Impulse: Momentously Fast, General, and Portable Probabilistic Programming via Compiler Augmentation

Siyuan Brant Qian*
University of Illinois Urbana-Champaign
USA
siyuanq4@illinois.edu

Vimarsh Sathia
University of Illinois Urbana-Champaign
USA
vsathia2@illinois.edu

Jesse Michel
Massachusetts Institute of Technology
USA
jmmichel@csail.mit.edu

William S. Moses
University of Illinois Urbana-Champaign and Google
USA
wsmoses@illinois.edu

Abstract

Probabilistic programming frameworks automate Bayesian inference, but their performance is limited by a structural mismatch: the code a scientist writes to naturally express a model is often not the code an efficient sampler should execute. In practice, inference is often dominated by expensive computations entirely determined by fixed data and model structure, not by the sampled parameters. Because this structure is not explicit to a general-purpose compiler, the inference computation appears parameter-dependent and is therefore redundantly repeated across iterations.

To eliminate this redundancy, we present Impulse, a probabilistic programming system built on MLIR that preserves probabilistic semantics as first-class compiler primitives. We introduce *Sample-Invariant Code Motion* (SICM), a compiler pass that performs dataflow analysis to understand operation dependencies on sampled parameters, applies algebraic rewrites to factor out invariant computation, and hoists the invariant computation out of the inference process. On standard benchmarks, Impulse achieves a geomean speedup of 1.7× over NumPyro, 6.4× over Stan, and 53× over Turing. SICM enables speedups from 8.5× to 190× on models with sample-invariant structure.

1 Problem and Motivation

Probabilistic programming languages (PPLs) [Abril-Pla et al. 2023; Bingham et al. 2019; Carpenter et al. 2017; Cusumano-Towner et al. 2019; Ge et al. 2018; Phan et al. 2019] democratize Bayesian inference by allowing scientists to express models as generative programs. However, the code a scientist writes to naturally express a model is often not the code a sampler should execute. Figure 1 illustrates this mismatch: the scientist directly translates a physics equation into a probabilistic program, but solve($k \cdot K, q$) couples the sampled parameter k with a fixed matrix K, resulting in an $\Theta(n^3)$ solve that repeats every iteration.

The Problem. Inference computations are typically dominated by expensive operations (e.g., linear solves, matrix factorizations, forward simulations) whose cost is determined by fixed data and model structure, not the sampled parameters. However, after lowering to inference and differentiation, the algebraic structure needed for optimization is no longer visible to the compiler.

*Role of the participating student: lead the design of Impulse and perform all implementation and evaluation.

The Solution. These computations can often be decomposed into *sample-invariant* factors (determined by fixed data) and *sample-dependent* factors that vary across iterations. Hoisting the invariant factors outside the inference loop eliminates the redundant work. However, existing systems place the burden of recognizing and applying such rewrites on the user (e.g., Stan's `transformed data` block [Carpenter et al. 2017]) as their compilers have no visibility of the probabilistic structure before lowering. Performing the same factorization after lowering is intractable, since the inference loop is interleaved with gradient computation.

The Challenge. Automating this factorization requires algebraic reasoning *before* lowering of probabilistic constructs obscures the model's structure. Existing PPLs resolve probabilistic semantics before compilation (Figure 2): in NumPyro [Phan et al. 2019], effect handlers intercept `sample` statements at the Python level before JAX traces the program [Bradbury et al. 2018]; what reaches the compiler is a numerical program with no notion of which operations depend on the sampled parameters. Other models may require counterintuitive restructuring that *adds* operations to the forward model to eliminate redundancy in primal and adjoint (§3.2).

Our approach. Our core insight is that a compiler already provides the machinery to automate this: dataflow analysis [Kildall 1973] to classify operations by their dependence on sampled parameters, rewrite rules to decompose operations into sample-invariant and sample-dependent factors, and code motion to hoist invariant factors out of loops. We propose deferring the lowering of probabilistic constructs to happen *within* the compiler, preserving sample sites and inference as first-class compiler primitives. This creates a new optimization surface (Figure 2): standard passes such as loop-invariant code motion (LICM) and common subexpression elimination (CSE) operate before probabilistic structure is obscured, and new probabilistic-specific passes can perform optimizations enabled by the probabilistic semantics. We present Impulse, a language-agnostic probabilistic programming system built on MLIR [Lattner et al. 2021], and introduce *Sample-Invariant Code Motion* (SICM), a compiler pass that automatically decomposes and hoists sample-invariant computation out of the inference loop.

