Inference Plans for Hybrid Probabilistic Inference

by

Ellie Y. Cheng

B.S., University of California - Los Angeles (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by: Ellie Y. Cheng
Department of Electrical Engineering and Computer Science
May 17, 2024

Certified by: Michael Carbin
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Inference Plans for Hybrid Probabilistic Inference

by

Ellie Y. Cheng

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

Advanced probabilistic programming languages (PPLs) use *hybrid inference* systems to combine symbolic exact inference and Monte Carlo sampling to improve inference performance. These systems use heuristics to partition random variables within the program into variables that are represented symbolically and variables that are represented by sampled values, and in general, they make no guarantee that the partitioning is optimal. In this thesis, I present *inference plans*, a programming interface that enables developers to choose a specific partitioning of random variables during hybrid inference. I further present Siren, a new PPL that enables developers to use annotations to specify inference plans. To assist developers with statically reasoning about whether an inference plan can be implemented, I present an abstract-interpretation-based static analysis for Siren for determining inference plan *satisfiability*, and prove the analysis is sound with respect to Siren's semantics. In my evaluation, the results show that custom inference plans can produce up to 1000x better accuracy compared to the default heuristics. They further show that the static analysis is precise in practice, identifying all satisfiable inference plans in 6 out of 7 benchmarks.

Thesis supervisor: Michael Carbin
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I thank my advisor Professor Michael Carbin for all his help and guidance in shaping this thesis, for pushing me hard to do the best I can and more, and for all his pithy wisdom that I will never forget. I thank Eric Atkinson, Guillaume Baudart, and Louis Mandel for all their help and feedback in writing the paper and in shaping the technical development of the work. I thank Jesse Michel, Charles Yuan, Logan Weber, Alex Renda, and Tian Jin for providing helpful feedback. I thank Makar Loktyukhin for supporting me throughout it all. I thank my parents, my sisters, and my friends for encouraging me to pursue my ambitions.

The main contributions of this thesis are my work, except the following: The formalization of the syntax and the big-step semantics with checkpoints in Chapter 3 is primarily the work of Guillaume Baudart and Louis Mandel; The formalization and proof of soundness for the analysis in Section 4.4 is joint work with Eric Atkinson; Chapters 1, 2 and 5 benefited greatly from the writing and editing by Michael Carbin and Eric Atkinson. Eric Atkinson, Guillaume Baudart, Louis Mandel, and Michael Carbin provided detailed feedback throughout the work.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 2141064.

Contents

List of Figures

List of Tables

Chapter 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, control flow. PPLs serve as an expressive and accessible tool for solving such problems. Users can focus on modeling the problem, rather than the details of inference techniques.

1.1 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), automatically marginalized MCMC (Lai et al., 2023) – automatically incorporate exact inference with Monte Carlo methods to improve performance without exposing programmers to the details of the inference techniques. They utilize a *symbolic representation* of random variables to encode 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).

Figure 1.1 depicts hybrid inference using particle filtering as the approximate inference method. The algorithm maintains a collection of parallel instances of execution, represented by the 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 treated 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 point the system *resamples* the instances, meaning the system adjusts the distribution of instances based on the weights.

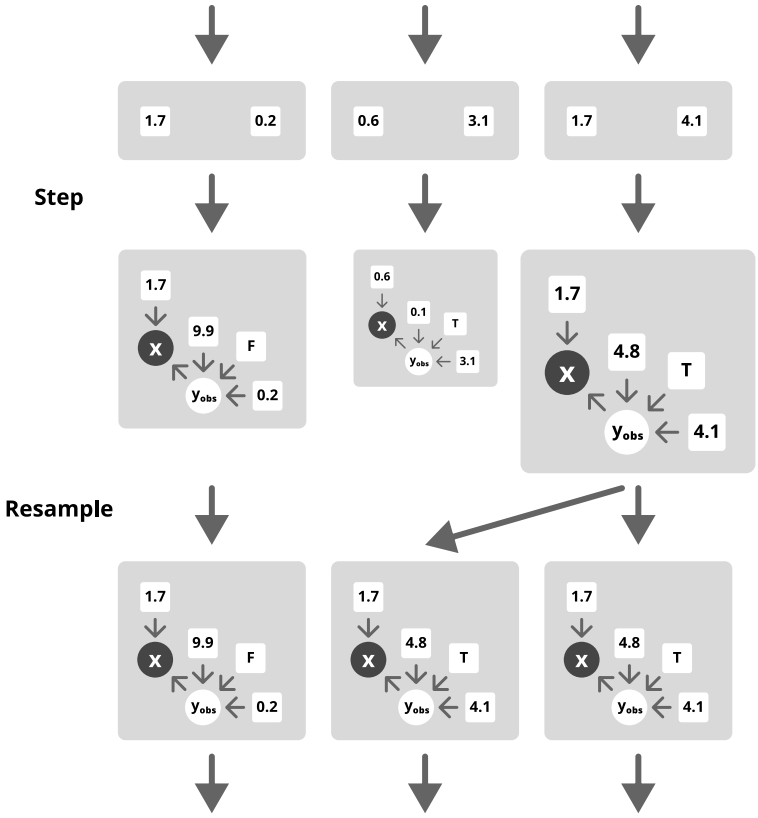


Figure 1.1: Hybrid inference with particle filtering.

1.2 No Global Guarantee

When hybrid inference systems cannot solve the entire program with symbolic computation, they use local heuristics to partition the random variables into variables that are represented symbolically throughout the program and variables that are represented by sampled constant values. Consequently, the systems make no general guarantee that the partitioning of how variables are represented is globally optimal. As this thesis empirically shows, the built-in heuristics do not always make partitions that produce the best performance. In fact, there might not be a globally optimal partition at all, so the better partition depends on the objective of the problem, which the heuristics are oblivious to. In these cases, users need a mechanism to guide the inference system to partition the random variables in a specific way.

1.3 Thesis

Programming constructs such as annotations or types can serve as a mechanism for developers to guide hybrid inference systems in choosing how to represent random variables. In this thesis, I investigate using annotations and program analyses to guide inference systems.

Inference Plans I present *inference plans*, a programming interface that enables developers to choose a specific partitioning of random variables during hybrid inference. Developers may add

distribution encoding annotations to encode constraints on whether random variables must be represented as symbolic expressions or as samples.

Satisfiability Analysis A key challenge to delivering this interface is that not all inference plans can be implemented by an inference algorithm. Some variable partitionings are *unsatisfiable* because the inference algorithm does not know how to analytically solve the encoded symbolic structure. An additional key challenge to reasoning about the satisfiability of an inference plan is that many hybrid inference algorithms dynamically perform local optimizations because some symbolic structures can only be fully exploited at runtime (Lai et al., 2023, Lundén, 2017). While the design choice to operate dynamically rather than statically enables these systems to identify more opportunities for symbolic optimizations than if they only reasoned statically about the program, the satisfiability of an inference plan is then only fully determined at runtime. To assist developers with statically reasoning about inference plans without requiring in-depth knowledge of the inference algorithm, I present a static analysis that identifies the satisfiability of inference plans.

Contributions In this thesis, I present the following contributions:

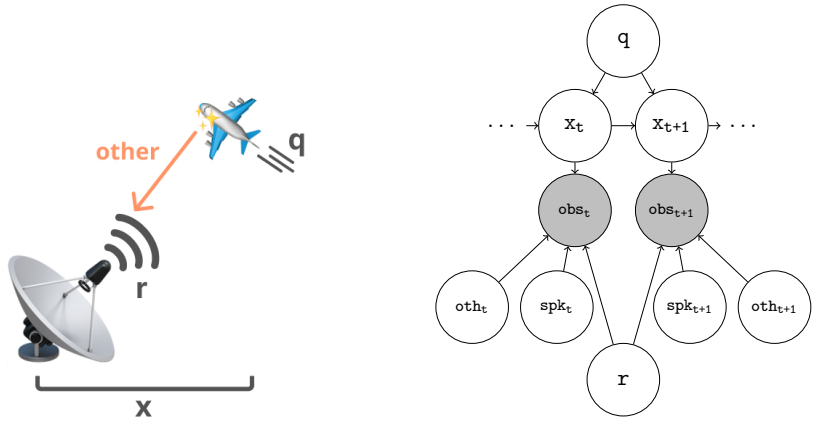
- I present Siren, a first-order functional PPL. Siren introduces distribution encoding annotations that programmers can use to assert an overall specification for how variables are represented by the inference algorithm; I term this specification an inference plan. Siren implements several existing hybrid inference systems, unified via the hybrid inference interface, an extension of the symbolic interface (Atkinson et al., 2022). I define the syntax and semantics of the language in Chapter 3.
- I present an *inference plan satisfiability analysis*, which determines statically if the user-provided inference plan is satisfiable for all executions of a program. I formalize this analysis via abstract interpretation and present a proof of its soundness in Chapter 4.
- I implement the interface with semi-symbolic inference, delayed sampling, and Sequential Monte Carlo with belief propagation, and empirically show in Chapter 5 that using custom inference plans can produce up to 1000x better accuracy compared to the default plan.
- I empirically evaluate the completeness of the analysis in Chapter 5. Against a suite of 7 benchmarks, using the 3 algorithms, the analysis identifies all satisfiable plans in 20 out of the 21 benchmark-algorithm combinations.

With inference plans and the satisfiability analysis, developers can customize how hybrid inference algorithms execute programs to improve performance without detailed knowledge of the algorithms.

Chapter 2

Example

To demonstrate how a developer can use inference plans to customize and improve inference performance, I present an example adapted from Wu (1993) and Li et al. (2014). Figure 2.1a shows a cartoon diagram of a radar tracker. The goal of the radar tracker is to plot the movement of an aircraft on a map by estimating the position of the aircraft as it moves 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 captures the uncertainty in the model's prior belief of the movement of the aircraft relative to the tracker's estimate in the last timestep. The measurement noise captures the fact that 1) radar measurements naturally have white noise and, additionally, 2) the angle of the aircraft can induce electromagnetic wave reflections that result in random, spiking noise in a measurement as well (Li et al., 2014).



(a) Diagram. (b) Graphical representation.

Figure 2.1: A radar tracker estimating the position `x` of an aircraft, subject to movement noise `q` and measurement noise, which is modeled by white noise `r` with a `spk` random chance of encountering spiking noise `other`.

Figure 2.1b presents the graphical representation of the radar tracker's probabilistic model. A series of $\texttt{x}_\texttt{t}$ random variables model the position at each timestep `t`. Every $\texttt{x}_\texttt{t}$ depend on the movement noise random variable `q`. The radar measurements are the observed variables $\texttt{obs}_\texttt{t}$. All of the $\texttt{obs}_\texttt{t}$ random variables depend on the white noise variable `r` and the spike noise variables $\texttt{spk}_\texttt{t}$ and $\texttt{other}_\texttt{t}$.

```
1  let step = fun (obs, (xs,q,r)) ->
2    let x0 = hd(xs) in
3    let symbolic x <- gaussian(x0+5.,q) in
4    let sample spk <- bernoulli(0.0001) in
5    let sample other <- invgamma(1.,1.) in
6    let v = if spk then r+other else r in
7    let () = observe(gaussian(x,v),obs) in
8    (cons(x,xs),q,r) in
9  let sample q <- invgamma(1.,1.) in
10 let sample r <- invgamma(1.,1.) in
11 fold_resample(step,data,([0.],q,r))
```

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

```
1  let step = fun (obs, (xs,q,r)) ->
2    let x0 = hd(xs) in
3    let sample x <- gaussian(x0+5.,q) in
4    let sample spk <- bernoulli(0.0001) in
5    let sample other <- invgamma(1.,1.) in
6    let v = if spk then r+other else r in
7    let () = observe(gaussian(x,v),obs) in
8    (cons(x,xs),q,r) in
9  let sample q <- invgamma(1.,1.) in
10 let symbolic r <- invgamma(1.,1.) in
11 fold_resample(step,data,([0.],q,r))
```

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

Figure 2.2: A program written in SIREN that implements target position tracker with smoothing.

2.1 Specification in SIREN

Figure 2.2 presents two implementations of the radar tracker as SIREN programs, each with different annotations that denote different inference plans. Each program models the movement noise `q` and white noise `r` defined on Lines 9 and 10 with Inverse-Gamma distributions. The `step` function defined on Lines 1-8 updates the predicted position and conditions the model on radar measurement data. The `x` positions are modeled by a Gaussian distribution with a mean equal to the sum of the previous position and a constant velocity and with variance equal to `q`. Lines 4 to 6 model the spiking measurement noise `v`. If `spk` is *true*, the program models the measurement noise as `r+other`; the variable `other` models the irregular spike as a separate Inverse-Gamma distribution from the white noise `r`. Otherwise, there is no spike in the noise, so the program models the measurement noise as only the white noise `r`. The measurement is modeled as a Gaussian distribution centered around the predicted position `x` and with the measurement noise as variance. Line 7 conditions the model on the measurement being equal to the data value `obs`. The `fold_resample` operation on Line 11 iterates `step` on the measurement data (provided by the free variable `data`) with an initial accumulator composed of an initial position and the noises `q` and `r`. The program returns the final accumulator: the list of estimated positions, the movement noise, and the white noise.

2.2 Inference Plans

The two programs in Figure 2.2 differ in only the annotations of the random variables on Lines 3 to 5, 9 and 10 with `symbolic` or `sample`. Each distribution encoding annotation directs the inference algorithm to encode the annotated random variable as either a symbolic expression encoding the specified distribution or a constant sample drawn from the distribution, respectively. SIREN executes a program using hybrid inference – a combination of particle filtering with symbolic computation. A particle is a program expression to be evaluated and a symbolic state encoding the symbolic distributions of random variables; random variables that are encoded as samples are treated as constant values in the expression. Each particle evaluates its program expression in parallel. The diagram in Figure 1.1 shows an execution of the program in Figure 2.2a.

For example, in one particle evaluation, on the first iteration, the variable `q` is bound to the

constant value 1.7, a random sample drawn from the Inverse-Gamma distribution on Line 9. Likewise, `r` is bound to 0.2. Then, the symbolic random variable X_x created on Line 3 has the symbolic distribution $\mathcal{N}(5., 1.7)$. This particle also has `spk` as *false*, so the observed variable on Line 7 has the symbolic distribution $\mathcal{N}(X_x, 0.2)$. In another particle, `q` is bound to 0.6, `r` to 3.1, `spk` to *true*, and `other` to 0.1. Then, X_x has the symbolic distribution $\mathcal{N}(5., 0.6)$ and the observed variable has $\mathcal{N}(X_x, 3.2)$, since `r+other` evaluates to 3.2

To evaluate the particle, the inference algorithm uses symbolic computation to solve symbolic expressions analytically where possible and falls back on sampling random variables in the expressions otherwise. In the example program, the algorithm solves for the analytical solution of the posterior symbolic distribution of X_x, making X_x a child of the observed variable. When a `symbolic` random variable has to be sampled because the algorithm cannot find a closed-form solution, the program throws an annotation violation error. The particle accumulates a weight from conditioning the model on input values. The evaluations pause at resampling checkpoints, where a new collection of particles is resampled from the existing collection. In Figure 2.2a, the resampling step occurs at the end of each `fold_resample` iteration. When all particles have been evaluated, SIREN returns the mixture distribution of the results using the weights.

Accuracy and Performance Analysis To deploy the radar tracker presented in Figure 2.2, the developer must ensure that it will achieve adequate performance at runtime. This task is challenging because predicting how many particles a particle filter requires to achieve adequate accuracy is difficult.[1] It is further complicated by the fact that the accuracy of hybrid inference depends unpredictably on the inference plan. The inference plan will determine which random variables the inference system samples and which ones it keeps symbolic. Keeping variables symbolic *can* reduce the number of particles the system requires to achieve adequate accuracy. However, this behavior is not guaranteed, although a developer can empirically determine an inference system's behavior by executing the program with multiple different inference plans to build a *performance profile*.

