Sound Probabilistic Inference via Guide Types

Di Wang
Carnegie Mellon University
USA

Jan Hoffmann
Carnegie Mellon University
USA

Thomas Reps
University of Wisconsin
USA

Abstract

Probabilistic programming languages aim to describe and automate Bayesian modeling and inference. Modern languages support *programmable inference*, which allows users to customize inference algorithms by incorporating *guide* programs to improve inference performance. For Bayesian inference to be sound, guide programs must be compatible with model programs. One pervasive but challenging condition for model-guide compatibility is *absolute continuity*, which requires that the model and guide programs define probability distributions with the same support.

This paper presents a new probabilistic programming language that *guarantees* absolute continuity, and features general programming constructs, such as branching and recursion. Model and guide programs are implemented as *coroutines* that communicate with each other to synchronize the set of random variables they sample during their execution. Novel *guide types* describe and enforce communication protocols between coroutines. If the model and guide are well-typed using the same protocol, then they are guaranteed to enjoy absolute continuity. An efficient algorithm infers guide types from code so that users do not have to specify the types. The new programming language is evaluated with an implementation that includes the type-inference algorithm and a prototype compiler that targets Pyro. Experiments show that our language is capable of expressing a variety of probabilistic models with nontrivial control flow and recursion, and that the coroutine-based computation does not introduce significant overhead in actual Bayesian inference.

CCS Concepts: • **Theory of computation → Probabilistic computation**; **Type structures**; • **Mathematics of computing → Probabilistic inference problems**.

Keywords: Probabilistic programming, Bayesian inference, type systems, coroutines

ACM Reference Format:
Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Sound Probabilistic Inference via Guide Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454077

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454077

1 Introduction

Probabilistic programming languages (PPLs) [1, 12, 22–24, 45, 48, 53, 58] provide a flexible way of describing statistical models and automatically performing Bayesian inference: a method for inferring the posterior of a statistical model from observed data. Bayesian inference accounts for uncertainty in latent variables that produce the observed data. It has applications in many fields, including artificial intelligence [21], cognitive science [25], and applied statistics [20].

Because there is not a single known inference algorithm that works well for all models [38], several PPLs have recently added support for *programmable inference* [7, 16, 19, 38, 43, 59]. This capability allows users to customize inference algorithms based on the characteristics of a particular model or dataset. Researchers have shown that programmable inference enables improved inference performance on a variety of modeling problems [7, 16, 18, 38].

Two important families of inference algorithms can be customized by incorporating *guide* programs, which are implemented by the user. The first family is *Monte-Carlo* methods, such as importance sampling and Markov-Chain Monte Carlo, where a guide program serves as a *proposal*, which generates random samples for latent variables. The second family is *variational inference*, where a guide program is a parameterized program that specifies a collection of *approximating* distributions on latent variables.

To ensure soundness of programmable inference, the guide programs have to be *compatible* with the implemented model program; incompatible guide programs could crash the inference process or lead to incorrect inference results [36, 37]. Recently, Lee et al. [36] developed a static analysis for finding bugs in model-guide pairs for variational inference in Pyro [7]. Lew et al. [37] proposed a type system that proves model-guide compatibility for multiple inference algorithms. However, neither approach handles general conditional statements that can influence the set of latent variables sampled by the model, and it is unclear how to extend them to analyze recursive programs precisely.

In this paper, we develop a new PPL that supports recursion and conditional statements, as well as guarantees *absolute continuity*, one of the most pervasive conditions for ensuring model-guide compatibility. Our PPL uses a new

paradigm for writing inference code: users implement the model and guide programs as *coroutines*, which can communicate with each other during their execution. We develop a new type system, which we dub *guide types*, to describe the communication protocols between coroutines. These guide types can be automatically inferred and are proof certificates of absolute continuity for model-guide pairs. They apply to multiple kinds of Bayesian-inference algorithms.

In our development, we follow a common scheme of *trace-based* programmable inference that underlies Pyro [7], Venture [38], Gen [16], etc. These PPLs define the meaning of a probabilistic program by a probability distribution on *sample traces* that record all the random samples that the program draws during its execution. A program p is *absolutely continuous* with respect to a program q, if any set of sample traces with non-zero probability under the program p must also have non-zero probability under the program q. In this paper, we reduce the problem of checking absolute continuity to the following verification task:

Given a model program p and a guide program q, verify that they define probability distributions with the same support, *i.e., they have the same set of possible sample traces.*

The major challenge in our development is to reason about the sets of possible sample traces for the model and guide programs, when the two programs can diverge in their execution, as always with relational reasoning. Control-flow constructs make it difficult to keep track of sample sites precisely; for example, a conditional statement can sample different sets of random variables in its two branches. It is intractable to enumerate all possible execution paths in the two programs and compare the sample sites path-to-path, especially when the programs are recursive.

The first part of our solution is to think of the model and guide programs as *coroutines* that can exchange messages. Conceptually, we use coroutine-style communication to *synchronize* each pair of sample sites that represent the same random variable, as well as each branch selection that influences control flow. The communication between the two coroutines should then be conducted according to a protocol so that messages always occur in *guidance* pairs: when one partner sends, the other receives; and when one partner offers a selection, the other branches.

The second part of our solution is to develop *guide types* as guidance protocols between the model and guide coroutines. In our formalization, we *structure* the sequence of messages between two coroutines, rather than describe it as a collection of unrelated messages. To handle general recursion, we parameterize the guide type for each coroutine by a *continuation* type that describes the guidance protocol for the computation that continues after a recursive invocation. We also develop an efficient algorithm that infers guide types automatically from the code.

There have been several type systems for coroutines [2, 3, 26], but all of them require that all messages from a coroutine to another have the same type; thus, they are not sufficient to handle *sample passing* and *branch selection* in our coroutine-based paradigm. In our development of guide types, we took inspiration from type systems for communication protocols in concurrent systems, such as *session types* [28, 29]. Guide types have different semantics from and are simpler than session types, and use a parametrization technique to model recursive computation.

We then establish formal guarantees of our new PPL. First, we prove that guide types ensure *safety* of communication between coroutines, i.e., the coroutines send and receive messages in a consistent manner. Second, we prove that guide types serve as proof certificates of absolute continuity between the model and guide programs; consequently, we use guide types to justify soundness of importance sampling, Markov-Chain Monte Carlo, and variational inference. Note that for variational inference, the soundness guarantee is *partial*, because sound inference requires some additional conditions (e.g., differentiability), whereas this paper focuses just on absolute continuity.

We implemented a type-inference algorithm for guide types and a prototype compiler from our PPL to Pyro. We evaluated our PPL on a broad suite of probabilistic models, and our experimental results show that (i) our PPL is more expressive than a state-of-the-art PPL that ensures soundness of programmable inference [37], and (ii) type inference completes in several milliseconds, and the performance of Bayesian inference on the compiled code is similar to hand-written Pyro code, i.e., coroutine communication does not introduce significant overhead.

Contributions. We make four main contributions.

- We develop a new PPL with a coroutine-based paradigm for implementing model and guide programs.
- We propose guide types, which prescribe guidance protocols between the model and guide coroutines, and develop an efficient inference algorithm for guide types.
- We prove type safety of guide types, and show that guide types ensure key soundness conditions of model-guide pairs for multiple kinds of Bayesian-inference algorithms.
- We implemented our PPL and evaluated its effectiveness on a variety of probabilistic models.

2 Overview

In this section, we first review Bayesian inference and trace-based programmable inference (§2.1). We then demonstrate the coroutine-based paradigm for implementing inference code and the use of guide types to enforce guidance protocols between coroutines. (§2.2).

2.1 Bayesian Inference

Probabilistic programs specify generative models that sample random variables. The semantics of a probabilistic program can be defined as a probability distribution on the *sample*

```
1 proc Model() =
2   v ← sample(@x, Gamma(2; 1));
3   if v < 2 then
4     _ ← sample(@z, Normal(−1; 1));
5     return(v)
6   else
7     m ← sample(@y, Beta(3; 1));
8     _ ← sample(@z, Normal(m; 1));
9     return(v)
```

Figure 1. A program *Model* with a conditional statement.

traces that record all the random values that a program draws during its execution [9, 35]. Consider the program *Model* in Fig. 1; it specifies a probabilistic model on random variables introduced by commands **sample**$(@\ell, d)$, where ℓ is a *label* that identifies a sample site in a program; and d is a *primitive distribution*, such as Gamma distributions whose support is the positive real line $\mathbb{R}_+$, Normal distributions whose support is the real line $\mathbb{R}$, and Beta distributions whose support is the unit interval $\mathbb{R}_{(0,1)}$. Two possible sample traces in the program *Model* are $[@x = 1; @z = -0.5]$ and $[@x = 3; @y = 0.9; @z = 0.7]$. More generally, the program specifies a distribution on sample traces whose support is

$$\begin{aligned} &\{[@x = a; @z = c] \mid 0 < a < 2\} \\ \cup\ &\{[@x = a; @y = b; @z = c] \mid a \geq 2, 0 < b < 1\}. \end{aligned} \tag{1}$$

Bayesian Inference amounts to conditioning a probabilistic model on *observations* and computing a posterior distribution on *latent variables*. For the program *Model*, we consider that $@z$ is the single $\mathbb{R}$-valued observation, while both $@x$ and $@y$ are latent variables. Intuitively, latent variables encode knowledge about the "ground truth" that we cannot observe directly, and the model program specifies a *prior* distribution on the "ground truth." Given a concrete value of the observation (e.g., $@z = 0.8$), the objective of Bayesian inference is to approximate the *posterior* distribution of the latent variables (e.g., likely values of $@x$ and $@y$ under the condition that $@z = 0.8$). Fig. 2 plots the prior distribution of the random variable $@x$, and its posterior distribution under the observation $@z = 0.8$.

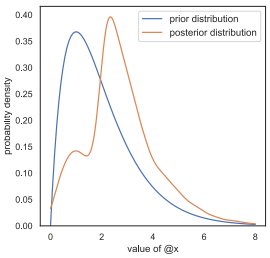

Figure 2. Probability densities of the prior and posterior distribution of the random variable $@x$.

It is usually intractable to sample directly from or even derive posterior distributions. There have been two popular families of inference algorithms: *Monte-Carlo methods* and *variational inference*. These inference algorithms usually require some *guide* programs, which can have a substantial influence on the performance of the inference. Although many PPLs provide mechanisms for automatically generating those guide programs, the ability to allow users to *customize* them, has been shown to be helpful, and sometimes crucial, for effective inference [7, 16, 18, 38]. However, customizability introduces non-trivial challenges to ensuring *soundness* of Bayesian inference. We now illustrate some mistakes when programming guide programs for Monte-Carlo methods and for variational inference.

Monte-Carlo methods. A Monte-Carlo method generates iteratively random samples such that empirical distribution of the samples approximates the posterior distribution. Two popular Monte-Carlo methods are *importance sampling* (IS) and *Markov-Chain Monte Carlo* (MCMC). IS generates independent and identically distributed samples from a *proposal* distribution, and reweights the samples by their *importance*, which corrects the discrepancy between the posterior and proposal distributions. MCMC generates iteratively a new random sample from an old one; that is, it constructs a Markov chain whose stationary distribution is the posterior distribution.

We now illustrate a mistake when programming guide programs for IS. For IS to converge asymptotically to the posterior distribution, the posterior distribution must be *absolutely continuous* with respect to the proposal distribution, i.e., any set of samples with non-zero probability under the posterior distribution must also have non-zero probability under the proposal distribution. In §5, we will show that it suffices to verify if the model program *conditioned* with respect to a concrete observation and the guide program have the same set of possible sample traces. For example, for the program *Model* shown in Fig. 1, the support of a sound guide program could be

$$\begin{aligned} &\{[@x = a] \mid 0 < a < 2\} \\ \cup\ &\{[@x = a; @y = b] \mid a \geq 2, 0 < b < 1\}, \end{aligned} \tag{2}$$

which is obtained by factoring out the observation $@z$ from the support of the unconditioned model shown in (1).

