Probabilistic Programming in Birch

www.birch-lang.org

Lawrence Murray

Department of Information Technology, Uppsala University

UPPSALA UNIVERSITET

SWEDISH FOUNDATION *for* STRATEGIC RESEARCH

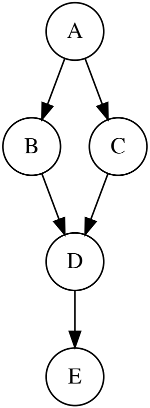
Outline

1. Graphical models $\longrightarrow$ probabilistic programs.
2. Birch: motivation and design.
3. Birch: language features.

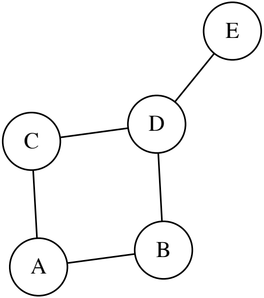
1 Graphical models $\longrightarrow$ probabilistic programs

Graphical models

(a) Directed

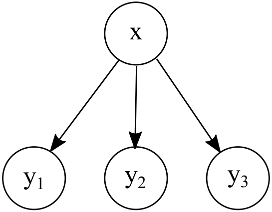


(b) Undirected

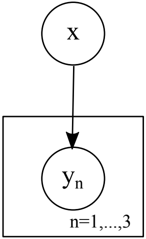


Graphical models

(a) Without plate notation



(b) With plate notation



Graphical models

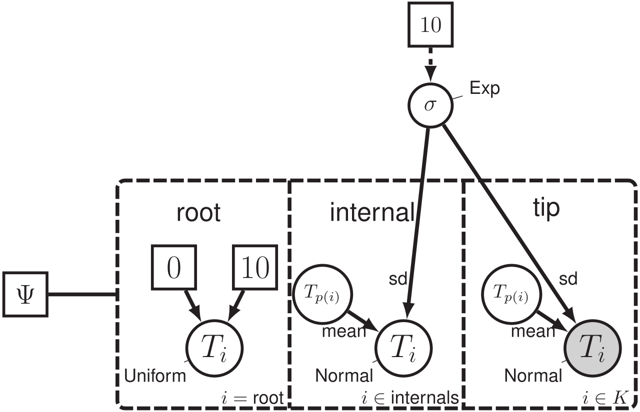


Figure: S. Höhna, M. J. Landis, T. A. Heath, B. Boussau, N. Lartillot, B. R. Moore, J. P.

Huelsenbeck, and F. Ronquist. Revbayes: Bayesian phylogenetic inference using graphical models and an interactive model-specification language. Systematic, 65(4):726–736, 2016.

doi: 10.1093/sysbio/syw021

Graphical models

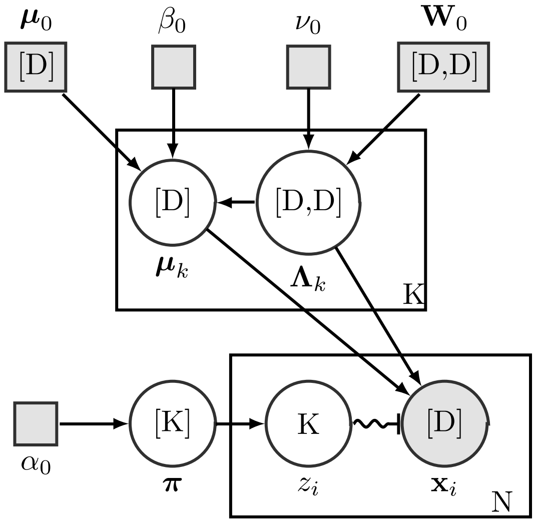
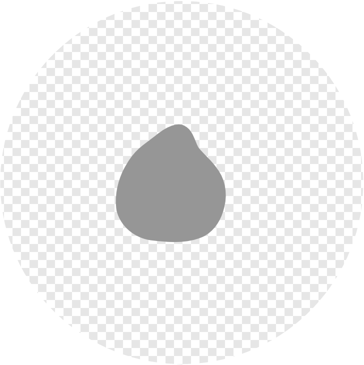


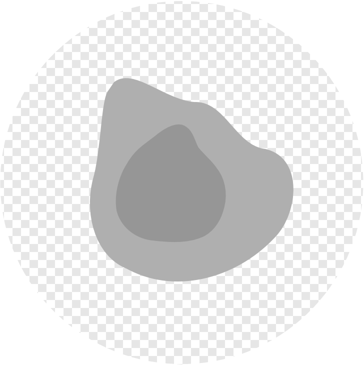
Figure: Benwing `https://commons.wikimedia.org/wiki/File:Bayesian-gaussian-mixture.svg`

Graphical models $\longrightarrow$ probabilistic programs

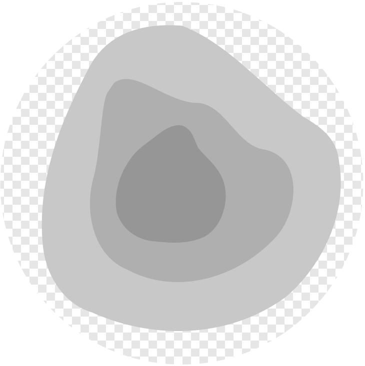
Graphical models $\longrightarrow$ probabilistic programs



Graphical models $\longrightarrow$ probabilistic programs



Graphical models $\longrightarrow$ probabilistic programs



Graphical models $\longrightarrow$ probabilistic programs

The most expressive languages are known as **universal**

Also known as **Turing complete**.

Models written in such languages are **universal probabilistic programs**.

These are the most expressive languages for model specification, but also the most difficult for which to do inference.

Graphical models $\longrightarrow$ probabilistic programs

An alternative perspective on probabilistic programming is that it is a **programming paradigm** for probabilistic modelling and inference.

Graphical models $\longrightarrow$ probabilistic programs

An alternative perspective on probabilistic programming is that it is a **programming paradigm** for probabilistic modelling and inference.

- Other programming paradigms include object-oriented programming, generic programming, procedural programming, functional programming, etc.
- From this perspective, probabilistic programming languages merely emphasise this particular programming paradigm, providing ergonomic features for writing probabilistic models and probabilistic inference methods.

2 Birch: motivation and design

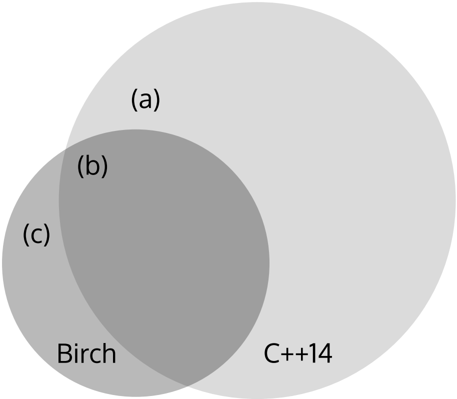
Birch

- Universal probabilistic programming language (PPL).
- Supports procedural, generic, object-oriented, and (of course) probabilistic programming paradigms.
- Both models and methods are written in the Birch language itself.
- Draws inspiration from many places, including existing PPLs such as LibBi (`www.libbi.org`), and modern object-oriented languages such as Swift.
- Free and open source, under the Apache 2.0 license.
- See `birch-lang.org`

Technical details

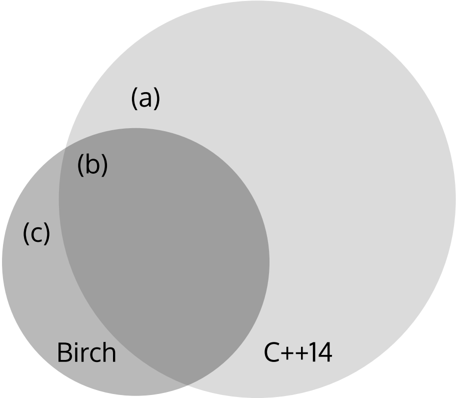
- Dynamic memory management with reference-counted garbage collection.
- Compiles to C++14 then native binaries.
- Uses standard C/C++ libraries for numerical computing, e.g. STL, Boost, Eigen.
- C/C++ code can be nested in Birch code to allow tight integration.

Birch $\longrightarrow$ C++14



Birch $\longrightarrow$ C++14

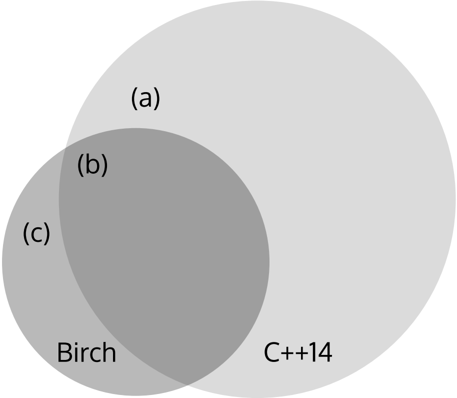
(a) C++14 provides a lot of things we would like to quarantine.



Birch $\longrightarrow$ C++14

(a) C++14 provides a lot of things we would like to quarantine.

(b) Most Birch code translates directly to C++14
e.g. object model, higher-order functions, user-defined conversions

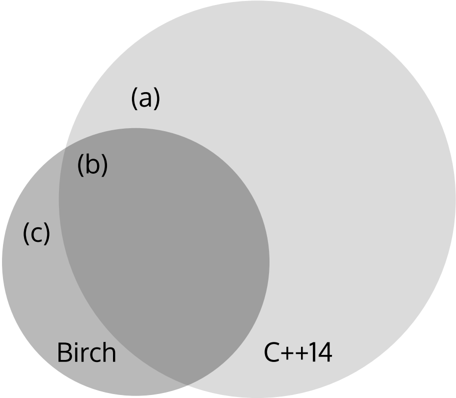


Birch $\longrightarrow$ C++14

(a) C++14 provides a lot of things we would like to quarantine.

