Genify.jl: Transforming Julia into Gen to enable programmable inference

Tan Zhi-Xuan
Massachusetts Institute of Technology
xuan@mit.edu

McCoy R. Becker
Charles River Analytics
mbecker@cra.com

Vikash K. Mansinghka
Massachusetts Institute of Technology
vkm@mit.edu

1 Introduction

There exists a wide variety of stochastic simulators written in Julia for the purposes of modeling natural, social, or economic phenomena [1–6]. However, these simulators are not generally amenable to efficient algorithms for Bayesian inference, as they do not provide likelihoods for execution traces, support the ability to constrain internal random variables, or allow random choices and subroutines to be selectively updated in Monte Carlo algorithms. As a result, inference is either limited to black-box methods such as approximate Bayesian computation [7], or implemented from scratch, which can be intensive and error-prone.

To address these limitations, we present *Genify.jl*, an approach to transforming Julia code into generative functions in Gen, a probabilistic programming platform implemented in Julia [8]. We accomplish this via staged compilation of lowered Julia code into Gen's dynamic modeling language, combined with a user-friendly random variable addressing scheme that enables straightforward implementation of custom inference programs. Unlike prior approaches that make existing simulators controllable by probabilistic programming systems [9–12], our approach is designed to support *programmable inference* [8, 13]. This allows users to rapidly implement and iterate upon inference algorithms that are customized to simulators. We demonstrate the utility of this approach by transforming and performing parameter estimation over an agent-based epidemic model implemented in the *Agents.jl* framework [4]. We use Gen to implement generic and custom sequential Monte Carlo (SMC) and Markov chain Monte Carlo (MCMC) algorithms for this simulator, showing that custom inference improves performance significantly.

Code	**Address**
`function step!(model, agent_step!)`	`:step!`
`    >> i = 0`	
`    for agent in values(model.agents)`	
`        >> i += 1`	
`        agent_step!(model.agents[index], model)`	`:agent_step! => i`
`    end`	
`end`	
`function agent_step!(agent, model)`	`:agent_step! => i`
`    migrate!(agent, model)`	`:migrate!`
`    transmit!(agent, model)`	`:transmit!`
`    update!(agent, model)`	`:update!`
`    recover!(agent, model)`	`:recover!`
`end`	
`function migrate!(agent, model)`	`:migrate!`
`    p = agent.pos`	
`    d = Categorical(model.migration_rates[p, :])`	
`    m = rand(d) ->> m ~ categorical(...)`	`:m`
`    if (m != p) move_agent!(agent, m, model) end`	
`end`	

Figure 1. Julia code excerpt of an *Agents.jl* SIR model of a viral epidemic. *Genify.jl* transforms the top-level function `step!` into a Gen model by: (i) recursively transforming all subroutines, (ii) forwarding (-») random primitives to the appropriate Gen primitives, (iii) inserting (») loop counters, and (iv) automatically generating address names. Each function is given a address namespace (violet), within which the addresses of subroutines and primitives are nested (magenta).

2 Code transformation

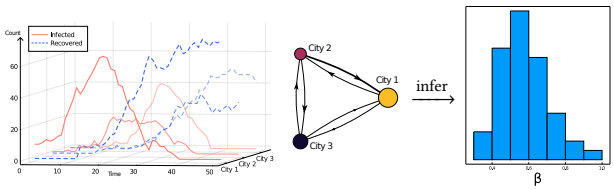
Our approach to code transformation exploits the multi-stage programming and reflection features supported by Julia. Given a function `f` and the types of its arguments, one can introspect the body of the function before type inference to acquire a lowered representation of the corresponding method. This method representation is in static single assignment (SSA IR) form. We apply the automatic addressing transform by walking the IR and replacing all calls to random primitives (`rand`, `randn`, etc. in Julia) with calls to primitive distributions in Gen (`uniform`, `normal`, etc.), annotated with automatically generated address names. The transformation is applied recursively, by acquiring a SSA representation of all non-primitive calls in the method body and repeating the process. Figure 1 shows an example of this process. Transformation occurs during just-in-time (JIT) compilation using a staging mechanism in Julia called *generated functions*. JIT compilation ensures that performance overhead is minimal, but also provides many of the benefits of non-standard interpretation at run-time. In particular, if the original method is modified during development, the transformed method is automatically recompiled upon its next execution.

