Type-Preserving, Dependence-Aware Guide Generation for Sound, Effective Amortized Probabilistic Inference

JIANLIN LI, University of Waterloo, Canada
LENI VEN, University of Waterloo, Canada
PENGYUAN SHI, University of Waterloo, Canada
YIZHOU ZHANG, University of Waterloo, Canada

In probabilistic programming languages (PPLs), a critical step in optimization-based inference methods is constructing, for a given *model* program, a trainable *guide* program. Soundness and effectiveness of inference rely on constructing good guides, but the expressive power of a universal PPL poses challenges. This paper introduces an approach to automatically generating guides for deep amortized inference in a universal PPL. Guides are generated using a type-directed translation per a novel behavioral type system. Guide generation extracts and exploits independence structures using a syntactic approach to conditional independence, with a semantic account left to further work. Despite the control-flow expressiveness allowed by the universal PPL, generated guides are guaranteed to satisfy a critical soundness condition and, moreover, consistently improve training and inference over state-of-the-art baselines for a suite of benchmarks.

CCS Concepts: • **Theory of computation → Probabilistic computation**; **Program semantics**; **Program reasoning**; **Type theory**; • **Mathematics of computing → Bayesian computation**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: Probabilistic programming, amortized inference, type systems, guide generation.

ACM Reference Format:
Jianlin Li, Leni Ven, Pengyuan Shi, and Yizhou Zhang. 2023. Type-Preserving, Dependence-Aware Guide Generation for Sound, Effective Amortized Probabilistic Inference. *Proc. ACM Program. Lang.* 7, POPL, Article 50 (January 2023), 29 pages. https://doi.org/10.1145/3571243

1 INTRODUCTION

A Bayesian probabilistic model denotes a distribution $p(\mathbf{Z}, \mathbf{X})$, where random variables $\mathbf{Z}$ are called *latent* and $\mathbf{X}$ *observed*. Bayesian inference is aimed at calculating $p(\mathbf{Z}|\mathbf{X} = \mathbf{x})$, the posterior distribution of latents $\mathbf{Z}$ given observed data $\mathbf{x}$.

Probabilistic programming languages (PPLs) are powerful tools for modeling Bayesian-inference problems. *Universal PPLs*, in particular, offer linguistic features including stochastic branching and general recursion, making it possible to express mixture models, probabilistic context-free grammars, kernel induction models [Le et al. 2019; Saad et al. 2019], and so forth.

PPLs also provide inference methods for solving inference problems. *Optimization-based* methods, such as variational inference [Jordan et al. 1999], are taking hold in some PPLs empowered by deep neural networks [Bingham et al. 2019; Tran et al. 2018]. They consist in approximating the true posterior $p(\mathbf{Z}|\mathbf{X} = \mathbf{x})$ of a *model* program using a neural *guide* program $q_\phi(\mathbf{Z})$ where ϕ stands for

Authors' addresses: School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, N2L 3G1, Canada.

2475-1421/2023/1-ART50
https://doi.org/10.1145/3571243

neural-network parameters: they optimize an objective function via gradient descent, searching for ϕ that makes $q_\phi(\mathbf{Z})$ close to $p(\mathbf{Z}|\mathbf{X} = \mathbf{x})$. Notice that guide programs must be free of observed random variables.

Deep amortized inference [Paige and Wood 2016; Ritchie et al. 2016; Le et al. 2017; van de Meent et al. 2021, §8] is based on optimization and profits from learning amortized guides. Amortization is an ingrained idea in machine learning literature [Hinton et al. 1995; Kingma and Welling 2014]. Rather than training neural networks every time a new observation is given, amortized inference trains *ahead of time* a guide program $q_\phi(\mathbf{Z}; \mathbf{x})$ that takes an observation $\mathbf{x}$ as input. Then, at *run time*, inferring $p(\mathbf{Z}|\mathbf{X} = \mathbf{x}_0)$ for actual observations $\mathbf{x}_0$ is cheap using $q_\phi(\mathbf{Z}; \mathbf{x}_0)$, amortizing the upfront cost of training ϕ. Optimization and amortization help inference scale for probabilistic programs that need to be applied repeatedly to different observations.

Constructing good guides is critical. Ahead-of-time training and run-time inference are (1) sound only when model–guide pairs satisfy *absolute continuity* and (2) are most effective when guides are *faithful* yet *parsimonious* in terms of the conditional dependences they encode:

(1) Absolute continuity, informally speaking, is a soundness condition requiring two distributions to have compatible supports. A guide with an incompatible support causes inference to crash or produce incorrect results [Lee et al. 2019].

(2) Faithful guides do not contain conditional independences not found in models—missing correlations make guides unable to express true posteriors even if they had unbounded neural-network capacities (i.e., number of trainable parameters). Parsimonious guides do not encode more conditional dependences than necessary—excessive correlations introduce computational burdens as ahead-of-time training has to unlearn them to uncover true posteriors [Webb et al. 2018].

Mainstream PPLs currently have limited support for emitting guide programs. In Pyro, for example, the autoguide [2022] library works only for non-universal probabilistic programs, does not guarantee absolute continuity, and ignores faithfulness altogether. Creating good guide programs remains a demanding, error-prone task.

This paper introduces an approach to automatically generating guide programs for a universal PPL. Generated guides enjoy strong soundness guarantees and consistently improve training and inference over state-of-the-art approaches, despite the expressiveness allowed by the universal PPL. We first review relevant background in Section 2. We then identify technical challenges and our core contributions in Section 3.

2 BACKGROUND

Bayesian Networks (BN). Absent of branching and recursion, straight-line probabilistic programs are essentially BNs [Pearl 1988]. An example is model in Figure 1. In the BN next to it, nodes $\boxed{\text{a}}$, $\boxed{\text{b}}$, and $\boxed{\text{obs}}$ are variables in the model. Shaded node $\boxed{\text{observe(...)}}$ stands for the observe statement. Edges signal dataflow: data in a, b, and obs flow to the observe statement.

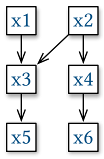
a and b are *conditionally dependent* on each other, because the observe statement *conditions* the model on a+b being close to obs. This correlation is visualized by the ground-truth plot labeled $p(\text{a}, \text{b}|\text{obs})$, which is obtained by the asymptotically exact inference method MCMC.

BN Active trails. Conditional independence can sometimes be read off from BNs. This reasoning is via a notion of active trails [Pearl 1988]. Let $X_1, X_2, ..., X_n$ be a *trail* (i.e., an undirected, acyclic path) in a BN. The trail is *active given* a (possibly empty) set $\mathbf{Z}$ of nodes in the BN when

(i) for any chain $X_{i-1} \to X_i \leftarrow X_{i+1}$ (called a *collider*), X_i or one of its descendants is either in $\mathbf{Z}$ or a conditioning node (namely $\boxed{\text{observe(...)}}$); and
(ii) no other nodes in the trail is in $\mathbf{Z}$ or is a conditioning node.

Whenever there is no active trail between any $X \in \mathbf{X}$ and $Y \in \mathbf{Y}$ given $\mathbf{Z}$, we have that $\mathbf{X}$ and $\mathbf{Y}$ are independent given $\mathbf{Z}$ (notated $\mathbf{X} \perp\!\!\!\perp \mathbf{Y} \mid \mathbf{Z}$).

Consider the BN to the right. We can assert x3 $\perp\!\!\!\perp$ x4 | x2: knowing x2 falsifies condition (ii), deactivating the trail $\boxed{\text{x3}} \longleftarrow \boxed{\text{x2}} \longrightarrow \boxed{\text{x4}}$. Similarly, we have x1 $\perp\!\!\!\perp$ x4 | x2. But we cannot assert x1 $\perp\!\!\!\perp$ x4 | x3 or x1 $\perp\!\!\!\perp$ x4 | x5 due to condition (i): knowing x3 or its descendant x5 activates the collider $\boxed{\text{x1}} \longrightarrow \boxed{\text{x3}} \longleftarrow \boxed{\text{x2}}$. In Figure 1, we cannot assert a $\perp\!\!\!\perp$ b: the collider $\boxed{\text{a}} \longrightarrow \boxed{\text{observe(...)}} \longleftarrow \boxed{\text{b}}$ is active because of the conditioning.



$$\begin{tikzcd} \boxed{\text{x1}} \arrow[d] & \boxed{\text{x2}} \arrow[d] \arrow[dl] \\ \boxed{\text{x3}} \arrow[d] & \boxed{\text{x4}} \arrow[d] \\ \boxed{\text{x5}} & \boxed{\text{x6}} \end{tikzcd}$$

Traces and universal probabilistic programming. A *universal PPL* equips a Turing-complete language with constructs for sampling and conditioning [Goodman et al. 2008]. It supports linguistic features including stochastic branching and general recursion, so it allows a stochastic set of random variables (RVs) to be sampled every time a program is run. BNs assume a fixed set of RVs, so in general, BNs cannot express programs in universal PPLs.

Traces are a common semantic notion underlying many universal PPLs [Wood et al. 2014; Siddharth et al. 2017; Bingham et al. 2019; Mansinghka et al. 2018; Cusumano-Towner et al. 2019; Tran et al. 2018]. Running a probabilistic program generates a trace: the trace records RVs and their values sampled in that run. The semantics of a program is characterized by a distribution over the measure space of all possible traces.

Addresses are unique names identifying RVs in a trace. Consider the recursive function `pcfg` (written in Pyro) in Figure 2a, which models a probabilistic context-free grammar. A possible trace of this program is

$$[\texttt{"\_A"}: .7, \texttt{"\_D1\_A"}: .3, \texttt{"\_D1\_C"}: 2, \texttt{"\_D2\_A"}: .1, \texttt{"\_D2\_C"}: 9], \tag{2.1}$$

which will result in the expression `Add(Const(2), Const(9))` in the context-free language.

Deep amortized inference. The ahead-of-time training stage of amortized inference searches for ϕ that makes $q_\phi(\mathbf{Z}; \mathbf{x})$ similar to $p(\mathbf{Z}|\mathbf{X} = \mathbf{x})$ on average for most $\mathbf{x}$. More precisely, it solves the following optimization problem:

$$\operatorname{argmin}_\phi \ \mathbb{E}_{\mathbf{x} \sim p(\mathbf{X})} \left[\mathrm{KL}\big(p(\mathbf{Z}|\mathbf{x}) \,\big\|\, q_\phi(\mathbf{Z}; \mathbf{x})\big) \right], \tag{2.2}$$

where $p(\mathbf{Z}|\mathbf{x})$ abbreviates $p(\mathbf{Z}|\mathbf{X}{=}\mathbf{x})$, and the Kullback–Leibler divergence quantifies how similar two distributions are. Objective (2.2) can be simplified to

$$\operatorname{argmin}_\phi \ \mathbb{E}_{\mathbf{z},\mathbf{x} \sim p(\mathbf{Z},\mathbf{X})} \left[-\log q_\phi(\mathbf{z}; \mathbf{x}) \right], \tag{2.3}$$

which is convenient: training data $\mathbf{z}, \mathbf{x} \sim p(\mathbf{Z}, \mathbf{X})$ can be generated simply by running the model program $p(\mathbf{Z}, \mathbf{X})$ repeatedly. We note that a variant of amortized inference [Ritchie et al. 2016] optimizes a different objective, known as the evidence lower bound (ELBo):

$$\operatorname{argmax}_\phi \sum_{\mathbf{x}_i} \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{Z}; \mathbf{x}_i)} \left[\log p(\mathbf{z}, \mathbf{x}_i) - \log q_\phi(\mathbf{z}; \mathbf{x}_i) \right]. \tag{2.4}$$

Objective (2.4) requires a training dataset $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ be provided, as indicated by the summation over $\mathbf{x}_i$ in (2.4). In this paper, we focus on the first type of amortized inference that uses objective (2.3), but our technical contributions can in principle be applied to the ELBo objective (2.4).

The run-time inference stage performs importance sampling (IS), per the equation

$$p(\mathbf{Z}|\mathbf{x}_0) = p(\mathbf{Z}, \mathbf{x}_0) \Big/ \mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{Z}; \mathbf{x}_0)} \left[\frac{p(\mathbf{z}, \mathbf{x}_0)}{q_\phi(\mathbf{z}; \mathbf{x}_0)} \right]. \tag{2.5}$$

Given observation $\mathbf{x}_0$, IS approximates $p(\mathbf{Z}|\mathbf{x}_0)$ by sampling $\mathbf{z}$ from the trained *proposal* distribution $q_\phi(\mathbf{Z}; \mathbf{x}_0)$ and weighting samples $\mathbf{z}$ by their importance ratio $p(\mathbf{z}, \mathbf{x}_0)/q_\phi(\mathbf{z}; \mathbf{x}_0)$.

3 PROBLEMS AND CONTRIBUTIONS

3.1 Absolute Continuity (AC)

Crucially, soundness of both training and inference relies on AC. First, per formula (2.5), IS is unbiased only when $p(\mathbf{Z}, \mathbf{x}_0)$ is *absolutely continuous* with respect to $q_\phi(\mathbf{Z}; \mathbf{x}_0)$, written as $p(\mathbf{Z}, \mathbf{x}_0) \ll q_\phi(\mathbf{Z}; \mathbf{x}_0)$. That is, in a trace-based PPL, for samples from $q_\phi(\mathbf{Z}; \mathbf{x}_0)$ to asymptotically approximate the posterior, $q_\phi(\mathbf{Z}; \mathbf{x}_0)$ is required to have non-null probability on any measurable set of traces on which $p(\mathbf{Z}, \mathbf{x}_0)$ has non-null probability. Second, objective (2.3) is defined only when $p(\mathbf{Z}, \mathbf{x}) \ll q_\phi(\mathbf{Z}; \mathbf{x})$, lest logarithm-of-zero errors.[1] Furthermore, the ELBo objective (2.4) is defined only when AC holds in the other direction: $q_\phi(\mathbf{Z}; \mathbf{x}_i) \ll p(\mathbf{Z}, \mathbf{x}_i)$. Indeed, AC is a pervasive soundness condition required by many inference methods (e.g., MCMC).

It is simple to verify AC when probabilistic programs can be described as BNs: it suffices to check that two programs contain the same set of RVs. For example, in Figure 1, model is absolutely continuous with respect to both guides, even though the guides reverse the order a and b are sampled. All three programs have the same support made up of two RVs: a over $\mathbb{R}$ and b over $\mathbb{R}_+$.

Control-flow expressiveness of universal PPLs pose challenges to a formal argument, however: branching and recursion allow a stochastic set of RVs to be sampled every time a program is run. Consider the guide in Figure 2b, manually created for the PCFG model in Figure 2a. It might appear that the guide is compatible with the model: half the time it samples a leaf node of the syntax tree, and the other half the time it recursively generates subtrees. But it has two sources of unsoundness:

- Its branching condition `a > .5`, different from the model's, causes the guide to execute different branches—and thus sample different RVs—than the model. The support mismatch causes accesses to unavailable addresses, crashing a typical inference engine. Notice, however, that reordering of the subtrees by the guide does not change the distribution's support and is thus not unsoundness.
- Second, RVs at addresses `L + "_C"` have a smaller support ($\mathbb{R}_+$) than they do in the model ($\mathbb{R}$). The mismatch can lead to logarithm-of-zero errors, NaNs, or biased estimates.

The run-time errors are frustrating when they happen deep into the gradient-descent loop. Even worse is when programs return normally but with incorrect results.

State-of-the-art approaches. The challenge has led to recent work on type systems for ensuring AC statically [Lew et al. 2019; Wang et al. 2021]: a model and a guide are guaranteed to satisfy AC if they have compatible types. Unfortunately, these systems are not geared to automatic guide generation, and more critically, they restrict expressible models or guides. Lew et al.'s type system supports only limited forms of branching and loops, and it disallows general recursion. Wang et al. address specifically these limitations, but their coroutine-based semantics requires a guide to perform computations in exactly the same order as its model does (thus failing to type-check trivially sound guides such as guide1 and guide2 in Figure 1).

These restrictions have implications to modeling power and to performance. Without general recursion, it is difficult to express useful models including probabilistic context-free grammars [Manning and Schütze 1999]. Without the possibility to reorder RVs, faithful guides may require denser dependence structures than otherwise avoidable (see Figure 16, for example), which can reduce training performance as we show later.

