Inference Plans for Hybrid Particle Filtering

ELLIE Y. CHENG, MIT CSAIL, USA
ERIC ATKINSON, Binghamton University, USA
GUILLAUME BAUDART, Université Paris Cité – CNRS – Inria – IRIF, France
LOUIS MANDEL, IBM Research, USA
MICHAEL CARBIN, MIT CSAIL, USA

Advanced probabilistic programming languages (PPLs) using *hybrid particle filtering* combine symbolic exact inference and Monte Carlo methods to improve inference performance. These systems use heuristics to partition random variables within the program into variables that are encoded symbolically and variables that are encoded with sampled values, and the heuristics are not necessarily aligned with the developer's performance evaluation metrics. In this work, we present *inference plans*, a programming interface that enables developers to control the partitioning of random variables during hybrid particle filtering. We further present Siren, a new PPL that enables developers to use annotations to specify inference plans the inference system must implement. To assist developers with statically reasoning about whether an inference plan can be implemented, we present an abstract-interpretation-based static analysis for Siren for determining inference plan *satisfiability*. We prove the analysis is sound with respect to Siren's semantics. Our evaluation applies inference plans to three different hybrid particle filtering algorithms on a suite of benchmarks. It shows that the control provided by inference plans enables speed ups of 1.76x on average and up to 206x to reach a target accuracy, compared to the inference plans implemented by default heuristics; the results also show that inference plans improve accuracy by 1.83x on average and up to 595x with less or equal runtime, compared to the default inference plans. We further show that our static analysis is precise in practice, identifying all satisfiable inference plans in 27 out of the 33 benchmark-algorithm evaluation settings.

CCS Concepts: • **Mathematics of computing → Sequential Monte Carlo methods**; • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: Probabilistic programming languages, static analysis, abstract interpretation

ACM Reference Format:
Ellie Y. Cheng, Eric Atkinson, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2025. Inference Plans for Hybrid Particle Filtering. *Proc. ACM Program. Lang.* 9, POPL, Article 10 (January 2025), 29 pages. https://doi.org/10.1145/3704846

1 Introduction

Probabilistic programming languages (PPLs) support primitives for modeling random variables and performing probabilistic inference [Goodman and Stuhlmüller 2014; Holtzen et al. 2020; Murray and Schön 2018; Narayanan et al. 2016; Tolpin et al. 2016]. They provide high-level abstractions for probabilistic modeling that hide away the complex details of inference algorithms while leveraging common programming language constructs such as functions, loops, and control flow. PPLs serve

Authors' Contact Information: Ellie Y. Cheng, MIT CSAIL, USA, ellieyhc@csail.mit.edu; Eric Atkinson, Binghamton University, USA, eatkinson2@binghamton.edu; Guillaume Baudart, Université Paris Cité – CNRS – Inria – IRIF, France, guillaume.baudart@inria.fr; Louis Mandel, IBM Research, USA, lmandel@us.ibm.com; Michael Carbin, MIT CSAIL, USA, mcarbin@csail.mit.edu.

ACM 2475-1421/2025/1-ART10
https://doi.org/10.1145/3704846

as an expressive tool for solving such problems. Users can focus on modeling the problem, rather than the details of inference techniques.

Hybrid Inference Systems. Hybrid inference systems – such as delayed sampling [Lundén 2017; Murray et al. 2018], semi-symbolic inference [Atkinson et al. 2022], Sequential Monte Carlo with belief propagation [Azizian et al. 2023], and automatically marginalized MCMC [Lai et al. 2023] – automatically incorporate exact inference with Monte Carlo methods to improve performance. They utilize a *symbolic encoding* of random variables to represent some or all parts of the model, enabling symbolic computation that lowers the variance of estimations. Hybrid inference algorithms that apply to particle filters [Gordon et al. 1993] implement an automatic *Rao-Blackwellization* of the particle filter [Doucet et al. 2000]; hybrid inference algorithms that apply to MCMC algorithms automatically implement *collapsed sampling* [Liu 1994]. In this work, we focus on hybrid inference systems that perform symbolic computations dynamically at runtime. These systems are able to take advantage of exact inference opportunities that only become available once the inference system replaces some variables with concrete Monte Carlo samples.

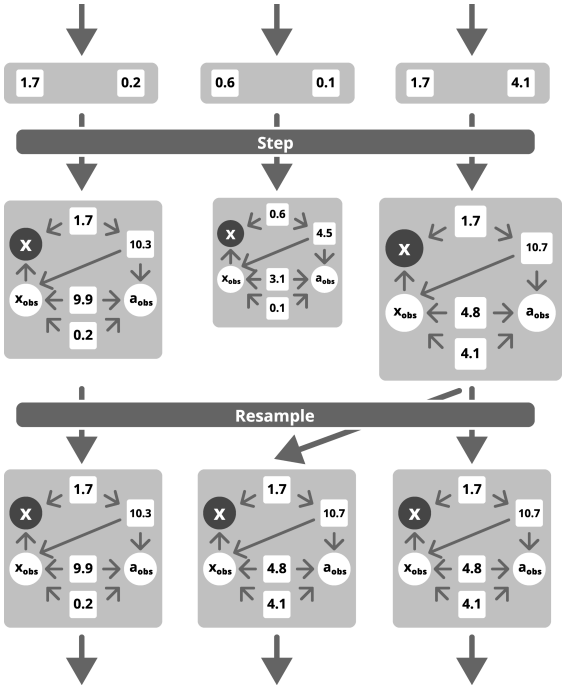


Fig. 1. Hybrid inference with particle filtering.

Figure 1 depicts hybrid inference using particle filtering as the approximate inference method. The algorithm maintains a collection of parallel instances of executions, represented by the big boxes. Each instance contains a symbolic structure that encodes certain random variables symbolically, as shown by the circles in the diagram; other random variables are encoded as constant samples drawn from probability distributions, depicted as squares. The sizes of the boxes correspond to the associated *weight*, which indicates how likely these instances are based on observed data (the white circles). The instances execute in parallel until they hit a checkpoint at which they are *resampled*, meaning the distribution of instances is adjusted based on the weights. The arrows between each box indicate the transition or transformation of a particle from one stage to the next, including when a particle is duplicated during the resampling step.

Objective Oblivious Heuristics. When a hybrid inference system cannot solve the entire program with symbolic computation, it must partition the random variables into variables that it encodes symbolically and variables it encodes with concrete sampled values. Different partitions make different subsets of random variables more accurate. Choosing which partition to use then depends on 1) the probabilistic model and 2) the metrics used by the developer to evaluate the program. Hybrid inference systems automatically choose a partition to use based on the program structure using built-in heuristics. However, they are oblivious to the objectives of the developer and to how the developer measures performance. Their selected partition might not produce good inference performance as a result. In these cases, developers need an interface for applying alternative heuristics that incorporate their evaluation objectives to achieve better performance.

Inference Plans. In this work, we present *inference plans*, a programming interface that gives developers fine-grained control over how random variables are partitioned into sampled and symbolic variables during hybrid inference. An inference plan consists of a sequence of *distribution encoding* annotations – `symbolic` and `sample` – that specify whether the inference runtime should encode each random variable with a symbolic distribution or an approximate Monte Carlo sample. Our evaluation shows that the control provided by inference plans enables speed ups of 1.76x on average and up to 206x to reach a target accuracy, compared to the inference plans implemented by default heuristics; the results also show that inference plans improve accuracy by 1.83x on average and up to 595x with less or equal runtime, compared to the default inference plans.

In applications, such as control systems [Burkhart and Bishop 1996; Huang et al. 2017b; Mehra et al. 1995], robotics [Doucet et al. 2000], signal processing [Duník et al. 2017], and data science [Mansinghka et al. 2018], developers must tradeoff between accuracy and runtime during program development before deploying the program in production. By testing different inference plans during program development, developers can apply heuristics best-suited to their applications and optimize their programs on custom performance metrics.

Satisfiability Analysis. A key challenge in delivering the inference plans interface is that, depending on the program, the inference algorithm may not be able to maintain a random variable symbolically if the corresponding inference problem is too hard for the system to solve exactly. Furthermore, because hybrid inference systems are dynamic and can change their behavior based on both the program inputs and random samples, the failure to maintain symbolic computation may only manifest in some executions. This can lead to unpredictable or random accuracy degradations. We call an inference plan *unsatisfiable* in some execution if in that execution, the algorithm cannot evaluate the program while encoding the `symbolic` annotated variables symbolically. We present in this work an abstract-interpretation-based program analysis that can statically determine whether or not an inference plan is satisfiable in all executions for the given program. Against a suite of 11 benchmarks, using the 3 algorithms, our analysis identifies all satisfiable plans in 27 out of the 33 benchmark-algorithm evaluation settings.

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

- We present Siren, a first-order functional PPL. Siren introduces distribution encoding annotations that programmers can use to assert an overall specification for how variables are represented by the inference algorithm; we term this specification an inference plan. Siren implements several existing hybrid particle filtering systems, unified via the hybrid inference interface, an extension of the symbolic interface from [Atkinson et al. 2022]. We define the syntax and semantics of the language in Section 3.
- We present an *inference plan satisfiability analysis*, which determines statically if the annotated inference plan is satisfiable for all executions of a program. We formalize this analysis via abstract interpretation and present a proof of its soundness in Section 4.
- We implement the hybrid inference interface with semi-symbolic inference, delayed sampling, and Sequential Monte Carlo with belief propagation, and empirically show in Section 5 that inference plans speed up inference by 1.76x on average and up to 206x to reach target accuracy, compared to the default inference plans. We also show that inference plans improve accuracy by 1.83x on average and up to 595x with less or equal runtime, compared to the default plans.
- We empirically evaluate the precision of our analysis in Section 5. Against a suite of 11 benchmarks, using the 3 algorithms, our analysis identifies all satisfiable plans in 27 out of the 33 benchmark-algorithm evaluation settings.

Inference plans enable developers to apply alternative heuristics to hybrid particle filtering algorithms within a probabilistic program. These customizations can improve inference performance, all the while maintaining the separation of probabilistic modeling and inference.

2 Example

To demonstrate how a developer can use inference plans to customize and improve inference performance, we present a simplified example adapted from Bilik and Tabrikian [2010]. Figure 2 shows a cartoon diagram of a radar tracker. The goal of the radar tracker is to track the movement of an aircraft by estimating the position `x` and altitude `alt` of the aircraft over time. The radar tracker can be specified as a probabilistic model. The model captures both the aircraft's *movement noise* and also the radar's *measurement noise*. The movement noise `q` captures the uncertainty in the model's belief of the aircraft movement relative to its previous position due to external factors such as turbulence. The measurement noise captures inaccuracies in the radar measurements. For example, measurements naturally have white noise from radio interference. Additionally, when the aircraft is at lower altitudes, the measurements can be affected by spiking noise induced by electromagnetic waves reflecting off of the aircraft at an angle [Bilik and Tabrikian 2010]. In our example tracker, when the aircraft is at an altitude greater than 5, the measurement noise is modeled by only the white noise `r`. Otherwise, it has additional noise `other`.

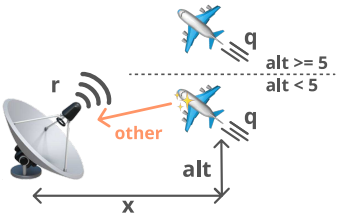


Fig. 2. Diagram of a radar tracker.

2.1 Specification in Siren

Figure 3 presents two versions of the Siren program that implements the radar tracker. The programs share the same probabilistic model, but differ in the annotations used to specify random variable encodings. Each program models the movement noise `q` and white noise `r` defined on Lines 10 and 11 with Inverse-Gamma distributions. The `step` function defined on Lines 1-9 updates the predicted position and altitude and conditions the model on radar measurement data.

On Lines 3 and 4, the `x` positions and `alt` altitudes are modeled as Gaussian random walks, i.e. as Gaussian distributions with a mean equal to the respective previous value. They have variance equal to `q`. Lines 5 and 6 model the measurement noise `v`. If the current estimated altitude `alt` is less than 5, the program models the measurement noise as `r+other`; the variable `other` models the additional spiking noise as a separate, time-varying Inverse-Gamma distribution. Otherwise, the program models the measurement noise as only the white noise `r`. The measured position and altitude are modeled as Gaussian distribution centered around the estimated position `x` and estimated altitude `alt`, respectively, and with the measurement noise as variance.

Lines 7 and 8 condition the model on the measured position being equal to the data value `x_o` and on the measured altitude being equal to `a_o`. The `fold_resample` operation on Line 12 iterates `step` on the measurement data (provided by program inputs stored in variable `data`) with initial position 0, altitude 10, and the noises `q` and `r`. The program returns the final accumulator: the list of estimated positions, estimated altitudes, the movement noise, and the white noise.

2.2 Inference Plans

Each random variable in Figure 3 has an optional *distribution encoding* annotation that specifies how the inference system should encode the variable. The inference system should encode a variable annotated with `symbolic` as a symbolic expression representing the corresponding distribution, and should encode one annotated with `sample` with a concrete sample drawn from the distribution. The two programs in Figure 3 differ in only the annotations on Lines 3 and 11.

```
1  let step = fun ((x_o,a_o),(xs,alts,q,r)) ->
2    let x0,alt0 = hd(xs),hd(alts) in
3    let symbolic x <- gaussian(x0,q) in
4    let sample alt <- gaussian(alt0,q) in
5    let sample other <- invgamma(1.,10.) in
6    let v = if alt < 5 then r+other else r in
7    let () = observe(gaussian(x,v),x_o) in
8    let () = observe(gaussian(alt,v),a_o) in
9    (cons(x,xs),cons(alt,alts),q,r)
10 let sample q <- invgamma(1.,1.) in
11 let sample r <- invgamma(1.,1.) in
12 fold_resample(step,data,([0.],[10.],q,r))
```

(a) Annotated with *Symbolic x Inference Plan.*

```
1  let step = fun ((x_o,a_o),(xs,alts,q,r)) ->
2    let x0,alt0 = hd(xs),hd(alts) in
3    let sample x <- gaussian(x0,q) in
4    let sample alt <- gaussian(alt0,q) in
5    let sample other <- invgamma(1.,10.) in
6    let v = if alt < 5 then r+other else r in
7    let () = observe(gaussian(x,v),x_o) in
8    let () = observe(gaussian(alt,v),a_o) in
9    (cons(x,xs),cons(alt,alts),q,r)
10 let sample q <- invgamma(1.,1.) in
11 let symbolic r <- invgamma(1.,1.) in
12 fold_resample(step,data,([0.],[10.],q,r))
```

(b) Annotated with *Symbolic r Inference Plan.*

Fig. 3. A program written in Siren that implements a radar tracking.

The annotations control the execution of Siren's hybrid particle filtering implementation – a combination of particle filtering with symbolic computation. Each particle contains a symbolic state, which collects the symbolic distributions of symbolically-encoded random variables. The Siren runtime treats variables encoded using samples as constant values during program execution.

