The Birch Probabilistic Programming Language

birch-lang.org

Lawrence Murray
Department of Information Technology, Uppsala University

Outline

1. Graphical models → programmatic models.
2. Birch: motivation and design.
1 Graphical models → programmatic models
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.

Graphical models

Figure: Benwing https://commons.wikimedia.org/wiki/File:Bayesian-gaussian-mixture.svg
Graphical models to probabilistic programs

- Each of these extensions makes the language more compact, more expressive, or both.
Each of these extensions makes the language more compact, more expressive, or both.

The most expressive languages are known as Turing complete or universal languages. These include all general-purpose programming languages.

Programmatic models are a superset of graphical models. Models that typically fall in the programmatic class but not the graphical class include anything nonparametric or with an unbounded number of variables, multi-object tracking models, structure learning models, multi-level models.
Graphical models to probabilistic programs

- Each of these extensions makes the language more compact, more expressive, or both.

- The most expressive languages are known as Turing complete or universal languages. These include all general-purpose programming languages.

- Probabilistic models written in such languages are known as universal probabilistic programs.
Graphical models to programmatic models

- Each of these extensions makes the language more compact, more expressive, or both.

- The most expressive languages are known as **Turing complete** or **universal** languages. These include all general-purpose programming languages.

- Probabilistic models written in such languages are known as **programmatic models**.
Graphical models to programmatic models

- Each of these extensions makes the language more compact, more expressive, or both.

- The most expressive languages are known as Turing complete or universal languages. These include all general-purpose programming languages.

- Probabilistic models written in such languages are known as programmatic models.

- Programmatic models are a superset of graphical models.

- Models that typically fall in the programmatic class but not the graphical class include anything nonparametric or with an unbounded number of variables, multi-object tracking models, structure learning models, multi-level models.

2. Countably infinite set of random variables \( \{ V_k \}_{k=1}^{\infty} \).

3. As the model executes we encounter a finite subset of \( \{ V_k \}_{k=1}^{\infty} \) in some order \( \sigma \), producing a sequence \( (V_{\sigma(k)})_{k=1}^{\sigma} \), where \( \sigma[k] \) is given by a deterministic function \( \text{Ne} \) of the random variates so far: \( \sigma[k] = \text{Ne}(v_{\sigma[1]}, \ldots, v_{\sigma[k-1]}) \).

4. When \( V_{\sigma[k]} \) is encountered, it is associated with a distribution

\[
V_{\sigma[k]} \sim p_{\sigma[k]} \left( \text{d}v_{\sigma[k]} \mid \text{Pa}(v_{\sigma[1]}, \ldots, v_{\sigma[k-1]}) \right),
\]

with \( \text{Pa} \) a deterministic function selecting a subset of its arguments.
Programmatic models

At some point execution terminates, having simulated

\[ p_{\sigma}(dv_{\sigma[1]}, \ldots, dv_{\sigma[|\sigma|]}) = \prod_{k=1}^{|\sigma|} p_{\sigma[k]}(dv_{\sigma[k]} \mid Pa(v_{\sigma[1]}, \ldots, v_{\sigma[k-1]})) \, . \]
Programmatic models

At some point execution terminates, having simulated

\[
p_{\sigma}(dv_{\sigma[1]}, \ldots, dv_{\sigma[|\sigma|]}) = \prod_{k=1}^{|\sigma|} p_{\sigma[k]}(dv_{\sigma[k]} | Pa(v_{\sigma[1]}, \ldots, v_{\sigma[k-1]})) .
\]

We will be interested in executing the code several times. The \(n\)th execution will be associated with the distribution \(p_{\sigma_n}\), given by

\[
p_{\sigma_n}(dv_{\sigma_n[1]}, \ldots, dv_{\sigma_n[|\sigma_n|]}) = \prod_{k=1}^{|\sigma_n|} p_{\sigma_n[k]}(dv_{\sigma_n[k]} | Pa(v_{\sigma_n[1]}, \ldots, v_{\sigma_n[k-1]})) ,
\]

with \(\sigma_n[k] = Ne(v_{\sigma_n[1]}, \ldots, v_{\sigma_n[k-1]}))\). Subscript \(n\) is used to denote execution-dependent variables.
Programmatic models