Fig. 3 presents two guide programs for performing IS from the program *Model* shown in Fig. 1, where the supports of the Pois and Unif distributions are natural numbers $\mathbb{N}$ and the unit interval $\mathbb{R}_{(0,1)}$, respectively. The support of the program $Guide_1$ is exactly the one shown in (2); thus, $Guide_1$ is a sound guide program; that is, $Guide_1$ samples the latent random variables $@x$, $@y$ from the same space as *Model* does. On the other hand, the support of the program $Guide_1'$ does not match (2), and it is actually an *unsound* guide program for two reasons:

- In the model program, the latent variable $@x$ can be any positive real number, whereas the program $Guide_1'$ only samples natural numbers for $@x$.
- In the model program, when the value of v (i.e., the latent variable $@x$) is greater than 2, the other latent variable $@y$ should be present in the sample trace. However, when

A *sound* guide

```
1 proc Guide1() =
2   v ← sample(@x, Gamma(1; 1));
3   if v < 2 then
4     return()
5   else
6     _ ← sample(@y, Unif);
7     return()
```

An *unsound* guide

```
1 proc Guide'1() =
2   v ← sample(@x, Pois(4));
3   if v > 10 then
4     return()
5   else
6     _ ← sample(@y, Unif);
7     return()
```

Figure 3. Sound and unsound guide programs for IS.

A *sound* guide

```
1 proc Guide2(θ1,θ2,θ3,θ4) =
2   v ← sample(@x,
3         Gamma(θ1; θ2));
4   if v < 2 then
5     return()
6   else
7     _ ← sample(@y,
8           Beta(θ3; θ4));
9     return()
```

An *unsound* guide

```
1 proc Guide'2(θ1,θ2) =
2   v ← sample(@x,
3         Normal(θ1; θ2));
4   if v < 2 then
5     return()
6   else
7     _ ← sample(@y,
8           Unif);
9     return()
```

Figure 4. Sound and unsound guide programs for VI.

the value of v is greater than 10, the program $Guide'_1$ will not produce a sample for $@y$.

Variational inference (VI). In contrast to Monte-Carlo methods, VI uses *optimization* (e.g., stochastic gradient descent) to find a candidate from an approximating family of distributions that minimizes the distance between the posterior distribution and the approximating distributions. In PPLs such as Pyro, users specify the approximating family by a *parameterized* probabilistic program called a *guide*; instantiating the parameters with a concrete valuation that produces a member of the approximating family. A widely used distance is the Kullback-Leibler (KL) divergence from the posterior distribution to the guide distribution. For the KL divergence to be well-defined, the guide distribution must be *absolutely continuous* with respect to the posterior distribution. In §5, we again reduce the verification of absolute continuity to checking a sufficient condition, namely, that the model conditioned with respect to a concrete observation and the guide have the same support. Note that VI requires several more conditions (such as differentiability) for inference to be sound [36]. In this paper, we focus on verification of absolute continuity.

Fig. 4 presents two guide programs for performing VI on the program *Model* shown in Fig. 1. The real-valued parameters of the guide programs are $\theta_1, \ldots, \theta_4$. The support of the program $Guide_2$ (instantiated with concrete parameters) is exactly the one shown in eq. (2). On the other hand, the program $Guide'_2$ defines an *unsound* guide, because it samples $@x$ from a normal distribution, whose support is the whole real line, whereas the program *Model* always samples a positive value for $@x$.

2.2 Sound Bayesian Inference via Guide Types

Programs as coroutines. Our first contribution is a *coroutine*-based paradigm for implementing the model and guide programs for Bayesian inference. In an inference algorithm, the model program and its guide program have many connections. The two most significant patterns we can observe in common inference algorithms are as follows:

- The guide program is used to generate sample traces, and then the model program is simulated with these traces to compute likelihoods.
- The guide program needs to have similar control-flow structure to that of the model program. For example, if the model program has a conditional command whose two branches sample different sets of latent variables, the guide program should also have a conditional command with an equivalent branch condition.

The first pattern illustrates a form of *sample passing* from the guide program to the model program, and the second pattern indicates that the model program should provide *branch selection* to the guide program. Such *bidirectional guidance* inspired us to treat the model and guide programs as *coroutines* that communicate with each other during their execution, rather than as totally independent programs. On the other hand, we do *not* want the coroutines to be tightly coupled: Bayesian practitioners usually maintain a separation between the model and the guide so that they can refine the guide iteratively to improve inference performance.

Therefore, we use *message-passing communication* to implement the coroutines; this formalism allows us to separate the model and the guide as individual programs, but connect them via *channels* over which coroutines exchange messages. Fig. 5 reimplements the model and guide programs in Fig. 1 and Fig. 3, respectively, by making the guidance communication explicit. The **sample**$(\cdot)$ commands and conditional commands are annotated with rv (i.e., "receive") or sd (i.e., "send") to indicate the direction of communication, and associated with a name of the channel on which the communication is carried out. In this example, we use two channels: *latent* for communication between the guide and the model, and *obs* for identifying observations in the model. Every channel has a unique provider and a unique consumer. Note that in this way we do *not* need to use labeled samples—as Pyro and some other PPLs do—because the sampling sites are synchronized through guidance communication.

Operationally, when a coroutine is executing a command associated with a channel c, it resumes the other coroutine that accesses channel c, until the other coroutine encounters a command that also communicates on channel c. Then they perform *synchronization*; for example,

- When *Model* executes $\mathbf{sample}_{\mathsf{rv}}\{latent\}(\textsc{Gamma}(2;1))$, it resumes the other end of the *latent* channel, i.e.,

Model

```
1  proc Model() consume latent provide obs =
2   v ← sample_rv{latent}(Gamma(2; 1));
3   if_sd{latent} v < 2 then
4    _ ← sample_sd{obs}(Normal(−1; 1));
5    return(v)
6   else
7    m ← sample_rv{latent}(Beta(3; 1));
8    _ ← sample_sd{obs}(Normal(m; 1));
9    return(v)
```

Guide

```
1  proc Guide_1() consume . provide latent =
2   v ← sample_sd{latent}(Gamma(1; 1));
3   if_rv{latent} ⋆ then
4    return()
5   else
6    _ ← sample_sd{latent}(Unif);
7    return()
```

Figure 5. Probabilistic programs as coroutines.

the coroutine $Guide_1$, until $Guide_1$ reaches the command $\mathbf{sample}_{sd}\{latent\}(\textsc{Gamma}(1;1))$. Recall that the guide program is used in importance sampling; thus, the coroutine $Guide_1$ draws a sample from the distribution $\textsc{Gamma}(1;1)$, and then sends it to the coroutine *Model*, which uses the sample and the prior distribution $\textsc{Gamma}(2;1)$ to calculate the importance weight.

- When $Guide_1$ executes the conditional command on line 3 (where the ⋆ symbol indicates that the branch selection is received from the other coroutine), it resumes the other end of the *latent* channel, i.e., the coroutine *Model*, until *Model* reaches the conditional command on line 3. The coroutine *Model* is the sender of the branch selection; thus, it evaluates the branch predicate $v < 2$, and sends the result back to $Guide_1$.

When the synchronization is completed, either coroutine can continue to execute.

Guide types. Our second contribution is *guide types* that enforce guidance protocols between coroutines, and an efficient algorithm that infers guide types from code.

We take inspiration from type systems for communication protocols in concurrent systems, such as *session types* [28, 29]. The key idea is to *structure* the sequence of guidance messages on a channel, rather than describe it as a collection of unrelated messages.

We sketch some type constructors in our development of guide types. The type $\mathbf{1}$ types an ended channel, where no messages can be exchanged. The type $A \,\&\, B$ types a channel whose provider waits for a branch selection, and continues with a protocol of type A or a protocol of type B based on the received selection. The type $\tau \wedge A$ types a channel whose provider samples and sends a random value of type τ, and then continues with a type A protocol. The guide type for a channel is the same for the provider and the consumer of the channel, but the two ends of a channel interpret the guide type for the channel *dually* (e.g., sends as receives).

With these three type constructors, we can express the protocols for the *latent* and *obs* channels shown in Fig. 5 as

$$latent\colon \mathbb{R}_+ \wedge (\mathbf{1} \,\&\, (\mathbb{R}_{(0,1)} \wedge \mathbf{1})), \tag{3}$$

$$obs\colon \mathbb{R} \wedge \mathbf{1}. \tag{4}$$

The provider and the consumer of the channel *latent* are the coroutines $Guide_1$ and *Model*, respectively. From the provider $Guide_1$'s perspective, the protocol shown as type (3) guides $Guide_1$ to draw a $\mathbb{R}_+$-valued sample and send it on *latent*, then wait for a branch selection, and finally end the communication on *latent* if the received branch selection is then-branch, otherwise draw an $\mathbb{R}_{(0,1)}$-valued sample before ending the communication. The coroutine $Guide_1$ implements this guidance protocol exactly. Meanwhile, from the consumer *Model*'s perspective, the type constructors have *dual* semantics, i.e., *send* becomes *receive* and vice versa; thus, the protocol for *latent* guides *Model* to receive an $\mathbb{R}_+$-valued sample, and then send out a branch selection on channel *latent*; if *Model* selects the else-branch, then it further receives an $\mathbb{R}_{(0,1)}$-valued sample on channel *latent*.

The channel *obs*, whose provider is the coroutine *Model*, is used to identify observations in the probabilistic model. The coroutine *Model* accesses *obs* on lines 4 and 8, each of which lies in a branch of the conditional command on line 3. Because the conditional command is associated with *latent*, it should *not* bother with the communication on channel *obs*; thus, we require that the two branches of the conditional command have the same guidance protocol for *obs*. The protocol shown as type (4) specifies that the coroutine *Model* produces a single $\mathbb{R}$-valued observation, and *Model* implements this protocol exactly.

Recursion. Probabilistic programs can use recursion to express complex generative models, such as a *probabilistic context-free grammar* (PCFG), which is a popular model for constructing languages [33]. Fig. 6 shows a recursive model that generates a random expression tree with two constructors: $\mathrm{Const}(\cdot)$ for leaf nodes and $\mathrm{Add}(\cdot;\cdot)$ for internal nodes.

To support recursion in probabilistic programs, we add a standard recursive-type constructor to guide types. However, *composition* of the guide types from multiple procedure calls in a non-tail-recursive program remains a challenge. One straightforward approach is to add a sequencing type $A \fatsemi B$ that types a channel whose provider starts with a type A protocol and then continues with a type B protocol, but such sequencing types will complicate the type system, because they allow a guidance protocol to be described by *different* types. For example, both $(\mathbb{R} \wedge \mathbb{R} \wedge \mathbf{1})$ and $((\mathbb{R} \wedge \mathbf{1}) \fatsemi (\mathbb{R} \wedge \mathbf{1}))$ describe a channel whose provider sends two $\mathbb{R}$-valued random samples.

```
1  proc Pcfg() consume latent provide . =
2   k ← sample_rv{latent}(Beta(3; 1));
3   call PcfgGen(k)
4
5  proc PcfgGen(k) consume latent provide . =
6   u ← sample_rv{latent}(Unif);
7   if_sd{latent} u < k then
8    v ← sample_rv{latent}(Normal(0; 1));
9    return(Const(v))
10  else
11   lhs ← call PcfgGen(k);
12   rhs ← call PcfgGen(k);
13   return(Add(lhs; rhs))
```

Figure 6. A recursive probabilistic model.