(b) Most Birch code translates directly to C++14
e.g. object model, higher-order functions, user-defined conversions

(c) Some Birch code translates to verbose or intrusive C++14 that one would not want to code by hand
e.g. probabilistic operators, fibers, copy-on-write



Models in Birch

In Birch, a model is specified by writing a program that simulates from the **joint distribution**.

- In many other PPLs, there is a distinction between which variables are observed and which are latent **within the program**.
  - i.e. the program already factors the joint distribution into likelihood and prior.
- In Birch, the preference is to distinguish which variables are observed and which are latent **at runtime**.
  - i.e. at runtime, the user, or the inference method, chooses which conditionals or marginals of the joint distribution are of interest.

Models in Birch

In Birch, a model is specified by writing a program that simulates from the **joint distribution**.

- In many other PPLs, there is a distinction between which variables are observed and which are latent **within the program**.
  - i.e. the program already factors the joint distribution into likelihood and prior.
- In Birch, the preference is to distinguish which variables are observed and which are latent **at runtime**.
  - i.e. at runtime, the user, or the inference method, chooses which conditionals or marginals of the joint distribution are of interest.
  - (Ideally, at least, as this is not always possible.)

Example: Bayesian linear regression model

```
class LinearRegressionModel < Model {
  X:Real[_,_];
  σ2:Random<Real>;
  β:Random<Real[_]>;
  y:Random<Real[_]>;

  fiber simulate() -> Real {
    N:Integer <- rows(X);
    P:Integer <- columns(X);
    if (N > 0 && P > 0) {
      σ2 ~ InverseGamma(3.0, 0.4);
      β ~ Gaussian(vector(0.0, P), identity(P)*σ2);
      y ~ Gaussian(X*β, σ2);
    }
  }
}
```

Example: linear-Gaussian state-space model

```
class LinearGaussianSSM = MarkovModel<LinearGaussianSSMState,
    LinearGaussianSSMParameter>;

class LinearGaussianSSMParameter < Parameter {
  a:Real <- 0.8;
  σ2_x:Real <- 1.0;
  σ2_y:Real <- 0.1;
}

class LinearGaussianSSMState < State {
  x:Random<Real>;
  y:Random<Real>;

  fiber initial(θ:LinearGaussianSSMParameter) -> Real {
    x ~ Gaussian(0.0, θ.σ2_x);
    y ~ Gaussian(x, θ.σ2_y);
  }
```

Example: linear-Gaussian state-space model

```
  fiber transition(z:LinearGaussianSSMState,
      θ:LinearGaussianSSMParameter) -> Real {
    x ~ Gaussian(θ.a*z.x, θ.σ2_x);
    y ~ Gaussian(x, θ.σ2_y);
  }
}
```

Example: nonlinear state-space model

```
class SIRModel = MarkovModel<SIRState,SIRParameter>;

class SIRParameter < Parameter {
  λ:Random<Real>;
  δ:Random<Real>;
  γ:Random<Real>;

  fiber parameter() -> Real {
    λ <- 10.0;
    δ ~ Beta(2.0, 2.0);
    γ ~ Beta(2.0, 2.0);
  }
}

class SIRState < State {
  τ:Random<Integer>;
  Δi:Random<Integer>;
  Δr:Random<Integer>;
```

Example: nonlinear state-space model

```
  s:Random<Integer>;
  i:Random<Integer>;
  r:Random<Integer>;

  fiber transition(x:SIRState, θ:SIRParameter) -> Real {
    τ ~ Binomial(x.s, 1.0 - exp(-θ.λ*x.i/(x.s + x.i + x.r)));
    Δi ~ Binomial(τ, θ.δ);
    Δr ~ Binomial(x.i, θ.γ);

    s ~ Delta(x.s - Δi);
    i ~ Delta(x.i + Δi - Δr);
    r ~ Delta(x.r + Δr);
  }
}
```

Models in Birch

- Knowing something about the structure of a model may help tailor the inference algorithm, so it will be useful if programs reveal something of this.
- One option is static analysis, but this is hard.
- The approach at this stage is for it to be the programmer's responsibility to reveal this by construction, e.g. using the `MarkovModel` class.
- Details are still developing.

Methods in Birch

Inference methods are also written in the Birch language.

- Currently available are:
  - Analytical solutions
  - Importance sampling
  - Bootstrap particle filter
  - Alive particle filter
  - Auxiliary particle filter (automated)
  - Rao–Blackwellized particle filter (automated)
- Not far off are:
  - Particle MCMC methods
  - Other MCMC methods.

3 Birch: language features

Optionals

Optionals allow variables to have a value of a particular type, or no value at all.

- They are used in other programming languages (e.g. Swift) to eliminate boilerplate that checks for `null` values, e.g. a function checking its arguments.
- In Birch, they are used for the same purpose, but also a second role: to represent **missing values**.

Randoms

Randoms are optionals to which a probability distribution can be attached.

- When they **don't have a value**, the probability distribution can be used to automatically **simulate a value**.

- Once a random has a value, that value is final, it cannot be overwritten.

Delayed sampling

- Randoms are essential for the **delayed sampling** mechanism within Birch.
- This is a heuristic algorithm for performing analytical optimizations at runtime.
- It automatically yields optimizations such as variable elimination/collapsing, Rao–Blackwellization and locally-optimal proposals.

See:

L. M. Murray, D. Lundén, J. Kudlicka, D. Broman, and T. B. Schön. Delayed sampling and automatic Rao–Blackwellization of probabilistic programs. Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS), 2018.

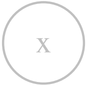
URL `https://arxiv.org/abs/1708.07787`

Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	

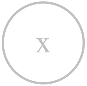
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	**assume** x
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	



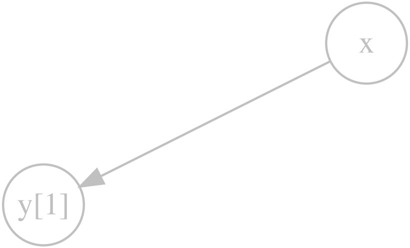
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	



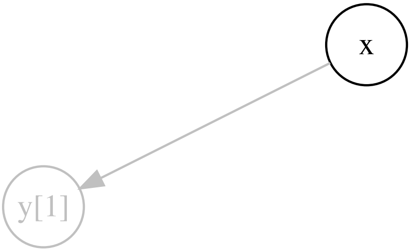
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



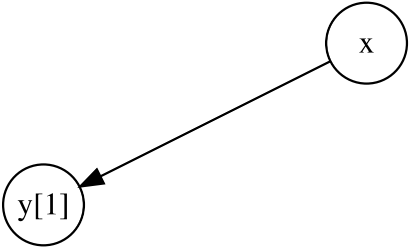
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



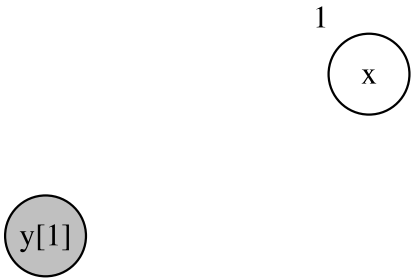
Delayed sampling example

Code	**Checkpoint**
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



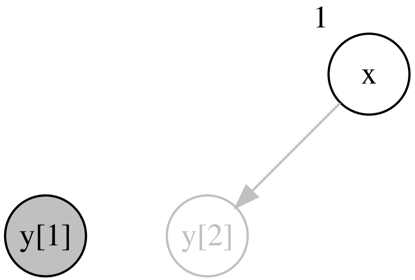
Delayed sampling example

Code	**Checkpoint**
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



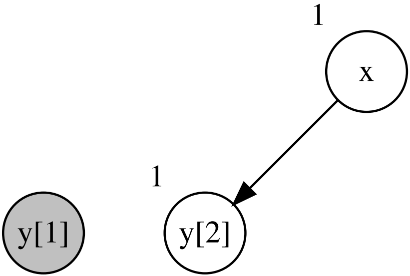
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



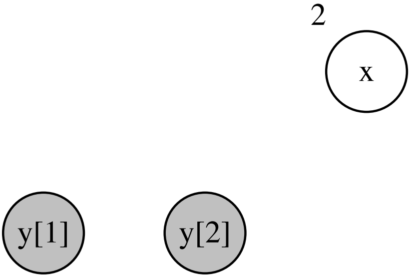
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



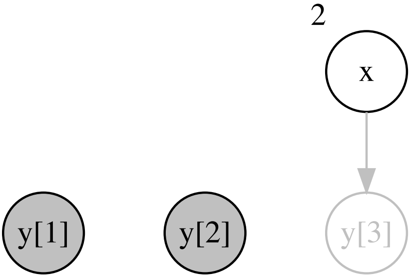
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



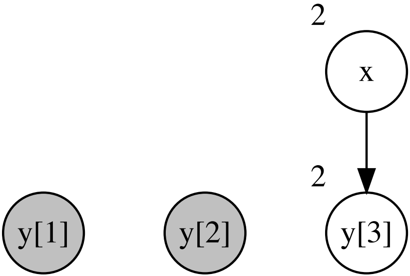
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



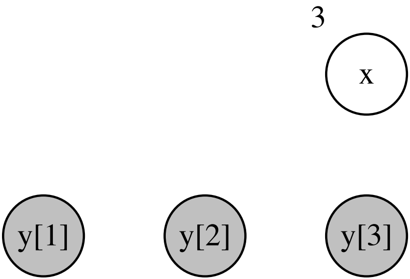
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



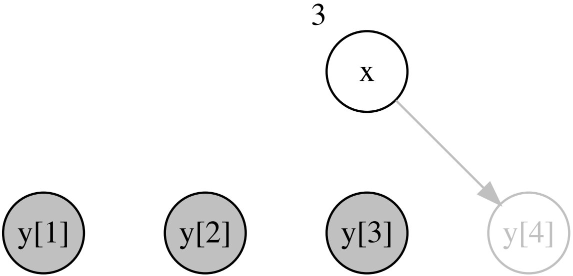
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