For different executions \( n \) and \( m \), it is possible for:

- the number of random variables encountered, \(|\sigma_n|\) and \(|\sigma_m|\), to differ,
- the sequences \((V_{\sigma_n[k]})_{k=1}^{\sigma_n}\) and \((V_{\sigma_m[k]})_{k=1}^{\sigma_m}\) to differ, and
- the two subsets \(\{V_{\sigma_n[k]}\}_{k=1}^{\sigma_n}\) and \(\{V_{\sigma_m[k]}\}_{k=1}^{\sigma_m}\) to be different (and even disjoint).
Programmatic models

For different executions \( n \) and \( m \), it is possible for:

- the number of random variables encountered, \( |\sigma_n| \) and \( |\sigma_m| \), to differ,
- the sequences \( (V_{\sigma_n[k]})_{k=1}^{\sigma_n} \) and \( (V_{\sigma_m[k]})_{k=1}^{\sigma_m} \) to differ, and
- the two subsets \( \{V_{\sigma_n[k]}\}_{k=1}^{\sigma_n} \) and \( \{V_{\sigma_m[k]}\}_{k=1}^{\sigma_m} \) to be different (and even disjoint).

In general, \( p_{\sigma_n} \) and \( p_{\sigma_m} \) are not the same, but rather components of a mixture.
Programmatic models

For different executions \( n \) and \( m \), it is possible for:

- the number of random variables encountered, \( |\sigma_n| \) and \( |\sigma_m| \), to differ,
- the sequences \( (V_{\sigma_n[k]})_{k=1}^{\sigma_n} \) and \( (V_{\sigma_m[k]})_{k=1}^{\sigma_m} \) to differ, and
- the two subsets \( \{V_{\sigma_n[k]}\}_{k=1}^{\sigma_n} \) and \( \{V_{\sigma_m[k]}\}_{k=1}^{\sigma_m} \) to be different (and even disjoint).

In general, \( p_{\sigma_n} \) and \( p_{\sigma_m} \) are not the same, but rather components of a mixture.

The big issue is **stochastic branching**, i.e. conditionals that depend on random outcomes. This means that the set of random variables, and their relationships, can only be revealed through execution.
Challenges

Inference is hard:

- For MCMC: programmatic models may be transdimensional.

- For SMC: programmatic models may be misaligned.
Challenges

Inference is hard:

- For MCMC: programmatic models may be **transdimensional**.
- For SMC: programmatic models may be **misaligned**.

**But aren’t necessarily.** This is only the most general case. Part of probabilistic programming is identifying, either at compile time or runtime, simpler cases for which specialized inference methods can be applied. (But this isn’t easy either.)
2 Birch: motivation and design
Birch

- Generic, object-oriented, and probabilistic programming paradigms.
- Draws inspiration from many places, including LibBi, for which it is something of a successor, but also modern object-oriented languages such as Swift.
- Maintains the C/C++ basis of LibBi: compiles to C++14, uses standard C/C++ libraries for numerical computing such as STL, Boost, Eigen, OpenMP, CUDA.
- Free and open source, under the Apache 2.0 license.

- See birch-lang.org
C++14 provides a lot of things we would like to quarantine.

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

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.
Birch

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

(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
Birch

(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
Birch

The big step up from LibBi is that Birch is a **universal probabilistic programming language**.

- Birch is a Turing-complete programming language, with support for conditionals, recursion, higher-order functions, etc.
Birch

The big step up from LibBi is that Birch is a universal probabilistic programming language.

- Birch is a Turing-complete programming language, with support for conditionals, recursion, higher-order functions, etc.
  → programmatic models
The big step up from LibBi is that Birch is a **universal probabilistic programming language**.

- Birch is a Turing-complete programming language, with support for conditionals, recursion, higher-order functions, etc. → **programmatic models**

- This requires dynamic memory management, which is provided by reference-counted garbage collection.

- Performance tuning is much more difficult, as fewer opportunities for vectorization.
Models in Birch

▶ Preferably, models are implemented by defining the **joint distribution**.

▶ That is, program code does not distinguish between latent and observed variables, this distinction is made at runtime according to value assignment.

▶ Models are implemented so as to reveal **structure** and **form**.

