Statically Bounded-Memory Delayed Sampling for Probabilistic Streams

Eric Atkinson
MIT

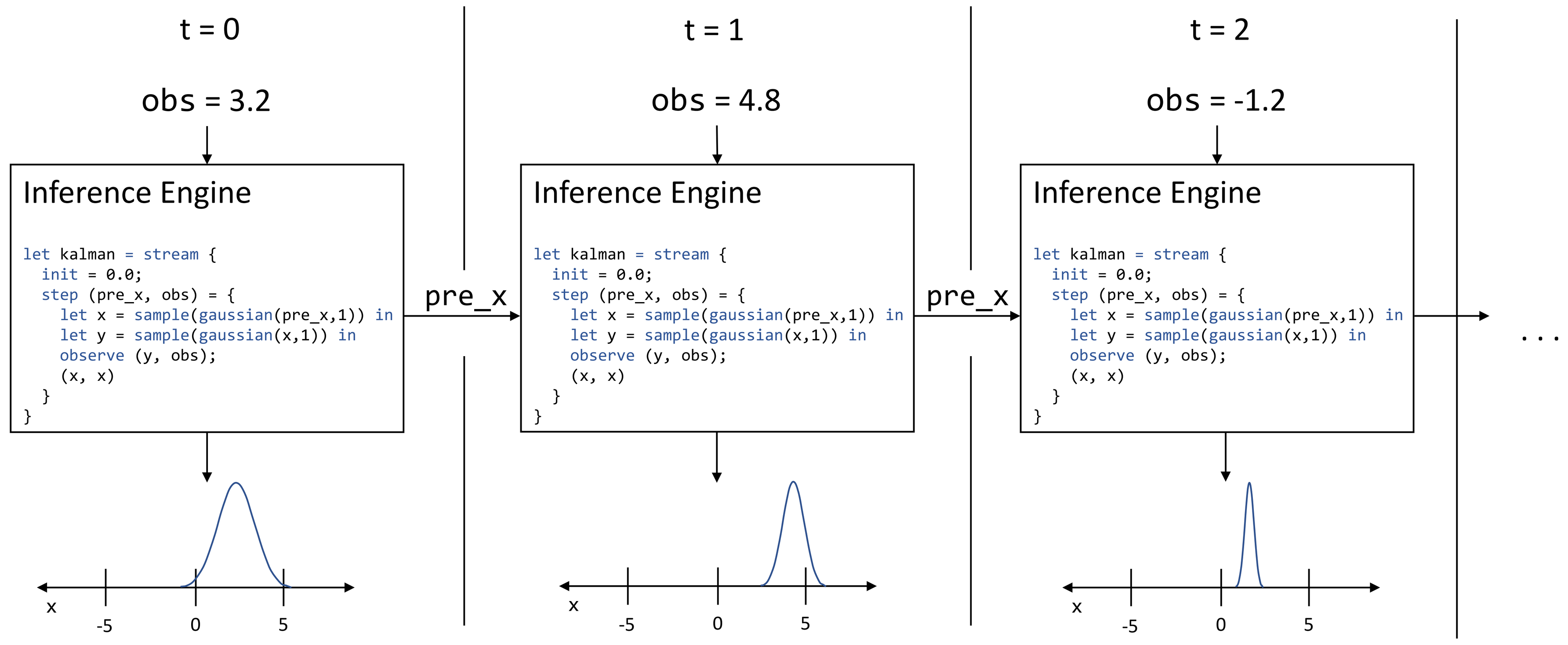
Guillaume Baudart
INRIA/ENS

Louis Mandel
IBM Research

Charles Yuan
MIT

Michael Carbin
MIT

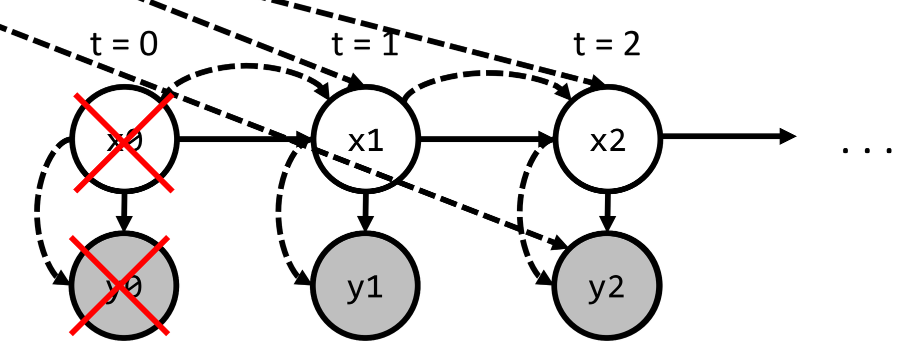
Probabilistic Programming with Streams



- **Goal:** Run inference in *bounded memory*.
- I.e. even if programs run for infinite time, should have a finite memory footprint