Performance Profile Figure 2.3 presents the performance profiles for the programs in Figure 2.2. These graphs present scatter plots of accuracy and run time 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`, `q`, or `r`. The red squares present the time and accuracy tradeoff for the default inference plan that makes no annotations. The green diamonds present the tradeoff for annotating with the *Symbolic x Inference Plan* from Figure 2.2a. The blue triangles present the tradeoff for the program annotated with the *Symbolic r Inference Plan* from Figure 2.2b. Additionally, the yellow crosses present the tradeoff for annotating with the *Sample All Plan*, where all random variables are given the `sample` annotation.

In the context of radar tracking, the most important variable is the target's position `x`, so a developer can conclude from the performance profile in Figure 2.3 that the *Symbolic x Inference Plan* is better than the other plans because it leads to the lowest error on the `x` random variable. However, this property is not universal. In a context where the developer is trying to estimate the noise `r`, the developer could conclude that the *Symbolic r Inference Plan* is better than the other two plans because its annotations enable the inference system to achieve lower error on the `r` random variable when the execution time is greater than 1 second. The heuristics of the inference

[1]The execution time of a particle filter is generally proportional to the number of particles used during execution.

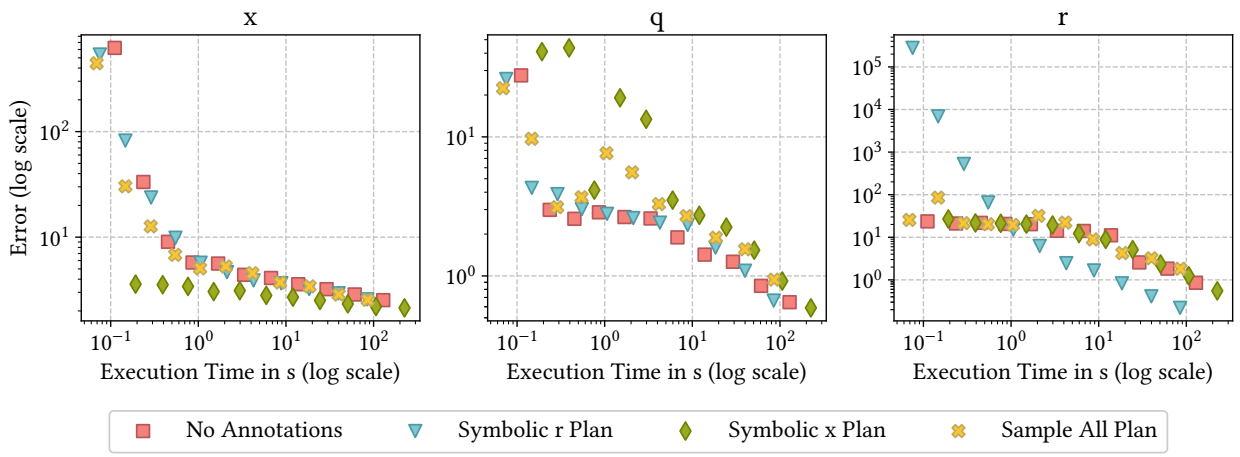


Figure 2.3: Accuracy and runtime performance of the example program from Figure 2.2. Each scatter plot presents the program execution time and accuracy for particle counts ranging from 1 to 1024 and multiple inference plans. Each data point measures an experiment that – across 100 runs – measures the median runtime and the 90th percentile of the Mean Squared Error for the relevant random variable – either `x`, `q`, or `r`.

algorithms are ultimately oblivious to the developer's objectives and cannot adapt to those metrics. This demonstrates the value of inference plans: They enable a developer to optimize the inference algorithm to achieve the best accuracy on the random variables relevant to their objectives.

Inference Plan Violations However, the annotated program in Figure 2.2b reveals a challenge with empirically determining the behavior of a hybrid inference algorithm that is also a randomized algorithm. In Figure 2.2b, if the inference system samples the variable `spk` and receives a result of *false* – which will happen 99.99% of the time according to `spk`'s distribution specified on Line 4 – then the observed variable on Line 7 has a symbolic Gaussian distribution with variance X_r. The Inverse-Gamma distribution of X_r is conjugate with the Gaussian distribution, which means that inference systems can find a closed-form solution to the model and can keep `r` symbolic. However, in the very rare case that the system samples `spk` and the result is *true*, the program specifies that the observed variable on Line 7 has a symbolic Gaussian distribution 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 distribution on Line 7. Without a conjugacy relationship to exploit, the inference system cannot solve the model analytically. Instead, it will need to sample the variable `r` to reduce the number of symbolic random variables and simplify the model such that it can find a closed-form solution. This produces a different inference plan in which all random variables are sampled i.e. the *Sample All Plan*.

The performance profiles in Figure 2.3 show that the *Sample All Plan* performs worse than the ideal *Symbolic r Plan* where `spk` is always *false* and `r` is always symbolic. This unexpected performance degradation resulted from an *assertion violation*; the developer annotated variable `r` as `symbolic`, but the inference system requires `r` to be `sample`d to solve the program. When there is an assertion violation in the program, SIREN will throw an exception to alert the developer of the system's deviation from the specified plan. Thus, the behavior of the program in Figure 2.2b

presents a major challenge to developers. Developers evaluating this program for a radar tracking system may not find such a rarely occurring exception during testing, leading them to believe their program is working properly when it could in fact suffer a catastrophic failure in production.

2.3 Inference Plan Satisfiability Analysis

When a program executes without failures, the annotations encode a *satisfiable* inference plan. To enable developers to test for this satisfiability property without repeatedly executing a program on a large space of potential inputs that is difficult to cover, SIREN performs the inference plan satisfiability static analysis to determine whether the annotated inference plan is satisfiable during all possible executions of the program using the hybrid inference algorithm.

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 program from Figure 2.2b. On Line 6, because the analysis does not know the exact value of `spk`, it uses an abstract value to over-approximate the subexpressions in the branches. If `spk` is *false*, the abstract subexpression is $\hat{X}_r$, referring to the abstract random variable from Line 10. If `spk` is *true*, the subexpression is $\hat{X}_r \mathbin{\hat{+}} \texttt{UnkC}$. The abstract value `UnkC` represents some unspecified constant. All random samples are constant values, so `other` evaluates to `UnkC`. The analysis over-approximates these subexpressions as a single abstract expression: $\texttt{UnkE}(\{\hat{X}_r\})$ – an unspecified abstract expression that references the random variables $\hat{X}_r$.

The analysis also performs an abstraction of how the inference system finds closed-form solutions. On Line 7, the observed variable has the abstract distribution $\widehat{\mathcal{N}}(\texttt{UnkC}, \texttt{UnkE}(\{\hat{X}_r\}))$. The variance $\texttt{UnkE}(\{\hat{X}_r\})$ represents any expressions that reference $\hat{X}_r$, including complex expressions such as $\hat{X}_r \mathbin{\hat{+}} \texttt{UnkC}$. The inference system cannot find closed-form solutions when the variance is a complex expression. So, $\hat{X}_r$ (i.e. the program variable `r`) must be sampled. Because `r` is annotated with `symbolic`, the analysis rejects the program, correctly identifying the inference plan as unsatisfiable.

The satisfiability analysis will always reject unsatisfiable inference plans, but it may be overly conservative and erroneously reject satisfiable inference plans. Nevertheless, the analysis is precise in practice. For example, it correctly determines the program in Figure 2.2a 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 a constant, the mean is a linear expression, and $\hat{X}_x$ also has a Gaussian symbolic distribution. This program only consists of linear-Gaussian distributions – a class of probabilistic models that the inference system can find closed-form solutions of. Then, `x` will always be kept symbolic as the annotation requires. No annotations are violated, so the analysis accepts the program. The developer can have confidence that the performance profile for the *Symbolic x Plan* in Figure 2.3 is representative of the inference system's behavior in production.

Using annotations, the developer can select an inference plan that produces better performance. With the satisfiability analysis, the developer can further guarantee the inference plan will never produce an annotation violation error 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.

Chapter 3

Language Syntax and Semantics

In this chapter, I present the syntax and semantics of the first-order functional PPL, SIREN, adapted from Staton (2017). I 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 3.1 presents the syntax of SIREN. The type t of an expression is either `unit`, `real`, `int`, a product type, or a list. An expression e is either a value v (constant; variable; pair; or the application of an operator, e.g., an arithmetic operation, a distribution, or a list), a function application, a conditional, or a local definition. To manipulate lists, I add the classic `fold` operator. SIREN also supports probabilistic operators. The expression `let` $x \leftarrow op(v)$ `in` e introduces a new local random variable x with distribution $op(v)$ to be used in expression e. Optionally, a `symbolic` or `sample` annotation adorns a random variable declaration. The `observe`(v_1, v_2) expression conditions the model on the variable with distribution v_1 having the value v_2. The `resample` operator instructs the program to perform resampling for particle filtering. The `fold_resample` operation used in Chapter 2 is syntactic sugar for applying `resample` at the end of each `fold` iteration. A program is a sequence of function declarations d followed by a main expression.

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

Figure 3.1: Syntax of the SIREN language.

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

Figure 3.2: Grammar of symbolic expressions.

$$
\begin{aligned}
\textsc{assume} &: A \times D \times G \to \mathcal{RV} \times G \\
\textsc{observe} &: \mathcal{RV} \times \mathbb{V} \times G \to G \times \mathbb{R} \\
\textsc{value} &: \mathcal{RV} \times G \to \mathbb{V} \times G \mid \textbf{fail} \\
\textsc{value}^* &: E \times G \to \mathbb{V} \times G \mid \textbf{fail}
\end{aligned}
$$

Figure 3.3: Hybrid Inference Interface.

3.2 Operational Semantics

Siren interprets the model described in a program 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, basic Monte Carlo methods launch a set of independent executions of the sampler, the *particles*, and return a categorical distribution that associates each value with its score. Unfortunately, for models involving multiple random variables, there is very little chance that a series of random guesses returns a meaningful result. To mitigate this issue, Siren implements a *particle filter*, an advanced weighted sampler that occasionally resamples the set of particles according to their score during executions. The resampling step duplicates the most probable particles and drops the least probable. Following (Lundén et al., 2021), I add an explicit `resample` operator to the language to enable programs to explicitly trigger resampling.[1] In this section, I present the operational semantics of Siren which is a big-step semantics extended with checkpoints for resampling.

3.2.1 Hybrid Inference Interface

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

Symbolic Expressions Figure 3.2 presents the grammar of symbolic expressions used by hybrid inference 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 3.2 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`. The meaning of the conditional operator is that $\texttt{ite}(E_i, E_t, E_e)$ returns the value of the expression E_t if the condition E_i is true, and otherwise returns the value of E_e.

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

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

Interface The hybrid inference interface uses four operations to manipulate the symbolic state, shown in Figure 3.3. The ASSUME operation takes an annotation, a distribution, and a symbolic state, and returns a new random variable with the given distribution and 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. It must also check the annotation for violation; if the annotation is `symbolic`, it throws a runtime error. The SIREN semantics also makes use of the operation, VALUE*, which calls VALUE on all random variables in the given symbolic expression and fully evaluates it into a constant value. It propagates failure if any VALUE call fails. The VALUE* operation is the same across implementations. I will discuss different implementations of the interface in Section 3.3.

3.2.2 Big-step Semantics with Checkpoints

Next, I present the semantics of SIREN. Figures 3.4 and 3.5 show the semantics formalized as a big-step semantics extended with checkpoints. 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 $\{e_i, g_i\}_{1 \le i \le N} \downdownarrows D$ 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 3.4 shows the particle evaluation rules. A constant v is already fully reduced so the resample flag r is set to *true*. 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. To evaluate a function call $f(v)$, the rule evaluates the function body. 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 with VALUE* and uses it to decide which branch to execute. The semantics of a local declaration `let` x `=` e_1 `in` e_2 depends on `resample` operators. If there is a `resample` in e_1 the

$$\frac{}{v, g \downarrow^{false} v, g, 1} \qquad \frac{}{\texttt{resample}, g \downarrow^{true} \texttt{unit}, g, 1} \qquad \frac{\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x \rightarrow\ e \qquad e[x \leftarrow v], g \downarrow^{r} e', g', w}{f(v), g \downarrow^{r} e', g', w}$$

$$\frac{\text{PURE}(e_1, e_2) \qquad \neg\text{CONST}(v) \qquad e_1, g \downarrow^{false} v_1, g_1, 1 \qquad e_2, g_1 \downarrow^{false} v_2, g', 1}{\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2, g \downarrow^{false} \texttt{ite}(v, v_1, v_2), g', 1}$$

$$\frac{\text{VALUE}^*(v, g) = \mathit{true}, g_v \qquad e_1, g_v \downarrow^{r} e_1', g', w}{\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2, g \downarrow^{r} e_1', g', w} \qquad \frac{\text{VALUE}^*(v, g) = \mathit{false}, g_v \qquad e_2, g_v \downarrow^{r} e_2', g', w}{\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2, g \downarrow^{r} e_2', g', w}$$

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

$$\frac{}{\texttt{fold(}f\texttt{,}\ \texttt{nil}\texttt{,}\ v\texttt{)}, g \downarrow^{false} v, g, 1} \qquad \frac{\texttt{let}\ x\ \texttt{=}\ f\texttt{((}l_{hd}\texttt{,}\ v\texttt{))}\ \texttt{in}\ \texttt{fold(}f\texttt{,}\ l_{tl}\texttt{,}\ x\texttt{)}, g \downarrow^{r} e, g', w}{\texttt{fold(}f\texttt{,}\ \texttt{cons(}l_{hd}\texttt{,}\ l_{tl}\texttt{)}\texttt{,}\ v\texttt{)}, g \downarrow^{r} e, g', w}$$

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

$$\frac{\text{ASSUME}(\texttt{sample}, \mathit{dist}(v), g) = X, g_X \qquad \text{VALUE}(X, g_X) = v_x, g_X' \qquad e[x \leftarrow v_x], g_X' \downarrow^{r} e', g', w}{\texttt{let sample}\ x \leftarrow \mathit{dist}(v)\ \texttt{in}\ e, g \downarrow^{r} e', g', w}$$

$$\frac{\text{ASSUME}(\varepsilon, \mathit{dist}(v_1), g) = X, g_X \qquad \text{VALUE}^*(v_2, g_X) = v, g_v \qquad \text{OBSERVE}(X, v, g_v) = g', w}{\texttt{observe(}\mathit{dist}(v_1)\texttt{,}\ v_2\texttt{)}, g \downarrow^{false} \texttt{unit}, g', w}$$

Figure 3.4: Particle evaluation rules.

$$\frac{\bigwedge_{1 \le i \le N} (v_i \neq \textbf{fail}) \qquad \left\{e_i, g_i \downarrow^{false} v_i, g_i', w_i\right\}_{1 \le i \le N} \qquad \left\{\text{DISTRIBUTION}(v_i, g_i') = D_i\right\}_{1 \le i \le N} \qquad W = \sum_{1 \le i \le N} w_i}{\{e_i, g_i\}_{1 \le i \le N} \Downarrow \sum_{1 \le i \le N} \frac{w_i}{W} \times D_i}$$

$$\frac{\bigvee_{1 \le i \le N} r_i \qquad \bigwedge_{1 \le i \le N} (e_i' \neq \textbf{fail}) \qquad \mu = \text{CAT}\left(\left\{w_i, (e_i', g_i')\right\}_{1 \le i \le N}\right) \qquad \left\{\text{DRAW}(\mu)\right\}_{1 \le i \le N} \Downarrow v}{\{e_i, g_i\}_{1 \le i \le N} \Downarrow v}$$

$$\frac{\left\{e_i, g_i \downarrow^{r_i} e_i', g_i', w_i\right\}_{1 \le i \le N} \qquad \bigvee_{1 \le i \le N} (e_i' = \textbf{fail})}{\{e_i, g_i\}_{1 \le i \le N} \Downarrow \textbf{fail}}$$

Figure 3.5: Particle set evaluation rules.

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. Then, e_2 is evaluated and the total score is the product of the score of e_1 and e_2. The expression `fold(`f`,` l`,` v`)` is a standard `fold`, where the expression 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.

