Semi-symbolic Inference for Efficient Streaming Probabilistic Programming

ERIC ATKINSON, MIT, USA
CHARLES YUAN, MIT, USA
GUILLAUME BAUDART, DI ENS, École normale supérieure, PSL University, CNRS, Inria, France
LOUIS MANDEL, MIT-IBM Watson AI Lab, IBM Research, USA
MICHAEL CARBIN, MIT, USA

A streaming probabilistic program receives a stream of observations and produces a stream of distributions that are conditioned on these observations. Efficient inference is often possible in a streaming context using Rao-Blackwellized particle filters (RBPFs), which exactly solve inference problems when possible and fall back on sampling approximations when necessary. While RBPFs can be implemented by hand to provide efficient inference, the goal of streaming probabilistic programming is to automatically generate such efficient inference implementations given input probabilistic programs.

In this work, we propose *semi-symbolic inference*, a technique for executing probabilistic programs using a runtime inference system that automatically implements Rao-Blackwellized particle filtering. To perform exact and approximate inference together, the semi-symbolic inference system manipulates symbolic distributions to perform exact inference when possible and falls back on approximate sampling when necessary. This approach enables the system to implement the same RBPF a developer would write by hand. To ensure this, we identify *closed families* of distributions – such as linear-Gaussian and finite discrete models – on which the inference system guarantees exact inference. We have implemented the runtime inference system in the ProbZelus streaming probabilistic programming language. Despite an average 1.6× slowdown compared to the state of the art on existing benchmarks, our evaluation shows that speedups of 3×–87× are obtainable on a new set of challenging benchmarks we have designed to exploit closed families.

CCS Concepts: • **Mathematics of computing** → *Sequential Monte Carlo methods*; • **Theory of computation** → *Streaming models*; • **Software and its engineering** → *Data flow languages*.

Additional Key Words and Phrases: probabilistic programming, streaming inference

ACM Reference Format:
Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022. Semi-symbolic Inference for Efficient Streaming Probabilistic Programming. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 184 (October 2022), 29 pages. https://doi.org/10.1145/3563347

1 INTRODUCTION

Probabilistic programming languages enable developers to describe a probabilistic model in a programming language and let the language's compiler and runtime perform Bayesian inference [Bingham et al. 2019; Goodman and Stuhlmüller 2014; Murray and Schön 2018; Tolpin et al. 2016; Tran et al. 2017]. In this work, we focus on *streaming* probabilistic programs, as first formalized in Baudart

Authors' addresses: Eric Atkinson, MIT, USA; Charles Yuan, MIT, USA; Guillaume Baudart, DI ENS, École normale supérieure, PSL University, CNRS, Inria, France; Louis Mandel, MIT-IBM Watson AI Lab, IBM Research, USA; Michael Carbin, MIT, USA.

2475-1421/2022/10-ART184
https://doi.org/10.1145/3563347

et al. [2020]. In a streaming probabilistic program, the program receives a stream of observations and produces a stream of distributions that are conditioned on these observations.

Streaming Inference. Rao-Blackwellized particle filtering [Doucet et al. 2000] is a state-of-the-art inference technique that can be used in a streaming context. Rao-Blackwellized particle filters (RBPFs) exactly solve inference problems when possible (i.e., when a closed-form solution exists) and fall back on sampling-based approximate particle filtering [Gordon et al. 1993] when symbolic computations fail. The key challenge to applying RBPFs is designing an effective state representation that maintains both a sample-based representation and a symbolic representation, with as much of the state as possible in the symbolic representation.

Semi-Symbolic Inference. In this work, we propose *semi-symbolic inference*, in which a particle filter is augmented with a symbolic state consisting of mathematical expressions that encode distributions of random variables in the program. At runtime, the semi-symbolic inference system transforms the expressions in the symbolic state according to closed-form solutions from probability theory. Compared to previous work using a different state representation [Baudart et al. 2020], semi-symbolic inference can maintain an exact representation in more cases. In particular, we prove that the semi-symbolic inference system guarantees an exact representation on *closed families*, which include linear-Gaussian and finite discrete probabilistic models.

Contributions. In this paper, we present the following contributions:

- We present semi-symbolic inference, a new technique for Rao-Blackwellized particle filtering in streaming probabilistic programs. In Section 4, we define the state representation as well as the operations the semi-symbolic inference system uses to perform inference.
- We discuss the guarantees and advantages of semi-symbolic inference. In Section 5, we state and prove several theorems about closed families. In particular, we show that the semi-symbolic inference system provides guaranteed exact inference on linear-Gaussian and finite discrete probabilistic models, which developers can use to reliably write models that the inference system implements as RBPFs.
- We implement semi-symbolic inference inside the streaming probabilistic programming language ProbZelus [Baudart et al. 2020]. The implementation is available at https://github.com/ibm/probzelus, and has been accepted as an artifact [Atkinson et al. 2022a].
- We evaluate semi-symbolic inference in ProbZelus on a set of benchmarks in Section 6, and compare against prior work on *delayed sampling* [Murray et al. 2018] in ProbZelus. We show that semi-symbolic inference has a slowdown of 1.6× on existing benchmarks. In exchange for this overhead, by maintaining exact representations more often than delayed sampling, semi-symbolic inference can achieve speedups of 3×–87× on a set of challenging benchmarks that exercise the closed family guarantee. We discuss in detail two of these benchmarks that illustrate the situations in which delayed sampling fails to perform exact inference but semi-symbolic inference can.

By executing streaming probabilistic programs with semi-symbolic inference, developers can combine the efficiency of exact inference with the generality of approximate inference. The closed family guarantee further ensures that the programs will deliver the performance developers expect. An extended version of this paper including appendices is available [Atkinson et al. 2022b].

2 EXAMPLE

To demonstrate semi-symbolic inference, we use the streaming probabilistic programming language ProbZelus to model a robot with two wheels. Figure 2 shows a diagram depicting the robot. Our objective is to estimate the angular and forward velocity of the robot using sensors that measure

```
1 let proba wheels (left_rate, right_rate) = (vel, omega) where
2   rec init omega = 0. and init vel = 0.
3   and omega = sample (gaussian (last omega, omega_var))
4   and vel = sample (gaussian (last vel, vel_var))
5   and () = observe (gaussian (vel -. wb *. omega, sensor_err), left_rate)
6   and () = observe (gaussian (vel +. wb *. omega, sensor_err), right_rate)
```

Fig. 1. A program that encodes the robot model in ProbZelus. The model inputs are two streams `left_rate` and `right_rate` encoding the speed sensors on each wheel, and the outputs are streams of the estimated velocity `vel` and angular velocity `omega`. The constants `omega_var`, `vel_var`, and `sensor_err` specify variances of `omega` and `vel`, and the wheel sensors respectively, and `wb` specifies the width of the robot's wheel base. In our example, we assume values `omega_var` = 2500, `vel_var` = 2500, `wb` = 2, and `sensor_err` = 1.

the speed of each wheel. Such a task is often a precursor to estimating the position of the robot, as described, for example, in Larsen et al. [1999].

In this section, we first explain how to implement such a model in the ProbZelus streaming probabilistic programming language. We then explain how our implementation of a semi-symbolic runtime inference system for ProbZelus executes this program.

2.1 Implementation in ProbZelus

Figure 1 presents an implementation of the model in the ProbZelus streaming probabilistic programming language. The core objects in ProbZelus are *stream functions* that transform input streams into output streams. The `proba` keyword on Line 1 signifies the definition of a stream function whose definition may include probabilistic operators. The remainder of the line specifies that the `wheels` stream function takes as input a pair of streams `left_rate` and `right_rate` encoding the speed sensors on each wheel, and returns a pair of the robot's estimated velocity `vel` and angular velocity `omega`. These estimated velocities are defined by the subsequent mutually recursive equations.

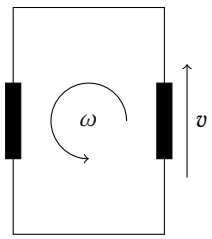


Fig. 2. Diagram of a two-wheeled robot. The objective is to estimate the angular velocity ω and the forward velocity v using wheel sensors.

Lines 2 and 3 specify a probabilistic model for the angular velocity `omega`. At each time step, Line 3 specifies that `omega` is a stream of values, each sampled from a Gaussian distribution using the `sample` probabilistic operator. The mean of the Gaussian is given by the previous value in the stream of `omega` values as specified using the `last omega` syntax, except that at the first time step, `last omega` takes on the initial value of 0 as specified on Line 2 by the `init` keyword. The variance of the Gaussian is given by the constant `omega_var`.

Lines 2 and 4 specify a model for the forward velocity `vel` symmetric to that for angular velocity.

Line 5 conditions the model by observing the left wheel's velocity from the forward velocity and angular velocity. In particular, it specifies that the left wheel's velocity is sampled from a Gaussian distribution. The mean of this distribution subtracts from the forward velocity `vel` the angular velocity `omega` multiplied by the constant `wb`, which represents the width of the robot's wheel base. The program then specifies, using the `observe` probabilistic operator, that the model is conditioned on this Gaussian random variable being equal to the input value `left_rate`. Line 6 specifies a similar observation to for the right wheel, except that the product of `omega` and `wb` is added to `vel` instead of subtracted. For both observations, the Gaussian's variance is the constant `sensor_err`.

2.2 Semi-Symbolic Inference

We now describe how to use semi-symbolic inference to execute the ProbZelus program in Figure 1. In particular, we describe how the *symbolic state* of the semi-symbolic runtime inference system evolves over the course of the first iteration of the `wheels` stream function. The symbolic states described in this section are depicted in Figure 3.

Sampling. First, the semi-symbolic inference system executes the `sample` operators on Lines 3 and 4. The semi-symbolic implementation of `sample` constructs symbolic terms representing distributions and returns the random variables X_v and X_o that point to these distributions. It stores handles to X_v and X_o inside the program variables `vel` and `omega`, respectively.

Figure 3a depicts the symbolic state after both of these `sample` operations. In these depictions, we assume the constant variances `omega_var` and `vel_var` are both equal to 2500, corresponding to a standard deviation of $\sqrt{2500} = 50$ for both forward and angular velocity.

Observation. Next, the semi-symbolic inference system executes the `observe` operation on Line 5. Like in standard probabilistic languages (e.g. Goodman and Stuhlmüller [2014]), the `observe` operation performs a scoring operation. However, in semi-symbolic inference, it also has to update the symbolic state. Here, we describe the symbolic state update. The inference system first constructs a random variable X_l representing the distribution specified on Line 5. Where the program refers to the variables `vel` and `omega`, in the corresponding symbolic distribution these expressions contain handles to the corresponding random variables X_v and X_o.

Figure 3b depicts the resulting symbolic state. In these depictions, we assume that the constant variance `sample_err` is 1 and the constant `wb` is 2.

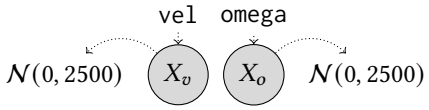
Swapping. To perform the observation, the semi-symbolic inference system first converts this new random variable into a *root*, meaning a random variable with no parents. A parent of a random variable is another random variable that is mentioned in any of its sub-expressions. To perform this conversion, the system performs a *swap*, the core operation of semi-symbolic inference. A swap changes the dependency order between two random variables in the symbolic state. The swap ensures that the overall joint distribution of variables in the symbolic state does not change.

In the example, the inference system tries to swap X_l with the random variable to which `vel` points, X_v. The system must first determine if such a swap is possible. Based on the state depicted in Figure 3b, the system detects that X_l can be written as an affine function of X_v, namely $X_l = a * X_v + b$ where $a = 1$ and $b = -2 * X_o$ (note that random variables other than X_v may appear in a and b). Because the two distributions are Gaussians with constant variance, and they are related by an affine function, the system determines that a swap is possible.

