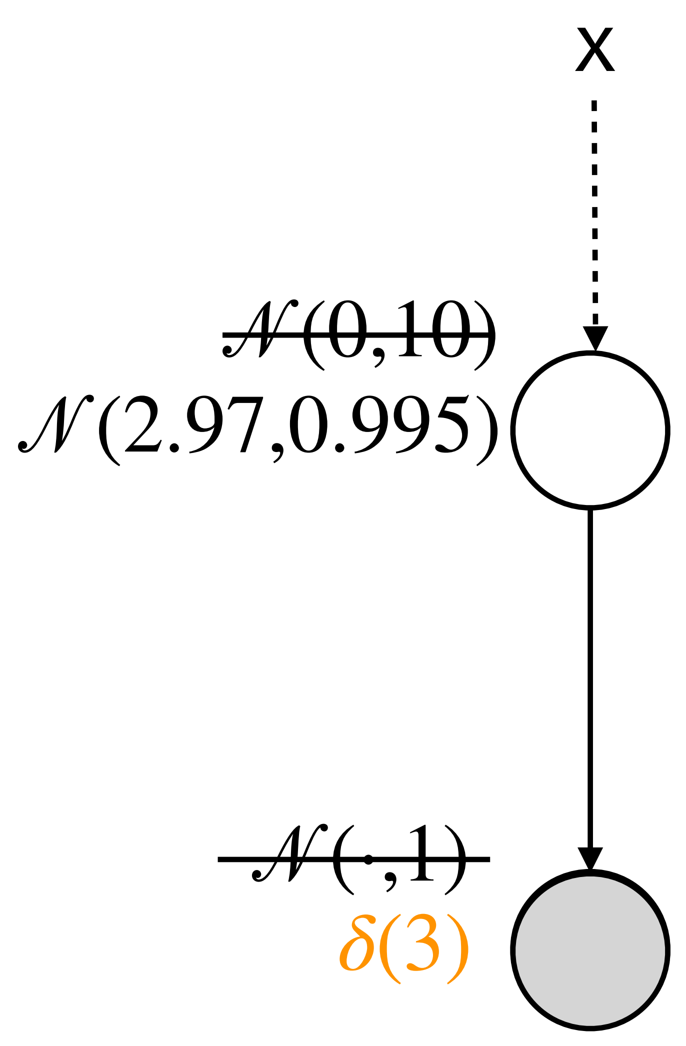
Murray et al. 2018

Delayed sampling

```
let 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

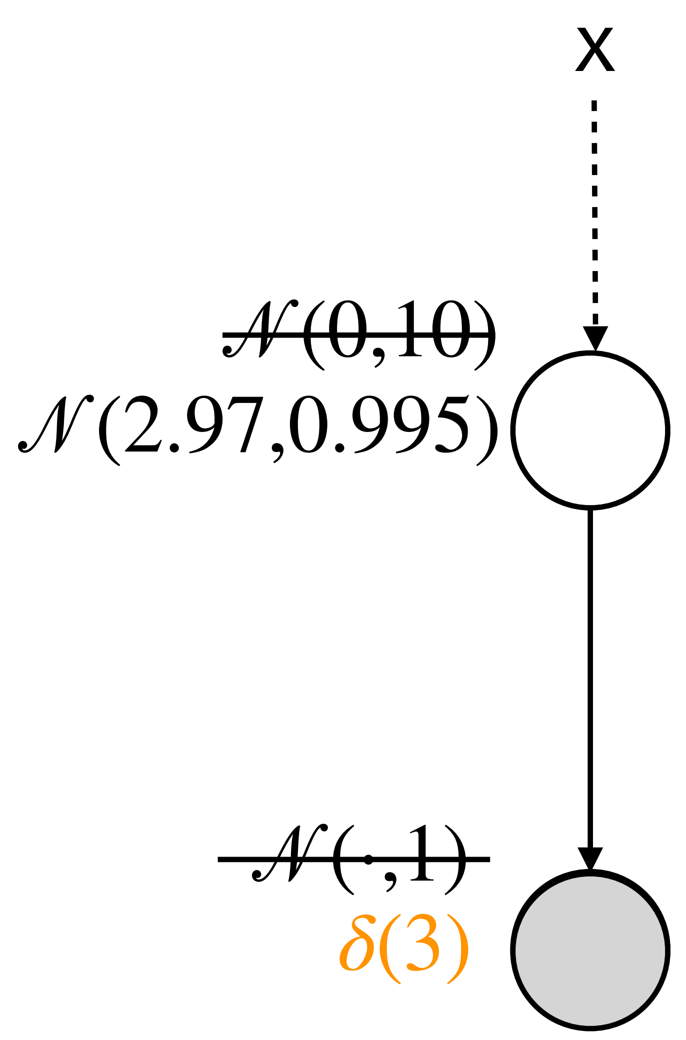
```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

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

$t = 1$

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



Murray et al. 2018

Delayed sampling

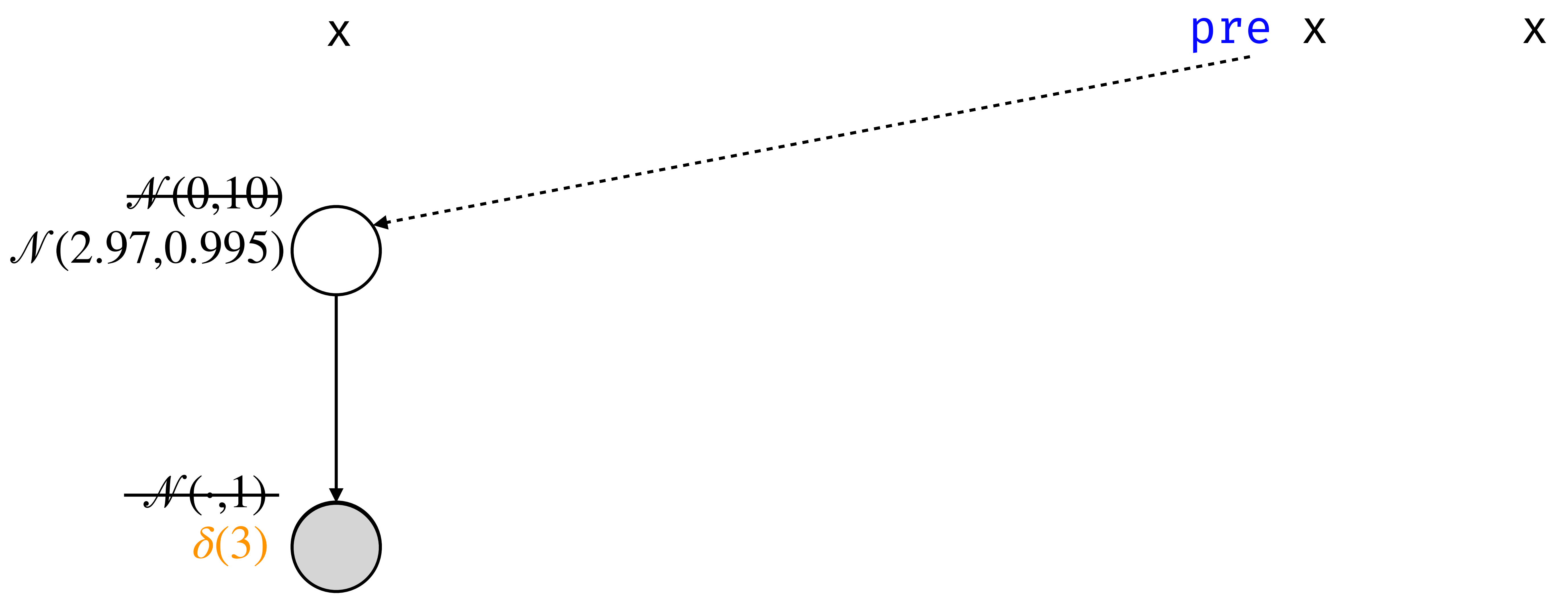
```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10)) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

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

$t = 1$

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



Murray et al. 2018

Delayed sampling

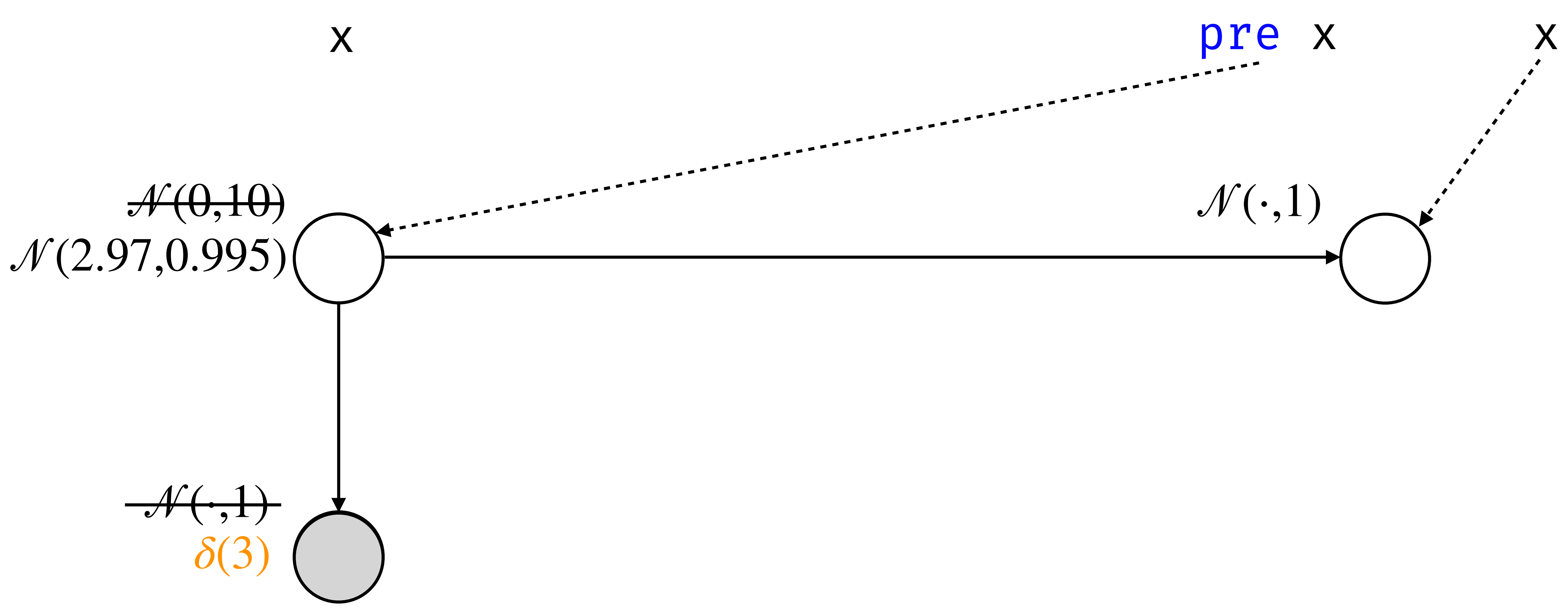
```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10)) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

