



HAL
open science

Verifying Performance Properties of Probabilistic Inference

Eric Atkinson, Ellie Y. Cheng, Guillaume Baudart, Louis Mandel, Michael Carbin

► **To cite this version:**

Eric Atkinson, Ellie Y. Cheng, Guillaume Baudart, Louis Mandel, Michael Carbin. Verifying Performance Properties of Probabilistic Inference. 2024. hal-04488233

HAL Id: hal-04488233

<https://hal.science/hal-04488233>

Preprint submitted on 4 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Verifying Performance Properties of Probabilistic Inference

Eric Atkinson¹, Ellie Y. Cheng¹, Guillaume Baudart², Louis Mandel³, and Michael Carbin¹

¹ Massachusetts Institute of Technology, USA

² École normale supérieure, PSL University, CNRS, Inria, France

³ IBM Research, USA

Introduction. Probabilistic inference is an NP-hard problem [5], meaning probabilistic inference systems can have poor performance in general. Nevertheless, efficient inference techniques exist for many special classes – including linear-Gaussian models [6], Bayesian networks that are polytrees [9], and Monte Carlo sampling techniques that work well in widespread practical applications. *Probabilistic Programming Languages (PPLs)* provide a general interface for developers to write a program to specify nearly any probabilistic inference problem. This raises the question: how can developers using PPLs have confidence that their programs will achieve good performance (e.g. by falling into one of the special classes)?

In this abstract, we discuss the opportunity to formally verify that inference systems for probabilistic programming guarantee good performance. In particular, we focus on *hybrid inference* systems that combine exact and approximate inference to try to exploit the advantages of each. Their performance depends critically on a) the division between exact and approximate inference, and b) the computational resources consumed by exact inference. We describe several projects in this direction:

- *Semi-symbolic Inference (SSI)* [2] and *Delayed Sampling (DS)* [8] are types of hybrid inference systems, with SSI providing limited guarantees by construction on the exact/approximate division.
- *Verifying the Exact/Approximate Division* is ongoing work to extend SSI’s guarantees to a more complex class of programs, requiring a program analysis to ensure the guarantees.
- *Verifying Memory Consumption* is prior work [1] on verifying that DS inference systems execute in bounded memory.

Together, these projects show that verification can deliver the performance guarantees that PPLs need.

Semi-Symbolic Inference and Delayed Sampling. SSI [2] and its predecessor DS [8] are instances of hybrid inference where the runtime performs exact inference on known efficient special classes with symbolic distributions, and falls back on general approximate sampling when necessary.

Consider the *outlier* example (pseudocode in Fig 1, adapted from [7, Section 2]). This models a latent random variable x that evolves according to a Gaussian random walk (Lines 2 and 4). Observations can be outliers with probability 10% (Line 5). At each step with an outlier, the observation y is modeled with a Gaussian unrelated to x (Line 6). Without an outlier, y is modeled with a Gaussian distribution with mean x as in a standard Kalman filter [6] (Line 7). The code returns x (Line 9), indicating inference of x from the last iteration, conditioned on all prior observations of y as values in the array `yobs`.

For this program, the goal for SSI and DS during inference is to approximately sample values for o at each step. Then, given concrete values for o , the symbolic distributions for the remaining variables form a linear-Gaussian model that is amenable to exact inference. While both SSI and DS can implement this system on the *outlier* example, one advantage of SSI is that it provides a guarantee: given *any* program where each variable is either a) approximate, or b) linear-Gaussian, SSI will never approximate any of the linear-Gaussian variables (see [2] for details, as well as examples where DS fails to provide this guarantee). This ensures that, under limited circumstances, SSI achieves the optimal division between exact and approximate inference.

Verifying the Exact/Approximate Division. Despite its guarantees, SSI’s division between exact and approximate inference within a program remains enigmatic. For example, in the *outlier* program, depending on the internal structure of the symbolic state, SSI may choose to approximate o and compute x and y using exact inference, or alternatively approximate all variables o , x , and y . In general, determining which random variables SSI will choose to approximate can require in-depth knowledge of the algorithm, and the wrong choice can significantly degrade performance [4,2].

```
1 function outlier(yobs) {
2   x ← gaussian(0., 100.);
3   for i in 1 .. N {
4     x ← gaussian(x, 1.);
5     o ← bernoulli(.1);
6     if (o) { y ← gaussian(0., 100.); }
7     else { y ← gaussian(x, 1.); }
8     observe(y, yobs[i]);
9   }; x
10 }
```

Fig. 1. The *outlier* example. Note that we use \leftarrow to sample from a distribution and assume the input `yobs` has length at least `N`.