The Figure 1 diagram shows an execution of the Figure 3a program. In one particle of the execution, the variable `q` is bound to the constant value 1.7, a random sample drawn from the Inverse-Gamma distribution on Line 10. Then, on the first iteration, the symbolic random variable X_x created on Line 3 has the symbolic distribution $\mathcal{N}(0., 1.7)$. This particle also has `alt` bound to the constant sample 10.3 and `r` to 0.2; the observed variable on Line 7 has the distribution $\mathcal{N}(X_x, 0.2)$. In another particle, `q` is bound to 0.6, `r` to 0.1, `alt` to 4.5, and `other` to 3.1. Then, X_x has the symbolic distribution $\mathcal{N}(0., 0.6)$ and the Line 7 variable has $\mathcal{N}(X_x, 3.2)$, as `r+other` evaluates to 3.2.

The inference algorithm uses symbolic computation to evaluate the symbolic components of the particle. For example, in Figure 3a, the observed variable on Line 7 has a conditional distribution dependent on the symbolic variable X_x. The algorithm symbolically transforms the conditional distribution into a marginal distribution so that the algorithm can condition the model on the input value. During evaluation, the particle accumulates a weight from conditioning on input values. Then, at resampling checkpoints, a new collection of particles is resampled from the existing collection. In Figure 3 the resampling step occurs at the end of each `fold_resample` iteration.

2.2.1 Accuracy and Performance. To deploy the radar tracker, the developer must ensure that it will achieve adequate performance. In this aircraft tracking application, the program must be within the acceptable margins of error and allowable latency or it can lead to catastrophic collisions [Ali et al. 2015]. This is challenging because the accuracy and runtime of hybrid particle filtering depend on both the number of particles and the inference plan used to perform inference.

Particle Count. While the runtime of a hybrid particle filter typically varies proportionally to the particle count, the relationship between accuracy and particle count is difficult to determine. In general, developers need to search for a count that meets their accuracy and runtime constraints.

Inference Plan. A hybrid inference system's accuracy and runtime also depend on the inference plan. This enables developers to use inference plans to control the inference system's performance and improve upon the system's default performance (i.e. its accuracy and runtime under the *default inference plan* automatically selected by the system when no annotations are present in the program). However, as inference plans operate by adjusting the partition between symbolic and sampled

variables, the effect on accuracy and runtime is often hard to predict. Keeping variables symbolic *can* reduce the number of particles the system requires to achieve adequate accuracy, but this behavior is not guaranteed. Additionally, different inference plans can achieve higher accuracy for *different* variables. In an application such as radar tracking where highly accurate results are required within a time constraint, the developer can determine the program's behavior by executing it with different inference plans to build a *performance profile*. The developer can then choose the best inference plan and particle count to use in production.

Performance Profile. In the tracking example, the most important variables are the target's position and altitude, so the developer would like to optimize the program for the highest accuracy for `x` and `alt`. However, the program runtime cannot exceed the acceptable latency, or the estimations will be too out of date. For this example, we assume 3 seconds as the maximum allowable runtime.

Figure 4 presents a performance profile for the programs in Figure 3. These graphs present scatter plots of accuracy and latency over a range of particle counts and for a variety of different inference plans. Each graph uses an accuracy metric measuring the error of the inference algorithm's estimate of either `x` or `alt`. The red squares present the time and accuracy tradeoff for the default inference plan that makes no annotations. An alternative inference plan – called the *Symbolic x Inference Plan* because it annotates `x` with `symbolic` – is the one from Figure 3a, and is shown in green diamonds. An additional alternative plan – called *Symbolic r Inference Plan* because it annotates `r` symbolic – is the one from Figure 3b, and is shown in purple circles.

Inference Plan Comparison. The developer can conclude from the performance profile in Figure 4 that, given a time constraint of 3 seconds, the best plan to optimize for `x` accuracy is the *Symbolic x Plan*. This demonstrates the value of using inference plans to control the inference system's behavior, as the default behavior of the system is oblivious to the developer's objectives and cannot adapt accordingly. Consequently, the default plan is inferior in accuracy.

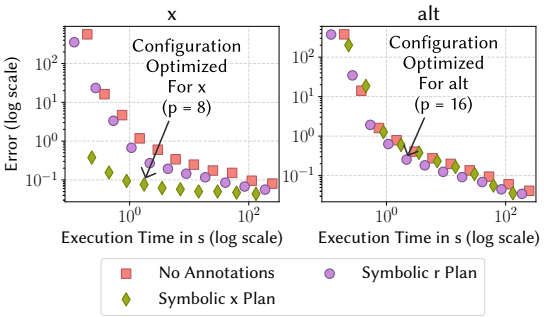


Fig. 4. Accuracy and runtime performance of the Figure 3 programs. Each scatter plot presents the program execution time and accuracy for particle counts p ranging from 1 to 1024 and multiple inference plans. Each data point is an experiment that – across 100 runs – measures the median runtime and the 90th percentile of the Mean Squared Error for the relevant random variable – `x` or `alt`.

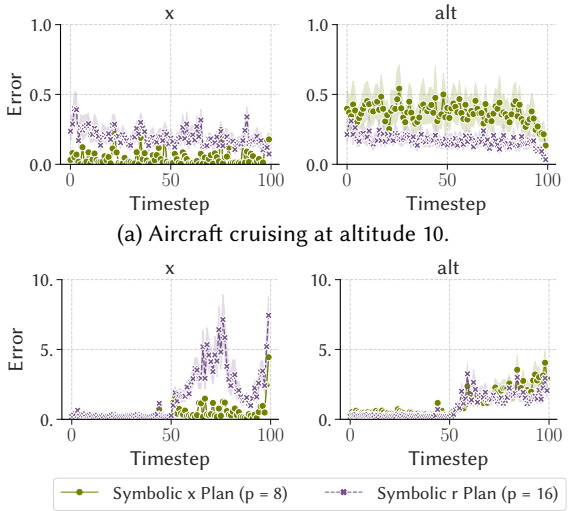
The profile also shows that the *Symbolic r Plan* achieves the best accuracy for `alt` within the 3-second constraint. This plan also achieves better accuracy than the default plan on both `x` and `alt`. None of the plans achieves the best accuracy within 3 seconds for both variables, so the developer has to decide which is more critical to optimize for. The differing performance outcomes further illustrate the importance of inference plans: Developers can apply alternative inference plans to achieve the best performance on the variables they care about the most.

2.2.2 Unsatisfiable Annotations. According to the performance profiles, the developer may select either the *Symbolic x Plan* of Figure 3a with 8 particles or the *Symbolic r Plan* of Figure 3b with 16 particles as the configuration to deploy in production with acceptable accuracy and latency. However, this profile also demonstrates the challenges in drawing conclusions from empirical data: The *Symbolic r Plan* is *unsatisfiable* in general.

To collect the performance profile, we generated the data for the performance profile assuming the aircraft stays at its cruising altitude of 10. If we instead generate data to model a descent where the aircraft eventually descends to an altitude below 5, the input altitude data passed to the variable `a_o` on Line 1 will be different. Then, the performance profile may no longer be valid. Namely, with the original generated cruising data, the estimated altitude was never less than 5. However, with the generated descent data, the probability of the estimated altitude being less than 5 is significantly higher and, as a result, so is the probability of encountering an *unsatisfiable annotation*. The `symbolic` annotation on `r` is unsatisfiable in those executions because the inference runtime cannot evaluate the program while encoding `r` symbolically.

In Figure 3b, if the altitude of the aircraft is at or above 5, then the observed variables on Lines 7 and 8 have symbolic Gaussian distributions with variance X_r. The Inverse-Gamma distribution of X_r is conjugate with the Gaussian distribution, which means that the inference system can find a closed-form solution to the model and can encode `r` symbolically. However, if the altitude of the aircraft drops below 5, the program specifies that the observed variables have symbolic Gaussian distributions with variance equal to the sum of two Inverse-Gamma random variables: `r` and `other`. This sum does not have an Inverse-Gamma distribution and is not conjugate with the Gaussian distributions on Lines 7 and 8. Without a conjugacy relationship to exploit, the inference system cannot solve the model analytically, even though the developer annotated variable `r` as `symbolic`.

Dynamic Encoding Cast. In such a scenario, Siren, like other hybrid inference systems [Atkinson et al. 2022; Azizian et al. 2023; Lundén 2017; Murray et al. 2018], will, conceptually, *dynamically cast* the offending `symbolic` annotation to a `sample` annotation, changing the underlying distribution encoding to a concrete sample. However, Figure 5 illustrates the impact of such a coercion. Figure 5 shows the accuracy of the position and altitude of a simulated aircraft over 100 timesteps using the two programs in the two different flight modes. In Figure 5a, where the aircraft is cruising at altitude 10, the pattern observed in the performance profiles is replicated. The *Symbolic x Plan* consistently achieves better accuracy for the `x` position, and the *Symbolic r Plan* for `alt`. In either case, both plans have errors less than 1 for both variables at all timesteps.



(a) Aircraft cruising at altitude 10.

(b) Aircraft descending from altitude 10 to 0.

Fig. 5. Accuracy of `x` and `alt` over 100 timesteps at altitudes, measured as the Squared Error of the estimated value to the true value at that timestep.

However, when the aircraft is descending, this pattern is broken. In Figure 5b, the aircraft is now operating in a noisier environment. The accuracy and runtime of a probabilistic model inherently depend on the conditioned inputs, so both plans experience an error spike in `alt`. The *Symbolic x Plan* still maintains a relatively low error for `x`, whereas the *Symbolic r Plan* has a significant error spike at around 60 timesteps such that the estimation error of `x` is a magnitude larger than before. The error spike cannot be explained by the noisier environment alone, because the *Symbolic x Plan* does not exhibit an `x` error spike of the same magnitude. The accuracy degradation in `x` by the *Symbolic r Plan* is due to a dynamic encoding cast on an unsatisfiable annotation.

Performance Degradation. A dynamic encoding cast has implications for inference performance, as it means that the runtime cannot implement the annotated inference plan and has to implement a different inference plan to continue execution. While flexible and enables the program execution to continue, it can cause imprecision that is unacceptable in certain applications. In this example, casting the unsatisfiable annotation causes a significant decrease in accuracy in `x`. The significant accuracy degradation negates the advantage the *Symbolic r Plan* has over the *Symbolic x Plan* for `alt`. Given the possible impact on accuracy, the developer should use the *Symbolic x Plan* instead.

2.3 Inference Plan Satisfiability Analysis

To enable developers to ensure their program is free from dynamic encoding casts, Siren performs the *inference plan satisfiability analysis* to determine whether the annotated inference plan is satisfiable during all possible executions of the program using the hybrid inference algorithm.

Abstract Interpretation. The analysis uses an abstract interpretation of the program. It maintains abstract symbolic distributions of random variables and uses abstract expressions to over-approximate program executions. Consider the unsatisfiable plan from Figure 3b. On Line 6, because the analysis does not know the exact value of `alt`, it uses an abstract value to over-approximate the subexpressions in the branches. If `alt` is above 5, the abstract subexpression is $\hat{X}_r$, referring to the abstract random variable from Line 11. If `alt` is below 5, the subexpression is $\hat{X}_r \hat{+} \texttt{UnkC}$. The abstract value `UnkC` represents some unspecified constant. All random samples are constant values, so `other` evaluates to `UnkC`. The analysis over-approximates the subexpressions as a single abstract expression: $\texttt{UnkE}(\{\hat{X}_r\})$ – an unspecified abstract expression that references $\hat{X}_r$.

Unsatisfiable Inference Plan. The analysis also approximates how the system decides when to perform symbolic computations. The Line 7 variable has the abstract distribution $\widehat{\mathcal{N}}(\texttt{UnkC}, \texttt{UnkE}(\{\hat{X}_r\}))$. The variance is the expression computed on Line 6. The inference algorithm cannot perform symbolic computations when the variance is a complex expression. Because `r` (corresponding to $\hat{X}_r$) is annotated with `symbolic`, the variable will be dynamically cast in those instances. The analysis rejects the program, correctly identifying that the inference plan is unsatisfiable in some executions.

Satisfiable Inference Plan. The satisfiability analysis will always reject unsatisfiable inference plans, but it may be overly conservative and erroneously reject satisfiable plans. Nevertheless, the analysis is precise in practice. For example, it correctly determines the program in Figure 3a is satisfiable. At Line 7, the observed variable has the symbolic distribution $\widehat{\mathcal{N}}(\hat{X}_x, \texttt{UnkC})$, where $\hat{X}_x$ refers to the symbolic random variable created on Line 3. The variance is an unknown constant, the mean is a linear expression, and $\hat{X}_x$ also has a Gaussian symbolic distribution. This model only consists of linear-Gaussian distributions – a class of probabilistic models that the inference algorithm can solve entirely symbolically. Thus, `x` will always be encoded symbolically as the annotation requires, so the analysis accepts the inference plan.

2.4 Summary

Using annotations, developers can select alternative inference plans that are better aligned with their objectives. With the satisfiability analysis, the developer can further guarantee the inference plan will always be satisfiable in any execution. If Siren successfully compiles a program, then any variable annotated with `symbolic` will always be represented symbolically in any execution, and any variable annotated with `sample` will always be sampled. This provides developers with the guarantee that their program performance will not be degraded by unsatisfiable annotations.

3 Language Syntax and Semantics

We present the syntax and semantics of the first-order functional PPL, Siren, adapted from Staton [2017]. We have extended the language to support distribution encoding annotations and adapted it to use hybrid inference, and specify its semantics via the *hybrid inference interface*.

3.1 Syntax

Figure 6 presents the syntax of Siren. An expression e is a value v (constant; variable; pair; or the application of an operator, e.g., arithmetic operation, distribution, or list), a function application, a conditional, or a local definition. We add the classic `fold` operator as well. Siren supports probabilistic operators. The expression $\texttt{let}\ x \leftarrow op(v)\ \texttt{in}\ e$ introduces a new local random variable x with distribution $op(v)$ to be used in e. Optionally, a `symbolic` or `sample` annotation adorns a random variable declaration. The $\texttt{observe}(v_1, v_2)$ expression conditions the model on a variable with distribution v_1 having value v_2. The `resample` operator instructs the program to perform resampling for particle filtering. The `fold_resample` operation used in Section 2 is syntactic sugar for applying `resample` at the end of all `fold` iterations. Finally, a program is a sequence of function declarations d with a main expression.

$$
\begin{aligned}
v &::= c \mid x \mid (v, v) \mid op(v) \mid \texttt{nil} \mid \texttt{cons}(v, v) \\
e &::= v \mid f(v) \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e \\
&\quad \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{fold}(f, v, v) \\
&\quad \mid \texttt{let}\ \kappa\ x \leftarrow op(v)\ \texttt{in}\ e \mid \texttt{observe}(v, v) \\
&\quad \mid \texttt{resample} \\
\kappa &::= \varepsilon \mid \texttt{symbolic} \mid \texttt{sample} \\
d &::= \texttt{let}\ f = \texttt{fun}\ x \rightarrow\ e \\
prog &::= d^*\ e
\end{aligned}
$$

Fig. 6. Syntax of the Siren language.