▶ Knowing something about structure and form helps to tailor the inference algorithm.
Example: Bayesian linear regression model

```bi
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

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

```c++
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

```cpp
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);
}
```
Methods in Birch

Inference methods are also written in the Birch language.

Include:

- Full and partial analytical solutions
- Importance sampling
- Bootstrap particle filter
- Alive particle filter
- Auxiliary particle filter (via partial analytical solutions)
- Rao–Blackwellized particle filter (via partial analytical solutions)
- Particle MCMC (soon)

Example #1

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

Code

\[ x \sim \text{Gaussian}(0.0, 1.0); \]
\[
\text{for (n in 1..N) { }
   y[n] \sim \text{Gaussian}(x, 1.0);
}\]
\[
\text{stdout.print(x);}
\]

Checkpoint

\text{assume } x
Example #1

Code

\[
x \sim \text{Gaussian}(0.0, 1.0);
\]

\[
\text{for (n in 1..N) {}
\]
\[
\quad y[n] \sim \text{Gaussian}(x, 1.0);
\]
\[
}\n\]

\[
\text{stdout.print(x);}
\]

Checkpoint
Example #1

Code

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

Checkpoint

```plaintext
observe y[n]
```
Example #1

Code

\[
x \sim \text{Gaussian}(0.0, 1.0);
\]

\[
\text{for } (n \text{ in } 1..N) \{
\quad y[n] \sim \text{Gaussian}(x, 1.0);
\}
\]

\text{observe } y[n]

\[
\text{stdout.print}(x);
\]

Checkpoint

\[x, y_1, y_2, y_3, y_4, y_5\]
Example #1

Code

\[
x \sim \text{Gaussian}(0.0, 1.0);
\]

for (n in 1..N) {
  \[
y[n] \sim \text{Gaussian}(x, 1.0);
\]
}

\[
\text{observe } y[n]
\]

stdout.print(x);

Checkpoint

\[
\text{x, y[1], y[2], y[3], y[4], y[5]}
\]
### Example #1

**Code**

\[ x \sim \text{Gaussian}(0.0, 1.0); \]

\[
\text{for (n in 1..N) {  y[n] \sim \text{Gaussian}(x, 1.0);  
}\]

\[
\text{observe y[n]} \]

\[
\text{stdout.print(x);  
}\]

**Checkpoint**

<table>
<thead>
<tr>
<th>Code</th>
<th>Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ x \sim \text{Gaussian}(0.0, 1.0); ]</td>
<td>[ \text{observe y[n]} ]</td>
</tr>
</tbody>
</table>
| \[
\text{for (n in 1..N) {  y[n] \sim \text{Gaussian}(x, 1.0);  
}\] | |
| \[
\text{stdout.print(x);  
]\] | |

![Diagram](attachment:image.png)
Example #1

Code

\[ x \sim \text{Gaussian}(0.0, 1.0); \]
\[
\text{for } (n \text{ in } 1..N) \{ \\
\quad y[n] \sim \text{Gaussian}(x, 1.0); \\
\}
\]
\[
\text{stdout.print}(x);
\]

Checkpoint

\[
\text{observe } y[n]
\]
Example #1

Code

\[ x \sim \text{Gaussian}(0.0, 1.0); \]
\[
\text{for} \ (n \ \text{in} \ 1..N) \ { \\
\quad y[n] \sim \text{Gaussian}(x, 1.0); \}
\]
\[ \text{observe} \ y[n] \]
\[ \text{stdout.print}(x); \]

Checkpoint

\[ x \sim \text{Gaussian}(0.0, 1.0); \]
\[
\text{for} \ (n \ \text{in} \ 1..N) \ { \\
\quad y[n] \sim \text{Gaussian}(x, 1.0); \}
\]
\[ \text{observe} \ y[n] \]
\[ \text{stdout.print}(x); \]
Example #1

Code

\[ x \sim \text{Gaussian}(0.0, 1.0); \]
\[
\text{for } (n \text{ in } 1..N) \{
\quad y[n] \sim \text{Gaussian}(x, 1.0); \\
\}
\]
\[ \text{observe } y[n] \]
\[ \text{stdout.print}(x); \]