3.2 Faithfulness and Parsimony

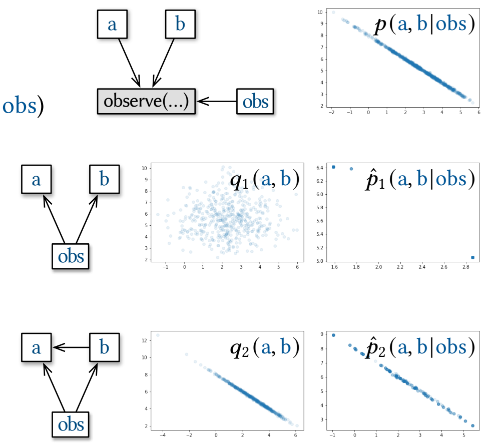
Faithfulness matters. Effectiveness of IS is highly dependent on having good proposals. In amortized inference, IS yields low-variance estimates if ahead-of-time training makes the proposal distribution $q_\phi(\mathbf{Z}; \mathbf{x}_0)$ similar in shape to the true posterior $p(\mathbf{Z}|\mathbf{x}_0)$. But an unfaithful guide could not represent

[1] Gradient descent further requires differentiability, which is studied in other work (e.g., Mak et al. [2021]).

```
def model (obs) =
  a = sample(Normal(0; 2))
  b = sample(Gamma(8; 2))
  observe(Normal(a+b, 0.05); obs)

def guide1 (obs) =
  x1, x2 = NNb(obs)
  b = sample(Gamma(x1; x2))
  y1, y2 = NNa(obs)
  a = sample(Normal(y1; y2))

def guide2 (obs) =
  x1, x2 = NNb(obs)
  b = sample(Gamma(x1; x2))
  y1, y2 = NNa(obs, b)
  a = sample(Normal(y1; y2))
```



$p(a, b|obs)$: true posterior, approximated via MCMC

$q_1(a, b)$ and $q_2(a, b)$: samples from trained guide1 and guide2 for a given obs

$\hat{p}_1(a, b|obs)$ and $\hat{p}_2(a, b|obs)$: posteriors inferred via importance sampling with trained guide1, guide2 as proposals

Figure 1. Soundness: model is (trivially) absolutely continuous with both guide1 and guide2, despite that the guides reorder a and b. Faithfulness: guide2 faithfully captures conditional dependence between a and b given obs, whereas guide1 considers a and b independent given obs. The payoff of being faithful is that guide2 leads to higher-quality posterior samples than guide1.

```
a = pcfg("")                # main program
x = tensorize(a)
sample("obs", Normal(x,1), obs=obs)
```

```
def pcfg(L):  # L is address prefix
  a = sample(L+"_A", Uniform)
  if a < .5:
    c = sample(L+"_C", Normal(0,1))
    b = Const(c)
  else:
    d1 = pcfg(L+"_D1")  # left subtree
    d2 = pcfg(L+"_D2")  # right subtree
    b = Add(d1, d2)
  return b
```

(a) PCFG model, written in Pyro

```
a = pcfg("", obs)                # main program
```

```
def pcfg (L, obs):
  a = sample(L+"_A", Uniform)
  if a > .5:
    C1,C2 = NNC(obs)
    c = sample(L+"_C", Gamma(C1,C2))
    b = Const(c)
  else:
    d2 = pcfg(L+"_D2", obs) # right subtree
    d1 = pcfg(L+"_D1", obs) # left subtree
    b = Add(d1, d2)
  return b
```

(b) PCFG guide, written in Pyro

Figure 2. Implementation of a probabilistic context-free grammar (PCFG). The guide breaks AC as required by formulas (2.2) and (2.5). It is unfaithful, as it introduces conditional independences not found in the model.

the true posterior even if it had infinite neural-network capacities, because it cannot encode some conditional dependences present in the model.

In Figure 1, guide2 faithfully expresses that a and b are conditionally dependent given obs using the highlighted code. By contrast, guide1 considers a and b independent. Thus, after training neural networks NNa and NNb, guide2 can approximate true posteriors much better than guide1, as shown by plots $q_2(a, b)$ and $q_1(a, b)$. In turn, at run time, IS with guide2 leads to much higher-quality samples than IS with guide1, as shown by plots $\hat{p}_2(a, b|obs)$ and $\hat{p}_1(a, b|obs)$.

Parsimony matters too. Parsimonious guides do not encode more conditional dependences than necessary. Given that training budgets and network capacities are finite, it would be wasteful to expend resources on unlearning spurious correlations that could otherwise be ruled out from the get-go. As we show later, parsimonious guides can indeed lead to better convergence.

Notice that parsimony of dependences is not in conflict with overparameterization of neural networks. Overparameterized networks have many more trainable parameters than there are

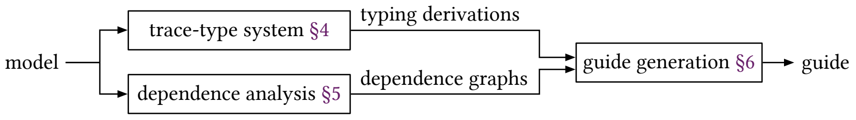


Figure 3. Workflow of the Fidelio guide generator

training examples. It has been observed that they can make training easier while improving generalizability [Zhang et al. 2017]. A parsimonious guide can choose to overparameterize its neural networks.

State-of-the-art approaches. Methods for automating the design of faithful guides exist in the machine learning literature. Webb et al. [2018] construct faithful, parsimonious guides for BN models, but the approach is not applicable to universal PPLs with branching or recursion. For instance, the approach cannot handle the PCFG model in Figure 2a.

In fact, it is not entirely clear how to even manually create a faithful, parsimonious guide for the PCFG model, because of general recursion. Unsoundness aside, the guide in Figure 2b is unfaithful. It is based on the simplifying *mean-field* assumption that RVs are conditionally independent (given obs). Mean-field guides are a standard default in practice for variational inference [Zhang et al. 2019], but as our experiments confirm, they may lead to unsatisfying results for PCFG-like models that exhibit strong correlations between RVs.

To handle branching and recursion, Le et al. [2017] construct guides using advanced recurrent neural networks (RNNs) such as an LSTM, effectively treating any two random variables as correlated. Thus, RNN guides may be non-parsimonious, and so ahead-of-time training must work extra hard to uncover independences. As we also show, RNNs' tendency to forget long-range dependences may weaken its correlation-preserving guarantee for RVs sampled far apart. What is needed of a guide generator is the ability to take into account structures of probabilistic programs.

3.3 Contributions

Figure 3 shows how our guide generator (called Fidelio) works. In Section 4, we design a behavioral trace-type system capturing probabilistic effects and control flow—compatible typing implies absolute continuity. In Section 5, we construct dependence graphs for probabilistic programs (PDGs) and adapt the notion of active trails from BNs to PDGs—active trails suggest possible correlation. Guide generation is then a type-guided, dependence-aware translation (Section 6).

A unique combination of features in the Fidelio semantics enables it to offer soundness guarantees while permitting expressiveness. First, traces are *tree-structured* rather than lists. So a guide can reorder computations that are subtrees of a parent, while generating the same set of traces as its model. Second, trace types are lightweight dependent types that can track control flow through what we call *checkpoint expressions*. So a model is free to use stochastic branching and general recursion, while being well-typed.

We prove that our type system is sound: compatibly typed model–guide pairs satisfy AC (Theorem 4.5). We prove that generated guides preserve typing (Theorem 6.1). As a result, AC is guaranteed by construction (Theorem 6.2). We also show that our type system allows more models and guides to be typed using a collection of programs from literature (Section 8.1).

Incorporating program dependence information in guide generation is new in itself; existing approaches assume either independence or correlation altogether. Our use of PDGs, augmented with hidden states and empowered by the idea of active trails, enables modularly capturing control dependences that cannot otherwise be expressed by BNs. We note that this treatment of conditional independence is syntactic and not proven sound; we leave a semantic account to future work.

§4.1	expressions	e	::=	$\mathsf{x} \mid \mathsf{a} \mid \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid r \mid n \mid \lambda \mathsf{x}.\, e \mid \mathsf{op}_\diamond(e_1, \ldots, e_n) \mid \mathsf{app}(e_1; e_2) \mid \mathsf{let}(e_1; \mathsf{x}.e_2) \mid \mathsf{if}(e; e_1; e_2) \mid D(e_1; \ldots; e_n)$
	distributions	D	::=	$\textsc{Bern} \mid \textsc{Unif} \mid \textsc{Beta} \mid \textsc{Gamma} \mid \textsc{Normal} \mid \textsc{Cat} \mid \textsc{Geo} \mid \textsc{Pois}$
	terms	t	::=	$\mathsf{sample}(e) \mid m \mid \mathsf{ite}(e; m_1; m_2) \mid \mathsf{call}(\mathsf{f}; e_1; e_2)$
	commands	m	::=	$\mathsf{ret}(e) \mid \mathsf{observe}(e_1; e_2);\ m \mid \mathsf{x} = e;\ m \mid \mathsf{a} = t;\ m$
	function definitions	$\mathcal{F}$	::=	$\mathsf{def}\ \mathsf{f}\langle \mathsf{a} \rangle(\mathsf{x}) = m$
§4.2	values	v, d	::=	$\mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid r \mid n \mid \lambda \mathsf{x}.\, e \mid D(v_1; \ldots; v_n)$
	events	s	::=	$\mathsf{atom}\, v \mid \mathsf{trace}\, \sigma \mid \mathsf{inj}\, \sigma \mid \mathsf{fold}\, \sigma$
	traces	σ	::=	$\{\overline{\mathsf{a} : s}\}$
§4.3	data types	τ	::=	$\mathbb{1} \mid \mathbb{2} \mid \mathbb{R}_{[0,1]} \mid \mathbb{R}_+ \mid \mathbb{R} \mid \mathbb{N}_n \mid \mathbb{N} \mid \tau_1 \to \tau_2 \mid \mathsf{dist}(\tau)$
	event types	S	::=	$\mathsf{atom}\, \tau \mid \mathsf{trace}\, \Sigma \mid \Sigma_1 +_e \Sigma_2 \mid \mathsf{F}\langle e \rangle$
	trace types	Σ	::=	$\left\{\overline{\mathsf{a} : S}\right\}_e$
	function signatures	$\mathcal{S}$	::=	$\mathsf{f} : \langle \tau_1 \rangle(\tau_2) \rightsquigarrow \tau_3 \ \#\ \mathsf{F}$
	trace-type definitions	$\mathcal{T}$	::=	$\mathsf{typedef}\ \mathsf{F} = \forall \mathsf{a} : \tau.\, \Sigma$

expression variables	$\mathsf{x}, \mathsf{y}, \mathsf{h}$
term variables	$\mathsf{a}, \mathsf{b}, \mathsf{c}$
function names	f
trace-type names	F

Figure 4. Syntax of the Fidelio core calculus (FCC)

Our evaluation of the dependence analysis focuses on assessing whether it benefits training and inference in practice (Section 8.2). Experiments show that being dependence-aware in guide generation pays off: Fidelio-generated guides consistently improve on state-of-the-art baselines for a diverse set of inference problems.

4 ABSOLUTE CONTINUITY BY TYPING

In Section 4, we establish a reasoning principle (Theorem 4.5) for proving absolute continuity of models and guides, based on typing. Section 6 then uses the reasoning principle to derive the static guarantee of absolute continuity for automatically generated guides.

As alluded to in Section 3, our system addresses AC while permitting expressiveness, by using *tree-structured traces* and *checkpoint-dependent trace types*. We formalize this semantics using the Fidelio Core Calculus (FCC), which captures core aspects of the language supported by Fidelio.

4.1 Syntax of FCC

Figure 4 defines the syntax for FCC programs. Identifiers (i.e., variables and global names) are notated in blue. Capture-free substitution is notated $\cdot\,\{\cdot/\cdot\}$: for example, $m\,\{v/\mathsf{a}\}$ substitutes a value v for a term variable a in command m.

FCC has a pure, deterministic fragment and a monadic, probabilistic fragment. The pure fragment consists of *expressions*. It is a simply typed lambda calculus equipped with booleans, natural numbers n, real numbers r, n-ary operations, and various primitive distributions. n-ary operators $\mathsf{op}_\diamond$ can be neural networks as well as arithmetic and logical operators. The monadic fragment consists of *terms* and *commands*. They have mutually recursive syntax:

- A term can sample from a distribution $\mathsf{sample}(e)$; be a nested command m; perform branching $\mathsf{ite}(e; m_1; m_2)$, where the branches are commands; or invoke a function $\mathsf{call}(\mathsf{f}; e_1; e_2)$. We shall write $\mathsf{ite}(e; m_1; m_2)$ using if-then-else in examples.
- A command sequences computations, ending in $\mathsf{ret}(e)$. Three forms can be sequenced with a command m to form a larger command: (i) $\mathsf{x} = e$ binds an *expression variable* x to expression e in m, (ii) $\mathsf{a} = t$ binds a *term variable* a to term t in m, and (iii) $\mathsf{observe}(e_1; e_2)$ conditions on observed data e_2 being drawn from distribution e_1.

```
a = call(pcfg)
x = embed(a)
observe(NORMAL(x; 1); obs)
ret(unit)
```

```
def pcfg () =
  a = sample(UNIF)
  b = if a < .5 then
          c = sample(NORMAL(0; 1))
          ret(Const(c))
        else
          d1 = call(pcfg)  // recursive call
          d2 = call(pcfg)  // recursive call
          ret(Add(d1, d2))
  ret(b)
```

(a) PCFG model, written in FCC

```
h = obs            // set hidden state
a = call(pcfg; h)  // call
ret(unit)
```

```
def pcfg (h) =
  x1, x2 = NN_a(h)
  a = sample(BETA(x1; x2))
  b = if a < .5 then
          y1, y2 = NN_c(h)
          c = sample(NORMAL(y1; y2))
          ret(Const(c))
        else
          h2 = h                       // set hidden state
          d2 = call(pcfg; h2)          // recursive call
          h1 = NN_d1(h, embed(d2))     // set hidden state
          d1 = call(pcfg; h1)          // recursive call
          ret(Add(d1, d2))
  ret(b)
```

(b) PCFG guide, generated

Figure 5. (a) PCFG model as in Figure 2a, in FCC syntax. (b) Generated guide, with the same trace type.

$$\boxed{s \vdash_t t \Downarrow^w v} \qquad \boxed{\sigma \vdash_m m \Downarrow^w v}$$

$$\textsc{(E:Sample)}\ \frac{e \Downarrow d \qquad v \in d.\text{support} \qquad w = d.\text{density}(v)}{\text{atom}\, v \vdash_t \text{sample}(e) \Downarrow^w v} \qquad \textsc{(E:Cmd)}\ \frac{\sigma \vdash_m m \Downarrow^w v}{\text{trace}\, \sigma \vdash_t m \Downarrow^w v} \qquad \textsc{(E:Branch)}\ \frac{e \Downarrow b \qquad \sigma \vdash_m m_b \Downarrow^w v}{\text{inj}\, \sigma \vdash_t \text{ite}(e; m_{\text{true}}; m_{\text{false}}) \Downarrow^w v}$$

$$\textsc{(E:Call)}\ \frac{\text{def } \mathrm{f}\langle \mathrm{a}\rangle(\mathrm{x}) = m \qquad e_1 \Downarrow v_a \qquad e_2 \Downarrow v_x \qquad \sigma \vdash_t m\,\{v_a/\mathrm{a}\}\,\{v_x/\mathrm{x}\} \Downarrow^w v}{\text{fold}\, \sigma \vdash_t \text{call}(\mathrm{f}; e_1; e_2) \Downarrow^w v} \qquad \textsc{(E:Observe)}\ \frac{v \in d.\text{support} \qquad e_1 \Downarrow d \qquad e_2 \Downarrow v_2 \qquad w_1 = d.\text{density}(v_2) \qquad \sigma \vdash_m m \Downarrow^{w_2} v}{\sigma \vdash_m \text{observe}(e_1; e_2);\, m \Downarrow^{w_1 \cdot w_2} v}$$

$$\textsc{(E:Bnd:Term)}\ \frac{s \vdash_t t \Downarrow^{w_1} v_1 \qquad \left\{\overline{\mathrm{a}' : s'}\right\} \vdash_m m\,\{v_1/\mathrm{a}\} \Downarrow^{w_2} v_2}{\left\{\mathrm{a} : s,\ \overline{\mathrm{a}' : s'}\right\} \vdash_m \mathrm{a} = t;\ m \Downarrow^{w_1 \cdot w_2} v_2} \qquad \textsc{(E:Bnd:Expr)}\ \frac{e \Downarrow v_e \qquad \sigma \vdash_m m\,\{v_e/\mathrm{x}\} \Downarrow^w v}{\sigma \vdash_m \mathrm{x} = e;\ m \Downarrow^w v} \qquad \textsc{(E:Ret)}\ \frac{e \Downarrow v}{\{\} \vdash_m \text{ret}(e) \Downarrow^1 v}$$

Figure 6. Operational semantics for the monadic fragment of FCC: terms and commands.

An FCC program is a global set of possibly mutually recursive function definitions $\overline{\mathcal{F}}$, as well as a "main" command. The syntax def $\mathrm{f}\langle\mathrm{a}\rangle(\mathrm{x}) = m$ assumes for simplicity that f takes two arguments as input (one for a and one for x), but we shall relax this restriction in examples.

