Programmable MCMC with Soundly Composed Guide Programs

LONG PHAM, Carnegie Mellon University, USA
DI WANG*, Peking University, China
FERAS A. SAAD, Carnegie Mellon University, USA
JAN HOFFMANN, Carnegie Mellon University, USA

Probabilistic programming languages (PPLs) provide language support for expressing flexible probabilistic models and solving Bayesian inference problems. PPLs with *programmable inference* make it possible for users to obtain improved results by customizing inference engines using *guide* programs that are tailored to a corresponding *model* program. However, errors in guide programs can compromise the statistical soundness of the inference. This article introduces a novel coroutine-based framework for verifying the correctness of user-written guide programs for a broad class of Markov chain Monte Carlo (MCMC) inference algorithms. Our approach rests on a novel type system for describing communication protocols between a model program and a sequence of guides that each update only a subset of random variables. We prove that, by translating guide types to context-free processes with finite norms, it is possible to check structural type equality between models and guides in polynomial time. This connection gives rise to an efficient *type-inference algorithm* for probabilistic programs with flexible constructs such as general recursion and branching. We also contribute a *coverage-checking algorithm* that verifies the support of sequentially composed guide programs agrees with that of the model program, which is a key soundness condition for MCMC inference with multiple guides. Evaluations on diverse benchmarks show that our type-inference and coverage-checking algorithms efficiently infer types and detect sound and unsound guides for programs that existing static analyses cannot handle.

CCS Concepts: • **Theory of computation** → **Probabilistic computation**; **Type theory**; *Grammars and context-free languages*; • **Mathematics of computing** → **Bayesian computation**.

Additional Key Words and Phrases: probabilistic programming, Bayesian inference, type systems, coroutines, context-free types

ACM Reference Format:
Long Pham, Di Wang, Feras A. Saad, and Jan Hoffmann. 2024. Programmable MCMC with Soundly Composed Guide Programs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 308 (October 2024), 30 pages. https://doi.org/10.1145/3689748

1 Introduction

Probabilistic programming languages (PPLs) enable users to write probabilistic models as programs and solve Bayesian-inference problems. PPLs have been successfully used in numerous applications, ranging from robotics [38] and computer vision [28] to cognition [7] and data science [42].

*Corresponding author.

Authors' Contact Information: Long Pham, Carnegie Mellon University, Pittsburgh, USA, longp@andrew.cmu.edu; Di Wang, Peking University, Beijing, China, wangdi95@pku.edu.cn; Feras A. Saad, Carnegie Mellon University, Pittsburgh, USA, fsaad@cmu.edu; Jan Hoffmann, Carnegie Mellon University, Pittsburgh, USA, jhoffmann@cmu.edu.

ACM 2475-1421/2024/10-ART308
https://doi.org/10.1145/3689748

Programmable inference. Traditional PPLs provide generic inference algorithms that apply to almost any model that can be expressed in the languages [6, 18, 53]. However, these inference algorithms may fail to return accurate results within a reasonable time frame. To circumvent this problem, modern PPLs support programmable inference, which lets users develop custom *guide programs* that are customized to the *model programs* [3, 12, 33]. Custom guide programs are supported by both variational and Monte-Carlo-based inference algorithms, enabling substantial improvements in accuracy and runtime performance as compared to generic algorithms [11]. However, they also create room for users to introduce bugs that invalidate the statistical soundness of the inference, causing the inference algorithms to crash or even silently return invalid results.

Verifying guide programs. A number of static-analysis methods have been recently developed to verify the correctness of user-implemented guide programs. At a high level, guide programs have to satisfy certain *compatibility* conditions with respect to model programs. Lee et al. [30] propose a static analysis that checks if a model-guide pair is compatible for variational inference in Pyro [3]. Lew et al. [31] develop a type system for traces of probabilistic programs to ensure that well-typed model-guide pairs are compatible for both Monte Carlo and variational inference. A limitation of these approaches is their lack of support for general conditional statements and recursive procedure calls. Li et al. [32] overcome the limitation for variational inference by extending trace types. Another approach is using *coroutine*-based programmable-inference [52], where model and guide programs are treated as coroutines that communicate by exchanging messages about branching and recursion. Communication protocols are automatically inferred and imposed via *guide types*.

In this article, we consider the problem of statically verifying the soundness of *Markov-Chain Monte Carlo* (MCMC) inference algorithms, and in particular the *multiple-block Metropolis-Hastings* [BMH; 8, §4.4] algorithm. The well-known Gibbs sampling and Metropolis-within-Gibbs algorithms are special cases of BMH [17]. MCMC, including BMH, simulates a Markov chain whose transition kernel is specified by one or more guide programs. MCMC repeatedly draws samples from these guide programs, which form successively better approximations of the posterior distribution of a model program. As the number of iterations becomes large, the samples from the Markov chain resemble samples from the target distribution.

Model-guide compatibility. A BMH sampler is said to be sound if the limiting distribution of the Markov chain is the target posterior distribution. Informally, a sufficient condition for the soundness of BMH is that a sequential composition of guide programs should be able to propose any sample in the support of a model program. If this condition does not hold, the Markov chain has a risk of never proposing a sample in the support of a model program. For example, suppose a model program draws a sample from a Normal distribution $\textsc{Normal}(0, 1)$, which has full support over $\mathbb{R}$. If a guide program draws a sample from a Gamma distribution $\textsc{Gamma}(1, 1)$, whose support is $\mathbb{R}_{>0}$, then the Markov chain induced by this guide program cannot propose negative values. Hence, the Markov chain cannot faithfully converge to the target distribution. Checking the compatibility of model and guide programs in BMH is especially challenging because it requires reasoning about the sequential composition of multiple guide programs, where each guide may propose a different subset of random variables and may use random control flow, recursion, and other flexible programming constructs.

This work. To verify the soundness of BMH algorithms, this article extends the coroutine-based programmable inference of Wang et al. [52] from handling only a single-guide program to handling the sequential composition of multiple guide programs. We build our framework on *trace-based* probabilistic inference programming [33], where a probabilistic program defines a distribution over execution traces that record samples for random variables. A guide program can also access (and

reuse) the execution trace of the previous guide program, and the BMH algorithm sequentially executes the guides to propose a new trace from the current one. We reduce the model-guide compatibility check to the following verification task: *given any initial trace, can the sequential composition of guides propose every possible new trace with a non-zero probability?* A major challenge is to augment the model-guide communication with a third party: a guide program now can communicate with both the model and the previous guide. We formulate a novel operational semantics for sequentially composed guides that is capable of monitoring and aligning the control flows of previous and current guides. Our semantics deals with the issue that the guides' control flows may diverge.

We then adapt guide types and automatic type inference from Wang et al. [52] to our new semantics. There are two challenges: (i) different guides may have different control-flow structures as long as their types are structurally equal (whereas the guide-type system in Wang et al. [52] only supports nominal types); (ii) a guide may sample a subset of random variables (whereas Wang et al. [52] only consider complete samples). For challenge (i), we develop a type-equality checking algorithm for guide types with structural equality. In our setting, guide types correspond to context-free types [47], which have *infinite* state spaces. By translating guide types to context-free processes with finite norms, whose bisimilarity is decidable in polynomial time [21], we prove that guide-type equality is decidable in polynomial time. For challenge (ii), we devise a *coverage*-checking algorithm for verifying that sequentially composed guides satisfy the compatibility condition that "the composition *covers* all possible sample traces in the model." We reduce coverage checking to verifying that every random variable in any control-flow path is freshly sampled by at least one guide. Our coverage-checking algorithm essentially *bisimulates* guide types alongside structures of guide programs.

We have implemented type-inference, type-equality-checking, and coverage-checking algorithms. An empirical evaluation of our system on a diverse benchmark set shows that the type-inference algorithm is more expressive than the algorithm from Wang et al. [52] and that the coverage-checking algorithm can efficiently handle many benchmarks in practice.

Contributions. This article makes the following contributions:

- We present a flexible coroutine-based framework for programmable inference with sequentially composed guides that can access and reuse previous traces (§3). Our system handles expressive constructs such as conditional branching and general recursion in both models and guides.
- We prove that—by translating guide types to context-free processes with finite norms—structural-type-equality checking in our framework is decidable in polynomial time (§4 and Theorem 4.7). This connection enables more expressive automatic type inference while remaining efficient.
- We present a novel coverage-checking algorithm (§5) for verifying that sequentially composed guide programs have full coverage over the support of the target model program; along with a proof that our algorithm is sound (Theorem 5.1).
- We implement and evaluate type-equality and coverage-checking algorithms on a diverse benchmark set (§6), showing that our system (i) can analyze programs beyond the reach of previous static analyses; and (ii) efficiently identifies both correct and incorrect guide programs.

2 Overview

2.1 Bayesian Inference, Markov-Chain Monte Carlo, and Block Metropolis-Hastings

Bayesian inference is the problem of conditioning a probabilistic model on *observed data* and computing (or approximating) a posterior distribution on *latent variables*, which encode information

about the "ground truth" that cannot be observed directly. Probabilistic programming [2, 19] provides a framework for implementing probabilistic models and performing Bayesian inference.

Markov-Chain Monte Carlo (MCMC) is a family of algorithms that generate a sequence $\{lat_i\}_{i=1,\ldots,T}$ of correlated samples of latent variables from a suitable Markov chain whose stationary distribution is the target posterior. MCMC uses *kernels* to generate a new state lat_i from the previous state lat_{i-1}. The *Metropolis-Hastings* (MH) algorithm [20, 34] is a generic method to construct kernels via custom *proposal distributions* (called *guide programs* in probabilistic programming), which generate new values for latent variables. In each iteration, MH computes an *acceptance ratio* for a proposed state and then accepts it with a probability equal to the ratio.

The program *Model* in Figure 1a describes a probabilistic model on random variables specified by commands $\mathbf{sample}(@\ell, d)$, where ℓ is a *label* that uniquely identifies a random variable and d is a *primitive distribution*, such as Cat (categorical) distributions whose support is the integer ring $\mathbb{N}_k$ (where k is the number of categories), Normal distributions whose support is the real line $\mathbb{R}$, and InvGamma (inverse-gamma) distributions whose support is the positive real line $\mathbb{R}_+$. The program specifies a regression model with univariate polynomials with degree at most two. Figure 1b plots 50 randomly generated polynomials. Figure 1d implements a proposal distribution for this model as a guide program $Guide_1$. The program takes the previous *sample trace*—which records the values of latent variables from the previous iteration—as its input and generates a new trace that is compatible with the regression model. By "compatible," we mean (informally) that this guide program generates latent variables from a distribution with the same support as the model. This program implements a *single-block* MH proposal in the sense that it generates new values for latent variables jointly as one block. The left of Figure 1c plots the last 50 posterior samples from this run.

In a high-dimensional space of latent variables, using a single proposal can suffer from low acceptance rates during MCMC sampling, which leads to slow convergence. A run of MH using the single-block proposal in Figure 1d for 5,000 iterations resulted in a poor acceptance rate of only 2.3%. Figure 1f shows three trace plots for three latent variables ($@c_0$, $@c_1$, and $@c_2$) from the 5,000 samples, where the red lines plot the ground-truth values for them. We can see from the plots that this particular run was inefficient in exploring the posterior and did not seem to mix at all.

Multiple-block MH. A generalization of single-block MH is *multiple-block Metropolis-Hastings* (BMH), also known as Metropolis-within-Gibbs [17]. Algorithm 1 shows a simplified case of BMH where the target distribution $\pi(x)$ is defined over a fixed-dimensional space $\mathbb{R}^d$. The latent variables are partitioned into $B \geq 1$ blocks $(x_1, \ldots, x_B)$, where each $x_b \in \mathbb{R}^{n_b}$ and $n_1 + \cdots + n_B = d$. At each iteration, BMH updates a subset (*block*) of variables x_b by sampling from a proposal distribution q_b $(b = 1, \ldots, B)$. BMH makes more local steps in each iteration as compared to single-block MH and often obtains higher acceptance rates. The well-known *(block) Gibbs sampling* algorithm is a

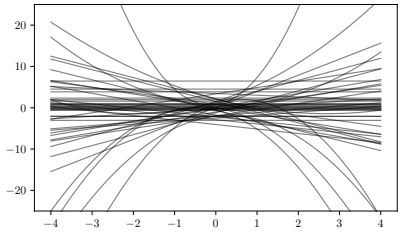
Algorithm 1 Multiple-Block Metropolis-Hastings (BMH)

Require: target distribution $\pi(x_1, \ldots, x_B)$; proposal distributions $(q_1, \ldots, q_B)$.

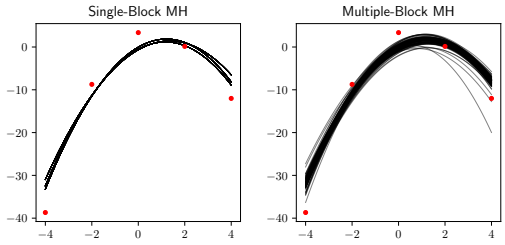
1: Initialize $x^0 \leftarrow (x_1^0, \ldots, x_B^0)$.
2: **for** $j = 1, 2, \ldots$ **do**
3: $\quad x^j \leftarrow (x_1^{j-1}, \ldots, x_B^{j-1})$
4: $\quad$ **for** $b = 1, \ldots, B$ **do**
5: $\quad\quad$ Propose a new value $\hat{x}_b \sim q_b(-; x^j)$ for block b.
6: $\quad\quad$ Compute the acceptance ratio $\alpha \leftarrow \dfrac{\pi(\hat{x}_b, x^j_{-b})}{\pi(x^j)} \dfrac{q_b(x_b^{j-1}; x^j_{-b}, \hat{x}_b)}{q_b(\hat{x}_b; x^j)}$.
7: $\quad\quad$ Update $x_b^j \leftarrow \hat{x}_b$ with probability $\min(1, \alpha)$.

```
1  proc Model(xs : vec[5](ℝ)) =
2    degree ← sample(@d, Cat(0.3; 0.5; 0.2));
3    c0 ← sample(@c0, Normal(0, 2));
4    f ← (
5      if degree = 0 then
6        return(λx. c0)
7      else
8        c1 ← sample(@c1, Normal(0, 2));
9        if degree = 1 then
10         return(λx. c0 + c1 ∗ x)
11       else
12         c2 ← sample(@c2, Normal(0, 2));
13         return(λx. c0 + c1 ∗ x + c2 ∗ x ∗ x)
14   );
15   noise2 ← sample(@n, InvGamma(1, 1));
16   noise ← return(sqrt(noise2));
17   ys ← foreach (i, x) in xs (
18     y ← sample(@yi, Normal(f(x), noise));
19     return(y)
20   );
21   return(ys)
```

(a) Probabilistic program *Model* over curves.



(b) 50 prior curves drawn randomly from *Model*.



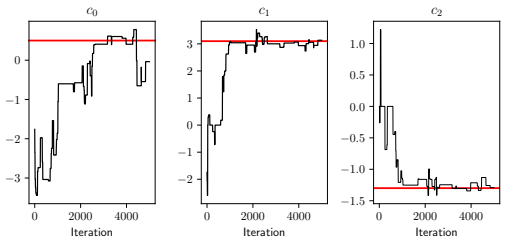
(c) 50 posterior curves given data.

```
1  proc Guide1(σ : trace) =
2    degree ← sample(@d, Cat(1/3; 1/3; 1/3));
3    c0 ← sample(@c0, Normal(σ[@c0], 0.5));
4    _ ← (
5      if degree = 0 then
6        return()
7      else
8        c1 ← sample(@c1, Normal(σ[@c1] or 0, 0.5));
9        if degree = 1 then
10         return()
11       else
12         c2 ← sample(@c2, Normal(σ[@c2] or 0, 0.5));
13         return()
14   );
15   noise2 ← sample(@n, InvGamma(1, 1));
16   return()
```

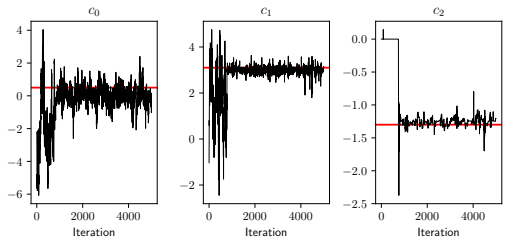
(d) Proposal program $Guide_1$ for Single-Block MH.

```
1  proc Guide2,d(σ : trace) =
2    degree ← sample(@d, Cat(2/5; 119/200; 1/200));
3    if degree = 0 then return() else
4      c1 ← (if degree ≤ σ[@d] then return(σ[@c1])
5        else sample(@c1, Normal(0, 0.5)));
6      if degree = 1 then return() else
7        c2 ← (if degree ≤ σ[@d] then return(σ[@c2])
8          else sample(@c2, Normal(0, 0.5)));
9        return()
10 proc Guide2,ci(σ : trace) =        (for i = 0, 1, 2)
11   ci ← (if σ[@d] < i then return(0)
12     else sample(@ci, Normal(σ[@ci], 0.5)));
13   return()
14 proc Guide2,n(σ : trace) =
15   noise2 ← sample(@n, InvGamma(1, 1));
16   return()
```

(e) Proposal programs $Guide_{2,*}$ for Multiple-Block MH.