A random variable `let` κ $x \leftarrow$ *dist*`(`v`)` `in` e uses ASSUME to create a new random variable in the symbolic state. The annotation `sample` denotes that the variable must be represented using samples, `symbolic` denotes it must be represented 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 3.5 shows the particle set evaluation rules. The rules handle resuming execution from checkpoints on a set of N particles. There are three possible cases during evaluation. If all the particles finish execution (i.e. the resample flag is *false*), the semantics gather the computed distributions into a mixture distribution where DISTRIBUTION(v, g) returns the distribution of v with respect to the symbolic state g and w_i/W are the normalized scores. If one of the particles stopped at a resample step, the rule builds a categorical distribution μ of particles using the weights and resamples a fresh set of particles $\{\text{DRAW}(\mu)\}$ before resuming the execution. Lastly, the entire evaluation immediately fails if any particle fails.

Model Evaluation To evaluate the main 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 or **fail**.

$$\frac{\{e, \emptyset\}_{1 \leq i \leq N} \Downarrow v}{e \Downarrow_N v}$$

3.3 Implementing the Hybrid Inference Interface

The hybrid inference interface can be implemented by different hybrid inference algorithms. The full implementations of these algorithms are quite complex, so I defer the full details to the respective works. To provide an intuition of how these algorithms implement hybrid inference and incorporate the distribution encoding annotations, I summarize here how semi-symbolic inference (Atkinson et al., 2022) and delayed sampling (Lundén, 2017, Murray et al., 2018) implements the interface operations and the data field.

3.3.1 Semi-symbolic Inference

In semi-symbolic inference (SSI), a random variable can only be sampled and scored if it is a *root* – a variable that does not have any upstream dependencies. The core of SSI is using a *hoisting* algorithm that turns random variables into root variables. I describe here the *hoisting* algorithm and how the hybrid inference interface is implemented, but defer auxiliary details to Atkinson et al. (2022).

Hoisting Algorithm The *hoisting* algorithm takes a random variable and symbolic state and returns the updated state wherein the input variable is now a root variable in the graph. To achieve this, the algorithm recursively swaps the parent-child dependency relationship between random variables by exploiting conjugate priors (Fink, 1997). A conjugate prior is a parent distribution to a child distribution that has a known analytical solution for rewriting the distributions such that the dependency relationship between the two random variables is reversed. The algorithm performs the swap by manipulating the symbolic distributions in the symbolic state, keeping the random variables represented as symbolic expressions. When the algorithm encounters a parent-child relationship that it cannot swap, it samples the parent using VALUE. Sampling the parent removes the dependency since the parent variable now represents a constant value rather than a symbolic distribution.

Algorithm 1 HOIST: Hoisting a random variable to be a root depending on no other variables.

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

I present the definition of HOIST in Algorithm 1. The HOIST_HELPER operation recursively calls itself on the parents of X_{cur} in topological order. The topological ordering is a requirement to prevent creating cycles in the dependencies of the random variables as I perform swaps. After each recursive call, the hoisted parent is added to the next calls' `roots` set. Members of the `roots` set are

not hoisted or swapped. The `roots` set enforces that subsequent parents will only be descendants of the earlier parents so no cycles will be created. Then, HOIST_HELPER iterates through all parents in reverse topological order and swaps X_{cur} with each. If X_{cur} cannot be swapped with one of its parents, the operation throws an exception that is caught at the outermost level. The HOIST operation calls HOIST_HELPER with an empty `roots` set, thus turning X_{in} into a root. In the event of an exception, it forces the parent of the offending variable to be sampled using the VALUE operation. It then uses the EVAL* operation to eliminate the resulting Delta distribution from the symbolic state. After thus eliminating this variable, it again attempts to HOIST the variable X_{in}.

The HOIST function relies on the SWAP function to detect conjugacies and invert the parent-child relationship between random variables. I summarize the workings of SWAP here to show how SSI decides whether to keep a random variable as a symbolic expression or to sample it.

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

The operation can be extended with any pair of conjugate distributions; here I 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. 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 with a *true* flag indicating a swap occurred. Otherwise, it returns the state unchanged and *false*. If SWAP returns false, HOIST calls VALUE to sample the parent variable.

Interface SSI implements the hybrid interface using HOIST. First, the ASSUME operation returns a new random variable and the updated state. SSI does not use the data field of the symbolic state.

$$
\begin{aligned}
\text{ASSUME}(\kappa, D, g) = \; & \textit{let } g' = g[X_{\text{new}} \mapsto (\kappa, D)] \textit{ in} \\
& (X_{\text{new}}, g')
\end{aligned}
$$

The VALUE operation replaces a random variable with a sample from its distribution. The INTERVENE operation updates the distribution of the given variable in the symbolic state.

$$
\begin{aligned}
\text{VALUE}(X, g) = \; & \textit{if } g(X)_{\text{a}} = \texttt{symbolic} \textit{ then } \textbf{fail} \\
& \textit{else let } g' = \text{HOIST}(X, g) \textit{ in let } v = \text{DRAW}(g'(X)) \textit{ in} \\
& (v, \text{INTERVENE}(X, \delta_s(v), g'))
\end{aligned}
$$

If the random variable has a `symbolic` annotation, VALUE throws a runtime annotation violation error. Otherwise, VALUE hoists the variable X with HOIST, turning it into a root. The function then draws a sample from the root random variable's distribution. Finally, the INTERVENE function uses sampled-Delta distribution to mark X as a random variable that was reduced to a sample.

The OBSERVE operation also uses HOIST to turn the input random variable into a root variable. Then, it computes the score, which is the probability density of variable's distribution at the input value. It updates the symbolic state with the input value.

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

3.3.2 Delayed Sampling

Delayed sampling (DS) is an alternative hybrid inference algorithm that also exploits conjugacy relationships (Lundén, 2017). It is an alternative implementation of the interface that does not reverse parent-child relationships between random variables. DS uses the symbolic state to represent the model as a forest of disjoint trees, where each node is a random variable. The variables maintained symbolically in the tree are all conjugate priors to their children.

Associated with each node in the tree are pointers to its children as well as its prior distribution before any observing or sampling, all represented with symbolic expressions. DS specifies three types of nodes: Initialized, Marginalized, and Realized nodes. Node types are used by DS to determine whether a variable needs to be marginalized. Initialized nodes represent random variables that have a conditional distribution dependent on their parent; Marginalized nodes represent variables that have marginal distributions, and may have an optional prior distribution (and associated 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 $g(X)_{\text{n}}$ to track the node type for each random variable, where $S_{rv} \subseteq \mathcal{RV}$ are the children of the node:

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

Grafting Algorithm DS may only observe or sample a random variable that is a *terminal* node (i.e. a Marginalized node with no Marginalized children). DS turns variables into terminal nodes with a *grafting* algorithm that recursively converts Initialized nodes into Marginalized nodes by symbolically marginalizing the parent of Initialized nodes and removing edges to its Marginalized child by sampling. The GRAFT operation uses the node type to determine which operations convert the node into a terminal node: If the node is a Marginalized node, GRAFT retrieves its Marginalized child and removes the edge to that subtree using PRUNE, which recursively samples Marginalized children using VALUE; if the node is an Initialized node, GRAFT ensures the parent node is also terminal by recursively calling GRAFT and then symbolically marginalizing with MARGINALIZE. At the end of GRAFT, the variable X is a terminal node. I defer further details of GRAFT to Lundén (2017) and Murray et al. (2018).

$$\begin{aligned}\textsc{graft}(X, g) = \; & \textit{match } g(X)_{\text{n}} \textit{ with} \\ & \mid \texttt{marginalized}(S_{rv}) \mid \texttt{marginalized}(X', D, S_{rv}) : \\ & \quad \textit{let } X_{\text{child}} = \textsc{marginalized\_child}(S_{rv}) \textit{ in } \textsc{prune}(X_{\text{child}}, g) \\ & \mid \texttt{initialized}(X', S_{rv}) : \\ & \quad \textit{let } g' = \textsc{graft}(X', g) \textit{ in } \textsc{marginalize}(X, g')\end{aligned}$$

Interface DS requires additional helper functions to implement the interface. To implement the ASSUME operation, DS uses CONJUGATE_DIST to ensure the new random variable has at most one

parent and that the parent must be a conjugate prior. If it has more than one parent or the parent is not a conjugate prior, those parents are sampled with VALUE. Then, ASSUME inserts the variable and its annotation into the symbolic state with the appropriate node type using INITIALIZE.

$$\begin{aligned}\textsc{assume}(\kappa, D, g) = \;& \textit{let } D', g' = \textsc{conjugate\_dist}(D, g) \textit{ in} \\ & (X_{\text{new}}, \textsc{initialize}(X_{\text{new}}, \kappa, D', g'))\end{aligned}$$

DS implements the VALUE and OBSERVE operation by first using GRAFT to turn the variable into a terminal node, before drawing a sample or computing the score. The REALIZE operation updates the symbolic state with the sample and makes the node a Realized node.

$$\begin{aligned}\textsc{value}(X, g) = \;& \textit{if } g(X)_{\text{a}} = \texttt{symbolic} \textit{ then } \textbf{fail} \\ & \textit{else let } g' = \textsc{graft}(X, g) \textit{ in let } v = \textsc{draw}(g'(X)) \textit{ in} \\ & (v, \textsc{realize}(X, \delta_s(v), g')) \\ \textsc{observe}(X, v, g) = \;& \textit{let } g' = \textsc{graft}(X, g) \textit{ in let } s = \textsc{score}(g'(X)) \textit{ in} \\ & (\textsc{realize}(X, \delta(v), g'), s)\end{aligned}$$

Chapter 4

Inference Plans

Using the `symbolic` and `sample` distribution encoding annotations, developers can express an *inference plan* specifying their requirements for how each random variable is represented. SIREN samples `sample` annotated variables upon instantiation. The hybrid inference interface dynamically enforces the `symbolic` annotation with a check in the VALUE operation and throws an annotation violation error if a `symbolic` variable is sampled. If a program annotated with an inference plan executes without any annotation violations, the inference plan is *satisfiable*.

Definition 4.0.1 (Satisfiable Inference Plan)**.** Given a program e and N particles, the inference plan is *satisfiable* if $e \Downarrow_N v$ and $v \neq$ **fail**.

For hybrid inference algorithms that are also randomized algorithms, the satisfiability of an inference plan depends on the runtime execution. This means that when a programmer annotates the program with an inference plan, 1) the prescribed inference plan may not be satisfiable, 2) the satisfiability is determined by running the program, and 3) the satisfiability is not guaranteed to be the same across executions due to data-dependent control flow.

I next present the *inference plan satisfiability analysis*, which statically determines inference plan satisfiability for all possible runtime executions of a SIREN program. The analysis is unified across different hybrid inference algorithms using the hybrid inference interface. I formalize the analysis as an abstract interpretation and prove the soundness of the analysis.

4.1 Abstract Hybrid Inference

The analysis performs an abstract interpretation of the program with an abstraction of the hybrid inference interface. I 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 $\widehat{\text{OBSERVE}}$ does not perform scoring or sampling.

Abstract Symbolic Expressions Figure 4.1 shows the grammar of the 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

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

Figure 4.1: Grammar of abstract symbolic expressions. Grayed-out expressions are identical to those in Figure 3.2.

variables; the $\texttt{UnkE}(\hat{S}_{rv})$ expression represents all expressions that reference any number of the random variables in $\hat{S}_{rv}$. Likewise, abstract distributions also can be $\texttt{TopD}$ or $\texttt{UnkD}(\hat{S}_{rv})$.

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

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

The abstract expression $\texttt{UnkC}$ subsumes all constants and is itself considered a constant. Abstract random variables subsume constants. The $\texttt{UnkE}(\{\hat{X}\})$ expression subsumes variable $\hat{X}$. The $\texttt{UnkE}(\hat{S}_{rv})$ expression is a refinement of the top expression $\texttt{TopE}$ 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 $\texttt{TopD}$. A plus expression subsumes another plus expression if the subexpressions also subsumes the subexpressions of the other expression. Other complex expressions and distributions are symmetric.

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 least upper bound 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 heuristics of the hybrid inference algorithms depend on identifying expressions of certain classes (e.g. linear-Gaussians).

Abstract Symbolic State An abstract symbolic state $\hat{g}$ is a finite mapping of abstract random variables 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 what entries exist in the concrete state. For example, $\hat{g}(\hat{X})_{\mathrm{d}} = \widehat{\mathcal{N}}(\texttt{UnkC}, \texttt{UnkC})$ requires the concrete state to map the corresponding random 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: $\varepsilon \leq \widehat{\texttt{sample}} \leq \widehat{\texttt{symbolic}}$.

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

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

I also define a weak equivalence between two abstract symbolic states that only compares random variables that are reachable from the given expression. Additional unreachable random variables in a symbolic state do not affect execution, so abstract symbolic states are equivalent for the given expression if all reachable variables from the expression have the same entries.

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

Precision of Joining Expressions When a program contains data-dependent 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. This can lose 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.

For example, consider the program in Figure 4.2, 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 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 $\texttt{UnkE}(\{\hat{X}_1, \hat{X}_2\}) \mathbin{\hat{+}} \texttt{UnkC}$. The analysis concludes the resulting abstract observed Gaussian is not necessarily linear-Gaussian, as the $\texttt{UnkE}(\{\hat{X}_1, \hat{X}_2\})$ expression also represents expressions that are non-linear to $\hat{X}_1$ and $\hat{X}_2$. Consequently, the analysis 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. Ideally, the join of the expressions should maintain that both are linear-Gaussians.

```
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)
```

Figure 4.2: Example program.

I define a special operation for joining expressions that takes advantage of the fact abstract random variables can be renamed without compromising soundness, which I prove in Section 4.4. 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. I 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:

$$\text{RENAME\_JOIN}(\hat{E}_1, \hat{E}_2, \hat{g}_1, \hat{g}_2) = \textit{let}\ (\hat{E}_3, \hat{g}_3) = \text{RENAME}(\hat{E}_1, \hat{E}_2, \hat{g}_2)\ \textit{in}\ (\hat{E}_1 \sqcup \hat{E}_3, \hat{g}_1 \sqcup \hat{g}_3)$$

where $\sqcup$ refers to the basic join operation implied by the partial orders for abstract expressions and abstract symbolic states. The $\text{RENAME}(\hat{E}_1, \hat{E}_2, \hat{g}_2)$ 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 4.2, the analysis renames the random variable $\hat{X}_2$ to $\hat{X}_1$. The joined expression is the more structurally precise expression $\hat{X}_1 \hat{+} \texttt{UnkC}$, and the joined state assigns $\hat{X}_1$ to $\widehat{\mathcal{N}}(\texttt{UnkC}, \hat{1})$. Then, the observed random variable has the distribution $\widehat{\mathcal{N}}(\hat{X}_1 \hat{+} \texttt{UnkC}, \hat{5})$. RENAME_JOIN retains the shared structure in the expressions, and analysis recognizes the observed variable as a linear-Gaussian.

Fail Throughout execution, the program may produce an annotation violation error, represented as **fail**. The abstract equivalent, $\widehat{\textbf{fail}}$ is the top of all abstract values, such that any value joined with $\widehat{\textbf{fail}}$ results in $\widehat{\textbf{fail}}$. Any operation that receives $\widehat{\textbf{fail}}$ as inputs also returns $\widehat{\textbf{fail}}$ as the output.

4.2 Abstract Interpretation Rules

Figures 4.3 to 4.5 present the interpretation rules of a SIREN program using the abstract hybrid inference operations. Most of the abstract interpretation rules follow from the concrete semantics. I discuss the rules that differ in detail in this section.