Bounded-Memory Delayed Sampling[1,2]

```
let kalman = stream {
  init = 0.0;
  step (pre_x, obs) = {
    let x = sample(gaussian(pre_x,1)) in
    let y = sample(gaussian(x,1)) in
    observe (y, obs);
    (x, x)
  }
}
```



1: Murray et. al. AISTATS 2018
2: Baudart et. al. PLDI 2020

- **Goal:** Bound the size of the delayed sampling graph.
- Maintain a finite set of *reachable* nodes that are accessible from pointers in the program state

1: Murray et. al. "Delayed Sampling and Automatic Rao—Blackwellization of Probabilistic Programs". AISTATS 2018
2: Baudart et. al. "Reactive Probabilistic Programming". PLDI 2020

Semantic Properties

- We can define properties on *traces* of probabilistic programs
- A trace records all operations the program executes

```
let kalman = stream {
  init = 0.0;
  step (pre_x, obs) = {
    let x = sample(gaussian(pre_x,1)) in
    let y = sample(gaussian(x,1)) in
    observe (y, obs);
    (x, x)
  }
}
```

```
x0 ← nil ::
y0 ← x0 ::
observe y0 ::
x1 ← x0 ::
y1 ← x1 ::
observe y1 ::
x2 ← x1 ::
y2 ← y1 ::
observe y2 ::
...
```

The *m*-consumed Property

Property: There exists a bound *m* such that every variable introduced is *m*-consumed.

```
let kalman = stream {
  init = 0.0;
  step (pre_x, obs) = {
    let x = sample(gaussian(pre_x,1)) in
    let y = sample(gaussian(x,1)) in
    observe (y, obs);
    (x, x)
  }
}
```

```
x0 ← nil ::
y0 ← x0 ::      → x0 is 1-consumed
observe y0 ::   → y0 is 0-consumed
x1 ← x0 ::
y1 ← x1 ::
observe y1 ::
x2 ← x1 ::
y2 ← y1 ::
observe y2 ::
...
```

The Unseparated Paths Property

Property: there exists an *n* such that at any time step t, no variable in the program state at t starts an unseparated path longer than *n*

```
let kalman = stream {
  init = 0.0;
  step (pre_x, obs) = {
    let x = sample(gaussian(pre_x,1)) in
    let y = sample(gaussian(x,1)) in
    observe (y, obs);
    (x, x)
  }
}
```

Program State

```
x0 ← nil ::
y0 ← x0 ::
observe y0 ::     t = 0     x0
x1 ← x0 ::
y1 ← x1 ::
observe y1 ::     t = 1     x1
x2 ← x1 ::
y2 ← y1 ::
observe y2 ::     t = 2     x2
...
```

x0 starts an unseparated path longer than *2*

The *m*-consumed Static Analysis

- **Key idea:** measure the variables introduced but not yet used
- Can approximate by taking a superset of introduced variables
- The analysis passes if the set is empty after a step

```
let kalman = stream {
  init = 0.0;
  step (pre_x, obs) = {
    let x = sample(gaussian(pre_x,1)) in
    let y = sample(gaussian(x,1)) in
    observe (y, obs);
    (x, x)
  }
}
```

	Introduced Variables
x0 ← nil ::	x0
y0 ← x0 ::	y0
observe y0	None

The Unseparated Paths Static Analysis

- **Key idea:** track an upper bound of unseparated paths between program variables
- Can approximate with a larger upper bound
- Analysis passes when longest path converges

```
let kalman = stream {
  init = 0.0;
  step (pre_x, obs) = {
    let x = sample(gaussian(pre_x,1)) in
    let y = sample(gaussian(x,1)) in
    observe (y, obs);
    (x, x)
  }
}
```

	Longest Unsep. Path
x0 ← nil ::	(x0, x0), 0
y0 ← x0 ::	(x0, y0), 1
observe y0 ::	(x0, y0), 1
x1 ← x0 ::	(x0, x1), 1
y1 ← x1 ::	(x1, y1), 1
observe y1 ::	(x1, y1), 1
x2 ← x1 ::	(x1, x2), 1
y2 ← y1 ::	(x2, y2), 1
observe y2 ::	(x2, y2), 1

Results

	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	✗	✓	✓	✓	✗	✓

Analysis is precise

Memory is Probabilistically Bounded

Memory is Always Bounded