(f) Trace plots for $@c_0$, $@c_1$, $@c_2$ (Single-Block MH).



(g) Trace plots for $@c_0$, $@c_1$, $@c_2$ (Multiple-Block MH).

Fig. 1. Bayesian inference for a regression model over polynomial curves of order up to 2.

special case of BMH, where the proposal distribution for a block of latent variables is its conditional distribution given the observed data and latent variables in all other blocks.

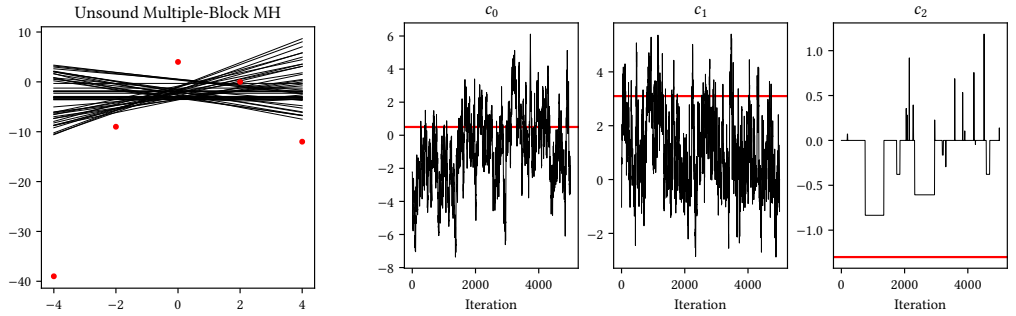
Figure 1e demonstrates a sequence of guide programs, each of which implements a block proposal distribution q_b for the regression model. The proposal $\mathit{Guide}_{2,\ell}$ with a subscript ℓ is intended to mutate the value of the random variable $@\ell$. We sequentially compose these proposals—each of which is followed by an MH acceptance routine—to obtain an MCMC kernel. Similar to the single-block proposal, these proposals must be *compatible* with the model; that is, after each guide program mutates a block of random variables, the mutated trace is valid with respect to the model. The proposal $\mathit{Guide}_{2,d}$ is intended to mutate $@d$, which is the degree of the regression polynomial, but it needs to take care of missing coefficients (see lines 5 and 8). Note that we deliberately implement $\mathit{Guide}_{2,d}$ to sample $@d$ from a "bad" distribution $\textsc{Cat}(2/5, 119/200, 1/200)$, which leads the inference to explore quadratic functions with a very small probability.

Figure 1g shows the trace plots for the random variables $@c_0$, $@c_1$, and $@c_2$ from a run of 5,000 iterations of the composition of the block proposals. Compared with Figure 1f, BMH is much more efficient in exploring the sample space: the acceptance rate is about 38.6%. The trace plots for all three coefficients indicate that the run mixes well. We plot the last 50 samples of this BMH in the right of Figure 1c. These curves capture uncertainty better and present more diverse samples than the single-block MH run. Note that though we use a "bad" proposal for $@d$, BMH is robust enough to converge after the first few hundreds of iterations that do not explore quadratic functions at all.

A number of case studies in the literature of PPLs demonstrate the benefit of BMH, where each constituent proposal mutates a different block of random variables. For example, Chib and Greenberg [9, §7.2] describe BMH involving two distinct block proposals to compute a posterior distribution of a stationary second-order autoregressive time-series model. More recent examples include discovering models (encoded as probabilistic context-free grammars) for time-series data by Mansinghka et al. [33, §3.1] and Cusumano-Towner et al. [12, §7.2] and linear regression with outlier detection by Mansinghka et al. [33, §3.2] and Cusumano-Towner et al. [12, §3.2].

Sound and unsound guides. In order for BMH to be sound (i.e., it defines a Markov chain that converges to the conditional distribution of a model given observed data), the sequential composition of guide programs in BMH must be compatible with the model program. More concretely, every set of positive-probability traces under the target distribution should have positive probability under the distribution defined by a sequential composition of guide programs [48, Theorem 1]. If this compatibility condition is not satisfied, then BMH may fail to explore positive-probability regions in the target distribution.

To illustrate unsound guide programs, consider Guide_{2,c_1} from Figure 1e. Suppose we modify the expression $\textsc{Normal}(\sigma[@c_1], 0.5)$ in line 12 by replacing the random variable $@c_1$ with $@c_2$. This change could easily result in a runtime error, because the random variable $@c_2$ is not guaranteed to exist in the previous trace. A more subtle example of unsound BMH is obtained by removing Guide_{2,c_2} from the sequential composition of guide programs. Then the random variable $@c_2$ is never resampled, unless $\mathit{Guide}_{2,d}$ increases the polynomial degree from 1 to 2. Likewise, if we replace the expression $\textsc{Normal}(\sigma[@c_2], 0.5)$ in line 12 with a Gamma distribution, whose support is $\mathbb{R}_{>0}$ rather than $\mathbb{R}$, the modified guide is unsound. This is because, if the preceding guide program $\mathit{Guide}_{2,d}$ keeps the random variable $@d$ unchanged, the resulting Markov chain cannot sample a negative value for the random variable $@c_2$, yielding a mismatch with the set of traces admitted by the model program *Model*. Figure 2 displays the Bayesian inference result of the unsound sequential composition of guide programs, where the Normal distribution in Guide_{2,c_2} has been replaced with a Gamma distribution. The posterior samples in Figure 2a fit poorly with the observed data (red points) as compared to the samples from sound BMH in Figure 1c, reflecting a failure of convergence



(a) 50 posterior curves. (b) Trace plots of the three polynomial coefficients.

Fig. 2. Results using an unsound BMH guide program for the inference problem in Figure 1.

to the targe distribution. In addition, the trace plot of the random variable @c_2 in Figure 2b indicates that the unsound BMH does not converge to the ground-truth value (denoted by the red horizontal line).

Programming BMH proposals is more difficult than programming single-block MH ones. To ensure the model-guide compatibility, each block-proposal guide needs to take care of the change in the model's control flow that might lead to different sets of random variables. The next sections discuss how our new framework achieves sound BMH via *coverage-annotated guide types*.

2.2 Programmable Block MH via Guide-Typed Coroutines

Guide-typed coroutines. We adapt a *coroutine*-based paradigm for implementing models and guides from Wang et al.'s work, which supports sound programmable single-block MH. The idea is to treat the model and guide as two communicating coroutines: the model determines the control flow (which influences the set of latent variables), so it sends *branching* information to the guide; meanwhile, the guide determines proposals for latent variables, so it sends *sampling* information to the model. Such message-passing communication can be easily realized through coroutines connected by bidirectional *channels*. Figure 3b reimplements the model shown in Figure 1a by making the communication explicit: the sampling (**sample**(...)) and branching (**if** ...) commands are annotated with rv (resp., sd) to indicate receiving (resp., sending) information, as well as the name of a channel on which the communication takes place. The model consumes a *lat* channel for communication with the guide, and provides an *obs* channel for identifying observed data.

Wang et al. [52] proposed *guide types* to enforce that the model and guide follow a communication protocol, which describes the support of the model distribution. The type **1** specifies an ended channel. The type $\tau \wedge A$ means the channel provider draws and sends a random sample of type τ, and proceeds with a type-A protocol. The *obs* channel is given a guide type $\mathsf{Obs} \coloneqq \mathbb{R} \wedge \mathbb{R} \wedge \mathbb{R} \wedge \mathbb{R} \wedge \mathbb{R} \wedge \mathbf{1}$. The type $A \,\&\, B$ means the channel provider receives a branch selection and proceeds with a type-A or B protocol accordingly. Figure 3a defines a guide-type operator $\mathsf{Coeffs}[\cdot]$ that corresponds to the communication carried out from lines 7 to 15 of Figure 3b. The type operator is parameterized by a *continuation* type that specifies the communication after the protocol described by the operator. The *lat* channel is given a guide type $\mathsf{Lat} \coloneqq \mathbb{N}_3 \wedge \mathbb{R} \wedge \mathsf{Coeffs}[\mathbb{R}_+ \wedge \mathbf{1}]$. We instantiate Coeffs with $\mathbb{R}_+ \wedge \mathbf{1}$ because the model samples @n—whose type is $\mathbb{R}_+$—after it samples the coefficients.

Figure 3c provides a template to implement MH proposals as guide coroutines. Ignoring the code with a yellow background, the template yields a reimplementation of the single-block MH proposal shown in Figure 1d. The compatibility is justified by the fact that the *Guide* coroutine provides the *lat* channel whose guide type is Lat, which is the same as *Model*'s signature. Dual to the model coroutine,

$$\mathsf{Lat} \stackrel{\text{def}}{=} \mathbb{N}_3 \wedge \mathbb{R} \wedge \mathsf{Coeffs}[\mathbb{R}_+ \wedge \mathbf{1}]$$
$$\mathsf{Coeffs}[X] \stackrel{\text{def}}{=} X \,\&\, (\mathbb{R} \wedge (X \,\&\, (\mathbb{R} \wedge X)))$$
$$\mathsf{Obs} \stackrel{\text{def}}{=} \mathbb{R} \wedge \mathbb{R} \wedge \mathbb{R} \wedge \mathbb{R} \wedge \mathbb{R} \wedge \mathbf{1}$$
$$\mathsf{OLat} \stackrel{\text{def}}{=} \mathbb{N}_3 \wedge \mathbb{R} \wedge \mathsf{OCoeffs}[\mathbb{R}_+ \wedge \mathbf{1}]$$
$$\mathsf{OCoeffs}[X] \stackrel{\text{def}}{=} X \oplus (\mathbb{R} \wedge (X \oplus (\mathbb{R} \wedge X)))$$

(a) Definitions of type operators.

```
1  proc Model(xs : vec[5](ℝ))
2  consume lat :: Lat
3  provide obs :: Obs =
4   degree ← sample_rv{lat}(Cat(0.3; 0.5; 0.2));
5   c0 ← sample_rv{lat}(Normal(0, 2));
6   f ← (
7    if_sd{lat} degree = 0 then
8     return(λx. c0)
9    else
10    c1 ← sample_rv{lat}(Normal(0, 2));
11    if_sd{lat} degree = 1 then
12     return(λx. c0 + c1 ∗ x)
13    else
14     c2 ← sample_rv{lat}(Normal(0, 2));
15     return(λx. c0 + c1 ∗ x + c2 ∗ x ∗ x)
16  );
17  noise2 ← sample_rv{lat}(InvGamma(1, 1));
18  noise ← return(sqrt(noise2));
19  ys ← foreach (i, x) in xs (
20   y ← sample_sd{obs}(Normal(f(x), noise));
21   return(y)
22  );
23  return(ys)
```

(b) The model coroutine.

```
1  proc Guide(σ : trace)
2  consume old :: OLat
3  provide lat :: Lat =
4   old_d ← oldsample{old}(); degree ← sample_sd{lat}(□1);
5   old_c0 ← oldsample{old}(); c0 ← sample_sd{lat}(□2);
6   f ← (
7    if_rv{lat} ⋆ then
8     oldif_rv{old} same then return() else return()
9    else
10    oldif_rv{old} same then
11     old_c1 ← oldsample{old}(); c1 ← sample_sd{lat}(□3);
12     if_rv{lat} ⋆ then
13      oldif_rv{old} same then return() else return()
14     else
15      oldif_rv{old} same then
16       old_c2 ← oldsample{old}(); c2 ← sample_sd{lat}(□4);
17       return()
18      else
19       c2 ← sample_sd{lat}(□6);
20       return()
21     else
22      c1 ← sample_sd{lat}(□7);
23      if_rv{lat} ⋆ then
24       return()
25      else
26       c2 ← sample_sd{lat}(□8);
27       return()
28  );
29  old_n ← oldsample{old}(); noise2 ← sample_sd{lat}(□5);
30  return()
```

(c) A template of guide coroutines.

Fig. 3. Guide-typed coroutines for the regression model and MH proposals.

the guide samples and sends random values on the *lat* channel, and receives branch selections from the same channel (see lines 7 and 12). The ⋆ symbol serves as a placeholder and it indicates that the branch selection is sent by the consumer of the *lat* channel, i.e., the model coroutine. We instantiate the boxes $\square_i$ for $i \in \{1, \ldots, 5\}$ as follows: $\square_1 = \textsc{Cat}(1/3, 1/3, 1/3), \square_2 = \textsc{Normal}(\sigma[@c_0], 0.5), \square_3 = \textsc{Normal}(\sigma[@c_1]$ **or** $0, 0.5), \square_4 = \textsc{Normal}(\sigma[@c_2]$ **or** $0, 0.5), \square_5 = \textsc{InvGamma}(1, 1)$.

Towards multiple-block MH. To support BMH proposals, a natural approach would be to introduce point distributions, e.g., $\textsc{Delta}(v)$ whose support is $\{v\}$, and refine the guide-type system to deal with such distributions. Using this construct, single-site proposals m_x and m_y for random variables $@x$ and $@y$, respectively, could be expressed as follows (where σ denotes the previous trace):

$$m_x \stackrel{\text{def}}{=} \_ \leftarrow \mathbf{sample}_{\text{sd}}\{lat\}(\textsc{Normal}(\sigma[@x], 0.5)); \_ \leftarrow \mathbf{sample}_{\text{sd}}\{lat\}(\textsc{Delta}(\sigma[@y])); \mathbf{return}()$$
$$m_y \stackrel{\text{def}}{=} \_ \leftarrow \mathbf{sample}_{\text{sd}}\{lat\}(\textsc{Delta}(\sigma[@x])); \_ \leftarrow \mathbf{sample}_{\text{sd}}\{lat\}(\textsc{Normal}(\sigma[@y], 0.5))); \mathbf{return}()$$

For a target distribution with full support over $\mathbb{R}^2$, the sequential composition m_x and m_y yields a sound kernel because it also has full support over $\mathbb{R}^2$. Unfortunately, there are fundamental challenges with designing a type system that can reason about arbitrary user-specified delta distributions. Consider changing the second command of m_x to instead be $\mathbf{sample}_{\text{sd}}\{lat\}(\textsc{Delta}(42))$. Clearly, the single-site update m_x is no longer sound, because every move for $@x$ would be rejected

(except when the previous trace σ satisfies $\sigma[@y] = 42$, which has probability zero under the target distribution). To correctly reason about the model-guide compatibility of BMH in the presence of general point distributions $\textsc{Delta}(e)$, the type system would therefore need to analyze the expressions e and distinguish between cases such as $\textsc{Delta}(\sigma[@y])$ and $\textsc{Delta}(42)$. This approach is as hard as checking for the semantic equivalence of two expressions, and also requires finding the locations of all point masses (if any) in the target distribution.

BMH guides as coroutines. The previous example suggests that our system should properly *align* the previous trace within a block proposal coroutine and add a command for "keeping the value of a random variable unchanged," which is a restricted type of point distribution. To deal with alignment, we grant BMH guide coroutines the access to another *read-only* channel, e.g., *old*, that records the messages exchanged between the model and a previous guide coroutine. To support this "keeping unchanged" behavior, we add two kinds of commands: one for retrieving an old sample from the previous trace, written $\textbf{oldsample}\{old\}()$, the other for forwarding an unchanged sample to the model, written $\textbf{sample}_{\mathsf{sd}}\{lat\}(\textsc{Keep})$. Meanwhile, the alignment of branching is nontrivial: the control flow of the model with respect to the previous guide could diverge from the model's flow with respect to the current guide. In our system, we deal with branch alignment by imposing the following structure:

$$\begin{array}{rl} \textbf{if}_{\mathsf{rv}}\{lat\} \star \textbf{then} & \textbf{oldif}_{\mathsf{rv}}\{old\}\ \underline{\mathsf{same}}\ \textbf{then}\ m_{\mathsf{true,true}}\ \textbf{else}\ m_{\mathsf{true,false}} \\ \textbf{else} & \textbf{oldif}_{\mathsf{rv}}\{old\}\ \underline{\mathsf{same}}\ \textbf{then}\ m_{\mathsf{false,false}}\ \textbf{else}\ m_{\mathsf{false,true}} \end{array}$$

We introduce the $\textbf{oldif}_{\mathsf{rv}}\{old\}\ \underline{\mathsf{same}}\ \ldots$ command to read a branch selection from the *old* channel. Such a structure identifies four branches m_{b_1,b_2} with $b_1, b_2 \in \{\mathsf{true}, \mathsf{false}\}$, where b_1 is the branch selection received from the model, and b_2 is the one read from the previous trace. When $b_1 \neq b_2$, the command m_{b_1,b_2} cannot access the previous trace, because the control flow diverges.