To sidestep the need for a nontrivial equivalence check in the type system, we adapt the idea of *type-level polymorphism*, and parameterize the guide type for a recursive coroutine by a *continuation* type that describes the communication after a procedure call to this coroutine returns. For example, consider the following parametric type $\mathsf{R}[\cdot]$.

$$\mathsf{R}[X] \stackrel{\text{def}}{=} \mathbb{R}_{(0,1)} \wedge ((\mathbb{R} \wedge X) \,\&\, \mathsf{R}[\mathsf{R}[X]]),$$

It specifies a guidance protocol by *prepending* messages to the continuation protocol defined by the type parameter X. The type $\mathsf{R}[X]$ precisely describes the behavior of the *PcfgGen* coroutine shown in Fig. 6: the coroutine first receives an $\mathbb{R}_{(0,1)}$-valued random sample (line 6); evaluates and sends out a branch selection (line 7); and then based on the branch selection, the coroutine either receives an $\mathbb{R}$-valued random sample (line 8) and then returns (i.e., continues with the continuation protocol X), or makes two recursive procedure calls (lines 11 and 12). The guide type of the else-branch can be justified by backward reasoning: at line 13, the coroutine returns (i.e., continues with the continuation protocol X); at line 12, because the guide type *after* the procedure call is X, we obtain the guide type *before* the procedure call by instantiating R with X; and at line 11, because the guide type *after* the procedure call is $\mathsf{R}[X]$, we again instantiate R, but with $\mathsf{R}[X]$, to derive the guide type of the else-branch. Finally, for the coroutine *Pcfg* shown in Fig. 6, we derive $\mathbb{R}_{(0,1)} \wedge \mathsf{R}[\mathbf{1}]$ as the guidance protocol for channel *latent*.

Control-flow divergence. In Fig. 5, the model program *Model* and the guide program $Guide_1$ have very similar control flow. In general, our type system permits the guide's control-flow structure to diverge from the model's, as long as the two programs communicate with each other in a consistent way, i.e., the two programs follow the same guidance protocol for the channel over which they communicate. For example, the program below implements a part of a Bayesian linear-regression model with outliers [16], where the latent variable *prob_outlier* describes how likely a data point does not conform to the linear relationship, and *is_outlier* is a Boolean-valued latent variable that indicates if a data point is an outlier.

```
1  prob_outlier ← sample_rv{latent}(Unif);
2  is_outlier ← sample_rv{latent}(Ber(prob_outlier));
3  return()
```

For MCMC algorithms, the guide program generates a new random sample from an old one; thus, for better inference performance, an MCMC guide usually behaves differently for different old samples. The following program implements a part of a guide that branches on *is_outlier* from the old sample [16]. Intuitively, this guide proposes the negation (with a small amount of noise) of the old *is_outlier*, which is bound to a program variable *old_is_outlier*; i.e., if the old *is_outlier* is true (resp., false), then the guide is likely to propose false (resp., true).

```
1  prob_outlier ← sample_sd{latent}(Beta(2; 5));
2  if old_is_outlier then
3   is_outlier ← sample_sd{latent}(Ber(0.1));
4   return()
5  else
6   is_outlier ← sample_sd{latent}(Ber(0.9));
7   return()
```

Although the model and the guide have divergent control-flow structures, in our type system, we can express the guidance protocol for channel *latent* as $\mathbb{R}_{(0,1)} \wedge \mathbb{2} \wedge \mathbf{1}$; that is, both programs sample an $\mathbb{R}_{(0,1)}$-valued random variable and then sample a Boolean-valued one.

Type inference. Guide types can be automatically inferred from code; in practice, they can still be used as specifications of the programs for better understanding. Our implementation can infer guide types for the examples mentioned so far, including the recursive one shown in Fig. 6.

3 A Coroutine-Based PPL

In this section, we formulate a core monadic calculus for coroutine-based probabilistic programming.

Syntax. Fig. 7 presents the grammar of basic types τ, expressions e, values v, commands m, and programs $\mathcal{D}$ in the core calculus via abstract binding trees [26]. There is a modal distinction in the core language: expressions describe *purely deterministic* computations, while commands describe *probabilistic* computations. Intuitively, we treat randomness as a kind of monadic effect [42].

The purely deterministic fragment is a simply-typed lambda calculus augmented with *scalar* types (i.e., nullary products $\mathbb{1}$, Booleans $\mathbb{2}$, unit interval $\mathbb{R}_{(0,1)}$, positive real numbers $\mathbb{R}_+$, real numbers $\mathbb{R}$, integer rings $\mathbb{N}_n$, and natural numbers $\mathbb{N}$), as well as a *distribution* type $\mathsf{dist}(\tau)$. The syntactic form $\mathsf{op}_\diamond(e_1; e_2)$ represents expressions that perform built-in binary operations $\diamond$ on scalar values. Inhabitants of $\mathsf{dist}(\tau)$ are the primitive distributions from which probabilistic programs can draw a random value of type τ; for example, Bernoulli distributions $\textsc{Ber}(\cdot)$ have type $\mathsf{dist}(\mathbb{2})$, the uniform distribution on unit interval $\textsc{Unif}$ has type $\mathsf{dist}(\mathbb{R}_{(0,1)})$, and geometric distributions $\textsc{Geo}(\cdot)$ have type $\mathsf{dist}(\mathbb{N})$. For each primitive distribution d, we assume that it admits two fields:

$$
\begin{aligned}
\tau &::= \mathbb{1} \mid \mathbb{2} \mid \mathbb{R}_{(0,1)} \mid \mathbb{R}_+ \mid \mathbb{R} \mid \mathbb{N}_n \mid \mathbb{N} \mid \tau_1 \to \tau_2 \mid \mathsf{dist}(\tau) \\
e &::= x \mid \mathsf{triv} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}(e; e_1; e_2) \mid \bar{r} \mid \bar{n} \mid \mathsf{op}_\diamond(e_1; e_2) \\
&\quad \mid \lambda(x.e) \mid \mathsf{app}(e_1; e_2) \mid \mathsf{let}(e_1; x.e_2) \\
&\quad \mid \textsc{Ber}(e) \mid \textsc{Unif} \mid \textsc{Beta}(e_1; e_2) \mid \textsc{Gamma}(e_1; e_2) \\
&\quad \mid \textsc{Normal}(e_1; e_2) \mid \textsc{Cat}(e_1, \cdots, e_n) \mid \textsc{Geo}(e) \mid \textsc{Pois}(e) \\
v &::= \mathsf{triv} \mid \mathsf{true} \mid \mathsf{false} \mid \bar{r} \mid \bar{n} \mid \mathsf{clo}(V, \lambda(x.e)) \\
&\quad \mid \textsc{Ber}(v) \mid \textsc{Unif} \mid \textsc{Beta}(v_1; v_2) \mid \textsc{Gamma}(v_1; v_2) \\
&\quad \mid \textsc{Normal}(v_1; v_2) \mid \textsc{Cat}(v_1, \cdots, v_n) \mid \textsc{Geo}(v) \mid \textsc{Pois}(v) \\
m &::= \mathsf{ret}(e) \mid \mathsf{bnd}(m_1; x.m_2) \mid \mathsf{call}(f; e) \\
&\quad \mid \mathsf{sample}_{\mathsf{rv}}\{a\}(e) \mid \mathsf{sample}_{\mathsf{sd}}\{a\}(e) \\
&\quad \mid \mathsf{cond}_{\mathsf{rv}}\{a\}(m_1; m_2) \mid \mathsf{cond}_{\mathsf{sd}}\{a\}(e; m_1; m_2) \\
\mathcal{D} &::= \overrightarrow{\mathsf{fix}\{a; b\}(f.x.m)}
\end{aligned}
$$

Figure 7. Syntax of the core calculus.

d.support and d.density are the support and the density function of the distribution, respectively. In the core calculus, the type of a primitive distribution characterizes the support of the distribution *precisely*: for a distribution d of type $\mathsf{dist}(\tau)$ and a value v, it holds that $v \in d$.support if and only if v is an inhabitant of type τ. Primitive distributions can be generalized to density-carrying expressions [5, 6] to further improve language expressibility.

The probabilistic fragment is a monadic calculus augmented with probabilistic constructs and communication primitives for coroutine-based programming. The *sampling* commands $\mathsf{sample}_{\mathsf{sd}}\{a\}(e)$ and $\mathsf{sample}_{\mathsf{rv}}\{a\}(e)$ first evaluate the expression e to a primitive distribution d. Then the *send* version $\mathsf{sample}_{\mathsf{sd}}\{a\}(d)$ draws a value from d and sends it on channel a, whereas the *receive* version $\mathsf{sample}_{\mathsf{rv}}\{a\}(d)$ receives a value from channel a and treats it as a sample from d. The random samples can influence the *likelihoods* of computations; thus, randomness can be seen as a source of side effects. The *branching* commands also have a *send* version $\mathsf{cond}_{\mathsf{sd}}\{a\}(e; m_1; m_2)$, which evaluates e to a Boolean value and sends it as the branch selection on channel a; and a *receive* version $\mathsf{cond}_{\mathsf{rv}}\{a\}(m_1; m_2)$, which receives a branch selection from channel a. The syntactic form $\mathsf{call}(f; e)$ represents a procedure call, where f is a procedure name and e is the argument.

A probabilistic program $\mathcal{D}$ is a collection of (mutually recursive) procedures, each of which has the form $\mathsf{fix}\{a; b\}(f.x.m)$, where f is the procedure name, x is the parameter, m is a command that represents the procedure body, a is the name of the channel consumed by f, and b is the name of the channel provided by f. Note that both a and b are optional; that is, the procedure f might not consume any channel, and it might not provide any channel.

Semantics. We develop a big-step operational semantics for the core calculus. Details of the semantics are included in the technical report [55]. The evaluation judgments for expressions have the form $V \vdash e \Downarrow v$, where V is an *environment* that maps program variables to values. The evaluation rules for expressions are skipped here because they are standard.

We adopt a trace-based approach [9, 35] in our semantics of probabilistic computations. A *guidance trace* σ is a finite sequence of guidance messages exchanged on a channel; each *guidance message* has the form $\boldsymbol{val}^{\mathrm{P}}(v)$ (resp., $\boldsymbol{dir}^{\mathrm{P}}(v)$) for a sample value v (resp., a branch selection v) from the provider to the consumer, the form $\boldsymbol{val}^{\mathrm{C}}(v)$ (resp., $\boldsymbol{dir}^{\mathrm{C}}(v)$) for a sample value v (resp., a branch selection v) from the consumer to the provider, or a procedure-call indicator $\boldsymbol{fold}$.[1] The evaluation judgments for commands have the form $V \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash m \Downarrow^w v$, where V is an environment, m is a command that consumes channel a and provides channel b, σ_a and σ_b are guidance traces on the channels, v is the evaluation result, and $w \geq 0$ is a *weight* that expresses how likely the guidance traces are. Intuitively, a probabilistic program specifies a probability distribution on guidance traces, and the weights represent *probability densities* with respect to the distribution.

Fig. 8 shows the evaluation rules for selected commands. We use the following notational conventions. We denote the empty environment by $\emptyset$, and updating a binding of x in an environment V to v by $V[x \mapsto v]$. We use the $+\!\!+$ operator to concatenate two traces. We write ite as a shorthand for if-then-else. The *Iverson brackets* $[\cdot]$ are defined by $[\varphi] = 1$ if φ is true and otherwise $[\varphi] = 0$.

The (EM:Sample:*) rules take a value from the guidance traces as the result of the sampling, and use the density functions of primitive distributions to calculate the weight for the guidance traces. The (EM:Cond:Send:L) rule evaluates the branch predicate to obtain a Boolean value, and enforce that the branch selection from the guidance trace of the consumed channel must be the same as the predicate's value; if the guidance trace sets the branch selection to a different value, we simply set the weight of this trace to zero. The (EM:Call) rule requires the guidance traces start with a $\boldsymbol{fold}$ message, and proceeds by evaluating the body of the callee.

