Multi-Language Probabilistic Programming

SAM STITES, Northeastern University, USA
JOHN M. LI, Northeastern University, USA
STEVEN HOLTZEN, Northeastern University, USA

There are many different probabilistic programming languages that are specialized to specific kinds of probabilistic programs. From a usability and scalability perspective, this is undesirable: today, probabilistic programmers are forced up-front to decide which language they want to use and cannot mix-and-match different languages for handling heterogeneous programs. To rectify this, we seek a foundation for sound interoperability for probabilistic programming languages: just as today's Python programmers can resort to low-level C programming for performance, we argue that probabilistic programmers should be able to freely mix different languages for meeting the demands of heterogeneous probabilistic programming environments. As a first step towards this goal, we introduce MultiPPL, a probabilistic multi-language that enables programmers to interoperate between two different probabilistic programming languages: one that leverages a high-performance exact discrete inference strategy, and one that uses approximate importance sampling. We give a syntax and semantics for MultiPPL, prove soundness of its inference algorithm, and provide empirical evidence that it enables programmers to perform inference on complex heterogeneous probabilistic programs and flexibly exploits the strengths and weaknesses of two languages simultaneously.

CCS Concepts: • **Mathematics of computing** → **Bayesian computation**; • **Theory of computation** → *Probabilistic computation.*

Additional Key Words and Phrases: Multi-language semantics, probabilistic programming, Bayesian inference

ACM Reference Format:
Sam Stites, John M. Li, and Steven Holtzen. 2025. Multi-Language Probabilistic Programming. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 124 (April 2025), 28 pages. https://doi.org/10.1145/3720482

1 Introduction

Scalable and reliable probabilistic inference remains a significant barrier for applying and using probabilistic programming languages (PPLs) in practice. The core of the inference challenge is that there is no universal approach: different kinds of inference strategies are specialized for different kinds of probabilistic programs. For example, Stan's inference strategy is highly effective on continuous and differentiable programs such as hierarchical Bayesian models, but struggles on programs with high-dimensional discrete structure such as graph and network reachability [9]. On the other extreme, languages like Dice and ProbLog scale well on purely-discrete problems, but the price for this scalability is that they must forego any support whatsoever of continuous probability distributions [16, 23]. In an ideal world, a probabilistic programmer would not have to commit to one language or the other: they could use a Dice-like language for high-performance scalable inference on the discrete portion of the program, a Stan-like language for the portion to

An extended version of this work including the appendix can be found at Stites et al. [57].

Authors' Contact Information: Sam Stites, Northeastern University, Boston, USA, stites.s@northeastern.edu; John M. Li, Northeastern University, Boston, USA, li.john@northeastern.edu; Steven Holtzen, Northeastern University, Boston, USA, s.holtzen@northeastern.edu.

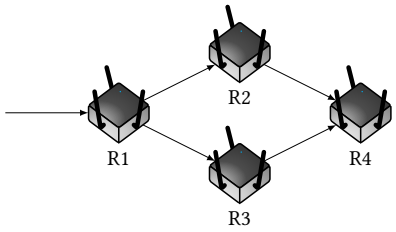
ACM 2475-1421/2025/4-ART124
https://doi.org/10.1145/3720482

which it is well-suited, and be able to transfer data and control-flow between these two languages for heterogeneous programs.

This raises a key question: how should we orchestrate the handoff between two probabilistic programming languages whose underlying semantics may be radically different and seemingly incompatible? This question of sound language interoperability has been extensively explored in the context of non-probabilistic languages [13, 31, 36, 39, 47, 61, 62], where the goal is to prove properties such as type-soundness and termination in a multi-language setting. As a starting point, Matthews and Findler [31] introduced an effective model for capturing the interaction between two languages by *language embedding*: the syntax and operational semantics of a Scheme-like language and an ML-like language are unified into a multi-language and syntactic boundary terms are added to mediate transfer of control and data between the two languages. Using this embedding approach, they were able to establish the type-soundness of the multi-language. Their approach relied on a careful enforcement of soundness properties on the boundaries, for instance inserting dynamic guards or contracts to ensure soundness when typed ML values are transferred to untyped Scheme.

We introduce the notion of *sound inference interoperability*: whereas sound interoperability of traditional languages ensures that multi-language programs are type-sound, sound *inference* interoperability ensures that probabilistic multi-language programs correctly represent the intended probability distribution. Our main goal will be to establish sound inference interoperability of two PPLs: the first is called Disc and is similar to Dice, and the second is called Cont and it makes use of importance-sampling-based inference. These two languages are a nice pairing: Disc provides scalable exact discrete inference at the expense of expressivity, as it does not support continuous random variables and unbounded loops. On the other hand, Cont provides significant expressivity (it supports continuous random variables) at the cost of relying on importance-sampling-based approximate inference. Following Matthews and Findler [31], we embed both Disc and Cont into a multi-language we call MultiPPL. Together, these two languages cover a broad spectrum of interesting probabilistic programs that are difficult to handle today. We will show in Section 2 and Section 4 that examples from networking and probabilistic graphical models benefit from the ability to flexibly use different languages and inference algorithms within a unified multi-language. Traditional multi-language semantics establishes sound interoperability by proving type-soundness of the combined multi-language [31]. Analogously, we establish sound inference interoperability between Disc and Cont, guaranteeing that well-typed MultiPPL programs correctly represent the intended probability distribution. Our contributions are as follows:

- We introduce MultiPPL, a multi-language in the style of Matthews and Findler [31] that enables interoperation between a discrete exact probabilistic programming language Disc and a continuous approximate probabilistic programming language Cont.
- In Section 3 we construct two models of MultiPPL by combining appropriate semantic domains for Disc and Cont programs: a high-level model capturing the probability distribution intended by a given MultiPPL program, and a low-level model capturing the details of our particular implementation strategy. We then prove that these two semantics agree, establishing correctness of the implementation (Theorem 3.6). We identify two key requirements for ensuring sound inference interoperability between exact and approximate programs: Disc programs must additionally enforce *sample consistency* for ensuring Disc values pass safely into Cont, and Cont programs must additionally perform *importance weighting* for ensuring the safety of Disc conditioning.
- In Section 4 we validate the practical effectiveness of MultiPPL through our provided implementation. We evaluate the efficacy of MultiPPL by modeling complex independence structures through real-world problems in the domain of networking and probabilistic graphical models.




```
1 function step() {
2   let lambda = uniform(0, 5) in
3   let numPackets = poisson(lambda) in
4   for i in 0..numPackets {
5     forwardPacket(R1)
6   }
7 }
```

Fig. 1. A small network and a fragment of a probabilistic program encoding of the packet arrival problem.

