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?

```plaintext
n := 0;
while f(n) == rand()
    n += 1;
```
Motivation: pseudorandom number generation

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

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

1. If \( \text{rand}() \) 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?

```python
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 \( f : \mathbb{N} \to [0, 1] \) does this program terminate?

\[
\begin{align*}
n &:= 0; \\
\text{while } f(n) = \text{rand}() &\rightarrow n += 1;
\end{align*}
\]

1. If \( \text{rand}() \) refers to the uniform distribution rather than a value, then for all \( f \), with probability 1, this program terminates at \( n = 0 \).

2. If \( \text{rand}() \) draws from a sequence of (Martin-Löf) random values, then for all computable \( f \), this program terminates at \( n \geq 0 \).

3. If \( \text{rand}() \) 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 $\Sigma X$ 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 $\Sigma X$ 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 $\Sigma X$ for samplers on $X$.
- Samplers $s : \Sigma X$ are distinct from the distributions $P \in \mathcal{P}X$ that they sample from.
- For example, $\text{flip} = (0, 1, 0, 1, \ldots)$ is a valid sampler for the uniform distribution on $\{0, 1\}$. 
Motivation: compositional sampler verification

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

```plaintext
while true
    a := \texttt{flip}();
    b := \texttt{flip}();
    if a \neq b
        return a;
```

If consecutive samples from \texttt{flip()} are 'independent', then \texttt{a} should be uniform on \{0, 1\}. This 'extractor' really inputs a biased sampler \texttt{flip}, and outputs an unbiased sampler; how can we write it that way?
Motivation: compositional sampler verification

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

```plaintext
while true
    a := flip();
    b := flip();
    if a \neq b
        return a;
```

- If consecutive samples from \texttt{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?

```plaintext
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.
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 $(x_1, x_2, \ldots)$, then $\text{map}(f, s)$ denotes $(f(x_1), f(x_2), \ldots)$. 

Silva, Dahlqvist, Smith (UCL)

Deterministic stream-sampling

LICS '23
Sampler operations

- Many approaches in probability and statistics are \textit{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 $(x_1, x_2, \ldots)$, then $\text{map}(f, s)$ denotes $(f(x_1), f(x_2), \ldots)$.
- Samplers can be \textit{weighted} streams: $\text{reweight}(g, s)$, where $g : X \rightarrow \mathbb{R}_{\geq 0}$, denotes the weighted stream $((x_1, g(x_1)), (x_2, g(x_2)), \ldots)$. 

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 \((x_1, x_2, \ldots)\), then \( \text{map}(f, s) \) denotes \((f(x_1), f(x_2), \ldots)\).

Samplers can be *weighted* streams: \( \text{reweight}(g, s) \), where \( g : X \rightarrow \mathbb{R}_{\geq 0} \), denotes the weighted stream \(((x_1, g(x_1)), (x_2, g(x_2)), \ldots)\).

If \( s : \Sigma X \) denotes \((x_1, x_2, x_3, x_4, \ldots)\), then \( s^2 : \Sigma (X \times X) \) denotes \(((x_1, x_2), (x_3, x_4), \ldots)\).
Samplers ‘target’ distributions

This program implements the unbiasing technique discussed earlier.

```
let accept? = \a, b : B \times B . if a \neq b then 1 else 0 in
let first = \a, b : B \times B . a in
map(first, reweight(accept?, flip²))
```
Samplers ‘target’ distributions

This program implements the unbiasing technique discussed earlier.

To ‘verify’ is to prove: if \( \text{flip}^2 \) generates independent, biased samples on \( B \times B \), then this program generates unbiased samples on \( B \).
Samplers ‘target’ distributions

This program implements the unbiasing technique discussed earlier.

To ‘verify’ is to prove: if \( \text{flip}^2 \) generates independent, biased samples on \( B \times B \), then this program generates unbiased samples on \( B \).

We formalise this using a ‘targeting’ relation: \( s \mapsto P \) 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 \texttt{map}, then to prove targeting, we introduce a \textbf{calculus} with inference rules such as

\[
\Gamma \vdash s : \Sigma S \Rightarrow P \\
\Gamma \vdash f : S \rightarrow T \\
\Gamma \vdash \text{map}(f, s) : \Sigma T \Rightarrow \gamma \mapsto \rightarrow (J f K (\gamma)) ^* P(\gamma)
\]
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

\[
\frac{\Gamma \vdash s : \Sigma S \rightsquigarrow P \quad \Gamma \vdash f : S \rightarrow T}{\Gamma \vdash \text{map}(f, s) : \Sigma T \rightsquigarrow \gamma \mapsto (\lbrack f \rbrack(\gamma)) \ast P(\gamma)}
\]

Caution: this rule does not hold for all (measurable) `f`!

For a sequence \((x_n)_{n \in \mathbb{N}}\), consider `f(x) = \{x_n : n \in \mathbb{N}\}(x)`. (This is related to the first program discussed; see our paper for how we handle this.)
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

\[
\begin{align*}
\Gamma \vdash s : \Sigma S \leadsto P & \quad \Gamma \vdash f : S \rightarrow T \\
\Gamma \vdash \text{map}(f, s) : \Sigma T \leadsto \gamma \mapsto (\llbracket f \rrbracket(\gamma))_* P(\gamma)
\end{align*}
\]

- 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?

\[
\mu \sim N(\alpha_\mu, B_\mu) \\
\Sigma \sim \text{InvWishart}(df, B_\Sigma) \\
w_c \sim N(\alpha_c, \Sigma) \\
W = (w_1^T, \ldots, w_C^T) \\
x \sim N(\mu, \Sigma) \\
y = \sigma(W^T x)
\]

- 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.