Figure 5a defines the same PCFG model program as Figure 2a, using FCC syntax. The program is composed of a main command (top) and a recursive function (bottom). Figure 5b is the automatically generated guide program, also in FCC syntax. By the end of this section, we will be able to show that these two programs have the same trace type.

Sections 4.2–4.4 give an operational semantics, a static semantics, and a measure semantics, respectively, allowing a formal notion of AC for FCC programs.

4.2 Operational Semantics of FCC

This section defines a trace-based, big-step operational semantics, in a style similar to prior trace-based semantics for universal PPLs (e.g., Borgström et al. [2016]). But unlike the prior semantics, traces in FCC are tree-structured rather than lists, and they can record not just sampling but also control-flow events while avoiding imposing a total ordering on these events.

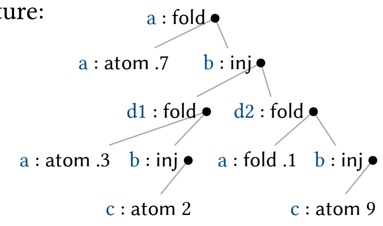
Expressions evaluate to *values*. Judgments have form $e \Downarrow v$. Rules in this pure fragment are standard and omitted herein. A complete semantics can be found in the accompanying technical report [Li et al. 2022].

Terms are evaluated on *events*, and commands are evaluated on *traces*. As Figure 4 shows, events s and traces σ have mutually recursive syntax.

A trace $\{\overline{\mathrm{a} : s}\}$ is an *unordered* map from term variables to events. Events have four forms, corresponding to the four forms of terms: (1) atom v stands for sampling a value v; (2) trace σ stands for a nested trace σ; (3) inj σ stands for branching, where σ traces the chosen branch; and (4) fold σ stands for a function call, where σ traces the function body.

For example, consider trace σ_{main} in (4.1) and its tree structure:

$$
\begin{aligned}
\sigma_{\text{main}} &\overset{\text{def}}{=} \{\mathrm{a} : \mathsf{fold}\,\sigma_0\} \\
\sigma_0 &\overset{\text{def}}{=} \{\mathrm{a} : \mathsf{atom}\,.7,\ \mathrm{b} : \mathsf{inj}\,\{\mathrm{d1} : \mathsf{fold}\,\sigma_1,\ \mathrm{d2} : \mathsf{fold}\,\sigma_2\}\} \\
\sigma_1 &\overset{\text{def}}{=} \{\mathrm{a} : \mathsf{atom}\,.3,\ \mathrm{b} : \mathsf{inj}\,\{\mathrm{c} : \mathsf{atom}\,2\}\} \\
\sigma_2 &\overset{\text{def}}{=} \{\mathrm{a} : \mathsf{atom}\,.1,\ \mathrm{b} : \mathsf{inj}\,\{\mathrm{c} : \mathsf{atom}\,9\}\}
\end{aligned}
\qquad (4.1)
$$



It is a trace of the PCFG model (Figure 5a). It is also a trace of the PCFG guide (Figure 5b), although the guide reorders d1 and d2. Evaluating either program on this trace generates Add(Const(2), Const(9)) in the context-free language.

Contrast (4.1) with (2.1). Unlike the trace in (2.1), FCC traces do not stipulate a *total* ordering on events. An added benefit is that unsafe manual name mangling of the kind found in Figure 2a is avoided. That Pyro program would lead to name collisions and crash the runtime if not for manual name mangling such as `L + "_C"`. By contrast, in FCC, RVs in trace σ_{main} are uniquely identified even if they may correspond to the same term variable in the program text.

Figure 6 defines how terms and commands evaluate. Rules have forms $s \vdash_t t \Downarrow^w v$ and $\sigma \vdash_m m \Downarrow^w v$, meaning that on trace σ (resp. event s), command m (resp. term t) evaluates to v and produces *weight* w. A command denotes a distribution over traces; weight w is the (unnormalized) density of the trace. In E:Sample, event atom v specifies the value v sampled, and the weight is adjusted by the distribution's density at v. In E:Branch, event inj σ specifies the nested trace σ on which to evaluate the chosen branch. In E:Call, event fold σ specifies the nested trace σ on which to evaluate the function body. E:Bnd:Term evaluates $\mathrm{a} = t;\, m$ by splitting the trace into an event s and a smaller trace $\{\overline{\mathrm{a}' : s'}\}$ under which t and m are respectively evaluated. E:Bnd:Expr need not split the trace, as expression variables are not captured in traces. E:Observe performs conditioning, multiplying the weight by the density of the observed data.

4.3 Static Semantics of FCC

Typing judgments of the pure fragment of FCC have form $\Delta;\, \Gamma \vdash_e e : \tau$. The rules are standard and omitted herein. Δ are Γ are environments of term variables and expression variables, respectively:

$$\Delta ::= \bullet \mid \Delta, \mathrm{a} : \tau \qquad \Gamma ::= \bullet \mid \Gamma, \mathrm{x} : \tau$$

A distribution has type dist(τ); samples from the distribution have type τ. For example, a Gamma distribution Gamma$(e_1; e_2)$ has type dist$(\mathbb{R}_+)$; the support is positive reals.

Typing commands and terms. *Trace types* (resp. *event types*) describe control-flow behaviors and probabilistic effects of commands (resp. *terms*). As Figure 4 shows, trace types Σ and event types S have mutually recursive syntax. The trace type of a command has form $\left\{\overline{\mathrm{a} : S}\right\}_e$, where $\overline{\mathrm{a} : S}$ is an

$$\boxed{\Delta;\,\Gamma \vdash_t t : \tau \,\#\, S} \qquad \boxed{\Delta;\,\Gamma \vdash_m m : \tau \,\#\, \Sigma} \qquad \boxed{\vdash_{\mathcal{F}} \mathcal{F} : \langle \tau_1 \rangle(\tau_2) \rightsquigarrow \tau_3 \,\#\, \mathsf{F}}$$

(S:Sample)
$$\frac{\Delta;\,\Gamma \vdash_e e : \mathsf{dist}(\tau)}{\Delta;\,\Gamma \vdash_t \mathsf{sample}(e) : \tau \,\#\, \mathsf{atom}\,\tau}$$

(S:Branch)
$$\frac{\Delta;\,\bullet \vdash_e e : 2 \qquad \Delta;\,\Gamma \vdash_m m_1 : \tau \,\#\, \Sigma_1 \qquad \Delta;\,\Gamma \vdash_m m_2 : \tau \,\#\, \Sigma_2}{\Delta;\,\Gamma \vdash_t \mathsf{ite}(e; m_1; m_2) : \tau \,\#\, \Sigma_1 +_e \Sigma_2}$$

(S:Cmd)
$$\frac{\Delta;\,\Gamma \vdash_m m : \tau \,\#\, \Sigma}{\Delta;\,\Gamma \vdash_t m : \tau \,\#\, \mathsf{trace}\,\Sigma}$$

(S:Call)
$$\frac{\mathsf{f} : \langle \tau_1 \rangle(\tau_2) \rightsquigarrow \tau_3 \,\#\, \mathsf{F} \qquad \Delta;\,\bullet \vdash_e e_1 : \tau_1 \qquad \Delta;\,\Gamma \vdash_e e_2 : \tau_2}{\Delta;\,\Gamma \vdash_t \mathsf{call}(\mathsf{f}; e_1; e_2) : \tau_3 \,\#\, \mathsf{F}\langle e_1 \rangle}$$

(S:Observe)
$$\frac{\begin{array}{c}\Delta;\,\Gamma \vdash_e e_1 : \mathsf{dist}(\tau) \\ \Delta;\,\Gamma \vdash_e e_2 : \tau \qquad \Delta;\,\Gamma \vdash_m m : \tau \,\#\, \Sigma\end{array}}{\Delta;\,\Gamma \vdash_m \mathsf{observe}(e_1; e_2);\ m : \tau \,\#\, \Sigma}$$

(S:Bnd:Term)
$$\frac{\Delta;\,\Gamma \vdash_t t : \tau_1 \,\#\, S \qquad \Delta,\, \mathsf{a} : \tau_1;\,\Gamma \vdash_m m : \tau_2 \,\#\, \left\{\overline{\mathsf{a}' : S'}\right\}_e}{\Delta;\,\Gamma \vdash_m \mathsf{a} = t;\ m : \tau_2 \,\#\, \left\{\mathsf{a} : S,\ \overline{\mathsf{a}' : S'}\right\}_e}$$

(S:Bnd:Expr)
$$\frac{\begin{array}{c}\Delta;\,\Gamma \vdash_e e : \tau_1 \\ \Delta;\,\Gamma,\, \mathsf{x} : \tau_1 \vdash_m m : \tau_2 \,\#\, \Sigma\end{array}}{\Delta;\,\Gamma \vdash_m \mathsf{x} = e;\ m : \tau_2 \,\#\, \Sigma}$$

(S:Return)
$$\frac{\Delta;\,\bullet \vdash_e e : \tau}{\Delta;\,\Gamma \vdash_m \mathsf{ret}(e) : \tau \,\#\, \{\}_e}$$

(S:Def)
$$\frac{\begin{array}{c}\mathsf{f} : \langle \tau_1 \rangle(\tau_2) \rightsquigarrow \tau_3 \,\#\, \mathsf{F} \qquad \mathsf{typedef}\ \mathsf{F} = \forall \mathsf{a} : \tau_1.\, \Sigma \\ \mathsf{a} : \tau_1;\ \mathsf{x} : \tau_2 \vdash_m m : \tau_3 \,\#\, \Sigma\end{array}}{\vdash_{\mathcal{F}} \mathsf{def}\ \mathsf{f}\langle \mathsf{a} \rangle(\mathsf{x}) = m : \langle \tau_1 \rangle(\tau_2) \rightsquigarrow \tau_3 \,\#\, \mathsf{F}}$$

Figure 7. Typing rules for the monadic fragment of FCC: terms, commands, and functions.

$$\begin{array}{l}\mathsf{typedef}\ \mathsf{F}_{\mathsf{pcfg}} = \{ \\ \quad \mathsf{a} : \mathsf{atom}\,\mathbb{R}_{[0,1]}, \\ \quad \mathsf{b} : \{\mathsf{c} : \mathsf{atom}\,\mathbb{R}\}_{\mathsf{Const}(\mathsf{c})} +_{\mathsf{a}<.5} \\ \qquad \{\, \mathsf{d1} : \mathsf{F}_{\mathsf{pcfg}},\ \mathsf{d2} : \mathsf{F}_{\mathsf{pcfg}} \,\}_{\mathsf{Add}(\mathsf{d1},\mathsf{d2})} \\ \}_{\mathsf{b}}\end{array}$$

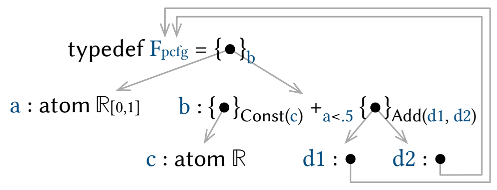


Figure 8. Trace type of the pcfg functions in Figure 5.

unordered map from term variables to event types, and e is the command's return. Event types have four forms, corresponding to the four forms of events (or terms): atom τ, trace Σ, $\Sigma_1 +_e \Sigma_2$, and $\mathsf{F}\langle e \rangle$.

Figure 7 defines the typing rules for commands and terms, which have forms $\Delta;\,\Gamma \vdash_m m : \tau \,\#\, \Sigma$ and $\Delta;\,\Gamma \vdash_t t : \tau \,\#\, S$. They make sure that sampling, branching, and invocation of (possibly) recursive functions are recorded in trace types and event types. Figure 7 also gives the typing rule for global function definitions (S:Def). Metavariable F ranges over names of functions' trace types. For example, function pcfg in Figure 5a has trace type $\mathsf{F}_{\mathsf{pcfg}}$, defined recursively in Figure 8. The main command in Figure 5a then has trace type $\{\,\mathsf{a} : \mathsf{F}_{\mathsf{pcfg}}\,\}_{\mathsf{unit}}$.

Two compatibly typed programs need not agree on the orders computations take place. In particular, it is possible for a compatibly typed guide to reorder computations under the same parent in a tree-structured trace type. For instance, the guide pcfg function in Figure 5b reorders subtrees d1 and d2, but it can still be typed as $\mathsf{F}_{\mathsf{pcfg}}$. The guide also does extra computations such as running neural networks. These computations are pure; they are not part of the guide's trace type.

Checkpoints are part of trace types. Two compatibly typed programs must agree on control-flow decisions. $\mathsf{F}_{\mathsf{pcfg}}$ captures the branching condition $\mathsf{a} < .5$. So for the guide pcfg function to also have type $\mathsf{F}_{\mathsf{pcfg}}$, it should use $\mathsf{a} < .5$ as the branching condition. More generally, two compatibly typed programs must agree on *checkpoint expressions*, which include branching condition e in $\mathsf{ite}(e; m_1; m_2)$, command return e in $\mathsf{ret}(e)$, and term argument e_1 in $\mathsf{call}(\mathsf{f}; e_1; e_2)$. For example, trace type $\mathsf{F}_{\mathsf{pcfg}}$ captures not only the branching condition $\mathsf{a} < .5$ but also command returns such as

$\{\ldots\}_{\mathsf{Const}(\mathrm{c})}$ and $\{\ldots\}_{\mathrm{b}}$. Consider the following command:

$$\mathrm{a} = \mathsf{call}(\mathrm{pcfg});\ \mathrm{b} = \mathsf{ite}(\mathsf{height}(\mathrm{a}) < 6; m_1; m_2);\ \ldots$$

If a model pcfg and its guide were allowed to disagree on command returns, then even though two programs use the same branching condition $\mathsf{height}(\mathrm{a}) < 6$, they could end up making different branching choices because they would disagree on a. Hence, to use trace typing to justify AC, the type system must keep track of checkpoints as part of trace types.

A consequence is that checkpoints can use only term variables—expression variables are not bound in trace types. In S:Branch, S:Return, and S:Call, checkpoint expressions are type-checked under an empty environment • of expression variables. The requirement does not restrict the class of model programs that can be expressed, because the programmer can always use term variables and monadic returns to fix a program that would otherwise not type-check with expression variables. For example, the program below does not type-check because z, an expression variable, is used in the checkpoint $\mathrm{z} > 0$:

$$\mathrm{a} = \mathsf{sample}(\textsc{Normal}(\mathrm{x}; \mathrm{y}));\ \mathrm{z} = \mathrm{a} + 1;\ \mathrm{c} = \mathsf{ite}(\mathrm{z} > 0; m_1; m_2);\ \ldots$$

This program is easily fixed by replacing z with a term variable b and replacing $\mathrm{a} + 1$ with $\mathsf{ret}(\mathrm{a} + 1)$.

It may appear restrictive that a guide must have syntactically identical checkpoints as its model. We found that in practice it is often trivial to modify manually created guides to meet this requirement. More importantly, it fits in a compiler pipeline where guides are automatically generated. Another benefit is that it allows typing to be entirely syntax-directed, thus striking a good balance between simplicity and expressivity.

There exist distributions whose supports depend on parameters. Like prior systems, the expressive power of FCC is limited by the lack of such dependently supported distributions in general. However, we believe that checkpoints offer a viable approach to incorporating them in the future. In particular, sampling from a delta distribution whose support is concentrated on e can be encoded as $\mathrm{a} = \mathsf{ret}(e)$: because this checkpoint is part of the command's trace type, two programs are required to agree on e to be compatibly typed.

Term variables are more than binders. Term variables in FCC are unusual, in the sense that they are not just binders, which have local scopes, but also labels, which can occur in trace types. Thus, renaming the term variables in a command will cause it to have a different trace type. In a trace type $\{\mathrm{a}_1 : S_1, \ldots, \mathrm{a}_N : S_N\}_e$, a term variable a_i can occur in e and the event types of its siblings.

We note that it is possible to adapt the FCC design so that binders and labels are decoupled, using a syntax like $\mathrm{x}\ =\ \mathsf{sample}(\ell; \textsc{Normal}(0; 1))$, where x is a binder and ℓ a label. We choose to represent the binder and the label as one single term variable, for less verbiage.

4.4 Measure Semantics of FCC

Definition 4.1. A closed command m of trace type Σ ($\bullet;\ \bullet \vdash_m m : \tau \mathbin{\#} \Sigma$) induces a measure $[\![m]\!]$ over a measurable space $[\![\Sigma]\!]$ of traces:

$$\mathbf{P}_m(\sigma) \overset{\text{def}}{=} \begin{cases} w, & \text{if } \sigma \vdash_m m \Downarrow^w v \\ 0, & \text{otherwise} \end{cases} \qquad [\![m]\!]\,(A) \overset{\text{def}}{=} \int_A \mathbf{P}_m(\sigma)\, \mu_\Sigma(\mathrm{d}\sigma)$$

Measure $[\![m]\!]$ is defined by integrating a density function $\mathbf{P}_m(\cdot)$ over a measurable set A of traces. As is usual, only terminating executions have a positive density. Measure $[\![m]\!]$ is in general unnormalized as m may perform conditioning. When the normalization constant is in $(0, \infty)$, the normalized measure exists and is given by $\int_A \mathbf{P}_m(\sigma)\mathrm{d}\sigma \Big/ \int \mathbf{P}_m(\sigma)\mathrm{d}\sigma$.