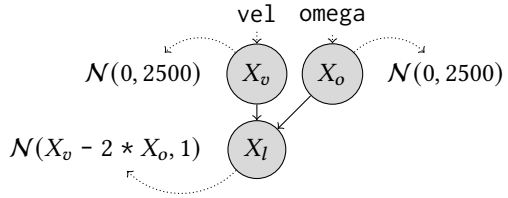
Conjugate Priors. The mathematical principle behind a swap is that of *conjugate priors* [Fink 1997]. Conjugate priors provide rules for manipulating distributions that are useful for performing exact inference. The rules change the direction of dependencies while preserving the overall joint distribution between the involved random variables. While an advanced understanding of the theory of conjugate priors is not necessary to understand semi-symbolic inference, we briefly explain how conjugate priors apply to the swap of X_l and X_v in the example. This swap uses the conjugate rule that given random variables $X_c \sim \mathcal{N}(\mu_c, \sigma_c^2)$ and $X_d \sim \mathcal{N}(X_c, \sigma_d^2)$, then the distribution of X_d is equivalent to $\mathcal{N}(\mu_c, \sigma_c^2 + \sigma_d^2)$, and the distribution of X_c is equivalent to

$$X_c \sim \mathcal{N}\left(\left(\frac{\mu_c}{\sigma_c^2} + \frac{X_d}{\sigma_d^2}\right) * \sigma_{\text{cond}}^2, \sigma_{\text{cond}}^2\right), \text{ where } \sigma_{\text{cond}}^2 = \left(\frac{1}{\sigma_c^2} + \frac{1}{\sigma_d^2}\right)^{-1}$$

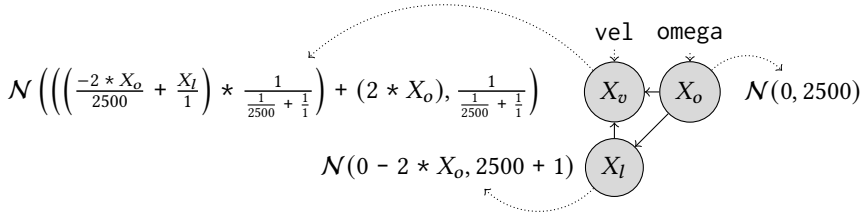
The swap also uses the general rule of linear transformations of Gaussians: if $X_a \sim \mathcal{N}(\mu_a, \sigma_a^2)$, and $X_b = a * X_a + b$, then $X_b \sim \mathcal{N}(a * \mu_a + b, a^2 * \sigma_a^2)$.



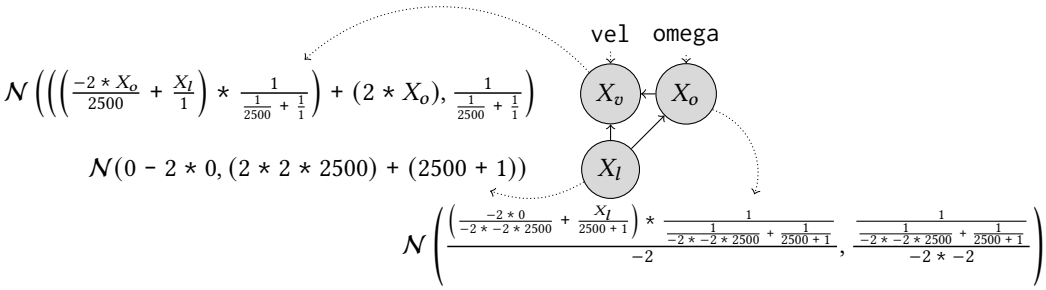
(a) The symbolic state after executing Lines 3 and 4. Each program variable points to a new random variable with a Gaussian distribution.



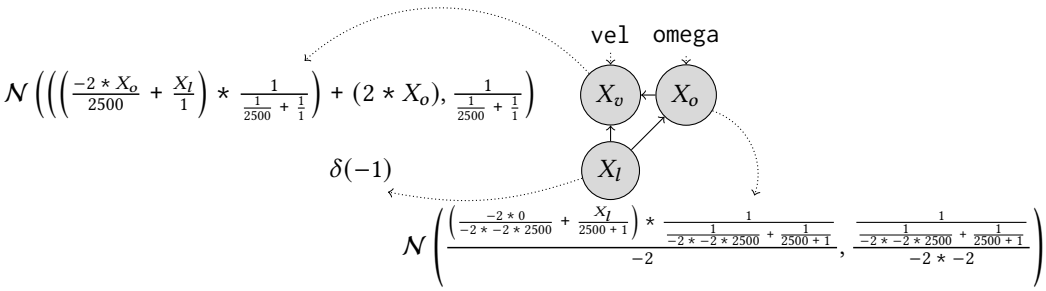
(b) The first step in executing the `observe` on Line 5 adds a new random variable with the appropriate distribution. The symbolic distribution contains references to both previous random variables.



(c) Next, the inference system swaps the new random variable X_l and X_v. The result is that X_l no longer depends on X_v and X_v depends on X_l and X_o.



(d) Next, the inference system swaps X_l with X_o. The distribution of X_l now has no dependencies.



(e) Then, the system intervenes to replace X_l's distribution with a Delta distribution.

Fig. 3. Depiction of the evolution of the symbolic state during the execution of the program in Figure 1 under semi-symbolic inference. We use gray circles represent random variables, and solid arrows to represent conceptual dependencies between random variables. Dotted lines point a) from program variables to the random variables they refer to, and b) from random variables to their symbolic distributions. We also depict the current values of the `vel` and `omega` program variables.

Figure 3c depicts the resulting state of the swap of X_l and X_v. The distribution of the variable to which `omega` points is unchanged, but the other two random variables have new distributions to satisfy their reversed dependence relationship while maintaining the correct joint distribution.

Next, the inference system swaps the variable to which `omega` points, X_o, with X_l. The system determines that these variables have an affine relationship, namely $X_l = a * X_o + b$ where $a = -2$ and $b = 0$, and can therefore be swapped. Figure 3d depicts the results. The system replaces the distribution of X_l and X_o, and leaves the distribution of X_v unchanged. Note that at this point, the distribution of X_l does not depend on any other random variables, and can be evaluated to produce a closed-form distribution.

Intervention. Next, the inference system performs the observation by intervening on the value of X_l. Let us assume that the input observed value (i.e., the value of `left_rate`) is -1. To condition the symbolic state on the fact that X_l is equal to -1, the system replaces the distribution of X_l with the *Delta distribution* $\delta(-1)$, the distribution with all mass at -1. The result is depicted in Figure 3e.

Simplification. The system further simplifies the symbolic state in Figure 3e using the fact that a random variable sampled from a Delta distribution can be replaced with the interior expression of the Delta. In this example, this means that when the variable X_l appears in a distribution expression, the system can replace it with -1. Using this fact and subsequent evaluation steps, the distribution for X_o evaluates to $\mathcal{N}(0.4, 500)$, and the distribution for X_v partially evaluates to $\mathcal{N}\left(\left(\frac{-2*X_o}{2500} + -1\right) * 1, 1\right)$. To facilitate this simplification, the semi-symbolic inference system incorporates a partial evaluator, which we describe in Section 4.2.

Iteration. The inference system performs the observation on Line 6 using a similar sequence of steps. This concludes the system's operations at the first iteration. Subsequent iterations execute similar operations, except that `vel` and `omega` are sampled by reference to their previous value instead of their initial value 0. In subsequent symbolic states, the random variables to which `vel` and `omega` point contain symbolic expressions that refer to variables sampled at earlier iterations.

Accuracy. As a result of the symbolic manipulations, the distributions of `vel` and `omega` become closed-form symbolic expressions. These expressions estimate the forward and angular velocity of the robot and may then be used to estimate additional properties such as its position. Importantly, the distributions are exact, avoiding any loss of accuracy introduced by sampling approximations.

This is a notable improvement over the prior implementation of ProbZelus, which implemented inference using *delayed sampling* [Murray et al. 2018]. Delayed sampling executes similarly to semi-symbolic inference, but has a different symbolic state representation. Delayed sampling cannot represent the distributions of `vel` and `omega` exactly, and thus falls back on approximate sampling. We further discuss the implementation of ProbZelus in Section 3, and provide more details on the specific differences between delayed sampling and semi-symbolic inference Section 6.4.

3 BACKGROUND: PROBZELUS SYNTAX AND SEMANTICS

In this section, we review from Baudart et al. [2020] the syntax and semantics of ProbZelus, the streaming probabilistic programming language within which we implement semi-symbolic inference. We consider a fragment of ProbZelus to illustrate the core concepts of how the system works. A full presentation of the semantics reviewed here can be found in Baudart et al. [2020].

Syntax. We describe the fragment of ProbZelus presented in Figure 4. A program is given by a sequence of stream function declarations each introduced by the keyword `proba`.

An expression is a variable x, a constant c, a pair of expressions, an operator application op, a function application f, an access to the previous value of a variable with `last`, or an expression

$$
\begin{aligned}
\mathit{Decl} &::= \texttt{let proba}\ f\ x = \mathit{Expr} \mid \mathit{Decl}\ \mathit{Decl} \\
\mathit{Expr} &::= x \mid c \mid (\mathit{Expr}, \mathit{Expr}) \mid \mathit{op}(\mathit{Expr}) \mid f(\mathit{Expr}) \mid \texttt{last}\ x \mid \mathit{Expr}\ \texttt{where rec}\ \mathit{Eq} \\
&\quad \mid \texttt{present}\ \mathit{Expr}\ \texttt{->}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr} \mid \texttt{reset}\ \mathit{Expr}\ \texttt{every}\ \mathit{Expr} \\
\mathit{Eq} &::= x = \mathit{Expr} \mid \texttt{init}\ x = c \mid \mathit{Eq}\ \texttt{and}\ \mathit{Eq}
\end{aligned}
$$

Fig. 4. Syntax for a fragment of the ProbZelus language.

whose free variables are defined by a set of mutually recursive equations given by `where rec`. The `present` construct executes one of its two branches depending on the value of the first expression which can be a Boolean expression or a sporadic signal. The `reset` expression re-initializes the state of the first expression (i.e. `last` x returns x's inital value) each time the second expression is true.

The contents of a `where rec` definition either assigns a variable to an expression or specifies an initial value for a variable using the `init` keyword. The initial value works in conjunction with the `last` expression to provide the value `last` returns on the first time step.

The operators of the language include standard arithmetic operators as well as the probabilistic operators `sample` to draw from a distribution and `observe` to condition on an observation. The operator `value` forces the inference system to explicitly draw a sample. The purpose of the `value` operator is to enable developers to explicitly control the location of random sampling.

Semantic Model. The semantics of a deterministic expression e is given by a pair of an initial memory state $[\![e]\!]^{\text{init}} \in M$ and a step function $[\![e]\!]^{\text{step}}_{\gamma} \in M \mapsto O \times M$. The step function is parameterized by an environment γ that contains the value of the free variables of e. The step function takes the current memory and outputs a result value and an updated memory.

A program produces a stream of outputs $(o_j)_{j \in \mathbb{N}}$ by repeatedly executing the step function on the stream of inputs contained in the environment γ_i starting from the initial memory m_0:

$$
\begin{aligned}
m_0 &= [\![e]\!]^{\text{init}} \\
o_1, m_1 &= [\![e]\!]^{\text{step}}_{\gamma_1}(m_0) \\
o_2, m_2 &= [\![e]\!]^{\text{step}}_{\gamma_2}(m_1) \\
&\cdots \\
o_n, m_n &= [\![e]\!]^{\text{step}}_{\gamma_n}(m_{n-1}) \\
&\cdots
\end{aligned}
$$

In contrast, the step function of a probabilistic expression takes the current memory and returns a measure over pairs (result, new memory): $\{\![e]\!\}^{\text{step}}_{\gamma} \in M \mapsto \Sigma_{O \times M} \mapsto [0, \infty)$ where $\Sigma_{O \times M}$ denotes the σ-algebra over pairs of results and memory. This measure is then normalized and split into a distribution of results and a distribution of memories. At each step, we integrate the step function over the current distribution of memories to compute the next distribution of results and the next distribution of memories. Details can be found in [Baudart et al. 2020, Section 3.3].

3.1 Particle Filtering Semantics

The semantics of an expression e may be given by approximate sampling through *particle filtering*. Particle filtering defines the semantics of e by performing multiple independent executions (particles) of e and tracking the likelihood of each execution. We define the semantics of each particle for an expression e by an initial memory $\{\![e]\!\}^{\text{init}}$ and a step function $\{\![e]\!\}^{\text{step}}_{\gamma}$. The step function of