Ignoring the code with a red background, Figure 3c can be used to reimplement the block guides shown in Figure 1e. The code with a yellow background deals with alignment. Below presents instantiations of boxes that correspond to the block-proposal guide programs given in Figure 1e.

$$\begin{array}{rl} \mathit{Guide}_{2,d}: & \square_1 = \textsc{Cat}(2/5; 119/200; 1/200), \square_2 = \square_3 = \square_4 = \square_5 = \textsc{Keep}, \square_6 = \square_7 = \square_8 = \textsc{Normal}(0, 0.5) \\ \mathit{Guide}_{2,c_0}: & \square_2 = \textsc{Normal}(\mathit{old\_c}_0, 0.5), \square_1 = \square_3 = \square_4 = \square_5 = \textsc{Keep}, \square_6 = \square_7 = \square_8 = \textsc{Normal}(0, 0.5) \\ \mathit{Guide}_{2,c_1}: & \square_3 = \textsc{Normal}(\mathit{old\_c}_1, 0.5), \square_1 = \square_2 = \square_4 = \square_5 = \textsc{Keep}, \square_6 = \square_7 = \square_8 = \textsc{Normal}(0, 0.5) \\ \mathit{Guide}_{2,c_2}: & \square_4 = \textsc{Normal}(\mathit{old\_c}_2, 0.5), \square_1 = \square_2 = \square_3 = \square_5 = \textsc{Keep}, \square_6 = \square_7 = \square_8 = \textsc{Normal}(0, 0.5) \\ \mathit{Guide}_{2,n}: & \square_5 = \textsc{InvGamma}(1, 1), \square_1 = \square_2 = \square_3 = \square_4 = \textsc{Keep}, \square_6 = \square_7 = \square_8 = \textsc{Normal}(0, 0.5) \end{array}$$

They all fill in $\square_6$, $\square_7$, and $\square_8$ in the same way: those sampling commands are in the branches where the current control flow diverges from the previous trace. For other boxes, the guide coroutines resample the random variable of interest and use $\textbf{sample}(\textsc{Keep})$ for other unchanged variables.

2.3 Coverage-Annotated Guide Types for Soundly Composed Guides

Coverage annotations. We now consider the guide types of the block guides shown above. Figure 3a defines a guide type OLat that prescribes the communication through the *old* channel. Dual to the $\&$ type constructor, the type $A \oplus B$ specifies a channel whose *receiver* receives a branch selection and proceeds with a type A or type B protocol. The type OLat has the same structure as the type Lat; the difference is that OLat can be obtained by replacing all the $\&$ constructor in Lat with $\oplus$.

The *lat* channel has a variant of the Lat guide type where primitive types (e.g., $\mathbb{R}$) are annotated with *coverage annotations* in subscripts. An annotation c ("covered") means a random variable is freshly resampled in this guide, and an annotation u ("uncovered") means an old value of the random variable, if exists in the previous trace, is reused. Below summarizes the coverage-annotated types for the five coroutines.

$$\mathsf{Lat}_{2,d} \stackrel{\text{def}}{=} (\mathbb{N}_3)_c \wedge \mathbb{R}_u \wedge \mathsf{Coeffs}_{2,d}[(\mathbb{R}_+)_u \wedge \mathbf{1}], \qquad \mathsf{Coeffs}_{2,d} \stackrel{\text{def}}{=} X \,\&\, (\mathbb{R}_u \wedge (X \,\&\, (\mathbb{R}_u \wedge X)))$$
$$\mathsf{Lat}_{2,c_0} \stackrel{\text{def}}{=} (\mathbb{N}_3)_u \wedge \mathbb{R}_c \wedge \mathsf{Coeffs}_{2,c_0}[(\mathbb{R}_+)_u \wedge \mathbf{1}], \qquad \mathsf{Coeffs}_{2,c_0} \stackrel{\text{def}}{=} X \,\&\, (\mathbb{R}_u \wedge (X \,\&\, (\mathbb{R}_u \wedge X)))$$
$$\mathsf{Lat}_{2,c_1} \stackrel{\text{def}}{=} (\mathbb{N}_3)_u \wedge \mathbb{R}_u \wedge \mathsf{Coeffs}_{2,c_1}[(\mathbb{R}_+)_u \wedge \mathbf{1}], \qquad \mathsf{Coeffs}_{2,c_1} \stackrel{\text{def}}{=} X \,\&\, (\mathbb{R}_c \wedge (X \,\&\, (\mathbb{R}_u \wedge X)))$$
$$\mathsf{Lat}_{2,c_2} \stackrel{\text{def}}{=} (\mathbb{N}_3)_u \wedge \mathbb{R}_u \wedge \mathsf{Coeffs}_{2,c_1}[(\mathbb{R}_+)_u \wedge \mathbf{1}], \qquad \mathsf{Coeffs}_{2,c_2} \stackrel{\text{def}}{=} X \,\&\, (\mathbb{R}_u \wedge (X \,\&\, (\mathbb{R}_c \wedge X)))$$
$$\mathsf{Lat}_{2,n} \stackrel{\text{def}}{=} (\mathbb{N}_3)_u \wedge \mathbb{R}_u \wedge \mathsf{Coeffs}_{2,c_1}[(\mathbb{R}_+)_c \wedge \mathbf{1}], \qquad \mathsf{Coeffs}_{2,n} \stackrel{\text{def}}{=} X \,\&\, (\mathbb{R}_u \wedge (X \,\&\, (\mathbb{R}_u \wedge X)))$$

Type-equality checking. To satisfy the model-guide compatibility, the model and guide(s) must have *equal* guide types for the *lat* channel. To this end, it is not enough to check their syntactic equality. For example, if for the $\mathit{Guide}_{2,n}$ coroutine we want the proposal distribution for the noise variable to depend on the degree of the polynomial, we would move the **sample** command in line 29 of Figure 3c into the branching commands and derive its guide type for the *lat* channel as

$$\mathsf{Lat}'_{2,n} \stackrel{\text{def}}{=} (\mathbb{N}_3)_u \wedge \mathbb{R}_u \wedge (((\mathbb{R}_+)_c \wedge \mathbf{1}) \,\&\, (\mathbb{R}_u \wedge (((\mathbb{R}_+)_c \wedge \mathbf{1}) \,\&\, (\mathbb{R}_u \wedge ((\mathbb{R}_+)_c \wedge \mathbf{1}))))),$$

which is *structurally* equal to $\mathsf{Lat}_{2,n}$. Wang et al. [52] developed a *nominal* type system, which cannot check the equality between $\mathsf{Lat}_{2,n}$ and $\mathsf{Lat}_{2,n'}$. Generally, guide types may have infinite state spaces, which enable guide types to express complex probabilistic models such as probabilistic context-free grammars [26]. However, infinite state spaces also pose a challenge to deciding structural type equality. In §4, we show that structural type equality is decidable in polynomial time by translating guide types to context-free processes with finite norms.

Coverage checking. In addition to the model-guide type equality, we must verify that every random variable is freshly sampled by at least some guide in the sequential composition. It is not enough to compute the superposition of all coverage-annotated guide types and check that the superposition is fully covered (i.e., all random variables come with subscript c). This is because old samples of one random variable can be reused for another random variable on a different execution path (§5.2). In §5, we present a coverage-checking algorithm that verifies the full coverage of sequentially composed guides by bisimulating guide types alongside the code of guides.

2.4 A Surface Syntax for Automatic Generation of BMH Guides

So far, block guide coroutines are verbose. As Figure 3c demonstrates, if guide coroutines share an identical structure that can be captured by a template, it is possible to automate block-guide generation. We propose a lightweight surface syntax to aid the users to implement such canonical guide coroutines easily. Figure 4 demonstrates a reimplementation of the model and proposal programs in Figure 1a and Figure 1e in our surface syntax. The model coroutine shown in Figure 4b is almost identical to the one shown in Figure 3b, except that the code with a blue background explicitly assigns a unique label to each sample site. We use those labels only to guide the *elaboration* of guide coroutines shown in Figure 4c into the form shown in Figure 3c. In essence, the elaboration process automatically

- transforms the model program with labels (Figure 4b) to two programs: a model coroutine without labels (Figure 3b) and a template of guide coroutines (Figure 3c); and then
- translates each guide program in the surface syntax (Figure 4c) to an instantiation of boxes, e.g., $\square_i$ for $i \in \{1, \ldots, 8\}$ in the template program shown in Figure 3c.

The first step can be realized by a straightforward syntax-directed transformation. The second step needs to translate each **resample** and **resample_if_none** command to an instantiation of one or more boxes. Both kinds of resampling commands are parameterized by a channel name and take two arguments: (i) the label for the random variable to be resampled, and (ii) a function that computes a proposal distribution from available random variables of the previous trace. A **resample** command is intended to mutate a random variable whose value is present in the previous trace, whereas a **resample_if_none** command is intended to generate a value for a random variable

$$\mathsf{Lat} \stackrel{\text{def}}{=} \mathbb{N}_3 \wedge \mathbb{R} \wedge \mathsf{Coeffs}[\mathbb{R}_+ \wedge \mathbf{1}]$$
$$\mathsf{Coeffs}[X] \stackrel{\text{def}}{=} X \,\&\, (\mathbb{R} \wedge (X \,\&\, (\mathbb{R} \wedge X)))$$
$$\mathsf{Obs} \stackrel{\text{def}}{=} \mathbb{R} \wedge \mathbb{R} \wedge \mathbb{R} \wedge \mathbb{R} \wedge \mathbb{R} \wedge \mathbf{1}$$

(a) Definitions of type operators.

```
1  proc Model(xs : vec[5](ℝ))
2  consume lat :: Lat
3  provide obs :: Obs =
4   degree ← sample_rv{lat}(@d, Cat(0.3; 0.5; 0.2));
5   c0 ← sample_rv{lat}(@c0, Normal(0, 2));
6   f ← (
7    if_sd{lat} degree = 0 then
8     return(λx. c0)
9    else
10    c1 ← sample_rv{lat}(@c1, Normal(0, 2));
11    if_sd{lat} degree = 1 then
12     return(λx. c0 + c1 ∗ x)
13    else
14     c2 ← sample_rv{lat}(@c2, Normal(0, 2));
15     return(λx. c0 + c1 ∗ x + c2 ∗ x ∗ x)
16   );
17   noise2 ← sample_rv{lat}(@n, InvGamma(1, 1));
18   noise ← return(sqrt(noise2));
19   ys ← foreach (i, x) in xs (
20    y ← sample_sd{obs}(Normal(f(x), noise));
21    return(y)
22   );
23   return(ys)
```

(b) The model coroutine.

```
1  proc Guide2,d() provide lat :: Lat =
2   degree ← resample{lat}(@d,
3    λold_d. Cat(2/5; 119/200; 1/200));
4   c1 ← resample_if_none{lat}(@c1,
5    λold_d. λold_c0. Normal(0, 0.5));
6   c2 ← resample_if_none{lat}(@c2,
7    λold_d. λold_c0. λold_c1. Normal(0, 0.5));
8   return()
```

```
1  proc Guide2,c0() provide lat :: Lat =
2   c0 ← resample{lat}(@c0,
3    λold_d. λold_c0. Normal(old_c0, 0.5));
4   return()
```

```
1  proc Guide2,c1() provide lat :: Lat =
2   c1 ← resample{lat}(@c1,
3    λold_d. λold_c0. λold_c1. Normal(old_c1, 0.5));
4   return()
```

```
1  proc Guide2,c2() provide lat :: Lat =
2   c2 ← resample{lat}(@c2,
3    λold_d. λold_c0. λold_c1. λold_c2. Normal(old_c2, 0.5));
4   return()
```

```
1  proc Guide2,n() provide lat :: Lat =
2   noise2 ← resample{lat}(@n,
3    λold_d. λold_c0. λold_n. InvGamma(1, 1));
4   return()
```

(c) The guide coroutines.

Fig. 4. Guide-typed coroutines (in the surface syntax) for the regression model and BMH proposals.

whose old value is *not* present. The set of available random variables is an under-approximation based on the data flow of the model program; for example, the values of $@d, @c_0, @c_1, @c_2$ are available for resampling $@c_2$ and the values of $@d, @c_0$ are available for resampling $@n$. In this way, we can associate each resampling command with one or more boxes. For example, for the guides in Figure 4c and the template in Figure 3c: **resample**$\{lat\}(@d, \ldots)$ corresponds to $\square_1$, **resample_if_none**$\{lat\}(@c_1, \ldots)$ corresponds to $\square_7$, **resample_if_none**$\{lat\}(@c_2, \ldots)$ corresponds to $\square_6$ and $\square_8$, **resample**$\{lat\}(@c_0, \ldots)$ corresponds to $\square_2$, **resample**$\{lat\}(@c_1, \ldots)$ corresponds to $\square_3$, **resample**$\{lat\}(@c_2, \ldots)$ corresponds to $\square_4$, and **resample**$\{lat\}(@n, \ldots)$ corresponds to $\square_5$.

In this article, we will focus on the more verbose core calculus demonstrated in Figure 3. Such verbosity allows the user to implement block guides more flexibly; for example, inside a program fragment that does not involve branching, the user can first read all the old samples and then use them to propose a value for a particular random variable.

3 Core Calculus for Coroutine-Based Programmable Inference

In coroutine-based programmable inference, model coroutines dictate control flows, while guide coroutines specify user-customized distributions of latent variables. Given a model M and a sequential composition of guides $G_1, \ldots, G_n$, Figure 5 illustrates the communication among a guide G_i, the model M, and a guide G_{i-1} ($i = 2, \ldots, n$). The guide G_i sends samples of *latent* variables to

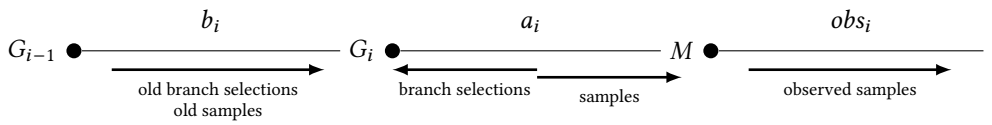


Fig. 5. Sequential composition of guides G_i ($i = 1, \ldots, n$). Black circles indicate the channel providers.

the model M across a channel a_i, and the model sends back branch selections to the guide. The model M sends samples of *observed* variables on a channel obs_i. A novelty of our new framework is that the guide G_i now has access to the old sample trace from the previous guide G_{i-1} and can choose to reuse old samples. The guide G_i receives old samples and branch selections from the previous guide G_{i-1} on a channel b_i.

3.1 Syntax

The core calculus consists of two layers: functional and coroutine layers. The former is a standard functional programming language augmented with probability distributions. The latter defines model and guide programs that communicate with each other by message passing across channels.

Functional layer. Types τ and expressions e in the functional layer are formed by this grammar:

$\tau ::= \mathbb{1} \mid \mathbb{2} \mid \mathbb{R} \mid \mathbb{R}_{(0,1)} \mid \mathbb{R}_+ \mid \mathbb{N}_n \mid \mathbb{N} \mid \tau_1 \to \tau_2 \mid \mathsf{dist}(\tau)$	base, arrow, and distribution types
$e ::= x \mid \mathsf{triv} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}(e; e_1; e_2) \mid \bar{r} \mid \bar{n} \mid \mathsf{op}_\diamond(e_1; e_2)$	expressions; $\bar{r} \in \mathbb{R}, \bar{n} \in \mathbb{N}$
$\mid \lambda(x.e) \mid \mathsf{app}(e_1; e_2) \mid \mathsf{let}(x; e_1.e_2)$	
$\mid \textsc{Ber}(e) \mid \textsc{Unif} \mid \textsc{Beta}(e_1, e_2) \mid \textsc{Pois}(e) \mid \cdots$	distribution expressions.

Probability distributions have types $\mathsf{dist}(\tau)$, where τ is the type of the supports of distributions.

Guide types. In the coroutine layer, guide types describe communication protocols between two endpoints of channels. Fix a set $\mathbb{X}$ of type variables and a set $\mathbb{T}$ of unary type operators. Guide types A are defined by

$t ::= \tau \mid \tau_c \mid \tau_u$	normal and coverage-annotated functional types
$A ::= X \mid \mathbf{1} \mid T[A]$	type variable, termination, and type application; $X \in \mathbb{X}, T \in \mathbb{T}$
$\mid t \wedge A \mid t \supset A$	send and receive samples
$\mid A_1 \oplus A_2 \mid A \,\&\, A_2$	send and receive branch selections
$\mathcal{T} ::= \overrightarrow{\mathsf{typedef}(T.X.A)}$	mutually recursive type definitions.

Type t is either an unannotated type τ from the functional layer or a coverage-annotated type (τ_c or τ_u), which ranges over coverage-annotated analogues ($\mathbb{1}_c$, $\mathbb{1}_u$, $\mathbb{R}_c$, $\mathbb{R}_u$, …) of the normal types. The subscript c ("covered") means the random variable is freshly sampled, and the subscript u ("uncovered") means the random variable is reused, whenever available, from the previous trace. Coverage-annotated guide types are only used for channels a_i that connect model and guide coroutines (Figure 5). Channels b_i are typed with unannotated guide types.