2 Background and Related Work

Probabilistic programming and inference. PPLs [Abril-Pla et al. 2023; Bingham et al. 2019; Carpenter et al. 2017; Ge et al. 2018; Phan et al. 2019] allow scientists to express generative models and

(a) Model

```
model {
  k ~ LogNormal(0, 1)
  T = solve(k * K, q)
  y ~ Normal(T, σ)
}

nuts = NUTS(num_samples=N)
trace = mcmc(model, nuts)
```

(b) Naively lowered, $\Theta(n^3)$ per iteration

```
def log_density(k):
  w  = dist.LogNormal(0, 1).logpdf(k)
  T  = linalg.solve(k * K, q)
  w += dist.Normal(T, σ).logpdf(y_obs)
  return w
for i in range(N):
  g = grad(log_density)(k)
  k = leapfrog(k, g, step_size)
```

(c) Impulse, $\Theta(n)$ per iteration

```
%v = impulse.solve(%K, %q)
%trace = impulse.infer_region
    <{alg = "NUTS", num_samples = N}> {
  %k = impulse.sample @LogNormal(0, 1)
  %T = arith.divf %v, %k
  %y = impulse.sample @Normal(%T, σ)
}
```

Figure 1: A scientist has collected measurements of a quantity y, which they know to be governed by the equation $y = (k \cdot K)^{-1}q$, though subject to (Gaussian) noise. Given a fixed $n \times n$ matrix K and length n vector q, they want to find the most likely setting of the scalar k, given their observations. (a) The scientist directly translates their equation, computing the intermediate matrix $k \cdot K$, and writing an explicit `solve`($k \cdot K$, q) to compute the inverse. The `solve` call runs in $\Theta(n^3)$. (b) Existing probabilistic frameworks lower the NUTS sampler to a series of gradient descent steps over the scalar parameter k. These systems cannot hoist the solve outside of loop because of the gradient call and the fact that its first argument depends on k. As a result, the per-iteration cost remains $\Theta(n^3)$. (c) Impulse performs a dataflow analysis on the parameters of the sample, and rewrites the solve to be independent of k, enabling the solve to be hoisted out of the inference loop, reducing the per-iteration cost to $\Theta(n)$.

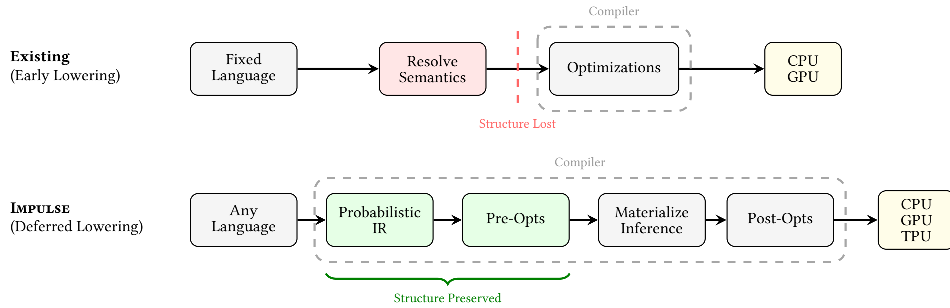

Figure 2: Compilation pipelines compared. Existing frameworks resolve probabilistic semantics before the compiler, losing the structure needed for optimizations. Impulse defers this lowering into the compiler, where both standard (LICM, CSE) and probabilistic (SICM) optimizations exploit the preserved structure before inference is materialized.

automate Bayesian inference. The dominant inference method for continuous models is the No-U-Turn Sampler (NUTS) [Hoffman and Gelman 2014], which extends Hamiltonian Monte Carlo [Neal 2011] by adaptively growing a trajectory until a U-turn criterion is met. Each leapfrog step in NUTS requires evaluating the gradient of the log-density via reverse-mode automatic differentiation (AD) [Griewank and Walther 2008].

Existing systems. Stan [Carpenter et al. 2017] compiles models to C++ but provides no high-level intermediate representation (IR); users must manually precompute invariant quantities in a `transformed data` block. NumPyro [Phan et al. 2019] resolves probabilistic semantics at the Python level before JAX compilation, so what reaches XLA is a flat tensor computation with no notion of sample dependence. Gen [Cusumano-Towner et al. 2019] and Turing [Ge et al. 2018] interpret their inference loops at the host-language level. All these systems erase probabilistic structure before the compiler runs, making the optimizations that Impulse enables (§3.2) infeasible.