Conditionals The analysis mirrors the concrete semantics when the subexpressions are pure and the condition is not constant, and when $\widehat{\text{VALUE}}^*$ returns $\widehat{\textit{true}}$ or $\widehat{\textit{false}}$. When $\widehat{\text{VALUE}}^*$ returns UnkC, the analysis cannot determine which branch 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 $\texttt{UnkE}(\emptyset)$, the analysis over-approximates the operation by computing the 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 $\texttt{UnkE}(\hat{S}_{rv})$ or TopE, which also over-approximate any particular item in the list. 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 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 $\texttt{UnkE}(\hat{S}_{rv})$ expressions. However, $\hat{S}_{rv}$ can grow arbitrarily large, so the analysis widens the joined expression by converting $\texttt{UnkE}(\hat{S}_{rv})$ to TopE if $|\hat{S}_{rv}| \geq N$ for some parameter N. My 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 particle set interpretation rule applies the particle evaluation rules and extracts the resulting distribution. The abstract interpretation of a program is then simply whether the abstract particle evaluation rules encounter failures or not. If they do not encounter failure, the model evaluation – and the analysis – return the judgment that the program is *satisfiable*.

$$\frac{}{\hat{v}, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}, \hat{g}} \qquad \frac{}{\texttt{resample}, \hat{g} \mathbin{\hat{\downarrow}} \widehat{\texttt{unit}}, \hat{g}} \qquad \frac{\texttt{let } f \texttt{ = fun } x \to\ e \qquad e[x \leftarrow \hat{v}], \hat{g} \mathbin{\hat{\downarrow}} \hat{v}', \hat{g}'}{f(\hat{v}), \hat{g} \mathbin{\hat{\downarrow}} \hat{v}', \hat{g}'}$$

$$\frac{\textsc{Pure}(e_1, e_2) \qquad \neg\widehat{\textsc{Const}}(\hat{v}) \qquad e_1, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}_1, \hat{g}_1 \qquad e_2, \hat{g}_1 \mathbin{\hat{\downarrow}} \hat{v}_2, \hat{g}'}{\texttt{if } \hat{v} \texttt{ then } e_1 \texttt{ else } e_2, \hat{g} \mathbin{\hat{\downarrow}} \widehat{\texttt{ite}}(\hat{v}, \hat{v}_1, \hat{v}_2), \hat{g}'}$$

$$\frac{\widehat{\textsc{Value}}^*(\hat{v}, \hat{g}) = \widehat{\mathit{true}}, \hat{g}_{\hat{v}} \qquad e_1, \hat{g}_{\hat{v}} \mathbin{\hat{\downarrow}} \hat{v}_1, \hat{g}'}{\texttt{if } \hat{v} \texttt{ then } e_1 \texttt{ else } e_2, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}_1, \hat{g}'} \qquad \frac{\widehat{\textsc{Value}}^*(\hat{v}, \hat{g}) = \widehat{\mathit{false}}, \hat{g}_{\hat{v}} \qquad e_2, \hat{g}_{\hat{v}} \mathbin{\hat{\downarrow}} \hat{v}_2, \hat{g}'}{\texttt{if } \hat{v} \texttt{ then } e_1 \texttt{ else } e_2, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}_2, \hat{g}'}$$

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

$$\frac{e_1, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}_1, \hat{g}_1 \qquad e_2[x \leftarrow \hat{v}_1], \hat{g}_1 \mathbin{\hat{\downarrow}} \hat{v}_2, \hat{g}_2}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}_2, \hat{g}_2} \qquad \frac{}{\texttt{fold(}f\texttt{, nil, } \hat{v}\texttt{)}, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}, \hat{g}}$$

$$\frac{\texttt{let } x \texttt{ = } f\texttt{((}\hat{l}_{hd}\texttt{, } \hat{v}\texttt{)) in fold(}f\texttt{, } \hat{l}_{tl}\texttt{, } x\texttt{)}, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}, \hat{g}'}{\texttt{fold(}f\texttt{, cons(}\hat{l}_{hd}\texttt{, } \hat{l}_{tl}\texttt{), } \hat{v}\texttt{)}, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}, \hat{g}'}$$

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

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

$$\frac{\kappa \in \{\varepsilon, \texttt{symbolic}\} \qquad \widehat{\textsc{Assume}}(\kappa, \mathit{dist}(\hat{v}), \hat{g}) = \hat{X}, \hat{g}_{\hat{X}} \qquad e[x \leftarrow \hat{X}], \hat{g}_{\hat{X}} \mathbin{\hat{\downarrow}} \hat{v}', \hat{g}'}{\texttt{let } \kappa\ x \leftarrow \mathit{dist}(\hat{v}) \texttt{ in } e, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}', \hat{g}'}$$

$$\frac{\widehat{\textsc{Assume}}(\widehat{\texttt{sample}}, \mathit{dist}(\hat{v}), \hat{g}) = \hat{X}, \hat{g}_{\hat{X}} \qquad \widehat{\textsc{Value}}(\hat{X}, \hat{g}_{\hat{X}}) = \hat{v}_x, \hat{g}'_{\hat{X}} \qquad e[x \leftarrow \hat{v}_x], \hat{g}'_{\hat{X}} \mathbin{\hat{\downarrow}} \hat{v}', \hat{g}'}{\texttt{let sample } x \leftarrow \mathit{dist}(\hat{v}) \texttt{ in } e, \hat{g} \mathbin{\hat{\downarrow}} \hat{v}', \hat{g}'}$$

$$\frac{\widehat{\textsc{Assume}}(\varepsilon, \mathit{dist}(\hat{v}_1), \hat{g}) = \hat{X}, \hat{g}_{\hat{X}} \qquad \widehat{\textsc{Value}}^*(\hat{v}_2, \hat{g}_{\hat{X}}) = \hat{v}, \hat{g}_{\hat{v}} \qquad \widehat{\textsc{Observe}}(\hat{X}, \hat{v}, \hat{g}_{\hat{v}}) = \hat{g}'}{\texttt{observe(}\mathit{dist}(\hat{v}_1)\texttt{, } \hat{v}_2\texttt{)}, \hat{g} \mathbin{\hat{\downarrow}} \widehat{\texttt{unit}}, \hat{g}'}$$

Figure 4.3: Abstract interpretation rules for particle evaluation.

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

Figure 4.4: Abstract particle set evaluation.

$$\frac{\{e, \emptyset\} \mathbin{\hat{\Downarrow}} \hat{D}}{e \Downarrow \mathit{satisfiable}} \qquad \frac{\{e, \emptyset\} \mathbin{\hat{\Downarrow}} \widehat{\textbf{fail}}}{e \Downarrow \widehat{\textbf{fail}}}$$

Figure 4.5: Abstract model evaluation.

4.3 Implementing the Abstract Hybrid Inference Interface

In this section, I define the abstract counterparts to the operations presented in Section 3.3 and summarize the implementations of the abstract hybrid inference interface for semi-symbolic inference and delayed sampling.

4.3.1 Semi-symbolic Inference

The abstract interpretation of SSI uses an abstract interpretation of the hoisting algorithm to implement the interface.

Hoisting Algorithm The abstract $\widehat{\text{HOIST}}$ operation, shown in Algorithm 2, is the same algorithm as the concrete HOIST, where the helper functions are replaced with their abstract counterparts. The helper function $\widehat{\text{GET\_PARENTS}}$ returns parent random variables of $\hat{X}_{\text{cur}}$.

Algorithm 2 Abstract HOIST.

function $\widehat{\text{HOIST\_HELPER}}(\hat{X}_{\text{cur}},\ \texttt{roots}, \hat{g})$
  `parents` $\leftarrow \text{TOPO\_SORT}(\widehat{\text{GET\_PARENTS}}(\hat{X}_{\text{cur}}, \hat{g}))$
  `roots'` $\leftarrow$ `roots`
  $\hat{g}' \leftarrow \hat{g}$
  for $\hat{X}_{\text{par}} \in$ `parents` **do**
    if $\hat{X}_{\text{par}} \notin$ `roots` **then**
      $\hat{g}' \leftarrow \widehat{\text{HOIST\_HELPER}}(\hat{X}_{\text{par}},\ \texttt{roots'}, \hat{g}')$
      `roots'` $\leftarrow \hat{X}_{\text{par}} ::$ `roots'`
    end if
  end for
  $\hat{g}'' \leftarrow \hat{g}'$
  for $\hat{X}_{\text{par}} \in \text{REVERSE}($`parents`$)$ **do**
    if $\hat{X}_{\text{par}} \notin$ `roots'` **then**
      $(\hat{g}'', \texttt{conjugate}) \leftarrow \widehat{\text{SWAP}}(\hat{X}_{\text{par}}, \hat{X}_{\text{cur}}, \hat{g}'')$
      if not `conjugate` **then**
        throw $(\hat{X}_{\text{par}}, \hat{X}_{\text{cur}})$
      end if
    end if
  end for
  return $\hat{g}''$
end function
function $\widehat{\text{HOIST}}(\hat{X}_{\text{in}}, \hat{g})$
  try
    return $\widehat{\text{HOIST\_HELPER}}(\hat{X}_{\text{in}}, \{\}, \hat{g})$
  catch $(\hat{X}_{\text{par}}, \hat{X}_{\text{child}})$
    $(\_, \hat{g}') \leftarrow \widehat{\text{VALUE}}(\hat{X}_{\text{par}}, \hat{g})$
    $\hat{g}'' \leftarrow \widehat{\text{EVAL}}^*(\hat{X}_{\text{child}}, \hat{g}')$
    return $\widehat{\text{HOIST}}(\hat{X}_{\text{in}}, \hat{g}'')$
  end try
end function

The abstract version of the SWAP operation simulates the concrete function by detecting conjugacy and performing computation where it can, as defined in Section 3.3. I show the match case for linear-Gaussians.

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

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 $\text{SET\_TOP}(\hat{X}_1)$. 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 the variable will be sampled), so the analysis conservatively returns **fail**. If $\widehat{\text{SWAP}}$ returns $\widehat{\textit{false}}$, the $\widehat{\text{HOIST}}$ operation calls $\widehat{\text{VALUE}}$ on the parent variable like in the concrete implementation. Then, the analysis simulates the runtime heuristics of SSI and detects if an $\widehat{\texttt{symbolic}}$ variable might be sampled. The analysis does not consider the expressions $\texttt{UnkE}(\hat{S}_{rv})$ and `TopE` affine. If abstract expressions are not precise enough, the analysis would not match with any distributions in $\widehat{\text{SWAP}}$ and would conservatively reject the program.

Interface The implementations of the abstract interface operations depend on the abstract counterparts of the operations implementing the concrete interface. The operation $\widehat{\text{ASSUME}}$ is the same as ASSUME but it operates over the abstract domain.

$$
\begin{aligned}
\widehat{\text{ASSUME}}(\hat{\kappa}, \hat{D}, \hat{g}) = \; & \textit{let } \hat{g}' = \hat{g}[\hat{X}_{\text{new}} \mapsto (\hat{\kappa}, \hat{D})] \textit{ in} \\
& (\hat{X}_{\text{new}}, \hat{g}')
\end{aligned}
$$

In $\widehat{\text{VALUE}}$, instead of drawing values, the abstract `UnkC` value is used instead to represent constant values. $\widehat{\text{INTERVENE}}$ updates the abstract symbolic state with the given abstract distribution, similar to INTERVENE. The $\widehat{\text{OBSERVE}}$ operation does not perform scoring.

$$
\begin{aligned}
\widehat{\text{VALUE}}(\hat{X}, \hat{g}) = \; & \textit{if } \hat{g}(\hat{X})_{\text{a}} = \widehat{\texttt{symbolic}} \textit{ then } \widehat{\textbf{fail}} \\
& \textit{else let } \hat{g}' = \widehat{\text{HOIST}}(\hat{X}, \hat{g}) \textit{ in} \\
& (\texttt{UnkC}, \widehat{\text{INTERVENE}}(\hat{X}, \hat{\delta}_s(\texttt{UnkC}), \hat{g}')) \\
\widehat{\text{OBSERVE}}(\hat{X}, \hat{v}, \hat{g}) = \; & \textit{let } \hat{g}' = \widehat{\text{HOIST}}(\hat{X}, \hat{g}) \textit{ in} \\
& \widehat{\text{INTERVENE}}(\hat{X}, \hat{\delta}(\hat{v}), \hat{g}')
\end{aligned}
$$

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 $\widehat{\textbf{fail}}$.

Grafting Algorithm The implementation of the interface using DS uses the abstract version of the grafting algorithm. However, because an abstract node can have more than one marginal child, $\widehat{\text{GRAFT}}$ has to invoke $\widehat{\text{PRUNE}}$ on each child.

$$\begin{array}{l}
\widehat{\text{GRAFT}}(\hat{X}, \hat{g}) = \mathit{match}\ \hat{g}(\hat{X})_{\text{n}}\ \mathit{with} \\
\quad \mid \widehat{\texttt{marginalized}}(\hat{S}_{rv}) \mid \widehat{\texttt{marginalized}}(\hat{X}', \hat{D}, \hat{S}_{rv}) : \\
\quad\quad \mathit{let}\ \hat{S}_{rv,\text{children}} = \widehat{\text{MARGINALIZED_CHILDREN}}(\hat{S}_{rv})\ \mathit{in} \\
\quad\quad \mathit{let}\ \hat{g}' = \hat{g}\ \mathit{in} \\
\quad\quad \mathit{for}\ \hat{X}_{\text{child}} \in \hat{S}_{rv,\text{children}} : \\
\quad\quad\quad \mathit{let}\ \hat{g}' = \widehat{\text{PRUNE}}(\hat{X}_{\text{child}}, \hat{g}')\ \mathit{in} \\
\quad\quad \hat{g}' \\
\quad \mid \widehat{\texttt{initialized}}(\hat{X}', \hat{S}_{rv}) : \\
\quad\quad \mathit{let}\ \hat{g}' = \widehat{\text{GRAFT}}(\hat{X}', \hat{g}_{)}\ \mathit{in}\ \widehat{\text{MARGINALIZE}}(\hat{X}, \hat{g}')
\end{array}$$

Interface Finally, as with SSI, the DS-implementation of the interface operations is the same as the concrete implementation, except the abstract operations and abstract values are used instead.

$$
\begin{aligned}
\widehat{\textsc{Assume}}(\hat{\kappa}, \hat{D}, \hat{g}) = \;& \textit{let } \hat{D}', \hat{g}' = \widehat{\textsc{conjugate\_dist}}(\hat{D}, \hat{g}) \textit{ in} \\
& (\hat{X}_{\text{new}}, \widehat{\textsc{initialize}}(\hat{X}_{\text{new}}, \hat{\kappa}, \hat{D}', \hat{g}')) \\
\widehat{\textsc{Value}}(\hat{X}, \hat{g}) = \;& \textit{if } \hat{g}(\hat{X})_{\text{a}} = \widehat{\texttt{symbolic}} \textit{ then } \widehat{\textbf{fail}} \\
& \textit{else let } \hat{g}' = \widehat{\textsc{graft}}(\hat{X}, \hat{g}) \textit{ in} \\
& (\texttt{UnkC}, \widehat{\textsc{realize}}(\hat{X}, \hat{\delta}_s(\texttt{UnkC}), \hat{g}')) \\
\textsc{observe}(\hat{X}, \hat{v}, \hat{g}) = \;& \textit{let } \hat{g}' = \widehat{\textsc{graft}}(\hat{X}, \hat{g}) \textit{ in} \\
& \widehat{\textsc{realize}}(\hat{X}, \hat{\delta}(\texttt{UnkC}), \hat{g}')
\end{aligned}
$$

4.4 Properties

In this section, I 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. I will highlight these nonstandard elements throughout the formal development. First, I define the collecting semantics for sets of program states that serves as the basis of the 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, I define the abstraction and concretization functions that relate abstract values to concrete values. Finally, I 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 the operational semantics. The program states collected differ between the three types of evaluation rules. Even though the operational semantics uses weight values and performs resampling, the collecting semantics ignores these aspects. The analysis is interested in 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, weight values are not collected and resampling is a no-op.

