DynamicPPL: Stan-like Speed for Dynamic Probabilistic Models

Mohamed Tarek[1,2], Kai Xu[3], Martin Trapp[4], Hong Ge[5], and Zoubin Ghahramani[5,6]

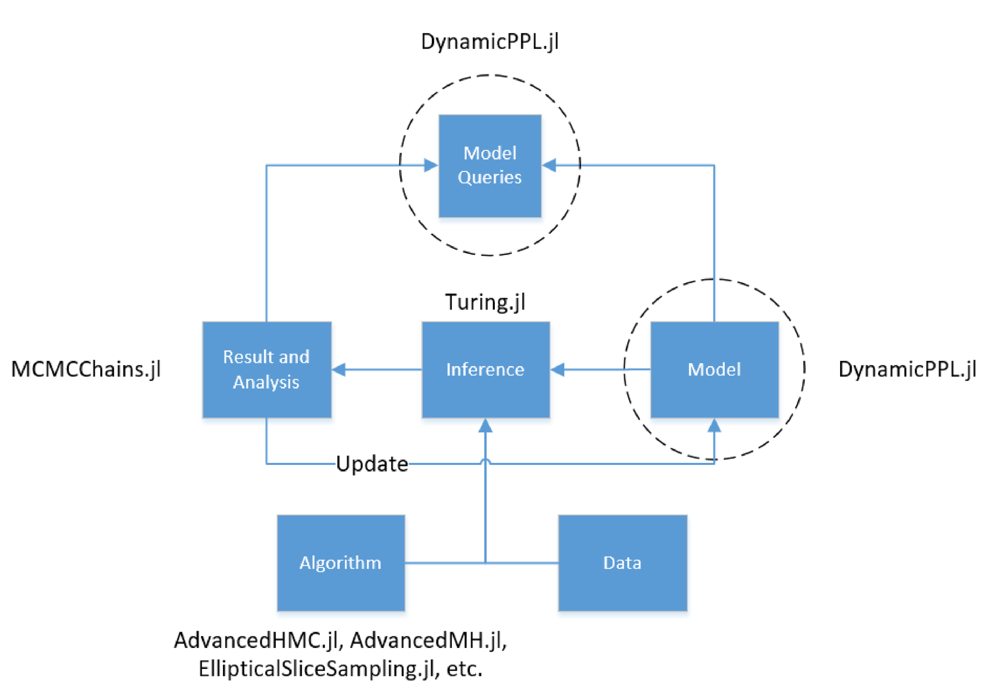
[1]University of New South Wales at Canberra; [2]Pumas-AI Inc.; [3] University of Edinburgh;
[4] Graz University of Technology; [5] University of Cambridge; [6] Google Research

1. Introduction

Practical Bayesian inference for probabilistic models with dynamic dimensionality and variable types is a challenging problem in probabilistic programming languages (PPLs). General-purpose PPLs enable users to introduce new model parameters anywhere in a model definition, freely influence control flows with random variables, and often eliminate the need to specify variable types. Such flexibility often leads to a more productive probabilistic modelling workflow – models are easy to read and write. Such increased productivity is critical for a broader adoption of Bayesian modelling and approximate inference. In this poster, we present DynamicPPL which enabled a significant speedup of inference in Turing – a general-purpose PPL implemented in the Julia programming language. Our approach leverages incremental tracing and type specialisation. We show through an extensive set of benchmarks that the proposed approach significantly accelerates inference in dynamic models (often an order of magnitude faster than a previous version of Turing) and achieves state-of-the-art performance for static models that is close to and sometimes better than Stan's (which is implemented in C++).

The main contribution of this poster is the `DynamicPPL` Julia package, a user-friendly and modular PPL frontend that makes use of incremental tracing and type specialization of model parameters in probabilistic programs allowing `Turing` [2] to achieve speeds close to and sometimes faster than Stan even for static models while efficiently supporting a family of dynamic models.

2. Overview of TuringLang Organisation

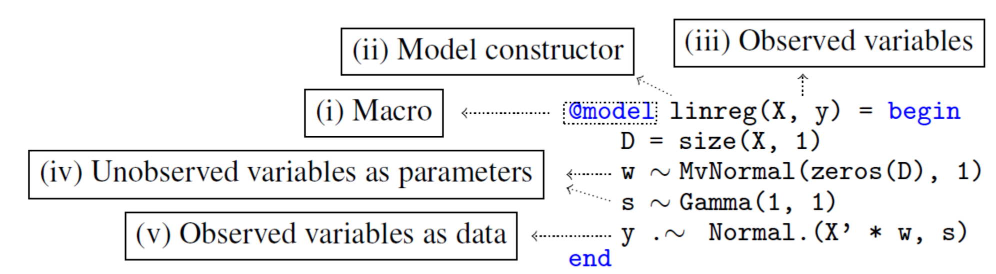


3. Dynamic vs Static Typing

- Dynamic PPLs lack the random variables' type information at compile time.
- Type information of variables is paramount to many compiler optimisations required to generate efficient machine code; Python is slower than C.
- The Julia programming language [1] is a fast dynamically typed programming language; its design and compiler allow it to behave both as a dynamic programming language as well as a static one in different contexts, thus combining the advantages of both paradigms.