Integration/Sampling as language primitives. Prior works have shown that introducing high-level compiler primitives improves performance and expressiveness [Achour and Rinard 2020; Obermeyer et al. 2019; Sherman et al. 2021]. For instance, a line of work in computer graphics [Li et al. 2018; Liu et al. 2019; Loubet et al. 2019] relies on the language having explicitly specified integration/sampling to enable advanced Monte Carlo algorithms [Veach 1998] and new automatic differentiation algorithms for programs involving integration [Bangaru et al. 2021; Lee et al. 2018; Michel et al. 2025, 2024]. Impulse is complementary: such algorithms could be implemented as additional inference strategies and benefit from pre-lowering optimization.

3 Approach and Uniqueness

Impulse compiles probabilistic programs through a multi-stage pipeline that consists of a PPL front end, IR design (§3.1), pre-lowering optimizations (§3.2), and materialization and backend lowering (§3.3). In this work, we implement our front end in Reactant.jl [Moses et al. 2024].

3.1 Probabilistic IR Design

The `impulse` MLIR dialect defines operations that capture the core abstractions of probabilistic programming (Figure 1c shows several in use). The fundamental primitive is the `sample` op, which draws a value from a distribution or invokes a generative sub-model, carrying a unique symbol and an optional log-density function reference. The `simulate` op forward-samples a generative function and returns an execution trace (a flat tensor of all sampled values with compile-time-known offsets) together with the joint

log-density. The `generate` op extends `simulate` with a tensor of observed values, i.e., at constrained sample sites it reads from the tensor instead of drawing fresh samples. The `infer` op declares a probabilistic inference task with algorithm-specific configuration (e.g., NUTS parameters), leaving the materialization to the probabilistic materialization pass (§3.3); the interface is algorithm-agnostic and also accepts a user-supplied `logpdf_fn` for custom log-density functions [Cabezas et al. 2024]. The modeling interface follows the design of Gen [Cusumano-Towner et al. 2019], but Impulse replaces dictionary-based traces with tensor representations. The IR is language-agnostic: any front end that emits `impulse` IR can reuse the entire compilation pipeline.

3.2 Probabilistic-Specific Optimizations

Inlining and region construction. The compilation pipeline inlines the model body into an `infer_region`, converting each `sample` op into a `sample_region` op and recursively dissolving sub-model calls until all sample sites are primitive distributions. It then constructs a *unified logpdf region* that merges all log-density contributions into a single region; everything captured from the enclosing scope is sample-invariant by construction.

The `infer_region` boundary tells the compiler that its body is repeatedly evaluated, so rewrites that increase forward computation may still pay off if they enable hoisting.

Sample-Invariant Code Motion (SICM). SICM automatically eliminates redundant deterministic computation from the inference loop. Unlike standard loop-invariant code motion, which only hoists expressions that are already loop-invariant, SICM creates invariant expressions that do not exist in the original program. SICM uses a forward dataflow analysis [Kildall 1973] (*sample-dependence analysis*) to classify every operation as sample-dependent or sample-invariant, and hoists invariant computation out of the `infer_region`. Figure 1c illustrates that the `impulse.solve` on line 1 was hoisted after SICM rewrote $\texttt{solve}(k \cdot K, q)$ as $\texttt{solve}(K, q)/k$ (the `divf` on line 5), making the $\Theta(n^3)$ solve sample-invariant.

When an expression is only *partially* sample-invariant, SICM applies algebraic rewrite patterns to decompose it into a cheap sample-dependent part and an expensive sample-invariant part. For example, $\text{cholesky}(s \cdot A) = \sqrt{s} \cdot \text{cholesky}(A)$ separates the $O(d^3)$ factorization of the fixed matrix A from the $O(1)$ scalar s, enabling the Cholesky to be hoisted. Similar identities cover triangular solves, scaled matrix products, eigendecomposition lift, and linear operator absorption (FFTs, scatters). The algebraic rewrite patterns compose through a fixpoint iteration: each rewrite may expose new hoisting opportunities for subsequent patterns. Each rewrite is registered as an MLIR `RewritePattern` that statically verifies its preconditions (e.g., $s > 0$ for the Cholesky identity above) before firing; new patterns extend the set without modifying the dataflow analysis. Standard passes (inlining, CSE, LICM) interleave with SICM. For example, recursive inlining exposes the full forward model IR to optimization, CSE deduplicates across sample sites.