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

$t = 1$

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



Murray et al. 2018

Delayed sampling

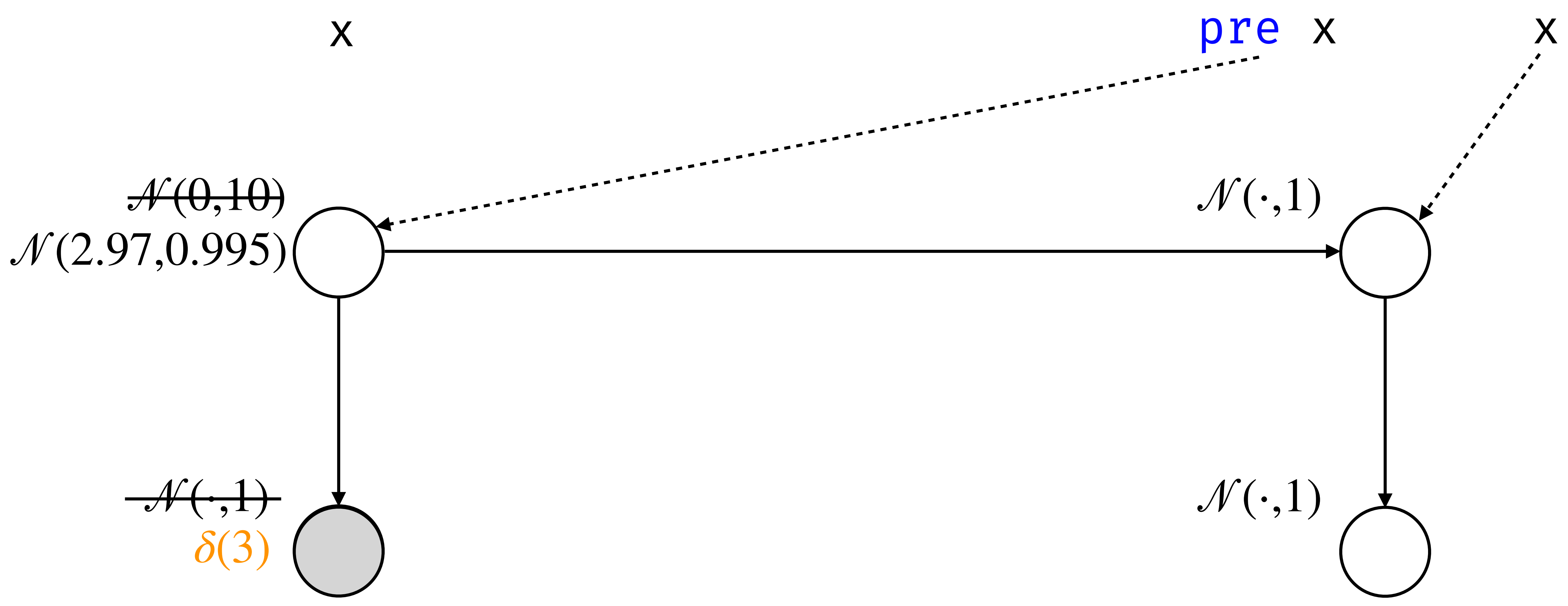
```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
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

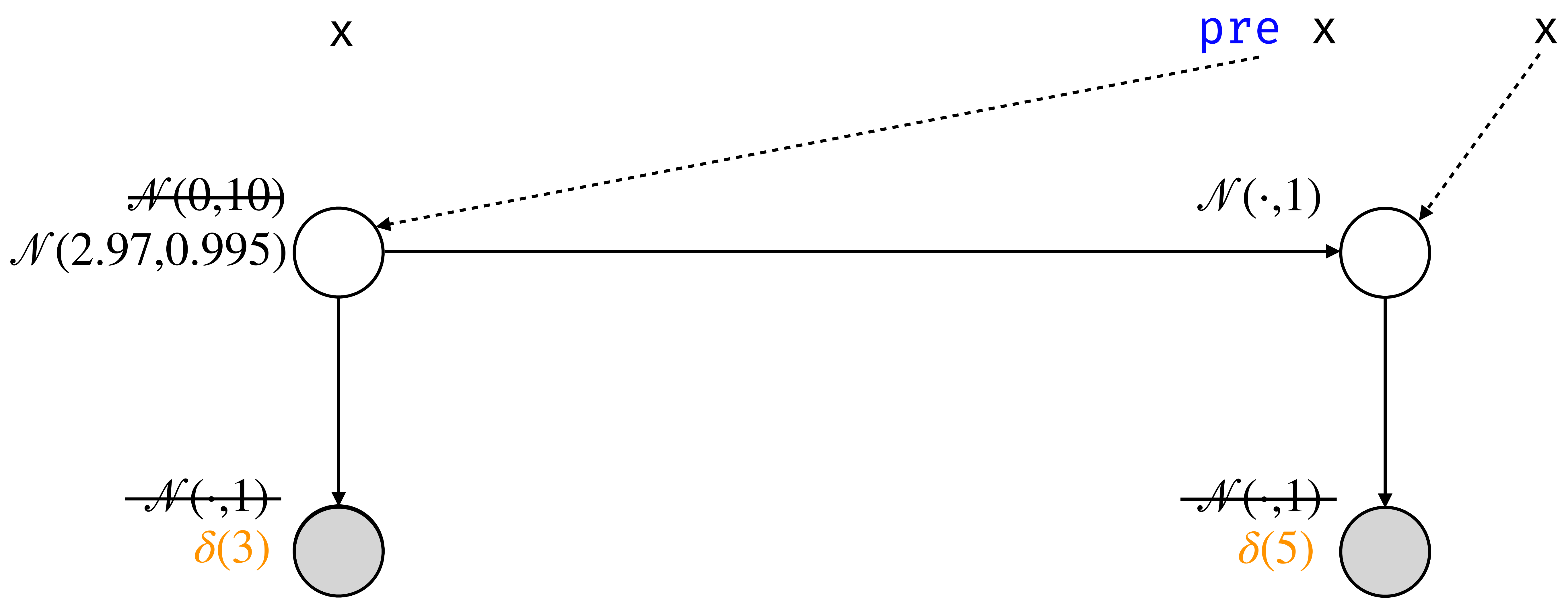
```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

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

$t = 1$

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


x
pre x
x
~~$\mathcal{N}(0,10)$~~
$\mathcal{N}(2.97,0.995)$
$\mathcal{N}(\cdot,1)$
~~$\mathcal{N}(\cdot,1)$~~
$\delta(3)$
~~$\mathcal{N}(\cdot,1)$~~
$\delta(5)$

Murray et al. 2018

Delayed sampling

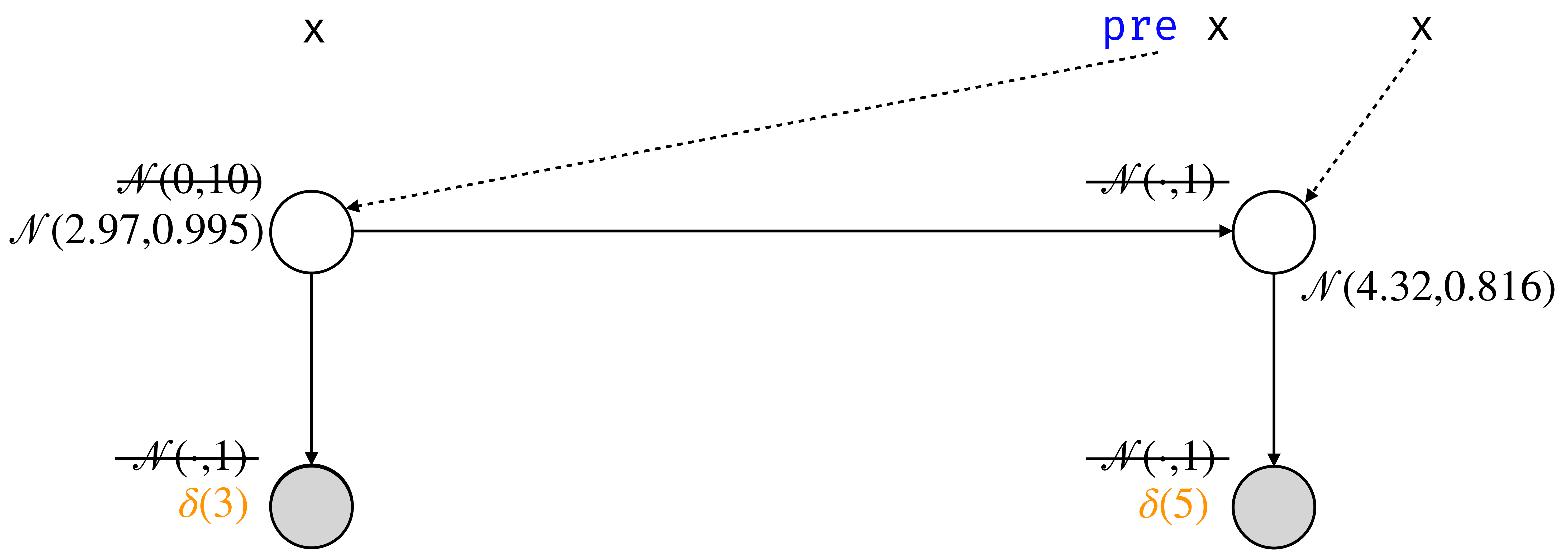
```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
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

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

$t = 0$

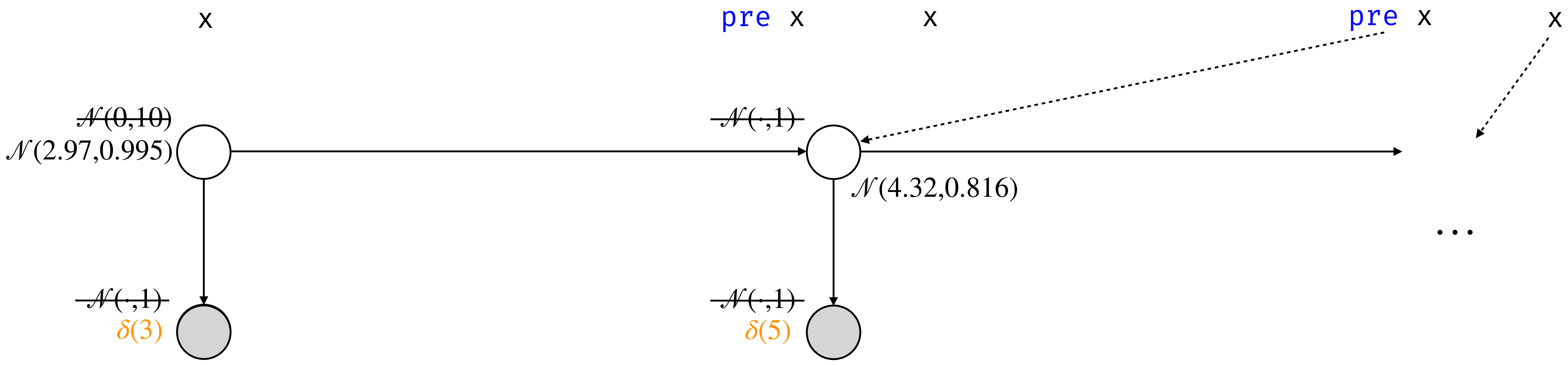
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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

$t = 2$

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


x
pre x
x
pre x
x
~~$\mathcal{N}(0,10)$~~
$\mathcal{N}(2.97,0.995)$
~~$\mathcal{N}(\cdot,1)$~~
$\mathcal{N}(4.32,0.816)$
~~$\mathcal{N}(\cdot,1)$~~
$\delta(3)$
~~$\mathcal{N}(\cdot,1)$~~
$\delta(5)$
…

Murray et al. 2018

Delayed sampling

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

$t = 0$

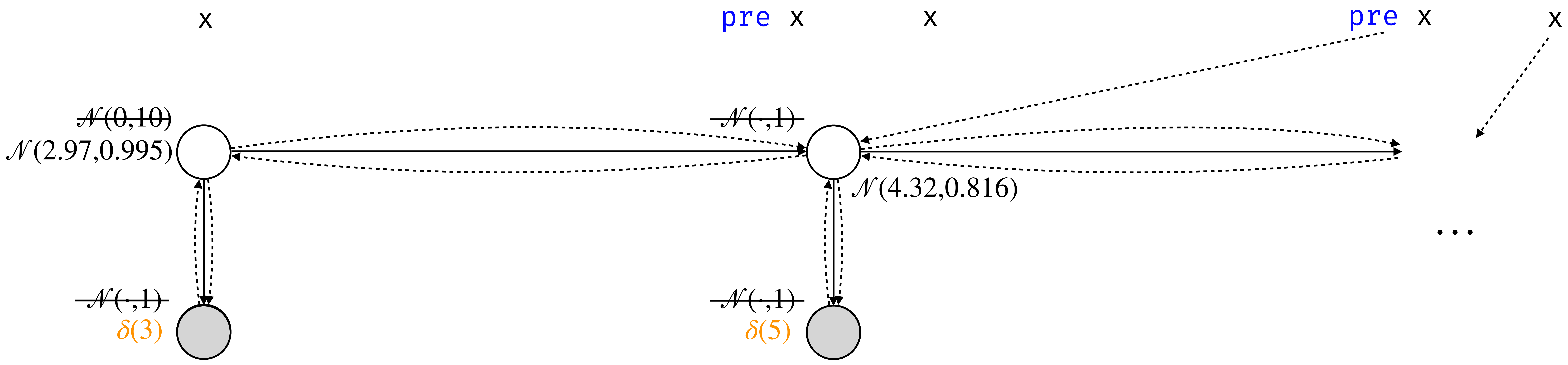
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