$$
\begin{aligned}
\{\!|c|\!\}^{\text{init}} &= () \\
\{\!|c|\!\}^{\text{step}}_{\gamma}(m, w) &= (c, m, w) \\
\{\!|x|\!\}^{\text{init}} &= () \\
\{\!|x|\!\}^{\text{step}}_{\gamma}(m, w) &= (\gamma(x), m, w) \\
\{\!|\texttt{last}\ x|\!\}^{\text{init}} &= () \\
\{\!|\texttt{last}\ x|\!\}^{\text{step}}_{\gamma}(m, w) &= (\gamma(x\_\texttt{last}), m, w) \\
\{\!|op(e)|\!\}^{\text{init}} &= \{\!|e|\!\}^{\text{init}} \\
\{\!|op(e)|\!\}^{\text{step}}_{\gamma}(m, w) &= \mathit{let}\ v, m', w' = \{\!|e|\!\}^{\text{step}}_{\gamma}(m, w)\ \mathit{in} \\
&\quad (op(v), m', w') \\
\{\!|\texttt{sample}(e)|\!\}^{\text{init}} &= \{\!|e|\!\}^{\text{init}} \\
\{\!|\texttt{sample}(e)|\!\}^{\text{step}}_{\gamma}(m, w) &= \mathit{let}\ \mu, m', w' = \{\!|e|\!\}^{\text{step}}_{\gamma}(m, w)\ \mathit{in} \\
&\quad (\textsc{draw}(\mu), m', w') \\
\{\!|\texttt{observe}(e_\mu, e_v)|\!\}^{\text{init}} &= (\{\!|e_\mu|\!\}^{\text{init}}, \{\!|e_v|\!\}^{\text{init}}) \\
\{\!|\texttt{observe}(e_\mu, e_v)|\!\}^{\text{step}}_{\gamma}((m_\mu, m_v), w) &= \mathit{let}\ \mu, m'_\mu, w_1 = \{\!|e_\mu|\!\}^{\text{step}}_{\gamma}(m_\mu, w)\ \mathit{in} \\
&\quad \mathit{let}\ v, m'_v, w_2 = \{\!|e_v|\!\}^{\text{step}}_{\gamma}(m_v, w_1)\ \mathit{in} \\
&\quad ((), (m'_\mu, m'_v), w_2 * \textsc{score}(\mu, v)) \\
\{\!|\texttt{value}(e)|\!\}^{\text{init}} &= \{\!|e|\!\}^{\text{init}} \\
\{\!|\texttt{value}(e)|\!\}^{\text{step}}_{\gamma}(m, w) &= \{\!|e|\!\}^{\text{step}}_{\gamma}(m, w) \\
\left\{\!\!\left|\begin{array}{l} e\ \texttt{where rec init}\ x_1 = c_1\ \texttt{and}\ \ldots\ \texttt{and init}\ x_k = c_k \\ \qquad\qquad \texttt{and}\ y_1 = e_1\ \texttt{and}\ \ldots\ \texttt{and}\ y_n = e_n \end{array}\right|\!\!\right\}^{\text{init}} & \\
\quad = ((c_1, \ldots, c_k), (\{\!|e_1|\!\}^{\text{init}}, \ldots, \{\!|e_n|\!\}^{\text{init}}), \{\!|e|\!\}^{\text{init}}) & \\
\left\{\!\!\left|\begin{array}{l} e\ \texttt{where rec init}\ x_1 = c_1\ \texttt{and}\ \ldots\ \texttt{and init}\ x_k = c_k \\ \qquad\qquad \texttt{and}\ y_1 = e_1\ \texttt{and}\ \ldots\ \texttt{and}\ y_n = e_n \end{array}\right|\!\!\right\}^{\text{step}}_{\gamma}(((v_1, \ldots, v_k), (m_1, \ldots, m_n), m), w) & \\
\quad = \mathit{let}\ \gamma_1 = \gamma[v_1/x_1\_\texttt{last}]\ \mathit{in} & \\
\quad\ \ldots & \\
\quad\ \mathit{let}\ \gamma_k = \gamma[v_k/x_k\_\texttt{last}]\ \mathit{in} & \\
\quad\ \mathit{let}\ v_1, m'_1, w_1 = \{\!|e_1|\!\}^{\text{step}}_{\gamma_k}(m_1, w)\ \mathit{in} & \\
\quad\ \mathit{let}\ \gamma'_1 = \gamma_k[y_1/v_1]\ \mathit{in} & \\
\quad\ \ldots & \\
\quad\ \mathit{let}\ v_n, m'_n, w_n = \{\!|e_n|\!\}^{\text{step}}_{\gamma'_{n-1}}(m_n, w_{n-1})\ \mathit{in} & \\
\quad\ \mathit{let}\ \gamma'_n = \gamma'_{n-1}[y_n/v_n]\ \mathit{in} & \\
\quad\ \mathit{let}\ v, m', w' = \{\!|e|\!\}^{\text{step}}_{\gamma'_n}(m, w_n)\ \mathit{in} & \\
\quad\ (v, ((\gamma'_n(x_1), \ldots, \gamma'_n(x_k)), (m'_1, \ldots, m'_n), m'), w') &
\end{aligned}
$$

Fig. 5. Particle filtering semantics for a subset of ProbZelus constructs.

each particle takes a memory m and a real-valued weight w as input, and returns the value of the expression, the updated memory for the next iteration, and the updated weight.

Figure 5 presents the particle filtering semantics of selected ProbZelus expressions. The initial memory of a constant is empty and is represented with the value `()`. Its step function returns the value of the constant and leaves the memory and weight unchanged. Similarly, an access to a variable or the last value of a variable does not update the memory, and its value is taken from

the environment γ. The semantics of the application of an operator $op(e)$, e.g. + or $*$, applies the operator to the evaluation of the sub-expression e and propagates the memory and the weight.

The expression e `where rec` Eq introduces and updates the state variables $x_1, \ldots, x_k$, which must be the subset of the variables $y_1, \ldots, y_n$ used as `last` y_i. To define the semantics, we assume that all `init` equations appear first and other equations are sorted according to their data dependencies where the `last` operator does not introduce a dependency. The memory is composed of a slot for each variable introduced by an `init` equation and the memory required for each sub-expression. The step function puts all the state variables in the environment γ_k and then evaluates each equation to compute the current value of each variable. Finally, the expression returns the value of e evaluated in the environment containing the value of all the variables. The memory is updated with the current value of the state variables.

The semantics of `sample`(e) introduces probability into program execution. The step function evaluates the sub-expression e to obtain a distribution μ and draws a random value from μ using the DRAW primitive. The memory of this expression is the memory of the sub-expression.

The expression `observe`(e_μ, e_v) conditions the model using the weight w. The step function evaluates the two sub-expressions to obtain a distribution μ and a value v and updates the weight by multiplying it by $\text{SCORE}(\mu, v)$, the value of the probability density of μ in v, i.e. $\text{SCORE}(\mu, v) = \mu_{\text{pdf}}(v)$.

In the particle filter semantics, the `value` function simply evaluates its argument.

Inference. In this work, we present inference as a transformation called `infer` operating on ProbZelus expressions that can contain free variables whose values are defined in the environment γ. The semantics of `infer` on expression e defines a stream of distributions as an initial memory and a step function. The memory is the distribution of possible memory configurations for e. The step function computes the distribution of outputs and the distribution of memories by performing N independent executions of e with the input memory sampled from the distribution of memories from the previous iteration.

For particle filtering (PF), the `infer` construct is defined as follows:

$$
\begin{array}{lcl}
[\![\texttt{infer}_{\text{PF}}(N, e)]\!]^{\text{init}} & = & \delta_{\{[\![e]\!\}^{\text{init}}} \\
[\![\texttt{infer}_{\text{PF}}(N, e)]\!]^{\text{step}}_{\gamma}(m) & = & \textit{let} \begin{bmatrix} v_i, m_i', w_i' = \textit{let } m_i = \text{DRAW}(m) \textit{ in} \\ \qquad\qquad \{[\![e]\!\}^{\text{step}}_{\gamma}(m_i, 1) \end{bmatrix}_{1 \le i \le N} \textit{in} \\
 & & \textit{let } \mu = \lambda U.\ \Sigma_{i=1}^{N} \overline{w_i} * \delta_{(v_i, m_i)}(U) \textit{ in} \\
 & & (\pi_{1*}(\mu), \pi_{2*}(\mu))
\end{array}
$$

The initial memory is a Delta distribution on the initial memory of e. The step function first builds a size-N array containing the result of N executions of e on memories randomly drawn from the input memory distribution. Then, the distribution μ of pairs of outputs and memory is computed using the weight from each particle: $\overline{w_i} = w_i' / \Sigma_{j=1}^{N} w_j'$. Finally, the individual distributions of outputs and memories are separated using the pushforward of μ across the projections π_1 and π_2.

3.2 Delayed Sampling Semantics

As an alternative to fully approximate particle filtering, ProbZelus programs may execute using *delayed sampling* [Murray et al. 2018], a technique for incorporating exact inference into the runtime inference system. With delayed sampling, the key idea is as follows: instead of eagerly drawing samples at each invocation of the `sample` operation, the system will execute lazily, delaying the sampling operation in the hope that it can find and exploit an opportunity to apply a known closed-form solution to the inference problem. A complete exposition of delayed sampling in ProbZelus may be found in Baudart et al. [2020].

$$
\begin{array}{rcl}
\{\!|c|\!\}^{\text{init}} & = & () \\
\{\!|c|\!\}^{\text{step}}_{\gamma}(m, \boldsymbol{g}, w) & = & (c, m, \boldsymbol{g}, w) \\
\{\!|x|\!\}^{\text{init}} & = & () \\
\{\!|x|\!\}^{\text{step}}_{\gamma}(m, \boldsymbol{g}, w) & = & (\gamma(x), m, \boldsymbol{g}, w) \\
\{\!|\texttt{last } x|\!\}^{\text{init}} & = & () \\
\{\!|\texttt{last } x|\!\}^{\text{step}}_{\gamma}(m, \boldsymbol{g}, w) & = & (\gamma(x\_\texttt{last}), m, \boldsymbol{g}, w) \\
\{\!|op(e)|\!\}^{\text{init}} & = & \{\!|e|\!\}^{\text{init}} \\
\{\!|op(e)|\!\}^{\text{step}}_{\gamma}(m, \boldsymbol{g}, w) & = & \textit{let } v, m', \boldsymbol{g'}, w' = \{\!|e|\!\}^{\text{step}}_{\gamma}(m, \boldsymbol{g}, w) \textit{ in} \\
 & & (\mathsf{app}(op, v), m', \boldsymbol{g'}, w') \\
\{\!|\texttt{sample}(e)|\!\}^{\text{init}} & = & \{\!|e|\!\}^{\text{init}} \\
\{\!|\texttt{sample}(e)|\!\}^{\text{step}}_{\gamma}(m, \boldsymbol{g}, w) & = & \textit{let } \mu, m', \boldsymbol{g'}, w' = \{\!|e|\!\}^{\text{step}}_{\gamma}(m, \boldsymbol{g}, w) \textit{ in} \\
 & & \textit{let } X, \boldsymbol{g''} = \textsc{assume}(\mu, \boldsymbol{g'}) \textit{ in} \\
 & & (X, m', \boldsymbol{g''}, w') \\
\{\!|\texttt{observe}(e_\mu, e_v)|\!\}^{\text{init}} & = & (\{\!|e_\mu|\!\}^{\text{init}}, \{\!|e_v|\!\}^{\text{init}}) \\
\{\!|\texttt{observe}(e_\mu, e_v)|\!\}^{\text{step}}_{\gamma}((m_\mu, m_v), \boldsymbol{g}, w) & = & \textit{let } \mu, m'_\mu, \boldsymbol{g_\mu}, w_1 = \{\!|e_\mu|\!\}^{\text{step}}_{\gamma}(m_\mu, \boldsymbol{g}, w) \textit{ in} \\
 & & \textit{let } X, \boldsymbol{g_x} = \textsc{assume}(\mu, \boldsymbol{g_\mu}) \textit{ in} \\
 & & \textit{let } v, m'_v, \boldsymbol{g_v}, w_2 = \{\!|e_v|\!\}^{\text{step}}_{\gamma}(m_v, \boldsymbol{g_x}, w_1) \textit{ in} \\
 & & \textit{let } v', \boldsymbol{g'_v} = \textsc{value}(v, \boldsymbol{g_v}) \textit{ in} \\
 & & \textit{let } \boldsymbol{g'}, s = \textsc{observe}(X, v', \boldsymbol{g'_v}) \textit{ in} \\
 & & ((), (m'_\mu, m'_v), \boldsymbol{g'}, s * w_2) \\
\{\!|\texttt{value}(e)|\!\}^{\text{init}} & = & \{\!|e|\!\}^{\text{init}} \\
\{\!|\texttt{value}(e)|\!\}^{\text{step}}_{\gamma}(m, \boldsymbol{g}, w) & = & \textit{let } X, m', \boldsymbol{g'}, w' = \{\!|e|\!\}^{\text{step}}_{\gamma}(m, \boldsymbol{g}, w) \textit{ in} \\
 & & \textit{let } v, \boldsymbol{g''} = \textsc{value}(X, \boldsymbol{g'}) \textit{ in} \\
 & & (v, m', \boldsymbol{g''}, w')
\end{array}
$$

$$
\begin{array}{rcl}
\left\{\!\!\left|\begin{array}{l} e \texttt{ where rec init } x_1 = c_1 \texttt{ and } \ldots \texttt{ and init } x_k = c_k \\ \qquad\qquad \texttt{and } y_1 = e_1 \texttt{ and } \ldots \texttt{ and } y_n = e_n \end{array}\right|\!\!\right\}^{\text{init}} & & \\
 & = & ((c_1, \ldots, c_k), (\{\!|e_1|\!\}^{\text{init}}, \ldots, \{\!|e_n|\!\}^{\text{init}}), \{\!|e|\!\}^{\text{init}}) \\
\left\{\!\!\left|\begin{array}{l} e \texttt{ where rec init } x_1 = c_1 \texttt{ and } \ldots \texttt{ and init } x_k = c_k \\ \qquad\qquad \texttt{and } y_1 = e_1 \texttt{ and } \ldots \texttt{ and } y_n = e_n \end{array}\right|\!\!\right\}^{\text{step}}_{\gamma} & & (((v_1, \ldots, v_k), (m_1, \ldots, m_n), m), \boldsymbol{g}, w) \\
 & = & \textit{let } \gamma_1 = \gamma[v_1/x_1\_\texttt{last}] \textit{ in} \\
 & & \ldots \\
 & & \textit{let } \gamma_k = \gamma[v_k/x_k\_\texttt{last}] \textit{ in} \\
 & & \textit{let } v_1, m'_1, \boldsymbol{g_1}, w_1 = \{\!|e_1|\!\}^{\text{step}}_{\gamma_k}(m_1, \boldsymbol{g}, w) \textit{ in} \\
 & & \textit{let } \gamma'_1 = \gamma_k[y_1/v_1] \textit{ in} \\
 & & \ldots \\
 & & \textit{let } v_n, m'_n, \boldsymbol{g_n}, w_n = \{\!|e_n|\!\}^{\text{step}}_{\gamma'_{n-1}}(m_n, \boldsymbol{g_{n-1}}, w_{n-1}) \textit{ in} \\
 & & \textit{let } \gamma'_n = \gamma'_{n-1}[y_n/v_n] \textit{ in} \\
 & & \textit{let } v, m', \boldsymbol{g'}, w' = \{\!|e|\!\}^{\text{step}}_{\gamma'_n}(m, \boldsymbol{g_n}, w_n) \textit{ in} \\
 & & (v, ((\gamma'_n(x_1), \ldots, \gamma'_n(x_k)), (m'_1, \ldots, m'_n), m'), \boldsymbol{g'}, w')
\end{array}
$$

Fig. 6. Delayed sampling semantics of ProbZelus programs. These semantics make use of the ASSUME, VALUE, and OBSERVE functions, which are the common interface for delayed sampling and semi-symbolic inference. Pairs are automatically lifted to n-ary arguments in operator application. Differences with Figure 5 are highlighted.