Disc	Expressions	M, N	::=	$X \mid \mathsf{true} \mid \mathsf{false} \mid M \wedge N \mid \neg M$
			$\mid$	$\langle\rangle \mid \langle M, N\rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M$
			$\mid$	$\mathsf{ret}\ M \mid \mathsf{let}\ X\ \mathsf{be}\ M\ \mathsf{in}\ N \mid \mathsf{if}\ e\ \mathsf{then}\ M\ \mathsf{else}\ N$
			$\mid$	$\mathsf{flip}\ e \mid \mathsf{observe}\ M \mid (\!\mid e \mid\!)_E$
	Types	A, B	::=	$\mathsf{unit} \mid \mathsf{bool} \mid A \times B$
	Contexts	Δ	::=	$X_1 : A_1, \ldots, X_n : A_n$
Cont	Expressions	e	::=	$x \mid \mathsf{true} \mid \mathsf{false} \mid r \mid e_1 + e_2 \mid -e \mid e_1 \cdot e_2 \mid e_1 \leq e_2$
			$\mid$	$() \mid (e_1, e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$
			$\mid$	$\mathsf{ret}\ e \mid \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$
			$\mid$	$d \mid \mathsf{obs}(e_o, d) \mid (\!\mid M \mid\!)_S$
	Distributions	d	::=	$\mathsf{flip}\ e \mid \mathsf{uniform}\ e_1\ e_2 \mid \mathsf{poisson}\ e$
	Types	σ, τ	::=	$\mathsf{unit} \mid \mathsf{bool} \mid \mathsf{real} \mid \sigma \times \tau$
	Contexts	Γ	::=	$x_1 : \tau_1, \ldots, x_n : \tau_n$
	Number literals	r	$\in$	$\mathbb{R}$

Fig. 2. Syntax of MultiPPL. In Disc, we require $e \in [0, 1]$ for flip. In Cont, the syntax of distributions d denotes probability distributions. In obs these terms condition, otherwise they are immediately sampled.

We provide insights into MultiPPL's approach to probabilistic inference and characterize the nuanced landscape that interoperation produces.

2 Overview

We argue that it is often the case that realistic probabilistic programs consist of sub-programs that are best handled by *different* probabilistic programming languages. Consider, for example, the *packet arrival* situation visualized in Fig. 1. In this set up, at each time step, network packets are arriving according to a Poisson distribution, a fairly standard setup in discrete-time queueing theory [32]. Then, each packet is forwarded through the network, whose topology is visualized as a directed graph. The goal is to query for various properties about the network's behavior: for instance, the probability of a packet reaching the end of the network, or of a packet queue overflowing. This example task is inspired by prior work on using probabilistic programming languages to perform network verification [18, 53].

The situation in Fig. 1 is a small illustrative example of packet arrival, but programs like it are extremely challenging for today's PPLs because they mix different kinds of program structure. Lines 2 and 3 manipulate continuous and countably-infinite-domain random variables, which precludes the use of Dice. However, graph reachability and queue behavior are complex discrete distributions, which are difficult for Stan due to their inherent non-differentiability and high-dimensional discrete

structure. In order to scale on this example, we would like to be able to use an inference algorithm like Stan's for lines 2 and 3, and an inference algorithm like Dice's for lines 4–6.

Our approach to designing a language capable of handling situations like that described in Fig. 1 is to enable the programmer to seamlessly transition between programming in two different PPLs: Cont, an expressive language that supports sampling-based inference and continuous random variables, and Disc, a restricted discrete-only language that supports scalable exact discrete inference. Following Matthews and Findler [31], we describe a *probabilistic multi-language* that embeds both languages into a single unified syntax: see Fig. 2. In Section 3.1 we discuss the intricacies of the syntax in Fig. 2 in full detail, including the typing judgments found in Fig. 7 and the appendix; here we briefly note its high-level structure and discuss examples.

These languages delineate our two syntactic categories:

(1) Disc terms, shown in purple, that support discrete probabilistic operations such as Bernoulli random variables and Bayesian conditioning. The syntax is standard for an ML-like functional language with the addition of probabilistic constructs: flip e introduces a Bernoulli random variable that is true with probability $e \in [0, 1]$ and false otherwise; the construct observe M conditions on M. Notably, Disc lacks introduction forms for continuous random variables or real numbers, and so in order to define the Bernoulli-distributed random variable using flip, we must rely on interoperation to construct our distribution.
(2) Cont terms, shown in orange, additionally support standard continuous operations and sampling capabilities from two distributions inexpressible in Disc: a Uniform distribution `uniform` e_1 e_2 over the interval $[e_1, e_2]$, with $e_1, e_2 \in \mathbb{R}$, and a Poisson distribution `poisson` with rate $e \in \mathbb{R}$ being greater than zero. The syntax $\texttt{obs}(e_o, d)$ denotes conditioning on the event that a sample drawn from distribution d is equal to e_o.

Mediating between the Disc and Cont sublanguages are the boundaries $(\!| e |\!)_E$ and $(\!| M |\!)_S$: the boundary $(\!| e |\!)_E$ allows passing from Cont to Disc, and the boundary $(\!| M |\!)_S$ allows passing from Disc to Cont. This style of presentation is similar to Patterson [40].