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

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

Particle Set 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 in Section 3.2. The relation $(S_p \tilde{\downarrow\downarrow} S_D)$ means that the set of particles S_p evaluates to the set of distributions S_D, and the definition of $\tilde{\downarrow\downarrow}$ refers to the definition of $\tilde{\downarrow}$. The given set of particles is first evaluated using the collecting semantics of the particle evaluation (which operates on a set of particles) to produce a set of configurations. If any of the particles return **fail**, the semantics return a singleton set containing **fail**. If all of the particles have a *false* resample flag (i.e. they all terminated), the semantics return a set of distributions using the DISTRIBUTION operation. If at least one of the particles has a *true* resample flag, the collecting semantics iterates on the particle set evaluation rule.

$$\frac{S_c = \left\{(e', g', r') \mid (e, g) \in S_p, (e, g \tilde{\downarrow} S'_c), (e', g', r') \in S'_c\right\} \quad \bigwedge_{(e,g,r)\in S_c} \neg r \quad \bigwedge_{(e,g,r)\in S_c} (e \neq \textbf{fail}) \quad S_D = \{\textsc{Distribution}(v, g) \mid (v, g, r) \in S_c\}}{S_p \tilde{\downarrow\downarrow} S_D}$$

$$\frac{S_c = \left\{(e', g', r') \mid (e, g) \in S_p, (e, g \tilde{\downarrow} S'_c), (e', g', r') \in S'_c\right\} \quad \bigvee_{(e,g,r)\in S_c} r \quad \bigwedge_{(e,g,r)\in S_c} (e \neq \textbf{fail}) \quad S'_p = \textsc{forgetr}(S_c) \quad S'_p \tilde{\downarrow\downarrow} S_D}{S_p \tilde{\downarrow\downarrow} S_D}$$

$$\frac{S_c = \left\{(e', g', r') \mid (e, g) \in S_p, (e, g \tilde{\downarrow} S'_c), (e', g', r') \in S'_c\right\} \quad \bigvee_{(e,g,r)\in S_c} (e = \textbf{fail})}{S_p \tilde{\downarrow\downarrow} \{\textbf{fail}\}}$$

Model Evaluation 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 $\tilde{\downarrow\downarrow}$. The collecting semantics of the model evaluation rules constructs a particle from the given program with an empty symbolic state and evaluates with the particle set evaluation rules.

$$\frac{\{e, \emptyset\} \tilde{\downarrow\downarrow} S_D}{e \tilde{\Downarrow} S_D}$$

4.4.2 Abstraction

The abstraction function α maps sets of concrete values to an abstract value. I 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 variable names 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 4.4.1 (Abstraction Function)**.** The abstraction function α uses the default mapping function $\widehat{RV}_{canon}$ maps every concrete variable to a unique, canonical abstract variable as follows:

$$
\begin{array}{rcl@{\qquad}rcl}
\alpha(\{c\}) & = & \hat{c} & \alpha(\{X\}) & = & \widehat{RV}_{canon}(X) \\
\alpha(\{E_1 \texttt{ + } E_2\}) & = & \alpha(\{E_1\}) \mathbin{\hat{+}} \alpha(\{E_2\}) & \alpha(\{(v_1, v_2)\}) & = & (\alpha(\{v_1\}), \alpha(\{v_2\})) \\
\alpha(\{\mathcal{N}(E_1, E_2)\}) & = & \widehat{\mathcal{N}}(\alpha(\{E_1\}), \alpha(\{E_2\})) & \alpha(\{\texttt{symbolic}\}) & = & \widehat{\texttt{symbolic}} \\
\multicolumn{6}{c}{\alpha(\{\texttt{marginalized}(X, D, S_{rv})\}) = \widehat{\texttt{marginalized}}(\widehat{RV}_{canon}(X), \alpha(\{D\}), \{\widehat{RV}_{canon}(X_s) \mid X_s \in S_{rv}\})} \\
\multicolumn{6}{c}{\cdots} \\
\alpha(\{g\}) & = & \left\{ \widehat{RV}_{canon}(X) \mapsto \alpha(\{g(X)\}) \;\middle|\; X \in \mathrm{dom}(g) \right\} & & & \\
\alpha(\{\textbf{fail}\}) & = & \widehat{\textbf{fail}} & \alpha(S_v) & = & \bigsqcup_{v \in S_v} \alpha(\{v\})
\end{array}
$$

For example, consider a particle (E, g) with symbolic expression $E = X_1 \texttt{ + } 1$ 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 I formalize abstraction with a default mapping function, the concretization needs to account for all possible mappings. I 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 $\hat{S}_{rv}$. I formalize this using the operation $FV(E, \widehat{RV})$ that returns the set of free variables in E, mapped to abstract names using $\widehat{RV}$.

The concretization of an abstract symbolic state is defined by a set of properties that the concrete symbolic states must satisfy. If a concrete random variable maps to an abstract random variable that is in the abstract state, the concrete random variable must be assigned to an entry in the concrete state. Additionally, the entry must be in the concrete set of the abstract entry of that abstract variable. The concrete state must also have no additional random variables that do not map to an abstract variable in the abstract state. If a concrete state exhibit these properties with respect to the given abstract symbolic state, then it is in the concrete set of the abstract state. Finally, I 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 4.4.2 (Concretization Function)**.** First, γ is defined 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.

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

$$
\gamma(\widehat{\texttt{marginalized}}(\hat{X}, \hat{D}, \hat{S}_{rv}), \widehat{RV}) = \left\{ \texttt{marginalized}(X, D, S_{rv}) \mid \begin{smallmatrix} \widehat{RV}(X) = \hat{X}, D \in \gamma(\hat{D}, \widehat{RV}), \\ S \subseteq \{X_s \mid \widehat{RV}(X_s) \in \hat{S}_{rv}\} \end{smallmatrix} \right\}
$$

$$
\cdots
$$

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

Now, I 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 $\gamma(\{\hat{X}_b \mapsto (\widehat{\texttt{sample}}, \texttt{UnkD}(\emptyset))\}, \widehat{RV})$. However, the concrete symbolic state $\{X_1 \mapsto (\texttt{sample}, \delta_s(1))\}$ is not. Additionally, the abstract distribution $\hat{D} = \widehat{\mathcal{N}}(\hat{X}_a, \hat{X}_b \mathbin{\hat{+}} \hat{1})$ has a concretization $\gamma(\hat{D})$, such that $\mathcal{N}(X_0, X_1 + 1) \in \gamma(\hat{D})$ and $\mathcal{N}(X_1, X_0 + 1) \in \gamma(\hat{D})$.

4.4.4 Soundness of Analysis

I present the formalization and proof of soundness for the analysis. The proofs rely on the soundness of abstract hybrid inference interface implementations. Implementations of the abstract interface must be sound with respect to those of the concrete interface:

Assumption 4.4.1 (Abstract Hybrid Inference Interface Soundness)**.** For every $i \in \{\textsc{Assume}, \textsc{Value}, \textsc{Observe}\}$ and input values v_i, it holds 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, I define a weak equivalence relation for concrete symbolic states, analogous to the weak equivalence relation for abstract states. The weak equivalence comparison between two symbolic states only compares random variables that are reachable from the given expression.

$$
g_1 \cong_e g_2 \iff \forall_{X \in \textsc{reachable}(e,\, g_1,\, g_2)}\; g_1(X) = g_2(X)
$$

From the weak equivalence relations, I prove the weakening properties of concrete and abstract symbolic states. Informally, this property states that adding random variables that are not already in a symbolic state does not alter the evaluated expression nor the original entries in the resulting symbolic state.

Lemma 4.4.1 (Weakening). *If* $(e, g \downarrow^r e', g', w)$ *and* $g \cong_e g_1$*, then* $(e, g_1 \downarrow^r e', g_1', w)$ *and* $g' \cong_{e'} g_1'$.

Proof. By structural induction on derivations of $\downarrow$. □

Lemma 4.4.2 (Abstract Weakening). *If* $(e, \hat{g} \hat{\downarrow} \hat{v}', \hat{g}')$ *and* $\hat{g} \cong_e \hat{g}_1$*, then* $(e, \hat{g}_1 \hat{\downarrow} \hat{v}', \hat{g}_1')$ *and* $\hat{g}' \cong_{\hat{v}'} \hat{g}_1'$.

Proof. By structural induction on derivations of $\hat{\downarrow}$. □

Next, I prove two helper lemmas for the soundness of the RENAME and RENAME_JOIN operations used when over-approximating control flow and loops during particle evaluation. First, $\text{RENAME}(\hat{E}_1, \hat{E}_2, \hat{g})$ preserves the concretization of the pair $(\hat{E}_2, \hat{g})$.

Lemma 4.4.3 (Renaming Soundness). *For any abstract expressions* $\hat{E}_1$ *and* $\hat{E}_2$*, and any abstract symbolic state* $\hat{g}$*, if* $(\hat{E}_2', \hat{g}') = \textit{RENAME}(\hat{E}_1, \hat{E}_2, \hat{g})$ *then* $\gamma((\hat{E}_2', \hat{g}')) = \gamma((\hat{E}_2, \hat{g}))$.

Proof. The key idea is that, for any $\widehat{RV}$ used in the computation of $\gamma((\hat{E}_2, \hat{g}))$, I can construct a $\widehat{RV}'$ that incorporates the renaming of $\hat{X}$ to $\hat{X}'$. Suppose that in $\widehat{RV}$, the concrete random variable X is mapped to $\hat{X}$. Let $\widehat{RV}'$ be a copy of $\widehat{RV}$, except that X maps to $\hat{X}'$ in $\widehat{RV}'$, and a fresh concrete random variable X_{new} maps to $\hat{X}$. Then, the computation of the union in $\gamma((\hat{E}_2', \hat{g}'))$ is simply a reordering of the one from $\gamma((\hat{E}_2, \hat{g}))$.

□

Next, the RENAME_JOIN operation soundly over-approximates its input, given that additional unused random variables may be present in the over-approximation.

Lemma 4.4.4 (Rename-Join Expression Soundness). *For any two abstract expressions* $\hat{E}_1$ *and* $\hat{E}_2$*, abstract symbolic states* $\hat{g}_1$ *and* $\hat{g}_2$*, there exists some abstract symbolic states* $\hat{g}_1', \hat{g}_2'$ *such that* $\hat{g}_1 \cong_{\hat{E}_1} \hat{g}_1'$*,* $\hat{g}_2 \cong_{\hat{E}_2} \hat{g}_2'$*, and* $(\gamma((\hat{E}_1, \hat{g}_1')) \cup \gamma((\hat{E}_2, \hat{g}_2'))) \subseteq \gamma(\textit{RENAME\_JOIN}(\hat{E}_1, \hat{E}_2, \hat{g}_1, \hat{g}_2))$.

Proof. Let $\hat{g}_1'$ and $\hat{g}_2'$ be the following abstract symbolic states:

$$\hat{g}_1' = \left\{ \hat{X} \mapsto \begin{cases} \hat{g}_1(\hat{X}) & \hat{X} \in \text{dom}(\hat{g}_1) \\ \hat{g}_2(\hat{X}) & \hat{X} \in \text{dom}(\hat{g}_2) \setminus \text{dom}(\hat{g}_1) \end{cases} \right\}$$

$$\hat{g}_2' = \left\{ \hat{X} \mapsto \begin{cases} \hat{g}_2(\hat{X}) & \hat{X} \in \text{dom}(\hat{g}_2) \\ \hat{g}_1(\hat{X}) & \hat{X} \in \text{dom}(\hat{g}_1) \setminus \text{dom}(\hat{g}_2) \end{cases} \right\}$$

By construction, $\hat{g}_1 \cong_{\hat{E}_1} \hat{g}_1'$ and $\hat{g}_2 \cong_{\hat{E}_2} \hat{g}_2'$. During rename-join, we have an intermediate, renamed abstract expression and abstract symbolic state, $\hat{E}_3, \hat{g}_3 = \text{RENAME}(\hat{E}_1, \hat{E}_2, \hat{g}_2)$. The operation may rename a random variable $X_2 \in \text{dom}(\hat{g}_2)$ to a random variable $\hat{X}_1 \in \text{dom}(\hat{g}_1)$ and $\hat{X}_1$ may or may not be in $\text{dom}(\hat{g}_2)$. However, the operation never renames a random variable $\hat{X}_1 \notin \text{dom}(\hat{g}_2)$ to a variable $\hat{X}_2 \in \text{dom}(\hat{g}_2)$. So the random variables in $\text{dom}(\hat{g}_1) \setminus \text{dom}(\hat{g}_2)$ either becomes a part of $\text{dom}(\hat{g}_3)$ or remains unreachable from $\hat{E}_3$. Thus, we have $\hat{E}_3, \hat{g}_3' = \text{RENAME}(\hat{E}_1, \hat{E}_2, \hat{g}_2')$ where

$$\hat{g}_3' = \left\{ \hat{X} \mapsto \begin{cases} \hat{g}_3(\hat{X}) & \hat{X} \in \text{dom}(\hat{g}_3) \\ \hat{g}_1(\hat{X}) & \hat{X} \in \text{dom}(\hat{g}_1) \setminus \text{dom}(\hat{g}_3) \end{cases} \right\}$$

It follows from the definition of basic expression join, abstract symbolic state join, and concretization that $(\gamma((\hat{E}_1, \hat{g}_1')) \cup \gamma((\hat{E}_3, \hat{g}_3'))) \subseteq \gamma((\hat{E}_1 \sqcup \hat{E}_3, \hat{g}_1 \sqcup \hat{g}_3))$. Following from Lemma 4.4.3, the concrete set of the intermediate, renamed abstract expression and abstract symbolic state preserves the concrete set of the original: $\gamma((\hat{E}_3, \hat{g}_3')) = \gamma((\hat{E}_2, \hat{g}_2'))$. Therefore, we have $(\gamma((\hat{E}_1, \hat{g}_1')) \cup \gamma((\hat{E}_2, \hat{g}_2'))) \subseteq \gamma(\text{RENAME\_JOIN}(\hat{E}_1, \hat{E}_2, \hat{g}_1, \hat{g}_2))$. □

I next prove a lemma stating that substituting abstract values that over-approximates the original abstract value, soundly over-approximates the original results.

Lemma 4.4.5 (Substitution Soundness)**.** *If* $(e, \hat{g}_1 \hat{\downarrow} \hat{v}_1', \hat{g}_1')$ *and* $(e[\hat{v}_1 \leftarrow \hat{v}_2], \hat{g}_2 \hat{\downarrow} \hat{v}_2', \hat{g}_2')$ *and* $\gamma((\hat{v}_1, \hat{g}_1)) \subseteq \gamma((\hat{v}_2, \hat{g}_2))$*, then* $\gamma((\hat{v}_1', \hat{g}_1')) \subseteq \gamma((\hat{v}_2', \hat{g}_2'))$.

Proof. By structural induction on derivations of $\hat{\downarrow}$, with appeal to Theorems 4.4.2 and 4.4.4.

□

Now, using the above helper lemmas, I show that the analysis is sound when the particle evaluation terminates. As shorthand, I write configuration sets that have weakly equivalent symbolic states as $S_c \cong S_c' \iff S_c' = \{(e, g', r) \mid (e, g, r) \in S_c, g \cong_e g'\}$. I also use an auxiliary operation to drop the resample flag in configurations: $\text{FORGETR}(S_c) = \{(e, g) \mid (e, g, r) \in S_c\}$.

Lemma 4.4.6 (Terminating Particle Evaluation Soundness)**.** *For every particle* (e, g)*, such that* $(e, g \tilde{\downarrow} S_c)$ *and* $\forall_{(v,\, g',\, r) \in S_c}\ (v = \textbf{\textit{fail}}) \vee \neg r$*, it holds that* $(e, \alpha(\{g\}) \hat{\downarrow} \hat{v}', \hat{g}')$ *and there exists a configuration set* S_c' *such that* $S_c \cong S_c'$ *and* $\text{FORGETR}(S_c') \subseteq \gamma((\hat{v}', \hat{g}'))$.

Proof. By structural induction on derivations of $\downarrow$, by definition of abstraction, concretization, and interpretation rules, with appeal to Assumption 4.4.1 and Theorems 4.4.1, 4.4.2, 4.4.4 and 4.4.5.

□

However, not all particle evaluations terminate; particle evaluations can be interrupted from `resample` operators. I prove that resuming evaluation from a configuration preserves soundness.