3 Automatic addressing

To facilitate programmable inference, random variables require user-friendly addresses. Automatic addressing should thus strive for consistency, readability, and correspondence with the source code so that a programmer familiar with the original code can deduce the address that corresponds to a given stochastic routine. We achieve this via the following scheme: following the hierarchical address format used by Gen [8], a subroutine `g` of `f` gets a nested address namespace `:f => :g`. Where possible, we set the address of a random

```
gen_step! = genify(step!, AgentBasedModel, Any)
@gen function bayesian_sir(T::Int, noise::Float64)
    β ~ uniform_continuous(0, 2.0)
    model = model_initiation(β, city_populations=[50, 50, 50])
    for t in 1:T
        {:step => t=> :agents} ~ gen_step!(model, agent_step!)
        {:step => t => :obs} ~ observe(model, noise)
    end
end
@gen function observe(model::AgentBasedModel, noise::Float64)
    for city in nodes(model)
        {:infected => city} ~ normal(n_infected(model, city), noise)
        {:recovered => city} ~ normal(n_recovered(model, city), noise)
    end
    for a in agents(model)
        {:location => a.id} ~ noisy_delta(a.pos, n_cities(model), 0.01)
    end
end
```

(a) Bayesian agent-based SIR model



(b) Observations, model topology, and inferences

Algorithm	$\hat{\beta}$ (Mean ± Std.)	$\hat{\beta}$ RMSE	Log Prob.	Time (s)
Resimulation MH	1.17 ± 0.64	0.90	-22552	120
Block MH	0.75 ± 0.55	0.58	-21754	270
Basic SMC	0.69 ± 0.23	0.29	-19986	**44**
Data-driven SMC	**0.51 ± 0.10**	**0.09**	**-5164**	59

(c) Inference results ($\beta_{\text{true}} = 0.5$, n_{iters} or $n_{\text{particles}} = 100$)

```
function block_mh(T, observations, n_iters, noise, blocklen)
    trace, _ = generate(bayesian_sir, (T, noise), observations)
    # Block addresses by time and process (migration / transmission)
    block_addrs = [select([:step=>t=>:agents=>:step!=>:agent_step!=>i=>fn
                           for t in block for i in agents(model)]...)
                   for block in partition(1:T, blocklen)
                   for fn in (:migrate!, transmit!)]
    # Alternate between resimulation of parameters and blocks
    for i in 2:n_iters
        trace, _ = metropolis_hastings(trace, select(:β))
        for block in block_addrs
            trace, _ = metropolis_hastings(trace, block)
        end
    end
    return trace
end
```

(d) Block MH with time and sub-routine blocking

```
function data_driven_smc(T, observations, n_particles, noise)
    # Initialize particle filter
    state = pf_initialize(bayesian_sir, (0, noise),
                          choicemap(), n_particles)
    for t=1:T
        # Extend the filter by one step using a custom migration proposal
        pf_update!(state, (t, noise), (UnknownChange(), NoChange()),
                   observations[t], migrate_proposal, (t, observations))
    end
    return state
end

@gen function migrate_proposal(prev_trace, t, observations)
    # Use observed locations to propose to true locations for each agent
    for i in 1:n_agents(prev_trace)
        loc = observations[:step => t => :obs => :location => i]
        {:step=>t=>:agents=>:step!=>:agent_step!=>i=>:migrate!=>:m} ~
            noisy_delta(loc, n_cities(prev_trace), 0.01)
    end
end
```

(e) Data-driven SMC with migration proposals

Figure 2. Modeling and inference over an inference-unaware SIR model. **(a)** We transform the *Agents.jl* `step!` function *(a1)* to construct a hierarchical Gen model with parameter uncertainty *(a3)* and observation noise *(a10)*. **(b)** Given case counts, agent locations (not shown) and model structure, we infer the infection rate β. Using custom MCMC **(d)** and SMC **(e)** programs that resimulate *(d4)* or propose *(e17)* internal randomness, we achieve **(c)** better results than generic PPL algorithms.

call to the name of the variable it is assigned to (e.g. `z = randn()` gets the name `:z`). Otherwise, we use the name of the called function. For repeated names, we append indices to ensure uniqueness. Finally, for variables sampled within loops, we insert loop counters into the IR and append the counters to the address of each in-loop random variable.

4 Experiments

To demonstrate the utility of programmable inference for Julia simulators, we transform an agent-based Susceptible-Infected-Recovered (SIR) model of virus spread written in the *Agents.jl* framework [4]. We then use this simulator in a Gen model with parameter uncertainty and observation noise (Figure 2a). Our inference task is as follows: given infection and recovery counts in three connected cities with 50 agents each (Figure 2b), as well as location data for 75% of the agents (e.g. from opt-in contact tracing), infer the infection rate β.