Listing 1. TwoCoins

```
1  let θ be uniform 0 1 in
2  (|let X be flip θ in
3   let Y be flip θ in
4   observe X ∨ Y in
5   ret X |)_S
```

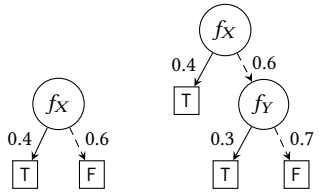
Listing 1 shows an example program in our multi-language which passes a uniformly-sampled real value θ from Cont into Disc and uses the resulting value as a prior for sampling two independent Bernoulli random variables. The outer-most language is Cont. On Line 1, θ is bound to a sample drawn from the uniform distribution on the unit interval. Then, on Lines 2–5, we begin evaluation of a Disc program inside the boundary term $(\!| - |\!)_S$. We flip two coins X and Y (Lines 2 and 3, respectively) in the Disc sublanguage, whose prior parameters are both θ. On Line 4, we observe that one of the two coins was true, taking advantage of syntactic sugar where observe is bound to a discarded variable name. Line 5 brings us to the final line of our program, where we query for the probability that X is true. The next two sub-sections will explain our approach to bridging the two languages.

2.1 Disc and Cont Inference

Before we describe the intricacies of language interoperation, we first provide some high-level intuition for how we wish to perform inference on Disc and Cont independently. First, we give a denotational semantics for MultiPPL that we denote $[\![-]\!]$ which associates each MultiPPL term with a probability distribution on MultiPPL values (see Section 3 for a formal definition of these semantics). Here we will briefly illustrate these semantics by example: the semantics $[\![\texttt{flip}\ p]\!]$ produces a Bernoulli distribution that is true with probability $p \in [0, 1]$; the semantics $[\![\texttt{uniform}\ e_1\ e_2]\!]$ produces a uniform distribution on the interval $[e_1, e_2] \in \mathbb{R}$.

```
1  let X be flip 0.4 in
2  let Y be flip 0.3 in
3  observe X ∨ Y in
4  ret X
```

(a) Example Disc program.



(b) BDD representations of formulas.

Fig. 3. Motivating example showing the compilation of the Disc program in 3a to BDDs in 3b. On the left of 3b is a BDD representing the distribution formula $\varphi = f_X$; on the right is the BDD representing the accepting formula $\alpha = f_X \vee f_Y$. T and F represent true and false values, respectively.

The goal of inference is to efficiently evaluate the denotation of a probabilistic program. While Disc and Cont share a unified denotation, they have very different approaches to inference. The key advantage of our multi-language approach is that we can specialize the design of Cont and Disc to take full advantage of structural differences between their underlying inference algorithms: for Disc we will use an exact inference strategy based on knowledge compilation similar to Dice [23], and for Cont we will rely on approximate inference via sampling. In the next two subsections we give a high-level overview of these standard approaches.

2.1.1 Exact Inference via Knowledge Compilation. Here, we illustrate the principles of exact inference in Disc via example; Section 3 provides a formal treatment of these semantics. In Fig. 3a, we reproduce the Disc program compiled in Lines 3–5 of Listing 1, but instantiate the priors of Y and Z with numeric literals 0.4 and 0.3, respectively. Our example in Fig. 3a denotes the probability distribution of Bernoulli 0.3, given that one of the two weighted coin flips is true; its semantics is $[\![\textit{Fig. 3a}]\!]\,(\text{true}) = \frac{0.4}{0.58} \approx 0.689$.

The exact inference strategy used by Disc is to perform probabilistic inference via weighted model counting [10, 14, 46], following Holtzen et al. [23]. The key idea is to interpret the probabilistic program as a weighted Boolean formula whose models are in one-to-one correspondence with paths through the program, and where each path is associated with a weight that matches the probability of that path in the program. Concretely, a *weighted Boolean formula* is a pair (φ, w) where φ is a Boolean formula and w is a *weight function* that associates literals (assignments to variables) in φ with real-valued weights. Then, the *weighted model count* of a weighted Boolean formula is the weighted sum of models:

$$\text{WMC}(\varphi, w) = \sum_{\{m \models \varphi\}} \prod_{\{\ell \in m\}} w(\ell). \tag{1}$$

To perform Disc inference by reduction to weighted model counting, we associate each Disc program with a pair of Boolean formulae in a manner similar to Holtzen et al. [23]: (1) an *accepting formula* α that encodes the paths through the program that does not violate observations; and (2) a *distribution formula* φ such that $\text{WMC}(\varphi \wedge \alpha)$ is the unnormalized probability of the program returning true. For instance, we would compile Fig. 3a into accepting formula $\varphi = f_X$ and $\alpha = f_X \vee f_Y$, where f_X is a Boolean variable that represents the outcome of flip 0.4 and f_Y represents the outcome of flip 0.3. Then, the weight function is $w(f_X) = 0.4$, $w(\overline{f_X}) = 0.6$, $w(f_Y) = 0.4$, $w(f_Y) = 0.3$, $w(\overline{f_Y}) = 0.7$. Then, we can compute the semantics of Fig. 3a by performing weighted model counting:

$$[\![\textit{Fig. 3a}]\!]\,(\text{true}) = \frac{\text{WMC}(\varphi \wedge \alpha, w)}{\text{WMC}(\alpha, w)} = \frac{\text{WMC}(f_X, w)}{\text{WMC}(f_X \vee f_Y, w)} = \frac{0.4}{0.4 + 0.6 \cdot 0.3} = \frac{0.4}{0.58} \approx 0.689$$

The weighted model counting task is well-studied, and there is an array of high-performance implementations for solving it [10, 23, 46]. One approach that is particularly effective is *knowledge compilation*, which compiles the Boolean formula into a representation for which weighted model counting can be performed efficiently (typically, polynomial-time in the size of the compiled representation). A common target for this compilation process is *binary decision diagrams* (BDDs), shown in Fig. 3b. A BDD is a rooted DAG whose internal nodes are labeled with Boolean variables and whose leaves are labeled with either true or false values. A BDD is read top-down: solid edges denote true assignments to variables, and dashed edges denote false assignments. Once a Boolean formula is compiled to a BDD, inference can be performed in polynomial time (in the size of the BDD) by performing a bottom-up traversal of the DAG.

While highly effective for discrete probabilistic inference tasks with finite domains, inference via knowledge compilation has a critical weakness: it cannot support continuous random variables or unbounded discrete random variables due to the requirement that each program be associated with a (finite) Boolean formula. Hence, the design of Disc must be carefully restricted to only permit programs that can be compiled to Boolean formulae, which is why it does not contain syntactic support for these features.