Following Murray et al. [2018], we define the semantics in terms of a *symbolic interface*, a set of functions that can be implemented to provide either delayed sampling, as presented in Baudart et al. [2020]; Murray et al. [2018], or semi-symbolic inference, as presented in Section 4.

Definition 3.1 (Symbolic Interface). The *symbolic interface* consists of three functions that manipulate a symbolic state g. The symbolic state g is a data structure that only interacts with the semantics through these functions.

- $X, g' = \textsc{assume}(\mu, g)$ returns a pair of a new random variable X and a new symbolic state g' that has X bound to a symbolic representation of the input distribution μ.
- $v, g' = \textsc{value}(X, g)$ samples the input random variable and returns the sampled value and the new symbolic state.
- $g', s = \textsc{observe}(X, v, g)$ conditions the model on the fact that the input random variable X takes on the value v, and returns the new state and the *score* (i.e., the value of the probability density) of v under the marginal distribution of X.

For the purpose of this section, the symbolic state g is abstract. Different implementations provide either delayed sampling (e.g. Murray et al. [2018]) or semi-symbolic inference (Section 4).

Semantics. In Figure 6, we redefine the semantics of Figure 5 to use delayed sampling constructs, by means of the symbolic interface. The semantics of an expression $\{\!|e|\!\}_\gamma^{\text{step}}$ is now a function that takes in a memory m, a symbolic state g, and a weight w, and returns a symbolic value, an updated memory, a new symbolic state, and an updated weight. The definition is similar to that in Figure 5 except that 1) the symbolic state g is threaded throughout the semantics, 2) an operator application yields a symbolic term $\mathsf{app}(op, v)$ for the application of op to the value v, and 3) the semantics of `sample`, `observe`, and `value` make use of the symbolic interface functions.

Inference. Inference under delayed sampling (DS) is also defined as a transformation on ProbZelus expressions. It is defined as follows:

$$\begin{aligned}
[\![\texttt{infer}_{DS}(N, e)]\!]^{\text{init}} &= \delta_{(\{\!|e|\!\}^{\text{init}}, \{\})} \\
[\![\texttt{infer}_{DS}(N, e)]\!]_\gamma^{\text{step}}(m) &= \mathit{let} \begin{bmatrix} d_i, m_i', g_i', w_i' = \mathit{let}\ m_i, g_i = \textsc{draw}(m)\ \mathit{in} \\ \qquad\qquad \mathit{let}\ e_i, m_i', g_i', w_i' = \{\!|e|\!\}_\gamma^{\text{step}}(m_i, g_i, 1)\ \mathit{in} \\ \qquad\qquad \textsc{distribution}(e_i, g_i'), m_i', g_i', w_i' \end{bmatrix}_{1 \le i \le N} \\
&\quad \mathit{let}\ \mu = \lambda U.\ \Sigma_{i=1}^{N} \overline{w_i} * d_i(\pi_1(U)) * \delta_{(m_i', g_i')}(\pi_2(U))\ \mathit{in} \\
&\quad (\pi_{1*}(\mu), \pi_{2*}(\mu))
\end{aligned}$$

The definition is similar to the particle-filter definition of `infer`, but additionally threads the symbolic state g through the execution. The initial memory is a Delta distribution of the initial memory of e and the empty symbolic state $\{\}$, and the output is a pair of a distribution of outputs and a distribution of memories and symbolic states. The step function of each particle can return a symbolic expression containing un-sampled variables, and the DISTRIBUTION function returns the distribution corresponding to the symbolic expression e_i without altering the symbolic state.

Correctness. The operations in the symbolic interface must be designed such that the symbolic state represents the same distribution as the particle filter. Lundén [2017] proposed a sufficient condition for the delayed sampling symbolic state g to be correct. Here we summarize the general idea behind this correctness condition and restate it for the symbolic interface in general. We leave the full formalization of correctness of the symbolic interface to the appendix of the extended version of the paper [Atkinson et al. 2022b].

Let $\Pr(\mathcal{V})$ be the distribution induced by ASSUME statements on the set of random variables $\mathcal{V}$. From this, for each random variable X, there is a conditional distribution $\Pr(X \mid \hat{V} = \hat{v}, \hat{O} = \hat{o})$ that

$$D ::= \mathcal{N}(E, E) \mid \text{Bernoulli}(E) \mid \beta(E, E) \mid \delta(E)$$

$$E ::= \texttt{app}(Op, E^*) \mid r \mid c \mid X$$

$$Op ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{sqrt} \mid \texttt{ite} \mid \texttt{=} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=}$$

$$r \in \mathbb{R}, c \in \mathbb{Z}, X \in \mathcal{V}$$

Fig. 7. Grammar of symbolic expressions.

specifies the *marginal distribution* of X conditioned on the random variables in $\hat{V}$ and $\hat{O}$ taking on the values $\hat{v}$ and $\hat{o}$, respectively. We assume that $\hat{V}$ is the set of random variables that have previously been passed to VALUE and $\hat{O}$ is the set of random variables that have been passed to OBSERVE. According to Lundén [2017], the symbolic interface functions are correct if:

- VALUE(X, g) draws a sample from $\Pr(X \mid \hat{V} = \hat{v}, \hat{O} = \hat{o})$, and
- OBSERVE(X, r, g) evaluates the probability density of $\Pr(X \mid \hat{V} = \hat{v}, \hat{O} = \hat{o})$ at the value r.

We provide a more detailed formalism that precisely specifies $\Pr(X \mid \hat{V} = \hat{v}, \hat{O} = \hat{o})$ in the appendix of the extended version [Atkinson et al. 2022b].

4 SEMI-SYMBOLIC INFERENCE

In this section, we present semi-symbolic inference as a series of definitions that implement the symbolic interface from Definition 3.1. The section begins by defining the syntax of symbolic expressions and giving a definition of the symbolic state. It then implements the functions of the symbolic interface through a series of definitions:

- Section 4.1 defines the SWAP operation, the core building block of semi-symbolic inference that changes the dependency order between two random variables in the symbolic state.
- Section 4.2 presents the EVAL and INTERVENE helper functions, which evaluate symbolic expressions and perform interventions on the symbolic state, respectively.
- Section 4.3 presents the HOIST operation, which combines a sequence of swap operations to turn a given random variable into a root.
- Section 4.4 shows how to implement the interface in Definition 3.1.
- Section 4.5 shows that these definitions satisfy correctness properties.

Symbolic Expressions. Figure 7 gives the grammar of symbolic expressions used by semi-symbolic inference. The grammar defines a set of symbolic distributions D, consisting of a Gaussian distribution (denoted $\mathcal{N}$), a Bernoulli distribution, a Beta distribution, and a Delta distribution. The parameters for each distribution type are elements of a grammar of expressions E. An expression is either a real number r, an integer c, a random variable X, or a n-ary operator Op applied recursively to sub-expressions: $\texttt{app}(Op, E^*)$. The operators consist of standard arithmetic operators (`+`, `-`, `*`, and `/`), the square root operator `sqrt`, a conditional operator `ite`, and standard comparison operators (`=`, `!=`, `<`, and `<=`). The interpretation of the conditional operator is that $\texttt{app}(\texttt{ite}, e_i, e_t, e_e)$ returns the value of the expression e_t if the condition expression e_i evaluates to true, and otherwise returns the value of the expression e_e.

For example, $\mathcal{N}(X_v - 2 * X_o, 1)$, the distribution of X_l in Figure 3b, would be encoded in the grammar of Figure 7 as

$$\mathcal{N}(\texttt{app(-,}X_v\texttt{,app(*,2,}X_o\texttt{))}, 1)$$

Definition 4.1 (Symbolic State). We define the *symbolic state* of the semi-symbolic runtime inference system as a finite mapping from random variables to distributions $g \in G = \mathcal{V} \mapsto D$.

For example, the symbolic state depicted in Figure 3b is the finite map

$$g_{example} = \{X_v \mapsto \mathcal{N}(0, 2500);\ X_o \mapsto \mathcal{N}(0, 2500);\ X_l \mapsto \mathcal{N}(\texttt{app(-,}X_v\texttt{,app(*,2,}X_o\texttt{))}, 1)\}$$

4.1 Swapping Random Variables

Algorithm 1 The definition of SWAP, incorporating exact inference for Gaussian distributions, Beta distributions, and Bernoulli distributions. Note that this algorithm uses shorthand notation for symbolic expression construction, e.g. $e_1 + e_2$ constructs a symbolic addition term $\texttt{app(+,}e_1\texttt{,}e_2\texttt{)}$.

function SWAP(X_1, X_2, g)
  switch $g(X_1)$, $g(X_2)$ **do**
    case $\mathcal{N}(\mu_0, var_0), \mathcal{N}(\mu, var)$ **if** AFFINE$(\mu, X_1) = (a, b) \wedge$ CONST(var_0, var)
      $\mu_0', var_0' \leftarrow (a * \mu_0) + b, (a * a) * var_0$
      $var_0'', \mu_0'' \leftarrow \frac{1}{\frac{1}{var_0} + \frac{1}{var}}, \left(\frac{\mu_0'}{var_0'} + \frac{X_2}{var}\right) * var_0''$
      return $(g[X_1 \mapsto \mathcal{N}\left(\frac{\mu_0'' - b}{a}, \frac{var_0''}{a * a}\right)]$
        $[X_2 \mapsto \mathcal{N}\left(\mu_0', var_0' + var\right)]$, **true**)
    case $\beta(a, b)$, Bernoulli(X_1)
      return $(g[X_1 \mapsto \beta(a + (\texttt{ite(}\ X_2, 1, 0\ \texttt{)}), b + (\texttt{ite(}\ X_2, 0, 1\ \texttt{)}))]$
        $[X_2 \mapsto \text{Bernoulli}\left(\frac{a}{a+b}\right)]$, **true**)
    case Bernoulli(p_1), Bernoulli(p_2)
      $p_1' \leftarrow \frac{p_1 * \texttt{ite(}\ X_2, p_2, (1 - p_2)\ \texttt{)}[X_1 \leftarrow 1]}{\texttt{ite(}\ X_2, p_2', (1 - p_2')\ \texttt{)}}$
      $p_2' \leftarrow (p_1 * p_2[X_1 \leftarrow 1]) + ((1 - p_1) * p_2[X_1 \leftarrow 0])$
      return $(g[X_1 \mapsto \text{Bernoulli}(p_1')]$
        $[X_2 \mapsto \text{Bernoulli}(p_2')]$, **true**)
    else
      return $(g$, **false**)
end function

The core operation of semi-symbolic inference is the *swap*. Swapping random variables changes the probabilistic dependence structure of the symbolic state without changing the overall distribution the symbolic state represents.

Requirements. To swap two random variables X_1 and X_2, the following must hold:

(1) X_1 is a parent of X_2 in the initial state, making X_2 a parent of X_1 in the new state.
(2) After the swap, all variables other than X_1 and X_2 have the same distributions as before.
(3) The symbolic state represents the same overall joint distribution before and after the swap.

Dependency Cycles. Furthermore, a swap is only legal in some circumstances because a swap may introduce new probabilistic dependencies between random variables. In general, a swap will introduce dependencies between each of X_1 and X_2 and all parents of either X_1 or X_2. A legal swap is one that does not create dependency cycles between random variables. The function CAN_SWAP(X_1, X_2, g) determines if the swap of X_1 and X_2 is legal in symbolic state g, i.e. whether or not swapping X_1 and X_2 will introduce a dependency cycle.