The construction of the measurable space $[\![\Sigma]\!]$ of traces, as well as that of its stock measure μ_Σ, is by a double induction, first on a "step index" for handling recursive trace types, and then on

$$\boxed{s \vdash_s S \Downarrow v} \qquad \boxed{\sigma \vdash_\sigma \Sigma \Downarrow v}$$

$$\text{(W:EVENT:ATOM)}\ \frac{v : \tau}{\mathsf{atom}\, v \vdash_s \mathsf{atom}\, \tau \Downarrow v} \qquad \text{(W:EVENT:TRACE)}\ \frac{\sigma \vdash_\sigma \Sigma \Downarrow v}{\mathsf{trace}\, \sigma \vdash_s \mathsf{trace}\, \Sigma \Downarrow v} \qquad \text{(W:EVENT:BRANCH)}\ \frac{e \Downarrow b \qquad \sigma \vdash_\sigma \Sigma_b \Downarrow v}{\mathsf{inj}\, \sigma \vdash_s \Sigma_{\mathsf{true}} +_e \Sigma_{\mathsf{false}} \Downarrow v}$$

$$\text{(W:EVENT:CALL)}\ \frac{\begin{array}{c}\mathsf{typedef}\ \mathrm{F} = \forall \mathrm{a} : \tau.\, \Sigma \\ e \Downarrow v_a \qquad \sigma \vdash_s \Sigma\, \{v_a/\mathrm{a}\} \Downarrow v\end{array}}{\mathsf{fold}\, \sigma \vdash_s \mathrm{F}\langle e \rangle \Downarrow v} \qquad \text{(W:TRACE:RET)}\ \frac{e \Downarrow v}{\{\} \vdash_\sigma \{\}_e \Downarrow v}$$

$$\text{(W:TRACE:BND)}\ \frac{s \vdash_s S \Downarrow v_1 \qquad \left\{\overline{\mathrm{a}' : s'}\right\} \vdash_\sigma \left\{\overline{\mathrm{a}' : S'}\right\}_e \{v_1/\mathrm{a}\} \Downarrow v}{\left\{\mathrm{a} : s, \overline{\mathrm{a}' : s'}\right\} \vdash_\sigma \left\{\mathrm{a} : S, \overline{\mathrm{a}' : S'}\right\}_e \Downarrow v}$$

Figure 9. Well-formedness of events and traces with respect to event types and trace types.

the structure of trace types and event types. The definition can be found in the technical report. We omit showing the measurability of $\mathbf{P}_m(\cdot)$; we expect a proof to follow the technique found in Borgström et al. [2016] and Szymczak and Katoen [2019].

With the measure semantics for commands defined, we can formalize the notion of absolute continuity, which says the measure induced by one command is "supported" by that of the other:

Definition 4.2. A command m_1 is *absolutely continuous* with respect to another m_2, if $[\![m_2]\!]\,(A) \neq 0$ on every measurable set A of traces for which $[\![m_1]\!]\,(A) \neq 0$.

4.5 Theoretical Results

Before we can establish the theorems, we introduce a notion of typing for traces and events, namely $s \vdash_s S \Downarrow v$ and $\sigma \vdash_\sigma \Sigma \Downarrow v$ defined in Figure 9. The typing rules evaluate checkpoint expressions in types to make sure that they agree with what happens in the trace. The evaluation happens in the pure fragment of FCC. Using the rules, we can derive that trace σ_{main} from (4.1) is well-typed: $\sigma_{\mathrm{main}} \vdash_\sigma \left\{\, \mathrm{a} : \mathrm{F}_{\mathrm{pcfg}} \,\right\}_{\mathrm{unit}} \Downarrow \mathsf{unit}$.

Typing of traces is needed solely as a theoretical device to establish Theorem 4.5 through Theorems 4.3 and 4.4: a command m of trace type Σ has positive density $\mathbf{P}_m(\sigma) > 0$ on a trace σ if and only if σ has type Σ. The guide generator need not type-check traces.

We first present type safety results for the operational semantics. The PRESERVATION theorem states that if a closed, well-typed command (resp. term) evaluates to some value under some trace (resp. event), then the value is well-typed and so is the trace (resp. event):

THEOREM 4.3 (PRESERVATION).

(1) If $\bullet;\ \bullet \vdash_m m : \tau \mathbin{\#} \Sigma$, *and* $\sigma \vdash_m m \Downarrow^w v$, *then* $\bullet;\ \bullet \vdash_e v : \tau$ *and* $\sigma \vdash_\sigma \Sigma \Downarrow v$.
(2) If $\bullet;\ \bullet \vdash_t t : \tau \mathbin{\#} S$ *and* $s \vdash_t t \Downarrow^w v$, *then* $\bullet;\ \bullet \vdash_e v : \tau$ *and* $s \vdash_s S \Downarrow v$.

The NORMALIZATION theorem states that if a command and a trace (resp. a term and an event) can be typed using the same trace type (resp. event type), then evaluation of the command (resp. term) under the trace (resp. event) terminates to some value:

THEOREM 4.4 (NORMALIZATION).

(1) If $\bullet;\ \bullet \vdash_m m : \tau \mathbin{\#} \Sigma$ *and* $\sigma \vdash_\sigma \Sigma \Downarrow v$, *then there exist* w *such that* $\sigma \vdash_m m \Downarrow^w v$.
(2) If $\bullet;\ \bullet \vdash_t t : \tau \mathbin{\#} S$ *and* $s \vdash_s S \Downarrow v$, *then there exist* w *such that* $s \vdash_t t \Downarrow^w v$.

Normalization of the pure fragment of FCC follows from the well-known result for the simply typed lambda calculus, and normalization of the monadic fragment is a result of traces being finite.

Using PRESERVATION and NORMALIZATION, we can show that two compatibly typed commands are mutually absolutely continuous:

Theorem 4.5 (Absolute Continuity By Typing). *If* $\bullet; \bullet \vdash_m m_1 : \tau \# \Sigma$ *and* $\bullet; \bullet \vdash_m m_2 : \tau \# \Sigma$, *then for any measurable set A of traces,* $[\![m_1]\!](A) \neq 0$ *if and only if* $[\![m_2]\!](A) \neq 0$.

Proofs of the theorems can be found in the technical report.

We now specialize Theorem 4.5 to amortized inference of the PCFG model in Figure 5. There, amortization is over a free variable obs in the model m_{m} and in the guide m_{g}. As discussed in Section 3.1, both training and importance sampling require that $m_{\mathrm{g}}\,\{v_{\mathrm{obs}}/\mathrm{obs}\}$ be absolutely continuous with respect to $m_{\mathrm{m}}\,\{v_{\mathrm{obs}}/\mathrm{obs}\}$. Let $\Sigma \overset{\text{def}}{=} \{\mathrm{a} : \mathrm{F}_{\mathrm{pcfg}}\}_{\mathsf{unit}}$. Because $\bullet; \bullet \vdash_m m_{\mathrm{m}}\,\{v_{\mathrm{obs}}/\mathrm{obs}\} : \mathbb{1} \# \Sigma$ and $\bullet; \bullet \vdash_m m_{\mathrm{g}}\,\{v_{\mathrm{obs}}/\mathrm{obs}\} : \mathbb{1} \# \Sigma$, we know that AC is guaranteed to hold for m_{m} and m_{g} by Theorem 4.5. What remains to be shown is that m_{g} can be automatically generated and that generated guides have the right trace types (Section 6).

5 IDENTIFYING CORRELATIONS AND INDEPENDENCES

We want a generated guide to faithfully yet parsimoniously respect conditional dependences. However, the control-flow expressiveness of a universal PPL poses challenges.

Consider conditional dependences in the PCFG model in Figure 5a. The model uses stochastic branching and general recursion, so it is impossible to express it as a BN. But given any of its traces, we can unfold recursion for that trace, as Figure 10 does. Nodes $\boxed{}$ stand for returns of recursive calls to pcfg, and arrows signify dataflow.

Absent of control flow, the dataflow graph is essentially a BN. Per the notion of BN active trails, colliders in Figure 10 mean that leaf nodes $\boxed{\mathrm{c}}$ are correlated with each other and with $\boxed{\mathrm{obs}}$. But we had to unroll recursion to identify such correlations for a given trace. Yet it is infeasible to unroll recursion for every possible trace, as general recursion yields traces of infinitely many shapes.

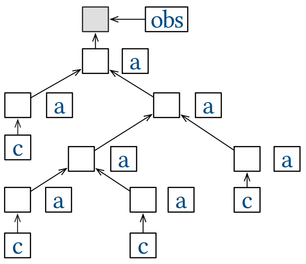


Figure 10. Dataflow graph for a single trace of the PCFG model.

Adding to the complexities is that *control dependences* may or may not transmit correlation.

- In Figure 5a, we know that a (line 6) and pcfg's return (line 14) must be *correlated.* We also know that this correlation is due to control dependence: a determines which branch is taken and thus influences pcfg's return value.
- In contrast, we know that a's value and c's value (line 8) must be *independent given* a $< .5$. Yet c is control-dependent on a: a determines if c is sampled in the first place.

How can a guide generator identify such correlations and independences that involve control flow?

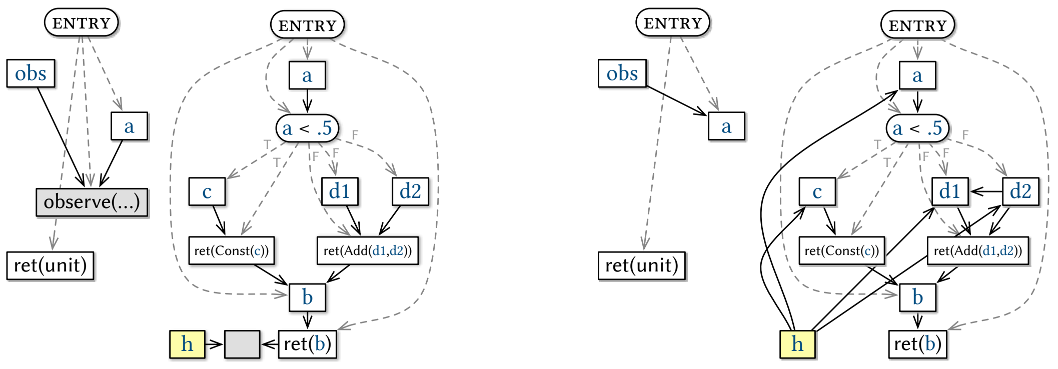
Fidelio extracts program dependences into a graph representation (Section 5.1). It uses a notion of active trails for the graph representation as an indicator of correlation (Sections 5.2 and 6).

5.1 A Graph Representation of Dependences

Program dependence graphs (PDGs) [Ferrante et al. 1987], employed by many optimizing compilers, are a classic idea to represent dependences between program elements. PDGs are appealing because they represent not only data dependences (as BNs do) but also control dependences. The exact definition of PDGs are tailored to specific languages and problems being considered.

Figure 11a shows two PDGs, one for the main command and the other for the pcfg function in Figure 5a. They are generated by the construction in Figure 12. While the presentation of Figure 12 is dense, the construction is mostly standard. It defines three sets $\mathrm{Nodes}_{\mathcal{F}}(\mathcal{F})$, $\mathrm{CDE}_{\mathcal{F}}(\mathcal{F})$, and $\mathrm{DDE}_{\mathcal{F}}(\mathcal{F})$ by structural induction. The sets contain nodes, control-dependence edges (CDEs), and data-dependence edges (DDEs) of the PDG.

There are two types of nodes in a PDG: *control nodes* and *data nodes.* Control nodes are drawn as rounded rectangles ◯. They include (ENTRY), one per function (or main command), and (*e*), one per



(a) PDGs of the PCFG model in Figure 5a (b) PDGs of the PCFG guide in Figure 5b

Figure 11. For the PCFG model program, Fidelio uses the construction in Figure 12 to build the PDGs in (a) before generating the guide program. For comparison, guide PDGs are shown in (b). The PDG construction inserts the collider $\boxed{h} \longrightarrow \boxed{} \longleftarrow \boxed{\text{ret}(b)}$, creating active trails between $\boxed{h}$ and nodes in the PDG. Informed by the active trails, the guide generator makes sample and call terms depend on h (e.g., $\boxed{h} \longrightarrow \boxed{a}$ in (b)), thus faithfully expressing their correlation to historical information stored in h (Section 6).

branching term ite($e; m_1; m_2$). Metavariable C ::= (ENTRY) | (e) ranges over control nodes. Control nodes are sources of CDEs, which are drawn as gray, dashed arrows.

CDEs can have labels ℓ ::= T | F | ε. CDEs from a branching node (e) are labeled either T or F, signifying the branch taken. For example, in Figure 11a, all CDEs from (a < .5) are labeled. CDEs from (ENTRY) are not labeled (i.e., they have the null label ε). In Figure 12, the construction $\text{CDE}_m(m)\ \ell\ C$ (resp. $\text{CDE}_t(t)\ \ell\ C$) defines CDEs induced by a command (resp. a term), where C is the current *control parent*, and ℓ labels the CDE from the control parent.

Data nodes are drawn as rectangles $\boxed{}$. They are created for variable bindings $\boxed{a}$ and $\boxed{\text{obs}}$, monadic returns $\boxed{\text{ret}(e)}$, call arguments $\boxed{e}$, and conditionings $\boxed{\text{observe}(e_1; e_2)}$. Data nodes are sources of DDEs, which are drawn as black, solid arrows.

In Figure 12, $\text{DDE}_m(m)\ \gamma\ N$ (resp. $\text{DDE}_t(t)\ \gamma\ N$) defines DDEs induced by a command (resp. a term). To focus on terms in the monadic fragment, $\text{Nodes}_m(m)$ does not create data nodes for expression variables. However, expression variables can transmit data dependence too. Thus, $\text{DDE}_m(m)\ \gamma\ N$ takes as input a substitution γ ::= • | γ, x ↦ e for expression variables occurring free in m, and it uses an auxiliary function $\text{TV}_\gamma(e)$ to obtain the term variables on which an expression e transitively depends through expression variables.

The other parameter N in $\text{DDE}_m(m)\ \gamma\ N$ is the data node to which the result of evaluating m flows. It can be either a term variable $\boxed{a}$ or a synthetic conditioning sink $\boxed{}$ to which a function's return flows. It is assumed that *stochastically* recursive functions like pcfg do not perform conditioning themselves, since in amortized inference, amortization is always over a *statically known* set of observe sites.

Besides the synthetic $\boxed{}$, construction $\text{Nodes}_{\mathcal{F}}(\mathcal{F})$ also introduces a synthetic $\boxed{h}$, and $\text{DDE}_{\mathcal{F}}(\mathcal{F})$ adds $\boxed{h} \longrightarrow \boxed{}$. For example, see collider $\boxed{h} \longrightarrow \boxed{} \longleftarrow \boxed{\text{ret}(b)}$ in Figure 11a. As Section 6 discusses, Fidelio makes $\boxed{h}$ store historical information to which RVs in the current function are correlated.