Checkpoint

\[ x^2 \]
\[ y[1] \]
\[ y[2] \]
Example #1

Code

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

Checkpoint

```r
observe y[n]
```
Example #1

x \sim \text{Gaussian}(0.0, 1.0);
for (n in 1..N) {
    y[n] \sim \text{Gaussian}(x, 1.0);
    \text{observe } y[n]
}

\text{stdout.print}(x);
Example #1

<table>
<thead>
<tr>
<th>Code</th>
<th>Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \sim \text{Gaussian}(0.0, 1.0);$</td>
<td></td>
</tr>
<tr>
<td>for (n in 1..N) {</td>
<td>observe $y[n]$</td>
</tr>
<tr>
<td>$y[n] \sim \text{Gaussian}(x, 1.0);$</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>stdout.print(x);</td>
<td></td>
</tr>
</tbody>
</table>

Example #1

Code

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

Checkpoint

```
observe y[n]
```

Diagram:

```
x

3

```
Example #1

Code

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

Checkpoint

```
x
```

```
3
```

```
y[4]
```

```
y[1]
y[2]
y[3]
y[5]
```

Lawrence Murray 22 / 37
Example #1

**Code**

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

**Checkpoint**

```
observe y[n]
```

Example #1

x ~ Gaussian(0.0, 1.0);
for (n in 1..N) {
    \textbf{y}[n] ~ Gaussian(x, 1.0);
    \texttt{observe y}[n]
}
stdout.print(x);
Example #1

Code

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

Checkpoint

```c
observe y[n]
```

stdout.print(x);
Example #1

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

5
x

### Example #1

<table>
<thead>
<tr>
<th>Code</th>
<th>Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \sim \text{Gaussian}(0.0, 1.0);$</td>
<td></td>
</tr>
<tr>
<td>for (n in 1..N) {</td>
<td></td>
</tr>
<tr>
<td>$y[n] \sim \text{Gaussian}(x, 1.0);$</td>
<td>value $x$</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>stdout.print($x$);</td>
<td></td>
</tr>
</tbody>
</table>

```
5
x
```

```
```

Lawrence Murray 22 / 37
Example #1

Code

\[ x \sim \text{Gaussian}(0.0, 1.0); \]
\[
\text{for (n in 1..N) { }
    \quad y[n] \sim \text{Gaussian}(x, 1.0);
\}
\]
\indent \textit{stdout.print(x);}
Example #1

**Code**

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

**Checkpoint**

```
x
```
Example #2

Code

```
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]);
```

Checkpoint
Example #2

Code

\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 1.0); \]
for (t in 2..T) {
    \[ x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0); \]
    \[ y[t] \sim \text{Gaussian}(x[t], 1.0); \]
}

\text{stdout.print}(x[1]);

Checkpoint

\textit{assume} \ x[1]
Example #2

<table>
<thead>
<tr>
<th>Code</th>
<th>Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x[1] \sim \text{Gaussian}(0.0, 1.0);$</td>
<td>observe $y[1]$</td>
</tr>
<tr>
<td>$y[1] \sim \text{Gaussian}(x[1], 1.0);$</td>
<td></td>
</tr>
</tbody>
</table>
| for (t in 2..T) {
  $x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0);$  |
|   $y[t] \sim \text{Gaussian}(x[t], 1.0);$  |
| }                                          |             |
| stdout.print($x[1]$);                     |             |

![Diagram of variables $x[1]$ and $y[1]$](image)
Example #2

Code

\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 1.0); \]

for (t in 2..T) {
    \[ x[t] \sim \text{Gaussian}(a \cdot x[t - 1], 1.0); \]
    \[ y[t] \sim \text{Gaussian}(x[t], 1.0); \]
}

stdout.print(x[1]);

Checkpoint

observe \( y[1] \)
Example #2

**Code**

```plaintext
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]);
```

**Checkpoint**

```plaintext
observe y[1]
```
Example #2

Code

\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 1.0); \]
for (t in 2..T) {
    \[ x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0); \]
    \[ y[t] \sim \text{Gaussian}(x[t], 1.0); \]
}
\text{stdout.print}(x[1]);

