

Deterministic stream-sampling for probabilistic programming: semantics and verification

Alexandra Silva, Fredrik Dahlqvist and Will Smith

Programming Principles, Logic, and Verification Group
Department of Computer Science
University College London

LICS '23

Motivation: pseudorandom number generation

- For which $f : \mathbb{N} \rightarrow [0, 1]$ does this program terminate?

```
n := 0;
while f(n) == rand()
  n += 1;
```

Motivation: pseudorandom number generation

- For which $f : \mathbb{N} \rightarrow [0, 1]$ does this program terminate?

```
n := 0;
while f(n) == rand()
  n += 1;
```

- 1 If `rand()` refers to the uniform *distribution* rather than a value, then for *all* f , with probability 1, this program terminates at $n = 0$.

Motivation: pseudorandom number generation

- For which $f : \mathbb{N} \rightarrow [0, 1]$ does this program terminate?

```
n := 0;
while f(n) == rand()
  n += 1;
```

- 1 If `rand()` refers to the uniform *distribution* rather than a value, then for *all* f , with probability 1, this program terminates at $n = 0$.
- 2 If `rand()` draws from a sequence of (Martin-Löf) random values, then for all *computable* f , this program terminates at $n \geq 0$.

Motivation: pseudorandom number generation

- For which $f : \mathbb{N} \rightarrow [0, 1]$ does this program terminate?

```
n := 0;
while f(n) == rand()
  n += 1;
```

- 1 If `rand()` refers to the uniform *distribution* rather than a value, then for *all* f , with probability 1, this program terminates at $n = 0$.
- 2 If `rand()` draws from a sequence of (Martin-Löf) random values, then for all *computable* f , this program terminates at $n \geq 0$.
- 3 If `rand()` is pseudorandom, then there is a computable f for which this program does not terminate.

Deterministic stream-sampling

- In actual practice, pseudorandom numbers are ubiquitous!

Deterministic stream-sampling

- In actual practice, pseudorandom numbers are ubiquitous!
- To address this ambiguity, we introduce a language in which samplers are infinite streams.

Deterministic stream-sampling

- In actual practice, pseudorandom numbers are ubiquitous!
- To address this ambiguity, we introduce a language in which samplers are infinite streams.
- We use a simply-typed lambda calculus with a **sampler type** ΣX for samplers on X .

Deterministic stream-sampling

- In actual practice, pseudorandom numbers are ubiquitous!
- To address this ambiguity, we introduce a language in which samplers are infinite streams.
- We use a simply-typed lambda calculus with a **sampler type** ΣX for samplers on X .
- Samplers $s : \Sigma X$ are distinct from the distributions $P \in \mathcal{P}X$ that they sample from.

Deterministic stream-sampling

- In actual practice, pseudorandom numbers are ubiquitous!
- To address this ambiguity, we introduce a language in which samplers are infinite streams.
- We use a simply-typed lambda calculus with a **sampler type** ΣX for samplers on X .
- Samplers $s : \Sigma X$ are distinct from the distributions $P \in \mathcal{P}X$ that they sample from.
- For example, $\text{flip} = (0, 1, 0, 1, \dots)$ is a valid sampler for the uniform distribution on $\{0, 1\}$.

Motivation: compositional sampler verification

- If `flip()` gives biased samples on $\{0, 1\}$, then what is the distribution of this program?

```
while true
  a := flip();
  b := flip();
  if a ≠ b
    return a;
```

Motivation: compositional sampler verification

- If `flip()` gives biased samples on $\{0, 1\}$, then what is the distribution of this program?

```
while true
  a := flip();
  b := flip();
  if a ≠ b
    return a;
```

- If consecutive samples from `flip()` are ‘independent’, then a should be uniform on $\{0, 1\}$.

Motivation: compositional sampler verification

- If `flip()` gives biased samples on $\{0, 1\}$, then what is the distribution of this program?

```
while true
  a := flip();
  b := flip();
  if a ≠ b
    return a;
```

- If consecutive samples from `flip()` are ‘independent’, then a should be uniform on $\{0, 1\}$.
- This ‘extractor’ really *inputs* a biased sampler `flip`, and *outputs* an unbiased sampler; how can we write it that way?

Sampler operations

- Many approaches in probability and statistics are *compositional*: built from a few fundamental sampler operations.

Sampler operations

- Many approaches in probability and statistics are *compositional*: built from a few fundamental sampler operations.
- In our language, these basic sampler operations are operations on infinite streams.

Sampler operations

- Many approaches in probability and statistics are *compositional*: built from a few fundamental sampler operations.
- In our language, these basic sampler operations are operations on infinite streams.
- For example, if $s : \Sigma X$ denotes (x_1, x_2, \dots) , then $\text{map}(f, s)$ denotes $(f(x_1), f(x_2), \dots)$.

Sampler operations

- Many approaches in probability and statistics are *compositional*: built from a few fundamental sampler operations.
- In our language, these basic sampler operations are operations on infinite streams.
- For example, if $\mathbf{s} : \Sigma X$ denotes (x_1, x_2, \dots) , then $\text{map}(f, \mathbf{s})$ denotes $(f(x_1), f(x_2), \dots)$.
- Samplers can be *weighted* streams: $\text{reweight}(g, \mathbf{s})$, where $g : X \rightarrow \mathbb{R}_{\geq 0}$, denotes the weighted stream $((x_1, g(x_1)), (x_2, g(x_2)), \dots)$.