2.1.2 Approximate Inference via Sampling. A powerful alternative to exact inference is approximate inference via sampling. The engine that drives sampling-based inference is the expectation estimator. The expectation estimator is widely used as a foundation for approximate inference strategies for probabilistic programs [9, 11, 29, 33, 43, 55]. We will use it to give an inference algorithm for Cont. Concretely, suppose we want to use the expectation estimator to approximate the semantics of the Cont program $[\![\texttt{flip}\ 1/4]\!]$. To do this, we can draw $N = 100$ samples from the program: in roughly $1/4$ of these samples, the program will output true. This approach is known as *direct sampling*, and is one way of utilizing the expectation estimator to design approximate inference algorithms.

Formally, let Ω be a sample space, Pr a probability density function, and let $X : \Omega \to \mathbb{R}$ be a real-valued random variable out of the sample space. Then, the expectation of X is defined as $\mathbb{E}_{\Pr}[X] = \int \Pr(\omega)X(\omega)d\omega$. The *expectation estimator* approximates the expectation of a random variable X by drawing N samples from Pr:

$$\mathbb{E}_{\Pr}[X] \approx \frac{1}{N}\sum_{x\sim\Pr}^{N} X(x). \tag{2}$$

There are many more advanced approaches to sampling-based inference beyond direct sampling such as Hamiltonian Monte-Carlo [9, 34]; at their core, all these approximate inference algorithms follow the same principle of drawing some number of samples from the program and using that to estimate the semantics.

When compared with the exact inference strategy described in Section 2.1.1, sampling-based inference has the key advantage that it only requires the ability to sample from the probabilistic program: each time a random quantity is introduced, it can be dealt with by eagerly sampling. This makes sampling an ideal inference algorithm for implementing flexible and expressive languages with many features: unlike Disc, it is straightforward to add interesting features like continuous random variable and unbounded loops to Cont without wholesale redesigning of its inference algorithm. This gain in expressivity comes at the cost of precision: unlike Disc, Cont is only able to provide an approximation to the final expectation.

2.2 Sound Interoperation

We now move on to our main goal, establishing sound interoperation between the underlying inference strategies of Disc and Cont by identifying two key invariants that must be maintained

```
1 let x be flip 0.20 in
2 ⟪let Y be flip 0.25 in
3   observe ⟪x⟫_E ∨ Y in
4   ret Y⟫_S
```

(a) Motivating example.

```
1 ⟪let Y be flip 0.25 in
2   observe true ∨ Y in
3   ret Y⟫_S
```

(b) Sampled x = true.

```
1 ⟪let Y be flip 0.25 in
2   observe false ∨ Y in
3   ret Y⟫_S
```

(c) Sampled x = false.

Fig. 4. Interpreting Cont values in Disc.

when transporting Disc and Cont values across boundaries: *importance weighting* and *sample consistency*. At first, there appears to be a straightforward way of establishing interoperation between these two languages: when a Cont value v_s is interpreted in a Disc context, it is lifted in a Dirac-delta distribution on v_s. Figure 4a gives an illustration of this scenario: first, on Line 1 we sample a value (either true or false) for x. Then, on Line 3 we interpret x within an exact context. Figures 4b and 4c show the two possible liftings: the sampled value is given its straightforward interpretation in the exact context. When a Disc value v_e is interpreted in a Cont context, one can draw a sample v_s from the *exact* distribution denoted by v_e. However, we will show in the next two sub-sections that a naive approach that fails to preserve key invariants will result in incorrect inference results, and that one must maintain careful invariants in order to ensure soundness of inference across boundaries.

2.2.1 Importance Weighting. Let us more carefully consider the situation shown in Fig. 4a. First, we observe that the desired semantics is $[\![\textit{Fig. 4a}]\!]\,(\text{true}) = 0.25/0.4 = 0.625$. Suppose we were to follow a naive multi-language inference procedure of drawing 100 samples by eagerly evaluating values for x. Following Section 2.1.2, approximately 20 of these samples will yield the program in Fig. 4b and approximately 80 will yield the program in Fig. 4c. Observe that $[\![\textit{Fig. 4b}]\!]\,(\text{true}) = 0.25$ and $[\![\textit{Fig. 4c}]\!]\,(\text{true}) = 1$. So our naive estimate $[\![\textit{Fig. 4a}]\!]\,(\text{true})$ would be:

$$[\![\textit{Fig. 4a}]\!]\,(\text{true}) \overset{?}{\approx} \frac{20}{100}\,[\![\textit{Fig. 4b}]\!]\,(\text{true}) + \frac{80}{100}\,[\![\textit{Fig. 4c}]\!]\,(\text{true}) = 0.85 \tag{3}$$

Something went wrong – we expected the result of Eq. (3) to be 0.625. This naive sampling approach significantly over-estimated $[\![\textit{Fig. 4a}]\!]\,(\text{true})$. The issue is that, in this naive approach, the observation that occurs on Line 3 (Fig. 4a) is not taken into account when sampling a value for x: samples where x = true are under-sampled relative to their true probability, and samples where x = false are over-sampled.

This example illustrates that a naive approach to interoperation is unsound. To fix it, one approach is to adjust the *relative importance* of each sample: we will still sample x = true roughly 20% of the time, but we will decrease the overall importance of this sample. The key idea comes from *importance sampling*, which is a refinement of the expectation estimator given in Eq. (2) but enables estimating an expectation $\mathbb{E}_p[X]$ by sampling from a *proposal distribution* q:

$$\mathbb{E}_p[X] = \int X(x)p(x)dx = \int X(x)\frac{p(x)}{q(x)}q(x)dx = \mathbb{E}_q\left[X(x)\frac{p(x)}{q(x)}\right]. \tag{4}$$

The above holds as long as the proposal q *supports* p (i.e., satisfies the property that, for all x, if $p(x) > 0$ then $q(x) > 0$). The ratio $p(x)/q(x)$ is called the *importance weight* of the sample x: intuitively, if x is more likely according to the true distribution p than the proposal q, the importance ratio will be greater than 1; similarly, if x is less likely according to p than q, its weight will be less than 1. In this instance, the proposal q is semantics of the program with all observe statements deleted, and p is $[\![\textit{Fig. 4a}]\!]$.