Example 3.1. Consider the command

$$
\begin{aligned}
m_1 \stackrel{\text{def}}{=}\ & \mathsf{bnd}(\ \mathsf{sample}_{\mathsf{rv}}\{a\}(\textsc{Normal}(0;1));\ x. \\
& \mathsf{bnd}(\ \mathsf{sample}_{\mathsf{sd}}\{b\}(\textsc{Normal}(x;1));\ y. \\
& \mathsf{ret}(\mathsf{op}_+(x;y))\ )\ ),
\end{aligned}
$$

which consumes a channel a and provides a channel b. Let $\varphi \stackrel{\text{def}}{=} \lambda x. \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$ be the probability density function of the standard normal distribution $\textsc{Normal}(0;1)$. Let $\sigma_a \stackrel{\text{def}}{=}$

[1] The $\boldsymbol{fold}$ message is only useful in the theoretical development; it can be seen as the introduction form for guidance traces whose type is a type-operator instantiation (see §4).

$$\text{(EM:Ret)}\quad \frac{V \vdash e \Downarrow v}{V \mid (a\colon []); (b\colon []) \vdash \mathsf{ret}(e) \Downarrow^{1} v}$$

$$\text{(EM:Bnd)}\quad \frac{V \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash m_1 \Downarrow^{w_1} v_1 \qquad V[x \mapsto v_1] \mid (a\colon \sigma_a'); (b\colon \sigma_b') \vdash m_2 \Downarrow^{w_2} v_2}{V \mid (a\colon \sigma_a +\!\!+ \sigma_a'); (b\colon \sigma_b +\!\!+ \sigma_b') \vdash \mathsf{bnd}(m_1; x.m_2) \Downarrow^{w_1 \cdot w_2} v_2}$$

$$\text{(EM:Sample:Recv:L)}\quad \frac{V \vdash e \Downarrow d \qquad v \in d.\mathsf{support} \qquad w = d.\mathsf{density}(v)}{V \mid (a\colon [\boldsymbol{val}^{\mathrm{P}}(v)]); (b\colon []) \vdash \mathsf{sample}_{\mathsf{rv}}\{a\}(e) \Downarrow^{w} v}$$

$$\text{(EM:Sample:Send:R)}\quad \frac{V \vdash e \Downarrow d \qquad v \in d.\mathsf{support} \qquad w = d.\mathsf{density}(v)}{V \mid (a\colon []); (b\colon [\boldsymbol{val}^{\mathrm{P}}(v)]) \vdash \mathsf{sample}_{\mathsf{sd}}\{b\}(e) \Downarrow^{w} v}$$

$$\text{(EM:Cond:Send:L)}\quad \frac{V \vdash e \Downarrow v_e \qquad i = \mathsf{ite}(v_a, 1, 2) \qquad V \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash m_i \Downarrow^{w} v}{V \mid (a\colon [\boldsymbol{dir}^{\mathrm{C}}(v_a)] +\!\!+ \sigma_a); (b\colon \sigma_b) \vdash \mathsf{cond}_{\mathsf{sd}}\{a\}(e; m_1; m_2) \Downarrow^{w \cdot [v_a = v_e]} v}$$

$$\text{(EM:Cond:Recv:R)}\quad \frac{i = \mathsf{ite}(v_b, 1, 2) \qquad V \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash m_i \Downarrow^{w} v}{V \mid (a\colon \sigma_a); (b\colon [\boldsymbol{dir}^{\mathrm{C}}(v_b)] +\!\!+ \sigma_b) \vdash \mathsf{cond}_{\mathsf{rv}}\{b\}(m_1; m_2) \Downarrow^{w} v}$$

$$\text{(EM:Call)}\quad \frac{\mathcal{D}(f) = \mathsf{fix}\{a; b\}(f.x_f.m_f) \qquad V \vdash e \Downarrow v_1 \qquad \emptyset[x_f \mapsto v_1] \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash m_f \Downarrow^{w} v_2}{V \mid (a\colon [\boldsymbol{fold}] +\!\!+ \sigma_a); (b\colon [\boldsymbol{fold}] +\!\!+ \sigma_b) \vdash \mathsf{call}(f; e) \Downarrow^{w} v_2}$$

Figure 8. Selected evaluation rules for commands.

$[\boldsymbol{val}^{\mathrm{P}}(\bar{1})]$ and $\sigma_b \stackrel{\text{def}}{=} [\boldsymbol{val}^{\mathrm{P}}(\bar{2})]$. Then we can derive the evaluation judgment

$$\emptyset \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash m_1 \Downarrow^{\varphi(1)\cdot\varphi(1)} \bar{3},$$

for the command m_1 and the guidance traces σ_a, σ_b.

Communication. There are a lot of formalisms for communication in (concurrent) programming systems, such as CCS [39], Theoretical CSP [27], and π-calculus [40, 41]. In this paper, we use a lightweight approach to handling communication; that is, in the semantics, we assume we have all the messages exchanged on all the communication channels. We use this formalism because (i) our focus is to reason about soundness of Bayesian inference, rather than concurrency-related properties (e.g., deadlock freedom); and (ii) the inference algorithms we study in §5 involve only two coroutines—one for the model and the other for the guide—so the communication in our system is much simpler than that in general concurrent systems.

Example 3.2. Consider the command

$$m_2 \stackrel{\text{def}}{=} \mathsf{bnd}(\ \mathsf{sample}_{\mathsf{sd}}\{a\}(\textsc{Normal}(3;1));\ \_.\ \mathsf{ret}(\mathsf{triv})\ ),$$

which provides a channel a that is consumed by the command m_1 in Ex. 3.1. To model the communication between m_2 and m_1, we simply use the guidance trace $\sigma_a = [\boldsymbol{val}^{\mathrm{P}}(\bar{1})]$ as the sequence of messages exchanged on channel a in the semantics, and derive evaluation judgments for m_2 and m_1 separately. We showed the judgment for m_1 in Ex. 3.1; here, we can derive the judgment

$$\emptyset \mid \varnothing; (a\colon \sigma_a) \vdash m' \Downarrow^{\varphi(-2)} \mathsf{triv},$$

for command m_2 and guidance trace σ_a. We use the $\varnothing$ symbol to indicate that m_2 does not consume any channel.

4 Guide Types

Type formation. We take inspiration from a *structuring* principle in session types [28, 29], and develop *guide types* to enforce protocols for guidance traces. The grammar shown below formulates the syntax of guide types. We write A, B for guide types, X for type variables, T for unary type operators, and F for procedure signatures.

$$A, B ::= X \mid \mathbf{1} \mid T[A] \mid \tau \wedge A \mid \tau \supset A \mid A \oplus B \mid A \,\&\, B$$
$$F ::= \tau_1 \rightsquigarrow \tau_2 \mid (a\colon T_a); (b\colon T_b)$$
$$\mathcal{T} ::= \overrightarrow{\mathsf{typedef}(T.X.A)}$$

The type $\mathbf{1}$ indicates an ended channel, where the guidance trace is empty. The type $T[A]$ instantiates a unary type operator T with a guide type A. For sample passing and branch selection, each type constructor has a dual version that reverses the role of the provider and the consumer. The type $\tau \wedge A$ types a channel whose *provider* samples a random value , sends it on the channel, and then continues with a type A guidance protocol; dually, the type $\tau \supset A$ types a channel whose *consumer* samples and sends a random value. Similarly, the type $A \oplus B$ types a channel whose *provider* evaluates a branch predicate, sends a branch selection on the channel, and then continues with a type A guidance protocol or a type B protocol based on the branch selection; dually, the type $A \,\&\, B$ types a channel whose *consumer* evaluates and sends a branch selection.

Remark 4.1. In the rest of this paper, we will not *use the dual types $\tau \supset A$ and $A \oplus B$. We introduce these types here for theoretical completeness, and they may be used in some future development.*

Type operators prescribe guidance protocols for procedures by parameterizing with a continuation type that describes the guidance protocol after a procedure call. A procedure signature $\tau_1 \rightsquigarrow \tau_2 \mid (a\colon T_a); (b\colon T_b)$ types a procedure that takes a parameter of type τ_1, returns a result of type τ_2, consumes a channel a, and provides a channel b, such that if the guidance protocols for a and b *after* a procedure call are A and B, respectively, then the guidance protocols for a and b *before* the procedure call are $T_a[A]$ and $T_b[B]$, respectively.

A *type definition* $\mathsf{typedef}(T.X.A)$ declares a unary type operator T that takes a type parameter X and produces a guide type A, which can reference X. Because type operators are used to prescribe procedure signatures, we assume that a

(TM:Ret)
$$\frac{\Gamma \vdash e : \tau}{\Gamma \mid (a\colon A);(b\colon B) \vdash \mathsf{ret}(e) \div \tau \mid (a\colon A);(b\colon B)}$$

(TM:Bnd)
$$\frac{\Gamma \mid (a\colon A);(b\colon B) \vdash m_1 \div \tau_1 \mid (a\colon A');(b\colon B') \quad \Gamma, x : \tau_1 \mid (a\colon A');(b\colon B') \vdash m_2 \div \tau_2 \mid (a\colon A'');(b\colon B'')}{\Gamma \mid (a\colon A);(b\colon B) \vdash \mathsf{bnd}(m_1; x.m_2) \div \tau_2 \mid (a\colon A'');(b\colon B'')}$$

(TM:Sample:Recv:L)
$$\frac{\Gamma \vdash e : \mathsf{dist}(\tau)}{\Gamma \mid (a\colon \tau \wedge A);(b\colon B) \vdash \mathsf{sample}_{\mathsf{rv}}\{a\}(e) \div \tau \mid (a\colon A);(b\colon B)}$$

(TM:Sample:Send:R)
$$\frac{\Gamma \vdash e : \mathsf{dist}(\tau)}{\Gamma \mid (a\colon A);(b\colon \tau \wedge B) \vdash \mathsf{sample}_{\mathsf{sd}}\{b\}(e) \div \tau \mid (a\colon A);(b\colon B)}$$

(TM:Cond:Send:L)
$$\frac{\Gamma \vdash e : 2 \quad \Gamma \mid (a\colon A_1);(b\colon B) \vdash m_1 \div \tau \mid (a\colon A');(b\colon B') \quad \Gamma \mid (a\colon A_2);(b\colon B) \vdash m_2 \div \tau \mid (a\colon A');(b\colon B')}{\Gamma \mid (a\colon A_1 \,\&\, A_2);(b\colon B) \vdash \mathsf{cond}_{\mathsf{sd}}\{a\}(e; m_1; m_2) \div \tau \mid (a\colon A');(b\colon B')}$$

(TM:Cond:Recv:R)
$$\frac{\Gamma \mid (a\colon A);(b\colon B_1) \vdash m_1 \div \tau \mid (a\colon A');(b\colon B') \quad \Gamma \mid (a\colon A);(b\colon B_2) \vdash m_2 \div \tau \mid (a\colon A');(b\colon B')}{\Gamma \mid (a\colon A);(b\colon B_1 \,\&\, B_2) \vdash \mathsf{cond}_{\mathsf{rv}}\{b\}(m_1; m_2) \div \tau \mid (a\colon A');(b\colon B')}$$

(TM:Call)
$$\frac{\Sigma(f) = \tau_1 \rightsquigarrow \tau_2 \mid (a\colon T_a);(b\colon T_b) \quad \Gamma \vdash e : \tau_1}{\Gamma \mid (a\colon T_a[A]);(b\colon T_b[B]) \vdash \mathsf{call}(f; e) \div \tau_2 \mid (a\colon A);(b\colon B)}$$

Figure 9. Selected typing rules for commands.

probabilistic program is always accompanied by a collection $\mathcal{T}$ of (mutually recursive) type definitions.

Example 4.2. We can formally declare the type operator Recur for the *PcfgGen* procedure shown in Fig. 6 as typedef(R. X. $\mathbb{R}_{(0,1)} \wedge ((\mathbb{R} \wedge X) \,\&\, \mathsf{R}[\mathsf{R}[X]])$).

Typing rules. The typing judgments for expressions have the form $\Gamma \vdash e : \tau$, where Γ is a *typing context* that maps program variables to basic types (defined in Fig. 7). A full list of typing rules is included in the technical report [55]. The typing rules for expressions are skipped here because they are standard.