3.3 Materialization and Lowering

The materialization pass expands `simulate`/`generate` by recursively unfolding sample sites into distribution calls and trace updates, and lowers `infer` into algorithm-specific loops. For HMC

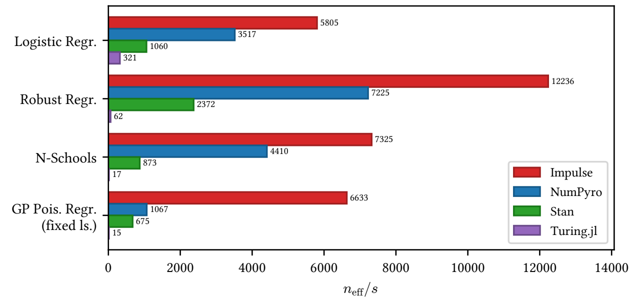

Figure 3: Effective samples per second (n_{eff}/s; higher is better) on standard benchmarks and the motivating example.

and NUTS [Hoffman and Gelman 2014; Neal 2011], the logpdf region defines the potential energy and Enzyme [Moses and Churavy 2020; Moses et al. 2021, 2022] synthesizes its gradient through compiler transformations. Since the log-density is materialized as IR before differentiation, standard passes (LICM, CSE) applied before AD yield asymptotic complexity reductions that post-AD optimization cannot recover [Moses and Churavy 2020] (see §4). Remaining operations lower to StableHLO [OpenXLA Project 2022] for CPU, GPU, and TPU execution via XLA [TensorFlow Team 2017].

4 Results and Contributions

Contribution 1: Impulse implementation and evaluation. We implement our approach in Impulse, built on MLIR [Lattner et al. 2021], Enzyme [Moses and Churavy 2020; Moses et al. 2021, 2022] for AD, and StableHLO [OpenXLA Project 2022] for hardware-accelerated execution. On the three continuous models from PPLBench [Tehrani et al. 2020], a standard PPL benchmark suite, Impulse achieves a geomean speedup of 1.7× over NumPyro [Phan et al. 2019], 6.4× over Stan [Carpenter et al. 2017], and 53× over Turing [Ge et al. 2018] (Figure 3). On models with sample-invariant redundancy, SICM achieves a throughput that is 6.2× higher than NumPyro, 9.8× higher than Stan, and 442× higher than Turing. Because Impulse targets MLIR, the optimizations are language-agnostic: any language (e.g., Julia, Python, C++) that targets MLIR can benefit from the same compiler pipeline.

Contribution 2: Deferred lowering as a design principle. Preserving probabilistic constructs in the compiler IR makes standard passes more effective when applied before lowering. On models with loop-invariant structure, loop-invariant code motion before AD yields 8× to 36× speedups.

Contribution 3: Sample-Invariant Code Motion. We present a new compiler pass, Sample-Invariant Code Motion (SICM), that eliminates redundant deterministic computation from the inference loop. SICM yields 8.5× to 190× speedups on benchmarks with sample-invariant redundancy in Cholesky factorizations, triangular solves, scaled matrix products, and linear operators (e.g., FFTs and scatters). On a GPU-accelerated radio imaging model [Tiede 2022] with 4428 parameters, SICM automatically discovers optimizations behind multiple layers of abstraction and function calls, reducing inference time by 3.7× with less than 5% compilation overhead.

References

Oriol Abril-Pla, Virgile Andreani, Colin Carroll, Larry Dong, Christopher J. Fonnesbeck, Maxim Kochurov, Ravin Kumar, Junpeng Lao, Christian C. Luhmann, Osvaldo A. Martin, Michael Osthege, Ricardo Vieira, Thomas Wiecki, and Robert Zinkov. 2023. PyMC: A Modern, and Comprehensive Probabilistic Programming Framework in Python. *PeerJ Computer Science* 9 (2023), e1516. https://doi.org/10.7717/peerj-cs.1516

Sara Achour and Martin Rinard. 2020. Noise-Aware Dynamical System Compilation for Analog Devices with Legno. In *Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3373376.3378449

Sai Bangaru, Jesse Michel, Kevin Mu, Gilbert Bernstein, Tzu-Mao Li, and Jonathan Ragan-Kelley. 2021. Systematically Differentiating Parametric Discontinuities. *ACM Transactions on Graphics* 40, 4, Article 107 (2021). https://doi.org/10.1145/3450626.3459775

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* (2019).