5.2 A Notion of Active Trails for PDGs

As Section 2 reviews, in BNs, absence of active trails implies conditional independence. We adapt the notion of active trails to PDGs. We do not claim soundness or completeness of this adapted notion in the general case, though it is sound and almost complete for BN-like programs. Soundness

$$
\begin{aligned}
\mathrm{Nodes}_{\mathcal{F}}(\mathrm{def}\ \mathrm{f}\langle \mathrm{a}\rangle = m) &\overset{\mathrm{def}}{=} \{\textcircled{\textsc{entry}}, \boxed{\mathrm{a}}, \boxed{\mathrm{h}}, \boxed{}\} \cup \mathrm{Nodes}_m(m) && \#\ \boxed{\mathrm{h}}\ \textit{and}\ \boxed{}\ \textit{are synthetic nodes}\\
\mathrm{Nodes}_m(\mathrm{ret}(e)) &\overset{\mathrm{def}}{=} \{\boxed{\mathrm{ret}(e)}\}\\
\mathrm{Nodes}_m(\mathrm{observe}(e_1; e_2);\ m) &\overset{\mathrm{def}}{=} \{\boxed{\mathrm{observe}(e_1; e_2)}\} \cup \mathrm{Nodes}_m(m)\\
\mathrm{Nodes}_m(\mathrm{x} = e;\ m) &\overset{\mathrm{def}}{=} \mathrm{Nodes}_m(m)\\
\mathrm{Nodes}_m(\mathrm{a} = t;\ m) &\overset{\mathrm{def}}{=} \{\boxed{\mathrm{a}}\} \cup \mathrm{Nodes}_t(t) \cup \mathrm{Nodes}_m(m)\\
\mathrm{Nodes}_t(m) &\overset{\mathrm{def}}{=} \mathrm{Nodes}_m(m)\\
\mathrm{Nodes}_t(\mathrm{ite}(e; m_1; m_2)) &\overset{\mathrm{def}}{=} \{\textcircled{e}\} \cup \mathrm{Nodes}_m(m_1) \cup \mathrm{Nodes}_m(m_2)\\
\mathrm{Nodes}_t(\mathrm{sample}(e)) &\overset{\mathrm{def}}{=} \varnothing\\
\mathrm{Nodes}_t(\mathrm{call}(\mathrm{f}; e)) &\overset{\mathrm{def}}{=} \boxed{e}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{CDE}_{\mathcal{F}}(\mathrm{def}\ \mathrm{f}\langle \mathrm{a}\rangle = m) &\overset{\mathrm{def}}{=} \mathrm{CDE}_m(m)\ \varepsilon\ \textcircled{\textsc{entry}} && \#\ m\ \textit{has control parent}\ \textcircled{\textsc{entry}}\\
\mathrm{CDE}_m(\mathrm{ret}(e))\ \ell\ C &\overset{\mathrm{def}}{=} \{C \overset{\ell}{\dashrightarrow} \boxed{\mathrm{ret}(e)}\}\\
\mathrm{CDE}_m(\mathrm{observe}(e_1; e_2);\ m)\ \ell\ C &\overset{\mathrm{def}}{=} \{C \overset{\ell}{\dashrightarrow} \boxed{\mathrm{observe}(e_1; e_2)}\} \cup \mathrm{CDE}_m(m)\ \ell\ C\\
\mathrm{CDE}_m(\mathrm{x} = e;\ m)\ \ell\ C &\overset{\mathrm{def}}{=} \mathrm{CDE}_m(m)\ \ell\ C\\
\mathrm{CDE}_m(\mathrm{a} = t;\ m)\ \ell\ C &\overset{\mathrm{def}}{=} \{C \overset{\ell}{\dashrightarrow} \boxed{\mathrm{a}}\} \cup \mathrm{CDE}_t(t)\ \ell\ C \cup \mathrm{CDE}_m(m)\ \ell\ C\\
\mathrm{CDE}_t(m)\ \ell\ C &\overset{\mathrm{def}}{=} \mathrm{CDE}_m(m)\ \ell\ C\\
\mathrm{CDE}_t(\mathrm{ite}(e; m_1; m_2))\ \ell\ C &\overset{\mathrm{def}}{=} \{C \overset{\ell}{\dashrightarrow} \textcircled{e}\} \cup \mathrm{CDE}_m(m_1)\ \mathsf{T}\ \textcircled{e} \cup \mathrm{CDE}_m(m_2)\ \mathsf{F}\ \textcircled{e} && \#\ m_i\ \textit{has control parent}\ \textcircled{e}\\
\mathrm{CDE}_t(\mathrm{sample}(e))\ \ell\ C &\overset{\mathrm{def}}{=} \varnothing\\
\mathrm{CDE}_t(\mathrm{call}(\mathrm{f}; e))\ \ell\ C &\overset{\mathrm{def}}{=} \{C \overset{\ell}{\dashrightarrow} \boxed{e}\}
\end{aligned}
$$

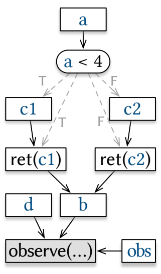
$$
\begin{aligned}
\mathrm{DDE}_{\mathcal{F}}(\mathrm{def}\ \mathrm{f}\langle \mathrm{a}\rangle = m) &\overset{\mathrm{def}}{=} \{\boxed{\mathrm{h}} \longrightarrow \boxed{}\} \cup \mathrm{DDE}_m(m) \bullet \boxed{} && \#\ \textit{function's return flows into}\ \boxed{}\\
\mathrm{DDE}_m(\mathrm{ret}(e))\ \gamma\ N &\overset{\mathrm{def}}{=} \{\boxed{\mathrm{a}} \longrightarrow \boxed{\mathrm{ret}(e)} \mid \mathrm{a} \in \mathrm{TV}_\gamma(e)\} \cup \{\boxed{\mathrm{ret}(e)} \longrightarrow N\}\\
\mathrm{DDE}_m(\mathrm{observe}(e_1; e_2);\ m)\ \gamma\ N &\overset{\mathrm{def}}{=} \{\boxed{\mathrm{a}} \longrightarrow \boxed{\mathrm{observe}(e_1; e_2)} \mid \mathrm{a} \in \mathrm{TV}_\gamma(e_1) \cup \mathrm{TV}_\gamma(e_2)\} \cup \mathrm{DDE}_m(m)\ \gamma\ N\\
\mathrm{DDE}_m(\mathrm{x} = e;\ m)\ \gamma\ N &\overset{\mathrm{def}}{=} \mathrm{DDE}_m(m)\ (\gamma,\ \mathrm{x} \mapsto e)\ N && \#\ \textit{substitution}\ \gamma\ \textit{is extended}\\
\mathrm{DDE}_m(\mathrm{a} = t;\ m)\ \gamma\ N &\overset{\mathrm{def}}{=} \mathrm{DDE}_t(t)\ \gamma\ \boxed{\mathrm{a}} \cup \mathrm{DDE}_m(m)\ \gamma\ N && \#\ \textit{evaluation result of}\ t\ \textit{flows into}\ \boxed{\mathrm{a}}\\
\mathrm{DDE}_t(m)\ \gamma\ N &\overset{\mathrm{def}}{=} \mathrm{DDE}_m(m)\ \gamma\ N\\
\mathrm{DDE}_t(\mathrm{ite}(e; m_1; m_2))\ \gamma\ N &\overset{\mathrm{def}}{=} \{\boxed{\mathrm{a}} \longrightarrow \textcircled{e} \mid \mathrm{a} \in \mathrm{TV}_\gamma(e)\} \cup \mathrm{DDE}_m(m_1)\ \gamma\ N \cup \mathrm{DDE}_m(m_2)\ \gamma\ N\\
\mathrm{DDE}_t(\mathrm{sample}(e))\ \gamma\ N &\overset{\mathrm{def}}{=} \{\boxed{\mathrm{a}} \longrightarrow N \mid \mathrm{a} \in \mathrm{TV}_\gamma(e)\}\\
\mathrm{DDE}_t(\mathrm{call}(\mathrm{f}; e))\ \gamma\ N &\overset{\mathrm{def}}{=} \{\boxed{\mathrm{a}} \longrightarrow \boxed{e} \mid \mathrm{a} \in \mathrm{TV}_\gamma(e)\} \cup \{\boxed{e} \longrightarrow N\}\\
\mathrm{TV}_\gamma(e) &\overset{\mathrm{def}}{=} \{\mathrm{a} \mid \mathrm{a} \in \mathrm{FV}(e)\} \cup \{\mathrm{a} \mid \mathrm{x} \in \mathrm{FV}(e) \wedge \mathrm{a} \in \mathrm{TV}_{\gamma \setminus \mathrm{x}}(\gamma(\mathrm{x}))\}
\end{aligned}
$$

Figure 12. Constructing PDGs for model programs.

and completeness are theoretically appealing properties, but they require much more measure-theoretic sophistication and we leave them to future work. We satisfy ourselves with experimental evidence that absence (resp. presence) of active trails is a good indicator of independence (resp. correlation) (Figure 13), and with an extensive evaluation of how dependence awareness impacts the effectiveness of deep amortized inference (Section 8.2).

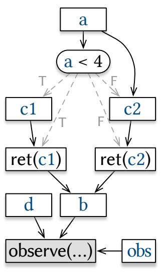
A definition of PDG active trails must take into consideration control dependences. Our definition uses a few terminologies. A *control path*, starting from $\textcircled{\textsc{entry}}$, is a directed path consisting solely of CDEs. A control node C is a *control ancestor* of a PDG node N if C is on the control path to N. Two nodes *control-contradict* each other if their control paths travel through mutually exclusive branches of the same branching node.

```
a = sample(Normal(4;10))
b = if a < 4 then
      c1 = sample(Normal(0;2))
      ret(c1)
    else
      c2 = sample(Normal(8;1))
      ret(c2)
d = sample(Normal(0;4))
observe(Normal(b+d;0.1); obs)
ret(unit)
```



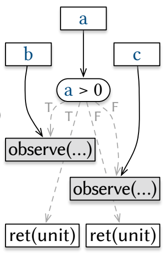
(a)

```
a = sample(Normal(4;10))
b = if a < 4 then
      c1 = sample(Normal(0;2))
      ret(c1)
    else
      c2 = sample(Normal(a;1))
      ret(c2)
d = sample(Normal(0;4))
observe(Normal(b+d;0.1); obs)
ret(unit)
```

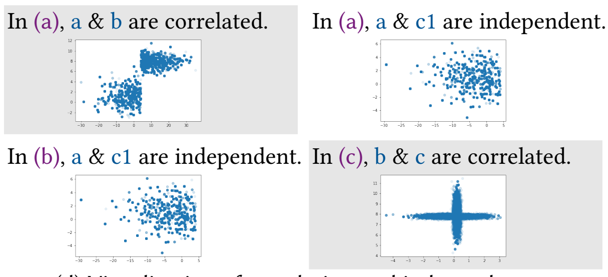


(b)

```
a = sample(Normal(0;10))
b = sample(Normal(0;1))
c = sample(Normal(8;1))
if a > 0 then
  observe(Normal(b;0.1); 0.2)
  ret(unit)
else
  observe(Normal(c;0.1); 7.8)
  ret(unit)
ret(unit)
```



(c)



(d) Visualization of correlation and independence

Figure 13. Independence (resp. correlation) as visualized in (d) agrees with the absence (resp. presence) of PDG active trails per Definition 5.1. (ENTRY nodes have been elided in PDGs.)

Definition 5.1 (PDG active trails). Let $\mathcal{G}$ be a PDG. Let $\boldsymbol{M}$ be a set of nodes in $\mathcal{G}$. Let $N_1, N_2, \ldots, N_n$ be a trail in $\mathcal{G}$ (i.e., an undirected, acyclic path composed of CDEs and DDEs). The trail is said to be *active given* $\boldsymbol{M}$ when all the following conditions hold:

(1) Control ancestors of N_1 and those of N_n are in $\boldsymbol{M}$.
(2) $\boldsymbol{M}$ is closed under control ancestry. That is, control ancestors of any node $M \in \boldsymbol{M}$ are in $\boldsymbol{M}$, too.
(3) No node in the trail control-contradicts N_1, N_n, or $\boldsymbol{M}$.
(4) The trail is active given $\boldsymbol{M}$, in the sense of Bayesian networks. That is,
  (i) for any collider $N_{i-1} \to N_i \leftarrow N_{i+1}$ in the trail, N_i or one of its descendants is either in $\boldsymbol{M}$ or a conditioning node (namely $\boxed{\text{observe(...)}}$ or $\boxed{}$);
  (ii) no other node in the trail is in $\boldsymbol{M}$ or is a conditioning node.

Only when two nodes are both reached in the same execution does it make sense to speak of correlation between them. Condition (1) ensures that when we speak of conditional dependence between N_1 and N_n, we do condition on the branching choices that caused N_1 and N_n to be reached in the first place. Condition (2) further ensures that the branching nodes themselves are reached. Condition (3) says that correlation cannot be transmitted through nodes impossible to be reached when N_1, N_n, and $\boldsymbol{M}$ are all reached. Finally, condition (4) requires that a PDG active trail be active in the BN sense.

By conditions (1) and (2), (ENTRY) is always in $\boldsymbol{M}$, which by condition (4) implies that (ENTRY) cannot be on an active trail. So we elide (ENTRY) (and CDEs from it) in PDG drawings hereafter. Also notice that conditions (1)–(3) are automatically satisfied when the PDG does not have branching nodes—that is, when the PDG can be reduced to a BN. In this case, results about soundness [Verma and Pearl 1988] and (almost sure) completeness of active trails [Meek 1995] carry over.

We use examples in Figure 13 to further explain and justify Definition 5.1:

- *An active trail can contain CDEs.* In program (a), a and b are correlated, as is visualized by the top-left plot in (d). The correlation agrees with the presence of active trails between a and b in the PDG. All the active trails contain a CDE (e.g., $\boxed{\mathrm{a}}\longrightarrow(\mathrm{a}<4)\overset{\mathrm{F}}{\dashrightarrow}\boxed{\mathrm{ret(c2)}}\longrightarrow\boxed{\mathrm{b}}$).
- *Control ancestors of end nodes are always considered given.* That is, they are in $\boldsymbol{M}$ per condition (1). Independence between a and c1 in (a) is visualized by the top-right plot in (d). In the PDG, because c1's dependences are in question, the value of its control parent $(\mathrm{a}<4)$ is considered known. Thus, given $(\mathrm{a}<4)$, the trail $\boxed{\mathrm{a}}\longrightarrow(\mathrm{a}<4)\overset{\mathrm{T}}{\dashrightarrow}\boxed{\mathrm{c1}}$ is inactive in the BN sense, and so is the trail $\boxed{\mathrm{a}}\longrightarrow(\mathrm{a}<4)\overset{\mathrm{F}}{\dashrightarrow}\boxed{\mathrm{c2}}\longrightarrow\boxed{\mathrm{ret(c2)}}\longrightarrow\boxed{\mathrm{b}}\longleftarrow\boxed{\mathrm{ret(c1)}}\longleftarrow\boxed{\mathrm{c1}}$.
- *An active trail cannot contain nodes that control-contradict any end node.* Program (b) differs from (a) in the line c2 = sample(Normal(a;1)). Hence, its PDG has an extra DDE $\boxed{\mathrm{a}}\longrightarrow\boxed{\mathrm{c2}}$. But still, a and c1 are independent, as visualized by the bottom-left plot in (d). The trail $\boxed{\mathrm{a}}\longrightarrow\boxed{\mathrm{c2}}\longrightarrow\boxed{\mathrm{ret(c2)}}\longrightarrow\boxed{\mathrm{b}}\longleftarrow\boxed{\mathrm{ret(c1)}}\longleftarrow\boxed{\mathrm{c1}}$ is inactive per condition (3): $\boxed{\mathrm{c2}}$ and $\boxed{\mathrm{ret(c2)}}$ in the trail are on the false branch, whereas end node $\boxed{\mathrm{c1}}$ is on the true branch.
- *An active trail can contain intermediary nodes that control-contradict each other.* In program (c), b and c are correlated, as visualized by the bottom-right plot in (d). The correlation agrees with an active trail in the PDG: $\boxed{\mathrm{b}}\longrightarrow\boxed{\mathrm{observe(...)}}\overset{\mathrm{T}}{\dashleftarrow}(\mathrm{a}>0)\overset{\mathrm{F}}{\dashrightarrow}\boxed{\mathrm{observe(...)}}\longleftarrow\boxed{\mathrm{c}}$. Although the trail contains nodes on mutually exclusive branches, it is still active per Definition 5.1.

Active trails are compositional across call boundaries. Definition 5.1 does not unroll recursive functions. A call can create correlations internal to the callee; they are addressed by the callee's PDG. A call can also create correlations across call boundaries. We can show that when a call is unrolled, a trail across the call boundary is active if and only if (1) the caller part of the trail is active and (2) the callee part of the trail is active. Below we illustrate this compositionality using PDGs in Figure 11a:

- obs in the main command is correlated with c in the callee pcfg. The correlation is indicated by two active trails: $\boxed{\mathrm{obs}}\longrightarrow\boxed{\mathrm{observe(...)}}\longleftarrow\boxed{\mathrm{a}}$ in the caller, and $\boxed{\mathrm{ret(b)}}\longleftarrow\boxed{\mathrm{b}}\longleftarrow\boxed{\mathrm{ret(Const(c))}}\longleftarrow\boxed{\mathrm{c}}$ in the callee, with the call's return value flowing from $\boxed{\mathrm{ret(b)}}$ to $\boxed{\mathrm{a}}$.
- Right subtree d2 is correlated with the c sampled by the left subtree d1 = call(pcfg), as indicated by two active trails: $\boxed{\mathrm{d2}}\longrightarrow\boxed{\mathrm{ret(Add(d1,d2))}}\longleftarrow\boxed{\mathrm{d1}}$ in the caller, and $\boxed{\mathrm{ret(b)}}\longleftarrow\boxed{\mathrm{b}}\longleftarrow\boxed{\mathrm{ret(Const(c))}}\longleftarrow\boxed{\mathrm{c}}$ in the callee, with the recursive call's return value flowing from $\boxed{\mathrm{ret(b)}}$ to $\boxed{\mathrm{d1}}$.