Lemma 4.4.7 (Preservation)**.** *If* $(e, g \downarrow^r e', g', w)$*, then* $(e, \alpha(\{g\}) \hat{\downarrow} \hat{v}, \hat{g}) \iff (e', \alpha(\{g'\}) \hat{\downarrow} \hat{v}', \hat{g}')$ *and there exists some abstract symbolic state* $\hat{g}''$ *such that* $\hat{g}' \cong_{\hat{v}'} \hat{g}''$ *and* $\gamma((\hat{v}', \hat{g}'')) \subseteq \gamma((\hat{v}, \hat{g}))$.

Proof. By structural induction on derivations of $\downarrow$, by definition of abstraction, concretization, and interpretation rules, with appeal to Theorems 4.4.2 and 4.4.4 to 4.4.6

□

Then, from the previous properties, repeatedly evaluating a particle until it has terminated is sound. I formalize this notion with an auxiliary operation $\tilde{\downarrow}^*$ for repeatedly evaluating a particle until termination.

Definition 4.4.3 (Particle Evaluation until Termination)**.** The operation $\tilde{\downarrow}^*$ evaluates an expression on a set of symbolic states until completion.

$$\frac{e, g \tilde{\downarrow} S_c \qquad \bigwedge_{(e', g', r') \in S_c} \neg r' \qquad \bigwedge_{(e', g', r') \in S_c} (e' \neq \textbf{fail})}{e, g \tilde{\downarrow}^* S_c} \qquad \frac{E, g \tilde{\downarrow} S_c \qquad \bigvee_{(e', g', r') \in S_c} e' = \textbf{fail}}{E, g \tilde{\downarrow}^* \{(\textbf{fail}, \textbf{fail}, \textbf{fail})\}}$$

$$\frac{\begin{array}{c} e, g \tilde{\downarrow} S_c \qquad \bigvee_{(e', g', r') \in S_c} r' \\ \bigwedge_{(e', g', r') \in S_c} (e' \neq \textbf{fail}) \qquad S_c'' = \left\{(e'', g'', r'') \mid (e', g', r') \in S_c, (e', g' \tilde{\downarrow}^* S_c'), (e'', g'', r'') \in S_c'\right\} \end{array}}{e, g \tilde{\downarrow}^* S_c''}$$

Lemma 4.4.8 (Particle Evaluation Soundness)**.** *For every particle* (e, g)*, such that* $(e, g \mathrel{\tilde{\downarrow}^*} S_c)$*, it holds that* $(e, \alpha(\{g\}) \mathrel{\hat{\downarrow}} \hat{v}, \hat{g})$ *and a configuration set* S_c' *such that* $S_c \cong S_c'$ *and* $\textsc{forgetr}(S_c') \subseteq \gamma((\hat{v}, \hat{g}))$*.*

Proof. By structural induction on derivations of $\tilde{\downarrow}^*$, with appeal to Theorems 4.4.6 and 4.4.7. □

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 4.4.9 (Particle Trace)**.** *If* $(S_p \mathrel{\tilde{\downdownarrows}} S_D)$*, then for all* $D \in S_D$*, there exists* $(e, g) \in S_p$ *such that* $(e, g \mathrel{\tilde{\downarrow}^*} S_c)$ *and* $D \in \{\textsc{Distribution}(v, g) \mid (v, g, r) \in S_c\}$*.*

Proof. By structural induction derivations of $\tilde{\downdownarrows}$. □

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

Theorem 4.4.10 (Particle Set Evaluation Soundness)**.** *For every particle set* S_p*, and distribution set* S_D *such that* $(S_p \mathrel{\tilde{\downdownarrows}} S_D)$*, it holds that* $\{e, \alpha(\{g\}) \mid (e, g) \in S_p\} \mathrel{\hat{\downdownarrows}} \hat{v}$ *and* $S_D \subseteq \gamma(\hat{v})$*.*

Proof. By Theorems 4.4.8 and 4.4.9 and the definition of abstraction and concretization. □

Corollary 4.4.11 (Model Evaluation Soundness)**.** *If* $e \mathrel{\tilde{\Downarrow}} \{\textbf{\textit{fail}}\}$*, then* $e \mathrel{\hat{\Downarrow}} \widehat{\textbf{\textit{fail}}}$*.*

Chapter 5

Evaluation

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

RQ1. Can inference plans improve hybrid inference performance?

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.

5.1 Benchmarks

I evaluate the performance of different hybrid inference algorithms and the analysis on a set of benchmark programs, each of which requires sampling because it cannot be solved purely with exact inference. I describe here the benchmarks and identify the variables evaluated for accuracy.

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 models the glint noise that may be present in radar target tracking applications (Wu, 1993) and is the example in Chapter 2. It is similar to *Noise* but 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 distribution – says there is glint 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).

Outlier models a one-dimensional particle filter with potential sensor errors producing outlier observations. The hidden state is modeled by Gaussian distributions (`xt`), the sensor error rate as a Beta prior (`outlier_prob`) to a Bernoulli, and observations are made on Gaussian distributions. This program was implemented by Atkinson et al. (2022) and adapted from Minka (2001).

OutlierHeavy is the same program as *Outlier* but the outlier observations are modeled by a long-tailed location-scale t distribution as used in Chang (2014).

Tree is a particle filter that makes an observation centered around a single random variable modeled by a Gaussian distribution (`a`) and another centered around random variables (`b`) that are

sampled from a Gaussian distribution every timestep. This program was previously implemented by Atkinson et al. (2022) and adapted from Lundén (2017).

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 uses 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.

5.2 Methodology

I implemented SIREN in Python with semi-symbolic inference and delayed sampling. I 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.

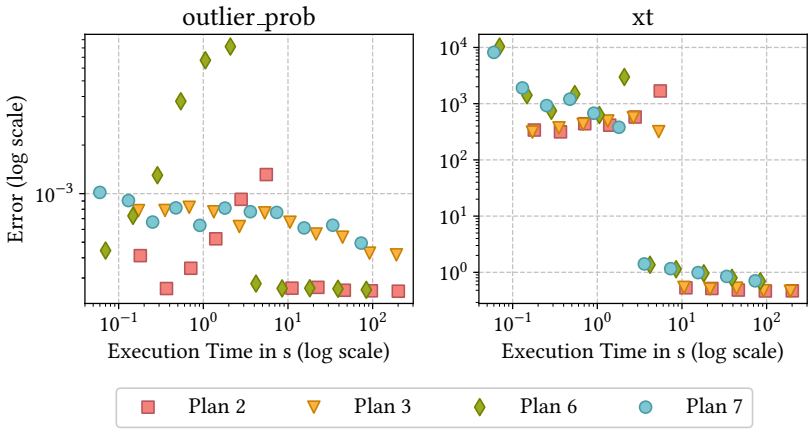
RQ1 Methodology To determine SIREN's performance properties for *RQ1*, I execute each benchmark 100 times for 100 timesteps with an exponentially increasing number of particles from 1 to 1024. I execute each benchmark using all satisfiable inference plans, except for *SLDS* with SSI and DS, where due to the large number of inference plans, I 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. I measure the accuracy by the Mean Squared Error (MSE) of the inferred expected value of the variable compared to its ground truth value, which is available as all data were generated by sampling from the prior. I conduct experiments on a 60-core Intel Xeon Cascade Lake (up to 3.9 GHz) node with 240 GB RAM.

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

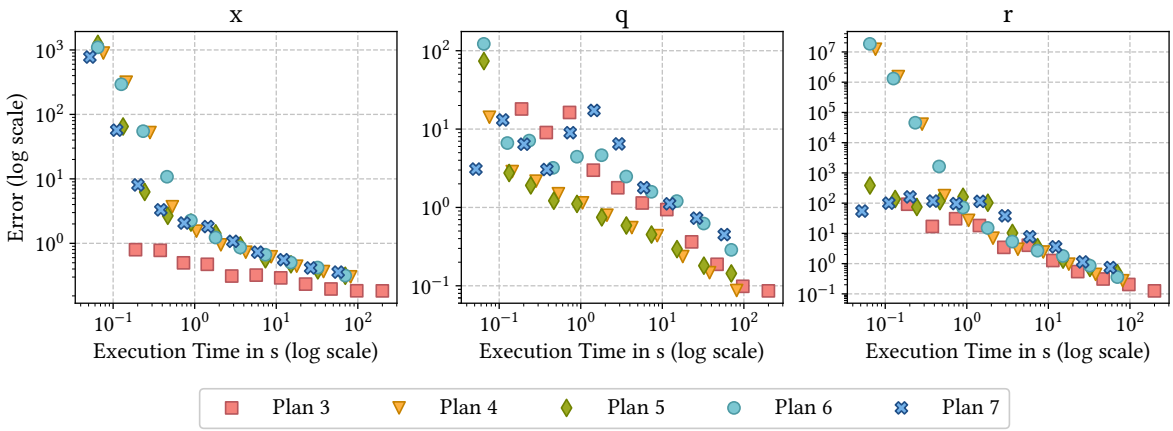
5.3 Results

RQ1 Results Figure 5.1 plots the experiment results of *Outlier* and *Noise* using the SSI algorithm. For each particle count, I plot the median runtime to the 90th percentile of error for each variable of interest. Plots for the remaining algorithms and benchmarks are in Appendix A.

The *Outlier* results show that, when the execution time is greater than 10 seconds, the annotated Plan 2 achieves the best accuracy for both `xt` and `outlier_prob` by a factor of 2.1x than the default Plan 6. However, if the runtime cannot exceed 10 seconds, Plan 6 achieves the best accuracy, and the alternative Plan 7 may also be acceptable, particularly in a context where the developer cares most about `xt`. The optimal inference plan depends on the context in which the system is deployed.



(a) *Outlier*. Unannotated program executes Plan 6.



(b) *Noise*. Unannotated program executes Plan 5.

Figure 5.1: For each particle count, I plot the median execution time to the 90th percentile of error for each variable using different satisfiable inference plan.

The results for *Noise* show that it achieves the lowest error for `x` with the annotated Plan 3 by a factor of 7.9x over the default Plan 5, but the alternative Plan 4 and the default Plan 5 achieve the lowest error for `q`. Thus, the optimal plan depends on the context – in particular which variable the developer considers most important.

Appendix A shows similar results for other algorithms. For example, executing *Runner* using SMC with BP with an annotated plan achieves a speedup of 1166x over the default plan.

Overall, these results demonstrate that inference performance depends strongly on the inference plan, that the optimal inference plan is context-dependent, and that annotated inference plans enable substantial speedups over the default.

RQ2 Results To evaluate the analysis precision for *RQ2*, we run the analysis on all benchmarks and inference algorithms, and summarize the results in Table 5.1. I manually count the total number of satisfiable plans in each case and how many plans the analysis identifies. The analysis identifies all the satisfiable plans in every case except for *SLDS* with SSI, where the analysis only identifies 28 out of 36 satisfiable plans. This shows that the analysis is precise in practice.

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; inconsistent branch conditions are not detected. Consider this 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, 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.

Table 5.1: 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*	*EnvNoise*	*Outlier*	*OutlierHeavy*	*Tree*	*SLDS*	*Runner*
SSI	5/5	3/3	3/3	4/4	2/2	4/4	**28/36**	4/4
DS	4/4	2/2	2/2	2/2	2/2	3/3	16/16	1/1
SMC w/ BP	2/2	2/2	2/2	2/2	1/1	4/4	4/4	4/4
Total Possible Plans	8	32	32	8	8	4	128	16

Chapter 6

Related Work

Static Hybrid Inference Hakaru (Narayanan et al., 2016), Autoconj (Hoffman et al., 2018), and automatic marginalization (Lai et al., 2023) perform static transformation to solve the model analytically, and allowing the rest to be solved with some approximate inference technique. In such systems, the partition of random variables that must be sampled and those that do not is inherently known at compile time. However, the concept of inference plans and distribution encodings can still provide an explicit interface for reasoning about these partitions.

Programmable Inference Autoconj (Hoffman et al., 2018) provides an interface for users to specify where the system should perform marginalization to implement inference, while automatically detecting exact inference opportunities. My approach only requires the generative model and the annotations from the programmer, while the system automatically figures out how to perform inference. Works in the programmable inference space (Cusumano-Towner et al., 2019, Lew et al., 2019, Mansinghka et al., 2014, 2018, Tehrani et al., 2020) also hand over control of the inference procedure to the user at varying stages of inference. My interface applies specifically to hybrid inference algorithms and enables users to provide high-level guidance to the inference system without requiring a deep understanding of the algorithm.

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., 2017, 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, Monniaux, 2000, 2001, Ngo et al., 2018, Smith, 2008, Trilla et al., 2020, Wang and Reps, 2024). My analysis infers properties about the runtime behavior of the probabilistic inference algorithm.

Chapter 7

Conclusion

Probabilistic programming languages take advantage of the expressivity and accessibility of programming abstractions to enable developers to easily create systems for probabilistic modeling and inference. These abstractions automate the intricate details of inference algorithms, so the languages do not require in-depth knowledge about the algorithms to use. Amongst general-purpose programming languages, different levels of memory abstractions present a tradeoff between ease of use and control: Python fulfills the need for high-level programming with little memory management required, whereas C fulfills the the need for highly optimized code at the expense of user-directed memory management; Java lies in the middle of the spectrum, balancing control and ease of use. Comparitively, the levels of abstraction in PPLs present a tradeoff between the required level of expertise for the inference algorithm of the system and the amount of control the developer has over the system. The more automation and abstraction, the less the developer is required to know about the inference algorithm and the less control the developer has over how the inference algorithm implements the system, and vice versa. Different problem constraints and requirements call for different points on the spectrum. For the PPL space to be mature, a suitable tool should fulfill each point on the spectrum.

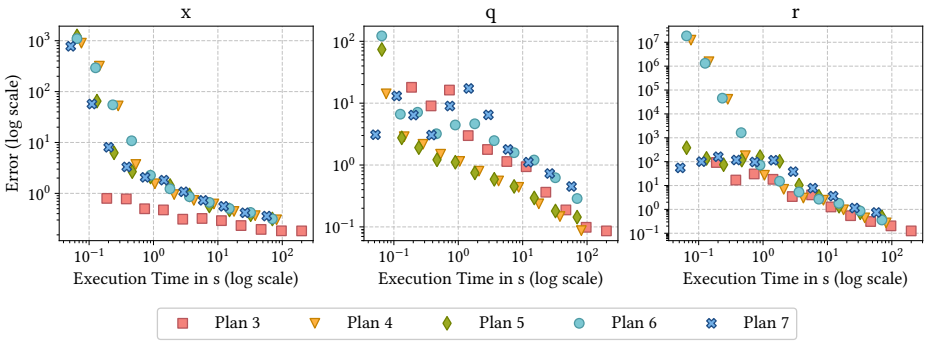
One type of abstractions in PPLs is the method of implementing hybrid inference, which incorporates symbolic exact inference with Monte Carlo sampling to improve inference performance. Works like semi-symbolic inference fully automate hybrid inference and require little inference expertise, whereas works like Autoconj require an understanding of marginalization and inference but allow for greater control of how inference is applied. This thesis provides developers with control over the partitioning of random variables into sampled and symbolic variables in hybrid inference systems. It presents a new PPL, Siren, that enables developers to use annotations to select an inference plan that specifies a particular partitioning. To assist programmers in reasoning about inference plans, Siren employs a static analysis that determines if an inference plan is satisfiable in all possible executions of the program. Through annotated inference plans and the satisfiability analysis, Siren enables developers to exercise fine-grained control over their inference algorithms without having to understand the intricate details of how they are implemented.

To the best of my knowledge, this thesis presents the first interface that balances control and ease of use for hybrid inference systems, such that users do not need to manually implement how exact inference is applied in the model but can still manipulate how it is applied. Other designs using programming constructs could provide developers with varying degrees of control and ease of use over hybrid inference. I hope this work serves as a basis for exploring other potential interfaces.