The guide type $\mathbf{1}$ means termination, $X \in \mathbb{X}$ is a type variable, and $T[A]$ is a unary type operator $T \in \mathbb{T}$ applied to a guide type A. For each channel, we designate one of its two endpoints as a provider[1] and the other endpoint as a client. The guide type of a channel is described from the channel provider's viewpoint. Guide type $t \wedge A$ means the provider sends a sample of type t to the client, and guide type $t \supset A$ means the provider receives a sample of type t from the client. Guide type $A_1 \oplus A_2$ means the provider sends a branch selection $v \in \{\mathsf{true}, \mathsf{false}\}$ and proceeds

[1]Although the two endpoints of a channel can send messages in both directions, they are assigned different roles (i.e., a provider and a client). These different roles are needed because guide types are based on binary session types, which in turn correspond to intuitionistic linear logic [5].

with guide type A_1 (if $v = \mathsf{true}$) or A_2 (otherwise). Guide type $A_1 \,\&\, A_2$ means the provider receives a branch selection and proceeds with guide type A_1 or A_2. Vector $\mathcal{T}$ stores mutually recursive type definitions of the form $T[X] \coloneqq A$.

Coroutines. Given a set $\mathbb{F}$ of procedure identifiers, commands m for model coroutines are

$$
\begin{array}{lll}
m ::= & \mathsf{ret}(e) \mid \mathsf{bnd}(m_1; x.m_2) \mid \mathsf{call}(f; e) & \text{return a value, let-binding, and procedure call; } f \in \mathbb{F} \\
& \mid \mathsf{sample}_{\mathsf{rv}}\{a\}(e) \mid \mathsf{sample}_{\mathsf{sd}}\{obs\}(e) & \text{receive a sample and send a sample} \\
& \mid \mathsf{cond}(e; m_1; m_2) & \text{conditional command for models} \\
\mathcal{D}_M ::= & \overrightarrow{\mathsf{fix}(f.x.m)} & \text{mutually recursive procedure definitions.}
\end{array}
$$

The syntax for model coroutines has two sampling commands: $\mathsf{sample}_{\mathsf{rv}}\{a\}(e)$ and $\mathsf{sample}_{\mathsf{sd}}\{obs\}(e)$. The former receives a sample from a guide on channel a. The latter draws a fresh sample for an observed variable, sending it on channel obs. Conditional command $\mathsf{cond}(e; m_1; m_2)$ branches on a Boolean expression e and proceeds to either command m_1 or m_2. Vector $\mathcal{D}_M$ stores mutually recursive procedure definitions of the form $f(x) \coloneqq m$.

Given a set $\mathbb{F}$ of procedure identifiers, commands m for guide coroutines are defined by

$$
\begin{array}{lll}
m ::= & \mathsf{ret}(e) \mid \mathsf{bnd}(m_1; x.m_2) \mid \mathsf{call}(f; e) & \text{return a value, let-binding, and procedure call; } f \in \mathbb{F} \\
& \mid \mathsf{sample}(e) \mid \mathsf{sample}(\mathsf{keep}) & \text{draw a fresh sample and reuse an old sample} \\
& \mid \mathsf{oldsample} & \text{return an old sample} \\
& \mid \mathsf{cond}(\star; m_1; m_2) \mid \mathsf{oldcond}(m_1; m_2) & \text{conditionals for current and old branch selections} \\
\mathcal{D}_G ::= & \overrightarrow{\mathsf{fix}(f.x.m)} & \text{mutually recursive procedure definitions.}
\end{array}
$$

Guide coroutines have two sampling commands[2]: $\mathsf{sample}(e)$ and $\mathsf{sample}(\mathsf{keep})$. The former draws a fresh sample from a distribution e, whereas the latter reuses the old sample. Command $\mathsf{oldsample}$ returns the old sample. Conditional commands $\mathsf{cond}(\star; m_1; m_2)$ and $\mathsf{oldcond}(m_1; m_2)$ are used inside guide programs. The first conditional command $\mathsf{cond}(\star; m_1; m_2)$ branches on the current branch selections sent from the model M, while the second conditional command $\mathsf{oldcond}(m_1; m_2)$ branches on the old branch selections from the previous guide.

Finally, an inference program for BMH is $\mathcal{P} = (\mathcal{D}_M \cup \mathcal{D}_G, m_M, (m_{G,1}, \ldots, m_{G,n}))$, consisting of a collection $\mathcal{D}_M \cup \mathcal{D}_G$ of procedure definitions, a model coroutine m_M, and a sequential composition of guide coroutines $m_{G,1}, \ldots, m_{G,n}$ ($n \geq 1$) interleaved with the MH acceptance routines.

3.2 Operational Semantics

We adapt the trace-based semantics of models and guides from prior work [52]. To support BMH, we propose a novel semantics of guide programs that access and reuse old samples.

Guidance traces. A guidance trace records the sequence of messages exchanged between two coroutines across a channel. Formally, a trace σ is a finite sequence of two kinds of messages: (i) $\boldsymbol{val}(v)$ containing a sample v and (ii) $\boldsymbol{dir}(v)$ containing a branch selection $v \in \{\mathsf{true}, \mathsf{false}\}$.

Models. The big-step operational semantics of a model program m is given by a judgment

$$V; \{a : \sigma_{a,1}\} \vdash m \Downarrow^{w} v; \{a : \sigma_{a,2}\}, \tag{3.1}$$

where V is an environment (i.e., a mapping from variables to values), a is a channel between the model and guide (Figure 5), $\sigma_{a,i}$ ($i = 1, 2$) is a trace on the channel a, $w \in [0, 1]$ is a density associated with m's run, and v is the final output. The judgment (3.1) means that, with an initial trace $\sigma_{a,1}$

[2]The sampling commands in guide coroutines are not annotated with the directions of messages or channel names, unlike the sampling commands $\mathsf{sample}_{\mathsf{rv}}\{a\}(e)$ and $\mathsf{sample}_{\mathsf{sd}}\{obs\}(e)$ in model coroutines. This is because the sampling command $\mathsf{sample}(e)$ and $\mathsf{sample}(\mathsf{keep})$ in guide coroutines are always sent from a guide to a model on channel a.

$$\frac{V \vdash e \Downarrow d \qquad v \in d.\mathsf{support} \qquad w = d.\mathsf{density}(v)}{V; \{a : (\boldsymbol{val}(v) :: \sigma_a), b : \sigma_b\}; Q \vdash \mathsf{sample}(e) \Downarrow^w v; \{a : \sigma_a, b : \sigma_b\}; \mathsf{pop}(Q)}\ \text{E:Sample}$$

$$\frac{v = \mathsf{get}(Q, \sigma_b)}{V; \{a : (\boldsymbol{val}(v) :: \sigma_a), b : \sigma_b\}; Q \vdash \mathsf{sample}(\mathsf{keep}) \Downarrow^1 v; \{a : \sigma_a, b : \sigma_b\}; \mathsf{pop}(Q)}\ \text{E:Sample:Keep}$$

$$\frac{}{V; \{a : \sigma_a, b : (\boldsymbol{val}(v) :: \sigma_b)\}; Q \vdash \mathsf{oldsample} \Downarrow^1 v; \{a : \sigma_a, b : \sigma_b\}; \mathsf{push}(Q, v)}\ \text{E:OldSample}$$

$$\frac{v_a = v_b \qquad i = \mathrm{ite}(v_a, 1, 2) \qquad V; \{a : \sigma_{a,1}, b : \sigma_{b,1}\}; Q_1 \vdash m_{i,1} \Downarrow^w v; \{a : \sigma_{a,2}, b : \sigma_{b,2}\}; Q_2}{V; \{a : (\boldsymbol{dir}(v_a) :: \sigma_{a,1}), b : (\boldsymbol{dir}(v_b) :: \sigma_{b,1})\}; Q_1 \vdash \mathsf{cond}(\star; \mathsf{oldcond}(m_{1,1}; m_{1,2}); \mathsf{oldcond}(m_{2,1}; m_{2,2})) \Downarrow^w v; \{a : \sigma_{a,2}, b : \sigma_{b,2}\}; Q_2}\ \text{E:Cond:Eq}$$

$$\frac{\begin{array}{c} v_a \neq v_b \qquad i = \mathrm{ite}(v_a, 1, 2) \\ j = \mathrm{ite}(v_b, 1, 2) \qquad V; \{a : \sigma_{a,1}\}; \cdot \vdash m_{i,2} \Downarrow^w v; \{a : \sigma_{a,2}\}; \cdot \qquad V; \{b : \sigma_{b,1}\}; Q_1 \vdash m_{j,1} \Downarrow^- \_; \{b : \sigma_{b,2}\}; Q_2 \end{array}}{V; \{a : (\boldsymbol{dir}(v_a) :: \sigma_{a,1}), b : (\boldsymbol{dir}(v_b) :: \sigma_{b,1})\}; Q_1 \vdash \mathsf{cond}(\star; \mathsf{oldcond}(m_{1,1}; m_{1,2}); \mathsf{oldcond}(m_{2,1}; m_{2,2})) \Downarrow^w v; \{a : \sigma_{a,2}, b : \sigma_{b,2}\}; Q_2}\ \text{E:Cond:Neq}$$

Fig. 6. Key rules for the operational semantics of guide programs.

on the channel a and an environment V, the model m runs successfully (without any deadlocks) with a density w, an output value v, and a continuation trace $\sigma_{a,2}$. The judgment (3.1) in Wang et al. [52] additionally mentions a channel *obs* for observed variables (Figure 5). But because observed variables are not important in this article, for brevity, we omit the channel *obs* from the judgment (3.1). Because we do not modify the semantics of model programs, the judgment (3.1) has the same definition as in Wang et al. [52].

Guides. For a guide program m, its new big-step operational semantics is given by a judgment

$$V; \{a : \sigma_{a,1}, b : \sigma_{b,1}\}; Q_1 \vdash m \Downarrow^w v; \{a : \sigma_{a,2}, b : \sigma_{b,2}\}; Q_2, \tag{3.2}$$

where V is an environment, a is a channel between the guide and model, b is a channel between this guide and the previous one, $w \in [0, 1]$ is a density associated with m's run, and v is an output value. The judgment (3.2) means that, with initial traces $\sigma_{a,1}$ and $\sigma_{b,1}$ (i.e., old trace containing old samples and branch selections) and an environment V, the command m runs successfully with a density w, an output value v, and continuation traces $\sigma_{a,2}$ and $\sigma_{b,2}$. Additionally, the judgment (3.2) contains an initial queue Q_1 and a continuation queue Q_2. The queues are used to track old samples. When the guide runs a command $\mathsf{sample}(\mathsf{keep})$, the old sample is sent to the model. Here, the queue comes in: the guide pops an element off the queue and sends it to the model.

The queue Q in the judgment (3.2) takes one of two forms: (i) $b : [v_1, \ldots, v_n]$ and (ii) $a : n$ for some $n \in \mathbb{N}$. To illustrate them, consider the communication between a guide G_i and a model M. Suppose the guide G_i has received $n \in \mathbb{N}$ more samples from the previous guide G_{i-1} than G_i has sent to the model M. In such a scenario, the n old samples $v_1, \ldots, v_n$ that have been received by the guide G_i but not yet sent to the model M are stored in a queue $Q \equiv b : [v_1, \ldots, v_n]$. Conversely, if the guide G_i has sent $n \in \mathbb{N}$ more samples to the model M than has received from the previous guide G_{i-1}, the queue takes the form $Q \equiv a : n$.

Definition. Figure 6 displays key rules for the operational semantics of guide programs. The rule E:Sample evaluates expression e to a distribution, draws a sample from it, and pops the queue Q. The rule E:Sample:Keep gets the old sample $v = \mathsf{get}(Q, \sigma_b)$ from the previous guide G_{i-1}. In this rule, both the queue Q and trace σ_b are necessary because the old value v is stored inside either the queue Q or the trace σ_b, depending on which of the channels a and b is ahead of the other. The rule E:OldSample returns the old sample, which is the first element of the old trace σ_b. We also push it to the queue Q so that it can later be sent to the model M if necessary.

The rules E:Cond:Eq and E:Cond:Neq concern a doubly nested conditional command that has four branches. The outer conditional $\mathsf{cond}(\star;\cdot;\cdot)$ branches on the model M's branch selection, and the inner conditional $\mathsf{oldcond}(\cdot;\cdot)$ branches on the previous guide's branch selection. In branch $m_{i,j}$ $(i, j \in \{1, 2\})$, i indicates the branch taken by the model M, and j indicates whether the model and previous guide have the same branch selection ($j = 1$ means identical branch selections).

If the model and previous guide have the same branch selection, the rule E:Cond:Eq applies, proceeding with a command $m_{i,1}$. Conversely, if the model and previous guide have different branch selections, the rule E:Cond:Neq applies. Because the current and previous traces diverge, the guide no longer has access to the old trace. Hence, we run $m_{i,2}$ without access to the channel b for the old trace. At the same time, we run $m_{j,1}$ with trace $\sigma_{b,1}$ on the channel b in order to determine the continuation trace $\sigma_{b,2}$ and continuation queue Q_2. When we exit the doubly nested conditional command, the current and previous traces join back, and the old trace $\sigma_{b,2}$ becomes accessible to the guide again.

Sequential composition of guides. The operational semantics of a sequential composition of closed guide coroutines $G_1, \ldots, G_n$ is defined as follows. For $i = 1, \ldots, n$, channel a_i connects model M and guide G_i, and channel b_i connects guides G_{i-1} and G_i (Figure 5). Consider an initial trace σ_0 that the model M can generate with a positive density $w_{M,0} > 0$ and an output value $v_{M,0}$:

$$\cdot; \{a : \sigma_0\} \vdash M \Downarrow^{w_{M,0}} v_{M,0}; \{a : [\,]\}. \tag{3.3}$$

The initial trace σ_0 is fed to the first guide G_1 on the channel b_1. Using σ_0 as the old trace, the guide G_1 produces a new trace σ_1^* on the channel a_1 with a positive density $w_{G,1} > 0$. We next perform the MH update, calculating a ratio r_1 (Eq. (3.7)) and setting $\sigma_1 \coloneqq \sigma_1^*$ with probability $\min\{r_1, 1\}$. Otherwise, we retain the old trace and set $\sigma_1 \coloneqq \sigma_0$. The trace σ_1 is then fed to the second guide G_2 as the old trace on the channel b_2, and the guide produces a new trace σ_2^*. This continues until we obtain the final trace σ_n.

Formally, guide G_i generates a trace σ_i^* with a positive density $w_{G,i} > 0$ and an output value $v_{G,i}$:

$$\cdot; \{a_i : \sigma_i^*, b_i : \sigma_{i-1}\}; Q_{\text{empty}} \vdash G_i \Downarrow^{w_{G,i}} v_{G,i}; \{a : [\,], b : [\,]\}; Q_{\text{empty}} \qquad i = 1, \ldots, n. \tag{3.4}$$

Here, Q_{empty} is the empty queue. The trace σ_i^* is generated by the model M with a positive density $w_{M,i} > 0$:

$$\cdot; a : \sigma_i^* \vdash M \Downarrow^{w_{M,i}} v_{M,i}; a : [\,] \qquad i = 1, \ldots, n. \tag{3.5}$$

Furthermore, we can swap the traces σ_i^* and σ_{i-1} in Eq. (3.4) while keeping the density positive:

$$\cdot; \{a_i : \sigma_{i-1}, b_i : \sigma_i^*\}; Q_{\text{empty}} \vdash G_i \Downarrow^{\hat{w}_{G,i}} \hat{v}_{G,i}; \{a : [\,], b : [\,]\}; Q_{\text{empty}} \qquad i = 1, \ldots, n \tag{3.6}$$

for an output value $\hat{v}_{G,i}$ and a positive density $\hat{w}_{G,i} > 0$. The acceptance ratio r_i in the MH update is

$$r_i \coloneqq \frac{p_M(\sigma_i^*)}{p_M(\sigma_{i-1})} \cdot \frac{p_{G_i}(\sigma_{i-1} | \sigma_i^*)}{p_{G_i}(\sigma_i^* | \sigma_{i-1})} = \frac{w_{M,i}}{w_{M,i-1}} \cdot \frac{\hat{w}_{G,i}}{w_{G,i}} \qquad i = 1, \ldots, n, \tag{3.7}$$

where $p_M(\sigma)$ is the density of a trace σ in the model M, and $p_{G_i}(\sigma_1 \mid \sigma_2)$ is the density of a trace σ_1 in the guide G_i with σ_2 being the old trace. As long as the guide G_i is well-typed, because all of $w_{M,i}, w_{M,i-1}, \hat{w}_{G,i}, w_{G,i}$ are positive, Eq. (3.7) is positive (and finite). Hence, we always have a positive probability of accepting the proposed trace σ_i^* in every MH update (Corollary A.8).

