Soss

CHAD SCHERRER, RelationalAI
TAINE ZHAO, University of Tsukuba

We present Soss, a declarative probabilistic programming language embedded in the Julia language. Soss represents statistical models in terms of abstract syntax trees, and uses staged compilation for on-demand generation of "inference primitives" (random sampling, log-density, etc) without requiring casual users to worry about such details.

The approach taken by Soss makes it easy to extend to take advantage of other packages in the rapidly-growing Julia ecosystem. At the time of this writing, Soss users can choose from several inference back-ends and connect easily with larger systems SymPy and Gen.

1 INTRODUCTION

There are many approaches to building a Probabilistic Programming Language ("PPL"). Soss is distinct from most alternatives in a number of ways:

- Model specification in Soss is *declarative*; statements are represented not by the order entered by the user, but by the partial order given by their variable dependencies.
- Soss models are typically expressed in terms of *inference combinators* (like "`For`") that combine distributions in some way to arrive at a new distribution.
- Soss models are *first-class*; models can be passed as arguments to other models, or used in place of distributions within a model.
- Soss models are *function-like*; abstractly, a model may be considered a function from its arguments to a joint distribution. In particular, specification of "observed variables" is separated from model definition.
- Values and distributions in Soss are represented internally as *abstract syntax trees*, keeping the internal representation close to that given by the user for maximum flexibility.
- Soss uses *staged compilation* for inference primitives like `rand` and `logpdf`, via novel *generalized generated functions* from `GeneralizedGenerated.jl`.
- Soss supports *model transformations*, functions that take a model and return another model.
- Soss supports *symbolic simplifcation*, making it easy to inspect or manipulate a symbolic representation of the log-density, or to use it to generate optimized code.
- Soss is *extensible*; users can define new inference primitives or model transformations externally, and use them as if they had been included in Soss.

Finally, Soss uses Cockney rhyming slang to address the well-known challenge of naming things in computer science ("sauce pan" rhymes with "Stan"). **S**oss is **o**pen-**s**ource **s**oftware.

2 MODELING IN SOSS

Figure 1 shows a plate diagram and Soss implementation of a simplified two-way ANOVA model. The Soss model m has *arguments* I and J. Each line of a model is either a *sample statement* (`v ~ rhs`) or an *assign statement* (`v = rhs`). In either case, `v` must be a valid Julia variable name, and `rhs` must be a valid Julia expression. For a sample statement, `rhs` should also be "distribution-like", supporting `logpdf` and/or `rand` methods (depending on the inference algorithm to be used).

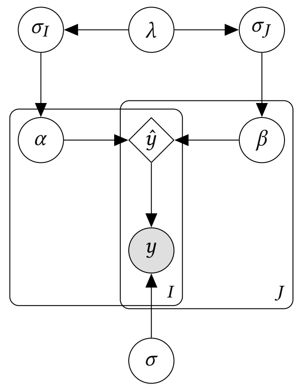
2.1 Joint Distributions

A Soss model is *function-like*, taking values for its arguments and returning a *joint distribution*.

Authors' addresses: Chad Scherrer, RelationalAI, Seattle, WA, chad.scherrer@gmail.com; Taine Zhao, University of Tsukuba, Department of Computer Science.

$$
\begin{aligned}
i &\in \{1, \cdots, I\} \\
j &\in \{1, \cdots, J\} \\
\lambda &\sim \mathrm{Cauchy}_+(0, 1) \\
\sigma_I &\sim \mathrm{Normal}_+(0, \lambda) \\
\sigma_J &\sim \mathrm{Normal}_+(0, \lambda) \\
\alpha_i &\sim \mathrm{Normal}(0, \sigma_I) \\
\beta_j &\sim \mathrm{Normal}(0, \sigma_J) \\
\hat{y}_{ij} &= \alpha_i + \beta_j \\
y_{ij} &\sim \mathrm{Normal}(\hat{y}_{ij}, \sigma)
\end{aligned}
$$

(a)