Swap Algorithm. Algorithm 1 defines the SWAP function for Gaussians, Beta distributions, and Bernoulli distributions. The function takes as input a parent random variable X_1, a child random variable X_2, and an initial state g. It returns a modified state and a boolean indicating whether or not a swap is possible using closed-form solutions known to the semi-symbolic inference system.

Note that there is a difference between a swap being *illegal* due to dependency cycles and being *impossible* due to the semi-symbolic inference system not identifying a closed-form solution. The function SWAP returns **false** if no closed-form solution exists and a swap is impossible, whereas CAN_SWAP returns **false** if the swap is illegal due to dependency cycles.

Also note that for ease of presentation, Algorithm 1 uses shorthand notation for symbolic expression construction. In particular, Algorithm 1 uses $e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$, $\frac{e_1}{e_2}$, and $\texttt{ite}(e_i, e_t, e_e)$ as shorthand for $\texttt{app}(+, e_1, e_2)$, $\texttt{app}(-, e_1, e_2)$, $\texttt{app}(*, e_1, e_2)$, $\texttt{app}(/, e_1, e_2)$, and $\texttt{app}(\texttt{ite}, e_i, e_t, e_e)$, respectively. We stress that Algorithm 1 performs no numerical computation, and that these operations are constructors for symbolic expressions.

The implementation of SWAP is separated into three cases:

Gaussian. The first case occurs when both X_1 and X_2 are Gaussian-distributed, and also requires the variance of each distribution to be constant (i.e., to not depend on any random variables) and for the mean of X_2 to be expressible as an affine function of X_1. We express the analysis for this condition as AFFINE$(\mu, X_1) = (a, b)$, which means that μ can be written as $a * X_1 + b$ where a and b are themselves symbolic expressions that may contain random variables other than X_1.

This case applies the following rules for Gaussian distributions: 1) the symbolic expressions μ_0' and var_0' represent a linear transformation of the Gaussian distribution of X_1, which is also a Gaussian distribution; 2) the expressions μ_0'', var_0'' are computed from the rules for conjugate priors [Fink 1997]; 3) the final expressions for the new distributions of X_1 and X_2 incorporate both the inverse linear transformation to the one used to generate μ_0' and var_0'.

Using these rules, the symbolic state is updated and returned. The notation $g[X \mapsto d]$ means that the symbolic state g is updated with the random variable X remapped to the distribution d.

Example. The example in Section 2 makes extensive use of swaps between Gaussian distributions. Here, we briefly discuss how SWAP executes to produce the distribution for X_l in Figure 3c. The runtime inference system produces the symbolic state depicted in Figure 3c from one depicted in Figure 3b by swapping the random variable X_v with its child X_l. Formally, it executes SWAP$(X_v, X_l, g_{example})$, where $g_{example}$ is the symbolic state depicted in Figure 3b.

This falls into the Gaussian case, with $\mu_0 = 0$, $var_0 = 2500$, $\mu = \texttt{app}(-, X_v, \texttt{app}(*, 2, X_o))$, and $var = 1$. The affine analysis concludes that the expression for μ can be written as $\mu = 1 * X_v + (-2 * X_o)$, and thus produces $a = 1$ and $b = \texttt{app}(*, -2, X_o)$. Next, the SWAP function uses these values of μ_0, var_0, a, and b to compute that $\mu_0' = \texttt{app}(+, \texttt{app}(*, 1, 0), \texttt{app}(*, -2, X_o))$ and $var_0' = \texttt{app}(*, \texttt{app}(*, 1, 1), 2500)$. Ultimately, using these values of μ_0', var_0', and var, the SWAP function will return a symbolic state in which X_l maps to the symbolic expression

$$\mathcal{N}(\texttt{app}(+, \texttt{app}(*, 1, 0), \texttt{app}(*, -2, X_o)), \texttt{app}(+, \texttt{app}(*, \texttt{app}(*, 1, 1), 2500), 1))$$

Similarly, the SWAP function will compute the symbolic expressions μ_0'' and var_0'' and use them to produce a new distribution for X_v.

Beta-Bernoulli. The second case occurs when X_1 is distributed according to a Beta distribution and X_2 is distributed according to a Bernoulli distribution. This is an instance of the Beta-Bernoulli conjugate model, with its own rules and transformations [Fink 1997].

Bernoulli-Bernoulli. The third case applies when both X_1 and X_2 have Bernoulli distributions. This case applies the rules of discrete probability to produce a new symbolic state. In particular, it sums out the random variable X_1 to produce an expression for the new probability p_2' of X_2. It then uses Bayes' rule to produce a new expression for X_1. Note that this definition uses the notation $e_1[X \leftarrow e_2]$ to denote expression e_1 with expression e_2 substituted for the random variable X.

Extensibility. In Algorithm 1, if none of these cases apply, SWAP returns the existing state and the value false. However, the definition of SWAP is extensible and can handle additional cases as necessary. In our implementation, we have added cases for multivariate Gaussians and categorical distributions. Additional cases for more conjugate priors (e.g. those in Fink [1997]) may follow the example of the Beta-Bernoulli conjugacy from the second case of Algorithm 1.

4.2 Evaluation and Intervention

Evaluation. To simplify symbolic expressions introduced by operator applications, semi-symbolic inference makes use of a *partial evaluator*: a function $\textsc{eval}(e, g) \in E \times G \mapsto E$ that takes in an expression and a symbolic state and produces a new expression that is evaluated to a constant if possible. The partial evaluator is defined recursively on the structure of expressions. For example, on addition, the evaluator proceeds according to:

$$\textsc{eval}(\mathsf{app}(+, e_1, e_2), g) = \begin{cases} r_1 + r_2 & \textsc{eval}(e_1, g) = r_1 \wedge \textsc{eval}(e_2, g) = r_2 \\ \mathsf{app}(+, \textsc{eval}(e_1, g), \textsc{eval}(e_2, g)) & \text{otherwise} \end{cases}$$

This equation specifies that if both subexpressions e_1 and e_2 evaluate to real numbers, the partial evaluator performs real-number arithmetic and returns the result. Otherwise, it recursively partially evaluates the subexpressions and leaves the result symbolic.

For example, the variance of the distribution for X_l in Figure 3c is the symbolic expression $\mathsf{app}(+, 2500, 1)$. Letting $g'_{example}$ be a depiction of the symbolic state in Figure 3c, this expression evaluates to $\textsc{eval}(\mathsf{app}(+, 2500, 1), g'_{example}) = 2501$. By contrast, the mean of this distribution is $\mathsf{app}(-, 0, \mathsf{app}(*, 2, X_o))$, which cannot be evaluated any further, which means that $\textsc{eval}(\mathsf{app}(-, 0, \mathsf{app}(*, 2, X_o)), g'_{example}) = \mathsf{app}(-, 0, \mathsf{app}(*, 2, X_o))$.

One unique feature of the semi-symbolic partial evaluator is how it handles Delta distributions. In particular, the evaluator leverages the fact that Delta-distributed variables must only take on one value and therefore can be substituted like normal program variables. The partial evaluator handles Deltas according to the equation:

$$\textsc{eval}(X, g) = \begin{cases} \textsc{eval}(e, g) & g(X) = \delta(e) \\ X & \text{otherwise} \end{cases}$$

This equation states that if a random variable has a Delta distribution, then the partial evaluator evaluates the Delta's internal expression and returns the result. For example, in Figure 3e, the distribution for X_o is only a function of constants and X_l, and X_l has a Delta distribution. Formally, if $g'''_{example}$ is the symbolic state depicted in Figure 3e, then $\textsc{eval}(g'''_{example}(X_o), g'''_{example}) = \mathcal{N}(0.4, 500)$.

We also define a version of the partial evaluator that updates the symbolic state, which we write $\textsc{eval}^*(X, g) \in \mathcal{V} \times G \mapsto G$. This evaluator updates each parameter of a given random variable's distribution to be its partially evaluated counterpart. For example, for Gaussian distributions:

$$\textsc{eval}^*(X, g) = g[X \mapsto \mathcal{N}(\textsc{eval}(\mu, g), \textsc{eval}(var, g))] \text{ if } g(X) = \mathcal{N}(\mu, var)$$

Intervention. An *intervention* replaces a root node with a Delta distribution. Intervention is defined as the function $\textsc{intervene} \in \mathcal{V} \times \mathbb{V} \times G \mapsto G$ that takes in a random variable, the intervention value, the symbolic state, and returns the new symbolic state. $\mathbb{V}$ denotes the sample space of the random variable. The function is defined as follows:

$$\textsc{intervene}(X, r, g) = g[X \mapsto \delta(r)]$$

In Figure 3 the transition from 3d to 3e depicts the execution of $g'''_{example} = \textsc{intervene}(X_I, -1, g''_{example})$, where $g''_{example}$ is the symbolic state depicted in Figure 3d and $g'''_{example}$ is the one depicted in Figure 3e.

4.3 Hoisting

Swaps are composed together to support the implementation of the operations from Definition 3.1 by an operation called HOIST. The objective of hoisting a random variable is to update the symbolic state so that the variable in question is a *root* that depends on no other random variables.

Preliminaries. The HOIST operation depends on functions manipulating lists of random variables:

- $\textsc{get\_parents}(X, g)$ returns a list of parents of X in g, i.e. a list of random variables that are free variables in the expression of the distribution of X.
- $\textsc{topo\_sort}(\hat{X}, g)$ sorts a list of random variables in topological order according to the parent-child relation of g.
- $\textsc{reverse}(\hat{X})$ reverses a list of random variables.

Hoisting Algorithm. Algorithm 2 defines $\textsc{hoist} \in \mathcal{V} \times G \mapsto G$. It makes use of a helper function called HOIST_HELPER that takes a set of root variables given by the parameter `roots`. HOIST_HELPER's objective is to turn the input variable X_{cur} into a root variable, except that variables that are in `roots` do not count for the purpose of determining whether or not X_{cur} is a root variable.

To do so, HOIST_HELPER first recursively calls itself on all parents of X_{cur} in topological order. The TOPO_SORT function yields an order such that the first element of the resulting `parents` list has no ancestors that are also in `parents`. Then, because on subsequent recursive calls, all previously visited parents are added to `roots` and are thus excluded from being hoisted or swapped, later elements of `parents` will be descendants of earlier elements of `parents` after the recursive calls.

After the recursion, the function iterates through all parents in reverse topological order. This reverse ordering ensures that the algorithm can always swap each parent with X_{cur} without creating a cycle. This is because in order to create a cycle, the algorithm would need to swap the child node with a parent whose descendant is the child node itself. By iterating in reverse topological order, any other parent that would enable such a path to exist in the dependency graph must already have been swapped and therefore does not have X_{child} as a dependency. The CAN_SWAP assertion encapsulates the above argument that the algorithm does not create cycles. In the appendix of the extended paper [Atkinson et al. 2022b], we formally prove that this assertion always passes at runtime.

In case the distributions are not conjugate and therefore a swap is impossible, as indicated by the `conjugate` variable being **false**, the algorithm throws an exception that is caught at the outermost level. It then calls the VALUE function (from Definition 3.1 and defined below in Section 4.4) to replace the parent variable with a random sample. It next evaluates the child random variable to

Algorithm 2 Hoisting a random variable to be a root depending on no other random variables.

```
function HOIST_HELPER(X_cur, roots, g)
    parents ← TOPO_SORT(GET_PARENTS(X_cur, g))
    roots' ← roots; g' ← g
    for X_par ∈ parents do
        if X_par ∉ roots then
            g' ← HOIST_HELPER(X_par, roots', g'); roots' ← X_par :: roots'
        end if
    end for
    g'' ← g'
    for X_par ∈ REVERSE(parents) do
        if X_par ∉ roots then
            assert CAN_SWAP(X_par, X_cur, g''); (g'', conjugate) ← SWAP(X_par, X_cur, g'')
            if not conjugate then
                throw (X_par, X_cur)
            end if
        end if
    end for
    return g''
end function
function HOIST(X_in, g)
    try
        return HOIST_HELPER(X_in, {}, g)
    catch (X_par, X_child)
        (_, g') ← VALUE(X_par, g); g'' ← EVAL*(X_child, g'); return HOIST(X_in, g'')
    end try
end function
```

eliminate the resulting Delta distribution (thus eliminating the need to perform this swap) and finally restarts the hoisting process from the beginning.

Example. In the example in Figure 3, Figures 3b–3d depict the result of executing $\textsc{hoist}(X_l, g_{example})$ where $g_{example}$ is the symbolic state depicted in Figure 3b. The function HOIST immediately calls HOIST_HELPER with $X_{\text{cur}} = X_l$ and roots = {}. The first stage of HOIST_HELPER is to recursively call HOIST_HELPER on its ancestors in topological order. The ancestors of X_l are X_o and X_v, and as they have no dependencies between each other, any order is a valid topological order. Furthermore, because neither X_o nor X_v has a parent, calling HOIST_HELPER on these variables has no effect. After all recursive calls, $\textsc{hoist\_helper}(X_l, \{\})$ swaps X_l with both X_o and X_v. In the example in Figure 3, we assume that HOIST_HELPER first swaps with X_v (Figure 3c) and then with X_o (Figure 3d). After these swaps, HOIST returns the resulting symbolic state.