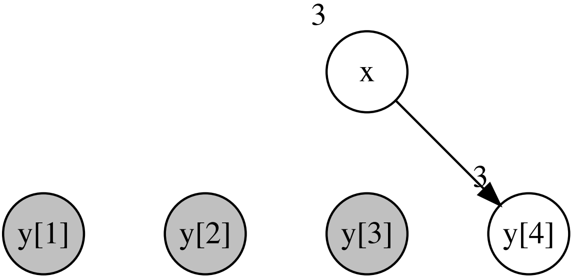
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



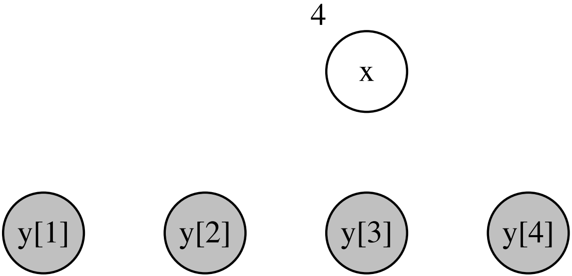
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



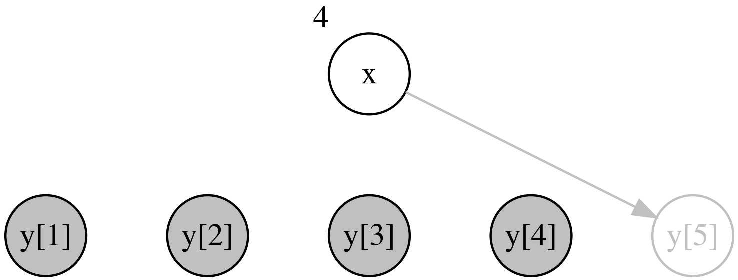
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



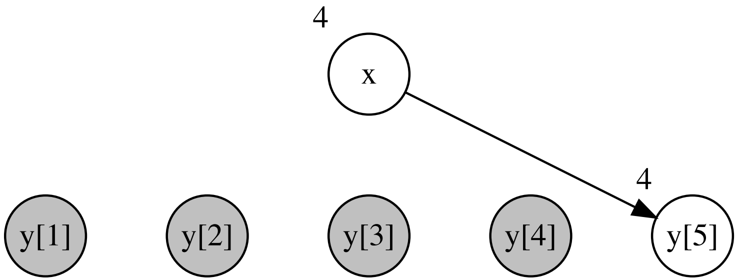
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



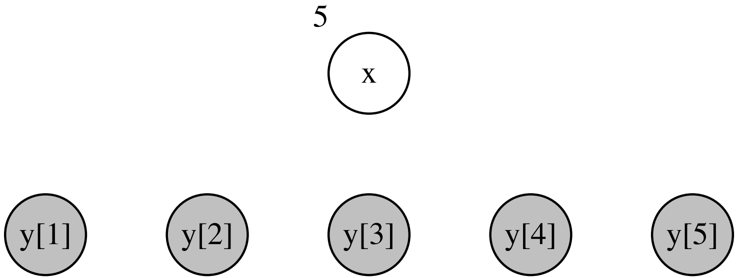
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



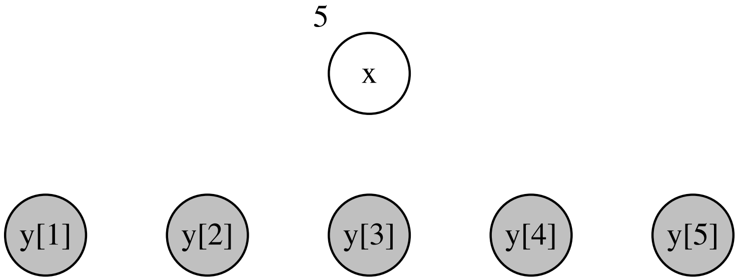
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



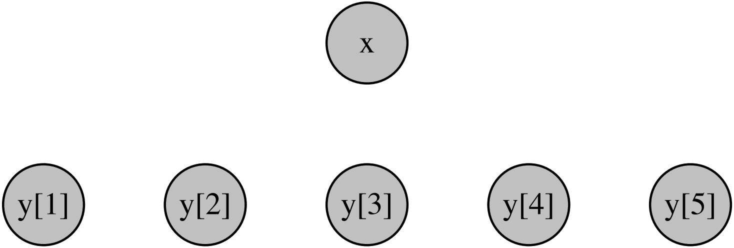
Delayed sampling example

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	**value** x



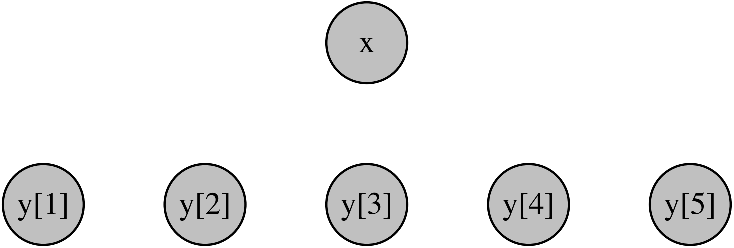
Delayed sampling example

Code	**Checkpoint**
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	



Delayed sampling example

Code	**Checkpoint**
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	



Delayed sampling

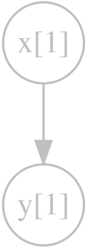
Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	

Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	**assume** x[1]
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	

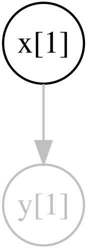
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	**observe** y[1]
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



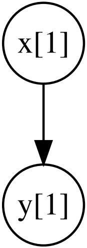
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	**observe** y[1]
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



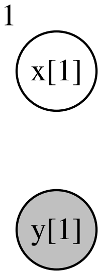
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	**observe** y[1]
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



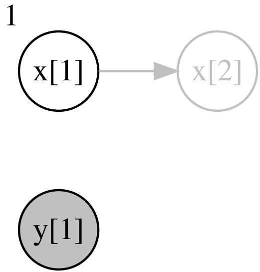
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	**observe** y[1]
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



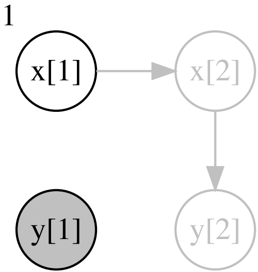
Delayed sampling

Code	**Checkpoint**
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	**assume** x[t]
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



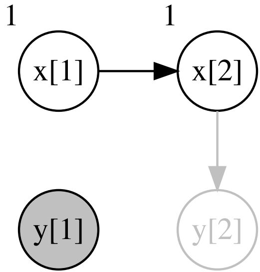
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



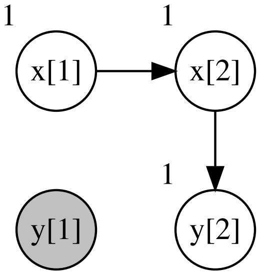
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



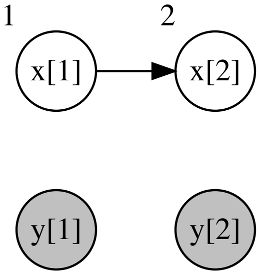
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



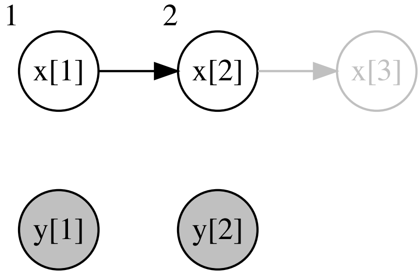
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



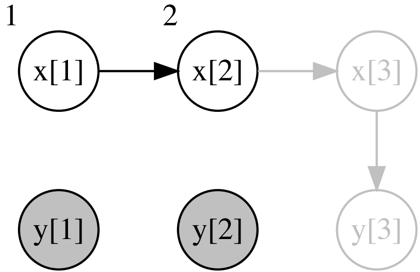
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	**assume** x[t]
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



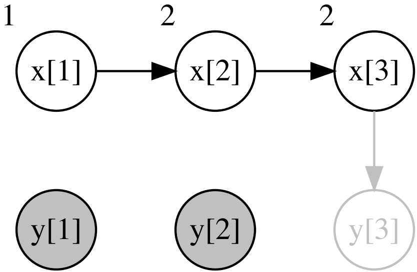
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



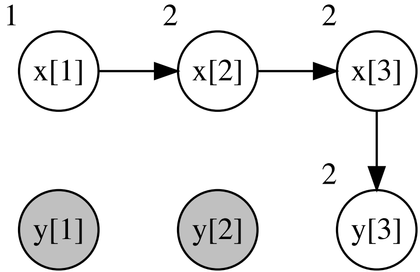
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



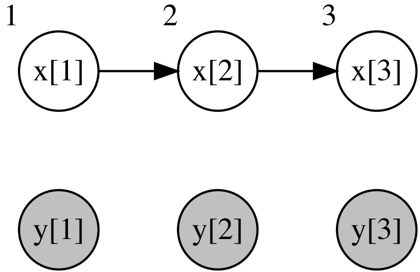
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



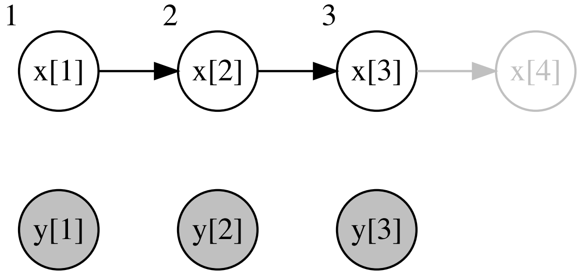
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