(b)

```
m = @model I,J begin
    λ ~ HalfCauchy()
    σI ~ HalfNormal(λ)
    σJ ~ HalfNormal(λ)

    α ~ Normal(0, σI) |> iid(I)
    β ~ Normal(0, σJ) |> iid(J)
    yhat = α .+ β'

    σ ~ HalfNormal()
    y ~ For(I,J) do i,j
            Normal(yhat[i,j], σ)
        end
    end
```

(c)

Fig. 1. A generative model (a), plate diagram (b), and Soss model (c) for a two-way analysis of variance model. Note the specification that y will later be observed is not part of the model, but is instead given at inference time.

2.2 Inference Primitives

A given inference algorithm can be expressed in terms of a collection of functions on a model. For example, we may need to evaluate the log-density or its gradient, map continuous variables into $\mathbb{R}^n$, or draw random samples. In Soss, these *inference primitives* generate model-specific code at execution time.

At this writing, the following inference primitive are available:

citation for MonteCarloMeasurements.jl

`rand` draws a random sample from a given joint distribution.
`particles` draws a *systematic sample* from a given joint distribution, using `MonteCarloMeasurements.jl`.
`logpdf` takes a joint distribution and a point in its sample space, and returns the log-density.
`symlogpdf`
`weightedSample`
`xform`

2.3 Model Transformations

Conditional predictive models.

Causal interventions ("Do" operator).

Markov blankets.

```
julia> markovBlanket(m, :α)
    @model (I, σI, β, σ, J) begin
            α ~ Normal(0, σI) |> iid(I)
            yhat = α .+ β'
            y ~ For(I, J) do i, j
                    Normal(yhat[i, j], σ)
                end
        end
```

2.4 Symbolic Manipulation

Symbolic Log-density.

Code Generation.

3 IMPLEMENTATION

```
struct Model{A,B,M}
    args  :: Vector{Symbol}
    vals  :: NamedTuple
    dists :: NamedTuple
    retn  :: Union{Nothing, Symbol, Expr}
end
```

4 PERFORMANCE

Let's implement models from https://statisticalrethinkingjulia.github.io/MCMCBenchmarks.jl/latest/benchmarks/ for benchmark comparisons

Can Soss really generate arbitrary code?

The ability to generate arbitrary Julia code makes it difficult to measure performance in Soss. When a performance bottleneck is found, developers or users can ask Soss to generate something different. This design places no a priori constraints on performance.

Restructure my writing?

Besides the purely static code generation via regular macros happening in parsing time, Soss heavily uses a mechanism of "zero-cost" runtime code generation originated by Julia's generated functions [Bezanson et al. 2012], a.k.a staged functions in the original paper and earlier versions of Julia Programming Language.

The generated functions provide the capability of performing code generation during type inference, generating programs computed by the body of function(a.k.a, generator), once and only once for each combination of argument types.

Further, Julia enables type inference and compiler optimizations equivalent to non-runtime ones in runtime for the callsites of generated functions, hence we gain runtime code generation without losing performance.

To take advantage of this, Soss designs a system to encode sufficient information for generating actual codes, into the types, or objects that can be dispatched like types, which allows the use of generated functions here.

Unfortunately, in current stage, there're some implementation restrictions to Julia's generated functions, and generated functions cannot generate arbitrary Julia codes, where some advanced programming constructions are missing, such as generators and function-related stuffs like closures(including functions with no free variables), multiple dispatch, etc.

To address this, we introduced the works of easing the restrictions of Julia generated functions,

cite GG here?

which allow us to generate closures from generated functions, and make the programs sufficiently powerful.