4. Design Principles

- User-friendly syntax




```
prob"w = [1.0, 1.0]', s = 1.0 | model = linreg"
prob"X = [1.0, 2.0], y = [2.0] | w = [0.5, 0.0], s = 1.0, model = linreg"
prob"X = [1.0, 2.0], y = [2.0], w = [0.0, 0.0], s = 1.0 | model = linreg"
```

- Inter-operability with the Julia ecosystem
  - Distributions.jl for distributions
  - ForwardDiff.jl, Tracker.jl, ReverseDiff.jl and Zygote.jl for automatic differentiation
  - Flux.jl for neural networks
  - Optim.jl for optimisation used in maximum a-posteriori and maximum likelihood estimation
  - CUDA.jl for GPU data parallelism
  - DifferentialEquations.jl for differential equation solvers
  - Memoization.jl for memoising expensive functions in dynamic Gibbs sampling
  - Arbitrary Julia code allowed including custom distributions
- Performance: comparable performance to Stan

5. Trace Type Specialisation

Trace type specialisation in `DynamicPPL` + `Turing`:

1. Run the model using `UntypedVarInfo`
2. Specialise the container types in the trace creating a `TypedVarInfo`
3. Run the rest of inference using a trace specialised for the model
4. (WIP) Incrementally "expand" the trace type as needed using a `MixedVarInfo`

`Any[1, 2.0, 2, 1.0]` ≈ UntypedVarInfo

↓

`Int[1, 2], Float64[2.0, 1.0]` ≈ TypedVarInfo

The use of `TypedVarInfo` allows for optimised code generation for:

- Static models, and
- A class of dynamic models where the random variable containers' names and types are static, but the number of random variables and their distributions can be dynamic.

The `MixedVarInfo` work will enable efficient code generation of fully dynamic models, where type information can also be dynamic, with 0 overhead for the currently efficient classes of models. New type information can get exploited incrementally to re-generate efficient code using the Julia compiler.

6. Benchmarks

- Static HMC benchmarks of Turing against Stan using AdvancedHMC.jl + ReverseDiff.jl from Turing.jl. Time is in seconds.

	Forward evaluation		Gradient evaluation	
	`Stan`	`Turing`	`Stan`	`Turing`
Gaussian unknown	3.296×10^{-5}	$\mathbf{1.898 \times 10^{-5}}$	$\mathbf{4.23 \times 10^{-5}}$	1.25×10^{-4}
Hierarchical Poisson	1.049×10^{-5}	$\mathbf{7.067 \times 10^{-6}}$	$\mathbf{2.44 \times 10^{-5}}$	7.84×10^{-5}
High dimensional Gaussian	1.307×10^{-3}	$\mathbf{4.579 \times 10^{-5}}$	1.81×10^{-3}	$\mathbf{1.44 \times 10^{-4}}$
Semi-supervised HMM	6.374×10^{-4}	$\mathbf{2.939 \times 10^{-4}}$	$\mathbf{5.97 \times 10^{-4}}$	3.81×10^{-3}
Latent Dirichlet allocation	1.624×10^{-3}	$\mathbf{7.583 \times 10^{-5}}$	1.89×10^{-3}	$\mathbf{1.62 \times 10^{-3}}$
Logistic regression	$\mathbf{1.137 \times 10^{-3}}$	1.525×10^{-3}	$\mathbf{1.59 \times 10^{-3}}$	3.23×10^{-3}
Naïve Bayes	3.672×10^{-4}	$\mathbf{1.246 \times 10^{-5}}$	3.23×10^{-4}	$\mathbf{3.60 \times 10^{-5}}$
Stochastic volatility	$\mathbf{1.298 \times 10^{-4}}$	7.553×10^{-4}	$\mathbf{1.40 \times 10^{-4}}$	7.90×10^{-4}

- Static HMC benchmarks of Turing against Stan using AdvancedHMC.jl + ReverseDiff.jl from Turing.jl. Time is in seconds.

	`Stan`	`Turing`
Gaussian unknown	$\mathbf{1.415 \pm 0.009}$	10.417 ± 0.392
Hierarchical Poisson	$\mathbf{0.310 \pm 0.007}$	5.223 ± 0.645
High dimensional Gaussian	40.418 ± 2.012	$\mathbf{19.576 \pm 6.448}$
Semi-supervised HMM	$\mathbf{42.464 \pm 0.095}$	342.516 ± 124.477
Latent Dirichlet allocation	147.210 ± 7.871	$\mathbf{91.551 \pm 25.194}$
Logistic regression	$\mathbf{105.097 \pm 8.748}$	390.126 ± 8.345
Naïve Bayes	21.282 ± 1.497	$\mathbf{4.273 \pm 0.158}$
Stochastic volatility	$\mathbf{5.552 \pm 0.041}$	230.465 ± 29.179

7. Conclusion

1. Incremental tracing and type specialization facilitates the acceleration of inference in dynamic PPLs such as `Turing`
2. `DynamicPPL` is a modular, high-performance, dynamic PPL frontend implementation with a user-friendly and extensible interface
3. In the future, we hope to extend `DynamicPPL` and `Turing` to support nested modeling, BUGS-style Gibbs sampling and Infer.net-style message passing.

8. References

[1] Jeff Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1 (2017), pp. 65–98.

[2] Hong Ge, Kai Xu, and Zoubin Ghahramani. "Turing: A Language for Flexible Probabilistic Inference". In: *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*. Ed. by Amos Storkey and Fernando Perez-Cruz. Vol. 84. Proceedings of Machine Learning Research. Playa Blanca, Lanzarote, Canary Islands: PMLR, 2018, pp. 1682–1690.