The typing judgments for commands have the form
$$\Gamma \mid (a\colon A);(b\colon B) \vdash_\Sigma m \div \tau \mid (a\colon A');(b\colon B'),$$
where Σ maps procedure identifiers to procedure signatures. The intuitive meaning of the typing judgment is that if the channels a and b are of the guidance protocols A and B, respectively, then we can evaluate the command m to a value of type τ, and after the evaluation, the channels a and b are of the guidance protocols A' and B', respectively.

Fig. 9 presents the typing rules for commands. We assume a fixed global Σ that we omit from the rules. Intuitively, the rules formulate a *backward*-reasoning system: we start with continuation types A' and B' for the channels a and b, respectively, and then prepend the guidance messages sent or received by the command m to A' and B', to obtain the guide types A and B for the channels a and b before the evaluation of m, respectively. For sample passing and branch selection, each guide type has two derivation rules: one for the consumed channel a, and the other for the provided channel b. For example, the type $\tau \wedge A$ represents a channel whose provider sends a sample of type τ; thus, if the consumed channel a has such a type, the rule (TM:Sample:Recv:L) *receives* a sample from the provider of a, and if the provided channel b has such a type, the rule (TM:Sample:Send:R) *sends* a sample to the consumer of b.

The rule (TM:Call) handles procedure calls. For a procedure call $\mathsf{call}(f; e)$, the rule fetches from Σ the procedure f's signature $\tau_1 \rightsquigarrow \tau_2 \mid (a\colon T_a);(b\colon T_b)$, and then instantiates the type operators T_a, T_b with continuation types A, B, respectively, to obtain the guide types $T_a[A]$ and $T_b[B]$ for the channels a and b before the procedure call, respectively.

Example 4.3. Consider the command
$$\begin{aligned} m_3 \stackrel{\text{def}}{=}\ & \mathsf{bnd}(\ \mathsf{call}(f; k);\ \_. \\ & \mathsf{bnd}(\ \mathsf{sample}_{\mathsf{rv}}\{\textsc{Normal}(0;1)\}(a);\ \_. \\ & \mathsf{bnd}(\ \mathsf{call}(f; k);\ \_. \\ & \mathsf{ret}(\mathsf{triv})\ )\ )\ ), \end{aligned}$$
where the variable k has type $\mathbb{R}_{(0,1)}$ and the procedure f has signature $\mathbb{R}_{(0,1)} \rightsquigarrow \mathbb{1} \mid (a\colon T);\varnothing$, i.e., the procedure f consumes channel a but does not provide any channel, and channel a is associated with a type operator T. Now we show that we can derive a typing judgment for m_3 by backward reasoning. First, by (TM:Ret), we have
$$k : \mathbb{R}_{(0,1)} \mid (a\colon \mathbf{1});\varnothing \vdash_\Sigma \mathsf{ret}(\mathsf{triv}) \div \mathbb{1} \mid (a\colon \mathbf{1});\varnothing.$$
Then by (TM:Call), we derive
$$k : \mathbb{R}_{(0,1)} \mid (a\colon T[\mathbf{1}]);\varnothing \vdash_\Sigma \mathsf{call}(f; k) \div \mathbb{1} \mid (a\colon \mathbf{1});\varnothing.$$
Define $m_4 \stackrel{\text{def}}{=} \mathsf{bnd}(\mathsf{call}(f; k); \_.\mathsf{ret}(\mathsf{triv}))$. Thus, by (TM:Bnd),
$$k : \mathbb{R}_{(0,1)} \mid (a\colon T[\mathbf{1}]);\varnothing \vdash_\Sigma m_4 \div \mathbb{1} \mid (a\colon \mathbf{1});\varnothing.$$
Define $m_5 \stackrel{\text{def}}{=} \mathsf{bnd}(\mathsf{sample}_{\mathsf{rv}}\{a\}(\textsc{Normal}(0;1)); \_.m_4)$. By (TM:Sample:Recv:L) and (TM:Bnd), we have
$$k : \mathbb{R}_{(0,1)} \mid (a\colon \mathbb{R} \wedge T[\mathbf{1}]);\varnothing \vdash_\Sigma m_5 \div \mathbb{1} \mid (a\colon \mathbf{1});\varnothing.$$
Finally, we again apply (TM:Call) and (TM:Bnd) to derive
$$k : \mathbb{R}_{(0,1)} \mid (a\colon T[\mathbb{R} \wedge T[\mathbf{1}]]);\varnothing \vdash_\Sigma m_3 \div \mathbb{1} \mid (a\colon \mathbf{1});\varnothing.$$

Type safety. We present some theoretical results about type safety of guide types. Proofs are included in the technical report [55].

We first formulate two judgments for well-formedness of values and guidance traces. The judgment $v : \tau$ means that value v has type τ. The judgment $\sigma : A$ means that the guidance trace is a sequence of messages that satisfies protocol A. Rules for these judgments are straightforward; we omit them here but include them in the technical report [55].

The theorem below states that if m is a well-typed closed command, and it evaluates to a value v under guidance traces σ_a, σ_b, then v is a well-typed value, and σ_a, σ_b are well-typed guidance traces.

Theorem 4.4. *If* $\cdot \mid (a\colon A); (b\colon B) \vdash_\Sigma m \div \tau \mid (a\colon \mathbf{1}); (b\colon \mathbf{1})$ *and* $\emptyset \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash m \Downarrow^w v$, *then* $\sigma_a : A$, $\sigma_b : B$, *and* $v : \tau$.

Furthermore, we can show some *normalization* properties of guide types. The theorem below states that if m is a well-typed closed command, and σ_a, σ_b are well-typed guidance traces, then m can evaluate to some well-typed v under σ_a, σ_b.

Theorem 4.5. *If* $\cdot \mid (a\colon A); (b\colon B) \vdash_\Sigma m \div \tau \mid (a\colon \mathbf{1}); (b\colon \mathbf{1})$, $\sigma_a : A$, *and* $\sigma_b : B$, *then there exist* w, v *such that* $\emptyset \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash m \Downarrow^w v$ *and* $v : \tau$.

We can strengthen the normalization property when a command will *not* send out any branch selections. The theorem below states that if a well-typed command m consumes a channel a with a type A that does not contain $\&$ and provides a channel b with a type B that does not contain $\oplus$, and σ_a, σ_b are well-typed guidance traces, then m can evaluate to some well-typed value v under σ_a, σ_b with a *strictly positive* weight w.

Theorem 4.6. *If* $\cdot \mid (a\colon A); (b\colon B) \vdash_\Sigma m \div \tau \mid (a\colon \mathbf{1}); (b\colon \mathbf{1})$, A *is* $\&$*-free,* B *is* $\oplus$*-free,* $\sigma_a : A$, *and* $\sigma_b : B$, *then there exist* w, v *such that* $\emptyset \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash m \Downarrow^w v$, $v : \tau$, *and* $w > 0$.

Type-inference algorithm. We now sketch a type-inference algorithm that derives guide types automatically from the implementation. In the algorithm, we assume we have information about basic types—such as the parameter and result types for procedures and the typing contexts that map program variables to basic types—because without guide types, our core language is a simply-typed lambda calculus, for which type inference is decidable.

First, for each procedure $\mathrm{fix}\{a; b\}(f.x.m)$ in the program, we create two fresh type operators T_a and T_b for the channels a and b, respectively, and obtains $\tau_1 \rightsquigarrow \tau_2 \mid (a\colon T_a); (b\colon T_b)$ as the signature of this procedure. Then we collect signatures of all the procedures in the program to obtain the map Σ.

Now the task is to derive definitions of the type operators. We observe that the rules in Fig. 9 are syntax directed, and they can be turned into an algorithmic system by interpreting

$$\Gamma \mid (a\colon A); (b\colon B) \vdash_\Sigma m \div \tau \mid (a\colon A'); (b\colon B')$$

as a function from $\Sigma, \Gamma, m, \tau, a, b, A', B'$ to A, B; i.e., we assume we know all the basic types, and we perform backward reasoning to infer guide types. Therefore, for each procedure $\mathrm{fix}\{a; b\}(f.x.m)$ with signature $\tau_1 \rightsquigarrow \tau_2 \mid (a\colon T_a); (b\colon T_b)$, we create two fresh type variables X_a and X_b, derive two guide types A and B through

$$x : \tau_1 \mid (a\colon A); (b\colon B) \vdash_\Sigma m \div \tau_2 \mid (a\colon X_a); (b\colon X_b),$$

and then add type definitions $\mathrm{typedef}(T_a.X_a.A)$ and $\mathrm{typedef}(T_b.X_b.B)$.

5 Soundness of Bayesian Inference

In this section, we use guide types to reason about Bayesian inference. We first present a measure-theoretic formulation of Bayesian inference in the coroutine-based PPL, and prove that guide types are *certificates* of absolute continuity (§5.1). We then sketch how guide types ensure key soundness conditions for multiple Bayesian-inference algorithms (§5.2). The technical report [55] includes the details (e.g., formalizations and proofs) of this section.

5.1 Verification of Absolute Continuity

We use the following notions from measure theory: σ-algebras, measurable spaces, measurable functions, measures, and Lebesgue integration.

Semantic domains. For each scalar type τ, we equip it with a *standard Borel space* $[\![\tau]\!]$ on the inhabitants of τ, i.e., $[\![\tau]\!]$ is a measurable space isomorphic to a countable set or the real line. We then equip each type τ with a *stock measure* $\lambda_{[\![\tau]\!]}$: if $[\![\tau]\!]$ is a countable set, we define $\lambda_{[\![\tau]\!]}$ to be the counting measure; otherwise, $[\![\tau]\!]$ is a subset of the real line, so we define $\lambda_{[\![\tau]\!]}$ to be the Lebesgue measure.

Because guidance traces are finite sequences of messages that contain values of scalar types, we can define $[\![A]\!]$ as a standard Borel space on guidance traces that satisfy protocol A. We then construct the stock measure $\lambda_{[\![A]\!]}$ for A by decomposing A to products and/or sums of scalar types, and then combining the stock measures for scalar types via product and/or coproduct measures.

Denotation of commands. For a well-typed closed command m, i.e., $\cdot \mid (a\colon A); (b\colon B) \vdash_\Sigma m \div \tau \mid (a\colon \mathbf{1}); (b\colon \mathbf{1})$, we define the *density function* of m as

$$\mathbf{P}_m(\sigma_a, \sigma_b) \stackrel{\text{def}}{=} \begin{cases} w & \text{if } \emptyset \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash m \Downarrow^w v \\ 0 & \text{otherwise} \end{cases}.$$

We can prove that $\mathbf{P}_m$ is a measurable function from $[\![A]\!] \otimes [\![B]\!]$—the product measurable space of $[\![A]\!]$ and $[\![B]\!]$—to non-negative real numbers. Thus, we construct a measure denotation $[\![m]\!]$ for m, by integrating $\mathbf{P}_m$ with respect to the stock measure on the product space $[\![A]\!] \otimes [\![B]\!]$, i.e.,

$$[\![m]\!](S_{a,b}) \stackrel{\text{def}}{=} \int_{S_{a,b}} \mathbf{P}_m(\sigma_a, \sigma_b) \lambda_{[\![A]\!] \otimes [\![B]\!]}(d(\sigma_a, \sigma_b)),$$

where $S_{a,b}$ is a measurable set in $[\![A]\!] \otimes [\![B]\!]$.