3.2 Operational Semantics

While Siren has an ideal measure-based semantics (see Appendix A), the measure is, in general, intractable. An alternative is to interpret the model as a *weighted sampler* that returns a value and a *score* measuring the likelihood of the result with respect to the model. To approximate the posterior distribution, a weighted sampler launches a set of independent executions of the sampler, the *particles*, and returns a categorical distribution that associates each value with its score. Siren implements a *particle filter* which occasionally resamples the set of particles according to their score during executions. Following [Lundén et al. 2021], we add an explicit `resample` operator to the language to enable programs to explicitly trigger resampling.[1] We present the operational semantics of Siren which is a big-step semantics extended with checkpoints for resampling.

3.2.1 Hybrid Inference Interface. Hybrid particle filtering algorithms reduce variance in particle filters by computing closed-form distributions where possible and only drawing random samples if symbolic computation fails. We first present definitions for the hybrid inference interface, an extension of the symbolic interface [Atkinson et al. 2022], that underpins our operational semantics.

Symbolic Expressions. Figure 7 presents the grammar of symbolic expressions used by algorithms implementing the hybrid inference interface. It specifies a grammar of distributions D that includes, but is not limited to, Gaussian, Bernoulli, Inverse-Gamma, and Dirac Delta distributions. D can also be a sampled-Delta distribution (denoted as δ_s), which is a Dirac Delta distribution that is used only to represent a sample drawn from a probability distribution. Figure 7 further specifies a grammar of expressions E that uses operators to combine constant values c and random variables X. The operators include standard arithmetic and comparison operators and a conditional operator `ite`.

[1] Automatic selection of resampling locations for optimal performance is an open problem [Lundén et al. 2021].

$$
\begin{aligned}
D &::= \mathcal{N}(E,E) \mid \mathrm{B}(E) \mid \Gamma^{-1}(E,E) \mid \delta(E) \mid \delta_s(E) \mid \ldots \\
E &::= c \mid X \mid E + E \mid E - E \mid E * E \\
&\quad \mid E \;/\; E \mid \texttt{ite}(E,E,E) \mid E\ Cmp\ E \\
Cmp &::= = \mid \texttt{!=} \mid < \mid \texttt{<=} \qquad c \in \mathbb{V}, X \in \mathcal{RV}
\end{aligned}
$$

Fig. 7. Grammar of symbolic expressions.

$$
\begin{aligned}
\textsc{Assume} &: A \times D \times G \to \mathcal{RV} \times G \\
\textsc{Observe} &: \mathcal{RV} \times \mathbb{V} \times G \to G \times \mathbb{R} \\
\textsc{Value} &: \mathcal{RV} \times G \to \mathbb{V} \times G
\end{aligned}
$$

Fig. 8. Hybrid Inference Interface.

Symbolic State. A symbolic state $g \in G = \mathcal{RV} \to A \times D \times N$ is a finite mapping whose domain is the set of random variable names. It maps each random variable to an entry $g(X)$ consisting of an optional annotation ($A = \{\texttt{sample}, \texttt{symbolic}, \varepsilon\}$ where ε represents no annotation) denoted $g(X)_{\mathrm{a}}$, a symbolic representation of a distribution $g(X)_{\mathrm{d}}$, and an implementation-specific data field $g(X)_{\mathrm{n}}$.

Interface. The hybrid inference interface uses three operations to manipulate the symbolic state, shown in Figure 8. The Assume operation takes an annotation, a distribution, and a symbolic state, and returns a new random variable with the updated symbolic state. The Observe operation conditions the symbolic state on the input variable having the given value and returns the updated state and a score for the particle filter to use as the weight. The Value operation replaces the input variable with a sample from its distribution, turning it into a sampled-Delta distribution. These operations decide whether the runtime samples a random variable or encodes a random variable symbolically in the symbolic state. By doing so, they determine the default inference plan in the absence of annotations. We will discuss different implementations of the interface in Section 3.3.

3.2.2 Unsatisfiable Annotation. When a distribution encoding annotation is unsatisfiable, the Siren runtime performs a dynamic encoding cast, enabling the execution to continue. In particular, the Value operation executes as if the annotation of the input random variable is ε even if the annotation of the input random variable is `symbolic`.

3.2.3 Big-Step Semantics with Checkpoints. Next, we present the semantics of Siren. Figures 9 to 11 show a fragment, and the full semantics is in Appendix B. A particle is represented by a pair (expression, symbolic state). A Siren program is described by three types of rules:

- *Particle Evaluation.* The evaluation relation $e, g \downarrow^r e', g', w$ evaluates a particle (e, g) and returns an updated particle (e', g'), the associated score w, and a resample flag r indicating if the evaluation was interrupted.
- *Particle Set Evaluation.* The evaluation relation $\{e_i, g_i\}_{1 \le i \le N} \Downarrow\!\Downarrow D$ gathers the results of a set of particles into a distribution D.
- *Model Evaluation.* The evaluation relation $e \Downarrow_N D$ evaluates a program expression e into a distribution D using N particles.

Particle Evaluation. Figure 9 shows a fragment of the particle evaluation rules. A constant v is already fully reduced so the resample flag r is set to *false*. Since it is a deterministic value, the associated score is 1. The `resample` operator interrupts reductions by setting the resample flag r to *true*; it reduces to the unit value. The semantics of `if` v `then` e_1 `else` e_2 consists of two cases. If e_1 and e_2 are pure (i.e. they do not perform any `observe` or `resample`) and the condition v is not a constant (i.e. it contains some symbolic random variables) then the rule reduces to an `ite` symbolic expression used for symbolic computation. Since there is no `observe`, the score is 1. Otherwise, the rule evaluates the condition to a constant value and uses it to decide which branch to execute. This rule is elided in Figure 9, but can be found in Appendix B. The Siren semantics uses the Value* helper operation to evaluate an expression to a constant by calling Value on all random variables in the expression before evaluation. The semantics of a local declaration `let` x = e_1 `in` e_2 depends

$$\frac{}{v, g \downarrow^{false} v, g, 1} \qquad \frac{}{\texttt{resample}, g \downarrow^{true} \texttt{unit}, g, 1} \qquad \frac{\textsc{pure}(e_1, e_2) \quad \neg\textsc{const}(v) \quad e_1, g \downarrow^{false} v_1, g_1, 1 \quad e_2, g_1 \downarrow^{false} v_2, g', 1}{\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2, g \downarrow^{false} \texttt{ite}(v, v_1, v_2), g', 1}$$

$$\frac{e_1, g \downarrow^{true} e_1', g', w}{\texttt{let } x = e_1 \texttt{ in } e_2, g \downarrow^{true} \texttt{let } x = e_1' \texttt{ in } e_2, g', w} \qquad \frac{e_1, g \downarrow^{false} v_1, g_1, w_1 \quad e_2[x \leftarrow v_1], g_1 \downarrow^{r} e_2', g_2, w_2}{\texttt{let } x = e_1 \texttt{ in } e_2, g \downarrow^{r} e_2', g_2, w_1 * w_2}$$

$$\frac{\texttt{let } x = f((l_{hd}, v)) \texttt{ in } \texttt{fold}(f, l_{tl}, x), g \downarrow^{r} e, g', w}{\texttt{fold}(f, \texttt{cons}(l_{hd}, l_{tl}), v), g \downarrow^{r} e, g', w}$$

$$\frac{\kappa \in \{\varepsilon, \texttt{symbolic}\} \quad \textsc{Assume}(\kappa, dist(v), g) = X, g_X \quad e[x \leftarrow X], g_X \downarrow^{r} e', g', w}{\texttt{let } \kappa\ x \leftarrow dist(v) \texttt{ in } e, g \downarrow^{r} e', g', w}$$

$$\frac{\textsc{Assume}(\texttt{sample}, dist(v), g) = X, g_X \quad \textsc{Value}(X, g_X) = v_x, g_X' \quad e[x \leftarrow v_x], g_X' \downarrow^{r} e', g', w}{\texttt{let sample } x \leftarrow dist(v) \texttt{ in } e, g \downarrow^{r} e', g', w}$$

$$\frac{\textsc{Assume}(\varepsilon, dist(v_1), g) = X, g_X \quad \textsc{Value}^*(v_2, g_X) = v, g_v \quad \textsc{Observe}(X, v, g_v) = g', w}{\texttt{observe}(dist(v_1), v_2), g \downarrow^{false} \texttt{unit}, g', w}$$

Fig. 9. Fragment of the particle evaluation rules. The full semantics is in Appendix B.

$$\frac{\left\{e_i, g_i \downarrow^{false} v_i, g_i', w_i\right\}_{1 \le i \le N} \quad \left\{\textsc{Distribution}(v_i, g_i') = D_i\right\}_{1 \le i \le N} \quad W = \sum_{1 \le i \le N} w_i}{\{e_i, g_i\}_{1 \le i \le N} \Downarrow \sum_{1 \le i \le N} \frac{w_i}{W} \times D_i}$$

$$\frac{\left\{e_i, g_i \downarrow^{r_i} e_i', g_i', w_i\right\}_{1 \le i \le N} \quad \bigvee_{1 \le i \le N} r_i \quad \mu = \textsc{Cat}\left(\left\{w_i, (e_i', g_i')\right\}_{1 \le i \le N}\right) \quad \{\textsc{Draw}(\mu)\}_{1 \le i \le N} \Downarrow D}{\{e_i, g_i\}_{1 \le i \le N} \Downarrow D}$$

Fig. 10. Particle set evaluation rules.

on `resample` operators. If there is a `resample` in e_1 the first rule reduces e_1 up to the first `resample` and stops the evaluation (i.e. the resample flag is set to *true*). Otherwise, e_1 fully reduces without interruption, and the total score is the product of the score of e_1 and e_2. The expression $\texttt{fold}(f, l, v)$ is standard; it evaluates to the accumulator v if the list is empty. Otherwise, the semantics evaluates the function call of f on the first element of l and the accumulator v and recurses on the rest of the list. We include the rules of function calls, impure `if`-expressions, and applying `fold` to an empty list in Appendix B.

A random variable $\texttt{let } \kappa\ x \leftarrow dist(v) \texttt{ in } e$ uses Assume to create a new random variable in the symbolic state. The annotation `sample` denotes that the variable must be encoded using samples, `symbolic` denotes it must be encoded symbolically, and ε denotes that the runtime should decide. If a random variable is annotated `sample`, the operation Value draws a random sample from the variable's distribution and updates the symbolic state. The `observe` operator uses Assume to create a new random variable without annotations in the symbolic state and uses Observe to condition the random variable and compute the score of the particle. The value to condition on must be a constant value, so the rule uses Value* to turn v_2 into a constant value.

Particle Set Evaluation. Figure 10 shows the particle set evaluation rules. The rules handle resuming execution from checkpoints on a set of N particles. If all the particles finish execution (i.e.

the resample flag is *false*), the rule gather the computed distributions into a mixture distribution where $\textsc{Distribution}(v, g)$ returns the distribution of v with respect to the symbolic state g and w_i/W are the normalized scores. Otherwise, the rule builds a categorical distribution μ of particles using the weights and resamples a fresh set of particles $\{\textsc{Draw}(\mu)\}$ before resuming the execution.

Model Evaluation. To evaluate the expression e with a set of N particles, the model evaluation rule launches the particle set evaluation with N independent particles $(e, \emptyset)$ where each particle starts with an initially empty symbolic state. The program evaluates to a distribution D. Figure 11 shows the rule.

$$\frac{\{e, \emptyset\}_{1 \le i \le N} \Downarrow D}{e \Downarrow_N D}$$

Fig. 11. Model evaluation.

3.3 Implementing the Hybrid Inference Interface

The Siren semantics enables inference plans to be used with different hybrid particle filtering algorithms through the hybrid inference interface. The hybrid inference interface serves as a barrier between two halves of hybrid particle filtering algorithms. Above the interface is the implementation of the programming language and the particle filtering component of the algorithm (Figures 9 to 11). Below the interface, i.e. the implementation of the interface operations from Figure 8, is the specification of how to apply symbolic computation. Developers can extend the Siren runtime with a new hybrid particle filtering algorithm by implementing only the interface operations, without needing to re-implement the programming language or the particle filter.

To illustrate how the interface can be implemented, we present the implementation for two algorithms – semi-symbolic inference [Atkinson et al. 2022] and delayed sampling [Lundén 2017; Murray et al. 2018] – to illustrate 1) how to detect opportunities for symbolic computation during inference and 2) how to extend the symbolic state with additional runtime information. We note that the interface is more general; for evaluation in Section 5, we implement a third inference algorithm – Sequential Monte Carlo with belief propagation [Azizian et al. 2023] – that exhibits both features. The full implementations of these algorithms are quite complex, so we defer the full details to the respective works. In this section, we focus on presenting only details that pertain to either 1) how an inference algorithm's internal control flow relates to its inference plans, and 2) providing a semantic foundation for the analysis and its soundness (in particular, for Section 4.3).

3.3.1 Semi-symbolic Inference. Semi-symbolic inference (SSI) implements the hybrid inference interface using a series of helper functions. For example, SSI defines the Value operation – which replaces a random variable with a sample from its distribution – using the hoist and intervene helper functions. The hoist function manipulates the given random variable to have no parent variables and intervene updates the variable with the sample. The Value operation is defined as:

$$\textsc{Value}(X, g) = \ \mathit{let}\ g' = \textsc{hoist}(X, g)\ \mathit{in}\ \mathit{let}\ v = \textsc{Draw}(g'(X)_d)\,\mathit{in}\ (v, \textsc{intervene}(X, \delta_s(v), g'))$$

We defer a full discussion of the implementation of Value, hoist, and other helper functions to [Atkinson et al. 2022]. However, the SSI implementation of the hybrid inference interface depends on a key core operation called swap. A partial definition of swap is as follows:

$$\begin{aligned}
\textsc{swap}(X_1, X_2, g) = \ & \mathit{match}\ g(X_1)_\mathrm{d}, g(X_2)_\mathrm{d}\ \mathit{with} \\
& \mid \mathcal{N}(\mu_0, var_0), \mathcal{N}(\mu, var)\ \mathit{if}\ (\mu = a * X_1 + b) \wedge \textsc{const}(var_0, var) : \\
& \quad \mathit{let}\ (\mu_0', var_0') = ((a * \mu_0) + b, (a * a) * var_0)\ \mathit{in} \\
& \quad \mathit{let}\ (var_0'', \mu_0'') = (1\ /\ (1\ /\ var_0 + 1\ /\ var), (\mu_0'\ /\ var_0' + X_2\ /\ var) * var_0'')\ \mathit{in} \\
& \quad (g[X_1 \mapsto \mathcal{N}((\mu_0'' - b)\ /\ a, var_0''\ /\ (a * a))][X_2 \mapsto \mathcal{N}(\mu_0', var_0' + var)], \mathit{true}) \\
& \ldots \\
& \mid _ : \ (g, \mathit{false})
\end{aligned}$$

The SWAP operation enables the SSI runtime to symbolically transform different conjugate distributions in different cases; here we show the case for linear-Gaussians. When 1) both X_1 and X_2 are Gaussian-distributed, 2) the variance of each distribution is constant (i.e., does not depend on any random variables), and 3) the mean of X_2 is expressible as an affine function of X_1, the SWAP operation performs linear-Gaussian swapping. Note that all operations inside the SWAP construct symbolic expressions and perform no actual computation (e.g. $a * a$ construct a symbolic expression representing a^2). The SWAP operation computes the new parameters of the swapped distributions according to the standard rules for conjugate priors [Fink 1997] and updates the new distributions in the symbolic state. It returns the updated state which represents the same distribution but where X_2 no longer depends on X_1 and X_1 now depends on X_2. It also returns a *true* flag indicating a swap occurred. If no conjugate distributions are available, the SWAP operation returns a *false* flag, indicating that exact inference is not possible and the algorithm must use approximate sampling.