3.3 Type System

Type system. The typing judgment for a guide program m is

$$\Gamma; a : A_1, b : B_1 \vdash m \div \tau; a : A_2, b : B_2, \tag{3.8}$$

where Γ is a functional typing context, A_1 and B_1 are the initial guide types of channels a and b, respectively, τ is the output type of command m, and A_2 and B_2 are the continuation guide types of channels a and b, respectively. The judgment (3.8) means that, starting with well-typed traces

$\sigma_{a,1} : A_1$ and $\sigma_{b,1} : B_1$ and an environment $V : \Gamma$, the guide program m will run successfully, with an output value of type τ and continuation traces of guide types A_2 and B_2.

A key typing rule is T:Cond for a doubly nested conditional command:

$$\frac{\begin{array}{c}\Gamma; a : A_1, b : B_1 \vdash m_{1,1} \div \tau; a : A, b : B \qquad \Gamma; a : A_1' \vdash m_{1,2} \div \tau; a : A \\ \Gamma; a : A_2, b : B_2 \vdash m_{2,1} \div \tau; a : A, b : B \qquad \Gamma; a : A_2' \vdash m_{2,2} \div \tau; a : A \qquad |A_1| = |A_1'| \qquad |A_2| = |A_2'|\end{array}}{\Gamma; a : A_1 \,\&\, A_2, b : B_1 \oplus B_2 \vdash \mathsf{cond}(\star; \mathsf{oldcond}(m_{1,1}; m_{1,2}); \mathsf{oldcond}(m_{2,1}; m_{2,2})) \div \tau; a : A, b : B}\,\text{T:Cond}$$

If the model M takes branch $i \in \{1, 2\}$ and so does the previous guide, the current guide proceeds with command $m_{i,1}$, which is typed with initial guide types A_i and B_i. Conversely, if the model and previous guide diverge, a command $m_{i,2}$ ($i \in \{1, 2\}$) is typed with (i) an initial guide type A_i' on channel a and (ii) no access to channel b for the previous trace. Thus, to be well-typed, command $m_{i,2}$ ($i \in \{1, 2\}$) must not use $\mathsf{sample}(\mathsf{keep})$ and $\mathsf{oldsample}$. The rule T:Cond also requires $|A_i| = |A_i'|$ ($i = 1, 2$), where $|A|$ is obtained by removing coverage annotations from guide type A.

Type inference. Guide types can be automatically inferred, relieving users of the need to manually provide possibly complex guide types. To each procedure $\mathsf{fix}(f.x.m)$, we assign fresh type operators $T_{f,a}$ and $T_{f,b}$ for channels a and b, respectively. We then construct type definitions $T_{f,a}[X] \coloneqq A_f$ and $T_{f,b}[X] \coloneqq B_f$ such that

$$\Gamma; a : A_f[X], b : B_f[X] \vdash m \div \tau; a : X, b : X. \tag{3.9}$$

We traverse a command m backwards, starting with a type variable X for a continuation and incrementally building A_f and B_f. Exploiting the fact that typing rules are syntax-directed, we can determine which typing rule to apply by looking at the syntactic form of the command m.

3.4 Translation of the Lightweight Surface Syntax to the Core Calculus

This section describes how to translate a model coroutine M and a guide coroutine G from the ergonomic lightweight surface syntax to the more verbose (but more expressive) core calculus. Figure 4b and Figure 4c show the lightweight surface syntax of a model and guide coroutine, respectively. Our goal is to translate them to Figure 3b and Figure 3c, respectively, which are written in the core calculus (§3.1). To translate the model M from the surface syntax to the core calculus, we simply drop the labels of latent variables. The rest of the section focuses on the translation of the guide G.

The translation of guide G consists of two stages. In the first stage, given a model coroutine M in the surface syntax, we translate it to a template G_{templ} for guide coroutines where each expression e inside any sampling command $\mathsf{sample}(e)$ is left blank. In the second stage, each e is filled with either concrete distributions or keep (i.e., the old value is reused).

The first stage of the translation is guided by a judgment

$$C \vdash M \rightsquigarrow G_{\text{templ}}, \tag{3.10}$$

where C is a set of channels, M is a model coroutine, and G_{templ} is a template for guide coroutines. The set C of channels is either $\{a\}$ or $\{a, b\}$, where channel a connects the guide G and model M and channel b connects the current guide G and its previous guide (Figure 5). Thus, the set C tracks whether the old trace is present or not. The judgment (3.10) means that, if channels C are accessible to a guide coroutine, the model M is translated from the surface syntax to the template G_{templ}. Given a collection $\mathcal{D}_M$ of procedure definitions for the model M, we translate each procedure $\mathsf{fix}(f.x.m) \in \mathcal{D}_M$ to two versions: (i) $\mathsf{fix}(f_a.x.m_a)$ such that $\{a\} \vdash m \rightsquigarrow m_a$ and (ii) $\mathsf{fix}(f_{a,b}.x.m_{a,b})$ such that $\{a, b\} \vdash m \rightsquigarrow m_{a,b}$.

Figure 7 shows inference rules for the judgment (3.10). The rule TR:Sample is for the sampling command $\mathsf{sample}_{\mathsf{rv}}\{a\}(@v, e)$ when the channel b is present (i.e., the old trace is accessible). Here,

$$\text{TR:Ret}\ \frac{}{C \vdash \mathsf{ret}(e) \rightsquigarrow \mathsf{ret}(e)} \qquad \text{TR:Bnd}\ \frac{C \vdash m_1 \rightsquigarrow m_1' \qquad C \vdash m_2 \rightsquigarrow m_2'}{C \vdash \mathsf{bnd}(m_1; x.m_2) \rightsquigarrow \mathsf{bnd}(m_1'; x.m_2')} \qquad \text{TR:Call}\ \frac{}{C \vdash \mathsf{call}(f; e) \rightsquigarrow \mathsf{call}(f_C; e)}$$

$$\text{TR:Sample}\ \frac{v_{\text{old}} \text{ is a fresh label of a latent variable}}{\{a, b\} \vdash \mathsf{sample}_{\mathsf{rv}}\{a\}(@v, e) \rightsquigarrow \mathsf{bnd}(\mathsf{oldsample}; v_{\text{old}}.\mathsf{sample}(\square_v))}$$

$$\text{TR:Sample:A}\ \frac{}{\{a\} \vdash \mathsf{sample}_{\mathsf{rv}}\{a\}(@v, e) \rightsquigarrow \mathsf{sample}(\square_v)} \qquad \text{TR:Sample:Obs}\ \frac{obs \notin C}{C \vdash \mathsf{sample}_{\mathsf{sd}}\{obs\}(e) \rightsquigarrow \mathsf{ret}(\mathsf{triv})}$$

$$\text{TR:Cond}\ \frac{\{a, b\} \vdash m_i \rightsquigarrow m_{i,1} \qquad \{a\} \vdash m_i \rightsquigarrow m_{i,2} \quad (i = 1, 2)}{\{a, b\} \vdash \mathsf{cond}(e; m_1; m_2) \rightsquigarrow \mathsf{cond}(\star; \mathsf{oldcond}(m_{1,1}; m_{1,2}); \mathsf{oldcond}(m_{2,1}; m_{2,2}))} \qquad \text{TR:Cond:A}\ \frac{\{a\} \vdash m_1 \rightsquigarrow m_1' \qquad \{a\} \vdash m_2 \rightsquigarrow m_2'}{\{a\} \vdash \mathsf{cond}(e; m_1; m_2) \rightsquigarrow \mathsf{cond}(\star; m_1'; m_2')}$$

Fig. 7. Inference rules for the translation of the lightweight surface syntax to the core calculus.

$@v$ is a label of a latent variable. The resulting command, $\mathsf{bnd}(\mathsf{oldsample}; v_{\text{old}}.\mathsf{sample}_{\mathsf{sd}}\{a\}(\square_v))$, receives the old value, binds it to a fresh variable v_{old}, and then draws a sample from $\square_v$, which is to be filled later. The rule TR:Sample:A applies to the sampling command $\mathsf{sample}_{\mathsf{rv}}\{a\}(@v, e)$ when the channel b is absent. The rule TR:Sample:Obs applies to the sampling command $\mathsf{sample}_{\mathsf{sd}}\{obs\}(e)$, which samples an observed variable and sends it on channel *obs*. Because guides do not involve observed variables, we translate this sampling command to the no-op command $\mathsf{ret}(\mathsf{triv})$. Finally, the rule TR:Cond translates the conditional command $\mathsf{cond}(e; m_1; m_2)$ in the model M to a doubly-nested conditional command $\mathsf{cond}(\star; \mathsf{oldcond}(m_{1,1}; m_{1,2}); \mathsf{oldcond}(m_{2,1}; m_{2,2}))$ for the guide template.

In the second stage of the translation, for every sampling command $\mathsf{sample}(\square_v)$ appearing in the template G_{templ}, we fill $\square_v$ with either a distribution e or keep, according to the guide G in the surface syntax. If the guide G contains **resample**$\{a\}(@v, f)$, where function f takes in latent variables' old values and returns a distribution, then every occurrence of $\square_v$ in the template G_{templ} is replaced with a distribution $f\ v_{\text{old},1}\ \cdots\ v_{\text{old},n}$, where $v_{\text{old},1}, \ldots, v_{\text{old},n}$ are variables representing the latent variables' old values. Here, we assume that these variables are in the scope of $\mathsf{sample}(\square_v)$. Conversely, if the guide G contains **resample_if_none**$\{a\}(@v, f)$, we replace each occurrence of $\square_v$ in the template with either (i) a distribution $f\ v_{\text{old},1}\ \cdots\ v_{\text{old},n}$ if $b \notin C$ (where C is the set of channels in the judgment (3.10) of $\mathsf{sample}(\square_v)$); or (ii) keep otherwise.

To improve programmability of our system, we use several constructs that aim to simplify the workflow. Firstly, in addition to the full syntax of the core calculus (§3), we provide the lightweight surface syntax (§3.4) that makes it easier to write guide programs when the full expressiveness of the core calculus is not needed. Secondly, the operational semantics of our PPL is conceptually simple: it extends the semantics of Wang et al. [52] with one extra channel b_i connecting the previous and current guide coroutines (Figure 5). Thirdly, the guide-type system automatically infers the guide types of guide coroutines, and their structural type equality with a model coroutine's guide type is also checked automatically (§4.2). Thus, the type system requires no user interaction, though some understanding of the type system's details may be needed to debug guide programs.

4 Type-Equality Checking

We check type equality of guide types (while disregarding their coverage annotations) in two places. First, in type inference, we check that the two branches of a conditional command $\mathsf{cond}(\star; m_1; m_2)$ have equal guide types (§3.3). Second, after inferring the guide types of a model M and a guide G,

we check that they have equal guide types. Otherwise, with unequal guide types, they may cause communication errors (e.g., deadlocks) at runtime, resulting in unsound probabilistic inference.

4.1 Context-Free Guide Types

Guide types are said to be *regular* if they encode regular (tree) languages that can be recognized by finite-state (tree) automata [14, 16, 40]. For example, a guide type $T[X] \coloneqq \mathbb{R} \wedge (X \,\&\, T[X])$ is regular because, as we traverse the type and unroll recursion, we encounter finitely many syntactically different types (i.e., subtrees [40, Section 21.7]).

Type-equality checking of guide types is straightforward if they are regular. Because regular guide types can be encoded as finite-state (tree) automata, the type-equality problem can be reduced to the *bisimilarity*-checking problem between two finite-state (tree) automata. Bisimilarity means two given types, viewed as transition system, can always make the same transitions to their next states in lockstep. To ensure termination of bisimulation, we must detect a cycle, which is straightforward because we can only ever visit finitely many states during the bisimulation.

The guide-type framework [52] admits more types than regular types. For example, a guide type $T[X] \coloneqq \mathbb{R} \wedge (X \,\&\, T[T[X]])$ is non-regular. As we traverse the type $T[X]$ and expand recursion, it yields infinitely many types (e.g., $T[X], T[T[X]], \ldots$). Furthermore, a guide type $T[X]$ is said to be context-free because it can be encoded as a context-free process, which can have infinitely many states. Context-free guide types are critical for expressing a number of Bayesian-inference problems; e.g., probabilistic context-free grammars (PCFG) [26].

We now formally define type equality of guide types. Given a guide type A and a collection $\mathcal{T}$ of type definitions, let $\mathsf{unfold}_{\mathcal{T}}(A)$ denote the operation of unfolding type A [13]:

$$\frac{\mathsf{typedef}(T.X.A) \in \mathcal{T}}{\mathsf{unfold}_{\mathcal{T}}(T[B]) = \mathsf{unfold}_{\mathcal{T}}(A[B/X])} \qquad \frac{A \neq T[\_]}{\mathsf{unfold}_{\mathcal{T}}(A) = A}.$$

In contrast to Wang et al. [52], which treats guide types *iso*-recursively, this work treats guide types *equi*-recursively. It is a widely adopted convention in the literature of session types [13, 15, 47, 49] to interpret session types—on which guide types are built—equi-recursively. Under the equi-recursive interpretation, structural type equality is defined by type bisimilarity [13, 47].

Definition 4.1 (Type bisimulation). Let Type be the set of closed guide types. A binary relation $R \subseteq \text{Type} \times \text{Type}$ is a type bisimulation if and only if $(A, B) \in R$ implies:

- If $\mathsf{unfold}_{\mathcal{T}}(A) = \tau \wedge A'$, then $\mathsf{unfold}_{\mathcal{T}}(B) = \tau \wedge B'$ and $(A', B') \in R$.
- If $\mathsf{unfold}_{\mathcal{T}}(A) = A_1 \,\&\, A_2$, then $\mathsf{unfold}_{\mathcal{T}}(B) = B_1 \,\&\, B_2$ and $(A_i, B_i) \in R$ for $i \in \{1, 2\}$. The case of $\mathsf{unfold}_{\mathcal{T}}(A) = A_1 \oplus A_2$ is defined analogously.
- If $\mathsf{unfold}_{\mathcal{T}}(A) = \mathbf{1}$, then $\mathsf{unfold}_{\mathcal{T}}(B) = \mathbf{1}$.

Definition 4.2 (Guide type equality). Two closed guide types A and B are equal (denoted by $A = B$) if and only if there exists a type bisimulation R such that $(A, B) \in R$.

4.2 Bisimilarity Checking

Challenge of infinite-state bisimulation. It is a non-trivial challenge to algorithmically check bisimilarity between two guide types because they generally correspond to infinite-state transition systems. For example, consider the problem of deciding the bisimilarity between two guide types:

$$T_1[X] \coloneqq \mathbb{R} \wedge (X \,\&\, T_1[T_1[X]]) \qquad T_2[X] \coloneqq \mathbb{R} \wedge (X \,\&\, T_2[T_2[X]]). \tag{4.1}$$

Suppose we bisimulate $T_1[X]$ and $T_2[X]$ and construct a type bisimulation R that witnesses the type equivalence. Initially, we place the pair $(T_1[X], T_2[X])$ in the type bisimulation R. Next, we unfold the pair $(T_1[X], T_2[X])$ and bisimulate it, spawning a new pair $(T_1[T_1[X]], T_2[T_2[X]])$ to be

included in the type bisimulation R. This pattern continues, resulting in an infinite sequence of guide-type pairs to be included in the type bisimulation R.

Context-free processes. To algorithmically decide type equality of guide types, we reduce the problem to bisimilarity checking of so-called context-free processes that simulate context-free grammars. We formally define context-free grammars and processes as follows.

Definition 4.3 (Context-free grammar in Greibach normal form). A context-free grammar is a four-tuple $G = (V, T, P, S)$, where (i) V is a finite set of variables; (ii) T is a finite set of terminal symbols; (iii) $P \subseteq V \times (V \cup T)^*$ is a finite set of production rules; and (iv) $S \in V$ is the starting variable. The context-free grammar G is said to be in Greibach normal form (GNF) if every $(X, \alpha) \in P$ satisfies $\alpha \in TV^*$. Every context-free grammar can transformed into GNF.

Definition 4.4 (Context-free process). A process is a transition system $(S, A, \rightarrow, \alpha_0)$, where (i) S is a (possibly infinite) set of states; (ii) A is a finite set of actions; (iii) $\rightarrow \subseteq S \times A \times S$ is a transition relation; and (iv) $\alpha_0 \in S$ is the initial state. With a context-free grammar (V, T, P, S) in GNF, we associate the process $(V^*, T, \rightarrow, S)$, where there are no transitions from ϵ (i.e., the empty string), and $X\sigma \xrightarrow{a} \alpha\sigma$ if and only if $(X \rightarrow a\alpha) \in P$. Such a process is called a context-free process.