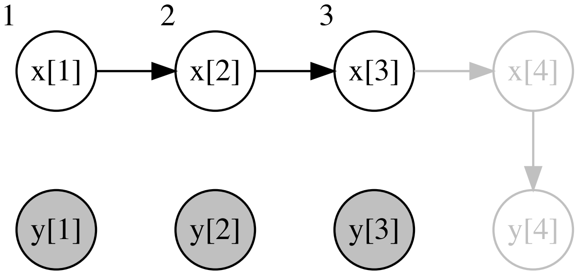
Delayed sampling

Code	**Checkpoint**
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	**assume** x[t]
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



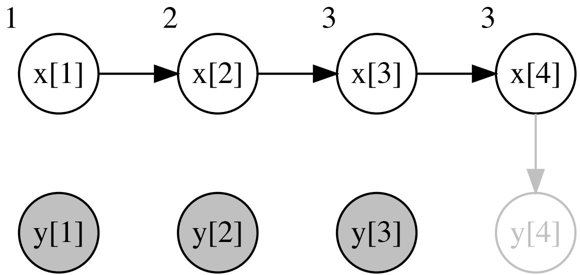
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



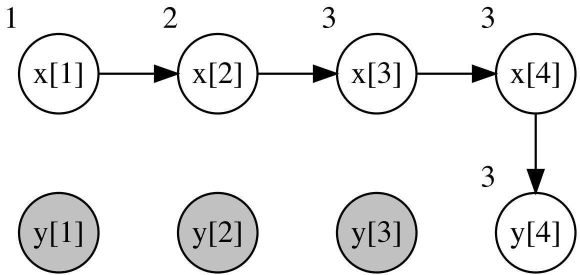
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



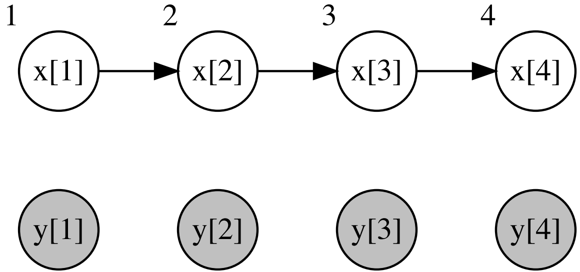
Delayed sampling

Code	**Checkpoint**
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



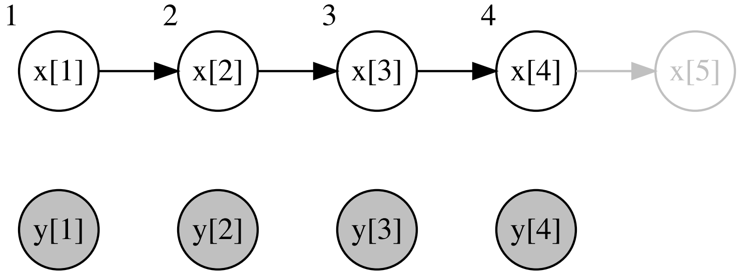
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



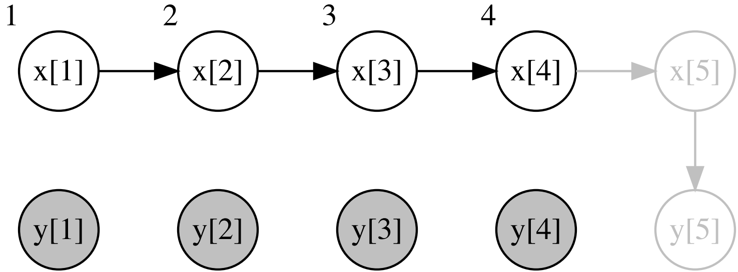
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	**assume** x[t]
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



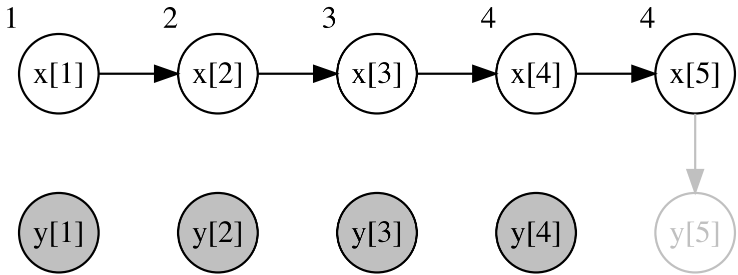
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



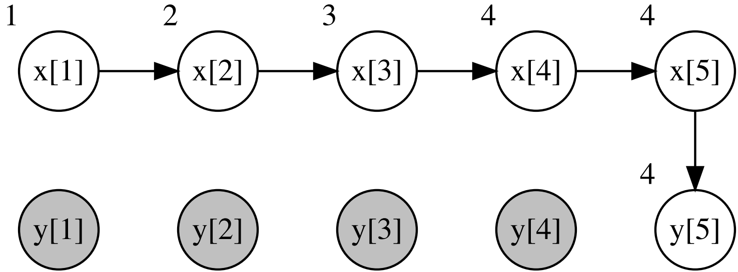
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



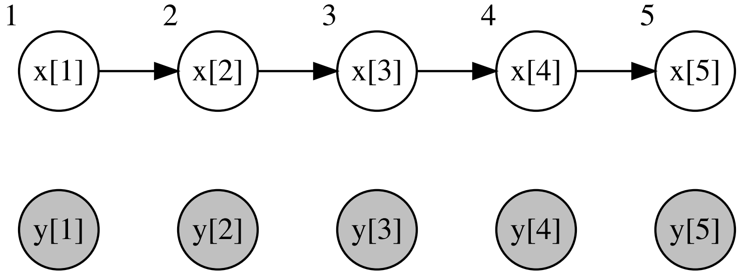
Delayed sampling

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



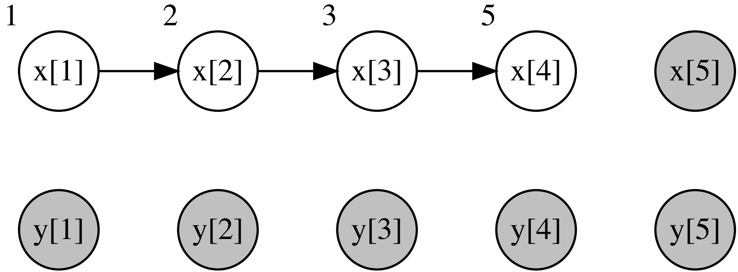
Delayed sampling: Kalman Filter

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



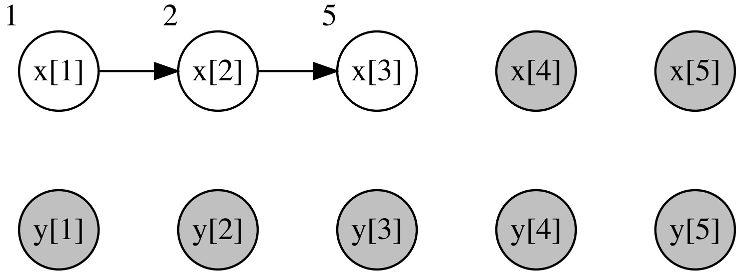
Delayed sampling: Kalman Filter

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	**value** x[1]



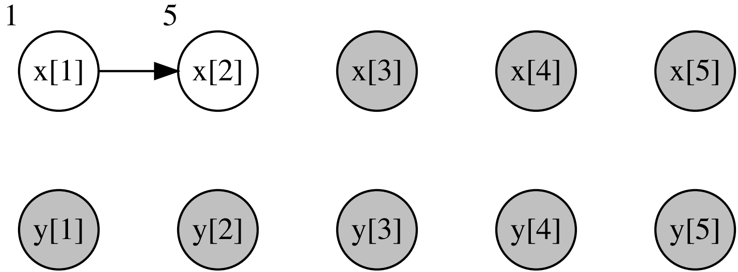
Delayed sampling: Kalman Filter

Code	**Checkpoint**
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	**value** x[1]



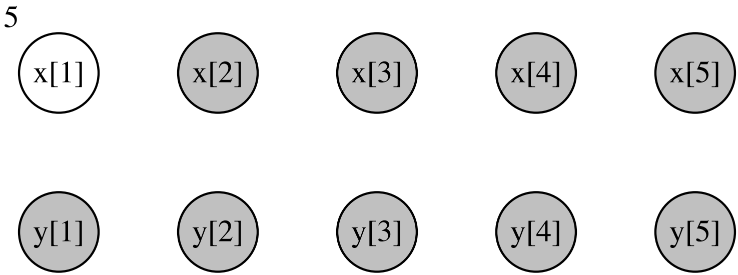
Delayed sampling: Kalman Filter

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	**value** x[1]



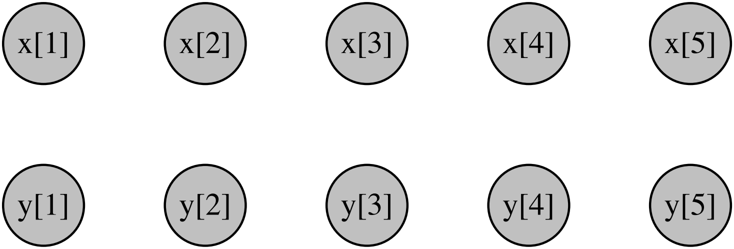
Delayed sampling: Kalman Filter

Code	**Checkpoint**
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	**value** x[1]



Delayed sampling: Kalman Filter

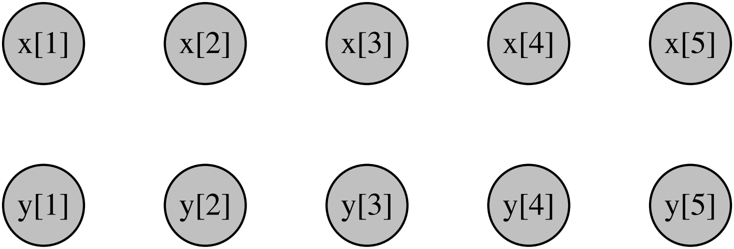
Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	**value** x[1]