Checkpoint

\text{observe} \ y[1]

\begin{center}
\begin{tikzpicture}
    \node [circle, fill=blue!20] (x1) at (0,0) {$x[1]$};
    \node [circle, fill=green!20] (y1) at (1,-1) {$y[1]$};
\end{tikzpicture}
\end{center}
Example #2

**Code**

\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 1.0); \]
\[
\text{for } (t \in 2..T) \{ \\
\quad x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0); \\
\quad y[t] \sim \text{Gaussian}(x[t], 1.0); \\
\}
\]
\[ \text{stdout.print}(x[1]); \]

**Checkpoint**

\[ \text{assume } x[t] \]

\[
\begin{array}{cc}
\text{Lawrence Murray} & 23 / 37 \\
\end{array}
\]
Example #2

Code

\[
x[1] \sim \text{Gaussian}(0.0, 1.0);
\]
\[
y[1] \sim \text{Gaussian}(x[1], 1.0);
\]
\[
\text{for (t in 2..T) }
\text{
    x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0);
    y[t] \sim \text{Gaussian}(x[t], 1.0);
}\]
\[
\text{observe } y[t]
\]
\[
\text{stdout.print(x[1]);}
\]
Example #2

Code

```r
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]
```

Checkpoint

```
stdout.print(x[1]);
```

Diagram:

```
1 |
```

```
   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]);
```
Example #2

<table>
<thead>
<tr>
<th>Code</th>
<th>Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x[1] \sim \text{Gaussian}(0.0, 1.0);$</td>
<td>observe $y[t]$</td>
</tr>
<tr>
<td>$y[1] \sim \text{Gaussian}(x[1], 1.0);$</td>
<td></td>
</tr>
<tr>
<td>for (t in 2..T) {</td>
<td></td>
</tr>
<tr>
<td>$x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0);$</td>
<td></td>
</tr>
<tr>
<td>$y[t] \sim \text{Gaussian}(x[t], 1.0);$</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>stdout.print(x[1]);</td>
<td></td>
</tr>
</tbody>
</table>
Example #2

Code

\[
x[1] \sim \text{Gaussian}(0.0, 1.0); \\
y[1] \sim \text{Gaussian}(x[1], 1.0); \\
\text{for} \ (t \ in \ 2..T) \ { \\
\quad x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0); \\
\quad y[t] \sim \text{Gaussian}(x[t], 1.0); \ \\
} \text{observe} \\ y[t] \\
\text{stdout.print}(x[1]);
\]

Checkpoint

\[
\text{observe} \ y[t] \\
\]

<table>
<thead>
<tr>
<th>x[1]</th>
<th>y[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Diagram:

1. \(x[1]\) -> \(x[2]\)
2. \(y[1]\)
3. \(y[2]\)
Example #2

Code

\[
x[1] \sim \text{Gaussian}(0.0, 1.0);
\]

\[
y[1] \sim \text{Gaussian}(x[1], 1.0);
\]

for (t in 2..T) {

\[
x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0);
\]

\[
y[t] \sim \text{Gaussian}(x[t], 1.0);
\]

}

stdout.print(x[1]);

Checkpoint

\textbf{assume } x[t]
Example #2

Code

\[x[1] \sim \text{Gaussian}(0.0, 1.0)\];
\[y[1] \sim \text{Gaussian}(x[1], 1.0)\];
for (t in 2..T) {
    \[x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0)\];
    \[y[t] \sim \text{Gaussian}(x[t], 1.0)\];
}
\text{observe } y[t]

\text{stdout.print}(x[1])

Checkpoint
Example #2

**Code**

\[
\begin{align*}
x[1] &\sim \text{Gaussian}(0.0, 1.0); \\
y[1] &\sim \text{Gaussian}(x[1], 1.0);
\end{align*}
\]

\[
\text{for (t in 2..T) }
\]

\[
\begin{align*}
x[t] &\sim \text{Gaussian}(a\times x[t - 1], 1.0); \\
y[t] &\sim \text{Gaussian}(x[t], 1.0);
\end{align*}
\]

\[
\text{observe } y[t]
\]

\[
\text{stdout.print(x[1]);}
\]