The success and failure of the SWAP transformation determine whether the SSI runtime encodes a random variable symbolically or samples it, influencing the inference plan it implements. For example, in Figure 3a, the `x` variable in Line 3 and the observed Gaussian in Line 7 are linear-Gaussians. The SWAP function will apply the linear-Gaussian swapping, maintaining `x` symbolically. Whereas, if `v` is non-constant, the linear-Gaussian case will not apply. If no other conjugate prior case applies, the SSI runtime will be forced to sample `x`.

3.3.2 Delayed Sampling. Delayed sampling (DS) is a hybrid inference algorithm that also exploits conjugacy relationships [Lundén 2017; Murray et al. 2018]. It is an alternative implementation of the interface that represents the symbolic state using a forest of disjoint trees, where each node in each tree is a random variable.

While we defer the full discussion of DS to prior work, we note here that compared to SSI, DS specifies additional information about each random variable, as each node is one of 3 types – Initialized, Marginalized, or Realized – and the inference plan satisfiability analysis needs to incorporate this additional information. In particular, Initialized nodes represent random variables that have a conditional distribution dependent on their parent; Marginalized nodes represent variables that have marginal distributions, and may need to track an optional prior distribution (and a reference to its original parent); and Realized nodes represent variables that have been replaced by a constant value through sampling or observing. The DS symbolic state uses the data field $g(X)_{\mathrm{n}}$ to track the node type for each random variable, where $S_{rv} \subseteq \mathcal{RV}$ are the children of the node:

$$N ::= \texttt{marginalized}(S_{rv}) \mid \texttt{marginalized}(X, D, S_{rv}) \mid \texttt{initialized}(X, S_{rv}) \mid \texttt{realized}$$

DS maintains invariants about the symbolic state, including that each tree contains at most one path of Marginalized nodes. These invariants further influence the inference plans implemented by the DS runtime and require DS to implement the symbolic interface using a series of unique helper functions. For example, DS implements the VALUE operation using the helpers GRAFT and REALIZE:

$$\textsc{Value}(X, g) = \ \textit{let}\ g' = \textsc{graft}(X, g)\ \textit{in let}\ v = \textsc{Draw}(g'(X)_d)\ \textit{in}\ (v, \textsc{realize}(X, \delta_s(v), g'))$$

While we defer the full details to prior work [Lundén 2017; Murray et al. 2018], we note that GRAFT and REALIZE utilize the node types to manipulate the symbolic state. These operations determine whether the DS runtime samples random variables as well as the default inference plan when no annotations are provided.

4 Inference Plan Satisfiability Analysis

Using the `symbolic` and `sample` distribution encoding annotations, developers can express an inference plan specifying their requirements for how each random variable is encoded. However, the hybrid inference runtime performs a dynamic encoding cast on unsatisfiable annotations,

$$\hat{D} ::= \widehat{N}(\hat{E},\hat{E}) \mid \widehat{\mathrm{B}}(\hat{E}) \mid \widehat{\Gamma^{-1}}(\hat{E},\hat{E}) \mid \hat{\delta}(\hat{E}) \mid \hat{\delta}_s(\hat{E}) \mid \ldots \mid \texttt{TopD} \mid \texttt{UnkD}(\hat{S}_{rv})$$
$$\hat{E} ::= \hat{c} \mid \hat{X} \mid \hat{E} \mathbin{\hat{+}} \hat{E} \mid \ldots \mid \texttt{UnkC} \mid \texttt{TopE} \mid \texttt{UnkE}(\hat{S}_{rv})$$
$$\hat{Cmp} ::= \hat{=} \mid \hat{!=} \mid \hat{<} \mid \hat{<=} \qquad \hat{c} \in \hat{\mathbb{V}}, \hat{X} \in \widehat{\mathcal{RV}}, \hat{S}_{rv} \subseteq \widehat{\mathcal{RV}}$$

Fig. 12. Grammar of abstract symbolic expressions. Grayed-out expressions are identical to those in Figure 7.

enabling the program execution to continue at the risk of potential accuracy degradation. In this section, we present the *inference plan satisfiability analysis*, which statically identifies unsatisfiable annotations to assist developers reason about which inference plans to use. If the analysis passes, the inference is guaranteed to encode all `sample` variables with samples and all `symbolic` variables symbolically. We next formalize the analysis as an abstract interpretation and prove its soundness.

4.1 Abstract Hybrid Inference

Our analysis performs an abstract interpretation of the program by relying on an analogous version the hybrid inference interface that operates over the abstract domain. We refer to this version of the interface as the *abstract hybrid inference interface*. We construct abstract symbolic expressions and abstract symbolic states that the abstract interface operates over and manipulates. The abstract interface operations mirror the concrete operations, except that $\widehat{\textsc{Observe}}$ and $\widehat{\textsc{Value}}$ do not perform scoring or sampling.

Abstract Symbolic Expressions. Figure 12 shows the grammar of abstract expressions. For every symbolic expression, there is a corresponding abstract symbolic expression. Abstract expressions can also be `UnkC`, representing all constants, or `TopE`, representing all possible expressions. Additionally, they can also be the $\texttt{UnkE}(\hat{S}_{rv})$ expression, where $\hat{S}_{rv}$ is a set of abstract random variables; the $\texttt{UnkE}(\hat{S}_{rv})$ expression represents expressions that reference any number of the random variables in $\hat{S}_{rv}$. Likewise, abstract distributions also can be `TopD` or $\texttt{UnkD}(\hat{S}_{rv})$.

Abstract symbolic expressions are equipped with a partial order, which we summarize as follows:

$$\begin{array}{rcll}
\hat{c} & \leq & \texttt{UnkC} & \qquad\qquad \texttt{UnkC} \leq \hat{X} \\
\hat{X} & \leq & \texttt{UnkE}(\{\hat{X}\}) & \\
\texttt{UnkE}(\hat{S}_{rv}) & \leq & \texttt{TopE} & \qquad\qquad \texttt{UnkD}(\hat{S}_{rv}) \leq \texttt{TopD} \\
\texttt{UnkE}(\hat{S}_{rv}) & \leq & \texttt{UnkE}(\hat{S}'_{rv}) & \Leftarrow \quad \hat{S}_{rv} \subseteq \hat{S}'_{rv} \\
\texttt{UnkE}(\hat{S}_{rv,1}) \mathbin{\hat{+}} \texttt{UnkE}(\hat{S}_{rv,2}) & \leq & \texttt{UnkE}(\hat{S}'_{rv}) & \Leftarrow \quad \hat{S}'_{rv} = \hat{S}_{rv,1} \cup \hat{S}_{rv,2} \\
\hat{E}_1 \mathbin{\hat{+}} \hat{E}_2 & \leq & \hat{E}'_1 \mathbin{\hat{+}} \hat{E}'_2 & \Leftarrow \quad \hat{E}_1 \leq \hat{E}'_1,\ \hat{E}_2 \leq \hat{E}'_2 \\
 & & \ldots &
\end{array}$$

The abstract expression `UnkC` subsumes all constants and is itself considered a constant. Abstract random variables subsume constants. The $\texttt{UnkE}(\{\hat{X}\})$ expression subsumes the variable $\hat{X}$. The $\texttt{UnkE}(\hat{S}_{rv})$ expression is a refinement of the top expression `TopE` and the relative ordering between $\texttt{UnkE}(\hat{S}_{rv})$ expressions is defined by their variable sets. Likewise, $\texttt{UnkD}(\hat{S}_{rv})$ is a refinement of `TopD`. A plus expression subsumes another plus expression if the subexpressions also subsumes the subexpressions of the other expression. Other complex expressions and distributions are analogous.

Multiple abstract expressions can be over-approximated as one expression via a *joining* expression, defined as taking the least upper bound according to the partial ordering. The partial ordering is designed to maintain more precise abstract expressions and distributions by recursing on subexpressions if the top-level expression type matches. For example, the join of $\hat{X}_1 \mathbin{\hat{+}} \hat{1}$ and $\hat{X}_2 \mathbin{\hat{+}} \hat{2}$ produces $\texttt{UnkE}(\{\hat{X}_1, \hat{X}_2\}) \mathbin{\hat{+}} \texttt{UnkC}$. Random variables are not equivalent to each other, so their join can only be the $\texttt{UnkE}(\{\hat{X}_1, \hat{X}_2\})$ expression.

Retaining a precise representation of expressions is essential to the precision of the analysis because the hybrid inference algorithms depend on identifying expressions of certain classes (e.g. linear-Gaussians) to perform symbolic computation. However, symbolic computations can also cause the expression size to grow exponentially and infinitely, which is computationally expensive. To make the abstract domain finite, while trading off between precision and runtime, the analysis widens abstract expressions that are over the expression tree depth threshold T to $\texttt{UnkE}(\hat{S}_{rv})$. We use $T = 5$ in our implementation.

Abstract Symbolic State. An abstract symbolic state $\hat{g}$ is a finite mapping of abstract random variables (which reside in their own namespace) to tuples of annotations, abstract distributions, and the implementation-specific abstract data field: $\hat{g} \in \hat{G} = \widehat{\mathcal{RV}} \to \hat{A} \times \hat{D} \times \hat{N}$. Each entry is a constraint on entries in the concrete state. For example, $\hat{g}(\hat{X})_{\text{d}} = \hat{\mathcal{N}}(\texttt{UnkC}, \texttt{UnkC})$ requires the concrete state to map the corresponding variable to Gaussian distributions with constant parameters.

$\hat{A} = \left\{\widehat{\texttt{sample}}, \widehat{\texttt{symbolic}}, \varepsilon\right\}$ is the abstraction of annotations that is equipped with a partial ordering that defines its join operation: $\varepsilon \leq \widehat{\texttt{sample}} \leq \widehat{\texttt{symbolic}}$. The partial ordering is designed such that only a single abstract annotation needs to be tracked for each abstract random variable. An abstract random variable represents one or more concrete random variables, and so its abstract annotation must represent one or more concrete annotations. A `sample` annotation is always satisfiable because the Siren semantics collapses the random variable to a sampled value upon instantiation. This means the analysis does not need to remember $\widehat{\texttt{sample}}$ annotations. However, a `symbolic` annotation could be an unsatisfiable annotation if the Siren runtime needs to sample it to continue execution. Then, the analysis must identify if any concrete random variable annotated with `symbolic` will be sampled in the program, so $\widehat{\texttt{symbolic}}$ is the greatest in the partial ordering, to encompass the presence of one or more `symbolic` annotations.

Abstract states are equipped with a partial order and a join operation:

$$\hat{g}_1 \leq \hat{g}_2 \iff \forall_{\hat{X} \in \text{dom}(\hat{g}_1)}\ \hat{g}_1(\hat{X}) \leq \hat{g}_2(\hat{X})$$

During the analysis, the join operation may introduce random variables that are unreachable from the abstract expression accompanying the abstract state. However, additional unreachable random variables in a symbolic state do not affect executions, so abstract symbolic states are equivalent for a given expression if all reachable variables from the expression have the same entries. This notion is useful when comparing two abstract symbolic states during the analysis. We define a weak equivalence between two abstract symbolic states that only compares random variables that are reachable from the given expression to capture this notion.

$$\hat{g}_1 \cong_e \hat{g}_2 \iff \forall_{\hat{X} \in \text{REACHABLE}(e,\, \hat{g}_1,\, \hat{g}_2)}\ \hat{g}_1(\hat{X}) = \hat{g}_2(\hat{X})$$

Precision of Joining Expressions. When a program contains data-dependent or stochastic control flow, the static analysis does not know which branch would be evaluated. In such cases, the analysis must over-approximate the true states of the program by joining the abstract expressions and symbolic states from the branches. It then loses critical information about the structures of the subexpressions. Hybrid inference relies on matching symbolic expressions to detect exact inference opportunities, so the over-approximation can significantly impact the precision of the analysis.

```
let symbolic x1 <- gaussian(1.,1.) in
let symbolic x2 <- gaussian(0.,1.) in
let x = if cond then x1+1. else x2+2. in
observe(gaussian(x,5.), obs)
```

Fig. 13. Example program.

For example, consider the program in Figure 13, where `cond` and `obs` are constant values. The variables `x1` and `x2` refer to the abstract random variables $\hat{X}_1$ and $\hat{X}_2$. No matter which branch the runtime executes, the observed Gaussian is a linear-Gaussian. In SSI, the parent variable in both

$$\frac{\widehat{\textsc{Value}}^{*}(\hat{v},\hat{g}) = \mathsf{UnkC}, \hat{g}_{\hat{v}} \qquad e_1, \hat{g}_v \mathbin{\hat{\downarrow}} \hat{v}_1, \hat{g}_1' \qquad e_2, \hat{g}_v \mathbin{\hat{\downarrow}} \hat{v}_2, \hat{g}_2' \qquad \textsc{rename\_join}(\hat{v}_1, \hat{v}_2, \hat{g}_1', \hat{g}_2') = \hat{v}'', \hat{g}'}{\texttt{if}\ \hat{v}\ \texttt{then}\ e_1\ \texttt{else}\ e_2, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}'', \hat{g}'}$$

$$\frac{f((\hat{l},\ \hat{v})), \hat{g} \mathbin{\hat{\downarrow}} \hat{v}_f, \hat{g}_f \qquad \textsc{rename\_join}(\hat{v}, \hat{v}_f, \hat{g}, \hat{g}_f) = \hat{v}_j, \hat{g}_j \qquad \hat{v} = \hat{v}_j \qquad \hat{g} \cong_{(\hat{l},\hat{v})} \hat{g}_j}{\texttt{fold}(f,\ \hat{l},\ \hat{v}), \hat{g} \mathbin{\hat{\downarrow}} \hat{v}, \hat{g}}$$

$$\frac{f((\hat{l},\ \hat{v})), \hat{g} \mathbin{\hat{\downarrow}} \hat{v}_f, \hat{g}_f \qquad \textsc{rename\_join}(\hat{v}, \hat{v}_f, \hat{g}, \hat{g}_f) = \hat{v}_j, \hat{g}_j \qquad \texttt{fold}(f,\ \hat{l},\ \hat{v}_j), \hat{g}_j \mathbin{\hat{\downarrow}} \hat{v}', \hat{g}'}{\texttt{fold}(f,\ \hat{l},\ \hat{v}), \hat{g} \mathbin{\hat{\downarrow}} \hat{v}', \hat{g}'}$$

$$\frac{\hat{S}_D = \left\{\widehat{\textsc{Distribution}}(\hat{v}, \hat{g}') \mid (e, \hat{g}) \in \hat{S}_p, (e, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}, \hat{g}')\right\}}{\hat{S}_p \mathbin{\hat{\Downarrow}} \bigsqcup_{\hat{D}_i \in \hat{S}_D} \hat{D}_i} \qquad \frac{\{e, \emptyset\} \mathbin{\hat{\Downarrow}} \widehat{\textbf{fail}}}{e \mathbin{\hat{\Downarrow}} \widehat{\textbf{fail}}} \qquad \frac{\{e, \emptyset\} \mathbin{\hat{\Downarrow}} \hat{D}}{e \mathbin{\hat{\Downarrow}} \textit{satisfiable}}$$

Fig. 14. Fragment of the abstract interpretation rules. The full set of rules is in Appendix C.

cases would remain as symbolic expressions, so the inference plan is satisfiable for all possible executions. However, the analysis does not know the value of `cond`, so it must over-approximate the program state by joining $\hat{X}_1 \mathbin{\hat{+}} \hat{1}$ and $\hat{X}_2 \mathbin{\hat{+}} \hat{2}$ into $\mathsf{UnkE}(\{\hat{X}_1, \hat{X}_2\}) \mathbin{\hat{+}} \mathsf{UnkC}$. It concludes the resulting abstract observed Gaussian is not necessarily linear-Gaussian, as the $\mathsf{UnkE}(\{\hat{X}_1, \hat{X}_2\})$ expression also represents expressions that are non-linear to $\hat{X}_1$ and $\hat{X}_2$. Consequently, it cannot be sure $\hat{X}_1$ and $\hat{X}_2$ will not be sampled and cannot determine the inference plan is satisfiable even though it is.