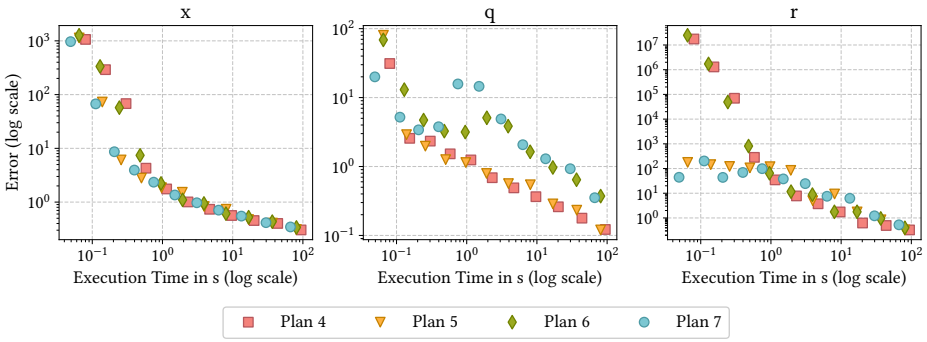
Appendix A

Additional Performance Evaluation

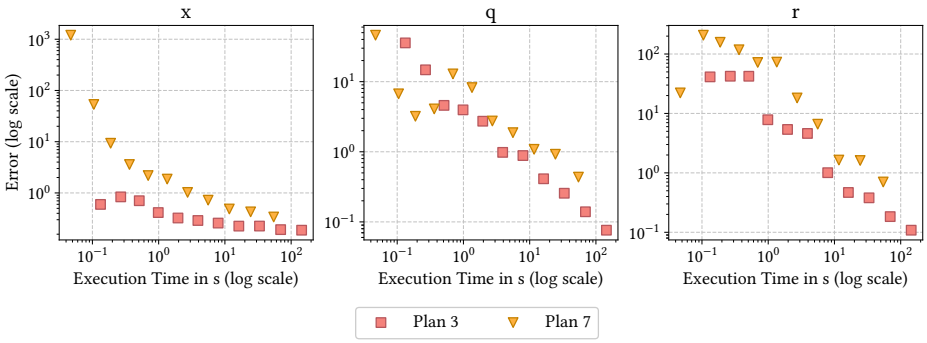
For each particle count, I plot the median execution time to the 90th percentile of error for each variable of interest using each satisfiable inference plan. I execute each benchmark using all satisfiable inference plans, except for *SLDS* with SSI and DS, where due to the large number of inference plans, I 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. The SMC with BP algorithm only has one inference plan on *OutlierHeavy* and likewise for DS on *Runner*. I omit the performance evaluation in those cases.



(a) SSI. Unannotated program executes Plan 5.

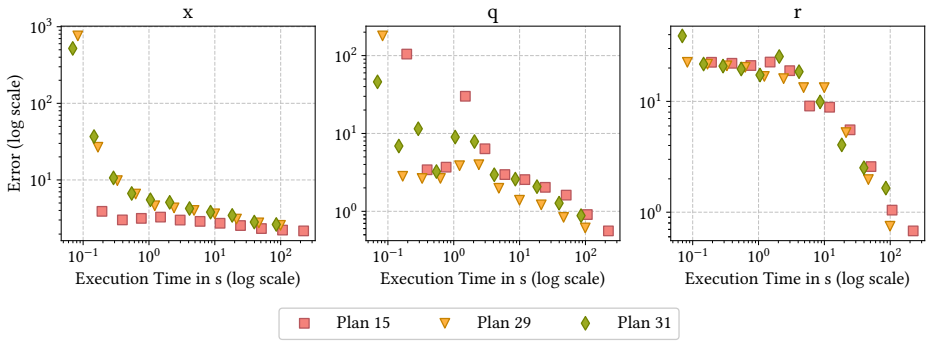


(b) DS. Unannotated program executes Plan 5.

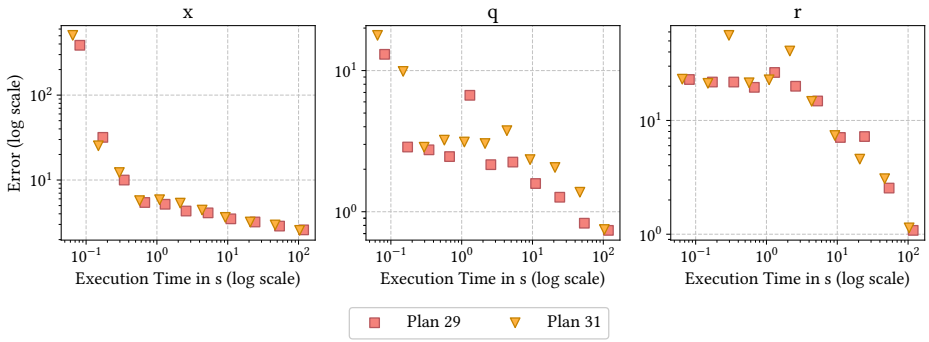


(c) SMC w/ BP. Unannotated program executes Plan 3.

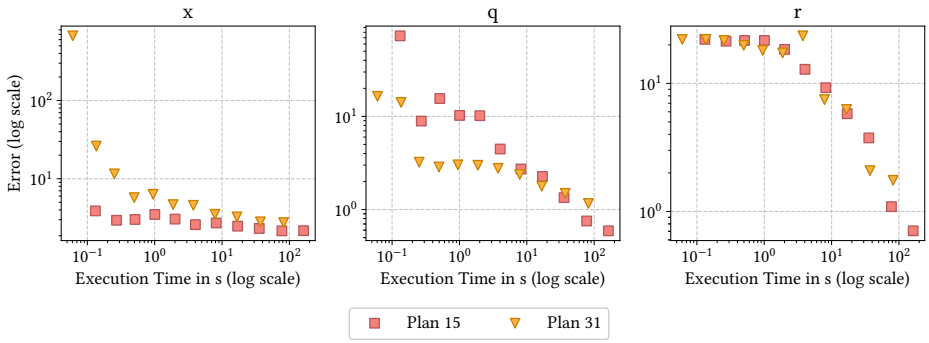
Figure A.1: *Noise*



(a) SSI. Unannotated program executes Plan 29.

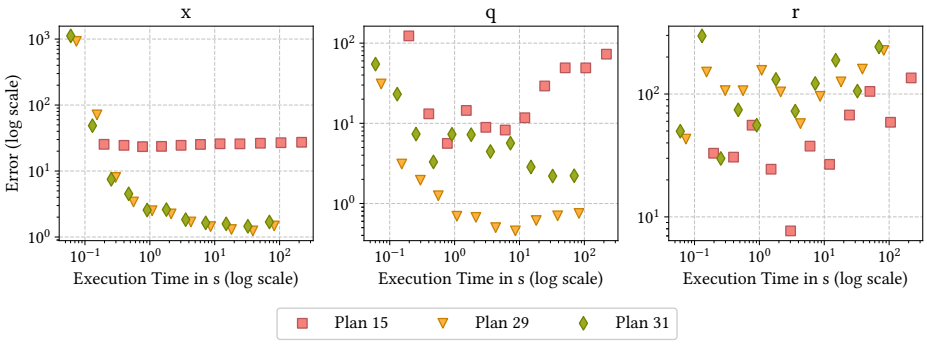


(b) DS. Unannotated program executes Plan 29.

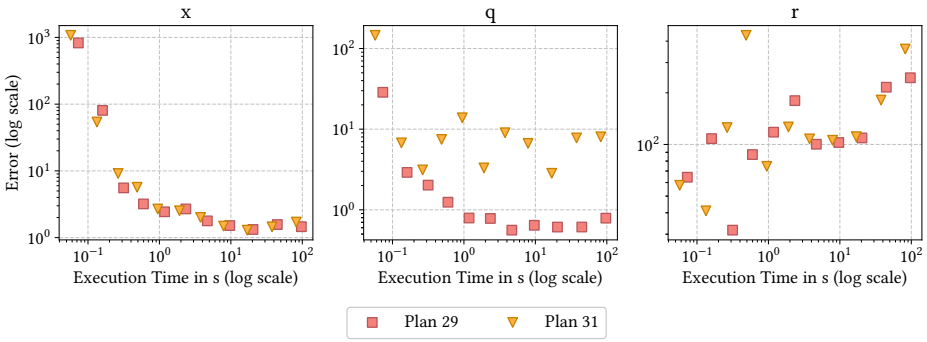


(c) SMC w/ BP. Unannotated program executes Plan 15.

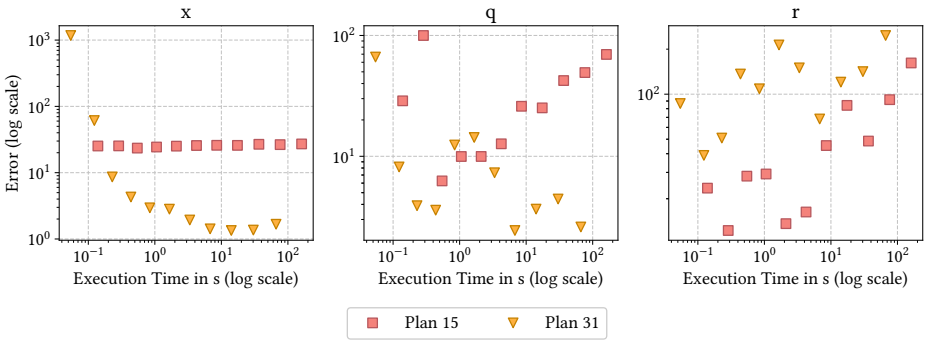
Figure A.2: *Radar*



(a) SSI. Unannotated program executes Plan 29.

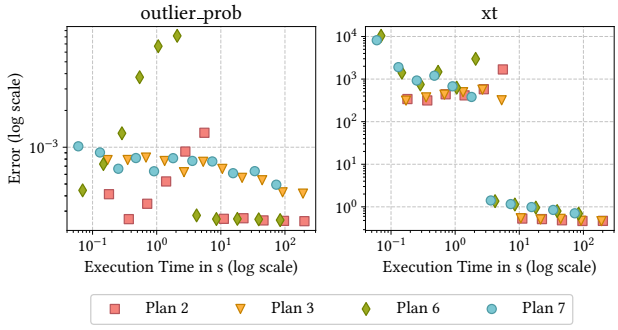


(b) DS. Unannotated program executes Plan 29.

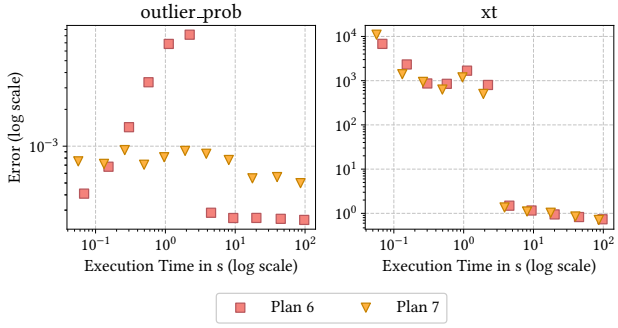


(c) SMC w/ BP. Unannotated program executes Plan 15.

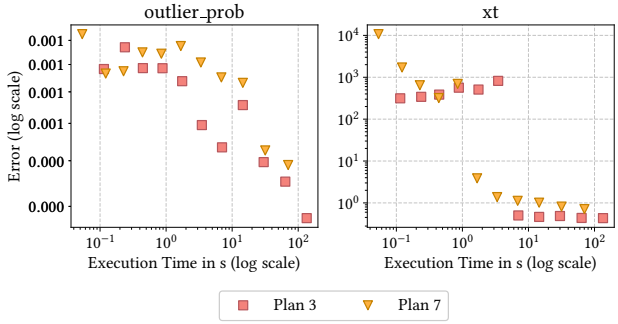
Figure A.3: *EnvNoise*



(a) SSI. Unannotated program executes Plan 6.

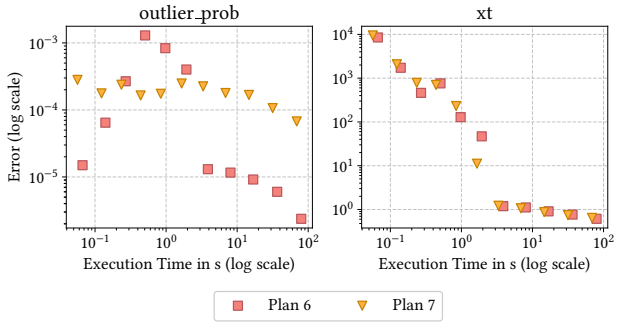


(b) DS. Unannotated program executes Plan 6.

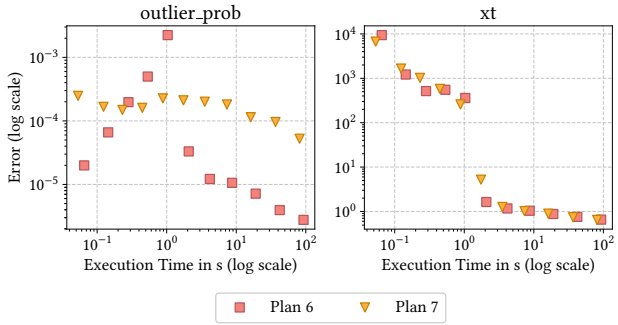


(c) SMC w/ BP. Unannotated program executes Plan 3.

Figure A.4: *Outlier*

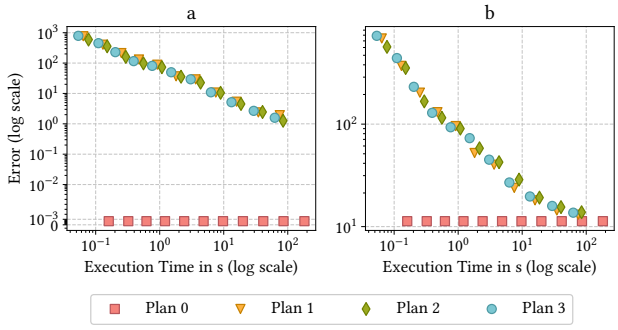


(a) SSI. Unannotated program executes Plan 6.

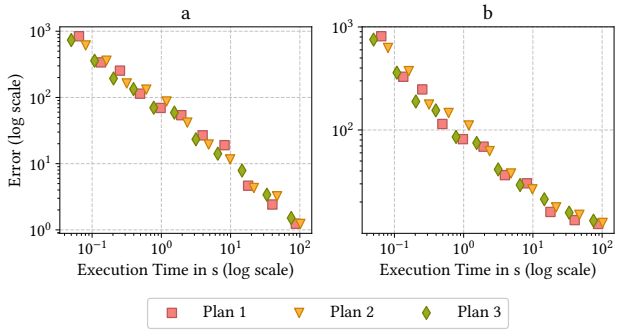


(b) DS. Unannotated program executes Plan 6.

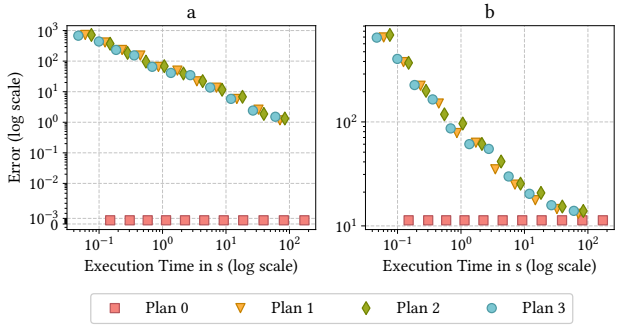
Figure A.5: *OutlierHeavy*. SMC w/ BP only has one satisfiable inference plan on this program.



(a) SSI. Unannotated program executes Plan 0.