```
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

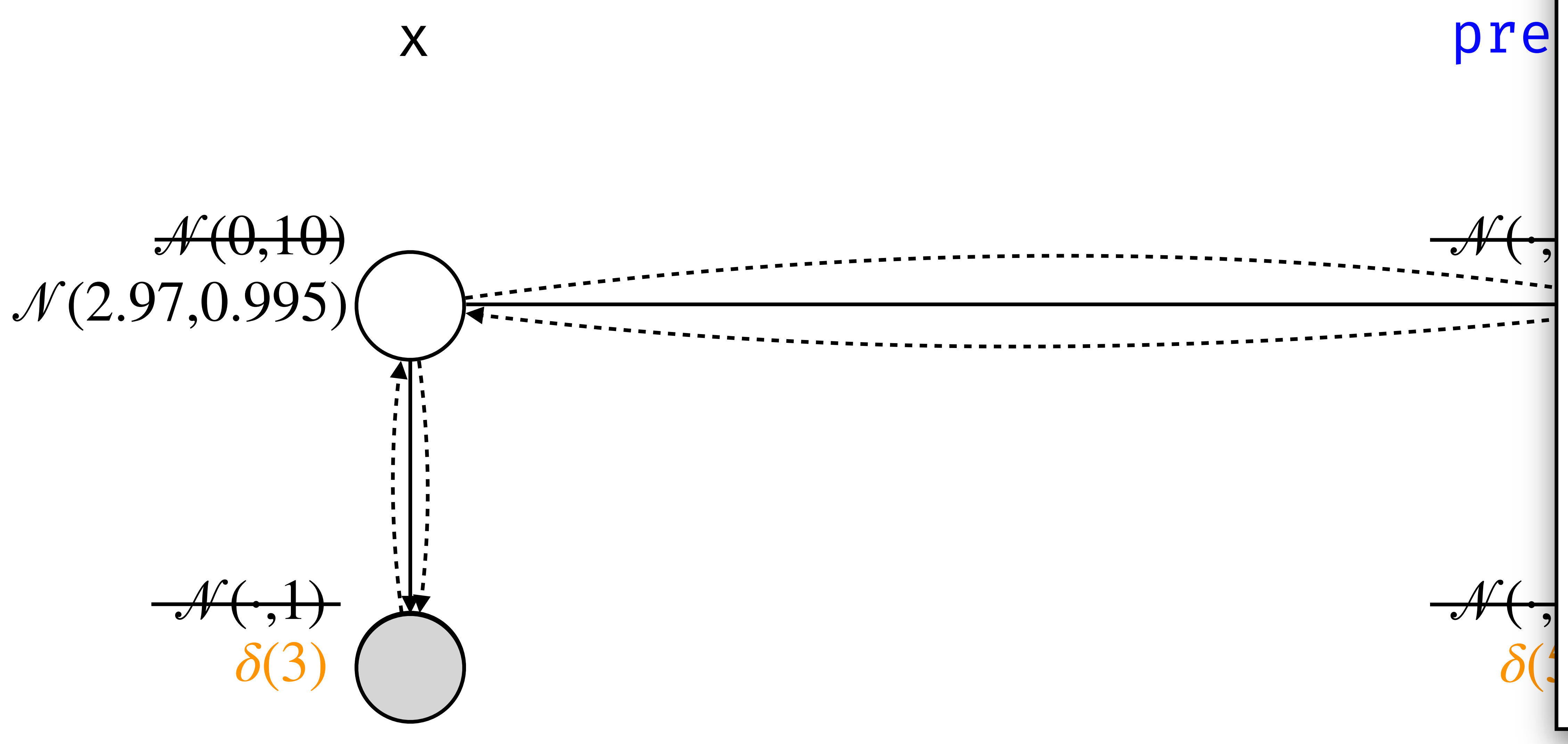
Delayed sampling

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

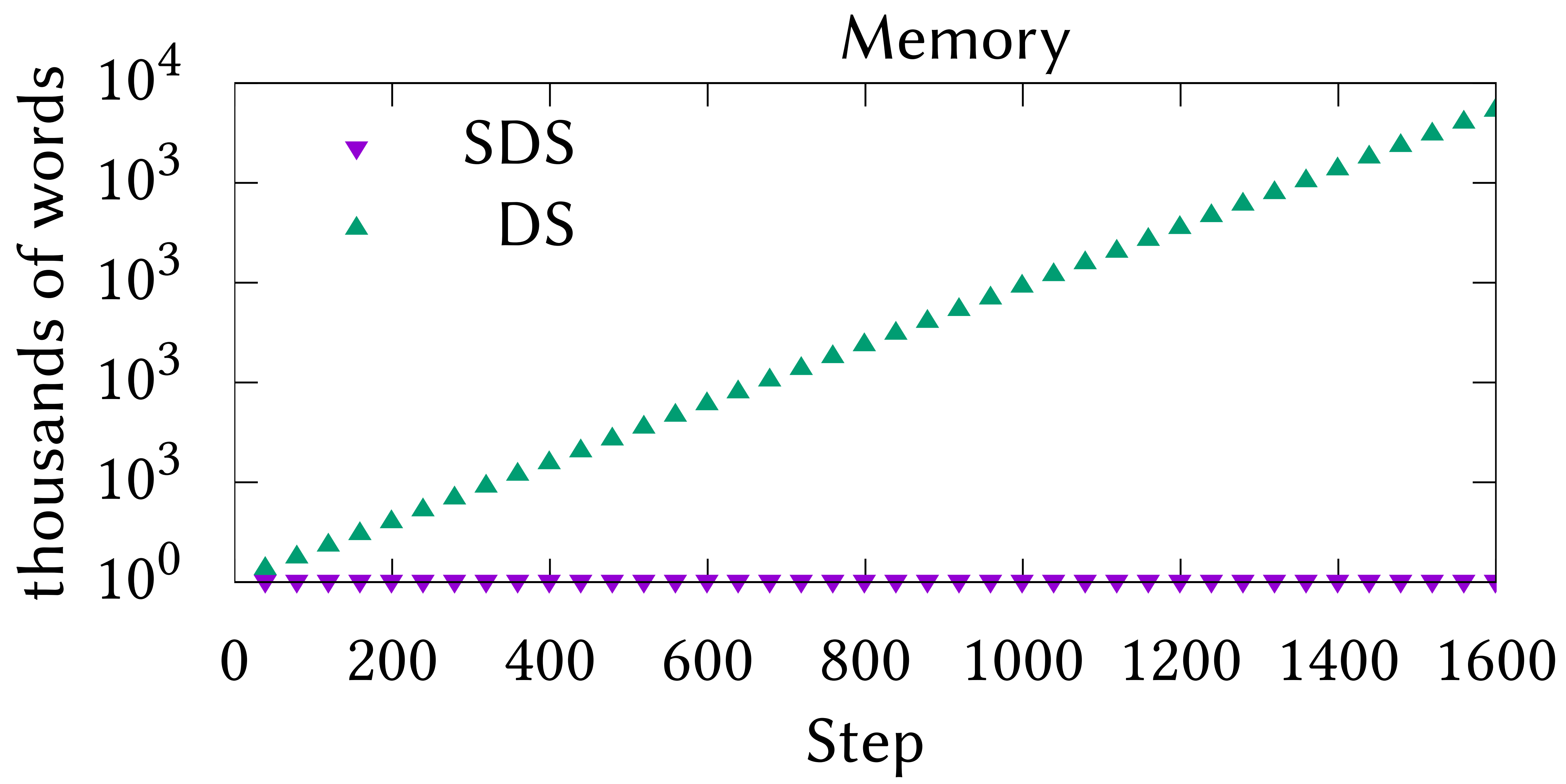
$t = 0$

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

```
sample (gaus
observe (gau
```



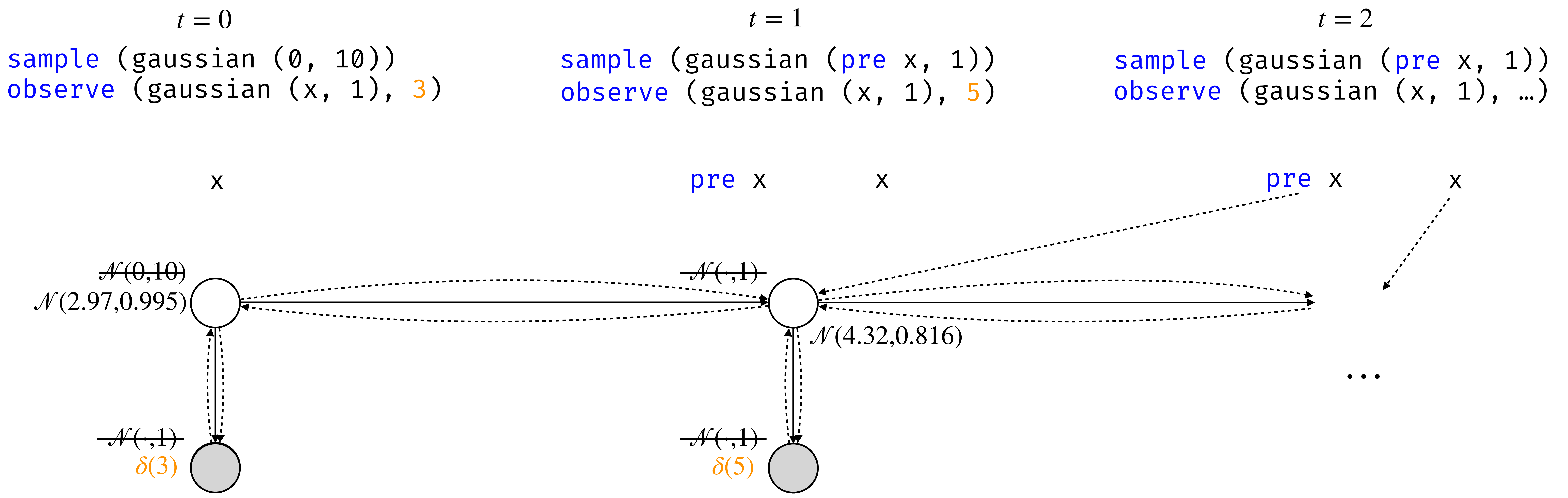
Unbounded resources



Murray et al. 2018

Delayed sampling

```
let 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), …)
x
pre x
x
pre x
x
~~$\mathcal{N}(0,10)$~~
$\mathcal{N}(2.97,0.995)$
~~$\mathcal{N}(\cdot,1)$~~
$\mathcal{N}(4.32,0.816)$
~~$\mathcal{N}(\cdot,1)$~~
$\delta(3)$
~~$\mathcal{N}(\cdot,1)$~~
$\delta(5)$
…

Murray et al. 2018

Streaming
Delayed sampling

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

$t = 0$

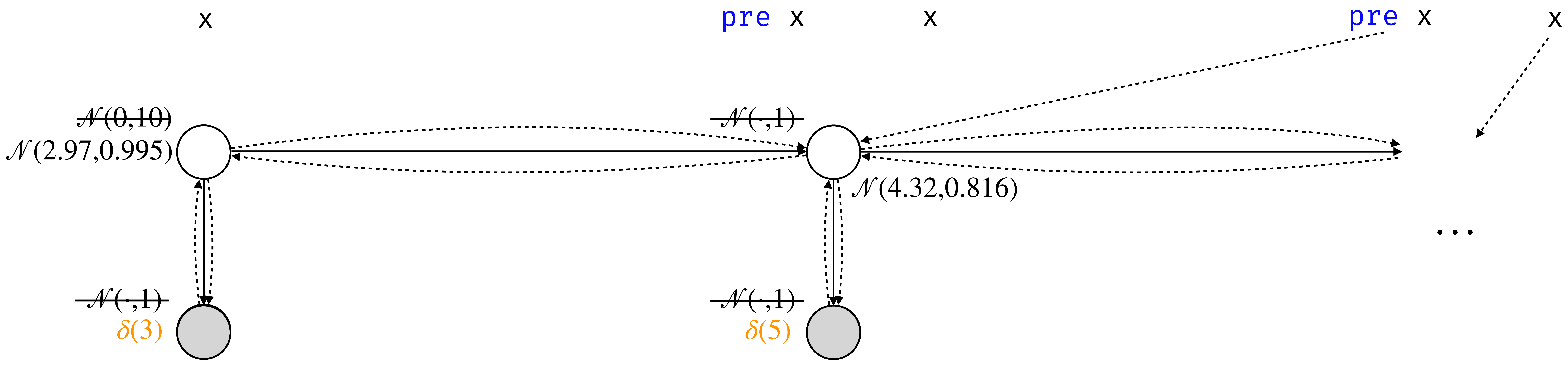
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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

$t = 2$

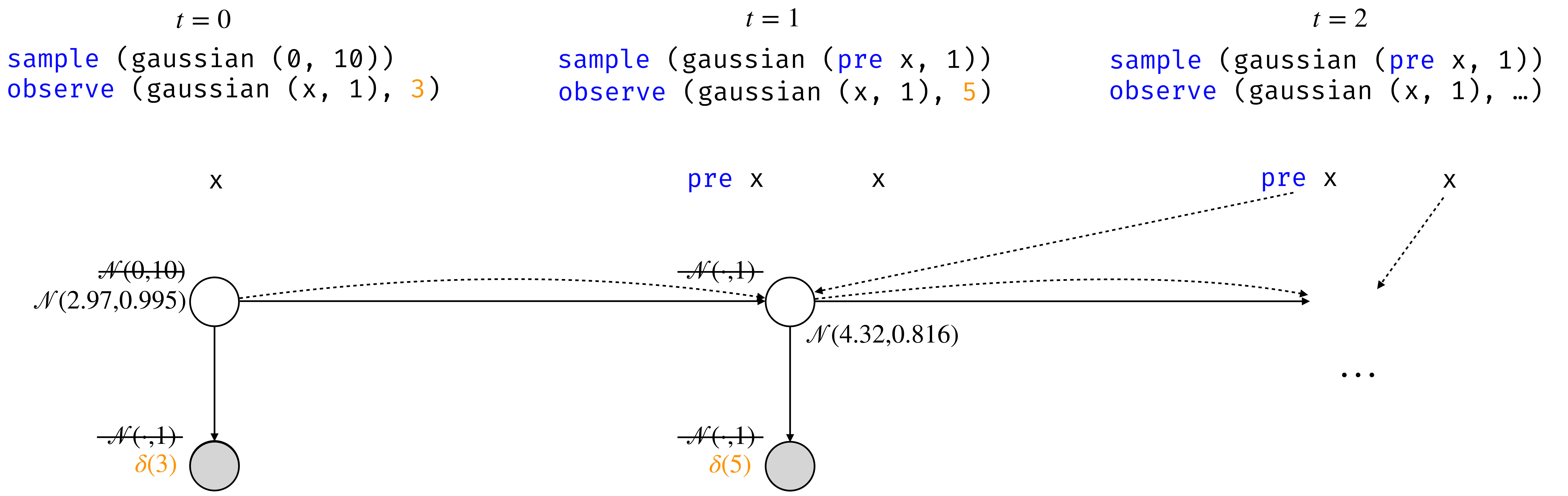
```
sample (gaussian (pre x, 1))
observe (gaussian (x, 1), …)
```



x
pre x
x
pre x
x
~~$\mathcal{N}(0{,}10)$~~
$\mathcal{N}(2.97{,}0.995)$
~~$\mathcal{N}(\cdot{,}1)$~~
$\mathcal{N}(4.32{,}0.816)$
~~$\mathcal{N}(\cdot{,}1)$~~
$\delta(3)$
~~$\mathcal{N}(\cdot{,}1)$~~
$\delta(5)$
…

Streaming
Delayed sampling

```
let 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), …)
x
pre x
x
pre x
x
~~$\mathcal{N}(0,10)$~~
$\mathcal{N}(2.97,0.995)$
~~$\mathcal{N}(\cdot,1)$~~
$\delta(3)$
~~$\mathcal{N}(\cdot,1)$~~
$\mathcal{N}(4.32,0.816)$
~~$\mathcal{N}(\cdot,1)$~~
$\delta(5)$
…