We define a special operation for joining expressions that takes advantage of the fact that abstract random variables can be renamed without compromising soundness, which we prove in Appendix D. The key idea is that abstract random variables and concrete random variables exist in different namespaces. A single abstract variable can represent more than one concrete variable if its abstract symbolic state entry over-approximates the entries of those concrete variables. By renaming two otherwise disparate variables to the same name, their entries are forced to be joined into one. We define the special join of two expressions $\hat{E}_1$ and $\hat{E}_2$ under symbolic states $\hat{g}_1$ and $\hat{g}_2$ as:

$$\textsc{rename\_join}(\hat{E}_1, \hat{E}_2, \hat{g}_1, \hat{g}_2) = \textit{let}\ (\hat{E}_3, \hat{g}_3) = \textsc{rename}(\hat{E}_1, \hat{E}_2, \hat{g}_2)\ \textit{in}\ (\hat{E}_1 \sqcup \hat{E}_3, \hat{g}_1 \sqcup \hat{g}_3)$$

where $\sqcup$ refers to the basic join operation implied by the partial orders for abstract expressions and abstract symbolic states. The $\textsc{rename}(\hat{E}_1, \hat{E}_2, \hat{g}_2)$ function returns renamed versions of $\hat{E}_2$ and $\hat{g}_2$ that maximize the similarities between $\hat{E}_1$ and $\hat{E}_2$ with capture-avoiding substitution.

For the program in Figure 13, the analysis renames the variable $\hat{X}_2$ to $\hat{X}_1$. The joined expression is the more structurally precise expression $\hat{X}_1 \mathbin{\hat{+}} \mathsf{UnkC}$, and the joined state assigns $\hat{X}_1$ to $\widehat{\mathcal{N}}(\mathsf{UnkC}, \hat{1})$. Then, the observed variable has the distribution $\widehat{\mathcal{N}}(\hat{X}_1 \mathbin{\hat{+}} \mathsf{UnkC}, 5)$. RENAME_JOIN retains the shared structure in the expressions, and the analysis recognizes the observed variable as a linear-Gaussian.

Fail. A program execution may encounter an unsatisfiable annotation that is dynamically cast to be satisfiable. When the analysis detects the possibility of this event (i.e. when a `symbolic` abstract random variable could be sampled), it returns $\widehat{\textbf{fail}}$, the top of all abstract values: Any value joined with $\widehat{\textbf{fail}}$ results in $\widehat{\textbf{fail}}$. Any operation that receives $\widehat{\textbf{fail}}$ as input also returns $\widehat{\textbf{fail}}$ as the output.

4.2 Abstract Interpretation Rules

Figure 14 presents a fragment of the interpretation rules of a SIREN program using the abstract hybrid inference operations. We present here only the rules that differ from the concrete semantics and include the full abstract semantics in Appendix C.

Conditionals. When $\widehat{\textsc{Value}}^*$ returns UnkC, the analysis cannot determine which branch of a conditional is taken, so it interprets both branches and joins the resulting abstract expressions with RENAME_JOIN to approximate the program execution.

Fold. If the `fold` operation receives a list argument $\hat{l}$ that is not a constant list, such as $\mathsf{UnkE}(\emptyset)$, the analysis over-approximates the operation by computing the abstract fixpoint of the function f. The analysis first interprets f on $(\hat{l}, \hat{v})$. Since $\hat{l}$ is not a constant list, it is either $\mathsf{UnkE}(\hat{S}_{rv})$ or TopE, each of which also over-approximates any particular item in the list, respectively. The analysis computes $(\hat{v}_j, \hat{g}_j)$ using RENAME_JOIN, which are the over-approximations of the current inputs and the inputs of the next iteration for f. If they are weakly equal to the current inputs, no further application of f could be different; the fixpoint computation stops when $\hat{v}$ and $\hat{v}_j$ are equal and $\hat{g}$ and $\hat{g}_j$ are weakly equal with respect to $(\hat{l}, \hat{v})$.

During fixpoint computations, the analysis could be joining $\mathsf{UnkE}(\hat{S}_{rv})$ expressions. However, $\hat{S}_{rv}$ can grow arbitrarily large. To bound the growth, the analysis widens the joined expression by converting $\mathsf{UnkE}(\hat{S}_{rv})$ to TopE if $|\hat{S}_{rv}| \geq N$ for some parameter N. Our implementation uses $N = 4$, but the parameter may be adjusted for greater precision at the cost of more fixpoint iterations.

Particle Set and Model Evaluation. Unlike the concrete semantics, the abstract semantics spawns only a singleton set of particles to evaluate, as there are no weights to consider. All possible particles from a program are accounted for in a single abstract particle evaluation. Thus, there is no abstract equivalent of the resampling step, and `resample` is a no-op. The abstract interpretation of a program is then simply whether the abstract particle evaluation rules encounter failures or not.

4.3 Implementing the Abstract Hybrid Inference Interface

The analysis, analogous to the concrete semantics, relies on the abstract hybrid inference interface. As such, the analysis is also unified across different hybrid particle filtering algorithms. Only the implementation of the interface is required to extend the analysis to a new algorithm. This section presents how the analysis implements an abstract version of the hybrid interface for SSI and DS. While we defer the full details to Appendix C.2, we note the similarity of the abstract operations to the concrete operations from Section 3.3, except that the abstract operations have extensions to handle the analysis's imprecision. We present here the abstract version of SSI's SWAP operation and how the analysis incorporates the additional information in DS's node types.

4.3.1 Semi-Symbolic Inference. Each SSI operation has an abstract version that mirrors the concrete operation and differs only in how it handles abstract values like $\mathsf{UnkD}(\hat{S}_{rv})$. For instance, the $\widehat{\textsc{Value}}$ operation depends on $\widehat{\textsc{hoist}}$ and $\widehat{\textsc{intervene}}$, and it uses UnkC instead of drawing values. However, for $\widehat{\textsc{Value}}$, there is the additional difference that it returns $\widehat{\textbf{fail}}$ if the input variable has the $\widehat{\texttt{symbolic}}$ annotation i.e. the concrete random variables represented by the abstract random variable may have an unsatisfiable annotation.

$$\begin{aligned}\widehat{\textsc{Value}}(\hat{X}, \hat{g}) = \ & \textit{if } \hat{g}(\hat{X})_{\mathrm{a}} = \widehat{\texttt{symbolic}} \textit{ then } \widehat{\textbf{fail}} \\ & \textit{else let } \hat{g}' = \widehat{\textsc{hoist}}(\hat{X}, \hat{g}) \textit{ in } (\mathsf{UnkC}, \widehat{\textsc{intervene}}(\hat{X}, \hat{\delta}_s(\mathsf{UnkC}), \hat{g}'))\end{aligned}$$

To show how implementations handle abstract values, we next describe the abstract SWAP operation used in SSI. The abstract operation simulates the concrete SWAP operation as defined in Section 3.3.1 by detecting conjugacy and performing the equivalent computation where it can:

$$
\begin{aligned}
\widehat{\text{SWAP}}(\hat{X}_1, \hat{X}_2, \hat{g}) = \; & \textit{match } \hat{g}(\hat{X}_1)_{\text{d}}, \hat{g}(\hat{X}_2)_{\text{d}} \textit{ with} \\
& \mid \widehat{\mathcal{N}}(\mu_0, var_0), \widehat{\mathcal{N}}(\mu, var) \textit{ if } (\mu = a \mathbin{\hat{*}} \hat{X}_1 \mathbin{\hat{+}} b) \wedge \widehat{\text{CONST}}(var_0, var) : \\
& \quad \textit{let } (\mu_0', var_0') = ((a \mathbin{\hat{*}} \mu_0) \mathbin{\hat{+}} b, (a \mathbin{\hat{*}} a) \mathbin{\hat{*}} var_0) \textit{ in} \\
& \quad \textit{let } (var_0'', \mu_0'') = (\hat{1} \mathbin{\hat{/}} (\hat{1} \mathbin{\hat{/}} var_0 \mathbin{\hat{+}} \hat{1} \mathbin{\hat{/}} var), (\mu_0' \mathbin{\hat{/}} var_0' \mathbin{\hat{+}} \hat{X}_2 \mathbin{\hat{/}} var) \mathbin{\hat{*}} var_0'') \textit{ in} \\
& \quad (\hat{g}[\hat{X}_1 \mapsto \widehat{\mathcal{N}}((\mu_0'' \mathbin{\hat{-}} b) \mathbin{\hat{/}} a, var_0'' \mathbin{\hat{/}} (a \mathbin{\hat{*}} a))][\hat{X}_2 \mapsto \widehat{\mathcal{N}}(\mu_0', var_0' \mathbin{\hat{+}} var)], \textit{true}) \\
& \ldots \\
& \mid \texttt{UnkD}(\_), \_ : \; (\text{SET\_TOP}(\hat{X}_1, \hat{g}), \widehat{false}) \\
& \mid \_ : \; (\hat{g}, \widehat{false})
\end{aligned}
$$

Because the analysis is performed at compile-time, it can only perform a best-effort detection of conjugates, given that parameters might be represented by the opaque $\texttt{UnkE}(\hat{S}_{rv})$ and $\texttt{UnkD}(\hat{S}_{rv})$ expressions. If the abstract distribution of the parent variable $\hat{X}_1$ is $\texttt{UnkD}(\hat{S}_{rv})$ or TopD, the analysis recursively sets $\hat{X}_1$ and its ancestors to TopD using $\text{SET\_TOP}(\hat{X}_1, \hat{g})$. During this process, if the random variable or any of its ancestors (i.e. the parents of the variable and their parents and so on) are annotated $\widehat{\texttt{symbolic}}$, the analysis cannot be sure if the variable is a conjugate prior nor that it is not a conjugate prior (meaning that it will be sampled), so the analysis conservatively $\widehat{\textbf{fail}}$s.

For example, consider an abstract symbolic state where that the parent variable $\hat{X}_1$ has the abstract distribution $\texttt{UnkD}(\{\hat{X}_3\})$, and variable $\hat{X}_3$ has $\widehat{\mathcal{N}}(1., 1.)$. Also, $\hat{X}_3$ has the $\widehat{\texttt{symbolic}}$ annotation. Because the parent variable $\hat{X}_1$ can be any distribution referencing $\hat{X}_3$, the analysis cannot determine if $\hat{X}_1$ has a conjugate prior distribution to the child variable $\hat{X}_2$. Subsequently, it cannot determine if $\hat{X}_1$ will be sampled or not and will invoke $\text{SET\_TOP}(\hat{X}_1, \hat{g})$, resulting in $\hat{X}_1$ and $\hat{X}_3$ both having the distribution TopD, because it also cannot determine the representations of upstream distributions. $\hat{X}_3$ has the $\widehat{\texttt{symbolic}}$ annotation, so the analysis will return $\widehat{\textbf{fail}}$.

4.3.2 Delayed Sampling. The abstract node types of DS can be Initialized, Marginalized, or Realized. Initialized and Marginalized nodes still track their parent variables, their prior distributions (using abstract expressions), and their children. The fourth abstract node type, TopN, is the top of all node states. It indicates that the analysis does not know the node's type. No prior and no parent to the random variable are tracked for TopN, only its children.

$$
\hat{N} ::= \widehat{\texttt{marginalized}}(\hat{S}_{rv}) \mid \widehat{\texttt{marginalized}}(\hat{X}, \hat{D}, \hat{S}_{rv}) \mid \widehat{\texttt{initialized}}(\hat{X}, \hat{S}_{rv}) \mid \widehat{\texttt{realized}} \mid \texttt{TopN}(\hat{S}_{rv})
$$

Here, $\hat{S}_{rv} \subseteq \widehat{\mathcal{RV}}$ represents the set of possible child random variables. The abstract node states are equipped with a partial ordering:

$$
\begin{array}{rcll}
\widehat{\texttt{realized}} & \leq & \texttt{TopN}(\emptyset) & \\
\widehat{\texttt{marginalized}}(\hat{S}_{rv}) & \leq & \widehat{\texttt{marginalized}}(\hat{S}'_{rv}) & \Leftarrow \; \hat{S}_{rv} \subseteq \hat{S}'_{rv} \\
\widehat{\texttt{marginalized}}(\hat{S}_{rv}) & \leq & \texttt{TopN}(\hat{S}'_{rv}) & \Leftarrow \; \hat{S}_{rv} \subseteq \hat{S}'_{rv} \\
\widehat{\texttt{marginalized}}(\hat{X}, \hat{D}, \hat{S}_{rv}) & \leq & \widehat{\texttt{marginalized}}(\hat{X}', \hat{D}', \hat{S}'_{rv}) & \Leftarrow \; (\hat{X}, \hat{D}) \leq (\hat{X}', \hat{D}'), \hat{S}_{rv} \subseteq \hat{S}'_{rv} \\
\widehat{\texttt{marginalized}}(\hat{X}, \hat{D}, \hat{S}_{rv}) & \leq & \texttt{TopN}(\hat{S}'_{rv}) & \Leftarrow \; \hat{S}_{rv} \subseteq \hat{S}'_{rv} \\
\widehat{\texttt{initialized}}(\hat{X}, \hat{S}_{rv}) & \leq & \widehat{\texttt{initialized}}(\hat{X}', \hat{S}'_{rv}) & \Leftarrow \; \hat{X} \leq \hat{X}', \hat{S}_{rv} \subseteq \hat{S}'_{rv} \\
\widehat{\texttt{initialized}}(\hat{X}, \hat{S}_{rv}) & \leq & \texttt{TopN}(\hat{S}'_{rv}) & \Leftarrow \; \hat{S}_{rv} \subseteq \hat{S}'_{rv} \\
\texttt{TopN}(\hat{S}_{rv}) & \leq & \texttt{TopN}(\hat{S}'_{rv}) & \Leftarrow \; \hat{S}_{rv} \subseteq \hat{S}'_{rv}
\end{array}
$$

The join operation defined by the partial ordering is used when the analysis has to compute the join of two abstract symbolic states. Even though Initialized and Marginalized node states may be tracking parent and prior distributions, TopN does not. This is because as soon as an abstract node type becomes TopN, the analysis recursively over-approximates the parent and Marginal children to

the node by setting them to `TopN` node states as well. During this process, if any of these random variables has a $\widehat{\texttt{symbolic}}$ annotation, the analysis returns $\widehat{\textbf{fail}}$.