We implement four inference algorithms using Gen to solve this task. Two are generic[1]: (1) Cascading resimulation Metropolis-Hastings (MH) [14], which can be used with likelihood-free simulators, and (2) 'basic' SMC with proposals from the prior (no resampling). Two are custom, requiring transformation into Gen to manipulate internal random variables: (3) block resimulation MH [15], where variables internal to the simulator are blocked by both time and process (migration or transmission), and (4) data-driven SMC with custom migration proposals. These proposals are based on the observed locations of agents when available. In Figure 2c, we demonstrate that our custom algorithms out-perform the corresponding generic algorithms in root mean squared error (RMSE) for a fixed number of 100 iterations or particles, albeit with mildly higher runtime cost. Our data-driven SMC algorithm appears particularly efficient, recovering the true value $\beta = 0.5$ with little overhead relative to basic SMC.

[1]We also implemented single-site MH as a generic algorithm [9], but found that even a single iteration over all 15001 latent variables led to minimal convergence while running 16–100 times slower than the other algorithms.

5 Future Work

We plan to extend this work in several promising directions: compilation into Gen's static modeling language and generative function combinators, enabling incremental computation by identifying variable dependencies, program analysis to determine when random variables should share the same addresses, and user-friendly address exploration via annotated source-code visualizations.

6 Acknowledgements

We thank Alex Lew and Marco Cusumano-Towner for their constructive suggestions and advice on program transformation into probabilistic programming systems.

References

[1] Jesse Perla, Thomas Sargent, and John Stachurski. *Quantitative economics with Julia*. QuantEcon, 2015.

[2] Maxim Egorov, Zachary N. Sunberg, Edward Balaban, Tim A. Wheeler, Jayesh K. Gupta, and Mykel J. Kochenderfer. POMDPs.jl: A framework for sequential decision making under uncertainty. *Journal of Machine Learning Research*, 18(26):1–5, 2017.

[3] Alfonso Landeros, Timothy Stutz, Kevin L Keys, Alexander Alekseyenko, Janet S Sinsheimer, Kenneth Lange, and Mary E Sehl. Biosimulator. jl: Stochastic simulation in julia. *Computer methods and programs in biomedicine*, 167:23–35, 2018.

[4] Ali R Vahdati. Agents. jl: agent-based modeling framework in julia. *Journal of Open Source Software*, 4(42):1611, 2019.

[5] Justin Angevaare, Zeny Feng, and Rob Deardon. Pathogen. jl: Infectious disease transmission network modelling with julia. *arXiv preprint arXiv:2002.05850*, 2020.

[6] Jeffrey Regier, Andrew C Miller, David Schlegel, Ryan P Adams, Jon D McAuliffe, et al. Approximate inference for constructing astronomical catalogs from images. *The Annals of Applied Statistics*, 13(3):1884–1926, 2019.

[7] Mark A Beaumont, Wenyang Zhang, and David J Balding. Approximate bayesian computation in population genetics. *Genetics*, 162(4):2025–2035, 2002.

[8] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236, 2019.

[9] David Wingate, Andreas Stuhlmüller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.

[10] Bradley Gram-Hansen, Christian Schröder de Witt, Tom Rainforth, Philip HS Torr, Yee Whye Teh, and Atılım Güneş Baydin. Hijacking malaria simulators with probabilistic programming. *arXiv preprint arXiv:1905.12432*, 2019.

[11] Atilim Gunes Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Saeid Naderiparizi, Andreas Munk, Jialin Liu, Bradley Gram-Hansen, Gilles Louppe, Lawrence Meadows, et al. Efficient probabilistic inference in the quest for physics beyond the standard model. In *Advances in neural information processing systems*, pages 5459–5472, 2019.

[12] Frank Wood, Andrew Warrington, Saeid Naderiparizi, Christian Weilbach, Vaden Masrani, William Harvey, Adam Scibior, Boyan Beronov, and Ali Nasseri. Planning as inference in epidemiological models. *arXiv preprint arXiv:2003.13221*, 2020.

[13] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

[14] Marco F Cusumano-Towner, Alexey Radul, David Wingate, and Vikash K Mansinghka. Probabilistic programs for inferring the goals of autonomous agents. *arXiv preprint arXiv:1704.04977*, 2017.

[15] Daniel J Sargent, James S Hodges, and Bradley P Carlin. Structured Markov chain Monte Carlo. *Journal of Computational and Graphical Statistics*, 9(2):217–234, 2000.