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: Composable Transformations of Python+NumPy Programs*. http://github.com/jax-ml/jax

Alberto Cabezas, Adrien Corenflos, Junpeng Lao, Rémi Louf, Antoine Carnec, Kaustubh Chaudhari, Reuben Cohn-Gordon, Jeremie Coullon, Wei Deng, Sam Duffield, Gerardo Durán-Martín, Marcin Elantkowski, Dan Foreman-Mackey, Michele Gregori, Carlos Iguaran, Ravin Kumar, Martin Lysy, Kevin Murphy, Juan Camilo Orduz, Karm Patel, Xi Wang, and Rob Zinkov. 2024. BlackJAX: Composable Bayesian Inference in JAX. *arXiv preprint arXiv:2402.10797* (2024). https://arxiv.org/abs/2402.10797

Bob Carpenter, Andrew Gelman, Matthew D. 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), 1–32. https://doi.org/10.18637/jss.v076.i01

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 *Programming Language Design and Implementation*. ACM, 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 (Proceedings of Machine Learning Research)*.

Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

Matthew D. Hoffman and Andrew Gelman. 2014. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* 15, 47 (2014), 1593–1623. http://jmlr.org/papers/v15/hoffman14a.html

Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 194–206.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

Wonyeol Lee, Hangyeol Yu, and Hongseok Yang. 2018. Reparameterization gradient for non-differentiable models. In *International Conference on Neural Information Processing Systems*. https://dl.acm.org/doi/10.5555/3327345.3327459

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018. Differentiable Monte Carlo Ray Tracing through Edge Sampling. *ACM Transactions on Graphics* 37, 6 (2018). https://doi.org/10.1145/3272127.3275109

Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. 2019. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning. *International Conference on Computer Vision* (2019).

Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. 2019. Reparameterizing Discontinuous Integrands for Differentiable Rendering. *Special Interest Group on Computer Graphics and Interactive Techniques in Asia* (2019).

Jesse Michel, Wonyeol Lee, and Hongseok Yang. 2025. Semantics of Integrating and Differentiating Singularities. *Programming Language Design and Implementation* (2025). https://doi.org/10.1145/3729263

Jesse Michel, Kevin Mu, Xuanda Yang, Sai Praveen Bangaru, Elias Rojas Collins, Gilbert Bernstein, Jonathan Ragan-Kelley, Michael Carbin, and Tzu-Mao Li. 2024. Distributions for Compositionally Differentiating Parametric Discontinuities. *Proceedings of the ACM on Programming Languages* OOPSLA1 (2024). https://doi.org/10.1145/3649843

William Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 12472–12485. https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf

William S. Moses et al. 2024. *Reactant.jl: A Julia Front End for MLIR and XLA*. https://github.com/EnzymeAD/Reactant.jl

William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. https://doi.org/10.1145/3458817.3476165

William S. Moses, Sri Hari Krishna Narayanan, Ludger Paehler, Valentin Churavy, Michel Schanen, Jan Hückelheim, Johannes Doerfert, and Paul Hovland. 2022. Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 60, 18 pages.

Radford M. Neal. 2011. MCMC Using Hamiltonian Dynamics. In *Handbook of Markov Chain Monte Carlo*, Steve Brooks, Andrew Gelman, Galin L. Jones, and Xiao-Li Meng (Eds.). Chapman and Hall/CRC, Chapter 5. https://arxiv.org/abs/1206.1901

Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Du Phan, and Jonathan P. Chen. 2019. Functional Tensors for Probabilistic Programming. (2019). arXiv:1910.10775

OpenXLA Project. 2022. StableHLO: Backward Compatible ML Compute Opset Inspired by HLO/MHLO. https://github.com/openxla/stablehlo. Accessed: 2026-03-06.

Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. In *NeurIPS 2019 Program Transformations for Machine Learning Workshop*. https://arxiv.org/abs/1912.11554

Benjamin Sherman, Jesse Michel, and Michael Carbin. 2021. λ_S: Computable Semantics for Differentiable Programming with Higher-Order Functions and Datatypes. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 3 (2021). https://doi.org/10.1145/3434284

Nazanin Tehrani, Nimar S. Arora, Yucen Lily Li, Kinjal Divesh Shah, David Noursi, Michael Tingley, Narjes Torabi, Sepehr Masouleh, Eric Lippert, and Erik Meijer. 2020. PPL Bench: Evaluation Framework for Probabilistic Programming Languages. In *Workshop on Advances in Programming Languages and Neurosymbolic Systems (AIPLANS) at NeurIPS*.

TensorFlow Team. 2017. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla

Paul Tiede. 2022. Comrade: Composable Modeling of Radio Emission. *Journal of Open Source Software* 7, 76 (2022), 4457. https://doi.org/10.21105/joss.04457

Eric Veach. 1998. *Robust monte carlo methods for light transport simulation*. Ph. D. Dissertation. Advisor(s) Guibas, Leonidas J.