Sampler operations

- Many approaches in probability and statistics are *compositional*: built from a few fundamental sampler operations.
- In our language, these basic sampler operations are operations on infinite streams.
- For example, if $\mathbf{s} : \Sigma X$ denotes (x_1, x_2, \dots) , then $\text{map}(f, \mathbf{s})$ denotes $(f(x_1), f(x_2), \dots)$.
- Samplers can be *weighted* streams: $\text{reweight}(g, \mathbf{s})$, where $g : X \rightarrow \mathbb{R}_{\geq 0}$, denotes the weighted stream $((x_1, g(x_1)), (x_2, g(x_2)), \dots)$.
- If $\mathbf{s} : \Sigma X$ denotes $(x_1, x_2, x_3, x_4, \dots)$, then $\mathbf{s}^2 : \Sigma(X \times X)$ denotes $((x_1, x_2), (x_3, x_4), \dots)$.

Samplers 'target' distributions

```
while true
  a := flip();
  b := flip();
  if a ≠ b
    return a;
```

```
let accept? = λa,b : B×B . if a ≠ b then 1 else 0 in
let first = λa,b : B×B . a in
map(first, reweight(accept?, flip2))
```

- This program implements the unbiasing technique discussed earlier.

Samplers 'target' distributions

```
while true
  a := flip();
  b := flip();
  if a ≠ b
    return a;
```

```
let accept? = λa,b : B×B . if a ≠ b then 1 else 0 in
let first = λa,b : B×B . a in
map(first, reweight(accept?, flip2))
```

- This program implements the unbiasing technique discussed earlier.
- To 'verify' is to prove: if flip^2 generates independent, biased samples on $B \times B$, then this program generates unbiased samples on B .

Samplers 'target' distributions

```
while true
  a := flip();
  b := flip();
  if a ≠ b
    return a;
```

```
let accept? = λa,b : B×B . if a ≠ b then 1 else 0 in
let first = λa,b : B×B . a in
map(first, reweight(accept?, flip2))
```

- This program implements the unbiasing technique discussed earlier.
- To 'verify' is to prove: if flip^2 generates independent, biased samples on $B \times B$, then this program generates unbiased samples on B .
- We formalise this using a 'targeting' relation: $s \rightsquigarrow P$ means that the sampler $s : \Sigma X$ generates samples from the distribution $P \in \mathcal{P}X$.

A calculus for targeting

- If samplers are compositionally built from ‘basic’ sampler operations, such as `map`, then to prove targeting, we introduce a **calculus** with inference rules such as

A calculus for targeting

- If samplers are compositionally built from ‘basic’ sampler operations, such as `map`, then to prove targeting, we introduce a **calculus** with inference rules such as

$$\frac{\Gamma \vdash \mathbf{s} : \Sigma S \rightsquigarrow P \quad \Gamma \vdash f : S \rightarrow T}{\Gamma \vdash \mathbf{map}(f, \mathbf{s}) : \Sigma T \rightsquigarrow \gamma \mapsto (\llbracket f \rrbracket (\gamma))_* P(\gamma)}$$

A calculus for targeting

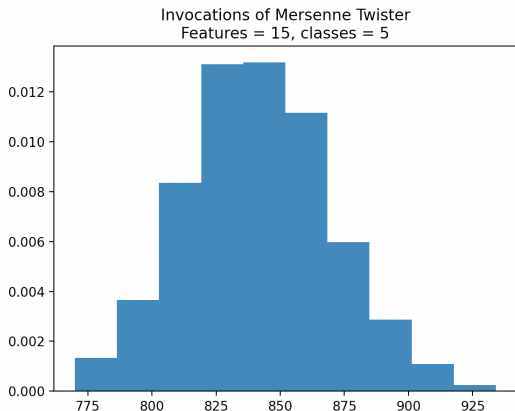
- If samplers are compositionally built from ‘basic’ sampler operations, such as `map`, then to prove targeting, we introduce a **calculus** with inference rules such as

$$\frac{\Gamma \vdash \mathbf{s} : \Sigma S \rightsquigarrow P \quad \Gamma \vdash f : S \rightarrow T}{\Gamma \vdash \mathbf{map}(f, \mathbf{s}) : \Sigma T \rightsquigarrow \gamma \mapsto (\llbracket f \rrbracket (\gamma))_* P(\gamma)}$$

- Caution: this rule does not hold for all (measurable) f ! For a sequence $(x_n)_{n \in \mathbb{N}}$, consider $f(x) = \mathbb{I}_{\{x_n : n \in \mathbb{N}\}}(x)$. (This is related to the first program discussed; see our paper for how we handle this.)

Does any of this matter in *practice*?

$$\begin{aligned} \mu &\sim N(\alpha_\mu, B_\mu) \\ \Sigma &\sim \text{InvWishart}(df, B_\Sigma) \\ w_c &\sim N(\alpha_c, \Sigma) \\ W &= (w_1^T, \dots, w_C^T) \\ x &\sim N(\mu, \Sigma) \\ y &= \sigma(W^T x) \end{aligned}$$



- The standard Mersenne twister in C++ is 623-equidistributed with 32-bit outputs; practical models can certainly exceed this!

Conclusion

- Our language enables the compositional construction of samplers using basic sampler operations.

Conclusion

- Our language enables the compositional construction of samplers using basic sampler operations.
- In parallel, we introduce a calculus for compositionally proving that samplers target desired distributions.

Conclusion

- Our language enables the compositional construction of samplers using basic sampler operations.
- In parallel, we introduce a calculus for compositionally proving that samplers target desired distributions.
- Our semantics is purely deterministic, and so distinguishes between samplers and distributions.

Conclusion

- Our language enables the compositional construction of samplers using basic sampler operations.
- In parallel, we introduce a calculus for compositionally proving that samplers target desired distributions.
- Our semantics is purely deterministic, and so distinguishes between samplers and distributions.
- Because of this, our methods are compatible with pseudorandom number generators as well as ‘truly random’ samplers.