Streaming
Delayed sampling

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

$t = 0$

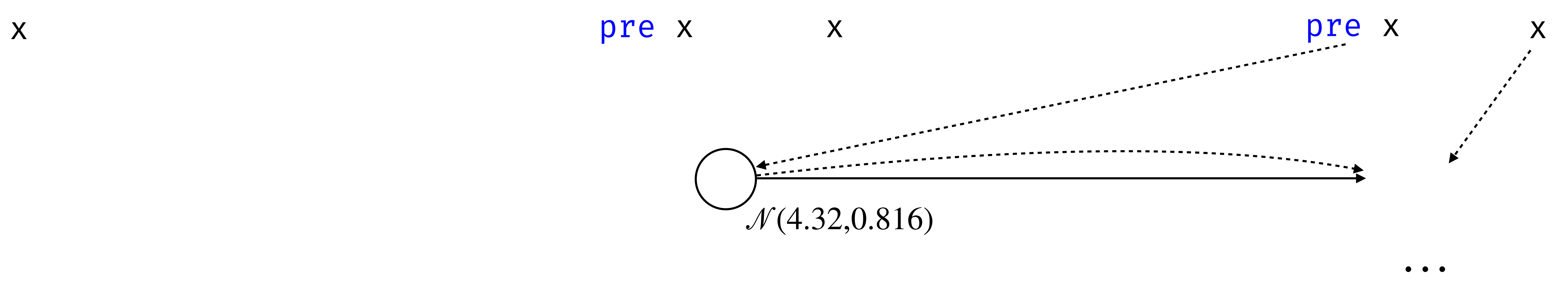
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

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

$t = 2$

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



Delayed Sampling semantics

$$\{\!\!\{op(e)\}\!\!\}_{\gamma,g,w} =$$
$$\quad let\ (e', g_e, w_e) = \{\!\!\{e\}\!\!\}_{\gamma,g,w}\ in\ (\mathsf{app}(op, e'), g_e, w_e)$$

$$\{\!\!\{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\}\!\!\}_{\gamma,g,w} =$$
$$\begin{aligned}&let\ e', g_e, w_e = \{\!\!\{e\}\!\!\}_{\gamma,g,w}\ in\\ &let\ v, g_v = value(e', g_e)\ in\\ &if\ v\ then\ \{\!\!\{e_1\}\!\!\}_{\gamma,g_v,w_e}\ else\ \{\!\!\{e_2\}\!\!\}_{\gamma,g_v,w_e}\end{aligned}$$

$$\{\!\!\{\texttt{sample}(e)\}\!\!\}_{\gamma,g,w} =$$
$$\begin{aligned}&let\ \mu, g_e, w' = \{\!\!\{e\}\!\!\}_{\gamma,g,w}\ in\\ &let\ X, g' = assume(\mu, g_e)\ in\ (X, g', w')\end{aligned}$$

$$\{\!\!\{\texttt{observe}(e_1, e_2)\}\!\!\}_{\gamma,g,w} =$$
$$\begin{aligned}&let\ \mu, g_1, w_1 = \{\!\!\{e_1\}\!\!\}_{\gamma,g,w}\ in\ let\ X, g_x = assume(\mu, g_1)\ in\\ &let\ e_2', g_2, w_2 = \{\!\!\{e_2\}\!\!\}_{\gamma,g_x,w_1}\ in\ let\ v, g_v = value(e_2', g_2)\ in\\ &let\ g' = observe(X, v, g_v)\ in\ ((), g', w_2 * \mu_{\mathrm{pdf}}(v))\end{aligned}$$

$$[\![\texttt{infer(fun}\ s\ \texttt{->}\ e,\ \sigma)]\!]_\gamma =$$
$$\begin{aligned}&let\ \mu = \lambda U.\ \sum_{i=1}^{N}\ let\ s_i, g_i = \mathrm{draw}([\![\sigma]\!]_\gamma)\ in\\ &\qquad let\ (e_i, s_i'), w_i, g_i' = \{\!\!\{\texttt{fun}\ s\ \texttt{->}\ e\}\!\!\}_{\gamma,1,g_i}(s_i)\ in\\ &\qquad let\ d_i = distribution(e_i, g_i')\ in\\ &\qquad \overline{w_i} * d_i(\pi_1(U)) * \delta_{s_i', g_i'}(\pi_2(U))\\ &in\ (\pi_{1*}(\mu), \pi_{2*}(\mu))\end{aligned}$$

$$\overline{w_i} = w_i / \sum_{i=1}^{N} w_i$$

Delayed Sampling semantics

$$\{\!\!\{ op(e) \}\!\!\}_{\gamma,g,w} = \quad let\ (e', g_e, w_e) = \{\!\!\{ e \}\!\!\}_{\gamma,g,w}\ in\ (\mathrm{app}(op, e'), g_e, w_e)$$

$$\begin{aligned}\{\!\!\{ \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \}\!\!\}_{\gamma,g,w} = \\ &let\ e', g_e, w_e = \{\!\!\{ e \}\!\!\}_{\gamma,g,w}\ in \\ &let\ v, g_v = value(e', g_e)\ in \\ &if\ v\ then\ \{\!\!\{ e_1 \}\!\!\}_{\gamma,g_v,w_e}\ else\ \{\!\!\{ e_2 \}\!\!\}_{\gamma,g_v,w_e}\end{aligned}$$

$$\begin{aligned}\{\!\!\{ \texttt{sample}(e) \}\!\!\}_{\gamma,g,w} = \\ &let\ \mu, g_e, w' = \{\!\!\{ e \}\!\!\}_{\gamma,g,w}\ in \\ &let\ X, g' = assume(\mu, g_e)\ in\ (X, g', w')\end{aligned}$$

$$\begin{aligned}\{\!\!\{ \texttt{observe}(e_1, e_2) \}\!\!\}_{\gamma,g,w} = \\ &let\ \mu, g_1, w_1 = \{\!\!\{ e_1 \}\!\!\}_{\gamma,g,w}\ in\ let\ X, g_x = assume(\mu, g_1)\ in \\ &let\ e_2', g_2, w_2 = \{\!\!\{ e_2 \}\!\!\}_{\gamma,g_x,w_1}\ in\ let\ v, g_v = value(e_2', g_2)\ in \\ &let\ g' = observe(X, v, g_v)\ in\ ((), g', w_2 * \mu_{\mathrm{pdf}}(v))\end{aligned}$$

$$\begin{aligned}[\![\texttt{infer}(\texttt{fun}\ s\ \texttt{->}\ e,\ \sigma)]\!]_{\gamma} = \\ &let\ \mu = \lambda U.\ \sum_{i=1}^{N}\ let\ s_i, g_i = \mathrm{draw}([\![\sigma]\!]_{\gamma})\ in \\ &\qquad let\ (e_i, s_i'), w_i, g_i' = \{\!\!\{ \texttt{fun}\ s\ \texttt{->}\ e \}\!\!\}_{\gamma,1,g_i}(s_i)\ in \\ &\qquad let\ d_i = distribution(e_i, g_i')\ in \\ &\qquad \overline{w_i} * d_i(\pi_1(U)) * \delta_{s_i', g_i'}(\pi_2(U)) \\ &in\ (\pi_{1*}(\mu), \pi_{2*}(\mu))\end{aligned}$$

Manipulate symbolic terms (e.g., app(+, ...))

$$\overline{w_i} = w_i / \sum_{i=1}^{N} w_i$$

Delayed Sampling semantics

$$
\begin{aligned}
&\{\!\![op(e)]\!\!\}_{\gamma,g,w} \ = \\
&\qquad \mathit{let}\ (e', g_e, w_e) = \{\!\![e]\!\!\}_{\gamma,g,w}\ \mathit{in}\ (\mathsf{app}(op, e'), g_e, w_e)
\end{aligned}
$$

$$
\begin{aligned}
&\{\!\![\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!\!\}_{\gamma,g,w} \ = \\
&\qquad \mathit{let}\ e', g_e, w_e = \{\!\![e]\!\!\}_{\gamma,g,w}\ \mathit{in} \\
&\qquad \mathit{let}\ v, g_v = \mathit{value}(e', g_e)\ \mathit{in} \\
&\qquad \mathit{if}\ v\ \mathit{then}\ \{\!\![e_1]\!\!\}_{\gamma,g_v,w_e}\ \mathit{else}\ \{\!\![e_2]\!\!\}_{\gamma,g_v,w_e}
\end{aligned}
$$

$$
\begin{aligned}
&\{\!\![\texttt{sample}(e)]\!\!\}_{\gamma,g,w} \ = \\
&\qquad \mathit{let}\ \mu, g_e, w' = \{\!\![e]\!\!\}_{\gamma,g,w}\ \mathit{in} \\
&\qquad \mathit{let}\ X, g' = \mathit{assume}(\mu, g_e)\ \mathit{in}\ (X, g', w')
\end{aligned}
$$

$$
\begin{aligned}
&\{\!\![\texttt{observe}(e_1, e_2)]\!\!\}_{\gamma,g,w} \ = \\
&\qquad \mathit{let}\ \mu, g_1, w_1 = \{\!\![e_1]\!\!\}_{\gamma,g,w}\ \mathit{in}\ \mathit{let}\ X, g_x = \mathit{assume}(\mu, g_1)\ \mathit{in} \\
&\qquad \mathit{let}\ e_2', g_2, w_2 = \{\!\![e_2]\!\!\}_{\gamma,g_x,w_1}\ \mathit{in}\ \mathit{let}\ v, g_v = \mathit{value}(e_2', g_2)\ \mathit{in} \\
&\qquad \mathit{let}\ g' = \mathit{observe}(X, v, g_v)\ \mathit{in}\ ((), g', w_2 * \mu_{\mathrm{pdf}}(v))
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{infer}(\texttt{fun}\ s\ \texttt{->}\ e,\ \sigma)]\!]_{\gamma} = \\
&\quad \mathit{let}\ \mu = \lambda U.\ \sum_{i=1}^{N}\
\begin{aligned}[t]
&\mathit{let}\ s_i, g_i = \mathrm{draw}([\![\sigma]\!]_{\gamma})\ \mathit{in} \\
&\mathit{let}\ (e_i, s_i'), w_i, g_i' = \{\!\![\texttt{fun}\ s\ \texttt{->}\ e]\!\!\}_{\gamma,1,g_i}(s_i)\ \mathit{in} \\
&\mathit{let}\ d_i = \mathit{distribution}(e_i, g_i')\ \mathit{in} \\
&\overline{w_i} * d_i(\pi_1(U)) * \delta_{s_i', g_i'}(\pi_2(U))
\end{aligned} \\
&\quad \mathit{in}\ (\pi_{1*}(\mu), \pi_{2*}(\mu))
\end{aligned}
$$

Manipulate symbolic terms (e.g., app(+, ...))

High-level API: graph manipulations
assume, observe, value

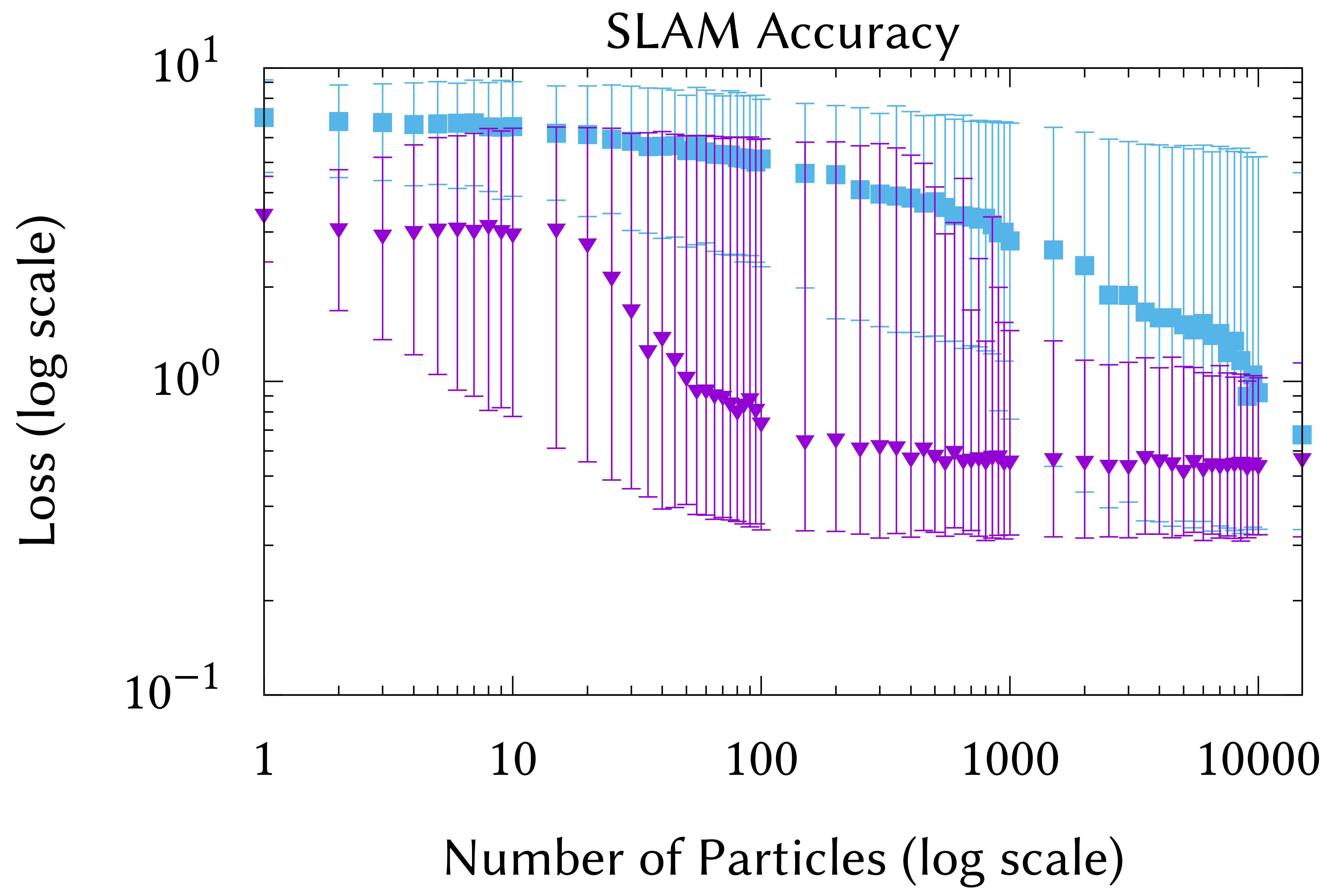
$$\overline{w_i} = w_i / \sum_{i=1}^{N} w_i$$