Bayesian inference. Let us fix a well-typed model program m_{m} that consumes latent random variables on a channel *latent* and provides observations on a channel *obs*, i.e.,

$$\cdot \mid (\mathit{latent}\colon A); (\mathit{obs}\colon B) \vdash_\Sigma m_{\mathrm{m}} \div \tau_{\mathrm{m}} \mid (\mathit{latent}\colon \mathbf{1}); (\mathit{obs}\colon \mathbf{1}).$$

Usually, the program m_{m} does *not* receive any branch selections, i.e., A is $\oplus$-free and B is $\&$-free. Given a concrete observation $\sigma_o : B$ such that $\int \mathbf{P}_{m_{\mathrm{m}}}(\sigma_\ell, \sigma_o) \lambda_{[\![A]\!]}(d\sigma_\ell) > 0$,

Bayesian inference is the problem of approximating the *posterior* $[\![m_{\mathrm{m}}]\!]_{\sigma_o}$, a measure *conditioned* with respect to σ_o, defined by

$$[\![m_{\mathrm{m}}]\!]_{\sigma_o}(S_\ell) \stackrel{\text{def}}{=} \frac{\int_{S_\ell} \mathbf{P}_{m_{\mathrm{m}}}(\sigma_\ell, \sigma_o)\lambda_{[\![A]\!]}(d\sigma_\ell)}{\int \mathbf{P}_{m_{\mathrm{m}}}(\sigma_\ell, \sigma_o)\lambda_{[\![A]\!]}(d\sigma_\ell)}, \tag{5}$$

where S_ℓ is a measurable set in $[\![A]\!]$, i.e., a set of guidance traces of type A. Note that if we fix the observation σ_o, then the denominator of eq. (5) is a constant independent of S_ℓ. Thus, it is sufficient for an inference algorithm to ignore the denominator and approximate the measure $S_\ell \mapsto \int_{S_\ell} \mathbf{P}_{m_{\mathrm{m}}}(\sigma_\ell, \sigma_o)\lambda_{[\![A]\!]}(d\sigma_\ell)$.

Guide programs. Bayesian-inference algorithms usually require some guide programs, such as proposals for importance sampling and approximating families for variational inference. These guide programs specify measures on latent random variables; in our system, we implement a guide program m_{g} as a coroutine that works with the model program m_{m} and provides the *latent* channel with guide type A that m_{m} consumes, i.e.,

$$\cdot \mid \varnothing; (\mathit{latent}\colon A) \vdash_\Sigma m_{\mathrm{g}} \div \tau_{\mathrm{g}} \mid \varnothing; (\mathit{latent}\colon \mathbf{1}),$$
$$\cdot \mid (\mathit{latent}\colon A); (\mathit{obs}\colon B) \vdash_\Sigma m_{\mathrm{m}} \div \tau_{\mathrm{m}} \mid (\mathit{latent}\colon \mathbf{1}); (\mathit{obs}\colon \mathbf{1}).$$

The guide and model have the *same* guide type A on channel *latent*. Because the guide *provides* the channel and the model *consumes* the channel, the two programs interpret the guide type A *dually*; thus, their communication is compatible.

The coroutine-based paradigm folds the model and guide programs into a single entity; thus, during the inference, both the model and guide coroutines execute. To model *possible* combinations of traces for a model-guide system, we introduce a reduction relation $V \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash_{\mathsf{red}} m \Downarrow v$, where V is an environment, m is a command, σ_a and σ_b are guidance traces on channel a and channel b, respectively, and v is the reduction result. The reduction relation is essentially the same as the evaluation relation for the operational semantics, except that reduction does *not* account for probabilities. Below are two example rules.

(RM:Sample:Send:R)
$$\frac{V \vdash e \Downarrow d \qquad v \in d.\mathrm{support}}{V \mid (a\colon []); (b\colon [\boldsymbol{val}^{\mathrm{P}}(v)]) \vdash_{\mathsf{red}} \mathsf{sample}_{\mathsf{sd}}\{b\}(e) \Downarrow v}$$

(RM:Cond:Send:L)
$$\frac{V \vdash e \Downarrow v_e \qquad i = \mathsf{ite}(v_e, 1, 2) \qquad V \mid (a\colon \sigma_a); (b\colon \sigma_b) \vdash_{\mathsf{red}} m_i \Downarrow v}{V \mid (a\colon [\boldsymbol{dir}^{\mathrm{C}}(v_e)] +\!\!+ \sigma_a); (b\colon \sigma_b) \vdash_{\mathsf{red}} \mathsf{cond}_{\mathsf{sd}}\{a\}(e; m_1; m_2) \Downarrow v}$$

With the reduction relation, we say that a combination of traces (σ_ℓ, σ_o) is *possible* for the model program m_{m} and the guide program m_{g}, if $\emptyset \mid (\mathit{latent}\colon \sigma_\ell); (\mathit{obs}\colon \sigma_o) \vdash_{\mathsf{red}} m_{\mathrm{m}} \Downarrow v_{\mathrm{m}}$ and $\emptyset \mid \varnothing; (\mathit{latent}\colon \sigma_\ell) \vdash_{\mathsf{red}} m_{\mathrm{g}} \Downarrow v_{\mathrm{g}}$ for some values v_{m} and v_{g}. We prove a lemma that connects the reduction relation with command denotations.

Lemma 5.1. *Suppose that A is $\oplus$-free, B is $\&$-free, and*

$$\cdot \mid \varnothing; (\mathit{latent}\colon A) \vdash_\Sigma m_{\mathrm{g}} \div \tau_{\mathrm{g}} \mid \varnothing; (\mathit{latent}\colon \mathbf{1}),$$
$$\cdot \mid (\mathit{latent}\colon A); (\mathit{obs}\colon B) \vdash_\Sigma m_{\mathrm{m}} \div \tau_{\mathrm{m}} \mid (\mathit{latent}\colon \mathbf{1}); (\mathit{obs}\colon \mathbf{1}).$$

Then a combination of traces (σ_ℓ, σ_o) is possible for the model m_{m} and the guide m_{g} if and only if $\mathbf{P}_{m_{\mathrm{m}}}(\sigma_\ell, \sigma_o) \neq 0$.

We can now define a denotation for the guide m_{g}, accompanied by the model m_{m} and conditioned on a concrete observation $\sigma_o : B$, as a measure defined on possible traces:

$$[\![m_{\mathrm{g}}]\!]^{m_{\mathrm{m}}}_{\sigma_o}(S_\ell) \stackrel{\text{def}}{=} \int_{S_\ell} [\mathbf{P}_{m_{\mathrm{m}}}(\sigma_\ell, \sigma_o) \neq 0] \cdot \mathbf{P}_{m_{\mathrm{g}}}(\sigma_\ell)\lambda_{[\![A]\!]}(d\sigma_\ell),$$

where S_ℓ is a measurable set in $[\![A]\!]$.

Absolute continuity. A measure μ is said to be *absolutely continuous* with respect to a measure ν, if μ and ν are defined on the same measurable space, and $\nu(S) \neq 0$ for every measurable set S for which $\mu(S) \neq 0$.

We prove that for a model-guide pair, guide types serve as certificates for absolute continuity.

Theorem 5.2. *Suppose that A is $\oplus$-free, B is $\&$-free,*

$$\cdot \mid \varnothing; (\mathit{latent}\colon A) \vdash_\Sigma m_{\mathrm{g}} \div \tau_{\mathrm{g}} \mid \varnothing; (\mathit{latent}\colon \mathbf{1}),$$
$$\cdot \mid (\mathit{latent}\colon A); (\mathit{obs}\colon B) \vdash_\Sigma m_{\mathrm{m}} \div \tau_{\mathrm{m}} \mid (\mathit{latent}\colon \mathbf{1}); (\mathit{obs}\colon \mathbf{1}),$$

and $\sigma_o : B$ such that $\int \mathbf{P}_{m_{\mathrm{m}}}(\sigma_\ell, \sigma_o)\lambda_{[\![A]\!]}(d\sigma_\ell) > 0$. Then the measure $[\![m_{\mathrm{m}}]\!]_{\sigma_o}$ is absolutely continuous with respect to the measure $[\![m_{\mathrm{g}}]\!]^{m_{\mathrm{m}}}_{\sigma_o}$, and vice versa.

5.2 Soundness of Inference Algorithms

We now describe how guide types can help us reason about inference algorithms.

Importance sampling (IS). IS approximates the posterior distribution by drawing latent variables using the guide program, and then reweights the samples by their *importance*. The operational rule below formulates a single step in the algorithm: given a model program m_{m}, a guide program m_{g}, and a concrete observation σ_o, IS performs joint execution of the two programs to draw a sample σ_ℓ with density w_{g} and compute $\frac{w_{\mathrm{m}}}{w_{\mathrm{g}}}$ as the importance of σ_ℓ.

$$\frac{\emptyset \mid \varnothing; (\mathit{latent}\colon \sigma_\ell) \vdash m_{\mathrm{g}} \Downarrow^{w_{\mathrm{g}}} \_ \qquad \emptyset \mid (\mathit{latent}\colon \sigma_\ell); (\mathit{obs}\colon \sigma_o) \vdash m_{\mathrm{m}} \Downarrow^{w_{\mathrm{m}}} \_}{m_{\mathrm{g}}; m_{\mathrm{m}}; \sigma_o \vdash^{w_{\mathrm{g}}}_{\mathrm{IS}} \langle \sigma_\ell, {}^{w_{\mathrm{m}}}/_{w_{\mathrm{g}}} \rangle}$$

By Thm. 5.2, if the model and guide programs are well-typed, then the posterior $[\![m_{\mathrm{m}}]\!]_{\sigma_o}$ is absolutely continuous with respect to $[\![m_{\mathrm{g}}]\!]^{m_{\mathrm{m}}}_{\sigma_o}$; thus, IS is able to sample any possible latent variables σ_ℓ in the posterior. With the importance ratios, IS can be seen as generating σ_ℓ with density $w_{\mathrm{g}} \cdot \frac{w_{\mathrm{m}}}{w_{\mathrm{g}}} = w_{\mathrm{m}}$. Thus, IS generates a measure proportional to $[\![m_{\mathrm{m}}]\!]_{\sigma_o}$.

Markov-Chain Monte Carlo (MCMC). MCMC uses a transition kernel to generate iteratively a new random sample from an old one. A popular MCMC algorithm is *Metropolis-Hastings* (MH), which constructs the transition kernel from a *proposal* subroutine. To implement proposal subroutines in our system, we extend the core calculus such that guidance

traces can be used as first-class data. Then we implement the proposal subroutine as a procedure g whose argument is a guidance trace on the channel for latent random variables. The operational rule below formulates a single step in the MH algorithm; given a proposal procedure g, a model m_{m}, an observation σ_o, and the current latent trace σ_ℓ, MH first performs joint execution of $\mathsf{call}(g;\sigma_\ell)$ and m_{m} to generate a new latent trace σ'_ℓ with density w_{fwd}, and then uses the new σ'_ℓ and the old σ_ℓ to calculate a *backward* density w_{bwd}. MH then computes an *acceptance ratio* $\alpha \stackrel{\mathrm{def}}{=} \min(1, \frac{w'_{\mathrm{m}} \cdot w_{\mathrm{bwd}}}{w_{\mathrm{m}} \cdot w_{\mathrm{fwd}}})$, and accepts the new sample σ'_ℓ with probability α.

$$\frac{\begin{array}{l}\emptyset \mid \varnothing; (\mathit{latent}\colon \sigma'_\ell) \vdash \mathsf{call}(g;\sigma_\ell) \Downarrow^{w_{\mathrm{fwd}}} \_ \\ \emptyset \mid (\mathit{latent}\colon \sigma'_\ell); (\mathit{obs}\colon \sigma_o) \vdash m_{\mathrm{m}} \Downarrow^{w'_{\mathrm{m}}} \_ \\ \emptyset \mid \varnothing; (\mathit{latent}\colon \sigma_\ell) \vdash \mathsf{call}(g;\sigma'_\ell) \Downarrow^{w_{\mathrm{bwd}}} \_ \\ \emptyset \mid (\mathit{latent}\colon \sigma_\ell); (\mathit{obs}\colon \sigma_o) \vdash m_{\mathrm{m}} \Downarrow^{w_{\mathrm{m}}} \_\end{array}}{g; m_{\mathrm{m}}; \sigma_o \vdash_{\mathrm{MH}} \sigma_\ell \overset{w_{\mathrm{fwd}} \cdot \alpha}{\Longrightarrow} \sigma'_\ell}$$

Similar to IS, MH requires that the command $\mathsf{call}(g;\sigma_\ell)$ be able to sample any possible latent variables σ'_ℓ in the posterior. We prove the soundness of MH by a variant of Thm. 5.2, where the programs do not need to be closed so that they can reference data in the environment (e.g., the old samples).