Translation from guide types to processes. Consider a closed guide type A_{main} together with a finite set $\mathcal{T}$ of type definitions of the form $\mathsf{typedef}(T.X.A)$. We translate $\mathcal{T}$ to rules of a context-free grammar/process and A_{main} to a string of variables (i.e., the initial state of the context-free process). For each type definition $\mathsf{typedef}(T.X.A) \in \mathcal{T}$, we assume A does not contain $\mathbf{1}$. This is a valid assumption in our setting because any $\mathsf{typedef}(T.X.A)$ inferred by the guide-type-inference algorithm (§3.3) for a procedure definition $\mathsf{fix}(f.x.m)$ never introduces $\mathbf{1}$.

In each type definition $\mathsf{typedef}(T.X.A)$, we preprocess A such that the type definition becomes

$$T[X] \coloneqq \tau \wedge T_1[\cdots T_n[X]\cdots], \qquad \text{or} \tag{4.2}$$

$$T[X] \coloneqq T_1[\cdots T_n[X]\cdots] \diamond T_1'[\cdots T_m'[X]\cdots] \qquad \text{where } \diamond \in \{\&, \oplus\}, \tag{4.3}$$

where $T_1, \ldots, T_n, T_1', \ldots, T_m'$ are type operators. Any type definition $T[X] \coloneqq A$ can be transformed to the forms (4.2) and (4.3) by introducing fresh type operators, as long as A does not contain $\mathbf{1}$.

Definition 4.5 (Translation of type definitions). Consider a type definition $\mathsf{typedef}(T.X.A) \in \mathcal{T}$ in either of the forms Eqs. (4.2) and (4.3). This type definition is translated to a GNF production rule(s) of a context-free grammar as

$$(T[X] \coloneqq \tau \wedge T_1[\cdots T_n[X]\cdots]) \rightsquigarrow \{T \xrightarrow{\tau\wedge} T_1T_2\cdots T_n\} \tag{4.4}$$

$$(T[X] \coloneqq T_1[\cdots T_n[X]\cdots] \diamond T_1'[\cdots T_m'[X]\cdots]) \rightsquigarrow \{T \xrightarrow{\diamond_{\mathsf{true}}} T_1\cdots T_n, T \xrightarrow{\diamond_{\mathsf{false}}} T_1'\cdots T_m'\}, \tag{4.5}$$

where $\diamond \in \{\&, \oplus\}$ in Eq. (4.5). Type operators T, T_i, T_j ($i = 1, \ldots, n$ and $j = 1, \ldots, m$) on the right-hand sides of Eqs. (4.4) and (4.5) are treated as variables of the context-free grammar. To obtain all production rules of the context-free grammar, we perform the above transformation to each type definition in $\mathcal{T}$ and aggregate the outputs.

The translation of a closed guide type A_{main} works similarly. First, it is transformed to a guide type $T_1[\cdots T_n[\mathbf{1}]\cdots]$. It is then translated to a word $T_1\cdots T_n$, where $T_1, \ldots, T_n$ are treated as variables of a context-free grammar. The result is used as the initial state of a context-free process.

Bisimilarity checking of context-free processes. The seminal work by Hirshfeld et al. [21] shows that we can check bisimilarity between context-free processes in polynomial time, provided that we impose one additional restriction: the context-free processes have finite norms.

Definition 4.6 (Norm). Consider a context-free process induced by a context-free grammar $G = (V, T, P, S)$. The norm of a word $\alpha \in V^*$ is the minimum number of transitions necessary to reach the empty string ϵ. A context-free process is said to be normed if if all states have finite norms.

Because traces must be finitely long [52], we require guide types to have finite norms as well. For example, an infinite-norm guide type $T[X] \coloneqq \mathbb{R} \wedge T[T[X]]$ should be rejected in the coroutine-based programmable inference because programs with such a guide type produce infinitely long traces of $\mathbb{R}$-typed samples in all execution paths. Finite norms are critical for polynomial-time complexity. Without this assumption, although bisimilarity checking remains decidable [10], its complexity becomes EXPTIME-hard [27] and 2-EXPTIME (double exponential) [25].

Theorem 4.7 (Polynomial-time checking of guide-type equality). *Given two guide types A_1 and A_2, if they have finite norms, their equality can be checked in polynomial time.*

Theorem 4.7 for polynomial-time type-equality checking is novel considering the fact that guide types build on context-free session types, whose type equality is EXPTIME-hard. Polynomial-time equality checking for guide types is enabled by the crucial difference between guide types and session types: the former is required to have finite norms, while the latter is not. Our contribution in this work is to spot this critical difference, show how to translate guide types to context-free processes with finite norms, and thereby conclude that guide-type equality is decidable in polynomial time.[3]

5 Coverage Checking

To verify the model-guide compatibility, in addition to the type equality between the model and guides, we check the coverage of random variables: they are each freshly sampled by some guide.

5.1 Problem Statement

We introduce the coverage-checking problem of a sequential composition of well-typed guide coroutines $G_1, \ldots, G_n$. For each $i = 1, \ldots, n$, channel a_i connects model M and guide G_i, and channel b_i connects guides G_{i-1} and G_i (Figure 5). For $i = 1, \ldots, n$, let A_i be the coverage-annotated guide type of channel a_i such that $\forall 1 \le i, j \le n.\ |A_i| = |A_j|$ and define $B = |A_i|[\oplus/\&]$ (for any i), where $|A_i|$ is the result of removing coverage annotations from A_i. Suppose we have for some functional type τ_i

$$\cdot; a_i : A_i, b : B \vdash G_i \div \tau_i; a_i : \mathbf{1}, b : \mathbf{1} \qquad i = 1, \ldots, n. \tag{5.1}$$

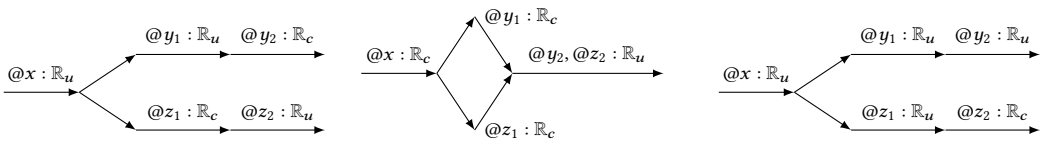
The coverage-checking problem asks the following: for any initial trace $\sigma_0 : B$ with a positive density in the model M (Eq. (3.3)) and any desirable final trace $\sigma_n : B$ also with a positive density in the model M (Eq. (3.5)), can we have

$$\cdot; \{a_i : \sigma_i, b_i : \sigma_{i-1}\}; Q_{\text{empty}} \vdash G_i \Downarrow^{w_{G,i}} v_{G,i}; \{a : [\,], b : [\,]\}; Q_{\text{empty}} \qquad (i = 1, \ldots, n) \tag{5.2}$$

for intermediate traces $\sigma_i : B$ ($i = 1, \ldots, n-1$), positive densities $w_{G,i} > 0$ ($i = 1, \ldots, n$), and output values $v_{G,i}$ ($i = 1, \ldots, n$)? If so, the Markov chain induced by the guides $G_1, \ldots, G_n$ is irreducible, which is a key soundness ingredient of multiple-block MH [41, 48].

As described in §3.2, each guide coroutine is followed by the MH acceptance routine. Guide G_i proposes a new candidate trace σ_i^*, and it is accepted with probability $\min\{r_i, 1\}$, where ratio r_i is defined in Eq. (3.7). In the formulation of the coverage-checking problem (Eq. (5.2)), without loss of generality, we focus on the case where every acceptance routine accepts the newly proposed trace σ_i^*. In our framework, as long as the old trace σ_{i-1} has a positive density in the model M (Eq. (3.5)),

[3] The original paper [21] shows a $O(n^{13})$-time algorithm, where n is the size of the input context-free grammar. [29] later improves the asymptotic complexity to $O(n^8 \text{polylog}(n))$.



(a) Guide G_1. (b) Guide G_2. (c) Guide G_3.

Fig. 8. Mismatch in the control flows of guide programs. Guide G_2 has a different control-flow graph from guides G_1 and G_3, but they all have equal guide types (ignoring their coverage annotations).

the acceptance routine is guaranteed to accept the proposed trace σ_i^* with a positive probability (Corollary A.8). Also, if the MH acceptance routine retains the old trace σ_{i-1}, we can simulate this effect by setting trace σ_i^* to σ_{i-1}, which is possible for any well-typed guide program.

5.2 Technical Challenge

A naïve solution to coverage checking is to verify that the superposition of coverage-annotated guide types $A_1, \ldots, A_n$ covers all random variables. However, this solution fails because old samples of a random variable can be reused for another random variable on a different execution path.

To demonstrate the issue, consider a sequential composition of guides G_1, G_2, G_3 whose control flows are illustrated in Figure 8. Label $@\ell : \mathbb{R}_c$ means random variable $@\ell$ is covered (i.e., freshly sampled) and $@\ell : \mathbb{R}_u$ means random variable $@\ell$ is uncovered (i.e., old sample is reused). In guides G_1 and G_3, the two branches of a conditional command remain diverged, while in guide G_2, the two branches join back after temporary divergence. This is because guide G_2 has code $\mathsf{bnd}(\mathsf{cond}(\star; m_{1,1}; m_{1,2}); x.m_2)$, where commands $m_{1,1}$ and $m_{1,2}$ join back before command m_2. The three guides have coverage-annotated guide types A_1, A_2, A_3, respectively, where

$$A_1 \coloneqq \mathbb{R}_u \wedge \& \begin{Bmatrix} \mathbb{R}_u \wedge \mathbb{R}_c \wedge \mathbf{1}, \\ \mathbb{R}_c \wedge \mathbb{R}_u \wedge \mathbf{1} \end{Bmatrix} \quad A_2 \coloneqq \mathbb{R}_c \wedge \& \begin{Bmatrix} \mathbb{R}_c \wedge \mathbb{R}_u \wedge \mathbf{1}, \\ \mathbb{R}_c \wedge \mathbb{R}_u \wedge \mathbf{1} \end{Bmatrix} \quad A_3 \coloneqq \mathbb{R}_u \wedge \& \begin{Bmatrix} \mathbb{R}_u \wedge \mathbb{R}_u \wedge \mathbf{1}, \\ \mathbb{R}_u \wedge \mathbb{R}_c \wedge \mathbf{1} \end{Bmatrix}. \tag{5.3}$$

The superposition of Eq. (5.3) covers all random variables: as we bisimulate Eq. (5.3) in lockstep, every random variables is covered by at least one of the three guides. However, this is a pitfall: the sequential composition of G_1, G_2, G_3 fails to generate some traces that model M can generate. Consider an initial trace $\sigma_0 = [v_{0,1}, \boldsymbol{dir}(\mathsf{false}), v_{0,3}, v_{0,4}]$ for some fixed values $v_{0,1}, v_{0,3}, v_{0,4} \in \mathbb{R}$. Ideally, the sequential composition of guides should be able to generate any trace

$$\sigma_3 \in \{[v_1, \boldsymbol{dir}(v_2), v_3, v_4] \mid v_1, v_3, v_4 \in \mathbb{R}, v_2 = M(v_1)\} \tag{5.4}$$

with a positive density, where $M(v_1) \in \{\mathsf{true}, \mathsf{false}\}$ denotes the branch chosen by model M given sample $v_1 \in \mathbb{R}$ for random variable $@x$. Suppose (roman*) guide G_1 takes the second branch, (roman*) G_2 takes the first branch, and (roman*) G_3 also takes the first branch because it reuses the previous sample v_1 freshly sampled by G_2. Consequently, guide G_i generates trace σ_i ($i = 1, 2, 3$):

$$\sigma_1 = [v_{0,1}, \boldsymbol{dir}(\mathsf{false}), v_3, v_{0,4}] \qquad \sigma_2 = \sigma_3 = [v_1, \boldsymbol{dir}(\mathsf{true}), v_3, v_{0,4}]. \tag{5.5}$$

Trace σ_3 still contains sample $v_{0,4}$ from the initial trace σ_0. Therefore, we cannot generate every trace from the set in Eq. (5.4) with a positive density, independently of the initial trace σ_0.

The root problem is that although G_1 and G_2 have different control flows, their guide types do not reflect this difference. Guide G_2 diverges from the old trace σ_1 after $@x$. But guide G_2 regains access to trace σ_1 after the two branches in G_2 join back. Interestingly, guide G_2 now reuses the old sample $v_{0,4}$ in σ_1, which is originally for random variable $@z_2$, for random variable $@y_2$. Thus, old samples can later be reused for different random variables in different branches. So in coverage checking, it is not sufficient to examine the superposition of coverage-annotated guide types.

$$\text{C:Call}\ \frac{\mathsf{fix}(f.x.m) \in \mathcal{D}_G \qquad \mathcal{A} \vdash m[e/x] : (A_X, \mathcal{B})}{\mathcal{A} \vdash \mathsf{call}(f; e) : (T[X], \mathcal{B})}$$

$$\text{C:Sample:Any}\ \frac{\forall i \in \mathcal{I}.\tau_i = \tau_c}{\{\tau_i \wedge A_i \mid i \in \mathcal{I}\} \vdash \mathsf{sample}(\_) : (\tau_c \wedge X, \{A_i \mid i \in \mathcal{I}\})}$$

$$\text{C:Sample:Dist}\ \frac{e : \mathsf{dist}(\tau)}{\{\tau_i \wedge A_i \mid i \in \mathcal{I}\} \vdash \mathsf{sample}(e) : (\tau_c \wedge X, \{A_i \mid i \in \mathcal{I}\})}$$

$$\text{C:Sample:Keep}\ \frac{\exists i \in \mathcal{I}.\tau_i = \tau_u}{\{\tau_i \wedge A_i \mid i \in \mathcal{I}\} \vdash \mathsf{sample}(\mathsf{keep}) : (\tau_u \wedge X, \{A_i \mid i \in \mathcal{I}\})}$$

$$\text{C:Cond}\ \frac{\{A_{i,1} \mid i \in \mathcal{I}\} \vdash m_{1,1} : (A_{1,X}, \mathcal{B}_1) \qquad \{A_{i,2} \mid i \in \mathcal{I}\} \vdash m_{2,1} : (A_{2,X}, \mathcal{B}_2)}{\{A_{i,1} \,\&\, A_{i,2} \mid i \in \mathcal{I}\} \vdash \mathsf{cond}(\star; \mathsf{oldcond}(m_{1,1}; m_{1,2}); \mathsf{oldcond}(m_{2,1}; m_{2,2})) : (A_{1,X} \,\&\, A_{2,X}, \mathcal{B}_1 \cup \mathcal{B}_2)}$$

Fig. 9. Key rules for bisimulating guide types alongside commands in the coverage-checking algorithm.

5.3 Coverage-Checking Algorithm

Key idea. To overcome the limitation described in §5.2, we propose a coverage-checking algorithm that reshapes a guide type according to the control flow of a guide program. In the example of Figure 8, we start with a fully uncovered guide type $A_0 \coloneqq \mathbb{R}_u \wedge ((\mathbb{R}_u \wedge \mathbb{R}_u \wedge \mathbf{1}) \,\&\, (\mathbb{R}_u \wedge \mathbb{R}_u \wedge \mathbf{1}))$. We bisimulate guide type A_0 alongside the code of guide G_1, updating coverage annotations whenever we encounter $\mathsf{sample}(e)$ in the code. This results in coverage-annotated guide type A_1 (Eq. (5.3)). Next, we bisimulate guide type A_1 alongside the code of guide G_2. During the bisimulation, when the two branches of G_2 merge back, we also merge the coverage-annotated base types $@y_2 : \mathbb{R}_c$ and $@z_2 : \mathbb{R}_u$ in guide type A_1, yielding $\mathbb{R}_u$ because it is their supertype. This results in a guide type A_2 (Eq. (5.3)). Finally, we bisimulate guide type A_2 alongside the code of G_3, obtaining

$$A_3' \coloneqq \mathbb{R}_c \wedge ((\mathbb{R}_c \wedge \mathbb{R}_u \wedge \mathbf{1}) \,\&\, (\mathbb{R}_c \wedge \mathbb{R}_c \wedge \mathbf{1})). \tag{5.6}$$

Guide type A_3' in Eq. (5.6) correctly indicates that random variable $@y_2$ may be uncovered.

Bisimulation of types and commands. To formalize the idea of bisimulating a guide type (and more generally a set $\mathcal{A}$ of guide types) alongside command m, we introduce a judgment

$$\mathcal{A} \vdash m : (A_X, \mathcal{B}), \tag{5.7}$$

where $\mathcal{A}$ is a set of input guide types, A_X is an output guide type containing type variable X, and $\mathcal{B}$ is a set of continuation guide types after the bisimulation. The judgment (5.7) means, given a set $\mathcal{A}$ of input guide types, as we bisimulate all guide types in $\mathcal{A}$ and command m in lockstep and update coverage annotations, we obtain an output guide type A_X, where type variable X stands for a continuation guide type, and a set $\mathcal{B}$ of continuation guide types.