For example, consider two abstract symbolic states $\hat{g}_1$ and $\hat{g}_2$. In both states, the variable $\hat{X}_1$ has the node type $\widehat{\texttt{marginalized}}(\{\hat{X}_2\})$. $\hat{X}_2$ has the node type $\widehat{\texttt{initialized}}(\hat{X}_1, \emptyset)$ in $\hat{g}_1$ but $\widehat{\texttt{marginalized}}(\hat{X}_1, \widehat{\mathcal{N}}(\hat{X}_1, 1.), \hat{X}_3)$ in $\hat{g}_2$. The symbolic state $\hat{g}_2$ also has $\hat{X}_3$ having the node type $\widehat{\texttt{marginalized}}(\hat{X}_2, \widehat{\mathcal{N}}(\hat{X}_2, 1.), \emptyset)$ and the $\widehat{\texttt{symbolic}}$ annotation. The join operation between these two states results in $\hat{X}_2$ having `TopN` node type. Because delayed sampling uses the node types to determine whether to sample variables, the analysis would not be able to determine whether its parents and children would be sampled. As such, it would then set $\hat{X}_1$ and $\hat{X}_3$ to both be `TopN` node types as well. The resulting state would have $\hat{X}_1$ having the node type $\texttt{TopN}(\{\hat{X}_2\})$, $\hat{X}_2$ having $\texttt{TopN}(\{\hat{X}_3\})$, and $\hat{X}_3$ having $\texttt{TopN}(\emptyset)$. $\hat{X}_3$ has the $\widehat{\texttt{symbolic}}$ annotation, so the analysis will return $\widehat{\textbf{fail}}$.

4.4 Properties

In this section, we show that the inference plan satisfiability analysis is sound. The approach is mostly standard [Cousot and Cousot 1977, 1992], except for how it handles random variable names and the variable sets in abstract expressions. We will highlight these nonstandard elements throughout the formal development. First, we define the collecting semantics for sets of program states that serves as the basis of our soundness proof. The collecting semantics accumulates from program executions the information relevant to the program properties under study. The abstract states computed by the analysis must over-approximate the collected concrete states to ensure soundness. Next, we define the abstraction and concretization functions that relate abstract values to concrete values. Finally, we present key lemmas and theorems that prove the analysis is sound.

4.4.1 Collecting Semantics. The collecting semantics is a forward collecting semantics [Cousot and Cousot 1992] based on our operational semantics. The program states collected differ between our three types of evaluation rules. Even though the operational semantics uses weight values and performs resampling, the collecting semantics ignores these aspects. The analysis only depends on the possible particles produced during program execution. The resampling step does not introduce any new particles to the execution, only duplicating or removing existing particles. Additionally, weight values do not affect the representation of random variables. As such, weights are not collected and the resampling step in the particle set evaluation rules is a no-op.

$$\frac{S_c = \left\{(e', g', r) \mid (e, g \downarrow^r e', g', w)\right\}}{e, g \,\tilde{\downarrow}\, S_c}$$

Fig. 15. Collecting particle evaluation rule.

Particle Evaluation. The collecting semantics of particle evaluation collects any particle that can result from applying concrete particle evaluation rules to the particle, shown in Figure 15. The rules may produce more than one evaluated particle due to data-dependent or randomized control flow. The collecting semantics returns all such possible evaluated particles with the resample flags, dropping weight values. We call these tuples *configurations*.

When a `symbolic` distribution encoding annotation is unsatisfiable, the Siren runtime performs a dynamic encoding cast by sampling the annotated random variable anyway, enabling the execution to continue. The cast is the event that the inference plan satisfiability analysis must detect. In the collecting semantics, the Value function must return the **fail** value if the annotation of the input random variable is `symbolic`. The **fail** value propagates in the standard way.

$$\begin{aligned}\textsc{Value}(X, g) = \;& \textit{if } g(X)_{\mathrm{a}} = \texttt{symbolic} \textit{ then } \textbf{fail} \\ & \textit{else let } g' = \textsc{hoist}(X, g) \textit{ in let } v = \textsc{Draw}(g'(X)) \textit{in } (v, \textsc{intervene}(X, \delta_s(v), g'))\end{aligned}$$

Particle Set and Model Evaluation. While the bulk of the soundness proof refers to the collecting particle evaluation $\tilde{\downarrow}$, the top-level theorems also refer to collecting analogs of the particle set and model evaluation semantics of Figure 10 and 11. The relation $(S_p \;\tilde{\downdownarrows}\; S_D)$ means that the set of particles S_p evaluates to the set of distributions S_D, and the definition of $\tilde{\downdownarrows}$ refers to the definition of $\tilde{\downarrow}$. Similarly, the relation $(e \;\tilde{\Downarrow}\; S_D)$ means that the program e evaluates to the set of distributions S_D, and the definition $\tilde{\Downarrow}$ depends on $\tilde{\downdownarrows}$. The definitions for both $\tilde{\downdownarrows}$ and $\tilde{\Downarrow}$ are in Appendix D.

4.4.2 Abstraction. The abstraction function α maps sets of concrete values to an abstract value. We define the function first for singleton sets of concrete values. The abstraction of sets of multiple values is then the join of the corresponding abstracted values.

Concrete random variables and abstract random variables have different namespaces. To account for this, the abstraction function assumes the existence of a default mapping $\widehat{RV}_{canon} : \mathcal{RV} \to \widehat{\mathcal{RV}}$ that maps concrete variable names to abstract variable names. The abstractions of both random variables and symbolic states use this to produce the appropriate name in abstract values.

Definition 4.1 (Abstraction Function)**.** We define the abstraction function α as follows. The default mapping function $\widehat{RV}_{canon}$ maps every concrete variable to a unique, canonical abstract variable.

$$
\begin{array}{rcl@{\qquad}rcl}
\alpha(\{c\}) & = & \hat{c} & \alpha(\{X\}) & = & \widehat{RV}_{canon}(X) \\
\alpha(\{E_1 + E_2\}) & = & \alpha(\{E_1\}) \mathbin{\hat{+}} \alpha(\{E_2\}) & \alpha(\{(v_1, v_2)\}) & = & (\alpha(\{v_1\}), \alpha(\{v_2\})) \\
\alpha(\{\texttt{symbolic}\}) & = & \widehat{\texttt{symbolic}} & & & \\
 & & \cdots & & & \\
\alpha(\{g\}) & = & \left\{ \widehat{RV}_{canon}(X) \mapsto \alpha(\{g(X)\}) \;\middle|\; X \in \mathrm{dom}(g) \right\} & & & \\
\alpha(\{\textbf{fail}\}) & = & \widehat{\textbf{fail}} & \alpha(S_v) & = & \bigsqcup_{v \in S_v} \alpha(\{v\})
\end{array}
$$

For example, consider a particle (E, g) with symbolic expression $E = X_1 + 1$ and symbolic state $g = \{X_1 \mapsto (\texttt{symbolic}, \Gamma(1,1), \texttt{realized})\}$. Assuming that $\widehat{RV}_{canon}$ maps X_1 to the abstract variable $\hat{X}_a$, we have that $\alpha(\{E, g\}) = (\hat{X}_a \mathbin{\hat{+}} \hat{1}, \alpha(\{g\}))$ where $\alpha(\{g\}) = \{\hat{X}_a \mapsto (\widehat{\texttt{symbolic}}, \widehat{\Gamma}(\hat{1}, \hat{1}), \widehat{\texttt{realized}})\}$.

4.4.3 Concretization. The concretization function γ plays the opposite role to the abstraction function: it maps every abstract value to a set of concrete values. While we formalize abstraction with a default mapping function, the concretization needs to account for all possible mappings. We first define a version of the concretization function that is parameterized by a surjective function $\widehat{RV} : \mathcal{RV} \to \widehat{\mathcal{RV}}$ that maps concrete random variables to abstract random variables. The function must be surjective since every abstract random variable must have a corresponding concrete random variable. The function does not have to be injective, because an abstract variable can represent multiple concrete variables that share properties in the symbolic state. The concretizations for $\texttt{UnkE}(\hat{S}_{rv})$ and $\texttt{UnkD}(\hat{S}_{rv})$ incorporate the variable set $\hat{S}_{rv}$ by ensuring that the concretization includes only those expressions whose *free variables* are a subset of $\hat{S}_{rv}$. We formalize this using the operation $FV(E, \widehat{RV})$ that returns the set of free variables in E, mapped to abstract names using $\widehat{RV}$. Finally, we define the concretization function by taking the union over all possible name-mapping functions; the set resulting from the concretization function is closed under name re-mappings.

Definition 4.2 (Concretization Function)**.** We define the concretization function γ as follows. First, we define $\underline{\gamma}$ as a function that takes in an abstract state and the name-mapping function $\widehat{RV}$. The function $\widehat{RV}$ is surjective and it maps concrete random variables into abstract random variables.

$$\begin{array}{rcl@{\qquad}rcl}
\gamma(\hat{c}, \widehat{RV}) & = & \{c\} & \gamma(\texttt{UnkC}, \widehat{RV}) & = & \mathbb{V} \\
\gamma(\hat{X}, \widehat{RV}) & = & \{X \mid \widehat{RV}(X) = \hat{X}\} \bigcup \mathbb{V} & \gamma(\texttt{UnkE}(\hat{S}_{rv}), \widehat{RV}) & = & \{E \mid FV(E, \widehat{RV}) \subseteq \hat{S}_{rv}\} \\
\gamma(\texttt{TopE}, \widehat{RV}) & = & E & \gamma(\widehat{\texttt{symbolic}}, \widehat{RV}) & = & \{\texttt{symbolic}, \texttt{sample}, \varepsilon\} \\
\gamma(\hat{E}_1 \mathbin{\hat{+}} \hat{E}_2, \widehat{RV}) & = & \{E_1 + E_2 \mid E_1 \in \gamma(\hat{E}_1, \widehat{RV}), E_2 \in \gamma(\hat{E}_2, \widehat{RV})\} & & & \\
\gamma((\hat{v}_1, \hat{v}_2), \widehat{RV}) & = & \{(v_1, v_2) \mid v_1 \in \gamma(\hat{v}_1, \widehat{RV}), v_2 \in \gamma(\hat{v}_2, \widehat{RV})\} & & & \\
& & \cdots & & &
\end{array}$$

$$\gamma(\hat{g}, \widehat{RV}) = \{g \mid \forall_X \textit{ if } \widehat{RV}(X) \in \mathrm{dom}(\hat{g}) \textit{ then } g(X) \in \hat{g}(\widehat{RV}(X)) \textit{ else } X \notin \mathrm{dom}(g)\}$$

Now, we define γ by taking the union over all possible surjective naming functions:

$$\gamma(\hat{v}) = \left\{ v \;\middle|\; v \in \gamma(\hat{v}, \widehat{RV}), \widehat{RV} \in X \to \hat{X}, \widehat{RV} \textit{ is surjective} \right\}$$

For example, the concrete symbolic state $\{X_1 \mapsto (\texttt{sample}, \delta(1)), X_2 \mapsto (\texttt{sample}, \delta(2))\}$ is included in the concretization $\gamma(\{\hat{X}_a \mapsto (\widehat{\texttt{sample}}, \texttt{UnkD}(\emptyset))\})$. Conversely, the concrete symbolic state $\{X_1 \mapsto (\texttt{sample}, \mathcal{N}(X_2, 1.)), X_2 \mapsto (\texttt{sample}, \delta_s(1))\}$ is not. Additionally, the concretization $\gamma(\hat{D})$ where $\hat{D} = \widehat{\mathcal{N}}(\hat{X}_a, \hat{X}_b \mathbin{\hat{+}} \hat{1}.)$ includes both $\mathcal{N}(X_0, X_1 + 1.)$ and $\mathcal{N}(X_1, X_0 + 1.)$.

4.4.4 Soundness of Analysis. We now present the key ideas and properties necessary to prove the soundness of the analysis and defer the full formalization and proofs to Appendix D. Our treatment of soundness is limited in that we assume the analysis has a sound implementation of the symbolic interface, and show that under this assumption, the overall analysis is sound. We formalize this assumption as follows:

Assumption 4.1 (Abstract Hybrid Inference Interface Soundness). *For every $i \in \{$Assume, Value, Observe$\}$ and input values v_i, we have that $i(v_i) \in \gamma(\hat{i}(\alpha(\{v_i\})))$.*

Because of the join operation on abstract symbolic states, an abstract operation might compute an abstract symbolic state that has variables that are not reachable from the computed expression. The concretization of abstract symbolic states retains those unreachable variables in the concrete symbolic states. Symbolic states with different domains are not strictly equal. However, unreachable variables do not alter the evaluated expression nor the reachable entries in the resulting symbolic state. To account for this property, we define a weak equivalence relation for concrete symbolic states, analogous to the weak equivalence relation for abstract states.

The formalization uses an auxiliary operation $\tilde{\downarrow}^*$ for repeatedly evaluating a particle until it has terminated, which we define precisely in Appendix D. We write configuration sets that have weakly equivalent symbolic states as $S_c \cong S_c' \iff S_c' = \{(e, g', r) \mid (e, g, r) \in S_c, g \cong_e g'\}$. We use an auxiliary operation to drop the resample flag in configurations: $\textsc{forgetr}(S_c) = \{(e, g) \mid (e, g, r) \in S_c\}$.

We first show the analysis is sound when the particle evaluation terminates, and resuming particle evaluation preserves the soundness of the analysis.

Lemma 4.1 (Terminating Particle Evaluation Soundness). *For every particle (e, g), such that $(e, g \mathbin{\tilde{\downarrow}} S_c)$ and $\forall_{(v, g', r) \in S_c}\ (v = \textbf{\textit{fail}}) \vee \neg r$, we have $(e, \alpha(\{g\})) \mathbin{\hat{\downarrow}} \hat{v}', \hat{g}')$ and there exists a configuration set S_c' such that $S_c \cong S_c'$ and $\textsc{forgetr}(S_c') \subseteq \gamma((\hat{v}', \hat{g}'))$.*

Lemma 4.2 (Preservation). *If $(e, g \downarrow^r e', g', w)$, then there exists abstract values $\hat{v}, \hat{v}'$ and abstract symbolic states $\hat{g}, \hat{g}', \hat{g}''$ such that 1) $(e, \alpha(\{g\})) \mathbin{\hat{\downarrow}} \hat{v}, \hat{g}) \iff (e', \alpha(\{g'\}) \mathbin{\hat{\downarrow}} \hat{v}', \hat{g}')$, 2) $\hat{g}' \cong_{\hat{v}'} \hat{g}''$, and 3) $\gamma((\hat{v}', \hat{g}'')) \subseteq \gamma((\hat{v}, \hat{g}))$.*

It follows that the analysis is sound for evaluating any particle until termination.