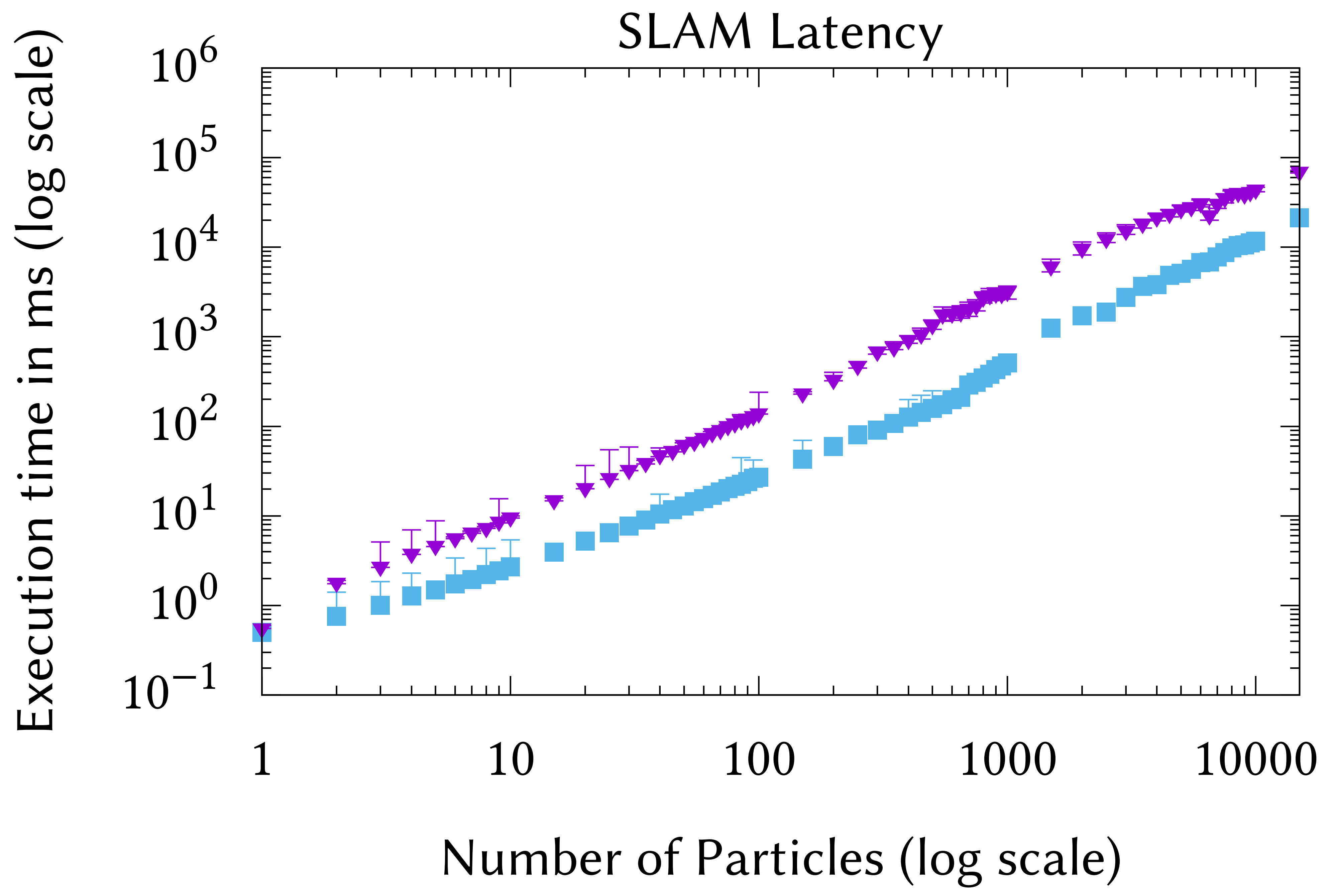
Evaluation

Algorithms comparison

■ PF Particle Filtering

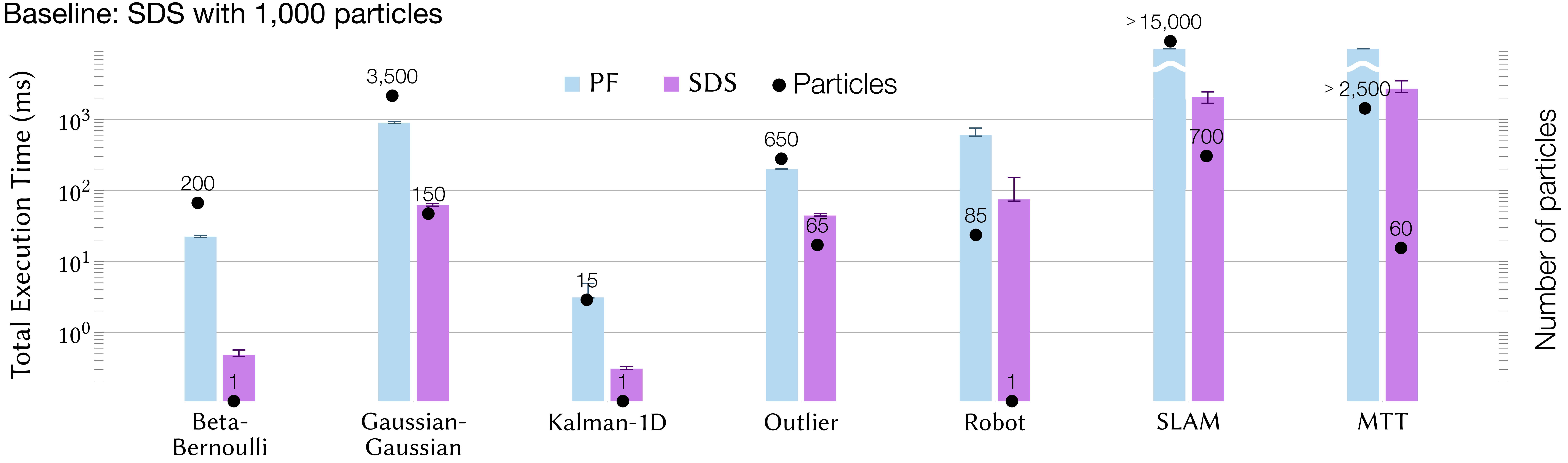
▼ SDS Streaming Delayed Sampling





Benchmarks

Baseline: SDS with 1,000 particles



Conclusions

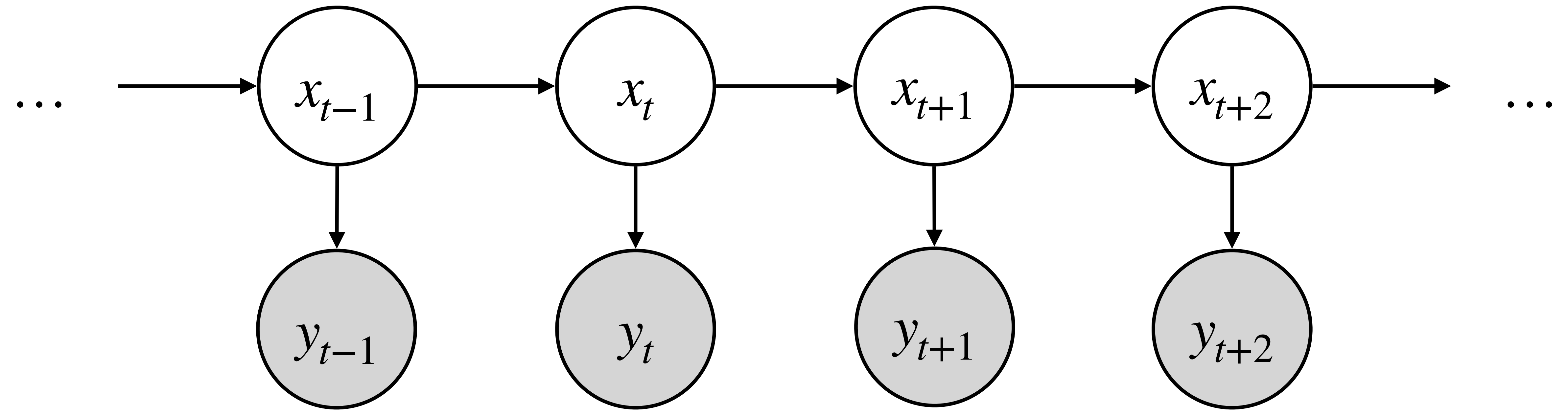
- SDS is always faster to match accuracy
- Reduction in particle count outweighs symbolic overhead
- SDS can be exact (1 particle)
- PF is impractical for advanced examples

Static analysis

Reactive Probabilistic Programming

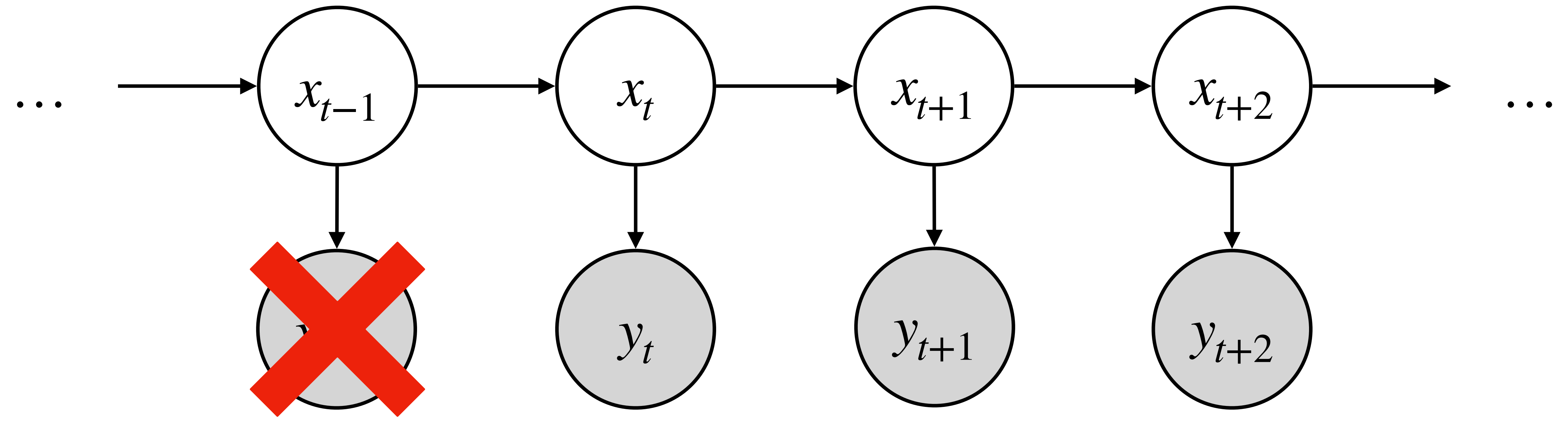
Bounded memory delayed sampling?

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



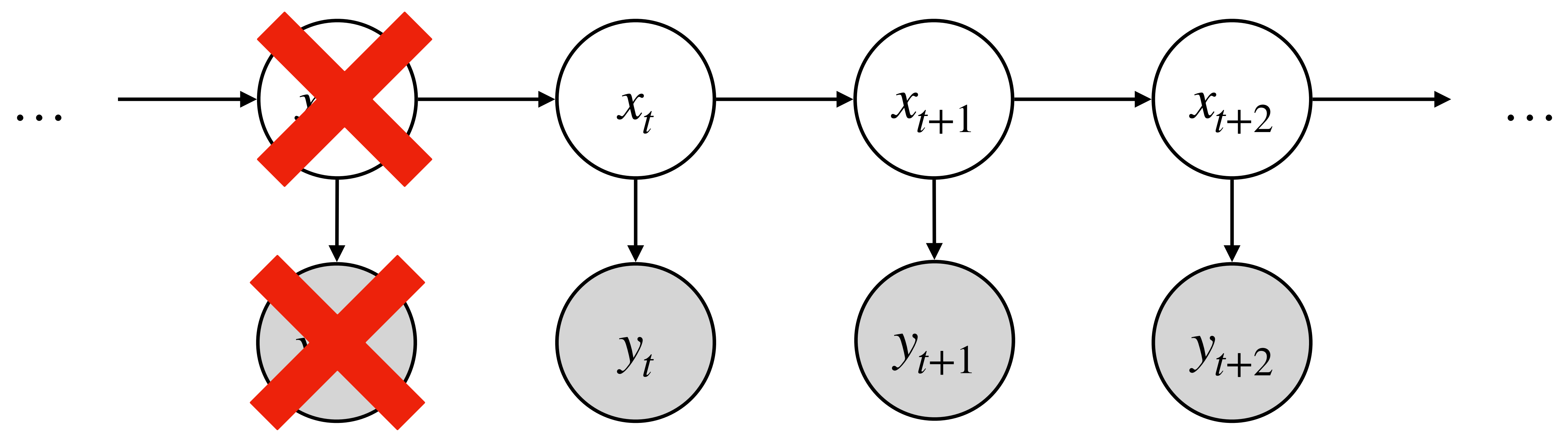
Bounded memory delayed sampling?