6 GUIDE GENERATION AS TYPE-DIRECTED, DEPENDENCE-AWARE TRANSLATION

We formalize guide generation as a translation of model programs. The translation makes sure that generated guides have the same trace type as their models, thereby guaranteeing absolute continuity. The translation can reorder computations under the same control parent and reconfigure data dependences using the notion of active trails, thereby offering faithfulness and parsimony.

Figure 14 presents the translation rules. The translation is by structural induction on *typing derivations* of commands, terms, and function definitions.

Translating function definitions. Compositionality of active trails (Section 5.2) implies that guide generation is intra-procedural and thus modular. A key insight is that guide functions can use an extra *hidden-state* parameter h as a proxy for transmitting correlations across call boundaries. Rule GG:Def defines the translation of a function, informed by its PDG $\mathcal{G}$. The translated function has an extra parameter h.

At a call site, the guide generator gathers into a hidden state the historical information correlated with the call's return, and passes the hidden state to the callee (GG:Tm:Call). In turn, inside the

(Reorder)

$$\frac{(l_i)_{i=1}^{N} \text{ is a permutation of } (i)_{i=1}^{N} \qquad (\exists N \in \text{Nodes}_t(t_{l_k}).\ \boxed{\mathsf{a}_{l_j}} \longrightarrow N \in \mathcal{G}.\text{edges}) \Rightarrow j < k}{(\mathsf{a}_{l_i} = t_{l_i})_{i=1}^{N} \text{ reorders}_{\mathcal{G}}\ (\mathsf{a}_i = t_i)_{i=1}^{N}}$$

$$\boxed{\mathcal{G};\ C;\ \Delta';\ \mathsf{h} : \mathbb{R}\ \vdash_m\ \langle \Delta;\ \Gamma \vdash_m m : \tau \,\#\, \Sigma \rangle\ \rightsquigarrow\ m'}$$

(GG:Cmd)

$$\frac{\begin{array}{c}\text{TermBindings}(m) = (\mathsf{a}_i = t_i)_{i=1}^{N} \qquad (\mathsf{a}_{l_i} = t_{l_i})_{i=1}^{N} \text{ reorders}_{\mathcal{G}}\ (\mathsf{a}_i = t_i)_{i=1}^{N} \\ \forall i.\ \mathcal{G};\ C;\ \Delta', \mathsf{a}_{l_1} : \tau_{l_1}, ..., \mathsf{a}_{l_{i-1}} : \tau_{l_{i-1}};\ \mathsf{h} : \mathbb{R};\ \mathsf{a}_{l_i}\ \vdash_t\ \langle \Delta_{l_i};\ \Gamma_{l_i} \vdash_t t_{l_i} : \tau_{l_i} \,\#\, S_{l_i} \rangle\ \rightsquigarrow\ t'_{l_i}\end{array}}{\mathcal{G};\ C;\ \Delta';\ \mathsf{h} : \mathbb{R}\ \vdash_m\ \langle \Delta;\ \Gamma \vdash_m m : \tau \,\#\, \{\mathsf{a}_1 : S_1, ..., \mathsf{a}_N : S_N\}_e \rangle\ \rightsquigarrow\ \mathsf{a}_{l_1} = t'_{l_1};\ ...;\ \mathsf{a}_{l_N} = t'_{l_N}; \mathsf{ret}(e)}$$

$$\boxed{\mathcal{G};\ C;\ \Delta';\ \mathsf{h} : \mathbb{R};\ \mathsf{a}\ \vdash_t\ \langle \Delta;\ \Gamma \vdash_t t : \tau \,\#\, S \rangle\ \rightsquigarrow\ t'}$$

(GG:Tm:Sample)

$$\frac{\begin{array}{c}\mathsf{b}_1 : \tau_1, ..., \mathsf{b}_n : \tau_n;\ \mathsf{h} : \mathbb{R} \vdash_e D_{\phi_a}(\alpha_1, ..., \alpha_k) : \mathsf{dist}(\tau) \qquad \boldsymbol{M} \overset{\text{def}}{=} C \cup \left\{ \boxed{\mathsf{b}_1}, ..., \boxed{\mathsf{b}_n}, \boxed{\mathsf{h}} \right\} \\ \{\alpha_1, ..., \alpha_k\} = \left\{ \alpha \in \{\mathsf{b}_1, ..., \mathsf{b}_n, \mathsf{h}\} \;\middle|\; \text{there is an active trail in } \mathcal{G} \text{ between } \boxed{\mathsf{a}} \text{ and } \boxed{\alpha} \text{ given } \boldsymbol{M} \setminus \left\{ \boxed{\alpha} \right\} \right\}\end{array}}{\mathcal{G};\ C;\ \mathsf{b}_1 : \tau_1, ..., \mathsf{b}_n : \tau_n;\ \mathsf{h} : \mathbb{R};\ \mathsf{a}\ \vdash_t\ \langle \Delta;\ \Gamma \vdash_t \mathsf{sample}(e) : \tau \,\#\, \mathsf{atom}\,\tau \rangle\ \rightsquigarrow\ \mathsf{sample}(D_{\phi_a}(\alpha_1, ..., \alpha_k))}$$

(GG:Tm:Call)

$$\frac{\begin{array}{c}\boldsymbol{M} \overset{\text{def}}{=} C \cup \left\{ \boxed{\mathsf{b}_1}, ..., \boxed{\mathsf{b}_n}, \boxed{\mathsf{h}}, \boxed{e} \right\} \\ \{\alpha_1, ..., \alpha_k\} = \left\{ \alpha \in \{\mathsf{b}_1, ..., \mathsf{b}_n, \mathsf{h}\} \;\middle|\; \text{there is an active trail in } \mathcal{G} \text{ between } \boxed{\mathsf{a}} \text{ and } \boxed{\alpha} \text{ given } \boldsymbol{M} \setminus \left\{ \boxed{\alpha} \right\} \right\}\end{array}}{\mathcal{G};\ C;\ \mathsf{b}_1 : \tau_1, ..., \mathsf{b}_n : \tau_n;\ \mathsf{h} : \mathbb{R};\ \mathsf{a}\ \vdash_t\ \langle \Delta;\ \Gamma \vdash_t \mathsf{call}(\mathsf{f}; e) : \tau \,\#\, \mathsf{F}\langle e \rangle \rangle\ \rightsquigarrow\ \mathsf{call}(\mathsf{f}; e; \mathsf{op}_{\phi_a}(\alpha_1, ..., \alpha_k))}$$

(GG:Tm:Branch)

$$\frac{\forall i.\ \mathcal{G};\ C \cup \{\textcircled{e}\};\ \Delta';\ \mathsf{h} : \mathbb{R}\ \vdash_m\ \langle \Delta;\ \Gamma \vdash_m m_i : \tau \,\#\, \Sigma_i \rangle\ \rightsquigarrow\ m'_i}{\mathcal{G};\ C;\ \Delta';\ \mathsf{h} : \mathbb{R};\ \mathsf{a}\ \vdash_t\ \langle \Delta;\ \Gamma \vdash_t \mathsf{ite}(e; m_1; m_2) : \tau \,\#\, \Sigma_1 +_e \Sigma_2 \rangle\ \rightsquigarrow\ \mathsf{ite}(e; m'_1; m'_2)}$$

(GG:Tm:Cmd)

$$\frac{\mathcal{G};\ C;\ \Delta';\ \mathsf{h} : \mathbb{R}\ \vdash_m\ \langle \Delta;\ \Gamma \vdash_m m : \tau \,\#\, \Sigma \rangle\ \rightsquigarrow\ m'}{\mathcal{G};\ C;\ \Delta';\ \mathsf{h} : \mathbb{R};\ \mathsf{a}\ \vdash_t\ \langle \Delta;\ \Gamma \vdash_t m : \tau \,\#\, \Sigma \rangle\ \rightsquigarrow\ m'}$$

$$\boxed{\mathcal{G};\ \mathsf{h} : \mathbb{R}\ \vdash_{\mathcal{F}}\ \langle \vdash_{\mathcal{F}} \mathcal{F} : \langle \tau_1 \rangle \rightsquigarrow \tau_2 \,\#\, \mathsf{F} \rangle\ \rightsquigarrow\ \mathcal{F}'}$$

(GG:Def)

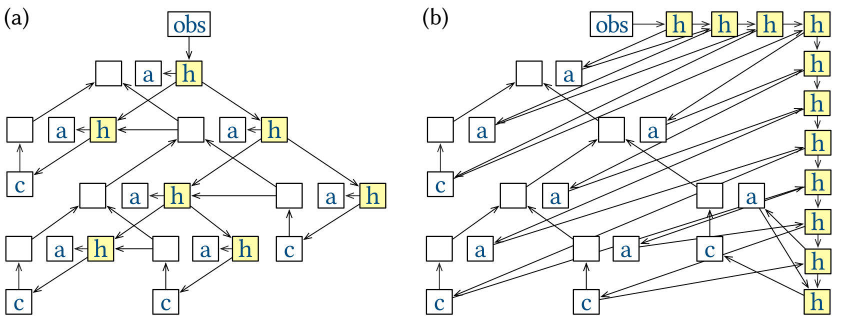
$$\frac{\begin{array}{c}\mathsf{f} : \langle \tau_1 \rangle \rightsquigarrow \tau_2 \,\#\, \mathsf{F} \qquad \text{typedef } \mathsf{F} = \forall \mathsf{a} : \tau_1.\, \Sigma \\ \mathcal{G};\ \{\textcircled{\text{ENTRY}}\};\ \mathsf{a} : \tau_1;\ \mathsf{h} : \mathbb{R}\ \vdash_m\ \langle \mathsf{a} : \tau_1;\ \bullet \vdash_m m : \tau_2 \,\#\, \Sigma \rangle\ \rightsquigarrow\ m'\end{array}}{\mathcal{G};\ \mathsf{h} : \mathbb{R}\ \vdash_{\mathcal{F}}\ \langle \vdash_{\mathcal{F}} \mathsf{def}\ \mathsf{f}\langle \mathsf{a} \rangle = m : \langle \tau_1 \rangle \rightsquigarrow \tau_2 \,\#\, \mathsf{F} \rangle\ \rightsquigarrow\ \mathsf{def}\ \mathsf{f}\langle \mathsf{a} \rangle(\mathsf{h}) = m'}$$

Figure 14. Guide generation as translation of typing derivations

callee,[2] if a sample or call term has an active trail to parameter h in the PDG—indicating correlation with history through the return value—then the guide generator makes that term depend on h (GG:Tm:Sample and GG:Tm:Call). We expand on this translation below.

Translating terms. Translation of a term t has form $\mathcal{G};\ C;\ \Delta';\ \mathsf{h} : \mathbb{R};\ \mathsf{a}\ \vdash_t\ \langle \Delta;\ \Gamma \vdash_t t : \tau \,\#\, S \rangle\ \rightsquigarrow\ t'$, where $\mathcal{G}$ is the PDG of the current function $\mathcal{F}$, C is the set of control ancestors, Δ' is the term variables in scope after reordering (done by GG:Cmd), h is the hidden-state parameter added for $\mathcal{F}$, and

[2]Recall that in Figure 12, the PDG construction adds a synthetic node $\boxed{\mathsf{h}}$ and connects it to the function's return via a collider $\boxed{\mathsf{h}} \longrightarrow \boxed{} \longleftarrow \boxed{\mathsf{ret}(...)}$. See, for instance, collider $\boxed{\mathsf{h}} \longrightarrow \boxed{} \longleftarrow \boxed{\mathsf{ret}(\mathsf{b})}$ in Figure 11a.



(a): Dataflow graph of the Fidelio guide (Figure 5b)

(b): Dataflow graph of an RNN guide

Figure 15. Dataflow in two guides, for the same trace of the PCFG model (cf. Figure 10).

a is the term variable to which t and t' are bound. GG:Tm:Branch and GG:Tm:Cmd translate branching terms and nested commands, by recursively invoking command translation. GG:Tm:Sample and GG:Tm:Call translate sample and call terms, and they must rewire data dependences.

Translating sample terms by reconfiguring dependences. GG:Tm:Sample translates sample(e). It first chooses a learnable distribution D_{ϕ_a} having the same type as e. For instance, a = sample(Unif) in line 6 of Figure 5a is translated using Beta(NN$_a$(·)) as the learnable distribution over $\mathbb{R}_{[0,1]}$ (lines 5–6 of Figure 5b). It is also possible to choose a universal density estimator using recent advances in normalizing flows [Huang et al. 2018].

GG:Tm:Sample then passes relevant data dependences $\alpha_1, ..., \alpha_k$ as input to the learnable distribution via $D_{\phi_a}(\alpha_1, ..., \alpha_k)$. Data dependences $\alpha_i \in \{b_1, ..., b_n, h\}$ are chosen from variables lexically in scope, including the hidden-state parameter h. The translation uses existence of PDG active trails as an indicator of correlation. That is, α_i is considered a dependency of a = sample($D_{\phi_a}(...)$) if an active trail exists in $\mathcal{G}$ between $\boxed{\alpha_i}$ and $\boxed{a}$ given all other variables in scope (as well as all control ancestors).

For instance, consider translating c = sample(Normal(0; 1)) in line 8 of Figure 5a. In the PDG (Figure 11a), trail $\boxed{c} \longrightarrow \boxed{\text{ret(Const(c))}} \longrightarrow \boxed{b} \longrightarrow \boxed{\text{ret(b)}} \longrightarrow \boxed{} \longleftarrow \boxed{h}$ is active given { (Entry), (a < .5), $\boxed{a}$ }. So h is input to the learnable distribution Normal(NN$_c$(·)) for c in lines 8–9 of Figure 5b:

$$\mathcal{G};\ \{(\text{ENTRY}),(a<.5)\};\ a:\mathbb{R}_{[0,1]};\ h:\mathbb{R};\ c\ \vdash_t \langle a:\mathbb{R}_{[0,1]};\ \bullet \vdash_t \text{sample}(\text{NORMAL}(0;1)) : \mathbb{R}\ \#\ \text{atom}\ \mathbb{R}\rangle$$
$$\rightsquigarrow \text{sample}(\text{NORMAL}(\text{NN}_c(h)))$$

Translating call terms by evolving hidden states. GG:Tm:Call translates function calls. As in GG:Tm:Sample, dependencies $\alpha_1, ..., \alpha_k$ are those in-scope variables to which active trails exist in the PDG. They are input to a learnable function op$_{\phi_a}$(·), whose output is then passed to the callee as its hidden-state parameter.

For instance, consider translating d1 = call(pcfg) in line 11 of Figure 5a. In the PDG,

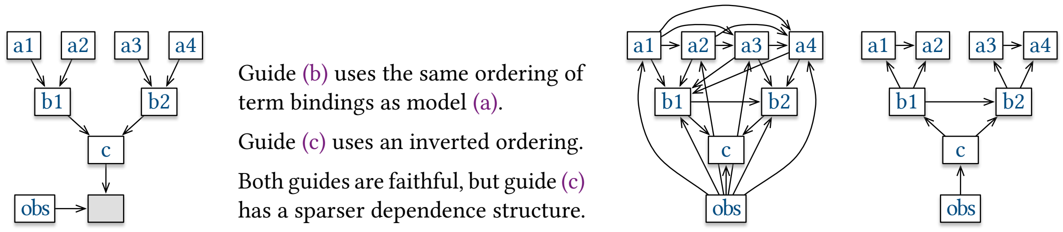
- $\boxed{d1} \longrightarrow \boxed{\text{ret(Add(d1,d2))}} \longleftarrow \boxed{d2}$ is an active trail given { (Entry), (a < .5), $\boxed{h}$, $\boxed{a}$ }, and
- $\boxed{d1} \longrightarrow \boxed{\text{ret(Const(c))}} \longrightarrow \boxed{b} \longrightarrow \boxed{\text{ret(b)}} \longrightarrow \boxed{} \longleftarrow \boxed{h}$ is an active trail given { (Entry), (a < .5), $\boxed{d2}$, $\boxed{a}$ }.

So both h and d2 are merged into the hidden state passed to the callee, in lines 14–15 of Figure 5b:

$$\mathcal{G};\ \{(\text{ENTRY}),(a<.5)\};\ a:\mathbb{R}_{[0,1]}, d2:\text{Expr};\ h:\mathbb{R};\ d1\ \vdash_t \langle a:\mathbb{R}_{[0,1]};\ \bullet \vdash_t \text{call(pcfg)} : \text{Expr}\ \#\ F_{\text{pcfg}}\rangle$$
$$\rightsquigarrow \text{call(pcfg}; \text{NN}_{d1}(h, \text{embed}(d2)))$$

d2 needs an embedding layer because it is a syntax tree (of type Expr) rather than a scalar value.