**Checkpoint**

- \( x[1] \sim \text{Gaussian}(0.0, 1.0) \)
- \( y[1] \sim \text{Gaussian}(x[1], 1.0) \)
- \( x[t] \sim \text{Gaussian}(a\times x[t - 1], 1.0) \)
- \( y[t] \sim \text{Gaussian}(x[t], 1.0) \)
- observe \( y[t] \)

\[
\text{stdout.print(x[1]);}
\]

Diagram:

1. \( x[1] \)
2. \( x[2] \)
3. \( x[3] \)
4. \( y[1] \)
5. \( y[2] \)
6. \( y[3] \)
Example #2

Code

\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 1.0); \]

\[
\text{for } (t \text{ in } 2..T) \{ \\
\quad x[t] \sim \text{Gaussian}(a \times x[t-1], 1.0); \\
\quad y[t] \sim \text{Gaussian}(x[t], 1.0); \quad \text{observe } y[t] \\
\}
\]

\text{stdout.print}(x[1]);

Checkpoint

\[
1 \quad x[1] \rightarrow 2 \quad x[2] \rightarrow 2 \quad x[3] \quad y[3] \\
\]
Example #2

Code

\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 1.0); \]
\[
\text{for (t in 2..T) { } }
\]
\[ x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0); \]
\[ y[t] \sim \text{Gaussian}(x[t], 1.0); \]
\[
\text{observe y[t]} \]
\}
\]
\[
\text{stdout.print(x[1]);} \]

Checkpoint

\[
\text{Lawrence Murray 23 / 37} \]
Example #2

**Code**

\[
x[1] \sim \text{Gaussian}(0.0, 1.0);
\]

\[
y[1] \sim \text{Gaussian}(x[1], 1.0);
\]

for (t in 2..T) {
    \[
x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0);
    \]
    \[
y[t] \sim \text{Gaussian}(x[t], 1.0);
    \]
}

stdout.print(x[1]);

**Checkpoint**

\[
\text{assume } x[t]
\]
Example #2

**Code**

\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 1.0); \]
\[
\text{for (t in 2..T) { } }
\]
\[
\quad x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0); \]
\[
\quad y[t] \sim \text{Gaussian}(x[t], 1.0); \]
\]
\[ \text{observe } y[t] \]
\[ \text{stdout.print}(x[1]); \]

**Checkpoint**

\[ x[1] \]
\[ y[1] \]
\[ x[2] \]
\[ y[2] \]
\[ x[3] \]
\[ y[3] \]
\[ x[4] \]
\[ y[4] \]
Example #2

Code

\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 1.0); \]
for (t in 2..T) {
  \[ x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0); \]
  \[ y[t] \sim \text{Gaussian}(x[t], 1.0); \]
}\n
\texttt{stdout.print(x[1]);}

Checkpoint

\texttt{observe y[t]}

\[
\begin{align*}
1 & \quad x[1] \\
2 & \quad x[2] \\
3 & \quad x[3] \\
3 & \quad y[4]
\end{align*}
\]
Example #2

```
Code
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]
```

```
Check
```

```
stdout.print(x[1]);
```

```
   1
  -----> 2
    x[1]       x[2]

   3
  -----> 4

   3
  -----> 5
```
Example #2

Code

\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 1.0); \]

\textbf{for} (t in 2..T) {
  \[ x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0); \]
  \[ y[t] \sim \text{Gaussian}(x[t], 1.0); \]
}\n
\text{observe } y[t]

\textbf{println}(x[1]);

Checkpoint

Lawrence Murray 23 / 37
Example #2

Code

\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 1.0); \]
\[
\text{for} \ (t \ \text{in} \ 2..T) \ { \\
\quad x[t] \sim \text{Gaussian}(ax[t - 1], 1.0); \\
\quad y[t] \sim \text{Gaussian}(x[t], 1.0); \\
}\]
\[
\text{stdout.print}(x[1]);
\]

Checkpoint

\[ \text{assume} \ x[t] \]