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



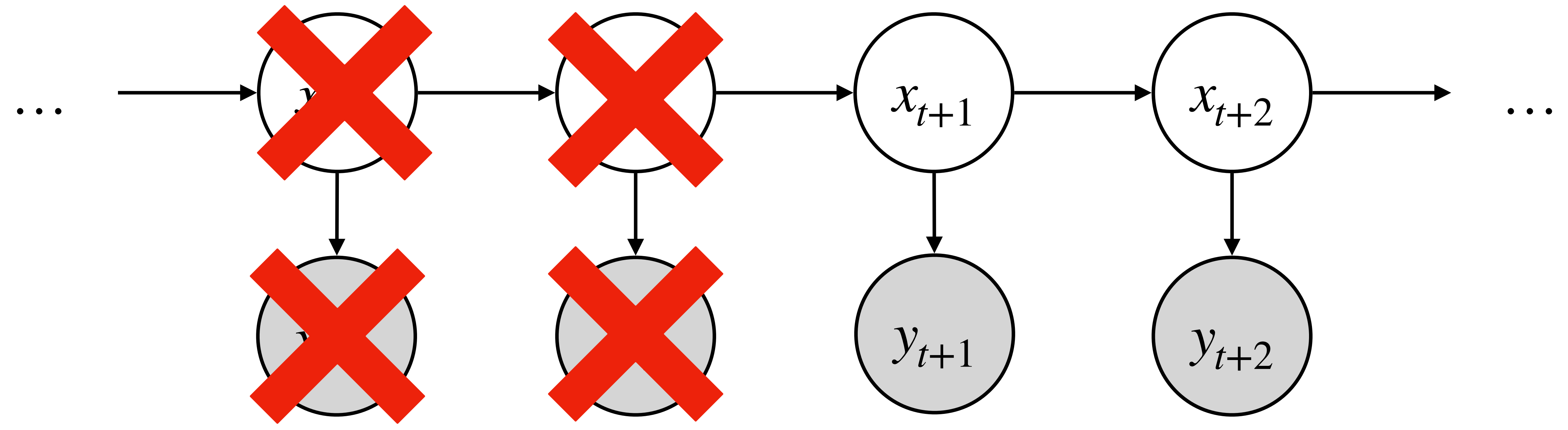
Bounded memory delayed sampling?

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



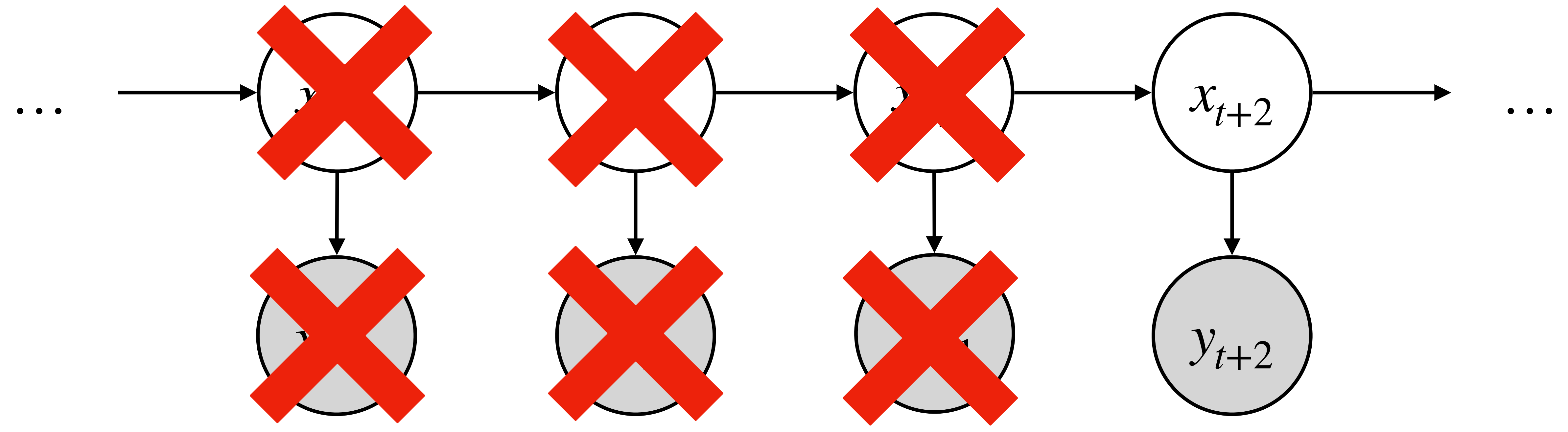
Bounded memory delayed sampling?

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



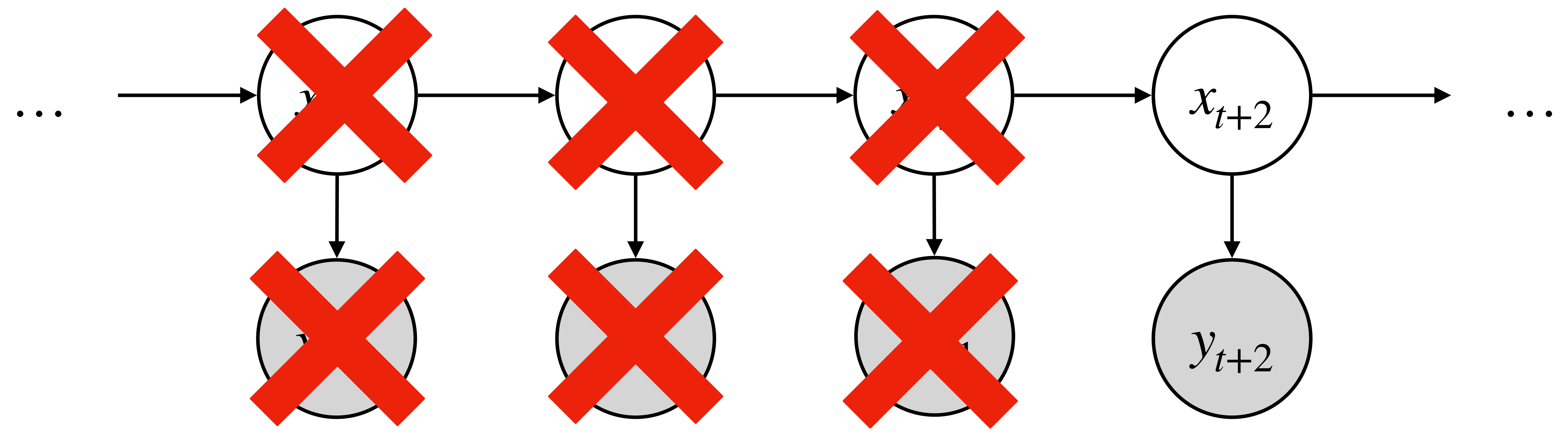
Bounded memory delayed sampling?

```
let 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?

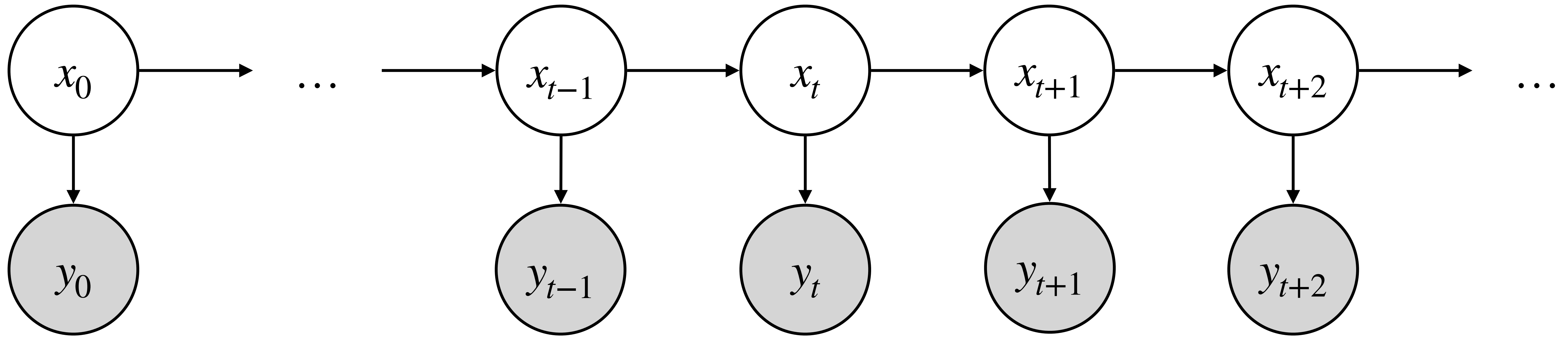
```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



Yes!

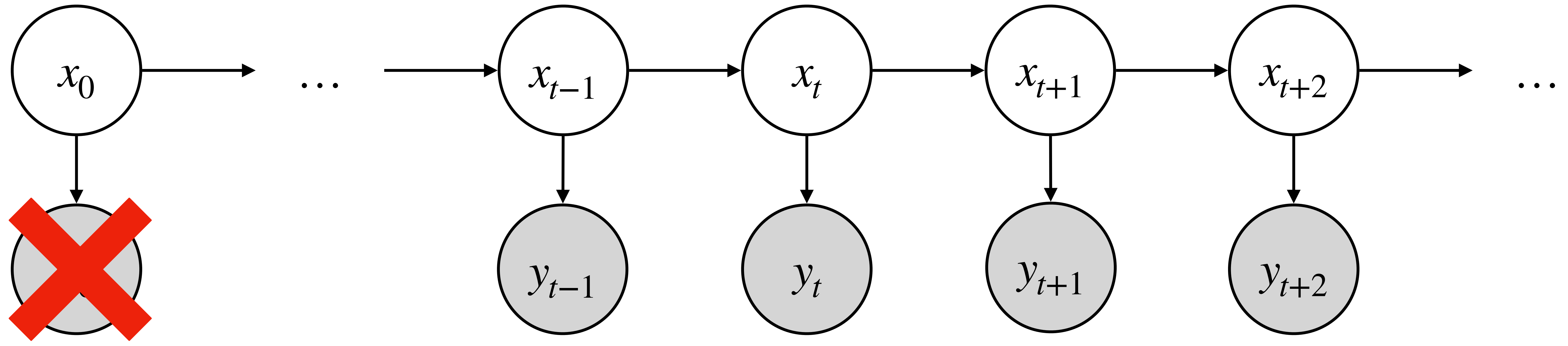
Bounded memory delayed sampling?

```
let 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)
```



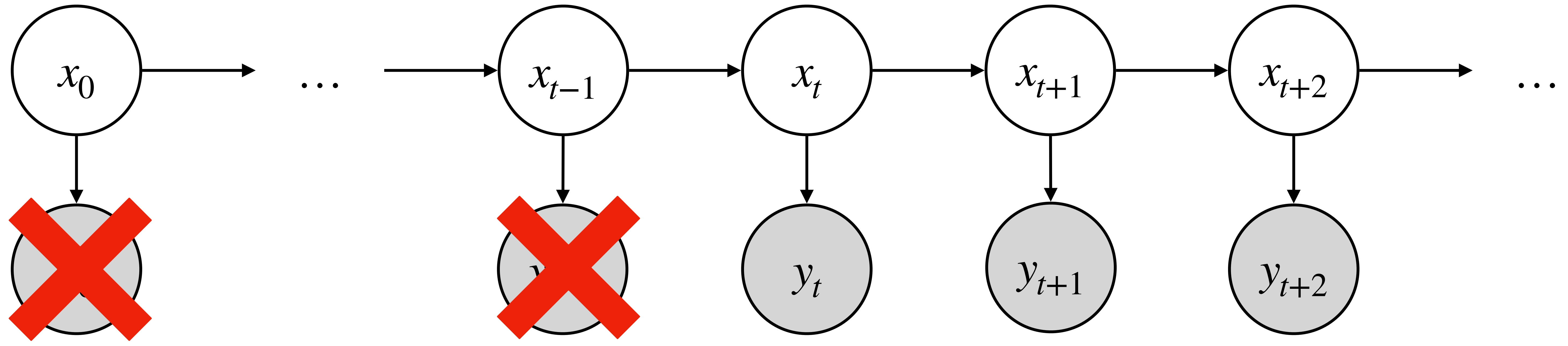
Bounded memory delayed sampling?

```
let 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)
```



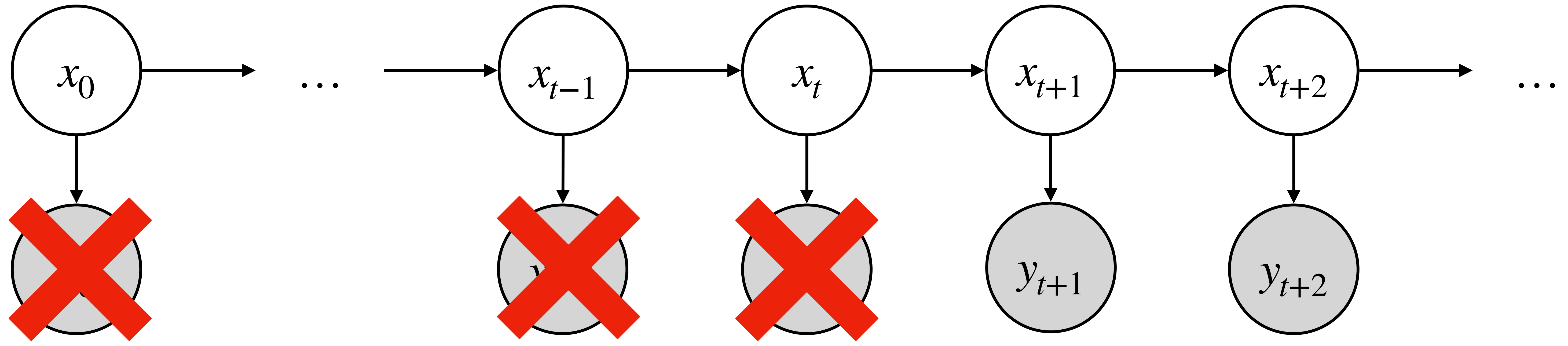
Bounded memory delayed sampling?

```
let 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)
```