Programs that can be generated by Soss are turing complete, as all constructs of Lambda Calculus can be generated.

```
abstract type LCTerm end
struct Lam{Arg, Body <: LCTerm} <: LCTerm end
struct App{Fn<:LCTerm, Arg<:LCTerm} <: LCTerm end
```

```
struct Var{N} <: LCTerm end

cg(::Type{Lam{Arg, Term}}) where {Arg, Term} =
  :($Arg -> $(cg(Term)))
cg(::Type{App{Fn, Arg}}) where {Fn, Arg} =
  :($(cg(Fn))($(cg(Arg))))
cg(::Type{Var{N}}) where N = N

function tolc(expr)
  @match expr begin
    :($f($arg)) => App{tolc(f), tolc(arg)}
    :($x => $y) => Lam{x, tolc(y)}
    a::Symbol => Var{a}
  end
end

@gg function lc(::Type{Term}) where Term <: LCTerm
    cg(Term)
end

macro lc(term)
  lc(tolc(term))
end

x = Core.Box()
y = Core.Box()
z = Core.Box()
f(x) = (x, z)
@assert (@lc x => x)(x) === (@lc y => y)(x)
@assert (@lc f => x => f(x))(f)(y) === (y, z)
```

5 EXTENDING SOSS

6 RELATED WORK

The idea of code generation and symbolic simplification in an embedded PPL goes back to *Passage* [Scherrer et al. 2012].

Hakaru [Narayanan et al. 2016] takes a more ambitious symbolic approach in a stand-alone PPL, allowing a wider variety of program transformations.

Soss began with a goal of representing models with continuous, fixed-dimensionality parameter spaces, inspired by *Stan* [Carpenter et al. 2017].

Gen [Cusumano-Towner et al. 2019] was developed independently from Soss, but takes a similar approach (and distinct from most PPLs) in its representation of a model as a function from its arguments to a "trace" (to use Gen's terminology). In both Soss and Gen's static DSL, this trace is a mapping from variable names to values. The similarity of these systems makes interoperability relatively straightforward, as demonstrated in *SossGen* (https://github.com/cscherrer/SossGen.jl).

Turing [Ge et al. 2018]

ACKNOWLEDGMENTS

We would like to acknowledge...

REFERENCES

Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. arXiv:cs.PL/1209.5145

$$
\begin{aligned}
\langle\text{model}\rangle &\models \texttt{@model}\ \langle\text{args}\rangle\ \texttt{begin}\ \langle\text{statements}\rangle\ \texttt{end}\\
\langle\text{args}\rangle &\models \lambda \mid \langle\text{Symbol}\rangle \mid \langle\text{Symbol}\rangle\texttt{,}\langle\text{args}\rangle\\
\langle\text{statements}\rangle &\models \langle\text{statement}\rangle \mid \langle\text{statement}\rangle\ \textit{Julia Line Sep}\ \langle\text{statements}\rangle \mid \langle\text{statements}\rangle\,\langle\text{retn}\rangle\\
\langle\text{statement}\rangle &\models \langle\text{assign}\rangle \mid \langle\text{sample}\rangle\\
\langle\text{assign}\rangle &\models \langle\text{Symbol}\rangle = \langle\text{Expr}\rangle\\
\langle\text{sample}\rangle &\models \langle\text{Symbol}\rangle \sim \langle\text{Measure}\rangle\\
\langle\text{retn}\rangle &\models \texttt{return}\ \langle\text{Expr}\rangle\\
\langle\text{Symbol}\rangle &\models \textit{Julia Symbol}\\
\langle\text{Expr}\rangle &\models \textit{Julia Expr}\\
\langle\text{Measure}\rangle &\models \textit{Probability measure (see text)}
\end{aligned}
$$

Fig. 2. Backus-Naur Form representation for a user-specified model.

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017).

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-purpose Probabilistic Programming System with Programmable Inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 221–236. https://doi.org/10.1145/3314221.3314642

Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*. 1682–1690. http://proceedings.mlr.press/v84/ge18b.html

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*. Springer, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5

Chad Scherrer, Iavor Diatchki, Levent Erkök, and Matthew Sottile. 2012. Passage : A Parallel Sampler Generator for Hierarchical Bayesian Modeling. In *Neural Information Processing Systems workshop on Probabilistic Programming*. 1–4.