Lawrence Murray 23 / 37
Example #2

Code

```
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]);
```

Checkpoint

```
observe y[t]
```
Example #2

<table>
<thead>
<tr>
<th>Code</th>
<th>Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x[1] \sim \text{Gaussian}(0.0, 1.0)$;</td>
<td></td>
</tr>
<tr>
<td>$y[1] \sim \text{Gaussian}(x[1], 1.0)$;</td>
<td></td>
</tr>
<tr>
<td>for (t in 2..T) {</td>
<td>observe $y[t]$</td>
</tr>
<tr>
<td>\hspace{1cm} $x[t] \sim \text{Gaussian}(a \times x[t-1], 1.0)$;</td>
<td></td>
</tr>
<tr>
<td>\hspace{1cm} $y[t] \sim \text{Gaussian}(x[t], 1.0)$;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>stdout.print($x[1]$);</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
x[1] 1
x[2] 2
x[3] 3
x[4] 4
x[5] 4
```
Example #2

Example #2:

Code:

\[
x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[
y[1] \sim \text{Gaussian}(x[1], 1.0); \]
\[
\text{for (} t \text{ in 2..} T \{ \}
\]
\[
\quad x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0); \]
\[
\quad y[t] \sim \text{Gaussian}(x[t], 1.0); \quad \textbf{observe} \quad y[t] \}
\]
\[
\text{stdout.print}(x[1]);
\]

Checkpoint:

\[
\text{observe } y[t]
\]
Example #2: Kalman Filter

Code

```r
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]);
```

Checkpoint

```
observe y[t]
```
Example #2: Kalman Filter

**Code**

\[
x[1] \sim \text{Gaussian}(0.0, 1.0);
\]

\[
y[1] \sim \text{Gaussian}(x[1], 1.0);
\]

\[
\text{for } (t \in 2..T) \{
\quad x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0);
\quad y[t] \sim \text{Gaussian}(x[t], 1.0);
\}
\]

\[
\text{stdout.print(x[1]);}
\]

**Checkpoint**

\[
\text{value } x[1]
\]

---

Lawrence Murray 23 / 37
Example #2: Kalman Filter

**Code**

\[
x[1] \sim \text{Gaussian}(0.0, 1.0); \\
y[1] \sim \text{Gaussian}(x[1], 1.0); \\
\text{for} \ (t \in 2..T) \ \\
\quad \text{\quad \quad \quad x}[t] \sim \text{Gaussian}(a \ast x[t - 1], 1.0); \\
\quad \text{\quad \quad \quad y}[t] \sim \text{Gaussian}(x[t], 1.0); \\
\quad \text{\quad \quad \quad } \}
\]

\text{stdout.print}(x[1]);

**Checkpoint**

\textbf{value} \ x[1]
## Example #2: Kalman Filter

### Code

```plaintext
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]);
```

### Checkpoint

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Value: x[1]
Example #2: Kalman Filter

Code

\[
x[1] \sim \text{Gaussian}(0.0, 1.0);
\]
\[
y[1] \sim \text{Gaussian}(x[1], 1.0);
\]
\[
\text{for} \ (t \ \text{in} \ 2..T) \ \{
\quad x[t] \ \sim \ \text{Gaussian}(ax[t - 1], 1.0);
\quad y[t] \ \sim \ \text{Gaussian}(x[t], 1.0);
\}
\]
\[
\text{stdout.print}(x[1]);
\]

Checkpoint

\[
\text{value} \ x[1]
\]
Example #2: Kalman Filter

Code

\[
\begin{align*}
x[1] & \sim \text{Gaussian}(0.0, 1.0); \\
y[1] & \sim \text{Gaussian}(x[1], 1.0); \\
\text{for } (t \text{ in } 2..T) \{} \\
& \quad x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0); \\
& \quad y[t] \sim \text{Gaussian}(x[t], 1.0); \\
\text{\}} \\
\text{stdout.print}(x[1]);
\end{align*}
\]

Checkpoint

value \(x[1]\)
### Example #2: Kalman Filter

<table>
<thead>
<tr>
<th>Code</th>
<th>Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x[1] \sim \text{Gaussian}(0.0, 1.0);$</td>
<td></td>
</tr>
<tr>
<td>$y[1] \sim \text{Gaussian}(x[1], 1.0);$</td>
<td></td>
</tr>
<tr>
<td>for (t in 2..T) {</td>
<td></td>
</tr>
<tr>
<td>$x[t] \sim \text{Gaussian}(a \times x[t - 1], 1.0);$</td>
<td></td>
</tr>
<tr>
<td>$y[t] \sim \text{Gaussian}(x[t], 1.0);$</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>stdout.print($x[1]$);</td>
<td></td>
</tr>
</tbody>
</table>
3 Birch: language features
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.
Optionals: idioms

Declare it:  

a:A?;

Assign it:  

a <- b;

Clear it:  

a <- nil;

Use it:  

if (a?) {
    f(a!);
}

A variable a:A always has a value; it is not necessary to check.
A variable a:A? may not have a value; it is usual to check.
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.
Randoms: idioms

Declare it: \[ a: \text{Random}\langle A \rangle; \]
Assign it a variate: \[ a \leftarrow b; \]
Associate it with a distribution: \[ a \sim \text{Distribution}(c); \]
Use it: \[ a \]

A random can be implicitly converted to the contained type (i.e. \( A \) in the example above), as long as it has had either a variate assigned to it, or a distribution associated to it (from which a variate can be simulated on-demand).
Fibers are like functions, but their execution can be suspended and resumed.

- A function, when called, executes to completion and **returns** a value to the caller.

- A fiber, when called, executes to its first suspension point and **yields** a value to the caller. The caller can then proceed with some other computation, and later resume the fiber.

- The same or similar is known elsewhere as **coroutines** or **generators**; the functional programming analogue is **continuation passing style**.
Fibers: idioms

Declare it:

```java
fiber f(a:A, b:B) -> C {
    c:C;
    // ...
    yield c;
    // ...
    yield c;
}
```

Use it:

```java
c:C! <- f(a, b);
while (c?) { // resume to the next yield point
    g(c!); // do something with the yield value
}
```
Probabilistic operators

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

\[ a \sim b \quad \text{simulate} \quad \text{the distribution} \quad b \quad \text{and assign the value to} \quad a, \]

\[ a \rightsquigarrow b \quad \text{observe} \quad \text{the value} \quad a \quad \text{with distribution} \quad b \quad \text{and yield its log-likelihood from the current fiber}, \]

\[ a \sim b \quad \text{if} \quad a \quad \text{has a value then} \quad \text{observe} \quad \text{it, otherwise} \quad \text{simulate} \quad \text{it (perhaps lazily)}. \]
Under the hood: fibers

- Fibers are used to simulate a probabilistic model. Each time an observation is encountered, the fiber suspends and yields a weight.

- Suspended fibers can be cloned, and their clones resumed independently.

- This is a key ingredient for inference methods that require resampling (e.g. Sequential Monte Carlo) or prospective computation (e.g. rejection sampling).

- A copy-on-write mechanism is used for efficient cloning.
Under the hood: copy-on-write

We know memory use for a particle filter is $O(T + N \log N)$ in time $T$ and particles $N$, i.e. a lot of memory can be shared.

Under the hood: computational graphs

- These are used to discover form for which analytical optimisation is possible, via delayed sampling.

- This includes finding opportunities for locally-optimal proposals, variable elimination, and Rao–Blackwellisation, either via conjugacy relationships or enumeration of discrete-valued random variables.

- In future they may be used for automatic differentiation too.

Under the hood: computational graphs

\[ \theta \sim \text{Uniform}(0.0, 1.0); \]
\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(g(x[1], \theta), 0.1); \]

for \( t \) in 2..T {
    \[ x[t] \sim \text{Gaussian}(f(x[t-1], \theta), 1.0); \]
    \[ y[t] \sim \text{Gaussian}(g(x[t], \theta), 0.1); \]
}
Under the hood: computational graphs

\[ \theta \sim \text{Uniform}(0.0, 1.0); \]
\[ x[1] \sim \text{Gaussian}(0.0, 1.0); \]
\[ y[1] \sim \text{Gaussian}(x[1], 0.1); \]
for t in 2..T {
  \[ x[t] \sim \text{Gaussian}(\theta \times x[t-1], 1.0); \]
  \[ y[t] \sim \text{Gaussian}(x[t], 0.1); \]
}
Probabilistic programming deals with the most expressive class of probabilistic models: programmatic models.

The Birch probabilistic programming language provides:

- Language features to support the paradigm, such as optionals, randoms, and fibers.
- Platform features to support the paradigm, such as copy-on-write for efficient memory use, and delayed sampling for partial analytical solutions.
- Parallel computing features.

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