Unfortunately, already having access to p defeats the purpose of approximating. In addition, our programs p always incorporate a normalization constant Z, such that

$$p(x) = \hat{p}(x)/Z, \tag{5}$$

with $\hat{p}$ being the unnormalized distribution. Summing the probability of $\hat{p}$ for all x in the domain of p yields $Z = \int \hat{p}(x)dx$. Computing this normalization constant is expensive, and amounts to calculating p directly. In our setting, calculating this normalization constant is identical to the denotation of Line 3 in the exact setting. To avoid solving for this in our importance sampler, we can incorporate Eq. (5) into our expectation Eq. (4) and jointly approximate our query alongside Z,

$$\mathbb{E}_p[X] = \int X(x)p(x)dx = \frac{\int X(x)\hat{p}(x)dx}{\int \hat{p}(x)dx} = \frac{\int X(x)\frac{\hat{p}(x)}{q(x)}q(x)dx}{\int \frac{\hat{p}(x)}{q(x)}q(x)dx} = \frac{\mathbb{E}_{x\sim q}\left[X(x)\frac{\hat{p}(x)}{q(x)}\right]}{\mathbb{E}_{x\sim q}\left[\frac{\hat{p}(x)}{q(x)}\right]}. \tag{6}$$

The above is called a *self-normalized importance sampler* [44]. Here, in the denominator, we construct the normalizing constant for q to be the ratio of the unnormalized $\hat{p}$ to q: the Line 2 in Fig. 4b when x = true and the Line 2 in Fig. 4c when x = false. Notice that the probability of evidence encoded by observe statements in Fig. 4b and Fig. 4b are $[\![\text{true} \vee Y]\!]\,(\text{true}) = 1$ and $[\![\text{false} \vee Y]\!]\,(\text{true}) = 0.25$, respectively.

Sampling 100 draws of x, again, with 20 samples yielding the program in Fig. 4b and 80 samples yielding the program in Fig. 4c, Eq. (6) now returns our expected result:

$$[\![\textit{Fig. 4a}]\!]\,(\text{true}) \approx \frac{\frac{20}{100}\,1 \cdot [\![\textit{Fig. 4b}]\!]\,(\text{true}) + \frac{80}{100}\,0.25 \cdot [\![\textit{Fig. 4c}]\!]\,(\text{true})}{\frac{20}{100}\,1 + \frac{80}{100}\,0.25} = \frac{20 \cdot 0.25 + 80 \cdot 0.25}{20 + 20} = 0.625$$

2.2.2 Sample Consistency. Importance weighting is not all that is necessary to ensure sound interoperability: we must also ensure that Disc values are safely interpreted with a Cont context. Consider the example in Listing 2. There are two observations to make about this program. The first is that we embed a Cont program into a Disc context; this results in a sampler that evaluates all Cont fragments while preserving the semantics of all Disc variables in order to produce a sample. The next thing to notice is that a Disc program denotes a distribution; in the semantics of Cont, when we come across a distribution a sample is immediately drawn from it.

Listing 2. Sample consistency

```
1  let X be flip 0.5 in
2  (| let y be (| X |)_S in
3    let z be (| X |)_S in
4    ret y ∧ z |)_E
```

Again, we can propose a naive strategy for performing inference on this program: one where we draw a new sample each time we encounter a distribution. Notice that Line 2 holds a reference X to flip 0.5, denoting a Bernoulli distribution. When we evaluate this boundary, with probability 1/2 we sample y = true; suppose we sample y = true. We encounter this reference, again, on Line 3 and suppose we sample z = false. Finally, on Line 4, we evaluate the Boolean expression, resulting in false, which is lifted into the Dirac-delta distribution in Disc. Running this program n number of times, we will expect to see the expectation of $y \wedge z$ with y and z as two independent draws of the fair Bernoulli distribution. At this point, something strange has occurred: by referencing a single variable in Disc, we have simulated two independent flips.

Intuitively, the sampled value for z must be *the same* as the sampled value for y. Operationally, to ensure this is the case, any samples drawn across at the $(|-|)_S$ boundary must additionally constrain a Disc program's accepting criteria so that all subsequent samples remain consistent.

3 MultiPPL: Multi-Language Probabilistic Programming

In this section we present MultiPPL, a multilanguage that supports both exact and sampling-based inference. Sections 3.1 and 3.2 describe the syntax of MultiPPL programs and MultiPPL's type system. We then present two semantic models of MultiPPL. First, Section 3.3 presents a high-level model $\mathcal{H}[\![-]\!]$ capturing the probability distribution generated by a MultiPPL program; this model specifies the intended behavior of our implementation. Second, Section 3.4 presents a low-level model $\mathcal{L}[\![-]\!]$ capturing our particular inference strategy, taking the intuition we have built up in Section 2.2 and providing the precise way in which our implementation combines knowledge compilation with importance sampling. Finally, Section 3.5 connects these two models: we show that $\mathcal{L}[\![-]\!]$ soundly refines $\mathcal{H}[\![-]\!]$, establishing sound inference interoperability between Disc and Cont with respect to our inference strategy.

3.1 Syntax

The syntax of MultiPPL is given in Fig. 2. MultiPPL is a union of two sublanguages, Disc and Cont, that support exact and sampling-based inference. To streamline the presentation of the models in Sections 3.3 and 3.4, each sublanguage is then subdivided into pure and effectful fragments.

The sampling-based sublanguage Cont is a first-order probabilistic programming language with Booleans, tuples, and real numbers. In Cont, the pure fragment includes not only the basic operations on Booleans and pairs, but also arithmetic operations on real numbers. The effectful fragment additionally includes primitive operations `uniform` e_1 e_2 for generating uniformly-distributed real numbers in the interval $[e_1, e_2]$, `poisson` e for generating Poisson-distributed integers with rate e, and obs-expressions denoting conditioning operators for these distributions.