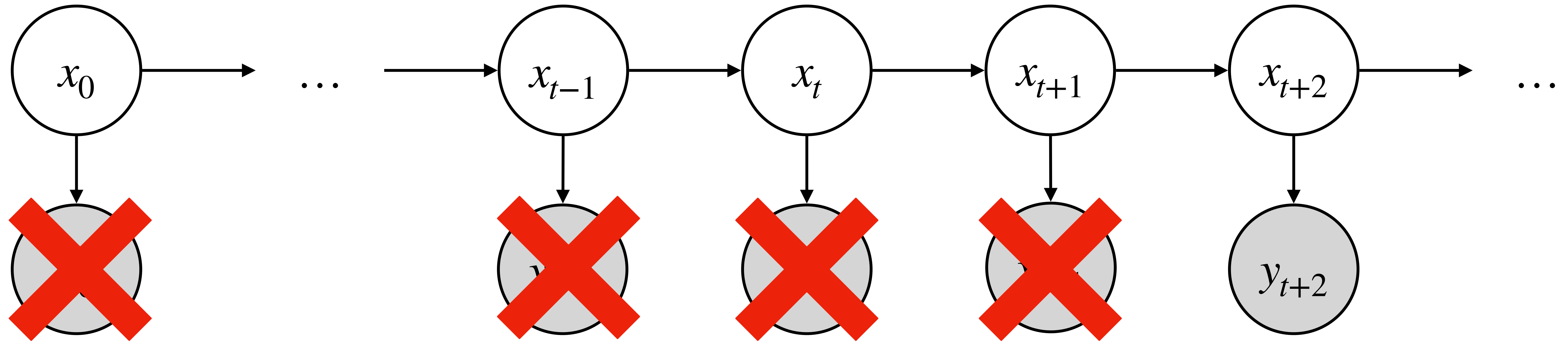
Bounded memory delayed sampling?

```
let 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)
```



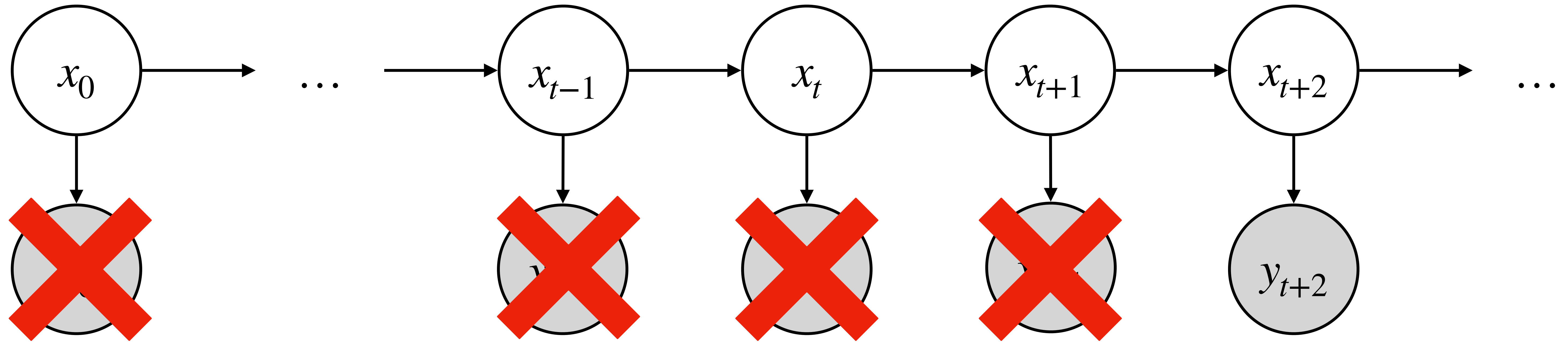
Bounded memory delayed sampling?

```
let 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?

```
let 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)
```

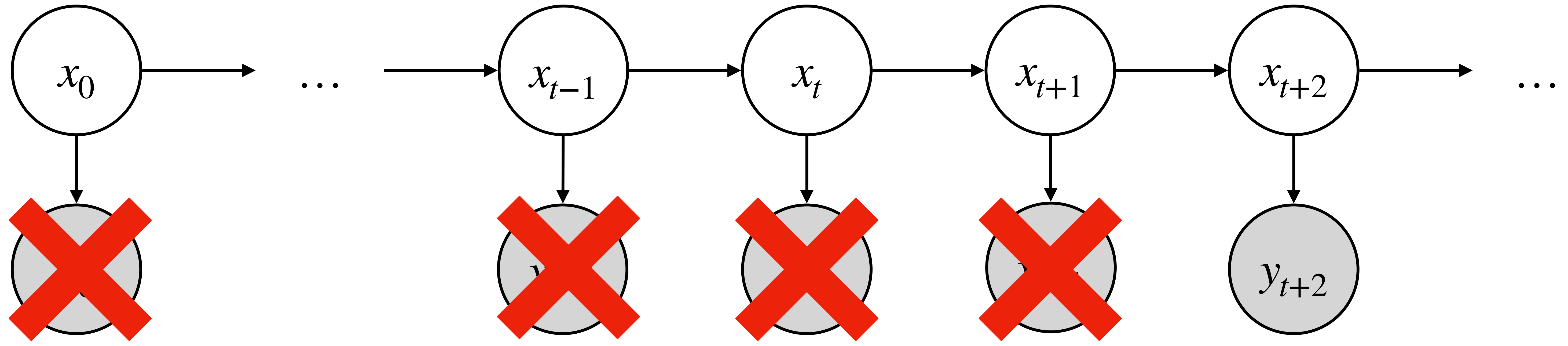


No!

Bounded memory delayed sampling?

```
let 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

```
let 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

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

random variable →

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	$t = 2$
$y_2 \leftarrow x_2$ ::		
$\ldots$		

Trace: abstract execution

```
let 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: Any *program that passes the analysis 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
$x_0 \leftarrow \bot$ ::	`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
$x_0 \leftarrow \perp$ ::	`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	$t = 2$
$y_2 \leftarrow x_2$ ::		
…		

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 $\longrightarrow$

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 $\longrightarrow$	$y_0 \leftarrow x_0$ ::		
y_0 is 0-consumed $\longrightarrow$	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 $\longrightarrow$	$y_0 \leftarrow x_0$ ::		
y_0 is 0-consumed $\longrightarrow$	observe y_0 ::		
	$x_1 \leftarrow x_0$ ::	`x = `x_1`, pre x = `x_0	$t = 1$
x_1 is 1-consumed $\longrightarrow$	$y_1 \leftarrow x_1$ ::		
y_1 is 0-consumed $\longrightarrow$	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	$t = 0$
x_0 is 1-consumed $\longrightarrow$	$y_0 \leftarrow x_0$ ::		
y_0 is 0-consumed $\longrightarrow$	observe y_0 ::		
	$x_1 \leftarrow x_0$ ::	`x = `x_1`, pre x = `x_0	$t = 1$
x_1 is 1-consumed $\longrightarrow$	$y_1 \leftarrow x_1$ ::		
y_1 is 0-consumed $\longrightarrow$	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
$x_0 \leftarrow \perp$ ::	`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	$t = 2$
$y_2 \leftarrow x_2$ ::		
$\dots$		

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	$t = 2$
$y_2 \leftarrow x_2$ ::		
…		

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 (y) = 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 (y) = 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