We propose the use of annotations to control the division of exact and approximate inference at the granularity of individual random variables. Developers force the runtime to approximate a random variable using the `approx` annotation (this is similar to the `value` construct from prior work [8,4,2]). In the *outlier* example, annotating `o` with `approx` would cause SSI to always approximate `o`, guaranteeing that SSI would perform exact inference on `x` and `y`. A developer could further apply the `exact` annotation to `x`, which functions as an assertion that the runtime will compute `x` exactly. Conversely, should the user fail to annotate `o` with `approx`, then an `exact` annotation on `x` will raise an error.

We further propose to formally verify `exact` annotations at compile-time, and automate this verification with an abstract-interpretation-based static analysis. Such an analysis would be provably sound – i.e. never consider a variable `exact` that SSI approximates in any execution – and we hypothesize it will empirically be precise enough for common examples. Thus, the analysis provides a certificate that a variable will be exact, which serves as an enhanced version of SSI’s performance guarantee.

Verifying Memory Consumption. For both SSI and DS, the exact inference component of the system maintains a symbolic distribution representation at runtime. This leads to the question: how large can this representation grow? Developers would like their inference runtimes to maintain *bounded memory*, meaning the size of the runtime state is a constant multiple of the number of variables in the program (see [1] for a detailed definition). For example, in the *outlier* example of Fig. 1, although each iteration instantiates three *new* random variables in the symbolic state (assigning them to the program variables `x`, `o`, and `y`) only a *total* of three random variables are ever need at runtime (the ones pointed to by the program variables). In this case, a bounded-memory runtime would use a constant amount of memory regardless of the value of `N`.

Whether or not DS runs in bounded memory depends on the probabilistic program in question [4,1]. The work in [1] identified two program properties – the *m*-consumed property and the unseparated paths property – which together form a necessary and sufficient condition for DS to be bounded-memory. These are dataflow properties of the program that are unique to the bounded-memory inference problem. The work in [1] also presents how to automatically verify these properties with a static analysis that is provably sound and empirically – across a range of benchmarks – precise enough to verify bounded-memory execution of numerous programs. Overall, this provides a provably sound way to ensure DS inference systems achieve good memory performance (we are not yet aware of any work extending this to SSI).

Conclusion. We have discussed techniques to formally verify that hybrid inference systems will achieve good performance on a program. We discussed SSI, its guarantee on the exact/approximate division, and ongoing work to extend this guarantee. We also discussed how to formally verify DS runs in bounded memory.

We close by proposing several directions for future work. A natural direction is to extend the DS bounded-memory guarantees to SSI. We hypothesize that the same DS properties are sufficient but not necessary for bounded-memory SSI, suggesting that this endeavor will require more work. A related direction is to determine bounds on SSI’s computational runtime to complement the memory bounds. We suspect computation bounds to be related to tree-width as is the case for Bayesian networks. Finally, one of the major sources of performance unpredictability in hybrid inference is the amount of approximation error. While reasoning about the error from approximate inference is a challenging problem, we predict that recent work on efficiency analysis of rejection sampling [3] can be adapted to formally verify the approximation error of hybrid inference systems. Together, tools for verifying the exact/approximate division, memory consumption, computational runtime, and approximation error would provide important guarantees for most if not all sources of performance issues in hybrid inference systems for probabilistic programming.

1. Atkinson, E., Baudart, G., Mandel, L., Yuan, C., Carbin, M.: Statically bounded-memory delayed sampling for probabilistic streams. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2021). <https://doi.org/10.1145/3485492>
2. Atkinson, E., Yuan, C., Baudart, G., Mandel, L., Carbin, M.: Semi-symbolic inference for efficient streaming probabilistic programming. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2022), <https://doi.org/10.1145/3563347>
3. Batz, K., Kaminski, B.L., Katoen, J.P., Mateja, C.: How long, O Bayesian network, will I sample thee? In: ESOP (2018)
4. Baudart, G., Mandel, L., Atkinson, E., Sherman, B., Pouzet, M., Carbin, M.: Reactive probabilistic programming. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2020). <https://doi.org/10.1145/3385412.3386009>
5. Cooper, G.F.: The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence* **42**(2-3), 393–405 (1990)
6. Kalman, R.E.: A new approach to linear filtering and prediction problems. *Journal of Basic Engineering* **82**(1), 35–45 (03 1960)
7. Minka, T.P.: Expectation propagation for approximate bayesian inference. In: UAI. pp. 362–369 (2001)
8. Murray, L.M., Lundén, D., Kudlicka, J., Broman, D., Schön, T.B.: Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In: International Conference on Artificial Intelligence and Statistics (2018)
9. Pearl, J., Rebane, G.: The Recovery of Causal Poly-Trees from Statistical Data. In: UAI (1987)