4.4 Symbolic Interface

In this section, we describe how to implement the symbolic interface presented in Definiton 3.1.

Assume. The operation $\textsc{assume} \in D \times G \mapsto \mathcal{V} \times G$ takes in a distribution and a current state, and returns a new random variable and the updated state. It is defined as follows:

$$\textsc{assume}(d, g) = (X_{\text{new}}, g[X_{\text{new}} \mapsto d]) \text{ where } X_{\text{new}} \text{ is not assigned in } g$$

This definition specifies that the ASSUME function returns a fresh random variable, and that it updates the symbolic state to have the new random variable point to the input distribution.

Value. The function VALUE $\in \mathcal{V} \times G \mapsto G \times \mathbb{V}$ instructs the inference system to replace a particular random variable with a sample from its marginal distribution and return the resulting sample. It is defined as follows, and is mutually recursive with the HOIST operation:

$$\begin{aligned}\textsc{value}(X, g) = \; & \textit{let } g' = \textsc{hoist}(X, g) \textit{ in} \\ & \textit{let } g'' = \textsc{eval}^*(X, g') \textit{ in} \\ & \textit{let } v = \textsc{draw}(g''(X)) \textit{ in} \\ & (v, \textsc{intervene}(X, v, g''))\end{aligned}$$

This definition specifies that the VALUE function first hoists the variable X, guaranteeing that in the resulting symbolic state it is a root. After hoisting, it evaluates the distribution of the random variable that was just hoisted, producing a closed-form distribution that it then samples from. It further updates the symbolic state by intervention.

Observe. The function OBSERVE $\in \mathcal{V} \times \mathbb{V} \times G \mapsto G \times \mathbb{R}$ conditions the symbolic state on the input random variable taking on the input value and returns the updated state and a score which corresponds to how likely this value is according to its marginal density. It is defined as follows:

$$\begin{aligned}\textsc{observe}(X, v, g) = \; & \textit{let } g' = \textsc{hoist}(X, g) \textit{ in} \\ & \textit{let } g'' = \textsc{eval}^*(X, g') \textit{ in} \\ & \textit{let } s = \textsc{score}(g''(X), v) \textit{ in} \\ & \textit{let } g''' = \textsc{intervene}(X, v, g'') \textit{ in} \\ & (g''', s)\end{aligned}$$

This definition specifies that the OBSERVE operation first hoists the input random variable and fully evaluates its distribution's parameters as it is now a root. It then calculates the probability density of the variable's marginal distribution using the SCORE function. It returns a combination of the new state, obtained by intervening to condition the symbolic state on the fact that the random variable takes on the specified value, and the new weight.

4.5 Correctness

In this section, we formalize the correctness of semi-symbolic inference. We present the key ideas necessary to specify and prove the correctness of each operation defined in the previous sections, leaving the full formalization to the appendix of the extended paper [Atkinson et al. 2022b].

Swap Correctness. We first present the correctness requirements for SWAP. The SWAP function is correct if it preserves the joint distribution of all random variables. This statement requires defining $[\![g]\!]$, the joint distribution that is the meaning of the symbolic state g . We defer this definition to Atkinson et al. [2022b], and formalize the correctness property as follows:

LEMMA 4.2 (SWAP PRESERVATION). *If* $(g', \_) = \textit{swap}(X_1, X_2, g)$, *then* $[\![g']\!] = [\![g]\!]$.

The proof of the theorem is also in Atkinson et al. [2022b]. The key idea is to incorporate known results about conjugate priors [Fink 1997].

Evaluation Correctness. Similarly, the EVAL* operation must also preserve the joint distribution:

LEMMA 4.3 (EVAL* CORRECTNESS). *If* $g' = \textit{eval}^*(X, g)$, *then* $[\![g']\!] = [\![g]\!]$.

Intervention Correctness. The key idea for INTERVENE is that if the random variable passed to it is a root, then it should perform conditioning. In this theorem, we use the notation $\Pr_g$ to refer to probability distributions implied by the symbolic state g. We construct conditional probability distributions from the overall joint distribution $[\![g]\!]$ using standard techniques; see Atkinson et al. [2022b] for details. We formalize this idea as follows:

LEMMA 4.4 (INTERVENE CORRECTNESS). *If X is a root in g, and $g' = \textit{INTERVENE}(X, r, g)$, then for any subset $\mathcal{V}'$ of random variables mapped in g, $\Pr_{g'}(\mathcal{V}') = \Pr_g(\mathcal{V}' \mid X = r)$.*

Hoist Correctness. We next present the correctness of the subroutine HOIST_HELPER. This subroutine must preserve the semantics of the overall joint distribution. It is also designed to turn its input variable into a root, with the exception of variables in `roots`. We formalize this as follows:

LEMMA 4.5 (HOIST_HELPER CORRECTNESS). *If $g' = \textit{HOIST\_HELPER}(X_{\text{cur}}, \texttt{roots}, g)$, and no exceptions are thrown, $[\![g']\!] = [\![g]\!]$, and in g', X_{cur} is a root, except it may depend on variables in* `roots`.

Next, we establish the correctness of HOIST. Due to the fact that HOIST and VALUE are mutually recursive, we combine their correctness properties into a single theorem. The correctness theorem for HOIST states that hoist turns the input random variable into a root and preserves the symbolic state, except that the new symbolic state will be conditioned on any variables that needed to be sampled due to lack of conjugacy. The correctness theorem for VALUE states that it produces a random sample from the appropriate marginal distribution, and conditions the new symbolic state on this and any other sampled variables.

THEOREM 4.6 (HOIST AND VALUE CORRECTNESS). *If $g' = \textit{HOIST}(X_{\text{in}}, g)$, then $\Pr_{g'}(\mathcal{V}) = \Pr_g(\mathcal{V} \mid \hat{V} = \hat{v})$, where $\hat{V}$ is the set of variables sampled during the execution of HOIST and $\hat{v}$ is the corresponding set of sampled values. Furthermore, after executing HOIST, X_{in} is a root in g'.*

Also, if $(g', v) = \textit{VALUE}(X, g)$, then v is a sample from $\Pr_g(X \mid \hat{V} = \hat{v})$ and $\Pr_{g'}(\mathcal{V}) = \Pr_g(\mathcal{V} \mid X = v, \hat{V} = \hat{v})$, where $\hat{V}$ and $\hat{v}$ are as above.

Observe Correctness. The final operation whose correctness we must ensure is OBSERVE. This operation is correct if it conditions the symbolic state on the input variable being equal to the input value. It further must return the density of the random variable's marginal distribution evaluated at the input value. We formalize this as follows:

THEOREM 4.7 (OBSERVE CORRECTNESS). *If $(g', w) = \textit{OBSERVE}(X, r, g)$, then we have that $\Pr_{g'}(\mathcal{V}) = \Pr_g(\mathcal{V} \mid X = r, \hat{V} = \hat{v})$, where $\hat{V}$ and $\hat{v}$ are the random variables that may need to be sampled during the observation, and $\hat{v}$ are their sampled values. Furthermore, w is the density of $\Pr_g(X \mid \hat{V} = \hat{v})$.*

Overall Correctness. As we explain in Section 3.2, Lundén [2017] provides conditions on the symbolic interface that are sufficient to ensure the overall correctness of the inference algorithms. These conditions depend, in general, on the total sequence of interface operations performed. We provide a detailed formalism of the overall correctness of the semi-symbolic operations of the symbolic interface in the appendix of the extended paper [Atkinson et al. 2022b].

5 CLOSED-FAMILY PROPERTIES OF SEMI-SYMBOLIC INFERENCE

In this section, we discuss the properties of semi-symbolic inference on *closed families*. Closed families are sets of symbolic states on which the semi-symbolic operations from Section 4 are guaranteed to maintain a symbolic representation without falling back on sampling-based approximations. Developers can use this guarantee to reliably write probabilistic programs that the semi-symbolic runtime inference system will automatically implement as Rao-Blackwellized particle filters. In

this section, we define closed families and formalize their properties. The definition of a closed family that it should be closed under all legal swaps:

Definition 5.1 (Closed Family). A closed family C is a set of symbolic states such that if $g \in C$, and X_1, X_2 in g such that CAN_SWAP(X_1, X_2), then SWAP$(X_1, X_2, g) = (g', \textbf{true})$ and $g' \in C$.

For example, all the states depicted in Figure 3 are in the linear-Gaussian closed family we define below. This means that any legal swap we may want to execute on one of these states will be possible (i.e. will find available conjugate distributions), and furthermore will not modify the symbolic state in such a way that future swaps could be impossible. Moreover, this means that the hoisting we perform in Figures 3b–3d does not perform any random sampling.

We now explain two classes of distributions and show they are closed families: linear-Gaussian distributions and finite discrete distributions.

THEOREM 5.2 (LINEAR-GAUSSIAN CLOSED FAMILY). *The set of linear-Gaussian symbolic states, consisting of states such that a) all distributions in the state are Gaussian with constant variance (i.e., the variance does not depend on any random variables), and b) the mean of each Gaussian distribution is an affine function of other random variables, is a closed family.*

THEOREM 5.3 (FINITE DISCRETE CLOSED FAMILY). *The set of finite discrete symbolic states, consisting of states such that every distribution is a Bernoulli random variable, is a closed family.*

The key step to prove that the linear-Gaussian family is closed is showing that the new means for the distributions (i.e. $\frac{\mu_0'' - b}{a}$ and μ_0' in Algorithm 1) are affine and can be analyzed as such by the AFFINE analysis. This is a constraint of the AFFINE analysis's precision. In our implementation, we use a recursive analysis that meets this constraint, but we do not formalize it here. The remainder of the proofs of these theorems follow straightforwardly from the definition of SWAP.

RBPF Guarantee. The goal of closed families is to enable developers to control when random sampling happens over the course of semi-symbolic inference. In particular, developers want to ensure the runtime does not perform any hidden calls to VALUE through the catch clause of HOIST, which would occur when there is no conjugacy available to SWAP. To ensure this condition, we present the following Rao-Blackwellized particle filtering (RBPF) guarantee of the semi-symbolic implementation of the symbolic interface. This theorem states conditions under which variables ASSUMEd from within the closed family are guaranteed to be exact. In particular, all variables must either be in the closed family, or immediately sampled by being passed to VALUE.

THEOREM 5.4 (RBPF GUARANTEE). *Given a closed family C, let $\hat{X}_e$ be the set of random variables the developer wants to keep exact, and $\hat{X}_s = \mathcal{V} \setminus \hat{X}_e$ the variables the developer wants to sample. If,*

- *For all calls $X, g' =$ ASSUME(μ, g) such that $X \in \hat{X}_e$, if $g \in C$ then $g' \in C$.*
- *For all calls $X, g' =$ ASSUME(μ, g) such that $X \in \hat{X}_s$, X is immediately passed to VALUE, and μ does not depend on any variables in $\hat{X}_e$, then*

all variables in $\hat{X}_s$ will be sampled by VALUE, and no variable in $\hat{X}_e$ will ever be passed to VALUE.

Proof Sketch. The key step of the proof is to show that the symbolic state will be in C during the execution of HOIST. Then, by the definition of closed families, SWAP will always return true and HOIST_HELPER will never throw an error.

6 EVALUATION

In this section, we evaluate our implementation of semi-symbolic inference (SSI) in the ProbZelus streaming probabilistic programming language. We compare this new algorithm with the two

main inference algorithms previously implemented in ProbZelus [Baudart et al. 2020], particle filtering (PF) and delayed sampling (DS). We address the following research questions:

RQ1 Does the new algorithm provide more accurate results?
RQ2 How fast is the new algorithm?

6.1 Benchmarks

To compare the different inference algorithms, we use the original benchmarks of ProbZelus [Baudart et al. 2020] as well as two new examples based on realistic applications and one example that is presented as challenging in the original delayed sampling paper [Murray et al. 2018]. The benchmarks from ProbZelus are the following.

Beta-Bernoulli estimates the bias of a coin from a series of observations. The coin is modeled with a Bernoulli distribution with a Beta prior distribution on the probability of heads.

Gaussian-Gaussian estimates the mean and variance of a Gaussian distribution from a series of observations. This model has Gaussian priors on the mean and standard deviation.

Kalman-1D is a one-dimensional Kalman filter that estimates a hidden state from noisy observations. The state is modeled by a stream of random variables with Gaussian distributions centered on the previous state.

Outlier is a variation on the Kalman-1D example where the sensor occasionally produces completely invalid observations [Minka 2001].

Robot implements a robot controller that computes the commands to reach a target. The controller uses a probabilistic model to estimate the state of the robot (position, velocity, acceleration) using a noisy accelerometer, some sparse noisy GPS observations, and the previous command.