Figure 9 lists key rules defining judgment (5.7). The rule C:Sample:Any applies to both $\mathsf{sample}(e)$ for a distribution expression e and $\mathsf{sample}(\mathsf{keep})$. The rule states that, if all guide types in the input set $\{\tau_i \wedge A_i \mid i \in \mathcal{I}\}$ cover the random variable, then it remains covered in the result $\tau_c \wedge A$. In the rule C:Sample:Dist, if a guide draws a fresh sample, the random variable is deemed covered in the result. Conversely, the rule C:Sample:Keep stipulates that, if the input set of guide types contains an uncovered type and the sampling command reuses an old value, the random variable is uncovered. The rule C:Call replaces a procedure call with the procedure definition. The rule C:Cond states that, for a conditional command, we consider commands $m_{1,1}$ (i.e., model M and the previous guide both take the first branch) and $m_{2,1}$ (i.e., model M and the previous guide take the second branch). The overall set of continuation guides is the union $\mathcal{B}_1 \cup \mathcal{B}_2$. It is unnecessary to consider commands $m_{1,2}$ and $m_{2,2}$ because they are disallowed from calling $\mathsf{sample}(\mathsf{keep})$ and hence always draw fresh samples.

Repeated bisimulation. The coverage-checking algorithm works as follows. Given a sequential composition of well-typed guides $G_1, \ldots, G_n$, let a_i $(i = 1, \ldots, n)$ be the channel connecting guide G_i

and model M. Let B be the unannotated guide type of all channels $a_1, \ldots, a_n$, and B_0 be the coverage-annotated guide type obtained from B by annotating all random variables with subscript u. We first bisimulate the fully uncovered guide type B_0 alongside guide G_1, resulting in $\{B_0\} \vdash G_1 : (B_{1,X}, \{\mathbf{1}\})$. Next, we bisimulate $B_1 \coloneqq B_{1,X}[\mathbf{1}/X]$ for guide G_2, repeating this step for all subsequent guides. Once we obtain the final guide type B_n, we check if it is fully covered.

Theorem 5.1 states the soundness of the coverage-checking algorithm.

Theorem 5.1 (Soundness of the coverage-checking algorithm). *Consider a sequential compositions of well-typed guides $G_1, \ldots, G_n$. Channel a_i ($i = 1, \ldots, n$) connects guide G_i and model M, and channel b_i ($i = 1, \ldots, n$) connects guides G_i and G_{i-1}. For each $i = 1, \ldots, n$, suppose*

$$\cdot; a_i : A_i, b_i : B \vdash G_i \div \tau_i; a_i : \mathbf{1}, b_i : \mathbf{1}, \tag{5.8}$$

where coverage-annotated guide types A_i and unannotated guide types B satisfy $\forall 1 \le i \le n.\ B = |A_i|$. Let B_0 be a fully uncovered coverage-annotated guide type obtained from B. Suppose

$$\{B_{i-1}\} \vdash G_i : (B_{i,X}, \{\mathbf{1}\}) \qquad B_i \coloneqq B_{i,X}[\mathbf{1}/X] \qquad i = 1, \ldots, n. \tag{5.9}$$

If B_n is fully covered (i.e., all random variables are marked with subscript c), then the Markov chain induced by the sequential composition of guides $G_1, \ldots, G_n$ is irreducible.

Implementation and heuristic. To algorithmically compute guide type A_X and set $\mathcal{B}$ in Eq. (5.7), we incrementally construct a typing tree bottom-up according to the rules in Figure 9. Every time we apply the rule C:Call for a procedure call $\mathsf{call}(f; e)$, we record the pair $(f, \mathcal{A})$, which are used to detect a cycle. If guide types are regular (i.e., they have finitely many states), we are guaranteed to detect a cycle because there can only be finitely many pairs $(f, \mathcal{A})$. However, if the guide types are context-free with infinitely many states (§4.1), then the algorithm may diverge.

To prevent the divergence caused by infinite-state context-free guide types, we can replace the rule C:Call with a heuristic rule for procedure calls:

$$\text{C:Call:Heuristic}\quad \frac{\mathcal{A} = \{T_i[A_i] \mid i \in \mathcal{I}\} \qquad \{T_i[\mathbf{1}] \mid i \in \mathcal{I}\} \vdash \mathsf{call}(f; e) : (A_X, \{\mathbf{1}\}) \qquad T_{f,\mathcal{A}} \text{ is a fresh type operator}}{\mathcal{A} \vdash \mathsf{call}(f; e) : (T_{f,\mathcal{A}}[X], \{A_i \mid i \in \mathcal{I}\})}.$$

The rule C:Call:Heuristic states that, if the set $\mathcal{A}$ of input guide types has the form $\{T_i[A_i] \mid i \in \mathcal{I}\}$, we split it into $\mathcal{A}_1 \coloneqq \{T_i[\mathbf{1}] \mid i \in \mathcal{I}\}$ and $\mathcal{A}_2 \coloneqq \{A_i \mid i \in \mathcal{I}\}$. We then bisimulate $\mathcal{A}_1$ alongside command $\mathsf{call}(f; e)$, ensuring that the output set of continuation guide types is $\{\mathbf{1}\}$. This heuristic assumes that each guide type $T_i[\mathbf{1}]$ ($i \in \mathcal{I}$) exactly matches the control flow of procedure f. Because the rule C:Call:Heuristic matches a procedure call with a set of the form $\{T_i[\mathbf{1}] \mid i \in \mathcal{I}\}$, of which there are finitely many, the coverage-checking algorithm eventually terminates. The rule C:Call:Heuristic works for infinite-state context-free guide types when all guides $G_1, \ldots, G_n$ have the same code structure with respect to their procedure-call sites: all guides call procedures in the same sites within code. However, if some procedures inline a procedure call while others do not, the heuristic C:Call:Heuristic no longer works, because some guide types in the set $\mathcal{A}$ of input guide types will not have the form $T_i[A_i]$. Thus, the coverage-checking algorithm with the heuristic is not complete, but it does not affect the soundness of coverage checking (Theorem 5.1).

6 Evaluation

Implementation. We implemented in OCaml (i) a type-inference algorithm (with equality checking) for individual guides and (ii) a coverage-checking algorithm for sequentially composed guides.

For type inference, we have extended the algorithm from [52], which only supports nominal type equality, with a saturation-based structural-type-equality checking algorithm for context-free guide types with finite norms [22] (§4). Its time complexity is $O(n^4 v)$, where n is the overall size of type definitions and v is the maximum norm of type operators [22]. This is not a polynomial-time

Table 1. Experiment results of guide-type inference and coverage checking of 28 benchmark programs.

				Type Inference		Coverage Check		
Program	**Description**	**Guide Type**	**LOC**	Time (ms)	Prior Work	Match	Mismatch	Time (ms)
`branching`	Random control flow [Anglican]	Finite	46	1.33	✓	True Pos.	True Neg.	0.46
`coordination`	Coordination game [Anglican]	Finite	24	0.19	✓	True Pos.	True Neg.	0.34
`drill`	Oil wildcatter problem [Anglican]	Finite	56	0.17	✓	True Pos.	True Neg.	0.37
`ex-1`	Ex. 1 [52]	Finite	42	1.31	✓	True Pos.	True Neg.	0.46
`gaussian`	Gaussian with unknown means [Anglican]	Finite	20	0.16	✓	True Pos.	True Neg.	0.46
`gbm`	Geometric Brownian motion [Anglican]	Finite	35	0.25	✓	True Pos.	True Neg.	0.52
`gda`	Gaussian discriminant analysis [Anglican]	Finite	40	1.86	✓	True Pos.	True Neg.	3.17
`gmm`	Gaussian mixture model [Anglican]	Finite	75	4.73	✓	True Pos.	True Neg.	7.71
`grw`	Gaussian random walk [Anglican]	Finite	24	0.17	✓	True Pos.	True Neg.	0.74
`hmm`	Hidden Markov model [Anglican]	Finite	76	2.56	✓	True Pos.	True Neg.	7.21
`kalman`	Kalman filter [Anglican]	Finite	72	4.44	✓	True Pos.	True Neg.	7.54
`kalman-chaos`	Kalman for chaotic attractors [Anglican]	Finite	114	5.86	✓	True Pos.	True Neg.	5.68
`lr`	Bayesian linear regression [Anglican]	Finite	36	0.19	✓	True Pos.	True Neg.	1.15
`run-factory`	Beta-binomial model [Anglican]	Finite	20	0.13	✓	True Pos.	True Neg.	0.61
`scientists`	Posterior estimation with Gaussians [54]	Finite	40	0.27	✓	True Pos.	True Neg.	0.52
`seq`	Non-recursive sequence [52]	Finite	22	0.23	✓	True Pos.	True Neg.	0.46
`sprinkler`	Bayesian network [Anglican]	Finite	26	0.14	✓	True Pos.	True Neg.	0.43
`user-behavior`	Dishonest form filling [Anglican]	Finite	64	1.22	✓	True Pos.	True Neg.	3.17
`vae`	Variational autoencoder [Pyro]	Finite	48	4.20	✓	True Pos.	True Neg.	22.39
`weight`	Unreliable weight [Pyro]	Finite	18	0.26	✓	True Pos.	True Neg.	0.70
`aircraft`	Aircraft detection [Anglican]	Regular	117	6.19	✗	True Pos.	True Neg.	5.96
`iter`	Regular iteration [52]	Regular	47	2.01	✗	True Pos.	True Neg.	0.54
`marsaglia`	Marsaglia algorithm [Anglican]	Regular	76	3.51	✗	True Pos.	True Neg.	5.13
`ptrace`	Poisson trace [Anglican]	Regular	47	1.49	✗	True Pos.	True Neg.	0.40
`ex-2`	Ex. 2 [52]	Context-Free	78	4.77	✗	True Pos.	True Neg.	4.70
			93	15.48	✗	False Neg.	True Neg.	3.26
`diter`	Double iteration [52]	Context-Free	52	1.48	✗	True Pos.	True Neg.	0.57
			62	2.09	✗	False Neg.	True Neg.	0.49
`gp-dsl`	Gaussian process DSL [52]	Context-Free	242	879.53	✗	True Pos.	True Neg.	4.71
			261	2487.91	✗	False Neg.	True Neg.	4.59
`recur`	Context-free recursion [52]	Context-Free	71	11.53	✗	True Pos.	True Neg.	16.32
			83	15.55	✗	False Neg.	True Neg.	6.35

algorithm, since v can be exponential in n in the worst case. Nonetheless, as long as the maximum norm is small, this algorithm has better asymptotic complexity than a worst-case polynomial-time algorithm [21], which has complexity $O(n^{13})$. This type-equality checking algorithm can also be used to verify that model and guide programs have equal guide types.

For coverage checking, starting with a fully uncovered guide type, we bisimulate the coverage-annotated guide type with each successive guide program to update coverage annotations (§5.3).

Evaluation setup. We evaluate our prototype on 28 benchmark guide programs collected from [52] and [Pyro, Anglican]. The benchmarks are modified as follows: (i) we add an extra channel b through which the guides access old traces and (ii) we split each guide program into multiple guides, each of which covers some but not all random variables.

Our benchmark set contains 20 programs with non-recursive guide types, 4 programs with regular recursive guide types, and 4 programs with infinite-state context-free guide types. Table 2 displays the guide types of these benchmarks. Each context-free benchmark has two versions: (i) all guides in the sequential composition have aligned code structures with respect to procedure calls and (ii) some of the guides' code structures are misaligned. For each benchmark (and each of the two versions of a context-free benchmark), we consider two kinds of sequentially composed guides: one where the composition is fully covered and another where the composition is not fully covered.

Results. Our goal is to evaluate the effectiveness of the type-inference and coverage-checking algorithms. Table 1 shows the experiment results on the 28 benchmark guide programs. Context-free benchmarks each have two rows in Table 1. The top row is the version where all guides in the

Table 2. Guide types of the 28 benchmarks. The notation τ^d expands to $\tau \wedge \cdots \wedge \tau$ with d many τ's. The functional type $\mathsf{tensor}(\tau; [d_1, \ldots, d_n])$ denotes a tensor of the element type τ and dimensions $[d_1, \ldots, d_n]$. The functional type $\mathsf{simplex}[d]$ denotes a d-dimensional simplex.

Program	Guide Types
`branching`	$\mathbb{N} \wedge (\mathbf{1} \,\&\, \mathbb{N} \wedge \mathbf{1})$
`coordination`	$2 \wedge 2 \wedge \mathbf{1}$
`drill`	$\mathbb{N}_3 \wedge \mathbf{1}$
`ex-1`	$\mathbb{R}_+ \wedge (\mathbf{1} \,\&\, \mathbb{R}_{(0,1)} \wedge \mathbf{1})$
`gaussian`	$\mathbb{R} \wedge \mathbf{1}$
`gbm`	$\mathbb{R} \wedge \mathbf{1}$
`gda`	$\mathsf{tensor}(\mathbb{R}; [3])^2 \wedge \mathsf{tensor}(\mathbb{R}; [2]) \wedge \mathbf{1}$
`gmm`	$\mathsf{simplex}[3] \wedge \mathsf{tensor}(\mathbb{R}; [2; 2])^6 \wedge \mathbb{N}_3^{100} \wedge \mathbf{1}$
`grw`	$\mathbb{R} \wedge \mathbb{R}_+ \wedge \mathbf{1}$
`hmm`	$\mathbb{N}_3^{17} \wedge \mathbf{1}$
`kalman`	$\mathsf{tensor}(\mathbb{R}; [2])^{101} \wedge \mathbf{1}$
`kalman-chaos`	$\mathbb{R}_{(0,1)}^2 \wedge \mathbb{R}^{153} \wedge \mathbf{1}$
`lr`	$\mathbb{R}^3 \wedge \mathbb{R}_+ \wedge \mathbf{1}$
`run-factory`	$\mathbb{R}_{(0,1)} \wedge \mathbf{1}$
`scientist`	$\mathbb{R} \wedge \mathbb{R}_{(0,1)}^7 \wedge \mathbf{1}$
`seq`	$\mathbb{R}^2 \wedge \mathbf{1}$
`sprinkler`	$2^2 \wedge \mathbf{1}$
`user-behavior`	$\mathbb{N}^2 \wedge 2^6 \wedge \mathbf{1}$
`vae`	$\mathsf{tensor}(\mathbb{R}; [50])^{256} \wedge \mathbf{1}$
`weight`	$\mathbb{R} \wedge \mathbf{1}$
`aircraft`	$\mathbb{N} \wedge T_1[\mathbf{1}]$ with $T_1[X] \coloneqq (\mathbb{R} \wedge \mathbb{N} \wedge T_2[T_1[X]]) \,\&\, \mathbf{1}$ and $T_2[X] \coloneqq (\mathbb{R} \wedge T_2[X]) \,\&\, \mathbf{1}$
`iter`	$T[\mathbf{1}]$ with $T[X] \coloneqq \mathbf{1} \,\&\, (\mathbb{R} \wedge T[X])$
`marsaglia`	$T[\mathbf{1}]$ with $T[X] \coloneqq \mathbb{R}_{(0,1)} \wedge \mathbb{R}_{(0,1)} \wedge (\mathbf{1} \,\&\, T[X])$
`ptrace`	$T[\mathbf{1}]$ with $T[X] \coloneqq \mathbb{R}_{(0,1)} \wedge (\mathbf{1} \,\&\, T[X])$
`ex-2`	$T_1[\mathbf{1}]$ with $T_1[X] \coloneqq \mathbb{R}_{(0,1)} \wedge T_2[X]$ and $T_2[X] \coloneqq \mathbb{R}_{(0,1)} \wedge ((\mathbb{R}_+ \wedge \mathbf{1}) \,\&\, T_2[T_2[X]])$
`diter`	$T[\mathbf{1}]$ with $T[X] \coloneqq \mathbf{1} \,\&\, \mathbb{R} \wedge T[T[X]]$
`gp-dsl`	$T[\mathbf{1}]$ with $T[X] \coloneqq 2 \wedge ((\mathbb{N}_3 \wedge ((\mathbb{R}_+ \wedge T[T[X]]) \,\&\, T[T[X]])) \,\&\, (\mathbb{N}_5 \wedge (\mathbb{R}_+ \wedge \mathbb{R}_+ \wedge \mathbf{1} \,\&\, \mathbb{R}_+ \wedge \mathbf{1})))$
`recur`	$T[\mathbf{1}]$ with $T[X] \coloneqq \mathbf{1} \,\&\, (\mathbb{R} \wedge T[\mathbb{R} \wedge T[\mathbb{R} \wedge T[\mathbf{1}]]])$

composition have the aligned code structure with respect to procedure call sites. The bottom row is where the guides have misaligned code structures.