Lemma 4.3 (Particle Evaluation Soundness). *For every particle (e, g), such that $(e, g \mathbin{\tilde{\downarrow}}^* S_c)$, we have $(e, \alpha(\{g\}) \mathbin{\hat{\downarrow}} \hat{v}, \hat{g})$ and a configuration set S_c' such that $S_c \cong S_c'$ and $\textsc{forgetr}(S_c') \subseteq \gamma((\hat{v}, \hat{g}))$.*

Additionally, every distribution resulting from a particle set evaluation can be traced back to a particle in the particle set and be equivalently derived by evaluating the particle until termination.

Lemma 4.4 (Particle Trace). *If* $(S_p \;\tilde{\Downarrow}\; S_D)$*, we have for all* $D \in S_D$*, there exists* $(e, g) \in S_p$ *such that* $(e, g \;\tilde{\downarrow}^{*}\; S_c)$ *and* $D \in \{\textit{Distribution}(v, g) \mid (v, g, r) \in S_c\}$.

From the particle trace property with the fact the analysis is sound when evaluating a particle until termination, we have that the analysis is sound with respect to evaluating sets of particles. The soundness of the model evaluation follows.

Theorem 4.5 (Particle Set Evaluation Soundness). *For every particle set* S_p*, and distribution set* S_D *such that* $(S_p \;\tilde{\Downarrow}\; S_D)$*, we have that* $\{e, \alpha(\{g\}) \mid (e, g) \in S_p\} \;\hat{\Downarrow}\; \hat{D}$ *and* $S_D \subseteq \gamma(\hat{D})$.

Corollary 4.6 (Model Evaluation Soundness). *If* $e \;\tilde{\Downarrow}\; \{\textbf{\textit{fail}}\}$*, then* $e \;\hat{\Downarrow}\; \widehat{\textbf{\textit{fail}}}$.

Overall, the soundness results show that if the analysis does not produce $\widehat{\textbf{fail}}$, the collecting semantics does not produce **fail** and therefore every execution of the program is satisfiable with respect to the inference plan.

5 Evaluation

In this section, we empirically evaluate the efficacy of Siren on a set of probabilistic programs. We also empirically evaluate how good the inference plan satisfiability analysis is at identifying whether an inference plan is satisfiable. We seek to answer these research questions:

RQ1. Can inference plans improve hybrid particle filtering performance? In other words, does there exist an inference plan that improves program performance compared to the default plan?

RQ2. How precise is the inference plan satisfiability analysis? Section 4.4 proves the analysis is sound, so it will never state an unsatisfiable inference plan is satisfiable. The task remains to empirically determine whether the analysis can detect satisfiable inference plans in practice.

RQ3. How long does the inference plan satisfiability analysis take?

5.1 Benchmarks

We evaluate the performance of different hybrid particle filtering algorithms on a set of benchmark programs. We describe here the 11 benchmarks and identify the variables evaluated for accuracy. The following benchmarks are benchmarks with multiple inference plans from prior work on SSI and DS by Atkinson et al. [2022] and Baudart et al. [2020]: *Outlier*, *Tree*, *SLAM*, and *Wheels*. We describe the programs and the evaluated variables in Appendix E. We also added the following additional benchmarks, each of which cannot be solved purely with exact inference. They demonstrate the advantages of using inference plans for improving performance against the default behavior.

Aircraft is the program presented in Section 2.

Noise is a one-dimensional particle filter with a hidden state modeled by Gaussian distributions (`x`) with variance modeled by an Inverse-Gamma distribution (`q`). Observations are made on Gaussian distributions centered around the previous state with variance also modeled by an inverse-Gamma distribution (`r`). This program is adapted from Duník et al. [2017].

Radar is a radar tracker with glint noise modeled as a random, rarely occurring spike [Wu 1993]. The observation noise is modeled by the sum of two independent Inverse-Gamma distributions (`r` and `other`) if the `env` random variable – modeled by a Bernoulli – indicates a spike. This differs from the *Aircraft* program in that it only models the Gaussian-distributed `x` position, and the Bernoulli random variable determines whether to observe a spike in the measurement noise.

EnvNoise is similar to *Radar*, but the noise variable `other` is modeled by the more flexible Beta distribution [Arazo et al. 2019; Ma and Leijon 2011].

OutlierHeavy models a one-dimensional particle filter where there might be sensor errors producing outlier observations. The hidden state is modeled by Gaussian distributions (`xt`) and the sensor error rate as a Beta prior (`outlier_prob`) to a Bernoulli. The regular observations are made on Gaussian distributions and the outlier observations are modeled by a long-tailed location-scale t distribution as used in Chang [2014]. This program is an extension of the *Outlier* benchmark implemented by Atkinson et al. [2022] and adapted from Minka [2001].

SLDS is a switching linear dynamical system adapted from Obermeyer et al. [2019]. The model switches between two nonlinear Kalman filters that each have unknown measurement noises. The switching label follows Markovian dynamics and is modeled by two Beta priors (`trans_prob0` and `trans_prob1`). The filters use Gaussian distributed hidden states (`x0` and `x1`). The measurement noises are modeled by Inverse-Gamma distributions (`obs_noise0`, `obs_noise1`).

Runner, adapted from Azizian et al. [2023], models the 2-D position (`x`, `y`) and speed (`sx`, `xy`) of a runner based on speedometer readings and the altitude, modeled by Gaussian distributions.

We include the source code and the annotations of each evaluated inference plan for each benchmark in Appendix E.

5.2 Methodology

We implemented Siren in Python. In addition to semi-symbolic inference and delayed sampling, we also implemented a third hybrid inference algorithm: SMC with belief propagation (SMC with BP) [Azizian et al. 2023]. The algorithm swaps parent-child dependencies using conjugate distributions similar to SSI and maintains node types in the data field like DS. The implementation is available at https://github.com/psg-mit/siren/tree/main.

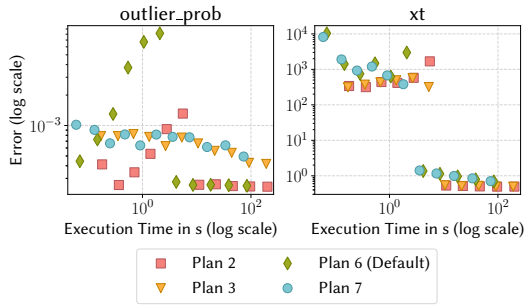
5.2.1 RQ1 Methodology. To determine Siren's performance for *RQ1*, we execute each benchmark 100 times for 100 timesteps with an exponentially increasing particle count from 1 to 1024. We execute each benchmark using all satisfiable inference plans, except for *SLDS* with SSI and DS, where due to the large number of inference plans, we sort the plans by the number of `symbolic` variables in descending order and only compare the first 4 plans (and any plans tied with those) against the plan with all `sample`d variables and the default plan. We set the timeout to 300 seconds. We measure the accuracy by the Mean Squared Error of the posterior expected value of the variable compared to its ground truth value, which is available as all data were generated by sampling from the prior. For each benchmark, we compute the speedup each inference plan achieves compared to the default plan to reach the *target accuracy*, defined as the 90th percentile of error by the default plan using the greatest particle count evaluated that did not timeout. Following Atkinson et al. [2022] and Baudart et al. [2020], reaching target accuracy is defined as $\log(P_{90\%}(loss)) - \log(loss_{target}) < 0.5$. We also compute the summary statistics of the increase in accuracy each plan achieves compared to the default plan with less than or equal runtime. We conduct experiments on a 60-vCPU Intel Xeon Cascade Lake (up to 3.9 GHz) node with 240 GB RAM.

5.2.2 RQ2 Methodology. To determine Siren's analysis precision for *RQ2*, we enumerated all satisfiable and unsatisfiable inference plans, and measured if the inference plan satisfiability analysis correctly determines the satisfiability of each plan.

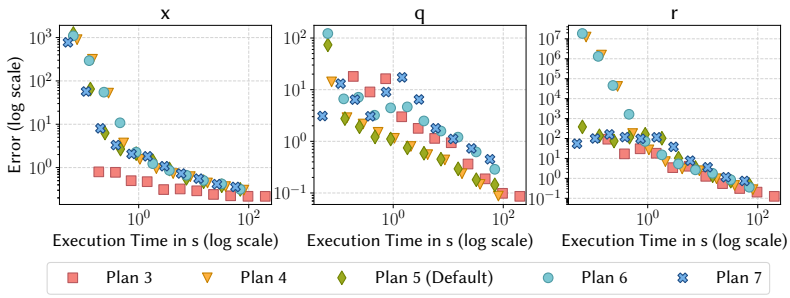
5.2.3 RQ3 Methodology. For each program, algorithm, and inference plan (satisfiable or not), we measured the runtime of the analysis to answer RQ3.

5.3 Results

5.3.1 RQ1 Results. Across all benchmarks, variables, and inference algorithms, using the best inference plans produces an average speedup of 1.76x to reach the same target accuracy compared



(a) *Outlier*.



(b) *Noise*.

Fig. 16. For each particle count, the plots show the median execution time to the 90th percentile of error for each variable using different satisfiable inference plan.

to the default plans, with a maximum speedup of 206x. The best inference plans achieve 1.83x better accuracy with equal or less runtime compared to the default plans, with a maximum increase of 595x. See Appendix F for the breakdowns for each benchmark.

Figure 16 plots the experiment results of *Outlier* and *Noise* using the SSI algorithm. For each particle count, we plot the median runtime to the 90th percentile of error for each evaluated variable. For the plots of the remaining algorithms and benchmarks, see Appendix F.

The *Outlier* results in Figure 16 show that, when the execution time is greater than 10 seconds, the annotated Plan 2 achieves better accuracy for `xt` than the default Plan 6 and would be preferred in a context where the developer cares the most about `xt`. Overall, aggregated across particle counts, the best accuracy achieved by any plan with less or equal runtime to the default plan is 2.75x better than the default for `outlier_prob` and 1.89x for `xt`. Additionally, while the default Plan 6 has the fastest execution time to reach target accuracy for `outlier_prob`, the alternative Plan 7 achieves the fastest execution time to reach target accuracy for `xt` offering a speedup of 1.16x over the default plan. Thus, the best inference plan to use depends on the context in which the system is deployed.

For *Noise*, the best accuracy achieved by any plan with less or equal runtime to the default plan is 2.36x better than the default plan for `x`, 1.33x for `q`, and 2.83x for `r` on average across particle counts. The results in Figure 16 show that the annotated Plan 3 achieves the lowest error for `x`, but the alternative Plan 4 and the default Plan 5 achieve the lowest error for `q`. In terms of efficiency, the default Plan 5 has the fastest execution time to reach target accuracy for `q`. But the alternative Plan 3 has the fastest execution time for `x` and `r`, with a speedup of 19x and 1.33x over the default plan, respectively. Thus, the inference plan that should be used depends on the context – in particular which variable the developer considers most important.

Table 1. Number of satisfiable inference plans identified by the inference plan satisfiability analysis out of the total number of satisfiable inference plans for each benchmark and algorithm.

	Identified Satisfiable Plans / Satisfiable Plans										
Algorithm	*Noise*	*Radar*	*EnvN*	*Out*	*OutH*	*Tree*	*SLDS*	*Runner*	*Wheels*	*SLAM*	*Air*
SSI	5/5	3/3	3/3	4/4	2/2	3/4	28/36	4/4	3/4	3/4	3/3
DS	4/4	2/2	2/2	2/2	2/2	3/3	16/16	1/1	1/3	2/2	2/2
SMC w/ BP	2/2	2/2	2/2	2/2	1/1	3/4	4/4	4/4	3/3	1/1	2/2
Total Plans	8	32	32	8	8	4	128	16	4	4	32

Overall, these results demonstrate that inference performance depends strongly on the inference plan, that the plan that should be used to execute the program is context-dependent, and that annotated inference plans enable substantial speedups and increase in accuracy over the default.

5.3.2 RQ2 Results. To evaluate the analysis precision for *RQ2*, we run the analysis on all 11 benchmarks and 3 inference algorithms, and summarize the results in Table 1. We manually count the total number of satisfiable plans in each case as well as how many of these plans the analysis identifies. The analysis identifies all satisfiable plans in 27 out of the 33 evaluated settings. This shows that the analysis is precise in practice. We describe below the edge cases where the analysis does not identify satisfiable plans.

Aliasing. The loss of precision in *SLDS* executed with SSI is due to *aliasing*. When joining expressions in conditionals and `fold` fixpoint computations, the analysis loses information. This introduces an aliasing problem because inconsistent branch conditions are not detected. This is illustrated in the following program,

```
let symbolic x1 <- gaussian(0.,1.) in
let symbolic var1 <- invgamma(1.,1.) in
observe(gaussian(if cond then x1 else 1., if !cond then var1 else 1.), obs)
```

where `cond` and `obs` are constants. Because `cond` and `!cond` are inconsistent branch conditions and the `else`-branches are both constants, in any execution SSI only needs a conjugacy relation for *either* `x1` or `var1`. Such a single-variable conjugacy relation always exists, so annotating both `x1` and `var1` `symbolic` will not throw an error in any execution. However, the analysis approximates the observed Gaussian distribution as $\widehat{\mathcal{N}}(\hat{X}_{x1}, \hat{X}_{var1})$, meaning that the distribution is potentially depending on *both* `x1` and `var1`, which would require a conjugacy relationship for both variables simultaneously. SSI does not support this, so the analysis concludes that SSI may throw an error when in fact no such error-throwing execution exists. This problem manifests in *SLDS* with SSI but does not affect any other benchmarks.

Widening Expressions. In *SLAM* with SSI, the analysis widens abstract expressions to $\texttt{UnkE}(\hat{S}_{rv})$ when the expression tree depth is over the preset threshold because large symbolic expressions can be computationally expensive. It also widens $\texttt{UnkE}(\hat{S}_{rv})$ to `TopE` when the number of variables in $\hat{S}_{rv}$ is over the preset threshold to hasten the convergence of the fixpoint computation during a `fold` because $\hat{S}_{rv}$ can grow indefinitely large. However, the `TopE` expression is not precise enough for the analysis to detect conjugacies that SSI needs to perform symbolic computation in *SLAM* when all the variables are annotated `symbolic`.

Syntactic Partial Ordering Comparisons. The analysis fails to identify satisfiable plans in *Tree* with SSI and SMC with BP and *Wheels* with SSI and DS because the partial ordering of abstract

Table 2. Time taken by the analysis averaged over all inference plans for each benchmark and algorithm in seconds. The analysis time for *SLAM* with SSI is exceptionally long due to the intractable symbolic computation of full exact inference.