SLAM (Simultaneous Localization And Mapping) is the problem where a robot has to build a map of an unknown environment in which it travels while estimating its position. In the model adapted from Doucet et al. [2000], the agent evolves on a one dimensional discrete black-and-white map where the robot's wheels can slip and the color sensor can produce faulty observations.

MTT (Multi-Target Tracker) tracks a variable number of moving objects. The model is adapted from Murray and Schön [2018]. It estimates the path of each object from a set of noisy observations that do not identify the objects.

In addition to these benchmarks, we add the following more challenging models that rely on the closed-family guarantees from Section 5 to achieve good performance under SSI. By contrast, delayed sampling is unable to maintain exact inference on these benchmarks, and thus falls back on approximate sampling.

Tree is adapted from Lundén [2017] to illustrate a challenge with delayed sampling. If the symbolic graph forms a binary tree with at least three levels, to observe the leftmost variable and then the rightmost variable, the delayed sampling algorithm samples intermediate nodes and thus fails to produce exact results.

Wheels is the model presented in Section 2.

Delayed GPS is an extension of the *Robot* benchmark adapted from Solomon et al. [2012] to include a variable delay to the GPS observations. It is also discussed in more detail in Section 6.4.2. The maximal delay is bounded, so the model can keep a bounded history of the estimated positions to condition the model when a new observation happens.

6.2 Methodology

To evaluate the accuracy of the inference algorithms, each benchmark must define an accuracy metric. Following Baudart et al. [2020], for the *Robot* and *Delayed GPS* benchmarks, the accuracy

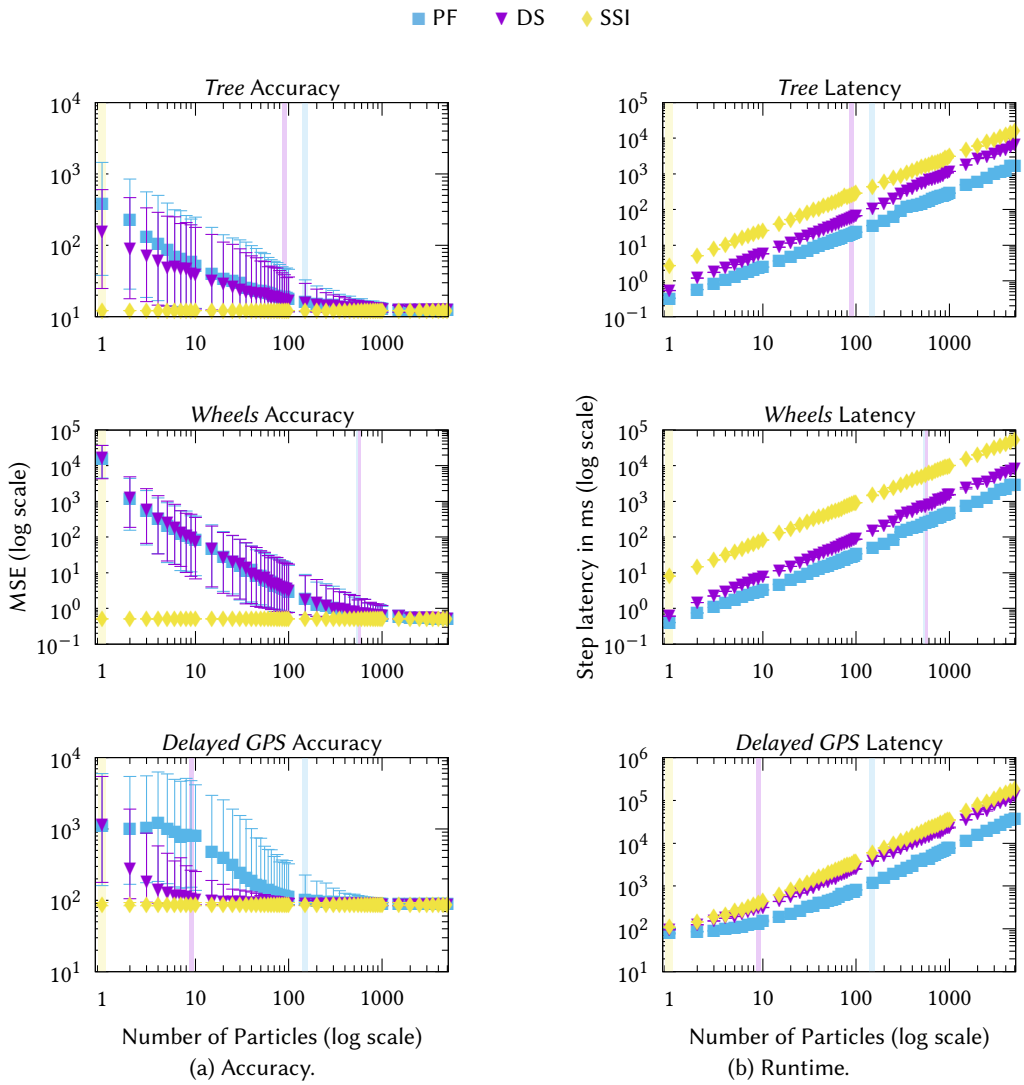


Fig. 8. Accuracy and runtime as a function of number of particles. The step latency is the delay between two successive time steps. Vertical bars indicates the number of particles where each algorithm reaches the target accuracy reported Table 1

metric is the *Linear-Quadratic Regulator* (LQR) loss [Sontag 2013]. For the *MTT* benchmark, the accuracy metric is a transformation of the *Multiple Object Tracking Accuracy* [Bernardin and Stiefelhagen 2008] such that it is defined on $[0, \infty)$: $\text{MOTA}^* = (1/\text{MOTA}) - 1$. For all the other benchmarks, the accuracy metric is the *Mean Squared Error* (MSE) of the inferred parameters compared to their exact values.

All the experiments were executed on a server with 64 Intel Xeon E5 CPUs (2.1GHz) and 128 GB of RAM. Each benchmark is executed with an increasing number of particles varying from 1 to 5000 on a fixed input stream of 500 time steps.

	PF		DS		SSI	
MODEL	# PART.	TIME (MS)	# PART.	TIME (MS)	# PART.	TIME (MS)
Beta-Bernoulli	200	23.05 (22.54-23.90)	✓ 1	0.28 (0.27-0.28)	✓ 1	0.65 (0.65-0.66)
Gaussian-Gaussian	3000	877.44 (689.04-890.68)	150	62.99 (61.69-65.86)	150	182.57 (181.40-183.54)
Kalman-1D	15	3.27 (3.26-3.28)	✓ 1	0.33 (0.33-0.34)	✓ 1	1.15 (1.14-1.15)
Outlier	700	222.27 (220.76-223.84)	65	43.81 (43.45-46.48)	65	125.88 (125.27-128.08)
Robot	85	771.32 (767.98-775.51)	✓ 1	91.44 (90.94-92.07)	✓ 1	96.40 (96.21-97.53)
SLAM		✗	800	2812.55 (2755.99-2853.89)	800	5649.30 (5619.59-5675.81)
MTT		✗	60	2889.11 (2615.76-3244.30)	60	4457.79 (4068.35-4996.20)
Tree	150	35.55 (35.41-35.68)	90	58.83 (58.55-59.74)	✓ 1	2.67 (2.66-2.70)
Wheels	550	246.48 (245.06-248.75)	550	699.12 (672.25-713.64)	✓ 1	8.04 (8.00-8.10)
Delayed GPS	150	1221.00 (1218.76-1230.67)	9	304.73 (303.17-306.31)	✓ 1	108.55 (108.02-109.07)

Table 1. For each benchmark, we report how many particles are required for 90% of the runs to reach an accuracy close to a target (DS with 1000 particles), the corresponding median execution time, and below and in gray the range between 10% and 90% execution time quantiles over 1000 runs. ✓ 1 indicates that the algorithm is able to compute the exact solution. ✗ indicates a timeout.

6.3 Results

Figure 8 presents the results of the evaluation for the *Tree*, *Wheels*, and *Delayed GPS* benchmarks. Figures for the other benchmarks are in the extended paper's appendix [Atkinson et al. 2022b].

Following Baudart et al. [2020], to summarize the results in Table 1, we evaluate how many particles are required for 90% of the 1000 runs to reach a target accuracy – in this case, the median loss of DS with 1000 particles: $\log(P_{90\%}(loss)) - \log(loss_{target}) < 0.5$.

RQ1. Accuracy. Figure 8a reports the median accuracy and the 90% and 10% quantiles over 1000 executions with different random seeds for the *Tree*, *Wheels*, and *Delayed GPS* benchmarks.

Overall, we observe in Table 1 that both DS and SSI outperform PF. For the three models where DS is exact (*Beta-Bernoulli*, *Kalman-1D*, and *Robot*), SSI is also able to compute the exact solution. Moreover, SSI is exact for the three more challenging models where DS requires multiple particles: *Tree*, *Wheels*, and *Delayed GPS*. For the four remaining models (*Gaussian-Gaussian*, *Outlier*, *SLAM*, and *MTT*), SSI behaves similarly to DS. More generally, SSI always outperforms DS in accuracy, i.e., SSI requires the same number of particles or less than DS to reach the same accuracy.

RQ2. Speed. Figure 8b reports the execution time in milliseconds for the three challenging benchmarks and Table 1 reports the median execution time to reach the target accuracy.

The results show that SSI is on average 10.5× faster than PF, but there is a noticeable overhead to run SSI compared to DS, as SSI is on average 1.6× slower than DS. However, SSI has a significant speedup for models where DS fails to compute the exact solution. For the more challenging models, compared to DS, SSI is 22× faster on the *Tree* model, 86.9× faster on the *Wheels* model, and 2.8× faster on the *Delayed GPS* model.

```
let proba model (u, acc, gps) = x where
  rec x = sample (mv_gaussian (last mu, noise))
  and init mu = x0
  and mu = (a *@ x) +@ (b *@ u)
  and () = observe (gaussian (project_2 *@ x, acc_noise), acc)
  and buff_x = buffer(max_delay, x)
  and present gps(pos_delay, pos) -> do () =
    observe (gaussian (project_0 *@ get(buff_x, pos_delay), gps_noise), pos) done
```

Fig. 9. The Delayed GPS example that exercises the single m-path constraint.

6.4 Discussion: Comparison to Delayed Sampling

In this section we discuss in more detail the reasons why semi-symbolic inference outperforms delayed sampling on the more challenging benchmarks. In particular, we identify two classes of probabilistic programs on which delayed sampling cannot support exact inference on, but that semi-symbolic inference can: programs with *multiple parents* and programs with *multiple paths*.

6.4.1 Multiple Parents. First, we compare delayed sampling and semi-symbolic inference on an example exercising the *multiple parents* case. This means that this example has a random variable that depends on more than one other random variable in the program (i.e. in the symbolic state, the variable will have more than one parent). This example is the *Wheels* benchmark, presented in full in Section 2. The observed random variables on Lines 5 and 6 of Figure 1 depend on both of the sampled random variables on Lines 3 and 4, and thus each has two parents.

Delayed Sampling. Delayed sampling does not support random variables with multiple parents [Murray et al. 2018]. Thus, to execute on this example, delayed sampling must adapt the example to not have multiple parents. The implementation in ProbZelus performs this adaptation by falling back on approximate sampling. When ProbZelus executes one of the `observe` statements and detects that the random variable depends on both of the random variables pointed to by `vel` and `omega`, it samples either `vel` or `omega`. Once this variable has been sampled, the remaining random variables all have at most a single parent, and delayed sampling execution can proceed normally. However, the sampling step introduces approximation error.

Semi-Symbolic Inference. This model satisfies the conditions of Theorem 5.4 under the linear-Gaussian closed family. Thus, because the program contains no calls to `value`, all symbolic states produced by the algorithm are in the Gaussian closed family and no sampling occurs.

6.4.2 Multiple Paths. One way, originally proposed in Murray et al. [2018], to circumvent the multiple-parent restriction of delayed sampling is to collapse random variables into *supernodes*. For example, for the program in Figure 1, the developer could rewrite the example to use multivariate Gaussians and rewrite the arithmetic operators to use matrix multiplication.

However, there exist programs using supernodes that delayed sampling cannot execute purely symbolically. In this subsection we consider the *Delayed GPS* benchmark that exercises a constraint on delayed sampling called the *single m-path constraint*. Due to this constraint, delayed sampling must also fall back on a sampling-based approximation for this program.

Example. We consider the example of a robot trying to infer its position using sensors. The robot's sensors consist of an accelerometer and a GPS receiver. At each time step, the accelerometer produces a noisy observation of the robot's acceleration. Intermittently, the robot also receives

noisy observations of its position from the GPS receiver. However, due to delays in the internal processing inherent to any GPS receiver [Solomon et al. 2012], the GPS signal may be delayed, in that it provides a noisy observation of the position at a previous time step. The delay can vary, but we assume that it is known as part of the GPS signal.