In the Guide Type column, "Finite" refers to non-recursive guide types; e.g., $A \coloneqq \mathbb{N} \wedge (\mathbf{1} \,\&\, (\mathbb{N} \wedge \mathbf{1}))$ in the benchmark `branching`. "Regular" refers to regular recursive guide types; e.g., $A \coloneqq \mathbf{1} \,\&\, (\mathbb{R} \wedge A)$ in the benchmark `iter`. "Context-free" refers to infinite-state context-free guide types; e.g., $T[X] \coloneqq \mathbb{R}_{(0,1)} \wedge ((\mathbb{R}_+ \wedge X) \,\&\, T[T[X]])$ in the benchmark `ex-2`. The LOC column states the number of lines of code. The Type Inference columns show (i) the running time of type inference and (ii) whether type-equality constraints generated during type inference can be verified using syntactic type-equality checking from Wang et al. [52]. The Cov. Check columns show (i) the output (True Pos. or False Neg.) for fully covered sequential compositions of guides, (ii) the output (True Neg. or False Pos.) for uncovered sequential compositions, and (iii) the total running time of checking the coverage of both the fully covered and uncovered sequential compositions.

For type inference, our algorithm successfully infers guide types for all benchmarks. Generally, more lines of code in a benchmark lead to longer time for type inference. This is because the type-inference algorithm traverses the source code to construct typing trees. For the eight regular recursive and context-free benchmarks, the prior work [52] fails in type inference because syntactic equality checking cannot verify the type-equality constraints generated by these benchmarks.

Table 3. Language features supported by various verification methods for checking model-guide compatibility.

Language Feature	Trace Types [31]	Guide Types [52]	Fidelio [32]	This Work
General branching	✗	✓	✓	✓
General recursion	✗	✓	✓	✓
Reorder variables	✓	✗	✓	✗
Sequentially compose guides	✓	✗	✗	✓
Reuse old samples	✗	✗	✗	✓
Structural type equality	✓	✗	✗	✓

For coverage checking, our algorithm successfully verifies the full coverage of all non-recursive and regular recursive benchmarks. For context-free benchmarks, we make use of the heuristic C:Call:Heuristic (§5.3). If all guides in a sequential composition have the same code structure with respect to procedure call sites, our algorithm with the heuristic C:Call:Heuristic (§5.3) can handle it. However, if the guides have misaligned code structures, the heuristic fails, terminating and returning an error message. Without this heuristic for context-free types, the algorithm would run forever in the context-free benchmarks. Because our coverage-checking algorithm is sound (Theorem 5.1), it returns True Neg. for all cases of uncovered sequential compositions.

7 Related Work

Model-guide compatibility in programmable inference. Lee et al. [30] are one of the first to develop static analyses for the model-guide compatibility (i.e., the model and guide have the same set of random variables in all execution paths) in programmable Bayesian inference. Trace types [31] characterize the space of possible execution traces. If the model and guide have equal trace types, they are guaranteed to satisfy (mutual) absolute continuity. Trace types can handle programs where execution paths may yield different sets of random variables. However, trace types do not support general (i.e., support-altering and deterministic) branching and recursion, but only stochastic ones. To address this limitation, Wang et al. [52] design a coroutine-based framework where models and guides communicate by passing messages as prescribed by guide types. Li et al. [32] study automatic generation of guide programs for deep amortized inference. They extend trace types [31] with powerful tree structures and checkpoints for recording branch conditions, thereby enabling expressive constructs such general branching, recursion, and variable reordering.

Our work considers sequential compositions of guides where each guide can choose between drawing fresh samples and reusing old samples. This is a more general setting than most of the aforementioned prior works [30–32, 52]. While trace types [31] offer a combinator for sequential composition and their guide programs can take previous traces as input, their approach does not support recursion or general branching. Our work verifies model-guide support match of sequentially composed guides with rich control-flow structures by combining novel type system techniques (§3.3 and §4) with an efficient coverage-checking algorithm (§5.3). Table 3 summarizes the comparison between the prior and present works on verifying the model-guide compatibility.

PPL verification. Tassarotti and Tristan [45] develop a formally verified compiler ProbCompCert for a fragment of the Stan PPL [6]. Instead of verifying PPL implementations, we focus on the verification of programmable inference where guide coroutines are sequentially composed.

Session types. Guide types are inspired by session types. Originally proposed by Honda [23], session types describe communication protocols of message-passing concurrent programs [5, 43, 49, 51]. Context-free session types [47] extend regular session types with sequential composition. Nested session types [13] extend session types with prenex polymorphism. Type-equality checking of context-free types is impractical due to it being EXPTIME-hard [27]. To make context-free

types practical, Padovani [36, 37] proposes a type-inference algorithm that leverages user-provided code annotations. Almeida et al. [1] implement a type-equality checking algorithm for context-free session types. Parameterized algebraic protocols [35] adopt the nominal and iso-recursive interpretation of context-free and nested session types, thereby achieving linear-time type checking.

Although guide types build on session types, they have a key difference. For guide programs to be sensible, guide types must have finite norms, while session types may have infinite norms. This difference allows guide types to admit practical type-equality checking algorithms (§4).

We could reuse the type-equality checking algorithm for context-free session types by Almeida et al. [1] because context-free types (with possibly infinite norms) are a generalization of guide types (with finite norms). However, because Almeida et al. [1] targets context-free session types, its algorithm has a different design from the algorithm in Hirshfeld and Moller [22], which specifically targets finite-norm context-free processes and is implemented in our prototype. Also, the worst-case complexity of the algorithm by Almeida et al. [1] is theoretically unknown in the setting of guide types. A key contribution of this article is to show that it is possible to decide structural type equality of guide types in polynomial time, and we do not intend to argue that a particular type-equality checking algorithm is superior to others.

Composable probabilistic inference. Many PPLs support rich compositional frameworks for programmable probabilistic inference [3, 4, 12, 24, 44, 50], including custom proposals for MCMC. These works do not study the problem of verifying or guaranteeing the correctness of custom user-written proposals (i.e., model-guide compatibility), which is the central focus of our work.

8 Conclusion

This article has presented a coroutine-based programmable inference framework for sequential compositions of guide programs where each guide can access and reuse old samples. By translating guide types to context-free processes with finite norms, we show that the structural type equality of guide types is decidable in polynomial time. This enables efficient type inference and type-equality checking between the model and guides, which is a key soundness ingredient for the multiple-block MH (BMH) algorithm. We also present a coverage-checking algorithm that verifies that sequentially composed guides freshly samples all random variables, another key soundness ingredient of BMH. We have implemented and evaluated a type-inference algorithm with structural type equality and a coverage-checking algorithm, demonstrating their expressiveness and practicality.

Data-Availability Statement

The artifact [39] for this paper is available at doi:10.5281/zenodo.12669572.

Acknowledgments

The authors wish to thank the anonymous referees for their valuable comments and helpful suggestions. This material is based upon work supported by the National Science Foundation under Grant Nos. 2311983, 2007784, and 1845514. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. 2020. Deciding the Bisimilarity of Context-Free Session Types. In *Proceeding of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Armin Biere and David Parker (Eds.). Springer, Cham, 39–56. https://doi.org/10.1007/978-3-030-45237-7_3

[2] Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). 2020. *Foundations of Probabilistic Programming.* Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/9781108770750

[3] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 1 (Jan. 2019), 973–978.

[4] Keith A. Bonawitz. 2008. *Composable Probabilistic Inference with Blaise.* Ph. D. Dissertation. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/41887

[5] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory*. Springer, Berlin, Heidelberg, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

[6] 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 (Jan. 2017), 1–32. Issue 1. https://doi.org/10.18637/jss.v076.i01

[7] Nick Chater, Joshua B. Tenenbaum, and Alan Yuille. 2006. Probabilistic Models of Cognition: Conceptual Foundations. *Trends in Cognitive Sciences* 10, 7 (July 2006), 287–291. https://doi.org/10.1016/j.tics.2006.05.007

[8] Siddhartha Chib. 2001. Markov Chain Monte Carlo Methods: Computation and Inference. In *Handbook of Econometrics*, James J. Heckman and Edward Leamer (Eds.). Vol. 5. Elsevier, Amsterdam, Chapter 57, 3569–3649. https://doi.org/10.1016/S1573-4412(01)05010-3

[9] Siddhartha Chib and Edward Greenberg. 1995. Understanding the Metropolis-Hastings Algorithm. *The American Statistician* 49, 4 (1995), 327–335. http://www.jstor.org/stable/2684568

[10] S. Christensen, H. Huttel, and C. Stirling. 1995. Bisimulation Equivalence Is Decidable for All Context-Free Processes. *Information and Computation* 121, 2 (Sept. 1995), 143–148. https://doi.org/10.1006/inco.1995.1129

[11] Marco F. Cusumano-Towner. 2020. *Gen: A High-Level Programming Platform for Probabilistic Inference.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[12] 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 *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 221–236. https://doi.org/10.1145/3314221.3314642

[13] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. 2021. Nested Session Types. In *Proceedings of the 30th European Symposium on Programming*. Springer, Cham, 178–206. https://doi.org/10.1007/978-3-030-72019-3_7

[14] Joost Engelfriet. 2015. Tree Automata and Tree Grammars. arXiv:1510.02036 [cs]

[15] Simon Gay and Malcolm Hole. 2005. Subtyping for Session Types in the Pi Calculus. *Acta Informatica* 42, 2 (Nov. 2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

[16] Ferenc Gécseg and Magnus Steinby. 2015. Tree Automata. arXiv:1509.06233 [cs]

[17] Alan E. Gelfand. 2000. Gibbs Sampling. *J. Amer. Statist. Assoc.* 95, 452 (Dec. 2000), 1300–1304. https://doi.org/10.1080/01621459.2000.10474335

[18] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith A. Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*. AUAI Press, Arlington, VA, USA, 220–229. https://doi.org/10.5555/3023476.3023503

[19] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *Future of Software Engineering Proceedings*. Association for Computing Machinery, New York, NY, USA, 161–181. https://doi.org/10.1145/2593882.2593900

[20] W. K. Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57 (April 1970), 97–109. Issue 1. https://doi.org/10.1093/biomet/57.1.97

[21] Y. Hirshfeld, M. Jerrum, and F. Moller. 1994. A Polynomial-Time Algorithm for Deciding Equivalence of Normed Context-Free Processes. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. IEEE Press, Piscantaway, NJ, USA, 623–631. https://doi.org/10.1109/SFCS.1994.365729

[22] Yoram Hirshfeld and Faron Moller. 1994. A Fast Algorithm for Deciding Bisimilarity of Normed Context-Free Processes. In *Proceedings of the 5th International Conference on Concurrency Theory*. Springer, Berlin, Heidelberg, 48–63. https://doi.org/10.1007/978-3-540-48654-1_5

[23] Kohei Honda. 1993. Types for Dyadic Interaction. In *Proceedings of the 4th International Conference on Concurrency Theory*. Springer, Berlin, Heidelberg, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[24] Daniel E Huang. 2017. *On Programming Languages for Probabilistic Modeling.* Ph. D. Dissertation. Harvard University. https://dash.harvard.edu/handle/1/40046525

[25] Petr Jancar. 2013. Bisimilarity on Basic Process Algebra Is in 2-ExpTime (an Explicit Proof). *Logical Methods in Computer Science* 9, 1 (March 2013), 10. https://doi.org/10.2168/LMCS-9(1:10)2013

[26] F. Jelinek, J. D. Lafferty, and R. L. Mercer. 1992. Basic Methods of Probabilistic Context Free Grammars. In *Speech Recognition and Understanding*, Pietro Laface and Renato Mori (Eds.). Springer, Berlin, Heidelberg, 345–360. https://doi.org/10.1007/978-3-642-76626-8_35

[27] Stefan Kiefer. 2013. BPA Bisimilarity Is EXPTIME-hard. *Inform. Process. Lett.* 113, 4 (Feb. 2013), 101–106. https://doi.org/10.1016/j.ipl.2012.12.004

[28] Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash K. Mansinghka. 2015. Picture: A Probabilistic Programming Language for Scene Perception. In *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE Press, Piscantaway, NJ, USA, 4390–4399. https://doi.org/10.1109/CVPR.2015.7299068

[29] Sławomir Lasota and Wojciech Rytter. 2006. Faster Algorithm for Bisimulation Equivalence of Normed Context-Free Processes. In *Proceedings of the 31st International Symposium on Mathematical Foundations of Computer Science*. Springer, Berlin, Heidelberg, 646–657. https://doi.org/10.1007/11821069_56

[30] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2019. Towards Verified Stochastic Variational Inference for Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 16 (December 2019), 33 pages. https://doi.org/10.1145/3371084

[31] 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. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 19 (December 2019), 32 pages. https://doi.org/10.1145/3371087

[32] Jianlin Li, Leni Ven, Pengyuan Shi, and Yizhou Zhang. 2023. Type-Preserving, Dependence-Aware Guide Generation for Sound, Effective Amortized Probabilistic Inference. *Proceedings of the ACM on Programming Languages* 7, POPL, Article 50 (January 2023), 29 pages. https://doi.org/10.1145/3571243

[33] Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin C. Rinard. 2018. Probabilistic Programming with Programmable Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 603–616. https://doi.org/10.1145/3192366.3192409

[34] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics* 21, 6 (June 1953), 1087–1092. https://doi.org/10.1063/1.1699114

[35] Andreia Mordido, Janek Spaderna, Peter Thiemann, and Vasco T. Vasconcelos. 2023. Parameterized Algebraic Protocols. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 163 (June 2023), 25 pages. https://doi.org/10.1145/3591277

[36] Luca Padovani. 2017. Context-Free Session Type Inference. In *Proceedings of the 26th European Symposium on Programming*. Springer, Berlin, Heidelberg, 804–830. https://doi.org/10.1007/978-3-662-54434-1_30

[37] Luca Padovani. 2019. Context-Free Session Type Inference. *ACM Transactions on Programming Languages and Systems* 41, 2, Article 9 (March 2019), 37 pages. https://doi.org/10.1145/3229062

[38] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. 2005. A Probabilistic Language based upon Sampling Functions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 171–182. https://doi.org/10.1145/1040305.1040320

[39] Long Pham, Di Wang, Feras Saad, and Jan Hoffmann. 2024. *Artifact for Programmable MCMC with Soundly Composed Guide Programs*. Zenodo. https://doi.org/10.5281/zenodo.12669572

[40] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.

[41] Gareth O. Roberts and Jeffrey S. Rosenthal. 2006. Harris Recurrence of Metropolis-within-Gibbs and Trans-Dimensional Markov Chains. *The Annals of Applied Probability* 16, 4 (Nov. 2006), 2123–2139. https://doi.org/10.1214/105051606000000510

[42] 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. *Proceedings of the ACM on Programming Languages* 3, POPL (January 2019), 32 pages. https://doi.org/10.1145/3290350

[43] Alceste Scalas and Nobuko Yoshida. 2019. Less Is More: Multiparty Session Types Revisited. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 30 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290343

[44] Sam Stites, Heiko Zimmermann, Hao Wu, Eli Sennesh, and Jan-Willem van de Meent. 2021. Learning Proposals for Probabilistic Programs with Inference Combinators. In *Proceedings of the 37th Conference on Uncertainty in Artificial Intelligence*. PMLR, Norfolk, MA, USA, 1056–1066.

[45] Joseph Tassarotti and Jean-Baptiste Tristan. 2023. Verified Density Compilation for a Probabilistic Programming Language. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 131 (June 2023), 22 pages. https://doi.org/10.1145/3591245

[46] Pyro Development Team. 2023. Getting Started With Pyro: Tutorials, How-to Guides and Examples — Pyro Tutorials 1.8.6 Documentation. https://pyro.ai/examples/index.html

[47] Peter Thiemann and Vasco T. Vasconcelos. 2016. Context-Free Session Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, New York, NY, USA, 462–475. https://doi.org/10.1145/2951913.2951926

[48] Luke Tierney. 1994. Markov Chains for Exploring Posterior Distributions. *The Annals of Statistics* 22 (Dec. 1994), 1701–1728. Issue 4. https://doi.org/10.1214/aos/1176325750

[49] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Proceedings of the 22nd European Symposium on Programming*. Springer, Berlin, Heidelberg, 350–369. https://doi.org/10.1007/978-3-642-37036-6_20

[50] Dustin Tran. 2020. *Probabilistic Programming for Deep Learning*. Ph. D. Dissertation. Columbia University. https://doi.org/10.7916/d8-95c9-sj96

[51] Philip Wadler. 2012. Propositions as Sessions. In *Proceedings of the 17th ACM SIGPLAN international Conference on Functional Programming*. Association for Computing Machinery, New York, NY, USA, 273–286. https://doi.org/10.1145/2364527.2364568

[52] 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*. Association for Computing Machinery, New York, NY, USA, 788–803. https://doi.org/10.1145/3453483.3454077

[53] Frank Wood, Jan Willem van de Meent, and Vikash K. Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*. PMLR, Norfolk, MA, USA, 1024–1032.

[54] Frank Wood, Jan-Willem van de Meent, David Tolpin, Tuan Anh Le, Brooks Paige, Yuav Perov, Tom Rainforth, and Hongseok Yang. 2023. The Anglican Probabilistic Programming System. https://probprog.github.io/anglican/examples/index.html