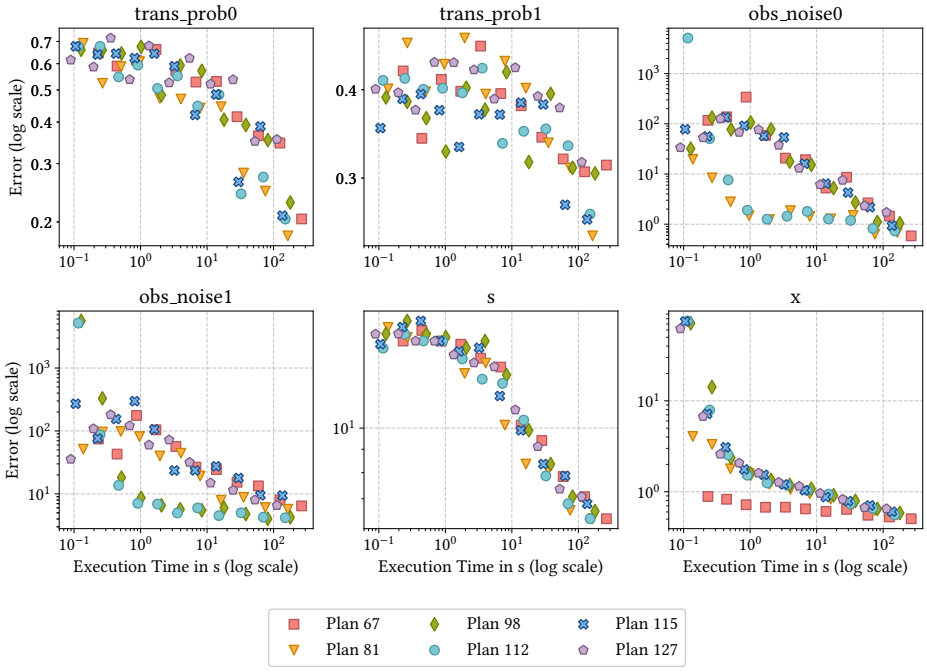


(b) DS. Unannotated program executes Plan 2.

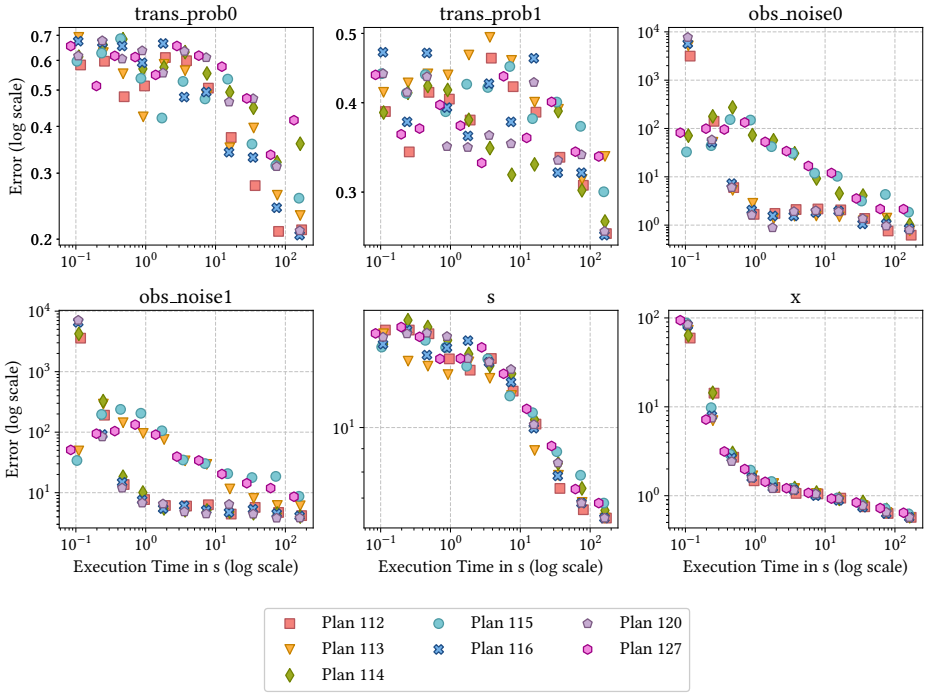


(c) SMC w/ BP. Unannotated program executes Plan 0.

Figure A.6: *Tree*

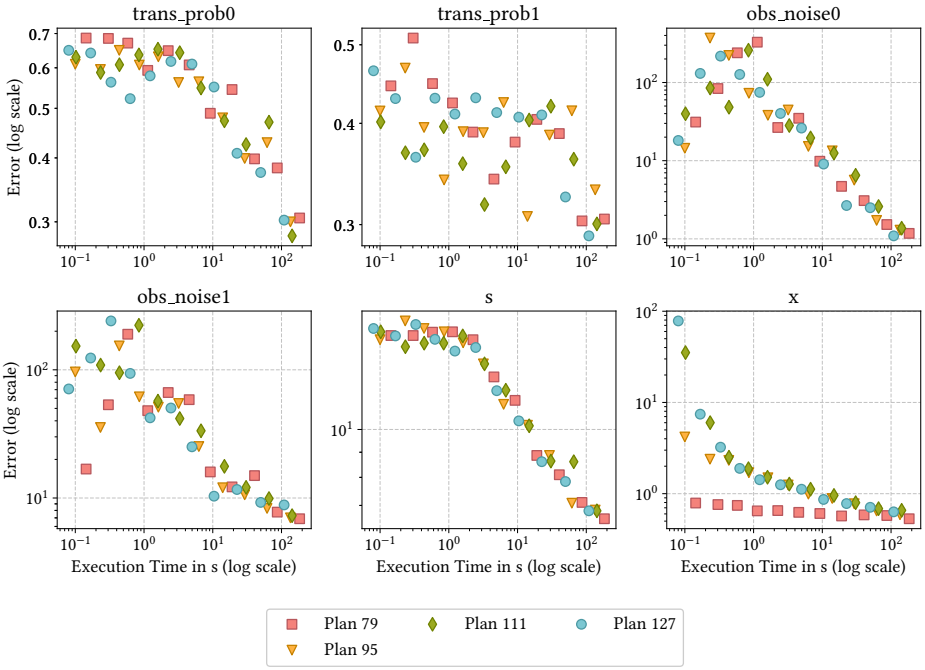


(a) SSI. Unannotated program executes Plan 115.



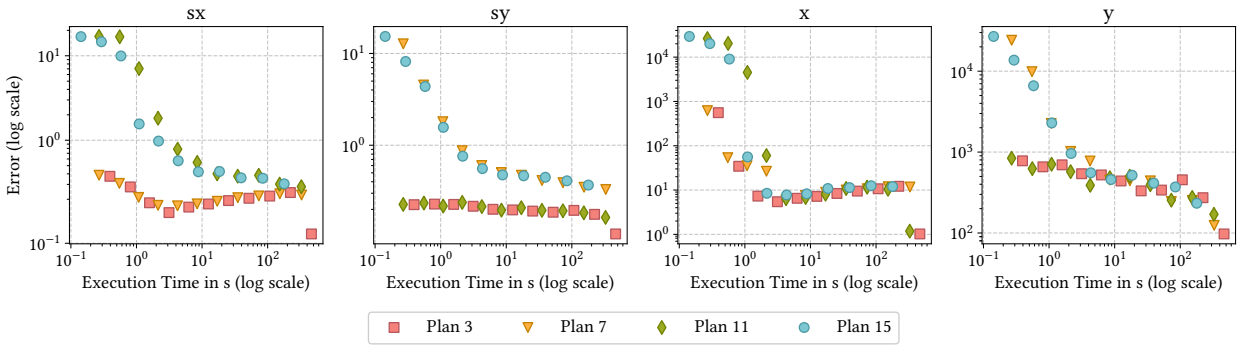
(b) DS. Unannotated program executes Plan 115.

Figure A.7: *SLDS*

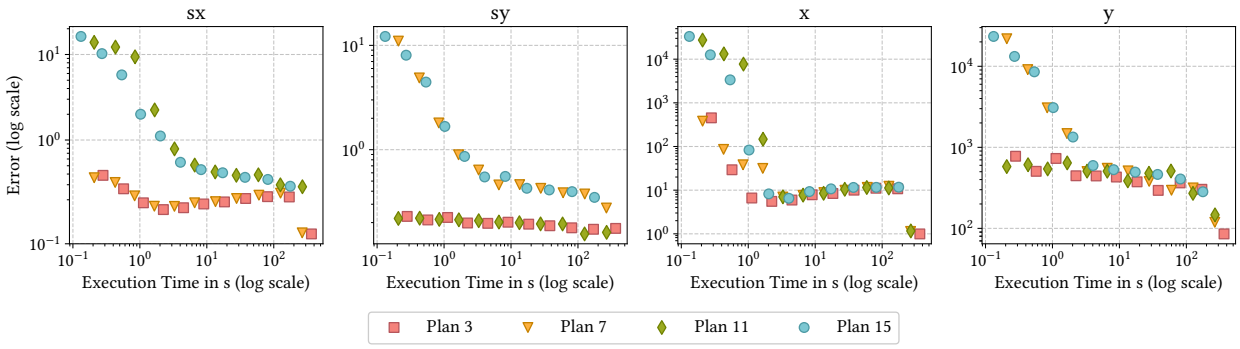


(a) SMC w/ BP. Unannotated program executes Plan 79.

Figure A.8: *SLDS* (continued)



(a) SSI. Unannotated program executes Plan 3.



(b) SMC w/ BP. Unannotated program executes Plan 11.

Figure A.9: *Runner*. DS only has one satisfiable inference plan on this program.

Bibliography

E. Arazo, D. Ortego, P. Albert, N. O'Connor, and K. McGuinness. Unsupervised label noise modeling and loss correction. In *International conference on machine learning*, 2019.

E. Atkinson, G. Baudart, L. Mandel, C. Yuan, and M. Carbin. Statically bounded-memory delayed sampling for probabilistic streams. In *Object-oriented Programming, Systems, Languages, and Applications*, 2021.

E. Atkinson, C. Yuan, G. Baudart, L. Mandel, and M. Carbin. Semi-symbolic inference for efficient streaming probabilistic programming. In *Object-oriented Programming, Systems, Languages, and Applications*, 2022.

W. Azizian, G. Baudart, and M. Lelarge. Automatic rao-blackwellization for sequential monte carlo with belief propagation. *arXiv preprint arXiv:2312.09860*, 2023.

G. Chang. Robust kalman filtering based on mahalanobis distance as outlier judging criterion. *Journal of Geodesy*, 88, 2014.

E. Y. Cheng, T. Millstein, G. V. d. Broeck, and S. Holtzen. flip-hoisting: Exploiting repeated parameters in discrete probabilistic programs. *arXiv preprint arXiv:2110.10284*, 2021.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, 1977.

P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2, 1992.

P. Cousot and M. Monerau. Probabilistic abstract interpretation. In *European Symposium on Programming*, 2012.

M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. In *Programming Language Design and Implementation*, 2019.

A. Di Pierro and H. Wiklicky. Concurrent constraint programming: Towards probabilistic abstract interpretation. In *Principles and Practice of Declarative Programming*, 2000.

A. Doucet, N. de Freitas, K. Murphy, and S. Russell. Rao-blackwellised particle filtering for dynamic bayesian networks. In *Conference on Uncertainty in Artificial Intelligence*, 2000.

J. Duník, O. Straka, O. Kost, and J. Havlík. 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.

D. Fink. A compendium of conjugate priors. 1997.

N. D. Goodman and A. Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014.

N. J. Gordon, D. J. Salmond, and A. F. Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. In *IEE Proceedings F (Radar and Signal Processing)*, volume 140, 1993.

M. Gorinova, D. Moore, and M. Hoffman. Automatic reparameterisation of probabilistic programs. In *International Conference on Machine Learning*, 2020.

M. I. Gorinova, A. D. Gordon, C. Sutton, and M. Vákár. Conditional independence by typing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(1), 2021.

M. D. Hoffman, M. J. Johnson, and D. Tran. Autoconj: Recognizing and exploiting conjugacy without a domain-specific language. In *Conference on Neural Information Processing Systems*, 2018.

S. Holtzen, G. Van den Broeck, and T. Millstein. Scaling exact inference for discrete probabilistic programs. *Object-oriented Programming, Systems, Languages, and Applications*, 4, 2020.

D. Huang, J.-B. Tristan, and G. Morrisett. Compiling markov chain monte carlo algorithms for probabilistic modeling. In *Programming Language Design and Implementation*, 2017.

J. Lai, J. Burroni, H. Guan, and D. Sheldon. Automatically marginalized mcmc in probabilistic programming. In *International Conference on Machine Learning*, 2023.

W. Lee, X. Rival, and H. Yang. Smoothness analysis for probabilistic programs with application to optimised variational inference. *Principles of Programming Languages*, 2023.

A. K. Lew, M. F. Cusumano-Towner, B. Sherman, M. Carbin, and V. K. Mansinghka. Trace types and denotational semantics for sound programmable inference in probabilistic languages. In *Principles of Programming Languages*, 2019.

W. Li, Y. Jia, J. Du, and J. Zhang. Phd filter for multi-target tracking with glint noise. *Signal Processing*, 94, 2014.

J. S. Liu. The collapsed gibbs aampler in bayesian computations with applications to a gene regulation problem. *Journal of the American Statistical Association*, 89(427), 1994.

D. Lundén. Delayed sampling in the probabilistic programming language anglican. 2017.

D. Lundén, J. Borgström, and D. Broman. Correctness of sequential monte carlo inference for probabilistic programming languages. In *European Symposium on Programming*, 2021.

Z. Ma and A. Leijon. Bayesian estimation of beta mixture models with variational inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(11), 2011.

V. Mansinghka, D. Selsam, and Y. Perov. Venture: A higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

V. K. Mansinghka, U. Schaechtle, S. Handa, A. Radul, Y. Chen, and M. Rinard. Probabilistic programming with programmable inference. In *Programming Language Design and Implementation*, 2018.

T. P. Minka. Expectation propagation for approximate bayesian inference. In *Conference on Uncertainty in Artificial Intelligence*, 2001.

D. Monniaux. Abstract interpretation of probabilistic semantics. In *International Static Analysis Symposium*, 2000.

D. Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. In *Principles of Programming Languages*, 2001.

L. Murray, D. Lundén, J. Kudlicka, D. Broman, and T. Schön. Delayed sampling and automatic rao-blackwellization of probabilistic programs. In *International Conference on Artificial Intelligence and Statistics*, 2018.

L. M. Murray and T. B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 46, 2018.

P. Narayanan, J. Carette, W. Romano, C.-c. Shan, and R. Zinkov. Probabilistic inference by program transformation in hakaru (system description). In *International Symposium on Functional and Logic Programming*, 2016.

V. C. Ngo, Q. Carbonneaux, and J. Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *Programming Language Design and Implementation*, 2018.

F. Obermeyer, E. Bingham, M. Jankowiak, D. Phan, and J. P. Chen. Functional tensors for probabilistic programming. *arXiv preprint arXiv:1910.10775*, 2019.

D. Ritchie, P. Horsfall, and N. D. Goodman. Deep amortized inference for probabilistic programs. *arXiv preprint arXiv:1610.05735*, 2016.

M. J. Smith. Probabilistic abstract interpretation of imperative programs using truncated normal distributions. *Electronic Notes in Theoretical Computer Science*, 220(3), 2008.

S. Staton. Commutative semantics for probabilistic programming. In *European Symposium on Programming*, 2017.

N. Tehrani, N. S. Arora, Y. L. Li, K. D. Shah, D. Noursi, M. Tingley, N. Torabi, E. Lippert, E. Meijer, et al. Bean machine: A declarative probabilistic programming language for efficient programmable inference. In *International Conference on Probabilistic Graphical Models*, 2020.

D. Tolpin, J.-W. van de Meent, H. Yang, and F. Wood. Design and implementation of probabilistic programming language anglican. In *Implementation and Application of Functional Languages*, 2016.

J. M. C. Trilla, M. Hicks, S. Magill, P. Mardziel, and I. Sweet. Probabilistic abstract interpretation: Sound inference and application to privacy. *Foundations of Probabilistic Programming*, 2020.

D. Wang and T. Reps. Newtonian program analysis of probabilistic programs. In *Object-oriented Programming, Systems, Languages, and Applications*, 2024.

W.-R. Wu. Target racking with glint noise. *IEEE Transactions on Aerospace and Electronic Systems*, 29, 1993.

Y. Zhou, H. Yang, Y. W. Teh, and T. Rainforth. Divide, conquer, and combine: A new inference strategy for probabilistic programs with stochastic support. In *International Conference on Machine Learning*, 2020.