Delayed sampling: Kalman Filter

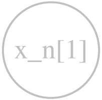
Code **Checkpoint**

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
stdout.print(x[1]);
```

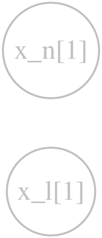


Delayed sampling

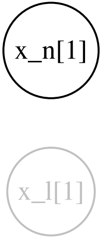
Delayed sampling



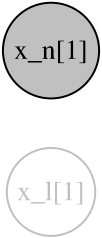
Delayed sampling



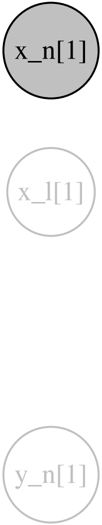
Delayed sampling



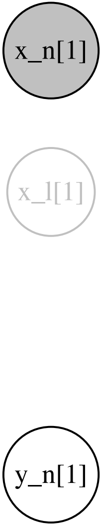
Delayed sampling



Delayed sampling

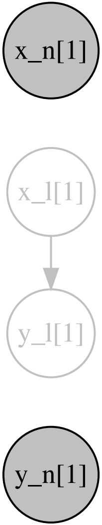


Delayed sampling

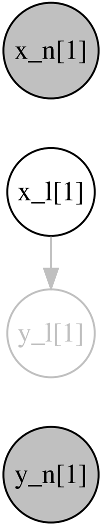


Delayed sampling

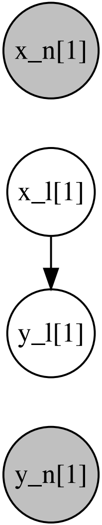
Delayed sampling



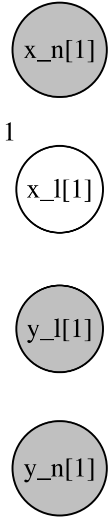
Delayed sampling



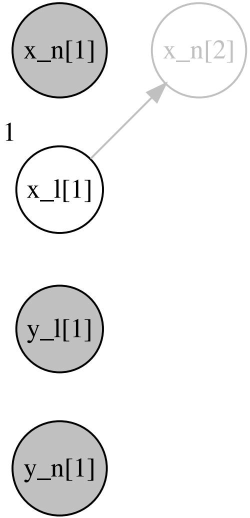
Delayed sampling



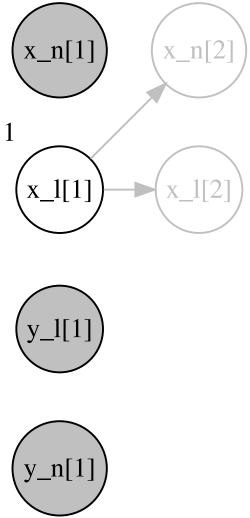
Delayed sampling



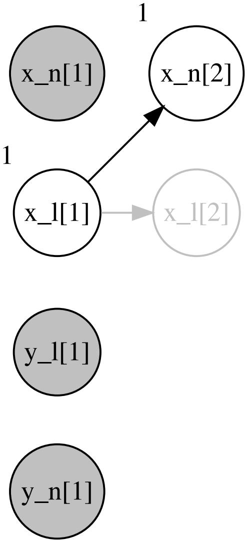
Delayed sampling



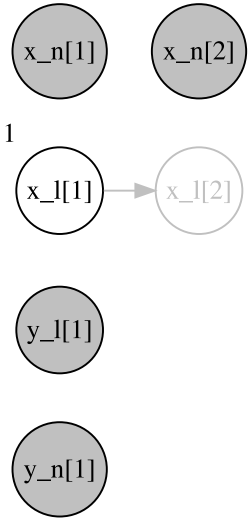
Delayed sampling



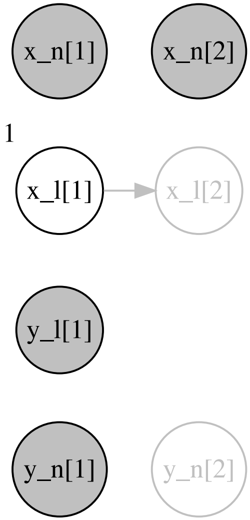
Delayed sampling



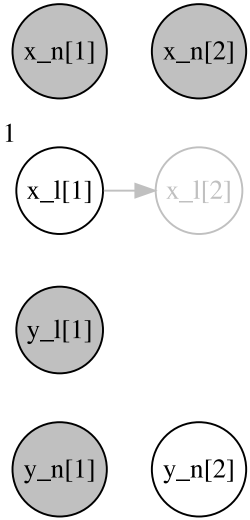
Delayed sampling



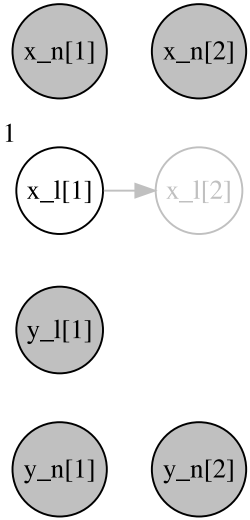
Delayed sampling



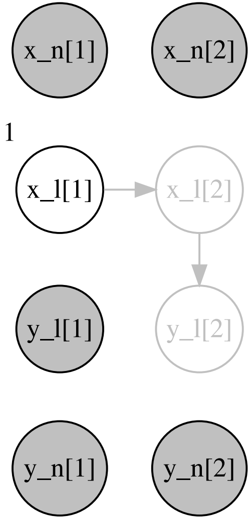
Delayed sampling



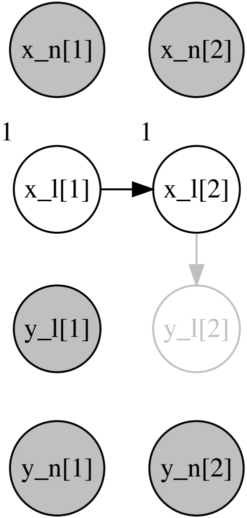
Delayed sampling



