Is it possible to compile any Stan program
to a generative probabilistic program?

Stan
- Declarative style
- Very large community

Generative PPLs
- Many instances: WebPPL, Pyro, ...
- General purpose programming language with sample, observe, and factor

Contributions
- Comprehensive compilation scheme
- Correctness proof
- A new Pyro backend for Stanc3
- Extending Stan with explicit variational guides and neural networks

Benefits
- Stan users have access to a new backend with different inference engines and new features
- PPLs developers have access to a large number of models

Generative compilation
- ~ on parameters: sampling
- ~ on data: conditioning
- Cannot handle all Stan models!

Stan features: example, prevalence and compilation

<table>
<thead>
<tr>
<th>Feature</th>
<th>% Example</th>
<th>Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left expression</td>
<td>7.7 sum(phi ~ normal(0, 0.001*N));</td>
<td>observe(Normal(0, 0.001*N), sum(phi))</td>
</tr>
<tr>
<td>Multiple updates</td>
<td>3.9 phi_y ~ normal(0, sigma_py); phi_y ~ normal(0, sigma_pt);</td>
<td>observe(Normal(0, sigma_py), phi_y); observe(Normal(0, sigma_pt), phi_y)</td>
</tr>
<tr>
<td>Implicit prior</td>
<td>60.7 real alpha;/ * missing 'alpha' */</td>
<td>alpha = sample(ImproperUniform())</td>
</tr>
<tr>
<td>Target update</td>
<td>16.3 target == 0.5 * dot_self( phi[node1] - phi[node2]);</td>
<td>factor(~ 0.5 * dot_self( phi[node1] - phi[node2]))</td>
</tr>
</tbody>
</table>

Comprehensive compilation
- All ~ statements are conditioning
- Parameters are initialized with uniform priors

Correctness proof
- The semantics of Stan is based on an extension of [Gorinova et al. 2018]
- The semantics of the generative PPL is based on [Staton 2017]
- The compilation is formalized as a continuation passing style transformation

\[
C(p) = \mathcal{P}(\operatorname{return}(\operatorname{params}(p))) \circ \mathcal{P}_k(\operatorname{params}(p))
\]

\[
\mathcal{P}_k(\operatorname{params}(p)) = \text{let } x_1 = D_1 \text{ in } \cdots \text{let } x_k = D_k \text{ in k}
\]

Proof:
- \[\mathbb{E}[C(p)]_D \propto \lambda U \int U \mathbb{E} \left[ \mathcal{S}(\operatorname{return}(\operatorname{params}(p))) \right]_D, \phi(\{\}) d\theta\]
- \[\mathbb{E}[C(p)]_D \propto \lambda U \int U \mathbb{E} \left[ \exp \left( \operatorname{params}(p) \right)_D, \phi(\{\}) \right]_D d\theta\]
- \[\mathbb{E}[C(p)]_D \propto \lambda U \int U \exp \left( \operatorname{params}(p) \right)_D, \phi(\{\}) d\theta\]

Evaluation
- Compiler implemented as a fork of Stanc3
- Tested based on the 97 Stan models provided by PosteriorDB
- 96 models are compiling (the 1 error also fails to compile with Stan 3)
- Inference runs on 77 models
- Yield distributions similar to Stan on 8 classic models

Extensions: SVI guides and NN

Stochastic Variational Inference (SVI)
- Explicit guides to specify the family of target distributions

Neural Networks
- Neural networks defined in PyTorch
- Deep probabilistic models: models using deep neural networks
- Bayesian Networks: parameters of the network are random variables

References
- Staton, Commutative Semantics for Probabilistic Programming, ESOP 2017.

Example

```r
data { int N; int<lower=0,upper=1> x[N]; } parameters { real<lower=0,upper=1> z; } model { z ~ beta(1, 1); for (i in 1:N) x[i] ~ bernoulli(z); }
```

Experiments were run on a MacBook Pro 6 cores i9 (2.9 GHz, 32 GB RAM). Stan compiles a statement `target += ~0.5 * dot_self(phi[node1] - phi[node2]);`