A MODEL SYNTAX

B SYMBOLIC SIMPLIFICATION

Before simplification:

After simplification:

C EXTENDED EXAMPLE

Break this up! This is just a staging area

```
julia> using Revise, Soss, Random

julia> Random.seed!(1);

julia> m = @model I,J begin
         λ ~ HalfCauchy()
         σI ~ HalfNormal(λ)
         σJ ~ HalfNormal(λ)
         α ~ Normal(0, σI) |> iid(I)
         β ~ Normal(0, σJ) |> iid(J)
         yhat = α .+ β'
```

$$
\begin{aligned}
\ell =& -3\log(2) - 5\log(\pi) - \sigma^2 \\
& - 4\log(\lambda) - 2\log\left(\lambda^2+1\right) - \frac{\sigma_I^2}{\lambda^2} - \frac{\sigma_J^2}{\lambda^2} \\
& + \sum_{i=1}^{I}\left(-\log(\sigma_I) - \frac{\log\pi}{2} - \frac{\log 2}{2} - \frac{\alpha_i^2}{2\sigma_I^2}\right) \\
& + \sum_{j=1}^{J}\left(-\log(\sigma_J) - \frac{\log\pi}{2} - \frac{\log 2}{2} - \frac{\beta_j^2}{2\sigma_J^2}\right) \\
& + \sum_{\substack{1\le i\le I\\1\le j\le J}}\left(-\log(\sigma) - \frac{\log\pi}{2} - \frac{\log 2}{2} - \frac{\left(y_{ij}-\hat{y}_{ij}\right)^2}{2\sigma^2}\right)
\end{aligned}
$$

(a) Before

$$
\begin{aligned}
\ell =& -7.8 - 0.92I - 0.92J - 0.92IJ - \sigma^2 \\
& - 4\log\lambda - 2\log\left(\lambda^2+1\right) - IJ\log\sigma \\
& - I\log\sigma_I - J\log\sigma_J - \frac{\sigma_I^2}{\lambda^2} - \frac{\sigma_J^2}{\lambda^2} \\
& - \frac{1}{2\sigma_I^2}\sum_{i=1}^{I}\alpha_i^2 - \frac{1}{2\sigma_J^2}\sum_{j=1}^{J}\beta_j^2 \\
& - \frac{1}{2\sigma^2}\sum_{\substack{1\le i\le I\\1\le j\le J}}\left(y_{ij}-\hat{y}_{ij}\right)^2
\end{aligned}
$$

(b) After

Fig. 3. Symbolic log-density of m before (a) and after (b) simplification steps. Reduction

```
        σ ~ HalfNormal()
        y ~ For(I,J) do i,j
                Normal(yhat[i,j], σ)
            end
    end;

julia> truth = rand(m(I=2,J=3)); pairs(truth);

julia> post = dynamicHMC(m(I=2,J=3), (y=truth.y,));

julia> postpred = predict(m(I=2,J=3), post) |> particles;

julia> postpred.yhat
2×3 Array{Particles{Float64,1000},2}:
 0.241 ± 0.26  0.527 ± 0.3   -0.556 ± 0.38
 0.22 ± 0.26   0.506 ± 0.32  -0.577 ± 0.37

julia> postpred.y
2×3 Array{Particles{Float64,1000},2}:
 0.224 ± 0.58  0.529 ± 0.58  -0.58 ± 0.64
 0.223 ± 0.56  0.483 ± 0.58  -0.58 ± 0.61

julia> m.args
2-element Array{Symbol,1}:
 :I
 :J

julia> pairs(m.vals)
pairs(::NamedTuple) with 1 entry:
  :yhat => :(α .+ β')

julia> pairs(m.dists)
pairs(::NamedTuple) with 7 entries:
  :λ  => :(HalfCauchy())
  :σI => :(HalfNormal(λ))
```

```
:σJ => :(HalfNormal(λ))
:α  => :(Normal(0, σI) |> iid(I))
:β  => :(Normal(0, σJ) |> iid(J))
:σ  => :(HalfNormal())
:y  => :(For(I, J) do i, j. . .
```