Algorithm	*Noise*	*Radar*	*EnvN*	*Out*	*OutH*	*Tree*	*SLDS*	*Runner*	*Wheels*	*SLAM*	*Air*
SSI	0.40	0.41	0.42	0.41	0.41	0.40	0.50	0.54	0.45	40.93	0.46
DS	0.39	0.40	0.41	0.41	0.40	0.38	0.50	0.54	0.45	0.53	0.46
SMC w/ BP	0.39	0.40	0.42	0.41	0.40	0.38	0.49	0.54	0.45	0.51	0.46

expressions performs comparisons syntactically. For example, $\texttt{UnkC} \mathbin{\hat{*}} \hat{X}$ and $\texttt{UnkC} \mathbin{\hat{*}} \hat{X} \mathbin{\hat{+}} \texttt{UnkC}$ do not share syntactic expression structure, so joining the expressions produces $\texttt{UnkE}(\{\hat{X}\})$. The over-approximated expression $\texttt{UnkE}(\{\hat{X}\})$ is not considered an affine expression with respect to $\hat{X}$, causing the analysis to fail to identify linear-Gaussian conjugacies.

5.3.3 RQ3 Results. We summarize the time taken by the analysis averaged over the all inference plans for each evaluated setting in Table 2. Overall, the analysis takes less than 1 second for all benchmarks, algorithms, and inference plans, except for SLAM with SSI using the default Plan 0. Plan 0 annotates all variables with symbolic, which SSI can implement on SLAM. However, the symbolic computation here results in performing exact inference on a program with only Bernoullis, which is intractable as the conjugacy transformation exponentially increases the expression size. By lowering the widening thresholds, this time could be sped up, at the expense of precision. Nevertheless, the analysis is in general fast to perform.

6 Related Work

Hybrid Inference. Siren supports hybrid inference algorithms based on particle filtering. Other Monte Carlo inference methods can also be combined with exact inference; Hakaru [Narayanan et al. 2016], Autoconj [Hoffman et al. 2018], and automatic marginalization [Lai et al. 2023] perform static transformations to solve the model analytically, and allow the rest to be solved with Monte Carlo methods such as Metropolis-Hastings and HMC. These Monte Carlo methods have the same key feature as particle filtering, which is that there is a subset of random variables that, when reduced to constant values, allows the algorithm to analytically solve the rest of the inference problem. This paradigm naturally leads to the concept of partitioning random variables in a probabilistic model into `sample` and `symbolic` random variables. In systems that perform static transformations, the partitioning is inherently known at compile time. However, the concept of inference plans can still provide an explicit interface for reasoning about these partitions. In the dynamic setting where the partitioning is only entirely determined at run time, a static analysis such as the one in Section 4 is necessary to determine the satisfiability of inference plans.

To the best of our knowledge, no prior works have combined dynamic symbolic computations on Monte Carlo methods other than particle filtering. Developing such an algorithm is out of the scope of this work. Nevertheless, as a proof of concept of how the key ideas presented in this work extend to other Monte Carlo-based methods, we present an alternative semantics for Siren using a basic Metropolis-Hastings implementation combined with symbolic computations in Appendix G. We show that using inference plans can improve performance and that the analysis is still precise.

Programmable Inference. Inference plans is an instance of programmable inference, where the programming system hands over control of the inference procedure to the user. Other works in the programmable inference space [Cusumano-Towner et al. 2019; Lew et al. 2019; Mansinghka et al. 2014, 2018; Tehrani et al. 2020] hand over control to the user at varying stages of inference for

different inference paradigms. Our interface applies specifically to enable users to use alternative heuristics for hybrid inference algorithms.

Probabilistic Program Analyses. Several efforts have been made on using program analyses to detect structure to optimize in probabilistic programs [Cheng et al. 2021; Gorinova et al. 2020; Huang et al. 2017a; Ritchie et al. 2016; Zhou et al. 2020]. Other works also use program analyses to statically infer properties about the outputs or resource usage of probabilistic programs [Atkinson et al. 2021; Cousot and Monerau 2012; Di Pierro and Wiklicky 2000; Gorinova et al. 2021; Lee et al. 2023; Manuel et al. 2020; Monniaux 2000, 2001; Ngo et al. 2018; Smith 2008; Wang and Reps 2024]. Our analysis infers properties about the runtime behavior of the probabilistic inference algorithm.

7 Conclusion

In this work, we present Siren, a new probabilistic programming language for hybrid inference. Siren enables developers to use *inference plans* to control the partitioning of random variables into sampled and symbolic variables. To assist programmers in reasoning about inference plans in hybrid inference systems, Siren employs a static analysis that determines if an inference plan is satisfiable in all possible executions of the program. Our design of the hybrid inference interface enables Siren to work with multiple hybrid inference algorithms, including semi-symbolic inference, delayed sampling, and SMC with belief propagation.

The promise of PPLs is to separate the task of probabilistic modeling from the complex low-level details of building an inference algorithm. However, to achieve good performance in practice, developers often need control over the behavior of the inference system. Siren brings custom hybrid inference to the paradigm of probabilistic programming: developers can adjust the behavior of the inference algorithm to achieve better performance while maintaining the separation of modeling and inference.

Data-Availability Statement

The artifact of this work is available on Zenodo [Cheng et al. 2024].

Acknowledgments

This material is based upon work supported in-part by the National Science Foundation Graduate Research Fellowship under Grant No. 2141064, Sloan Foundation, SRC JUMP 2.0 (CoCoSys), and Amazon MIT Science Hub. We thank Alex Renda, Charles Yuan, Tian Jin, Jesse Michel, and Logan Weber for helpful feedback on this work.

References

Busyairah Syd Ali, Arnab Majumdar, Washington Yotto Ochieng, Wolfgang Schuster, and Thiam Kian Chiew. 2015. A causal factors analysis of aircraft incidents due to radar limitations: The Norway case study. *Journal of air transport management* 44 (2015). https://doi.org/10.1016/j.jairtraman.2015.03.004

Eric Arazo, Diego Ortego, Paul Albert, Noel O'Connor, and Kevin McGuinness. 2019. Unsupervised label noise modeling and loss correction. In *International Conference on Machine learning*.

Eric Atkinson, Guillaume Baudart, Louis Mandel, Charles Yuan, and Michael Carbin. 2021. Statically Bounded-Memory Delayed Sampling for Probabilistic Streams. In *Object-oriented Programming, Systems, Languages, and Applications*. https://doi.org/10.1145/3485492

Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022. Semi-Symbolic Inference for Efficient Streaming Probabilistic Programming. In *Object-oriented Programming, Systems, Languages, and Applications*. https://doi.org/10.1145/3563347

Waïss Azizian, Guillaume Baudart, and Marc Lelarge. 2023. Automatic Rao-Blackwellization for Sequential Monte Carlo with Belief Propagation. *arXiv preprint arXiv:2312.09860* (2023).

Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive Probabilistic Programming. In *Programming Language Design and Implementation*. https://doi.org/10.1145/3385412.3386009

I. Bilik and J. Tabrikian. 2010. Maneuvering Target Tracking in the Presence of Glint using the Nonlinear Gaussian Mixture Kalman Filter. *IEEE Trans. Aerospace Electron. Systems* 46, 1 (2010). https://doi.org/10.1109/taes.2010.5417160

P. Daniel Burkhart and Robert H. Bishop. 1996. Adaptive Orbit Determination for Interplanetary Spacecraft. *Journal of Guidance, Control, and Dynamics* 19, 3 (1996). https://doi.org/10.2514/3.21676

Guobin Chang. 2014. Robust Kalman Filtering Based on Mahalanobis Distance as Outlier Judging Criterion. *Journal of Geodesy* 88 (2014). https://doi.org/10.1007/s00190-013-0690-8

Ellie Y. Cheng, Eric Atkinson, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2024. *Inference Plans for Hybrid Particle Filtering Artifact*. https://doi.org/10.5281/zenodo.13924216

Ellie Y. Cheng, Todd Millstein, Guy Van den Broeck, and Steven Holtzen. 2021. flip-hoisting: Exploiting Repeated Parameters in Discrete Probabilistic Programs. *arXiv preprint arXiv:2110.10284* (2021).

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages*. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation* 2 (1992). https://doi.org/10.1093/logcom/2.4.511

Patrick Cousot and Michael Monerau. 2012. Probabilistic Abstract Interpretation. In *European Symposium on Programming*. https://doi.org/10.1007/978-3-642-28869-2_9

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-Purpose Probabilistic Programming System with Programmable Inference. In *Programming Language Design and Implementation*. https://doi.org/10.1145/3314221.3314642

Alessandra Di Pierro and Herbert Wiklicky. 2000. Concurrent Constraint Programming: Towards Probabilistic Abstract Interpretation. In *Principles and Practice of Declarative Programming*. https://doi.org/10.1145/351268.351284

Arnaud Doucet, Nando de Freitas, Kevin Murphy, and Stuart Russell. 2000. Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks. In *Conference on Uncertainty in Artificial Intelligence*.

Jindřich Duník, Ondřej Straka, Oliver Kost, and Jindřich Havlík. 2017. Noise Covariance Matrices in State-Space Models: A Survey and Comparison Of Estimation Methods–Part I. *International Journal of Adaptive Control and Signal Processing* 31, 11 (2017). https://doi.org/10.1002/acs.2783

Daniel Fink. 1997. A Compendium of Conjugate Priors.

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

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 (Radar and Signal Processing)*, Vol. 140. https://doi.org/10.1049/ip-f-2.1993.0015

Maria Gorinova, Dave Moore, and Matthew Hoffman. 2020. Automatic Reparameterisation of Probabilistic Programs. In *International Conference on Machine Learning*.

Maria I. Gorinova, Andrew D. Gordon, Charles Sutton, and Matthijs Vákár. 2021. Conditional Independence by Typing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 1 (2021). https://doi.org/10.1145/3490421

Matthew D. Hoffman, Matthew J. Johnson, and Dustin Tran. 2018. Autoconj: Recognizing and Exploiting Conjugacy Without a Domain-Specific Language. In *Conference on Neural Information Processing Systems*.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. In *Object-oriented Programming, Systems, Languages, and Applications*. https://doi.org/10.1145/3428208

Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017a. Compiling Markov Chain Monte Carlo Algorithms for Probabilistic Modeling. In *Programming Language Design and Implementation*. https://doi.org/10.1145/3062341.3062375

Yulong Huang, Yonggang Zhang, Bo Xu, Zhemin Wu, and Jonathon A. Chambers. 2017b. A New Adaptive Extended Kalman Filter for Cooperative Localization. *IEEE Trans. Aerospace Electron. Systems* 54 (2017). https://doi.org/10.1109/taes.2017.2756763

Jinlin Lai, Javier Burroni, Hui Guan, and Daniel Sheldon. 2023. Automatically Marginalized MCMC in Probabilistic Programming. In *International Conference on Machine Learning*.

Wonyeol Lee, Xavier Rival, and Hongseok Yang. 2023. Smoothness Analysis for Probabilistic Programs with Application to Optimised Variational Inference. *Principles of Programming Languages*. https://doi.org/10.1145/3571205

Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. 2019. Trace Types and Denotational Semantics for Sound Programmable Inference in Probabilistic Languages. In *Principles of Programming Languages*. https://doi.org/10.1145/3371087

Jun S. Liu. 1994. The Collapsed Gibbs Aampler in Bayesian Computations with Applications to a Gene Regulation Problem. *J. Amer. Statist. Assoc.* 89, 427 (1994). https://doi.org/10.2307/2290921

Daniel Lundén. 2017. Delayed Sampling in the Probabilistic Programming Language Anglican.

Daniel Lundén, Johannes Borgström, and David Broman. 2021. Correctness of Sequential Monte Carlo Inference for Probabilistic Programming Languages. In *European Symposium on Programming*. https://doi.org/10.1007/978-3-030-72019-3_15

Zhanyu Ma and Arne Leijon. 2011. Bayesian estimation of beta mixture models with variational inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 11 (2011). https://doi.org/10.1109/tpami.2011.63

Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: A Higher-Order Probabilistic Programming Platform with Programmable Inference. *arXiv preprint arXiv:1404.0099* (2014).

Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic Programming with Programmable Inference. In *Programming Language Design and Implementation*. https://doi.org/10.1145/3192366.3192409

José Manuel, Calderón Trilla, Michael Hicks, Stephen Magill, Piotr Mardziel, and Ian Sweet. 2020. Probabilistic Abstract Interpretation: Sound Inference and Application to Privacy. *Foundations of Probabilistic Programming* (2020). https://doi.org/10.1017/9781108770750.012

R. Mehra, S. Seereeram, D. Bayard, and F. Hadaegh. 1995. Adaptive Kalman Filtering, Failure Detection and Identification for Spacecraft Attitude Estimation. In *International Conference on Control Applications*. https://doi.org/10.1109/cca.1995.555664

Thomas P. Minka. 2001. Expectation Propagation for Approximate Bayesian Inference. In *Conference on Uncertainty in Artificial Intelligence*.

David Monniaux. 2000. Abstract Interpretation of Probabilistic Semantics. In *International Static Analysis Symposium*. https://doi.org/10.1007/978-3-540-45099-3_17

David Monniaux. 2001. An Abstract Monte-Carlo Method for the Analysis of Probabilistic Programs. In *Principles of Programming Languages*. https://doi.org/10.1145/360204.360211

Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas 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). https://doi.org/10.1016/j.arcontrol.2018.10.013

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *International Symposium on Functional and Logic Programming*. https://doi.org/10.1007/978-3-319-29604-3_5

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Programming Language Design and Implementation*. https://doi.org/10.1145/3192366.3192394

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

Daniel Ritchie, Paul Horsfall, and Noah D. Goodman. 2016. Deep Amortized Inference for Probabilistic Programs. *arXiv preprint arXiv:1610.05735* (2016).

Michael J.A. Smith. 2008. Probabilistic Abstract Interpretation of Imperative Programs Using Truncated Normal Distributions. *Electronic Notes in Theoretical Computer Science* 220, 3 (2008). https://doi.org/10.1016/j.entcs.2008.11.018

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *European Symposium on Programming*. https://doi.org/10.1007/978-3-662-54434-1_32

Nazanin Tehrani, Nimar S. Arora, Yucen Lily Li, Kinjal Divesh Shah, David Noursi, Michael Tingley, Narjes Torabi, Eric Lippert, Erik Meijer, et al. 2020. Bean Machine: A Declarative Probabilistic Programming Language for Efficient Programmable Inference. In *International Conference on Probabilistic Graphical Models*.

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Implementation and Application of Functional Languages*. https://doi.org/10.1145/3064899.3064910

Di Wang and Thomas Reps. 2024. Newtonian Program Analysis of Probabilistic Programs. In *Object-oriented Programming, Systems, Languages, and Applications*. https://doi.org/10.1145/3649822

David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *International Conference on Artificial Intelligence and Statistics*.

Weng-Rong Wu. 1993. Target Racking with Glint Noise. *IEEE Trans. Aerospace Electron. Systems* 29 (1993). https://doi.org/10.1109/7.249123

Yuan Zhou, Hongseok Yang, Yee Whye Teh, and Tom Rainforth. 2020. Divide, Conquer, and Combine: A New Inference Strategy for Probabilistic Programs with Stochastic Support. In *International Conference on Machine Learning*.

Received 2024-07-07; accepted 2024-11-07