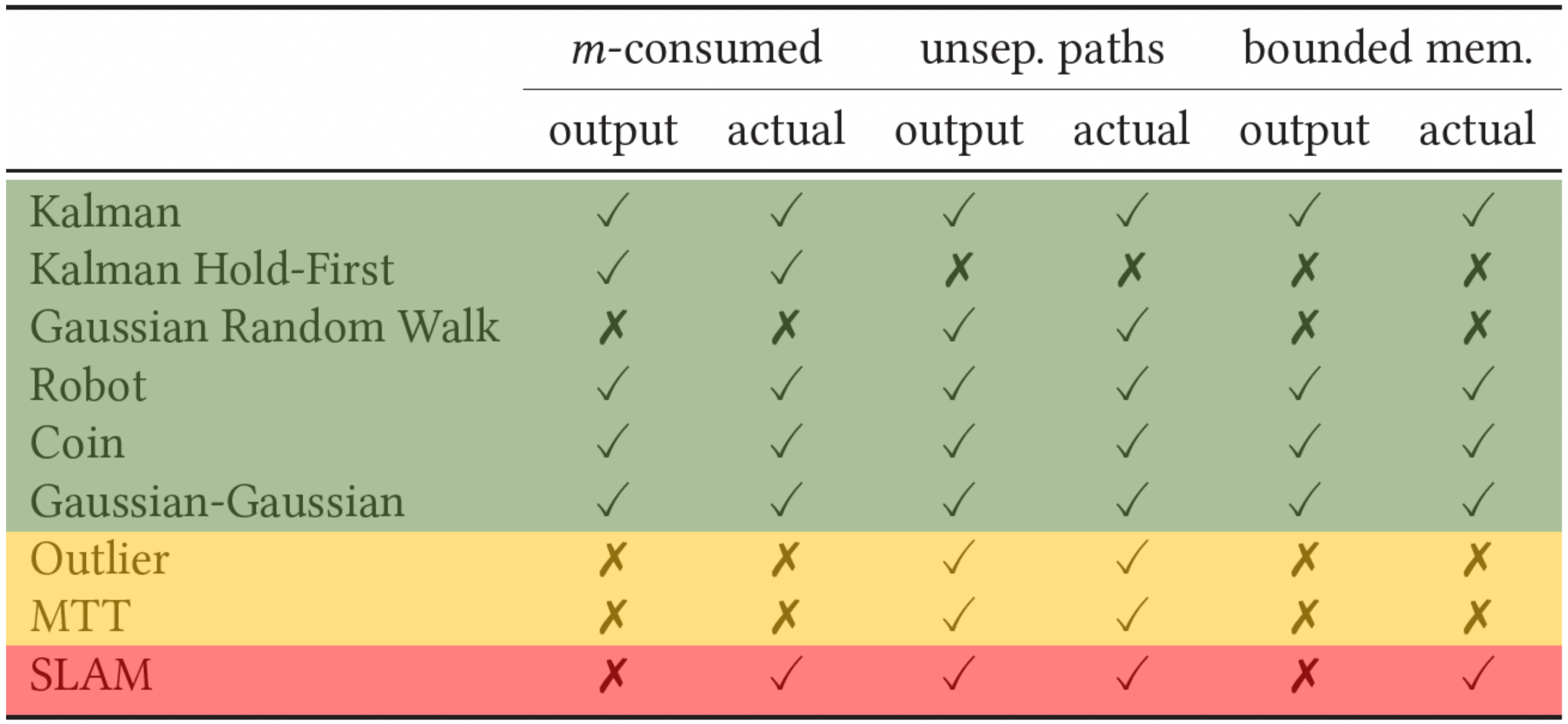
```
proba tracker (y) = x where
  rec init x0 = sample (gaussian (0, 10))
  and x   = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

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	
…		

No!

Evaluation



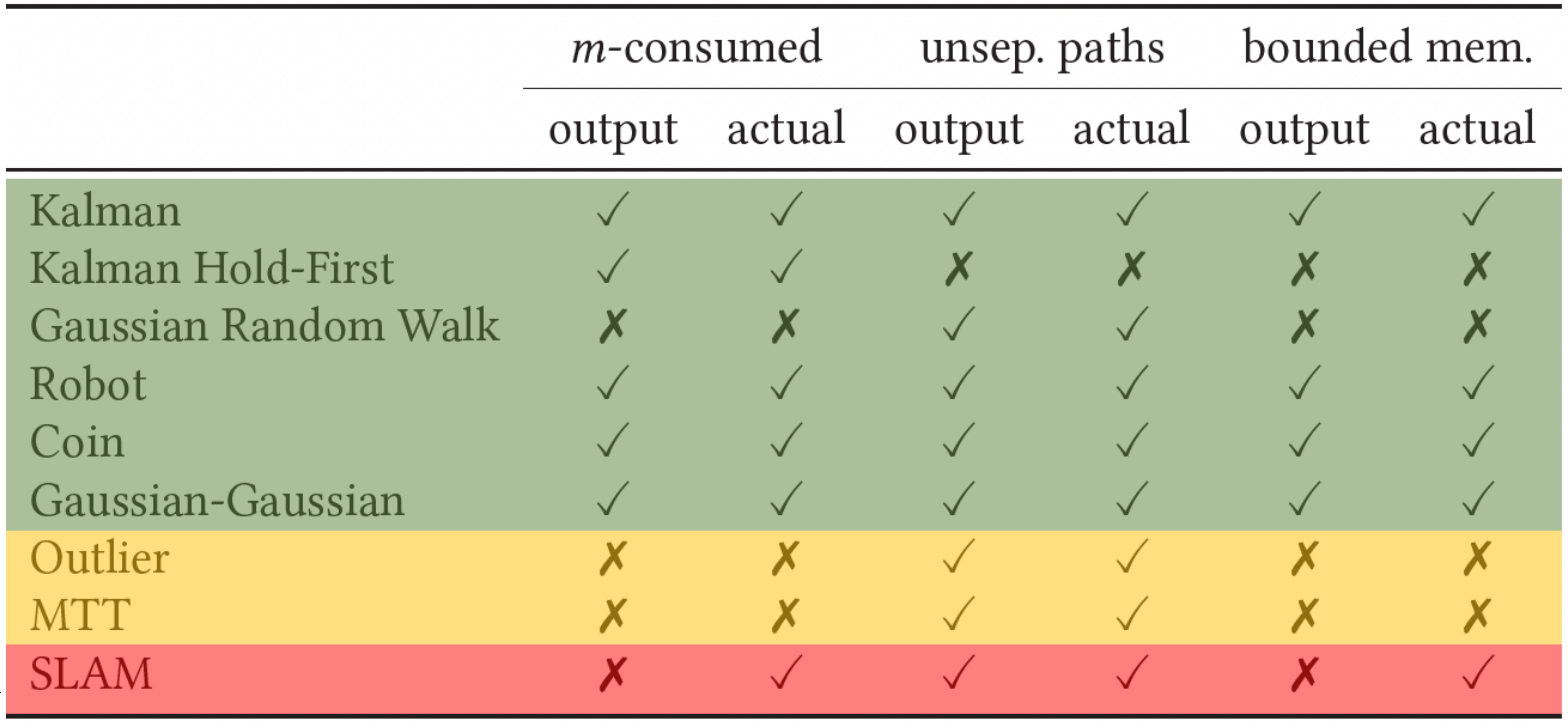
	m-consumed		unsep. paths		bounded mem.	
	output	actual	output	actual	output	actual
Kalman	✓	✓	✓	✓	✓	✓
Kalman Hold-First	✓	✓	✗	✗	✗	✗
Gaussian Random Walk	✗	✗	✓	✓	✗	✗
Robot	✓	✓	✓	✓	✓	✓
Coin	✓	✓	✓	✓	✓	✓
Gaussian-Gaussian	✓	✓	✓	✓	✓	✓
Outlier	✗	✗	✓	✓	✗	✗
MTT	✗	✗	✓	✓	✗	✗
SLAM	✗	✓	✓	✓	✗	✓

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

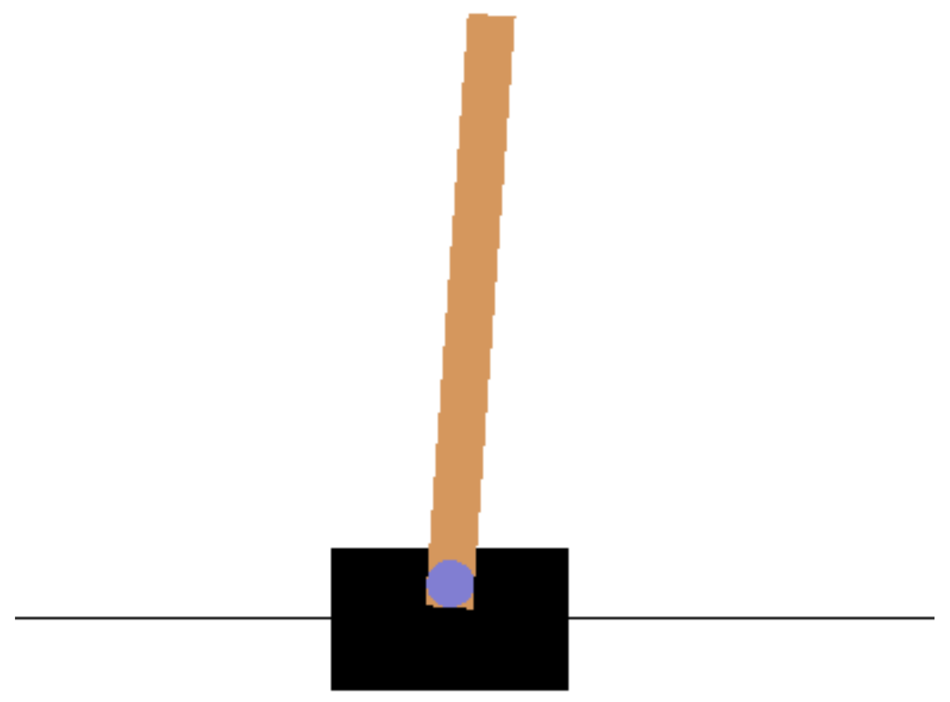
Applications: Control

Reactive Probabilistic Programming

Cartpole PID

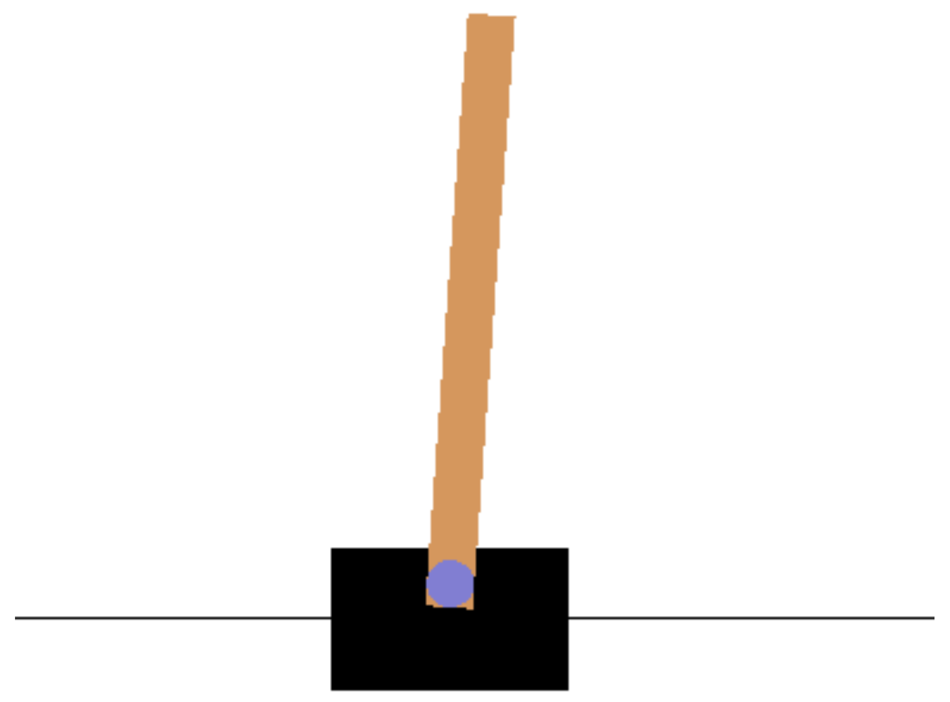
exec_simple_pid

```
let node controller (angle, (p,i,d)) = action where
  rec e = angle -. (0.0 → pre theta)
  and theta = p *. e +. i *. integr(0., e) +. d *. deriv(e)
  and action = if theta > 0. then Right else Left

let p = 0.0403884114239
let i = 0.041460471604
let d = 0.0705417538223

let node main () = () where
  rec obs, _, stop = cart_pole_gym true (Right → pre action)
  and reset action = controller (obs.pole_angle, (p, i, d))
      every stop
```



Cartpole learn from angle

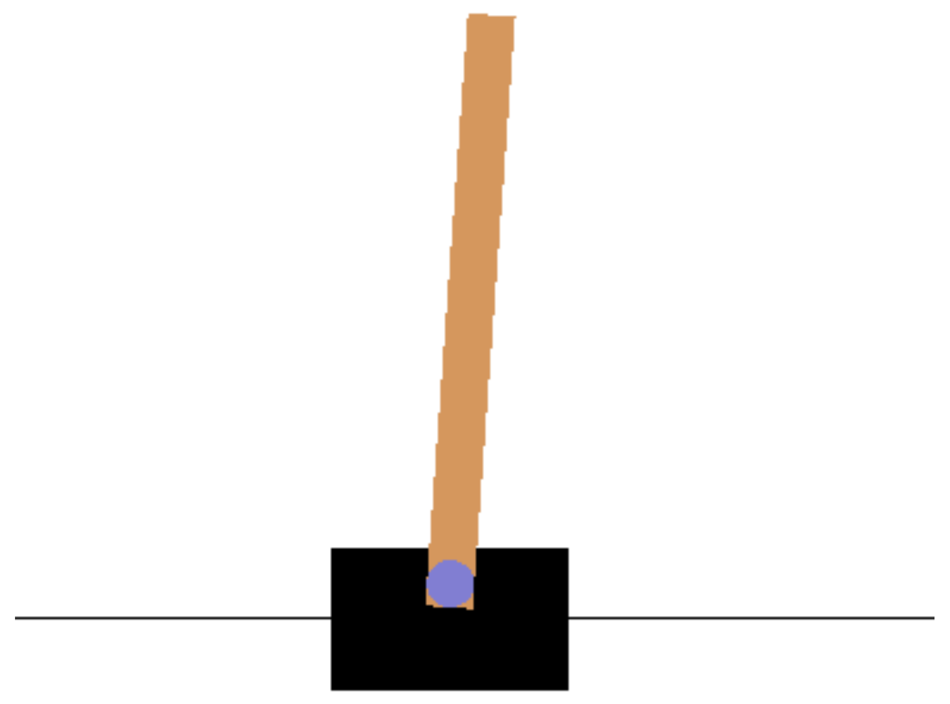
```
(* Learn the coefficients that minimize the angle *)
let proba model obs_init = p, (i, d) where
  rec init p = sample (gaussian 0. 0.1)
  and init i = sample (gaussian 0. 0.1)
  and init d = sample (gaussian 0. 0.1)
  and action = controller (obs.pole_angle, (p,i,d))
  and obs = simple_pendulum (obs_init,  Right → pre action)
  and () = factor (-10. *. abs_float (obs.pole_angle))

let node main () = () where
  rec obs, _, stop = cart_pole_gym true (Right → pre action)
  and reset action = controller (obs.pole_angle, (p, i, d))
      every stop
  and pid_dist = infer 1000 model obs
  and p, (i, d) = draw pid_dist
```



Cartpole learn from example

exec_smart_pid

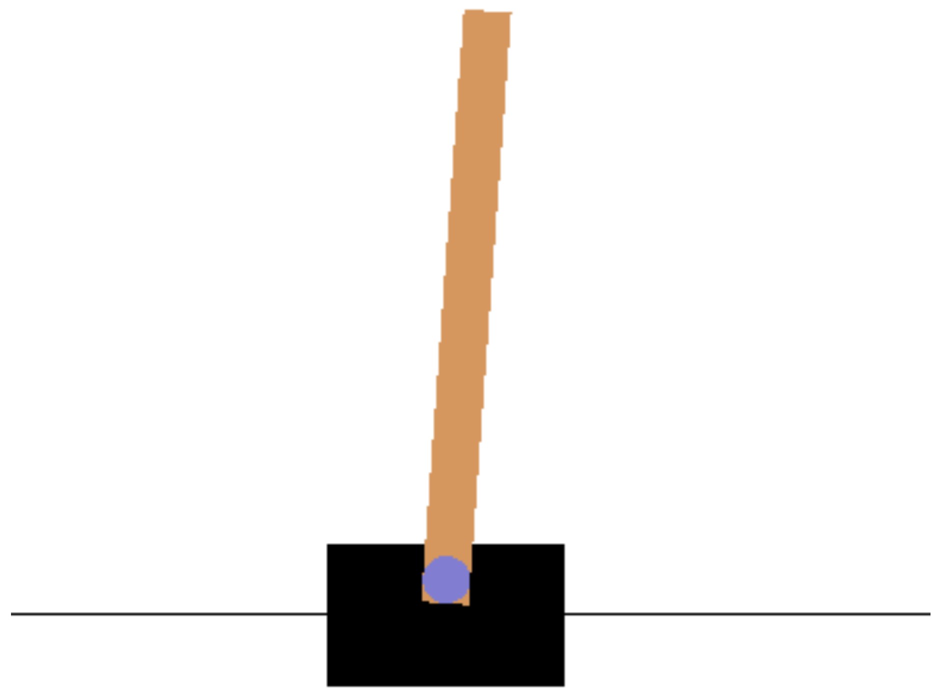
```
(* Favor action similar to example *)
let proba model (obs, ctrl_action) = p, (i, d) where
  rec init p = sample (gaussian 0. 0.1)
  and init i = sample (gaussian 0. 0.1)
  and init d = sample (gaussian 0. 0.1)
  and action = controller (obs.pole_angle, (p,i,d))
  and () = factor (if action = ctrl_action then 0. else -0.2)

let node main () = () where
  rec obs, _, stop = cart_pole_gym true (Right → pre action)
  and reset action = controller (obs.pole_angle, (p, i, d))
      every stop
  and pid_dist = infer 1000 model obs
  and p, (i, d) = draw pid_dist
```



Cartpole learn from example

exec_smart_pid

```
(* Favor action similar to example *)
let proba model (obs, ctrl_action) = p, (i, d) where
  rec init p = sample (gaussian 0. 0.1)
  and init i = sample (gaussian 0. 0.1)
  and init d = sample (gaussian 0. 0.1)
  and action = controller (obs.pole_angle, (p,i,d))
  and () = factor (if action = ctrl_action then 0. else -0.2)

let node main fix = () where
  rec obs, _, stop = cart_pole_gym true (Right → pre action)
  and reset action = controller (obs.pole_angle, (p, i, d))
      every stop
  and automaton
      | Learn → do pid_dist = infer 1000 model obs
                and p, (i, d) = draw pid_dist
        until fix then Fix
      | Fix → do until (not fix) then Learn
  end
```



References

Reactive probabilistic programming
Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, Michael Carbin
PLDI 2020

Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs.
Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, Thomas B. Schön
AISTATS 2017

A Co-iterative Characterization of Synchronous Stream Functions
Paul Caspi and Marc Pouzet
CMCS 1998

Statically Bounded-Memory Delayed Sampling for Probabilistic Streams
Eric Atkinson, Guillaume Baudart, Louis Mandel, Charles Yuan, Michael Carbin
OOPLSA 2021