Variational inference (VI). VI uses optimization to find a candidate from an approximating family of guide programs that minimizes the distance from the posterior distribution to the guide distribution. We focus on verifying if the distance is well-defined, whereas VI requires extra conditions for the optimization problem to be well-formed. Here, we parameterize the guide $m_{\mathrm{g},\theta}$ by a vector $\theta \in \Theta$ of parameters, and use KL divergence as the distance, which is defined by

$$\mathrm{KL}(\mu \parallel \nu) \stackrel{\mathrm{def}}{=} \int p_\mu(\sigma_\ell)(\log p_\mu(\sigma_\ell) - \log p_\nu(\sigma_\ell))\lambda_{[\![A]\!]}(d\sigma_\ell),$$

where μ and ν are measures on $[\![A]\!]$ with densities p_μ and p_ν, respectively, and μ is absolutely continuous with respect to ν. The rule below formulates the computation of KL divergence for a specific θ, via joint execution of the two programs.

$$\frac{\begin{array}{l}\emptyset \mid \varnothing; (\mathit{latent}\colon \sigma_\ell) \vdash m_{\mathrm{g},\theta} \Downarrow^{w_{\mathrm{g}}} \_ \\ \emptyset \mid (\mathit{latent}\colon \sigma_\ell); (\mathit{obs}\colon \sigma_o) \vdash m_{\mathrm{m}} \Downarrow^{w_{\mathrm{m}}} \_\end{array}}{m_{\mathrm{g},\theta}; m_{\mathrm{m}}; \sigma_o \vdash_{\mathrm{VI}}^{w_{\mathrm{g}}} \langle \sigma_\ell, \log w_{\mathrm{m}} - \log w_{\mathrm{g}} \rangle}$$

The rule can be seen as defining a map $\sigma_\ell \mapsto w_{\mathrm{g}} \cdot (\log w_{\mathrm{m}} - \log w_{\mathrm{g}})$, which is the integrand of the divergence $\mathrm{KL}([\![m_{\mathrm{g},\theta}]\!]^{m_{\mathrm{m}}}_{\sigma_o} \parallel [\![m_{\mathrm{m}}]\!]_{\sigma_o})$. By Thm. 5.2, if the model and guide programs are well-typed, then $[\![m_{\mathrm{g},\theta}]\!]^{m_{\mathrm{m}}}_{\sigma_o}$ is absolutely continuous with respect to $[\![m_{\mathrm{m}}]\!]_{\sigma_o}$; thus, the KL divergence used in VI is well-defined.

6 Experimental Evaluation

Implementation. We implemented the coroutine-based PPL in OCaml. Our implementation consists of about 2,000 LOC; it contains a parser, a type checker with automatic inference of guide types, and a prototype compiler from our PPL to Pyro [7]. Our implementation extends the core calculus with tensors (i.e., multi-dimensional matrices) and primitive iteration operators for them. The prototype compiler supports code generation for importance sampling and variational inference. We use the Python package greenlet [57] to support coroutines in the compiled code.

Evaluation setup. We evaluated our implementation to answer the following two research questions:

1. How expressive is the coroutine-based PPL, compared to a state-of-the-art probabilistic programming language that ensures soundness of programmable inference [37]?
2. How efficient is our implementation, in terms of the time for type inference, and the performance of Bayesian inference on the compiled code?

For the first question, we obtained 23 benchmarks from prior work [37] and collected 6 new benchmarks. The 29 benchmark programs consist of (i) example models from Anglican [58], Turing [19], and Pyro [7], as well as (ii) PCFG models, including a Gaussian-process domain-specific language (DSL) [46] and synthetic models (such as examples shown in this paper). Compared to prior work [37], a larger subset of benchmark models are expressible and type-checked in our PPL. Particularly, our PPL is capable of expressing models with recursion and general conditional branches, whereas prior work [37] is not.

For the second question, we ran Bayesian inference on the compiled code, and compared the performance with non-coroutine-based, but equivalent, Pyro code. We obtained guide programs from where we obtained the benchmark models, and then reimplemented them in our PPL; for example, we implemented the encoder component of a variational autoencoder as the guide program [7]. For those benchmark models without guides, we first invoked our PPL to type-check the model program and infer a guide type for the model, and then implemented a guide program whose type was the guide type. The compiled model and guide use Pyro's primitives (such as `pyro.sample`) to sample random data and condition on given data, as well as exchange messages and switch control with each other using the concurrent-programming package `greenlet`. We leveraged Pyro's inference engines to carry out importance sampling or variational inference. Type inference is very fast in practice; our implementation completed the type-inference phase in several milliseconds on all of the benchmarks. Our experiments showed that coroutines (implemented via messaging passing) do not introduce significant overhead in actual Bayesian inference.

The experiments were performed on a machine with an Intel Core i7 3.6GHz processor and 16GB of RAM under macOS Catalina 10.15.7.

Results. Tab. 1 gives an overview of selected benchmark models. Our benchmarks cover a wide range of Bayesian models, such as linear regression, Gaussian mixtures, hidden Markov models, Bayesian networks, and variational autoencoders. Our benchmarks also include the classic Marsaglia

```
1  proc Ptrace(λ) consume latent provide obs =
2   k ← call PtraceHelper(e^{-λ}, 0, 1);
3   sample_sd{obs}(Normal(k; 0.1))
4
5  proc PtraceHelper(l, k, p) consume latent provide . =
6   u ← sample_rv{latent}(Unif);
7   if_sd{latent} p · u ≤ l then
8    return(k)
9   else
10   call PtraceHelper(l, k + 1, p · u)
```

Figure 10. An algorithm to generate Poisson-distributed numbers given by Knuth [34].

Table 1. Selected benchmark descriptions. **T?** = is type-checked in our PPL; **LOC** = #lines of code of the model in our PPL; **TP?** = is type-checked by prior work [37].

Program	Description	T?	LOC	TP?
lr	Bayesian Linear Regression	✓	16	✓
gmm	Gaussian Mixture Model	✓	44	✓
kalman	Kalman Smoother	✓	32	✓
sprinkler	Bayesian Network	✓	22	✓
hmm	Hidden Markov Model	✓	31	✓
branching	Random Control Flow	✓	19	✗
marsaglia	Marsaglia Algorithm	✓	22	✗
dp	Dirichlet Process	✗	N/A	✗
ptrace	Poisson Trace	✓	11	✗
aircraft	Aircraft Detection	✓	32	✓
weight	Unreliable Weigh	✓	8	✓
vae	Variational Autoencoder	✓	26	✓
ex-1	Fig. 5	✓	13	✗
ex-2	Fig. 6	✓	21	✗
gp-dsl	Gaussian Process DSL	✓	58	✗

algorithm (which generates a normal distribution from a uniform distribution), a Poisson-trace algorithm (shown in Fig. 10, which generates a Poisson distribution from a uniform distribution), and a Gaussian-process DSL (which uses a PCFG to generate the kernel function of a Gaussian process).

As shown in Tab. 1, our coroutine-based PPL is capable of expressing most of the benchmarks, except those involving stochastic memoization [23], such as the program dp. The programs branching, marsaglia, ptrace, and ex-1 have non-trivial branching, and the programs marsaglia, ptrace, ex-2, and gp-dsl define recursive models; our implementation successfully inferred guide types for these programs, whereas prior work [37] could not express them. Our implementation derived guide types for 25 of the 29 benchmarks, whereas prior work was able to express only 18 of them.

For all the benchmarks, we assume that each guide program samples random variables in the *same* order as its corresponding model program does. However, this assumption can sometimes be too restrictive: it has been shown that the ability to allow the model and the guide to sample random variables in *different* orders is desirable for inference amortization methods [56]. Prior work [37] allows different sampling orders in the model and the guide, whereas our system cannot handle such scenarios.

Table 2. Selected performance statistics. **BI** = Bayesian-inference algorithm (IS or VI); **CG (ms)** = time for type inference and code generation in milliseconds; **GLOC** = #lines of code in compiled code (model + guide); **GI (s)** = time for Bayesian inference on compiled code in seconds; **HLOC** = #lines of code in handwritten code (model + guide); **HI (s)** = time for Bayesian inference on handwritten code in seconds.

Program	BI	CG (ms)	GLOC	GI (s)	HLOC	HI (s)
ex-1	IS	0.75	57	5.44	16	5.27
branching	IS	1.74	58	8.49	16	7.48
gmm	IS	8.03	185	64.13	38	56.00
weight	VI	0.66	35	2.76	7	2.66
vae	VI	10.36	72	34.96	26	32.69

Tab. 2 presents performance statistics of selected benchmark programs. We evaluated our PPL's performance under two criteria: (i) the time for type inference and code generation, and (ii) the time for Bayesian inference compared to handwritten inference code under the same set of hyperparameters (e.g., iteration rounds, optimization algorithms, and initial values of parameters). Our experiments showed that our implementation usually completes type inference and code generation in several milliseconds, and the compiled code, although using coroutines, has similar performance to handwritten inference code.

7 Related Work

Sound Bayesian inference. Most closely related to our work are techniques for reasoning about soundness of trace-based programmable inference. Lee et al. [36] developed a static analysis of stochastic variational inference with guide programs, which describe custom approximating families in Pyro. Their analysis supports nontrivial features of Pyro, such as tensor manipulation and *plates*, i.e., vectors of conditionally independent samples. Their approach aims at proving that the model and guide programs have the same support and satisfy differentiability-related conditions. Their static analysis does not handle the case when a conditional statement determines the set of random samples. Lew et al. [37] proposed *trace types* as precise signatures for sampling traces of probabilistic programs, and then used the type system to prove absolute continuity in multiple kinds of inference algorithms. Trace types can be seen as a type-and-effect system, where a trace type records the precise set of samples drawn by a single program. Trace types support higher-order functions, *stochastic* branches that can influence the set of random samples, as well as three forms of loops, including stochastic while-loops with an unbounded number of iterations, but not general recursion. Because the value of a conditional predicate cannot be determined in

general at static-analysis time, trace types do not support general conditional statements that can influence the set of random samples. Both Lee et al. [36]'s and Lew et al. [37]'s approach allow the model and the guide to sample random variables in different orders. In this paper, we propose a new PPL that guarantees absolute continuity between a model-guide pair, and features general programming constructs, including recursion and branching. A key innovation of our work is the coroutine-based paradigm of writing inference code; this paradigm makes the relational reasoning of the support-match property explicit, and in particular enables precise analysis of complex control flow. However, compared to prior work, our system only supports scenarios where the model and the guide sample random variables in the same order.

There has been a line of work on validating Monte-Carlo inference algorithms. Ścibior et al. [49] developed a semantic framework to verify the soundness of Monte-Carlo inference algorithms with generic proposal distributions. Atkinson et al. [4] presented a type system for verifying hand-coded Monte-Carlo algorithms that explicitly manipulate densities, rather than use proposal distributions. For MCMC methods, Borgström et al. [9] and Hur et al. [32] developed provably correct MH algorithms. Castellan and Paquet [13] proposed an *intensional* semantics, which captures execution traces of programs, to validate an incremental MH algorithm. Several systems [4, 8, 31, 37] studied sound combinators for kernels used by MCMC. In contrast to the aforementioned work, our PPL is based on trace-based programmable inference. It would be interesting to develop programmable versions of those sound inference algorithms in our PPL.

Narayanan et al. [44] and Zinkov and Shan [59] validated the soundness of program transformations in Hakaru, which contains a programmable MH algorithm. The development of Hakaru is not centered around sample traces, and it uses *symbolic disintegration* [14, 50] to calculate the marginal densities for computing the acceptance ratio in an MH step. In this paper, we focus on a trace-based scheme for programmable inference. Establishing the relationship among different schemes of programmable inference is an interesting future research direction.