The exact sublanguage Disc, reminiscent of Dice [23], is a discrete first-order probabilistic programming language with Booleans and tuples. The pure fragment of Disc includes the basic operations on Booleans and pairs, while the effectful fragment includes constructs for sequencing and branching, as well as the primitive operations flip e – for generating Bernoulli-distributed Booleans with parameter e of type `real`, and observe M – for conditioning on an event M.

The Disc branching construct if e then M else N requires the guard e to be a Cont term. This is not an essential restriction, but rather required for sound inference interoperability with respect to the specific implementation strategy we have chosen. As sketched in Sections 2.1.1 and 2.1.2, standard sampling-based inference maintains a weight for the current trace, while exact inference maintains a weight map and an accepting formula. In our implementation, we wanted a language whose inference algorithm would stay as close to these traditional inference algorithms as possible while avoiding incorrect weighting schemes. To do this while maintaining safe inference interoperability, one must have the rather subtle invariant that if-then-else expressions in the Disc sublanguage have then- and else- branches that importance-weight their respective traces by the same amount. The syntactic restriction on if e then M else N is a simple way of ensuring this is always the case: probabilistic choice is removed, and only one branch need ever be considered. In our implementation, we also permit if-then-else expressions where both branches are boundary-free Disc programs, as exact inference for such programs can be performed just as in Holtzen et al. [23], without touching the importance weight. These special cases could be avoided by maintaining an auxiliary Boolean formula tracking a *path condition*, which encodes during inference the then- and else- branches of if-then-else expressions taken to reach a given subterm. This would allow arbitrary if-then-else expressions in the Disc sublanguage, at the expense of additional overhead of maintaining this path condition during inference. In our design of MultiPPL, we have decided to restrict the syntax of the language rather than impose a performance cost; in practice, this has been sufficient to express all of the examples in Section 4.

$$\boxed{A \leftrightsquigarrow \tau}$$

$$\frac{}{\texttt{unit} \leftrightsquigarrow \texttt{unit}} \qquad \frac{}{\texttt{bool} \leftrightsquigarrow \texttt{bool}} \qquad \frac{A \leftrightsquigarrow \tau \qquad B \leftrightsquigarrow \sigma}{A \times B \leftrightsquigarrow \tau \times \sigma}$$

Fig. 5. Rules for convertibility between Disc types A and Cont types τ.

$$\boxed{\Gamma; \Delta \vdash_{\mathrm{c}} M : A}$$

$$\frac{\Delta \vdash M : A}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{ret}\ M : A} \qquad \frac{\Gamma; \Delta \vdash_{\mathrm{c}} M : A \qquad \Gamma; \Delta, X : A \vdash_{\mathrm{c}} N : B}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{let}\ X\ \mathsf{be}\ M\ \mathsf{in}\ N : B}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma; \Delta \vdash_{\mathrm{c}} M : A \qquad \Gamma; \Delta \vdash_{\mathrm{c}} N : A}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{if}\ e\ \mathsf{then}\ M\ \mathsf{else}\ N : A} \qquad \frac{\Gamma \vdash e : \texttt{real}}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{flip}\ e : \texttt{bool}} \qquad \frac{\Delta \vdash M : \texttt{bool}}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{observe}\ M : \texttt{unit}}$$

$$\frac{\Gamma; \Delta \vdash_{\mathrm{c}} e : \tau \qquad A \leftrightsquigarrow \tau}{\Gamma; \Delta \vdash_{\mathrm{c}} (\!| e |\!)_E : A}$$

$$\boxed{\Gamma; \Delta \vdash_{\mathrm{c}} e : \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{ret}\ e : \tau} \qquad \frac{\Gamma; \Delta \vdash_{\mathrm{c}} e_1 : \sigma \qquad \Gamma, x : \sigma; \Delta \vdash_{\mathrm{c}} e_2 : \tau}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma; \Delta \vdash_{\mathrm{c}} e_2 : \tau \qquad \Gamma; \Delta \vdash_{\mathrm{c}} e_3 : \tau}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau} \qquad \frac{\Gamma \vdash e : \texttt{real}}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{flip}\ e : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{real} \qquad \Gamma \vdash e_2 : \texttt{real}}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{uniform}\ e_1\ e_2 : \texttt{real}} \qquad \frac{\Gamma \vdash e : \texttt{real}}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{poisson}\ e : \texttt{real}}$$

$$\frac{\Gamma \vdash e_o : \texttt{bool} \qquad \Gamma \vdash e_1 : \texttt{real}}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{obs}(e_o, \mathsf{flip}\ e_1) : \texttt{unit}} \qquad \frac{\Gamma \vdash e_o : \texttt{real} \qquad \Gamma \vdash e_1 : \texttt{real} \qquad \Gamma \vdash e_2 : \texttt{real}}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{obs}(e_o, \mathsf{uniform}\ e_1\ e_2) : \texttt{unit}}$$

$$\frac{\Gamma \vdash e_o : \texttt{real} \qquad \Gamma \vdash e : \texttt{real}}{\Gamma; \Delta \vdash_{\mathrm{c}} \mathsf{obs}(e_o, \mathsf{poisson}\ e) : \texttt{unit}} \qquad \frac{\Gamma; \Delta \vdash_{\mathrm{c}} M : A \qquad A \leftrightsquigarrow \tau}{\Gamma; \Delta \vdash_{\mathrm{c}} (\!| M |\!)_S : \tau}$$

Fig. 6. Typing rules for the effectful fragment of MultiPPL.

3.2 Typing

The syntax of types and typing contexts is given in Fig. 2. Disc types A include Booleans and pairs; Cont types τ additionally include a type of real numbers. A Disc typing context Δ is a mapping of Disc variables to Disc types, and a Cont typing context Γ is a mapping of Cont variables to Cont types. By convention we will denote Disc syntactic elements with capital letters and Cont elements with lower-case Greek letters. This section is best read in color, where we use orange monotype font for Cont terms and purple sans-serif font for Disc terms.