If we unfold recursion in the PCFG guide for the same trace of execution as in Figure 10, we get Figure 15a, which makes it explicit how the hidden state is evolved to include history. Initially, the hidden state contains only information about obs. Visiting a right subtree simply uses the parent's



Guide (b) uses the same ordering of term bindings as model (a).

Guide (c) uses an inverted ordering.

Both guides are faithful, but guide (c) has a sparser dependence structure.

(a) model PDG (b) guide PDG (c) guide PDG

Figure 16. Ordering of term bindings has implications for the data-dependence sparsity of faithful guides.

hidden state. Before visiting a left subtree d1, the hidden state is updated to include information about the right subtree d2 just visited (see the leftward horizontal arrows $\boxed{h} \longleftarrow \boxed{}$).

While information about historically sampled c's flows into the hidden state via returns of recursive calls $\boxed{}$, information about historically sampled a's is never merged into h. As the beginning of Section 5 analyzes, an $\boxed{a}$ is independent of all RVs to be sampled later in the current subtree, *given* the branching condition $\boxed{a < .5}$. An $\boxed{a}$ is also independent of all RVs to be sampled after the current call returns, *given* the call's return value $\boxed{}$.

Contrast that with Figure 15b, which visualizes dataflow for the same trace but in a guide that uses an RNN to create correlations [Le et al. 2017]. Unaware of the program dependence structure, the RNN inevitably renders all RVs correlated. As the syntax tree is traversed, the RNN's hidden state accretes information about every RV visited thus far, causing a later RV to treat all earlier RVs as dependencies. As a result of the to-and-fro dataflow, hidden states contain unneeded information, diluting learning signals.

Translating commands and reordering terms. Rule GG:Cmd translates a model command, which has the trace type $\{\mathrm{a}_1 : S_1, \ldots, \mathrm{a}_N : S_N\}_e$, to a guide command of form $\mathrm{a}_{l_1} = t'_{l_1}; \ldots; \mathrm{a}_{l_N} = t'_{l_N}; \mathrm{ret}(e)$.

Importantly, the guide command can choose a different ordering $\left(\mathrm{a}_{l_i} = t_{l_i}\right)_{i=1}^{N}$ of term bindings than the original ordering $(\mathrm{a}_i = t_i)_{i=1}^{N}$. Rule Reorder specifies the constraint that reordering must satisfy. In particular, if a_{l_j} is a free variable in a checkpoint node in term t_{l_k}, then $\mathrm{a}_{l_j} = t_{l_j}$ must occur before $\mathrm{a}_{l_k} = t_{l_k}$. Recall from Section 4.3 that checkpoints are part of trace types, so satisfaction of the constraint helps ensure that the translated guide command can use the same checkpoints and thus be compatibly typed.

Each subterm of the model command is translated by recursively invoking the term translation rules. The term-variable environment for subterm t'_{l_i} is extended with $\mathrm{a}_{l_1}, \ldots, \mathrm{a}_{l_{i-1}}$, the term variables in scope for the guide term after reordering.

Reordering of term bindings is often desirable because it can lead to fewer dependences while maintaining faithfulness. Fidelio follows the heuristic of Stuhlmüller et al. [2013] and Paige and Wood [2016] to invert topological orderings (while respecting the reordering constraint).

Consider a model that has a PDG as shown in Figure 16a. There, a1, a2, a3, a4, b1, b2, c is a valid topological ordering of the term bindings. In (b) is the PDG of a faithful guide, using the same ordering. (Notice that removing any DDE makes it unfaithful by introducing conditional independences not found in the model.) In (c) is the PDG of another faithful guide, using an inverted topological ordering c, b1, b2, a1, a2, a3, a4. Whereas (b) is almost fully connected, guide (c) attains a sparse dependence structure using available conditional independences—e.g., given b2, a3 is independent of all variables in {obs, c, b1, a1, a2}. As we show in Section 8, for the same total number of neural-network parameters, guide (c) leads to better performance than guide (b), both for training and for inference.

AC by construction. We prove that the translation preserves trace typing:

Table 1. ● model and guide have compatible types. ❍ model and guide fail to be compatibly typed. Last seven programs appeared in the expressiveness evaluation conducted by Wang et al. [2021].

Program	Description	Lew et al. [2019]	Wang et al. [2021]	Fidelio
collider	Figure 1 (model & guide2)	●	❍	●
pcfg	Figure 5, (a) & (b)	❍	❍	●
treebn	Figure 16, (a) & (c)	●	❍	●
curvefit	Curve fitting, Lew et al. [2019, Fig. 4]	●	❍	●
kalman	Kalman Smoother	●	●	●
branching	Random Control Flow	❍	●	●
marsaglia	Marsaglia Algorithm	❍	●	●
aircraft	Aircraft Detection	●	●	●
gmm	Gaussian Mixture Model	●	●	●
sprinkler	Bayesian Network	●	●	●
dp	Dirichlet Process	❍	❍	❍

Theorem 6.1 (Type-Preserving Translation).
If $\mathcal{G};\ \{\textcircled{\small ENTRY}\};\ \bullet;\ \mathsf{obs}:\mathbb{R}\ \vdash_m\ \langle \bullet;\ \mathsf{obs}:\mathbb{R}\vdash_m m_{\mathrm{m}} : \tau \mathbin{\#} \Sigma\rangle \rightsquigarrow m_{\mathrm{g}}$*, then* $\bullet;\ \mathsf{obs}:\mathbb{R}\vdash_m m_{\mathrm{g}} : \tau \mathbin{\#} \Sigma$.

It follows from Theorems 4.5 and 6.1 that for a well-typed model, the translation is guaranteed to generate a guide that is mutually absolutely continuous with the model:

Theorem 6.2 (Absolute Continuity by Construction).
If $\mathcal{G};\ \{\textcircled{\small ENTRY}\};\ \bullet;\ \mathsf{obs}:\mathbb{R}\ \vdash_m\ \langle \bullet;\ \mathsf{obs}:\mathbb{R}\vdash_m m_{\mathrm{m}} : \tau \mathbin{\#} \Sigma\rangle \rightsquigarrow m_{\mathrm{g}}$ *and* $v_{\mathrm{obs}} : \mathbb{R}$*, then for any measurable set* A *of traces,* $\llbracket m_{\mathrm{m}}\ \{v_{\mathrm{obs}}/\mathsf{obs}\}\rrbracket\ (A) \neq 0$ *if and only if* $\llbracket m_{\mathrm{g}}\ \{v_{\mathrm{obs}}/\mathsf{obs}\}\rrbracket\ (A) \neq 0$.

7 IMPLEMENTATION

Fidelio uses Pyro [Bingham et al. 2019] as the underlying inference engine. A Fidelio model is compiled to a model and a guide in Pyro with explicit name mangling. The user can specify as arguments hyperparameters including the number of layers in a network, the number of neurons in each layer, and the activation function used. It is also easy to modify neural networks in generated guides, as we do in setting up our experiments (Section 8.2).

A few surface-syntax features help streamline guide generation. (1) Fidelio supports directives using which the programmer can hint shapes of tensors. (2) Fidelio supports directives for specifying embedding functions for data that warrant special embedding layers (e.g., convolutional networks as image embeddings). (3) Fidelio provides an option that generalizes MADE [Germain et al. 2015] to allow weight sharing among local networks.

8 EVALUATION

8.1 Expressiveness of the Trace-Type System

Table 1 evaluates the expressiveness of Fidelio's type system, comparing it with state-of-the-art type systems for AC [Lew et al. 2019; Wang et al. 2021]. The benchmark programs include examples in this paper and also those taken from prior work [Lew et al. 2019; Wang et al. 2021], Anglican [Wood et al. 2014], and Pyro, with reasonable modifications so that they conform to our syntax.

The type system of Lew et al. [2019] cannot type-check *models* of pcfg, branching, and marsaglia, because they use branching and recursion in their general forms. The type system of Wang et al. [2021] cannot type-check *guides* of collider, pcfg, treebn, and curvefit, because they reorder computations. dp uses stochastic memoization, a PPL feature not yet supported by these systems.

Table 2. Benchmarks for evaluating different approaches to guide generation. Column "#RVs": expected number of RVs sampled in a trace. Validation loss is computed per (2.3) on a validation sample set.

Program	Structures	#RVs	LoC (model)	Capacity	Validation Loss		
					Mean-Field	LSTM	Fidelio
treebn	BN	31	37	2K	51.88	45.76	41.07
				5K	51.88	43.96	41.04
ecoli70	BN	46	54	10K	11.89	15.28	16.56
				15K	11.94	15.31	16.60
longrange ($k = 5$)	loop	8	16	1.4K	25.39	24.49	21.88
longrange ($k = 10$)	loop	13	16	1.4K	44.05	43.69	40.38
longrange ($k = 20$)	loop	23	16	1.4K	81.23	81.32	77.59
longrange ($k = 40$)	loop	43	16	1.4K	155.8	156.0	152.1
captcha	recursion	6	43	4.4M	1.644	−1.325	−1.212
astropcfg	recursion	15.0	159	100K	6.226	2.352	1.980
				175K	6.217	2.362	2.035
				250K	6.216	2.387	2.114
mathcaptcha	recursion	14.5	237	1.1M	7.962	14.84	3.603
				1.25M	8.150	14.78	2.242
				1.5M	8.329	14.79	1.950
				3M	8.380	9.531	2.511
gmmcc	recursion	217.4	49	20K	111.1	257.8	16.22
				50K	111.2	170.1	15.16
				80K	111.1	152.4	14.55

8.2 Performance Implication of Dependence-Awareness for Training and Inference

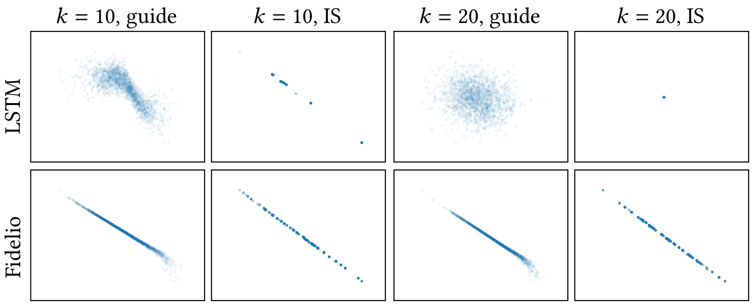
We assess the implications of adopting dependence-aware guide generation for both ahead-of-time training and run-time inference. Table 2 summarizes our benchmarks, which include a variety of applications, program sizes, dependence structures, and training setups.

We compare Fidelio mainly with two methods for guide generation in a universal PPL: mean-field guides, which make strong independence assertions, and RNN guides, which make all variables correlated. Generated Fidelio guides were modified so that they have the same number of trainable parameters (i.e., capacity) as the other two guides. Networks in the Fidelio guides and mean-field guides typically have no more than four linear layers and use ReLU activation. All guides for a benchmark program are trained for the same number of iterations. An appendix in the technical report documents experimental setups and additional statistics. The RNN we use in all experiments is LSTM [Hochreiter and Schmidhuber 1997], which deals with the vanishing gradient problem and is thus better at remembering correlation than vanilla RNNs.

Table 2 shows validation loss as a metric for evaluating training performance. Fidelio consistently leads to lower validation loss as computed by (2.3)—trained Fidelio guides are closer approximations to true posteriors than mean-field or LSTM guides, averaged on the validation sample set.

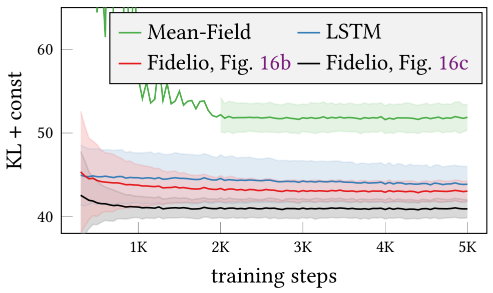
Validation losses alone do not tell the whole story, however. Below we explain our benchmarks and findings in more detail.

Long-range correlation. Benchmark longrange [Harvey et al. 2019] tests the ability for guides to capture correlation between variables sampled far apart. The model samples two Gaussian variables a and b—and also k noise variables in between—before observing the sum of all $k + 2$ variables.



k	ESS M.-F.	ESS LSTM	ESS Fidelio
5	1.53	3.08	50.5
10	1.00	2.21	41.9
20	2.19	1.54	56.0
40	2.66	1.19	45.6

Figure 17. Drawing samples from LSTM and Fidelio guides trained for the **longrange** benchmark. Plots visualize correlation between two variables that are sampled k variables apart in the model.



Capacity	ESS M.-F.	ESS LSTM	ESS Fidelio Fig. 16b	ESS Fidelio Fig. 16c
2K	1.39	3.13	8.67	82.8
5K	1.38	9.84	9.51	84.1

Figure 18. Experimental results for **treebn**. Left: Training steps vs. training loss profile, with all guides having capacity 5K. Right: Quality of IS samples as measured by ESS.

We assess how well the inferred posteriors capture the strong correlation between a and b sampled k variables apart, with $k \in \{5, 10, 20, 40\}$. We draw 2,000 samples after training each guide for 20,000 steps and plot them in Figure 17. In columns one and three, samples are drawn from the trained guides. In columns two and four, samples are drawn using IS, with the guides as proposals.

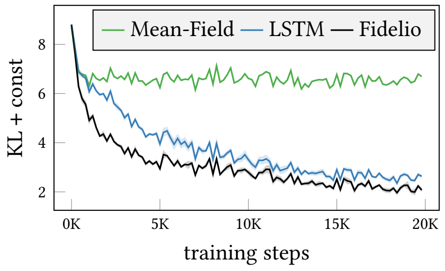
The LSTM guide manages to capture the correlation to some degree when $k = 10$. However, the LSTM guide fails to remember correlation when $k = 20$. Quality of the learned proposal distributions has a knock-on effect on IS. When $k = 20$, the LSTM guide results in visibly lower-quality IS samples than the Fidelio guide of equal capacity. We also compute the effective sample size (ESS) for each guide, as a quantitative (though not comprehensive) metric of sample quality [Kong 1992]; Fidelio guides lead to significantly higher effective sample sizes.

Computation reordering. Benchmark treebn assesses the impact of computation reordering on training and inference performance. The model is linear Gaussian BN, similar to Figure 16a in structure. In addition to the mean-field and LSTM guides, we compare two Fidelio guides that either retain or invert the model's topological ordering of variables (cf. Figures 16b and 16c).

Figure 18 plots the training loss (2.3) as training progresses. All four guides have the same capacity (5K). The sparser Fidelio guide converges to the best solution. The table in Figure 18 shows the Fidelio guide with an inverted topological ordering results in significantly higher ESS than all the other guides: a sparser yet faithful dependence structure makes inference more effective.

The sparser guide is rejected by a prior type system [Wang et al. 2021], which disallows reordering computations. Trace-type systems have performance implications.

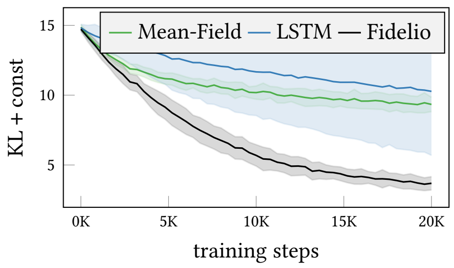
ECOLI70: a large Bayesian network. We use benchmark ecoli70 [Schäfer and Strimmer 2005] to evaluate the implication of dependence-aware guide generation for large BN models. The program is a linear Gaussian BN with 46 variables and 70 arcs. Results can be found in an appendix; they



Accuracy of sentence regeneration

Capacity	Regeneration Rate			Levenshtein Distance		
	M.-F.	LSTM	Fidelio	M.-F.	LSTM	Fidelio
100K	40.0%	57.4%	**71.7%**	4.36	2.99	2.72
175K	39.8%	60.8%	71.3%	4.41	3.08	**2.63**
250K	39.2%	62.4%	69.8%	4.36	3.04	2.75

Figure 19. Experimental results for **astropcfg**. Left: Training steps vs. training loss profile, with all guides having capacity 175K. Right: Inference accuracy, at three capacities.



Accuracy of captcha recognition

Capacity	Recognition Rate			Levenshtein Distance		
	M.-F.	LSTM	Fidelio	M.-F.	LSTM	Fidelio
100K+CNN	35.8%	8.94%	70.5%	2.50	4.25	0.809
250K+CNN	36.8%	9.82%	79.5%	2.45	4.21	0.579
500K+CNN	35.3%	9.67%	**83.1%**	2.59	4.22	**0.463**
2M+CNN	34.9%	39.4%	77.6%	2.62	2.66	0.650

Figure 20. Experimental results for **mathcaptcha**. Left: Training steps vs. training loss profile, with all guides having capacity 3M. Right: Inference accuracy, at four capacities. All guides use a CNN of capacity 1M.