Figure 9 presents how such an example can be encoded in ProbZelus. Line 1 specifies that the `model` stream function takes in three parameters: `u`, a stream of *commands* the robot issues to adjust its position; `acc`, a stream of accelerometer inputs; and `gps`, a stream of GPS inputs. It further specifies that the `model` stream function returns the hidden state `x`, which is defined by the subsequent set of mutually recursive equations.

Line 2 specifies that the hidden state `x` is sampled from a multivariate Gaussian distribution with mean `last mu`, the previous value of `mu`, and constant covariance matrix `noise`. The hidden state is a vector of length 3 containing numbers that represent the position, velocity, and acceleration of the robot. Line 3 specifies the initial value of `mu` – the mean of the hidden state – to be the constant value `x0`. Line 4 defines `mu` at subsequent time steps, which is given by a sum of two components. The first component uses the operator `*@` – specifying matrix multiplication – to multiply the value of the hidden state `x` with the fixed constant matrix `a`. The second component multiplies the the input command `u` with the fixed constant matrix `b`, and the two components are added together with the vector addition operator `+@`.

Line 5 describes the accelerometer process. It specifies a noisy observation of the component of `x` at index 2 – i.e., the robot's acceleration – which is extracted by multiplying `x` with the projection matrix given by the constant `project_2`. The noisy observation consists of a random variable drawn from a Gaussian distribution centered around the projected-out acceleration. The `observe` operation conditions the model on this random variable being equal to the input `acc`.

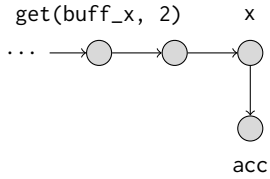
Line 6 defines `buff_x`, a sliding window keeping the `max_delay` previous values of `x`, and Line 7 describes the GPS observations. The `gps` input stream is a ProbZelus construct called a *signal* that may be present or not at each time step. The syntax `present gps(pos_delay, pos) ->` means that whenever the `gps` has the value – which is destructed to the pair of `pos_delay` and `pos` – the program will execute whatever follows. What follows in this case is the observation of the GPS signal.

The observation of the GPS signal makes use of `get(buff_x, pos_delay)`, which accesses the value of `x` delayed by `pos_delay` time steps in the buffered stream `buff_x`. The program then projects out the position by multiplying with the constant matrix `project_0`. The observation specifies that the input `pos` is sampled from a Gaussian distribution whose mean is the delayed position and variance is `gps_noise`.

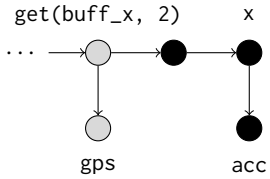
Delayed Sampling. This example exercises a constraint of delayed sampling called the *single m-path constraint*. In delayed sampling, random variables are represented using one of three types of nodes in a graph: initialized, marginalized, or realized. Over the course of a delayed sampling execution, nodes change state from initialized to marginalized and from marginalized to realized. Notably, realizing a variable samples from a distribution and thus loses accuracy.

In delayed sampling, the symbolic state may only contain a single *m*-path: a path from the root containing all marginalized nodes.[1] Furthermore, any variable that is being `observe`d and all of its ancestors must be marginalized. Thus, in the example in Figure 9, when the delayed GPS signal is observed after the acceleration, the inference system moves the *m*-path from the acceleration variable to the delayed GPS variable, and all marginalized hidden states in between the current and delayed time steps are realized to preserve the single *m*-path constraint. Figure 10 depicts this case.

[1] In general, delayed sampling can have multiple *m*-paths, but may only have one *m*-path per tree in the graph. Our example has only one tree, so may have only one *m*-path.



(a) The delayed sampling symbolic state after the inference system observes the acceleration acc all nodes must be marginalized at this point.



(b) The delayed sampling state after the inference system observes the delayed GPS signal gps. Two of the hidden states must become realized in order to satisfy the single m-path constraint.

Fig. 10. A depiction of how delayed sampling executes on the program in Figure 9. We depict marginalized nodes in gray and realized nodes in black. Arrows mean that a given random variable depends on another random variable (i.e., is sampled or observed from other variable). We have also labeled the random variables pointed to by the current hidden state x, the 2-step delayed hidden state `get(buff_x, 2)`, the observed acceleration acc, and the observed GPS signal gps.

Semi-Symbolic Inference. In our implementation of semi-symbolic inference in ProbZelus, we have implemented rules for multivariate Gaussians similar to the rules for the univariate Gaussians in Algorithm 1. Multivariate Gaussians also form a closed family, and this model always produces symbolic states in that closed family. Thus, the model will execute fully symbolically and the runtime inference system will not draw any samples. The underlying reason is that semi-symbolic inference's swaps are fully reversible transformations. By contrast, when delayed sampling changes a node's state, it cannot change it back, and sets the execution on an irreversible path towards sampling-based approximation.

7 RELATED WORK

In this section, we compare semi-symbolic inference to various related approaches.

7.1 Exact Inference Systems

Some probabilistic programming systems are designed specifically for exact inference. Examples include the PSI Solver [Gehr et al. 2016], Dice [Holtzen et al. 2020], SPPL [Saad et al. 2021], and Autoconj [Hoffman et al. 2018]. Some of these languages also take advantage of closed families. For example, Dice focuses on exact inference with problems that only have finite discrete random variables. Exact inference is always possible on these models, and Dice thus focuses on improving the computational efficiency of exact inference. Other systems, such as PSI, use a complex solver to potentially solve a much larger class of programs. The Autoconj system analyzes probabilistic programs using a similar symbolic representation to that presented in Section 4. In general, these systems do not support the symbolic interface we discuss in Section 3.2, inhibiting our ability to use them to build delayed sampling runtime inference systems.

7.2 Delayed Sampling

We compared this work against ProbZelus's delayed sampling system because prior work on ProbZelus [Baudart et al. 2020] established that delayed sampling is an effective technique for implementing RBPFs within the inference system of a streaming probabilistic programming language. However, as we discuss in Section 6.4, the limitations of ProbZelus's delayed sampling system means that is not able to provide exact inference in all cases that semi-symbolic inference can.

ProbZelus uses the same delayed sampling system as Birch [Murray et al. 2018] and Anglican [Lundén 2017], and we expect these limitations to apply to these other languages as well. Furthermore, because Birch and Anglican expose the same symbolic interface as ProbZelus, we anticipate semi-symbolic inference could be applied to these languages.

Pyro [Bingham et al. 2019] supports delayed sampling using an alternative symbolic representation called functional tensors or *funsors* [Obermeyer et al. 2019a]. Pyro performs exact inference on funsors using an alternative symbolic interface based on *variable elimination* [Obermeyer et al. 2019b; Zhang and Poole 1994]. Variable elimination works by removing variables from the symbolic state. For batch execution of probabilistic programs, all remaining variables at the end of the execution can be eliminated, but in a streaming context this approach needs to decide when to eliminate variables. This decision needs to balance 1) eliminating old variables to limit the size of the symbolic state, 2) eliminating variables to compute particle weights, and 3) keeping variables available for use at future time steps. Due to these tradeoffs, more work is needed to develop a delayed sampling system for streaming probabilistic programs based on variable elimination.

7.3 Alternatives for Combining Exact and Approximate Inference

Hakaru. Hakaru [Narayanan et al. 2016] is a probabilistic programming system that statically rewrites probabilistic programs into inference procedures. This includes transformations that introduce sampling-based approximations as well as analytically solving some distributions with a solver. One tradeoff between Hakaru and this work is the inherent tradeoff between static and dynamic techniques. Hakaru relies on a sufficiently powerful static analysis to determine if exact inference is possible, whereas semi-symbolic inference will exploit the closed-family guarantees so long as the program satisfies the necessary conditions at runtime.

Stochastic Procedures. Languages such as Venture [Mansinghka et al. 2018] support encapsulating exact inference components inside *stochastic procedure* objects. These stochastic procedures can then be used inside of various approximate inference algorithms. However, these require developers to rewrite their input probabilistic models to use stochastic procedures, which breaks the separation between modeling and inference. By contrast, semi-symbolic inference exists as an alternative inference technique inside the language runtime inference system and only requires developers to add calls to `value` to control where sampling occurs.

Shared Variables. Infer.NET [Minka et al. 2018] provides language support for inference on graphical models, including exact inference with belief propagation [Pearl 1982]. Through its feature of *shared variables*, Infer.NET supports combining belief propagation with the approximate inference algorithms of expectation propagation [Minka 2001], variational message passing [Winn and Bishop 2005], and Gibbs sampling [Geman and Geman 1984]. This does not present the same interface as Definition 3.1. Instead, it provides an alternative way of combining exact and approximate inference that does not result in an RBPF inference algorithm.

8 CONCLUSION

In this paper, we presented semi-symbolic inference, a novel technique for combining exact and approximate probabilistic inference. It enables developers to write models in a high-level streaming probabilistic programming language while the language runtime inference system automatically implements Rao-Blackwellized particle filtering. It presents developers of streaming probabilistic programs with the opportunity to combine the efficiency of exact inference with the generality of sampling-based approximate inference to achieve the performance they expect.

ACKNOWLEDGMENTS

This work was supported in part by the MIT-IBM Watson AI Lab and the Office of Naval Research (ONR N00014-17-1-2699). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Office of Naval Research.

REFERENCES

Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022a. Semi-Symbolic Inference for Efficient Streaming Probabilistic Programming. https://doi.org/10.5281/zenodo.7082520.

Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022b. Semi-Symbolic Inference for Efficient Streaming Probabilistic Programming. https://arxiv.org/abs/2209.07490

Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive probabilistic programming. In *PLDI*. ACM, 898–912.

Keni Bernardin and Rainer Stiefelhagen. 2008. Evaluating Multiple Object Tracking Performance: The CLEAR MOT Metrics. *EURASIP J. Image and Video Processing* 2008 (2008).

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6.

Arnaud Doucet, Nando de Freitas, Kevin P. Murphy, and Stuart J. Russell. 2000. Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks. In *UAI*. Morgan Kaufmann, 176–183.

Daniel Fink. 1997. A Compendium of Conjugate Priors.

Timon Gehr, Sasa Misailovic, and Martin T. Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV (1) (Lecture Notes in Computer Science, Vol. 9779)*. Springer, 62–83.

Stuart Geman and Donald Geman. 1984. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6, 6 (1984).

Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org Accessed April 2020.

N.J. Gordon, D.J. Salmond, and A.F.M. Smith. 1993. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In *IEE Proceedings-F*.

Matthew D Hoffman, Matthew J Johnson, and Dustin Tran. 2018. Autoconj: Recognizing and Exploiting Conjugacy Without a Domain-specific Language. In *NeurIPS*.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. In *OOPSLA*.

T.D. Larsen, K.L. Hansen, N.A. Andersen, and Ole Ravn. 1999. Design of Kalman filters for mobile robots; evaluation of the kinematic and odometric approach. In *Proceedings of the IEEE International Conference on Control Applications*.

Daniel Lundén. 2017. *Delayed sampling in the probabilistic programming language Anglican.* Master's thesis. KTH Royal Institute of Technology.

Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. In *PLDI*.

T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. 2018. Infer.NET 0.3. Microsoft Research Cambridge. http://dotnet.github.io/infer.

Thomas P. Minka. 2001. Expectation Propagation for approximate Bayesian inference. In *UAI*. Morgan Kaufmann, 362–369.

Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *International Conference on Artificial Intelligence and Statistics*.

Lawrence M. Murray and Thomas B. Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018), 29–43.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *FLOPS (Lecture Notes in Computer Science, Vol. 9613)*. Springer, 62–79.

Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Du Phan, and Jonathan Chen. 2019a. Functional Tensors for Probabilistic Programming. In *Program Transformations for ML Workshop at NeurIPS*.

Fritz Obermeyer, Elias Bingham, Martin Jankowiak, Neeraj Pradhan, Justin Chiu, Alexander Rush, and Noah Goodman. 2019b. Tensor Variable Elimination for for Plated Factor Graphs. In *ICML*.

Judea Pearl. 1982. Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach. In *AAAI*.

Feras Saad, Martin Rinard, and Vikash Mansinghka. 2021. SPPL: Probabilistic Programming with Fast Exact Symbolic Inference. In *PLDI*.

P. D. Solomon, Jinling Wang, and Chris Rizos. 2012. Latency Determination and Compensation in Real-Time Gnss/ins Integrated Navigation Systems. *ISPRS* 3822 (2012), 303–307.

Eduardo D Sontag. 2013. *Mathematical control theory: deterministic finite dimensional systems*. Vol. 6. Springer Science & Business Media.

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank D. Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *IFL*. ACM, 6:1–6:12.

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *ICLR (Poster)*. OpenReview.net.

John Winn and Christopher M. Bishop. 2005. Variational Message Passing. *Journal of Machine Learning Research* 6, 4 (2005).

N.L. Zhang and D. Poole. 1994. A Simple Approach to Bayesian Network Computations. In *Canadian Conference on Artificial Intelligence*.