Session types. Honda et al. [28, 29] introduced session types to prescribe binary communication protocols for message-passing processes. Session types can be interpreted either classically [54], or intuitionistically [10, 11]. To enable non-binary communication, researchers proposed multiparty session types [15, 30, 47]. The tail-recursive structure of standard session types imposes communication protocols that can be described by a *regular* language. Recently, several systems have been developed to go beyond tail-recursive protocols, such as context-free [51], label-dependent [52], and nested [17] session types.

In our development of guide types, we took inspiration from the structuring principle of session types. Compared to session types, guide types have different semantics (i.e., sending and receiving random samples drawn from probability distributions), have simpler forms (i.e., no process spawning or higher-order channels), and enjoy an efficient type-inference algorithm, which can also analyze non-tail-recursive communication protocols. Developing a truly concurrent probabilistic programming system, and concurrent Bayesian inference algorithms with general session types, would be interesting future work.

8 Conclusion

We have presented a new probabilistic programming language that supports programmable Bayesian inference, and guarantees model-guide absolute continuity, thereby ensuring key soundness properties of multiple kinds of inference algorithms. Our language implements the model and guide programs as *coroutines*, and we develop *guide types* to prescribe the communication protocols between coroutines. We have proved that well-typed model and guide coroutines execute safely, and they are guaranteed to enjoy absolute continuity. We have also developed an efficient type-inference algorithm that reconstructs guide types directly from the code. Finally, we have implemented our language with a prototype compiler to Pyro, and evaluated our implementation on a suite of diverse probabilistic models.

Acknowledgments

This article is based on research supported, in part, by a gift from Rajiv and Ritu Batra; by ONR under grants N00014-17-1-2889 and N00014-19-1-2318; by DARPA under AA contract FA8750-18-C0092; and by the NSF under SaTC award 1801369, SHF awards 1812876 and 2007784, and CAREER award 1845514. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Jessica Ai, Nimar S. Arora, Ning Dong, Beliz Gokkaya, Thomas Jiang, Anitha Kubendran, Arun Kumar, Michael Tingley, and Narjes Torabi. 2019. HackPPL: A Universal Probabilistic Programming Language. In *Int. Workshop on Machine Learning and Prog. Lang. (MAPL'19)*. https://doi.org/10.1145/3315508.3329974

[2] Konrad Anton and Peter Thiemann. 2010. Towards Deriving Type Systems and Implementations for Coroutines. In *Asian Symp. on Prog. Lang. and Systems (APLAS'10)*. https://doi.org/10.1007/978-3-642-17164-2_6

[3] Konrad Anton and Peter Thiemann. 2010. Typing Coroutines. In *Trends in Functional Programming (TFP'10)*. https://doi.org/10.1007/978-3-642-22941-1_2

[4] Eric Atkinson, Cambridge Yang, and Michael Carbin. 2018. Verifying Handcoded Probabilistic Inference Procedures. https://arxiv.org/abs/1805.01863

[5] Sooraj Bhat, Ashish Agarwal, Richard Vuduc, and Alexander Gray. 2012. A Type Theory for Probability Density Functions. In *Princ. of Prog. Lang. (POPL'12)*. https://doi.org/10.1145/2103656.2103721

[6] Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. 2013. Deriving Probability Density Functions from Probabilistic Functional Programs. In *Tools and Algs. for the Construct. and Anal. of Syst. (TACAS'13)*. https://doi.org/10.1007/978-3-642-36742-7_35

[7] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rishabh Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. *J. Machine Learning Research* 20, 1 (January 2018). https://dl.acm.org/doi/10.5555/3322706.3322734

[8] Keith A. Bonawitz. 2008. *Composable Probabilistic Inference with Blaise*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[9] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A Lambda-Calculus Foundation for Universal Probabilistic Programming. In *Int. Conf. on Functional Programming (ICFP'16)*. https://doi.org/10.1145/2951913.2951942

[10] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *Int. Conf. on Concurrency Theory (CONCUR'10)*. https://doi.org/10.1007/978-3-642-15375-4_16

[11] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear Logic Propositions as Session Types. *Math. Struct. Comp. Sci.* 26, 3 (March 2016). https://doi.org/10.1017/S0960129514000218

[12] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *J. Statistical Softw.* 76, 1 (2017). https://doi.org/10.18637/jss.v076.i01

[13] Simon Castellan and Hugo Paquet. 2019. Probabilistic Programming Inference via Intensional Semantics. In *European Symp. on Programming (ESOP'19)*. https://doi.org/10.1007/978-3-030-17184-1_12

[14] J. T. Chang and D. Pollard. 1997. Conditioning as disintegration. *Netherlands Society for Statistics and Operations Research* 51, 3 (November 1997). https://doi.org/10.1111/1467-9574.00056

[15] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *Formal Methods for Eternal Networked Software Systems (SFM'15)*. https://doi.org/10.1007/978-3-319-18941-3_4

[16] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-Purpose Probabilistic Programming System with Programmable Inference. In *Prog. Lang. Design and Impl. (PLDI'19)*. https://doi.org/10.1145/3314221.3314642

[17] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. 2020. Nested Session Types. https://arxiv.org/abs/2010.06482

[18] Adam Foster, Martin Jankowiak, Eli Bingham, Paul Horsfall, Yee Whye Teh, Tom Rainforth, and Noah D. Goodman. 2019. Variational Bayesian Optimal Experimental Design: Efficient Automation of Adaptive Experiments. In *Neural Info. Processing Syst. (NIPS'19)*. https://arxiv.org/abs/1903.05480

[19] Rong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A Language for Flexible Probabilistic Inference. In *Artificial Intelligence and Statistics (AISTATS'18)*.

[20] Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2013. *Bayesian Data Analysis*. Chapman and Hall/CRC. https://doi.org/10.1201/b16018

[21] Zoubin Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nature* 521 (May 2015). https://doi.org/10.1038/nature14541

[22] W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. 1994. A Language and Program for Complex Bayesian Modelling. *J. Royal Statistical Society* 43, 1 (January 1994). https://doi.org/10.2307/2348941

[23] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith A. Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A language for generative models. In *Uncertainty in Artificial Intelligence (UAI'08)*. https://dl.acm.org/doi/10.5555/3023476.3023503

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

[25] Thomas L. Griffiths, Charles Kemp, and Joshua B. Tenenbaum. 2008. Bayesian Models of Cognition. In *The Cambridge Handbook of Computational Psychology*. Cambridge University Press. https://doi.org/10.1017/CBO9780511816772.006

[26] Robert Harper. 2016. *Practical Foundations for Programming Languages*. Cambridge University Press. https://dl.acm.org/doi/book/10.5555/3002812

[27] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (August 1978). https://doi.org/10.1145/359576.359585

[28] Kohei Honda. 1993. Types for Dyadic Interaction. In *Int. Conf. on Concurrency Theory (CONCUR'93)*. https://doi.org/10.1007/3-540-57208-2_35

[29] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *European Symp. on Programming (ESOP'98)*. https://doi.org/10.1007/BFb0053567

[30] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Princ. of Prog. Lang. (POPL'08)*. https://doi.org/10.1145/1328438.1328472

[31] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov Chain Monte Carlo Algorithms for Probabilistic Modeling. In *Prog. Lang. Design and Impl. (PLDI'17)*. https://doi.org/10.1145/3062341.3062375

[32] Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. 2015. A Provably Correct Sampler for Probabilistic Programs. In *Leibniz International Proceedings in Informatics (LIPIcs'15)*. https://doi.org/10.4230/LIPIcs.FSTTCS.2015.475

[33] F. Jelinek, J. D. Lafferty, and R. L. Mercer. 1992. Basic Methods of Probabilistic Context Free Grammars. In *Speech Recognition and Understanding*. https://doi.org/10.1007/978-3-642-76626-8_35

[34] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley. https://dl.acm.org/doi/book/10.5555/270146

[35] Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (June 1981). https://doi.org/10.1016/0022-0000(81)90036-2

[36] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2019. Towards Verified Stochastic Variational Inference for Probabilistic Programs. *Proc. ACM Program. Lang.* 4, POPL (December 2019). https://doi.org/10.1145/3371084

[37] Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. 2019. Trace Types and Denotational Semantics for Sound Programmable Inference in Probabilistic Languages. *Proc. ACM Program. Lang.* 4, POPL (December 2019). https://doi.org/10.1145/3371087

[38] Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin C. Rinard. 2018. Probabilistic Programming with Programmable Inference. In *Prog. Lang. Design and Impl. (PLDI'18)*. https://doi.org/10.1145/3296979.3192409

[39] Robin Milner. 1989. *Communication and Concurrency*. Prentice-Hall, Inc. https://dl.acm.org/doi/book/10.5555/534666

[40] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Information and Computation* 100, 1 (September 1992). https://doi.org/10.1016/0890-5401(92)90008-4

[41] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, II. *Information and Computation* 100, 1 (September 1992). https://doi.org/10.1016/0890-5401(92)90009-5

[42] Eugenio Moggi. 1989. Computational lambda-calculus and monads. In *Logic in Computer Science (LICS'89)*. https://doi.org/10.1109/LICS.1989.39155

[43] Lawrence M. Murray. 2015. Bayesian State-Space Modelling on High-Performance Hardware Using LibBi. *J. Statistical Softw.* 67, 10 (2015). https://doi.org/10.18637/jss.v067.i10

[44] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *Int. Symp. on Functional and Logic Programming (FLOPS'16)*. https://doi.org/10.1007/978-3-319-29604-3_5

[45] Martyn Plummer. 2003. JAGS: A Program for Analysis of Bayesian Graphical Models using Gibbs Sampling. In *Int. Workshop on Distributed Statistical Comp. (DSC'03)*.

[46] Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. 2019. Bayesian Synthesis of Probabilistic Programs for Automatic Data Modeling. *Proc. ACM Program. Lang.* 3, POPL (January 2019). https://doi.org/10.1145/3290350

[47] Alceste Scalas and Nobuko Yoshida. 2019. Less Is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.* 3, POPL (January 2019). https://doi.org/10.1145/3290343

[48] Adam Ścibior, Zoubin Ghahramani, and Andrew D. Gordon. 2015. Practical Probabilistic Programming with Monads. In *Symp. on Haskell (Haskell'15)*. https://doi.org/10.1145/2887747.2804317

[49] Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2017. Denotational Validation of Higher-Order Bayesian Inference. *Proc. ACM Program. Lang.* 2, POPL (December 2017). https://doi.org/10.1145/3158148

[50] Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. In *Princ. of Prog. Lang. (POPL'17)*. https://doi.org/10.1145/3009837.3009852

[51] Peter Thiemann and Vasco T. Vasconcelos. 2016. Context-Free Session Types. In *Int. Conf. on Functional Programming (ICFP'16)*. https://doi.org/10.1145/2951913.2951926

[52] Peter Thiemann and Vasco T. Vasconcelos. 2019. Label-Dependent Session Types. *Proc. ACM Program. Lang.* 4, POPL (December 2019). https://doi.org/10.1145/3371135

[53] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *Int. Conf. on Learning Representations (ICLR'17)*.

[54] Philip Wadler. 2012. Propositions as Sessions. In *Int. Conf. on Functional Programming (ICFP'12)*. https://doi.org/10.1145/2364527.2364568

[55] Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Sound Probabilistic Inference via Guide Types. https://arxiv.org/abs/2104.03598

[56] Stefan Webb, Adam Golinski, Robert Zinkov, N. Siddharth, Tom Rainforth, Yee Whye Teh, and Frank Wood. 2018. Faithful Inversion of Generative Models for Effective Amortized Inference. In *Neural Info. Processing Syst. (NIPS'18)*. https://dl.acm.org/doi/10.5555/3327144.3327229

[57] Website. 2020. greenlet: Lightweight concurrent programming. Available on https://greenlet.readthedocs.io.

[58] Frank Wood, Jan Willem van de Meent, and Vikash K. Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Artificial Intelligence and Statistics (AISTATS'14)*.

[59] Robert Zinkov and Chung-chieh Shan. 2017. Composing Inference Algorithms as Program Transformations. In *Uncertainty in Artificial Intelligence (UAI'17)*. https://arxiv.org/abs/1603.01882