MultiPPL contains two sublanguages that each have a pure and effectful part, so there are correspondingly four forms of typing judgment. For the pure fragments,

$$\llbracket \mathtt{unit} \rrbracket = \{\star\} \qquad \llbracket \mathtt{unit} \rrbracket = \{\star\}$$
$$\llbracket \mathtt{bool} \rrbracket = \{\top, \bot\} \qquad \llbracket \mathtt{bool} \rrbracket = \{\top, \bot\}$$
$$\llbracket A \times B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket \qquad \llbracket \mathtt{real} \rrbracket = \mathbb{R}$$
$$\llbracket \sigma \times \tau \rrbracket = \llbracket \sigma \rrbracket \times \llbracket \tau \rrbracket \qquad \llbracket \Delta \rrbracket = \prod_{X \in \mathrm{dom}\,\Delta} \llbracket \Delta(X) \rrbracket \qquad \llbracket \Gamma \rrbracket = \prod_{x \in \mathrm{dom}\,\Gamma} \llbracket \Gamma(x) \rrbracket$$

Fig. 7. Interpreting types and typing contexts.

$$\llbracket \Delta \vdash M : A \rrbracket : \llbracket \Delta \rrbracket \to \llbracket A \rrbracket$$
$$\llbracket X \rrbracket(\delta) = \delta(X)$$
$$\llbracket \mathtt{true} \rrbracket(\delta) = \top$$
$$\llbracket \mathtt{false} \rrbracket(\delta) = \bot$$
$$\llbracket M \wedge N \rrbracket(\delta) = \begin{cases} \top, & \text{if } \llbracket M \rrbracket(\delta) = \llbracket N \rrbracket(\delta) = \top \\ \bot, & \text{otherwise} \end{cases}$$
$$\llbracket \neg M \rrbracket(\delta) = \begin{cases} \bot, & \text{if } \llbracket M \rrbracket(\delta) = \top \\ \top, & \text{otherwise} \end{cases}$$
$$\llbracket \langle\rangle \rrbracket(\delta) = \star$$
$$\llbracket \langle M, N \rangle \rrbracket(\delta) = (\llbracket M \rrbracket(\delta), \llbracket N \rrbracket(\delta))$$
$$\llbracket \mathtt{fst}\ M \rrbracket(\delta) = \pi_1(\llbracket M \rrbracket(\delta))$$
$$\llbracket \mathtt{snd}\ M \rrbracket(\delta) = \pi_2(\llbracket M \rrbracket(\delta))$$

$$\llbracket \Gamma \vdash e : \tau \rrbracket : \llbracket \Gamma \rrbracket \xrightarrow{\text{measurable}} \llbracket \tau \rrbracket$$
$$\llbracket x \rrbracket(\gamma) = \gamma(x)$$
$$\llbracket \mathtt{true} \rrbracket(\gamma) = \top$$
$$\llbracket \mathtt{false} \rrbracket(\gamma) = \bot$$
$$\llbracket r \rrbracket(\gamma) = r$$
$$\llbracket e_1 + e_2 \rrbracket(\gamma) = \llbracket e_1 \rrbracket(\gamma) + \llbracket e_2 \rrbracket(\gamma)$$
$$\llbracket - e \rrbracket(\gamma) = - \llbracket e \rrbracket(\gamma)$$
$$\llbracket e_1 \cdot e_2 \rrbracket(\gamma) = \llbracket e_1 \rrbracket(\gamma) \cdot \llbracket e_2 \rrbracket(\gamma)$$
$$\llbracket e_1 \le e_2 \rrbracket(\gamma) = \begin{cases} \top, & \text{if } \llbracket e_1 \rrbracket(\gamma) \le \llbracket e_2 \rrbracket(\gamma) \\ \bot, & \text{otherwise} \end{cases}$$
$$\llbracket () \rrbracket(\gamma) = \star$$
$$\llbracket (e_1, e_2) \rrbracket(\gamma) = (\llbracket e_1 \rrbracket(\gamma), \llbracket e_2 \rrbracket(\gamma))$$
$$\llbracket \mathtt{fst}\ e \rrbracket(\gamma) = \pi_1(\llbracket e \rrbracket(\gamma))$$
$$\llbracket \mathtt{snd}\ e \rrbracket(\gamma) = \pi_2(\llbracket e \rrbracket(\gamma))$$

Fig. 8. Interpreting pure terms.

- $\Delta \vdash M : A$ says the pure Disc term M has Disc type A in Disc context Δ.
- $\Gamma \vdash e : \tau$ says the pure Cont term e has Cont type τ in Cont context Γ.

These judgments are standard and deferred to the appendix.

The typing judgments for effectful MultiPPL terms are parameterized by a combined context $\Gamma; \Delta$, as an effectful term may mention variables from both Disc and Cont via boundaries:

- $\Gamma; \Delta \vdash_c M : A$ says the effectful Disc term M has Disc type A in combined context $\Gamma; \Delta$.
- $\Gamma; \Delta \vdash_c e : \tau$ says the effectful Cont term e has Cont type τ in combined context $\Gamma; \Delta$.

These judgments are defined in Fig. 6. Note that in the rule for flip e, the parameter e can be an arbitrary pure Cont term; this allows expressing the TwoCoins example from Section 2. In principle, one could allow arbitrary effectful Cont programs e as parameter to flip instead of just pure ones, but we have not found this to be useful in practice. The typing judgments for the boundaries $(\!|\, e \,|\!)_E$ and $(\!|\, M \,|\!)_S$ allow converting Disc terms of type A into Cont terms of type τ and vice versa, so long as A and τ are *convertible*, written $A \leftrightsquigarrow \tau$. The convertibility relation is defined in Fig. 5; it simply states that Disc types can be converted into their Cont counterparts in the expected way, and that the Cont type `real` has no Disc counterpart.