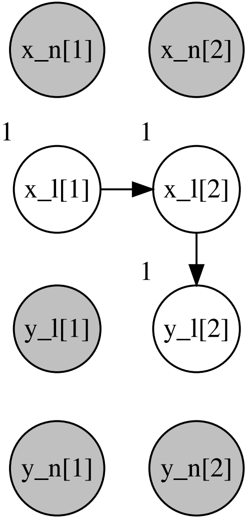
Delayed sampling



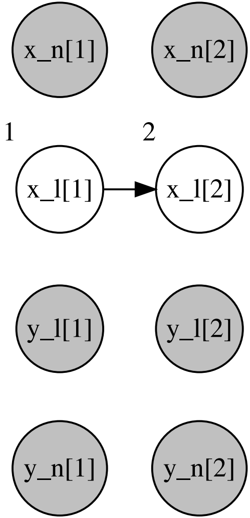
Delayed sampling



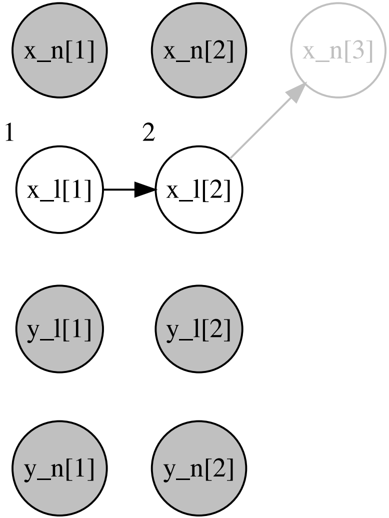
Delayed sampling



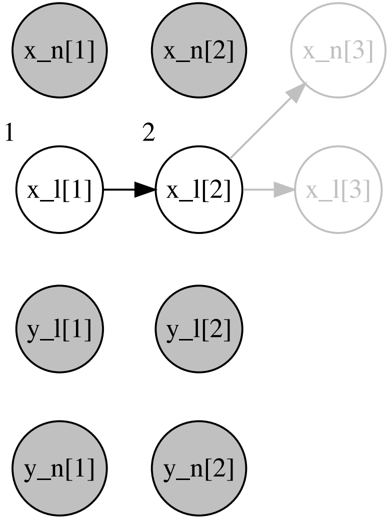
Delayed sampling



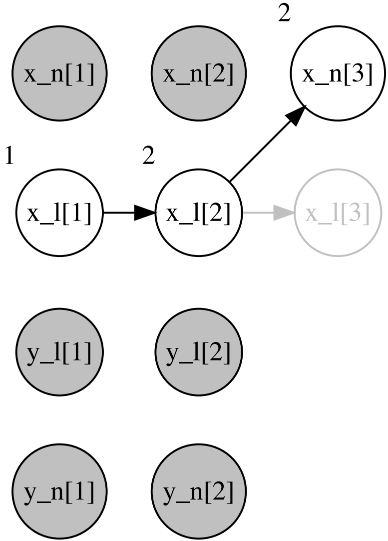
Delayed sampling



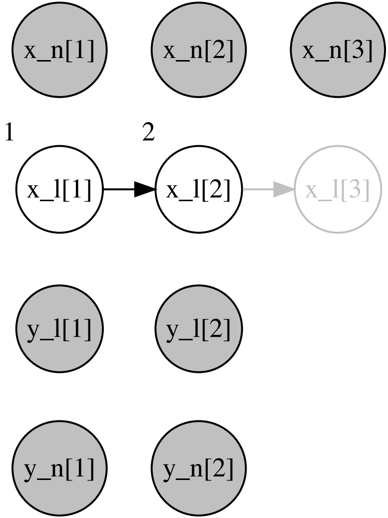
Delayed sampling



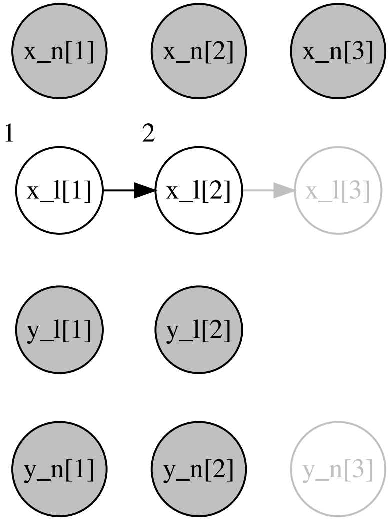
Delayed sampling



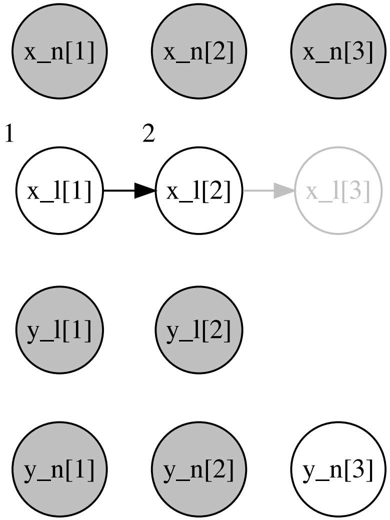
Delayed sampling



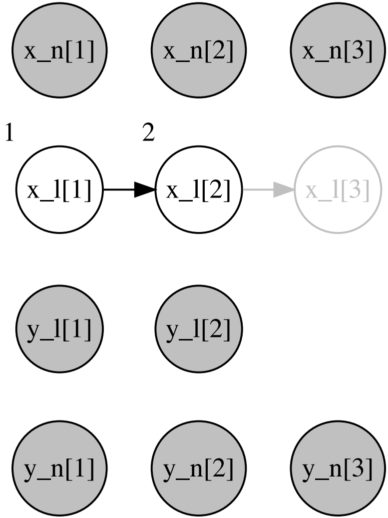
Delayed sampling



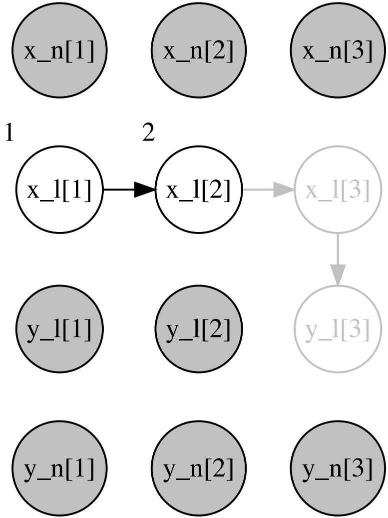
Delayed sampling



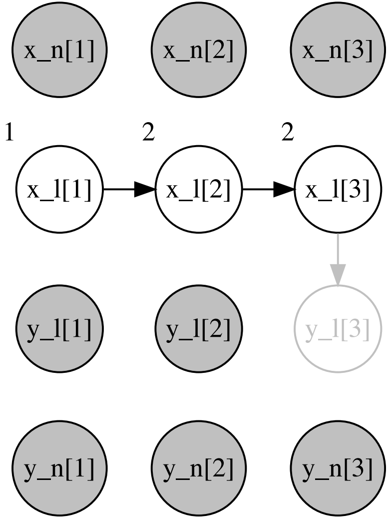
Delayed sampling



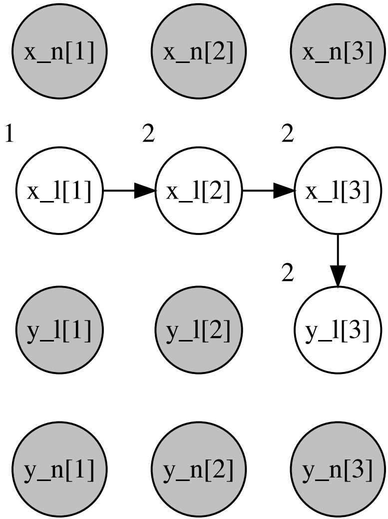
Delayed sampling



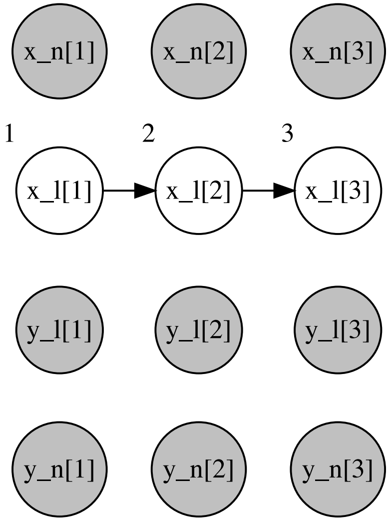
Delayed sampling



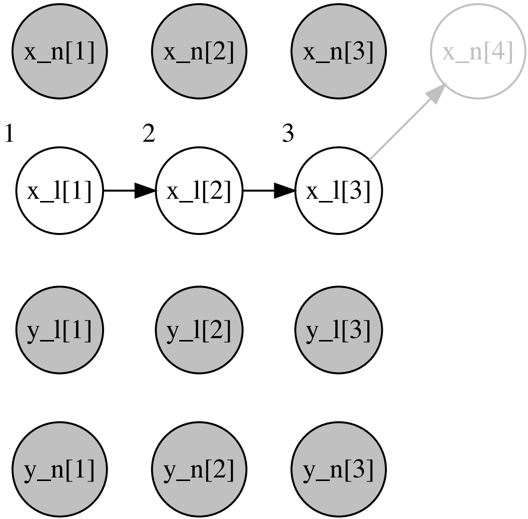
Delayed sampling



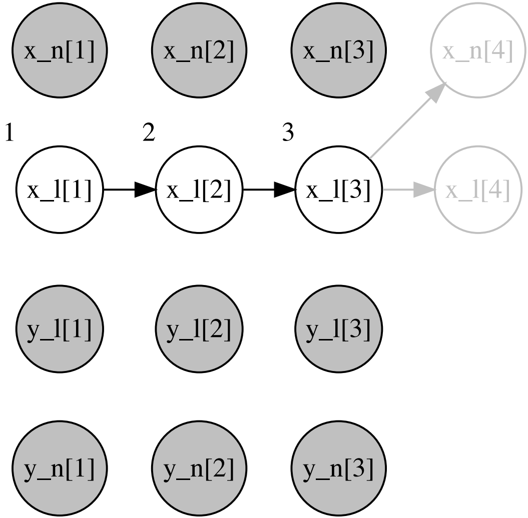
Delayed sampling



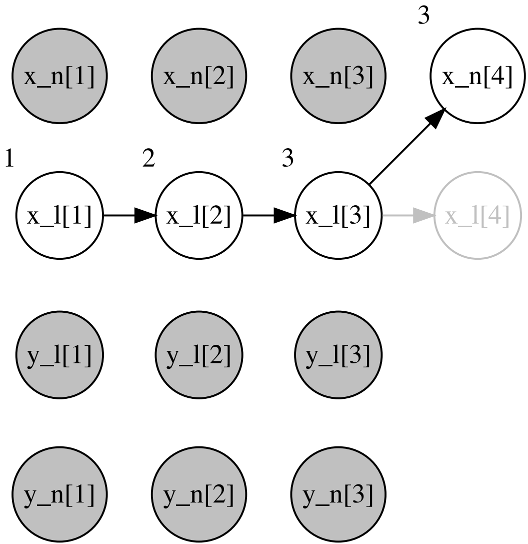
Delayed sampling



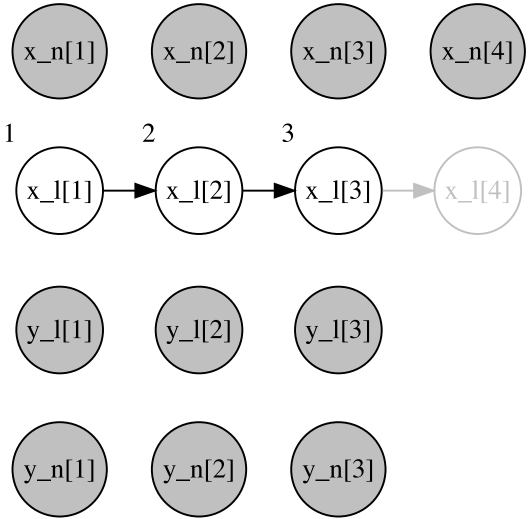
Delayed sampling



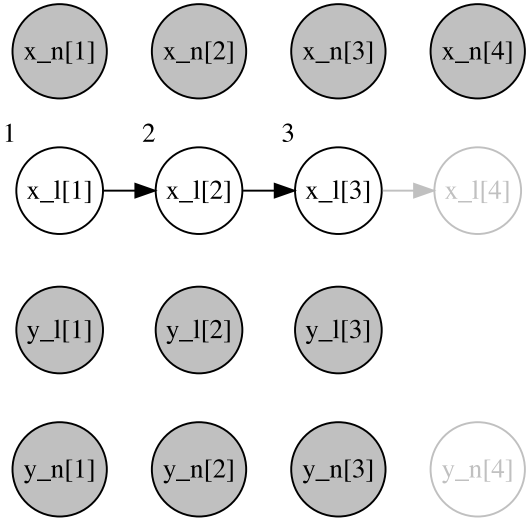
Delayed sampling



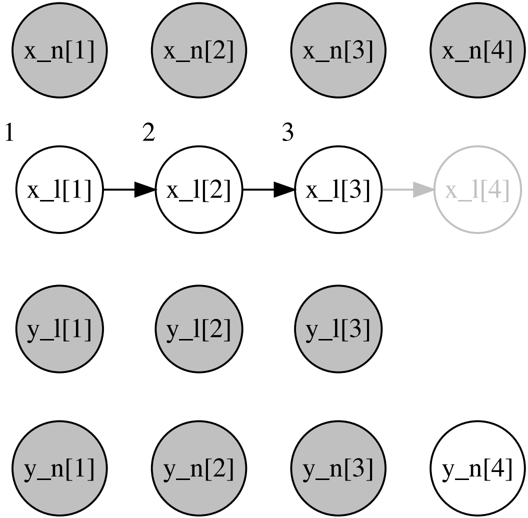
Delayed sampling



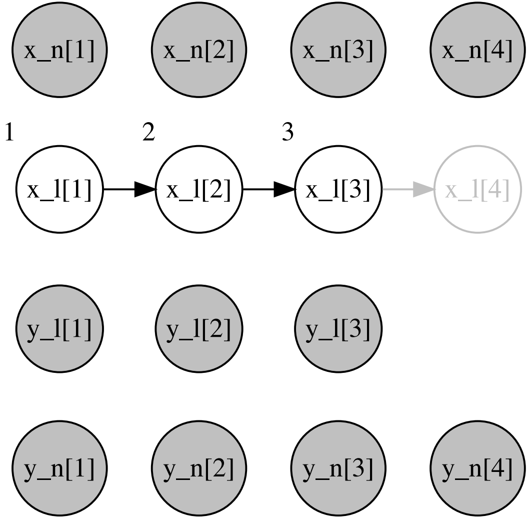
Delayed sampling



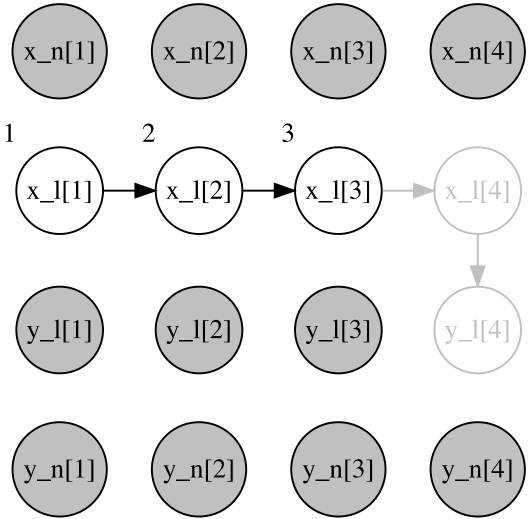
Delayed sampling



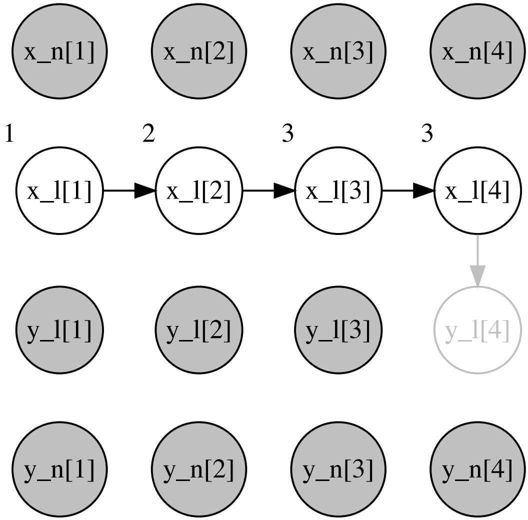