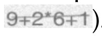
show (1) that the mean-field guide and the LSTM guide did equally poorly on ecoli70 and (2) that Fidelio significantly improves on these dependence-agnostic approaches.

Captcha: a regular grammar. In benchmark captcha [Mansinghka et al. 2013], the model generates a captcha image by sampling a probabilistic regular grammar, and the inference problem is to recognize captcha text given an image. Guide programs use convolutional networks as image embeddings (Section 7). Experimental results can be found in an appendix.

While the mean-field guide did poorly on captcha, the LSTM guide and the Fidelio guide did almost equally well, in terms of convergence and recognition rates. A possible explanation is given by the structure of the model. Because captcha samples from a regular—rather than context-free—language, the dataflow exhibits a linear-recursive pattern (shown in an appendix). As a result, the model has almost no conditional independence for Fidelio to exploit, and the unfolded dataflow graph for the Fidelio guide is almost identical to that induced by an RNN.

Astronomer: a context-free grammar. In benchmark astropcfg [Manning and Schütze 1999], the model samples sentences such as "astronomers saw stars with telescopes". We consider the inference task of regenerating observed sentences. The program consists of mutually recursive functions corresponding to nonterminal symbols in the PCFG.

Figure 19 shows the training-loss profile for all guides of the same capacity (175K). It also shows inference accuracy, as measured by sentence regeneration rates and Levenshtein edit distances for all guides and for three capacities. Fidelio demonstrates a clear advantage over the LSTM guide. In particular, it increases regeneration rate by 10% on average. The results suggest that it pays off to exploit the tree-recursive dataflow pattern and the resulting conditional independences in the astropcfg model (Figures 10 and 15).

Math captcha. We design a second captcha benchmark, mathcaptcha, that samples simple arithmetic expressions and renders them as captcha images (e.g., 
). Unlike captcha, mathcaptcha

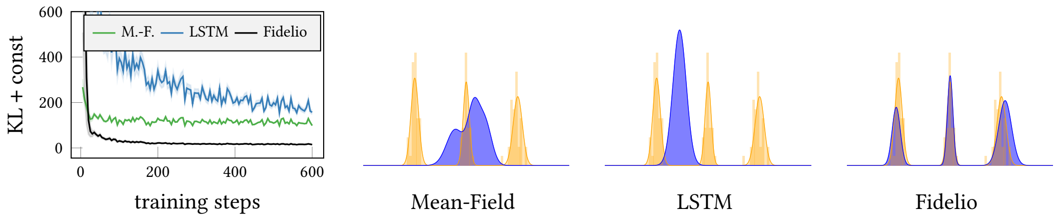


Figure 21. Experimental results for **gmmcc**. Left: training steps vs. training-loss profile. Right: ground-truth clusters (yellow) and clusters inferred using trained guides (blue). All guides have the same capacity.

samples from a context-free language, so we expect Fidelio to improve convergence and captcha recognition rate over the LSTM guide.

Figure 20 confirms our belief: Fidelio outperforms mean-field and LSTM guides, by a large margin. The LSTM guide did not start to pick up until network capacity was raised to 3M (including an invariant 1M for a CNN embedding). In contrast, with limited network capacity and training budget, Fidelio can do better. Conditional independences inherent in PCFG-like models make them favored applications for Fidelio.

Gaussian mixture model for clustering and classification. In benchmark gmmcc [Le et al. 2017], the model samples a stochastic number of Gaussians and further samples data points from them. Given an observed dataset, the inference task is to simultaneously identify clusters and classify data points into clusters. This model is challenging because it samples a large number of RVs (~200). Nonetheless, since data points are independently distributed, it offers conditional independences that Fidelio can potentially tap into.

Figure 21 plots the training-loss profile and kernel density estimations: with the same number of particles, the Fidelio guide is the only guide that correctly inferred the number of clusters.

Summary. We observe that Fidelio consistently improves over LSTM and mean-field guides, both on convergence of training and on accuracy of inference. Our experiments suggest that compared with LSTM, it is most favorable to use Fidelio for models that contain conditional independences and contain ~10 RVs or more.

For models containing ~20 RVs or more, a mean-field guide may be competitive against LSTM; see ecoli70, longrange ($k = 20, 40$), mathcaptcha, and gmmcc. Unaware of program dependence structures, LSTM has to accrete information about all RVs in its hidden state. Unlike Fidelio or mean-field guides, LSTM guides are unable to directly express any conditional independence, which leads to diluted learning signals especially when the number of RVs is large. In contrast, Fidelio guides capitalize on available conditional independences, so training can focus on learning the real correlations.

9 RELATED WORK

Language-based solutions to absolute continuity. Lew et al. [2019] and Wang et al. [2021] design type systems to check AC statically. Lew et al. [2019] formulate AC in terms of a denotational measure semantics for a trace-based PPL. Their type system allows some forms of stochastic control flow, including loops, but disallows general recursion or the kind of stochastic branching needed for models such as PCFGs. It allows guides to reorder computations in some way.

Wang et al. [2021] propose a coroutine-based PPL, where a model and a guide send and receive messages over communication channels to synchronize on sampled values, branching choices, and function calls. AC is formulated in terms of a measure semantics induced by this operational semantics. The approach is inspired by session types [Honda et al. 1999]: their *guide types* describe communication protocols of model–guide pairs. Unlike Lew et al. [2019], the type system allows branching and recursion in their general forms. However, it requires that computations happen in exactly the same (total) order as prescribed by guide types.

In both prior approaches, it is the programmer who authors guides and makes sure that guides have compatible types with models. In our approach, guides are automatically generated and are guaranteed to have compatible types with models.

Lee et al. [2019] introduce a static analysis for Pyro's stochastic variational inference. The analysis aims to prove that a guide has the same support as the model and satisfies differentiability conditions. It deals with Pyro features including tensors and plating, but it does not support stochastic branching or general recursion.

Generating guides automatically. A recent spate of work in machine learning has focused on generating guide programs [Stuhlmüller et al. 2013; Paige and Wood 2016; Le et al. 2017; Webb et al. 2018, 2019; Baudart and Mandel 2021]. Except for Le et al. [2017], the prior approaches deal with graphical models or non-universal PPLs; stochastic branching and general recursion are not supported. Le et al. [2017] address the challenge by having a guide program talk to an LSTM network, leading to faithful but "fully connected", non-parsimonious guides. Webb et al. [2018] create faithful, parsimonious guides, but only for BNs. Webb et al. [2019] and Weilbach et al. [2020] integrate normalizing flows [Rezende and Mohamed 2015; Huang et al. 2018] to increase the representational power of learnable continuous distributions; it is an orthogonal extension that can be incorporated into Fidelio.

Dependence analysis for PPLs. Hur et al. [2014] study program slicing for an imperative PPL with loops. Like guide generation, slicing takes into account conditional dependences as indicated by active trails. Unlike guide generation, the slicer is concerned with conditional dependencies of a program's return value, which exists in all finite executions, whereas a guide generator is concerned with conditional dependencies of `sample` and `call` terms, which are reached only stochastically and subject to reordering in guides. Gorinova et al. [2021] use an information-flow type system to identify conditional independences in a non-universal PPL. It is interesting to study if the information-flow perspective can be applied to a universal PPL, which could serve as a good basis for studying soundness of PDG active trails as proposed in Section 5.2.

10 CONCLUSION

We have presented Fidelio, a framework that can automatically generate guide programs for deep amortized inference in a universal PPL. It addresses challenges posed by the control-flow expressiveness of a universal PPL. Fidelio uses a novel trace-type system and generates guides via a type-guided, dependence-aware translation of models. Theoretical results show that the translation is type-preserving and thus ensures absolute continuity by construction. Experimental results show that *symbolic* dependence information in probabilistic programs can be used effectively to aid in *neural*-network-based inference.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We thank Nada Amin, Hongfei Fu, Edward Lee, Yingzao Li, Di Wang, and Rob Zinkov for discussions and help. This work was supported by the Natural Sciences and Engineering Research Council of Canada. The views and opinions expressed are those of the authors and do not necessarily reflect the position of any funding agency.

REFERENCES

autoguide 2022. Automatic guide generation (Pyro documentation). https://docs.pyro.ai/en/1.8.0/infer.autoguide.html.

Guillaume Baudart and Louis Mandel. 2021. Automatic guide generation for Stan via NumPyro. In *Int'l Conf. on Probabilistic Programming (PROBPROG)*. arXiv:2110.11790

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 (JMLR)* 20, 1 (2019). arXiv:1810.09538

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A lambda-calculus foundation for universal probabilistic programming. In *ACM SIGPLAN Conf. on Functional Programming (ICFP)*. https://doi.org/10.1145/2951913.2951942

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 *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3314221.3314642

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Tran. on Programming Languages and Systems (TOPLAS)* 9, 3 (July 1987). https://doi.org/10.1145/24039.24041

Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. 2015. MADE: Masked autoencoder for distribution estimation. In *Int'l Conf. on Machine Learning (ICML)*. http://proceedings.mlr.press/v37/germain15.pdf

Noah Goodman, Vikash K. Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A language for generative models. In *Conf. on Uncertainty in Artificial Intelligence (UAI)*. arXiv:1206.3255

Maria I. Gorinova, Andrew D. Gordon, Charles Sutton, and Matthijs Vákár. 2021. Conditional independence by typing. *ACM Tran. on Programming Languages and Systems (TOPLAS)* 44, 1 (Dec. 2021). https://doi.org/10.1145/3490421 arXiv:2010.11887

William Harvey, Andreas Munk, Atılım Güneş Baydin, Alexander Bergholm, and Frank Wood. 2019. Attention for inference compilation. arXiv:1910.11961

Geoffrey E. Hinton, Peter Dayan, Brendan J. Frey, and Radford M. Neal. 1995. The "wake-sleep" algorithm for unsupervised neural networks. *Science* 268, 5214 (1995). https://doi.org/10.1126/science.7761831

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (Nov. 1997). https://doi.org/10.1162/neco.1997.9.8.1735

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1999. Language primitives and type discipline for structured communication-based programming. In *European Symp. on Programming (ESOP)*. https://doi.org/10.1007/BFb0053567

Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. 2018. Neural autoregressive flows. In *Int'l Conf. on Machine Learning (ICML)*. http://proceedings.mlr.press/v80/huang18d/huang18d.pdf

Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. 2014. Slicing probabilistic programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2594291.2594303

Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. 1999. An introduction to variational methods for graphical models. *Machine learning* 37, 2 (1999). https://doi.org/10.1023/A:1007665907178

Diederik P. Kingma and Max Welling. 2014. Auto-encoding variational Bayes. In *Int'l Conf. on Learning Representations (ICLR)*. arXiv:1312.6114

Augustine Kong. 1992. *A Note on Importance Sampling Using Standardized Weights.* Technical Report 348. Department of Statistics, University of Chicago. https://d3qi0qp55mx5f5.cloudfront.net/stat/docs/tech-rpts/tr348.pdf

Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. 2017. Inference compilation and universal probabilistic programming. In *Int'l Conf. on Artificial Intelligence and Statistics (AISTATS)*. arXiv:1610.09900

Tuan Anh Le, Adam R. Kosiorek, N. Siddharth, Yee Whye Teh, and Frank Wood. 2019. Revisiting reweighted wake-sleep for models with stochastic control flow. In *Conf. on Uncertainty in Artificial Intelligence (UAI)*. arXiv:1805.10469

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2019. Towards verified stochastic variational inference for probabilistic programs. *Proc. of the ACM on Programming Languages (PACMPL)* 4, POPL (Dec. 2019). https://doi.org/10.1145/3371084

Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. 2019. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proc. of the ACM* *on Programming Languages (PACMPL)* 4, POPL (Dec. 2019). https://doi.org/10.1145/3371087

Jianlin Li, Leni Ven, Pengyuan Shi, and Yizhou Zhang. 2022. *Synthesizing Guide Programs for Sound, Effective Deep Amortized Inference.* Technical Report CS-2022-01. School of Computer Science, University of Waterloo.

Carol Mak, C.-H. Luke Ong, Hugo Paquet, and Dominik Wagner. 2021. Densities of almost surely terminating probabilistic programs are differentiable almost everywhere. In *European Symp. on Programming (ESOP).* https://doi.org/10.1007/978-3-030-72019-3_16 arXiv:2004.03924

Christopher Manning and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing.* MIT Press.

Vikash K. Mansinghka, Tejas D. Kulkarni, Yura N. Perov, and Joshua B. Tenenbaum. 2013. Approximate Bayesian image interpretation using generative probabilistic graphics programs. In *Conf. on Neural Information Processing Systems (NIPS).* arXiv:1307.0060

Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3192366.3192409

Christopher Meek. 1995. Strong completeness and faithfulness in Bayesian networks. In *Conf. on Uncertainty in Artificial Intelligence (UAI).* arXiv:1302.4973

Brooks Paige and Frank Wood. 2016. Inference networks for sequential Monte Carlo in graphical models. In *Int'l Conf. on Machine Learning (ICML).* arXiv:1602.06701

Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann. https://doi.org/10.1016/B978-0-08-051489-5.50001-1

Danilo Rezende and Shakir Mohamed. 2015. Variational inference with normalizing flows. In *Int'l Conf. on Machine Learning (ICML).* arXiv:1505.05770

Daniel Ritchie, Paul Horsfall, and Noah D. Goodman. 2016. Deep amortized inference for probabilistic programs. arXiv:1610.05735

Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. 2019. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proc. of the ACM on Programming Languages (PACMPL)* 3, POPL (2019). https://doi.org/10.1145/3290350 arXiv:1907.06249

Juliane Schäfer and Korbinian Strimmer. 2005. A shrinkage approach to large-scale covariance matrix estimation and implications for functional genomics. *Statistical Applications in Genetics and Molecular Biology* 4 (2005). https://doi.org/10.2202/1544-6115.1175

N. Siddharth, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank Wood, and Philip Torr. 2017. Learning disentangled representations with semi-supervised deep generative models. In *Conf. on Neural Information Processing Systems (NIPS).* arXiv:1706.00400

Andreas Stuhlmüller, Jessica Taylor, and Noah D. Goodman. 2013. Learning stochastic inverses. In *Conf. on Neural Information Processing Systems (NIPS).* https://proceedings.nips.cc/paper/2013/file/7f53f8c6c730af6aeb52e66eb74d8507-Paper.pdf

Marcin Szymczak and Joost-Pieter Katoen. 2019. Weakest preexpectation semantics for Bayesian inference. In *Int'l School on Engineering Trustworthy Software Systems (SETSS).* https://doi.org/10.1007/978-3-030-55089-9_3

Dustin Tran, Matthew D. Hoffman, Dave Moore, Christopher Suter, Srinivas Vasudevan, Alexey Radul, Matthew Johnson, and Rif A. Saurous. 2018. Simple, distributed, and accelerated probabilistic programming. In *Conf. on Neural Information Processing Systems (NeurIPS).* arXiv:1811.02091

Alan M. Turing. 1937. On computable numbers, with an application to the entscheidungsproblem. *Proc. of the London mathematical society* 2, 1 (1937). https://doi.org/10.1112/plms/s2-42.1.230

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2021. *An Introduction to Probabilistic Programming.* arXiv:1809.10756

Thomas Verma and Judea Pearl. 1988. Causal networks: Semantics and expressiveness. In *Conf. on Uncertainty in Artificial Intelligence (UAI).* arXiv:1304.2379

Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Sound probabilistic inference via guide types. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* arXiv:2104.03598

Stefan Webb, Jonathan P. Chen, Martin Jankowiak, and Noah Goodman. 2019. Improving automated variational inference with normalizing flows. In *6^{th} ICML Workshop on Automated Machine Learning.* https://www.automl.org/wp-content/uploads/2019/06/automlws2019_Paper23.pdf

Stefan Webb, Adam Goliński, Robert Zinkov, N. Siddharth, Tom Rainforth, Yee Whye Teh, and Frank Wood. 2018. Faithful inversion of generative models for effective amortized inference. In *Conf. on Neural Information Processing Systems (NIPS)*. arXiv:1712.00287

Christian Weilbach, Boyan Beronov, William Harvey, and Frank Wood. 2020. Structured conditional continuous normalizing flows for efficient amortized inference in graphical models. In *Int'l Conf. on Artificial Intelligence and Statistics (AISTATS)*. http://proceedings.mlr.press/v108/weilbach20a/weilbach20a.pdf

Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *Int'l Conf. on Artificial Intelligence and Statistics (AISTATS)*. arXiv:1507.00996

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2017. Understanding deep learning requires rethinking generalization. In *Int'l Conf. on Learning Representations (ICLR)*. arXiv:1611.03530

Cheng Zhang, Judith Bütepage, Hedvig Kjellström, and Stephan Mandt. 2019. Advances in variational inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)* 41, 8 (2019). https://doi.org/10.1109/TPAMI.2018.2889774 arXiv:1711.05597

Received 2022-07-07; accepted 2022-11-07