Delayed sampling



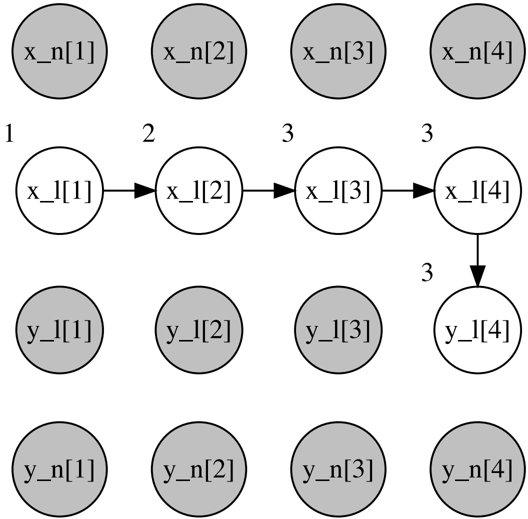
Delayed sampling



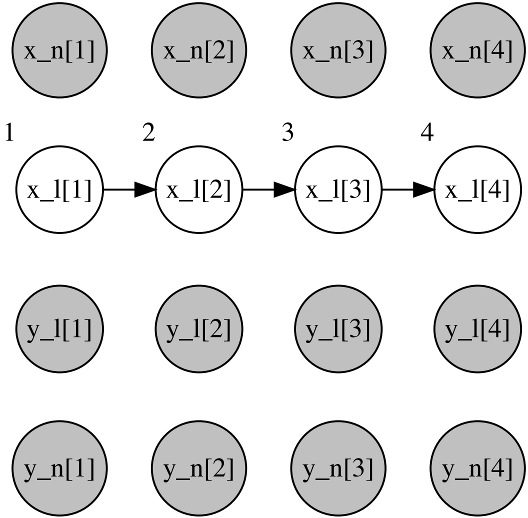
Delayed sampling



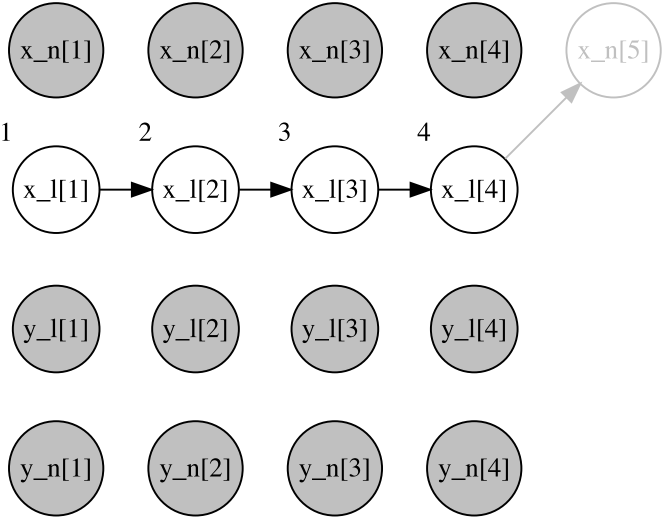
Delayed sampling



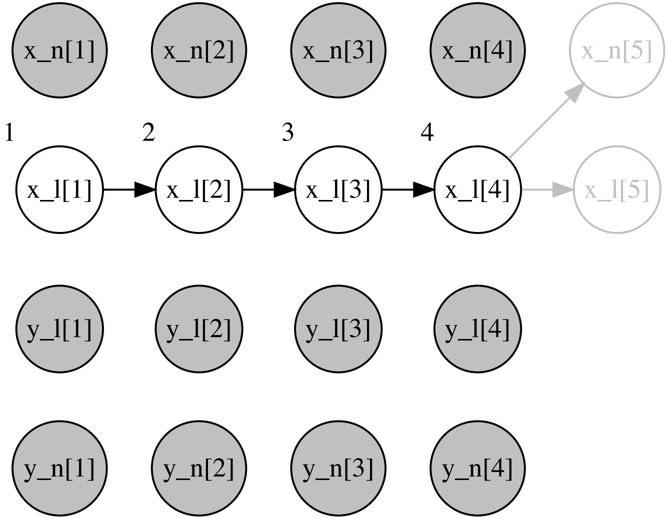
Delayed sampling



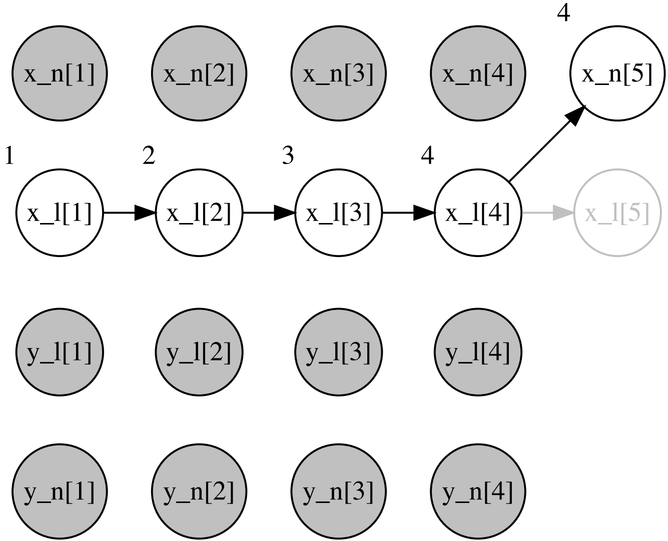
Delayed sampling



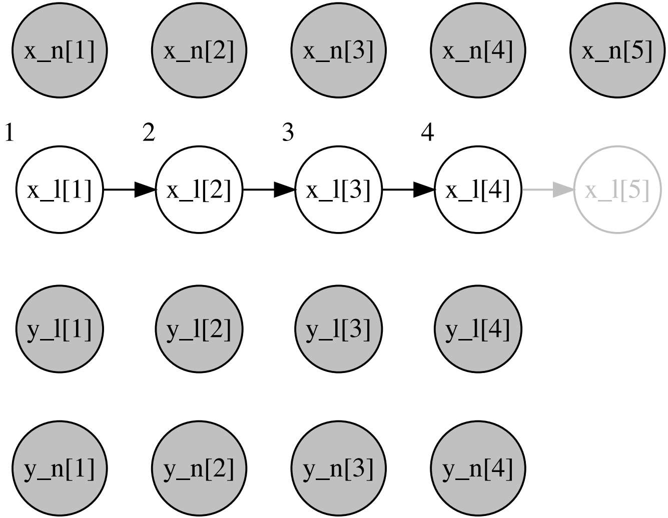
Delayed sampling



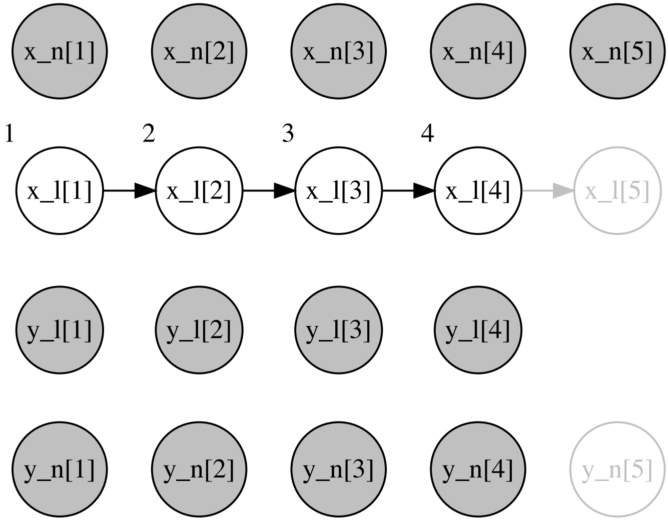
Delayed sampling



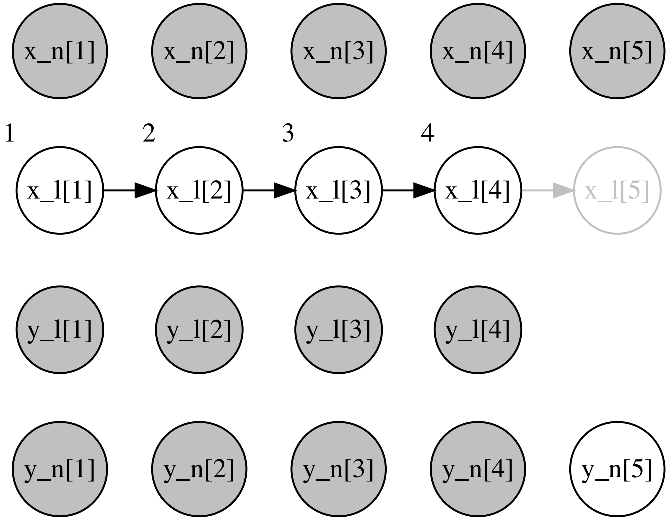
Delayed sampling



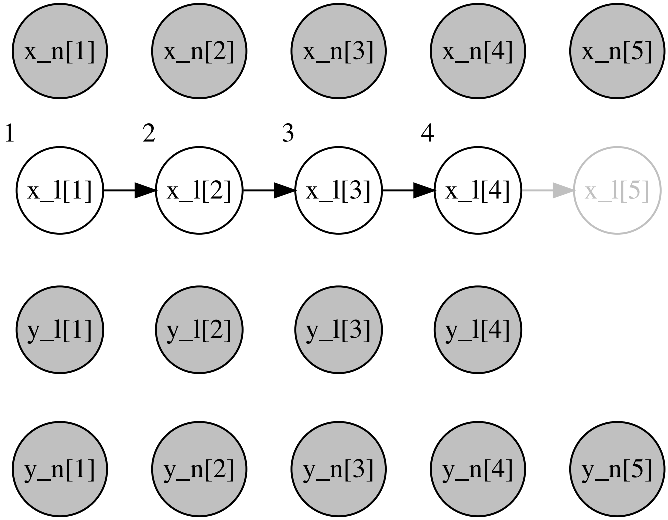
Delayed sampling



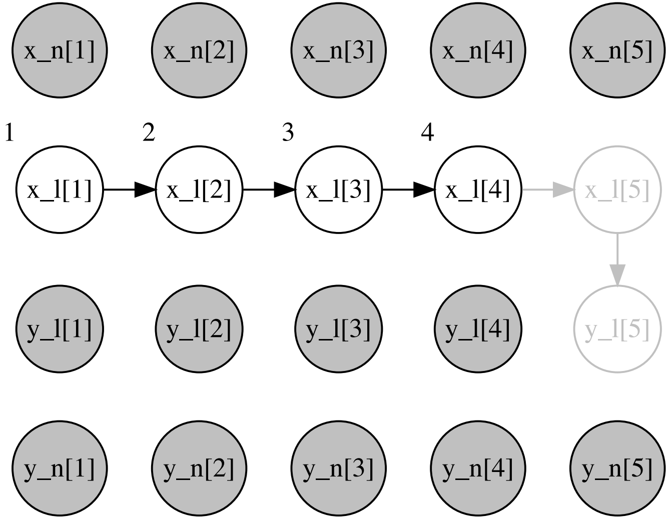
Delayed sampling



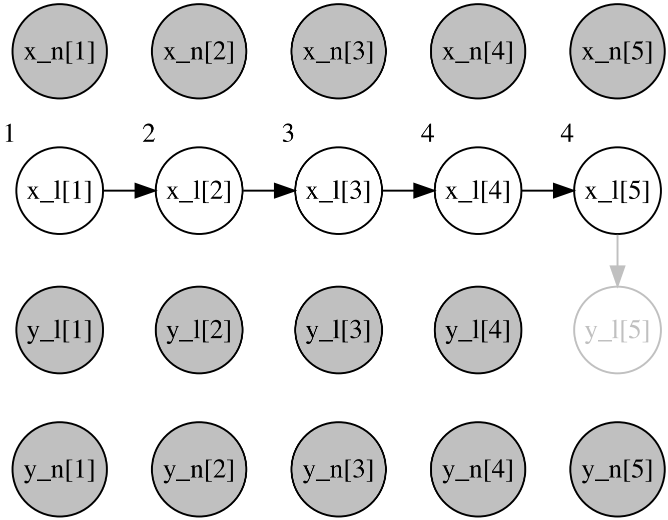
Delayed sampling



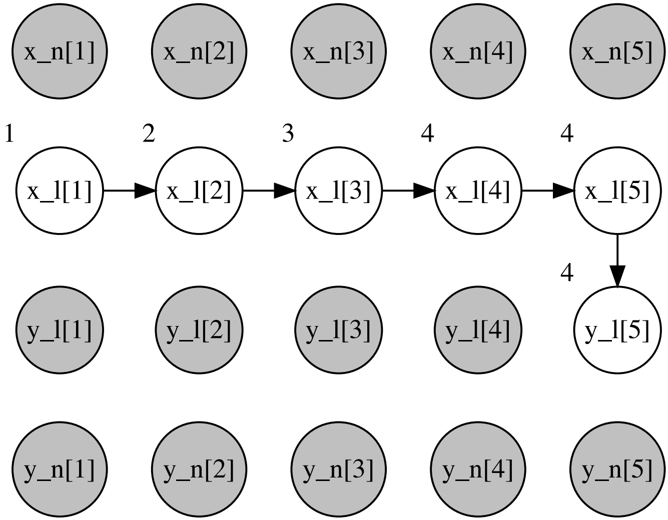
Delayed sampling



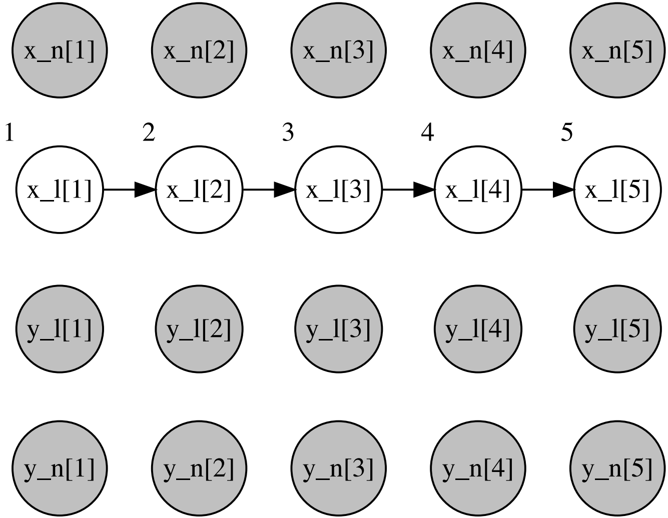
Delayed sampling



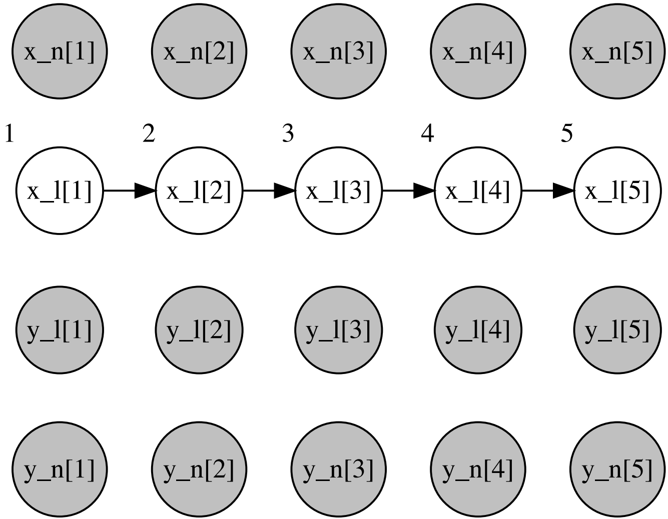
Delayed sampling



Delayed sampling



Delayed sampling: Rao–Blackwellized Particle Filter



Fibers

Fibers (also known as **coroutines** elsewhere) are like functions, but their execution can be paused and resumed.

- A function, when called, executes to completion and **returns** a value to the caller.
- A fiber, when called, executes to its first pause point and **yields** a value to the caller. The caller can then proceed with some other computation. Later, the caller may resume the fiber; it will execute to its next pause point and yield another value to the caller, and so on.

Fibers

- In Birch, fibers are used to simulate a probabilistic model. Each time an observation is encountered, the fiber pauses and **yields a weight**.
- This is a key ingredient for many inference methods (e.g. Sequential Monte Carlo).
- Fibers can be replicated. When resumed, replicated fibers proceed independently.
- A copy-on-write mechanism is used to minimise copying when replicating fibers.
- Can also be useful for **prospective computation**, e.g. anything with an accept/reject step.

Probabilistic operators

Optionals, randoms and fibers come together in the probabilistic operators of Birch. These are:

`a <~ b` **simulate** the distribution `b` and assign the value to `a`,

`a ~> b` **observe** the value `a` with distribution `b` and yield its log-likelihood from the current fiber,

`a ~ b` if `a` has a value then **observe** it, otherwise **simulate** it (perhaps lazily).

Looking ahead

- **Current focus** is pilot applications.
- **Near ahead** is adding new inference methods.
- **Further ahead** is performance tuning and parallelism.

Getting started guide and tutorial available on the website: `birch-lang.org`.