3.3 High-Level Semantic Model

This section defines a high-level model $\mathcal{H}\llbracket - \rrbracket$ of MultiPPL to serve as the definition of sound inference interoperability for the MultiPPL multilanguage.

$$\boxed{\mathcal{H}[\![\Gamma;\Delta \vdash_{\mathrm{c}} M : A]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \to \mathrm{Dist}_{\mathrm{w}}[\![A]\!]}$$

$$\mathcal{H}[\![\texttt{ret}\ M]\!](\gamma,\delta) = \mathrm{ret}([\![M]\!](\delta))$$
$$\mathcal{H}[\![\texttt{let}\ X\ \texttt{be}\ M\ \texttt{in}\ N]\!](\gamma,\delta) = \begin{pmatrix} x \leftarrow \mathcal{H}[\![M]\!](\gamma,\delta); \\ \mathcal{H}[\![N]\!](\gamma,\delta[X \mapsto x]) \end{pmatrix}$$
$$\mathcal{H}[\![(\!|\, e \,|\!)_E]\!](\gamma,\delta) = \mathcal{H}[\![e]\!](\gamma,\delta)$$

$$\mathcal{H}\left[\!\!\left[\begin{matrix}\texttt{if}\ e \\ \texttt{then}\ M \\ \texttt{else}\ N\end{matrix}\right]\!\!\right](\gamma,\delta) = \begin{pmatrix}\texttt{if}\ [\![e]\!](\gamma) \\ \texttt{then}\ \mathcal{H}[\![M]\!](\gamma,\delta) \\ \texttt{else}\ \mathcal{H}[\![N]\!](\gamma,\delta)\end{pmatrix}$$
$$\mathcal{H}[\![\texttt{flip}\ e]\!](\gamma,\delta) = \overline{\mathrm{flip}([\![e]\!](\gamma))}$$
$$\mathcal{H}[\![\texttt{observe}\ M]\!](\gamma,\delta) = \mathrm{score}(\mathbf{1}_{[\![M]\!](\delta)=\top})$$

$$\boxed{\mathcal{H}[\![\Gamma;\Delta \vdash_{\mathrm{c}} e : \tau]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \to \mathrm{Dist}_{\mathrm{w}}[\![\tau]\!]}$$

$$\mathcal{H}[\![\texttt{ret}\ e]\!](\gamma,\delta) = \mathrm{ret}([\![e]\!](\gamma))$$
$$\mathcal{H}[\![\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2]\!](\gamma,\delta) = \begin{pmatrix} x \leftarrow \mathcal{H}[\![e_1]\!](\gamma,\delta); \\ \mathcal{H}[\![e_2]\!](\gamma[x \mapsto x],\delta) \end{pmatrix}$$

$$\mathcal{H}\left[\!\!\left[\begin{matrix}\texttt{if}\ e_1 \\ \texttt{then}\ e_2 \\ \texttt{else}\ e_3\end{matrix}\right]\!\!\right](\gamma,\delta) = \begin{pmatrix}\texttt{if}\ [\![e_1]\!](\gamma) \\ \texttt{then}\ \mathcal{H}[\![e_2]\!](\gamma,\delta) \\ \texttt{else}\ \mathcal{H}[\![e_3]\!](\gamma,\delta)\end{pmatrix}$$
$$\mathcal{H}[\![(\!|\, M \,|\!)_S]\!](\gamma,\delta) = \mathcal{H}[\![M]\!](\gamma,\delta)$$

$$\mathcal{H}[\![\texttt{flip}\ e]\!](\gamma,\delta) = \overline{\mathrm{flip}([\![e]\!](\gamma))}$$
$$\mathcal{H}[\![\texttt{uniform}\ e_1\ e_2]\!](\gamma,\delta) = \overline{\mathrm{uniform}([\![e_1]\!](\gamma),[\![e_2]\!](\gamma))}$$
$$\mathcal{H}[\![\texttt{poisson}\ e]\!](\gamma,\delta) = \overline{\mathrm{poisson}([\![e]\!](\gamma))}$$
$$\mathcal{H}[\![\texttt{obs}(e_o,\ \texttt{flip}\ e_1)]\!](\gamma,\delta) = \mathrm{score}(\mathrm{flip}([\![e_1]\!](\gamma))([\![e_o]\!](\gamma)))$$
$$\mathcal{H}[\![\texttt{obs}(e_o,\ \texttt{poisson}\ e_1)]\!](\gamma,\delta) = \mathrm{score}(\mathrm{poisson}([\![e_1]\!](\gamma))([\![e_o]\!](\gamma)))$$
$$\mathcal{H}[\![\texttt{obs}(e_o,\ \texttt{uniform}\ e_1\ e_2)]\!](\gamma,\delta) = \mathrm{score}(\mathrm{uniform}([\![e_1]\!](\gamma),[\![e_2]\!](\gamma))([\![e_o]\!](\gamma)))$$

Fig. 9. Interpreting effectful terms. We use Haskell-style syntactic sugar for the usual monad operations.

Setting aside details of any particular inference strategy, a MultiPPL program $\bullet;\bullet \vdash_{\mathrm{c}} e : \tau$ should produce a conditional probability distribution over values of type τ. Following standard techniques for modelling probabilistic programs with conditioning [55], we interpret types and typing contexts as measurable spaces, pure terms as measurable functions, and effectful terms via a suitable monad.

Fig. 7 gives the interpretations of types and typing contexts. Disc types denote finite discrete measurable spaces and Cont types denote arbitrary measurable spaces. These interpretations are then lifted to typing contexts in the usual way: a Disc context Δ denotes the measurable space of substitutions δ such that $\delta(x) \in [\![\Delta(x)]\!]$ for all $x \in \mathrm{dom}\,\Delta$, and a Cont contexts Γ denotes the measurable space of substitutions γ such that $\gamma(x) \in [\![\Gamma(x)]\!]$ for all $x \in \mathrm{dom}\,\Gamma$.

Fig. 8 gives the standard interpretations of pure terms [55]. Pure Disc terms $\Delta \vdash M : A$ denote functions $[\![M]\!] : [\![\Delta]\!] \to [\![A]\!]$, automatically measurable because every Disc type denotes a discrete measurable space. Pure Cont terms $\Gamma \vdash e : \tau$ denote measurable functions $[\![e]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$.

Following Staton et al. [55], to interpret effectful terms we make use of the monad $\mathrm{Dist}_{\mathrm{w}}A = \mathrm{Dist}([0,1] \times A)$, obtained by combining the writer monad for the monoid $([0,1], \times, 1)$ of weights with the probability monad Dist [20]. Under this interpretation, a MultiPPL program $\bullet;\bullet \vdash_{\mathrm{c}} e : \tau$ denotes a distribution over pairs (w, v), where v is a value of type τ produced by a particular run of e and w is the weight accumulated by both Cont and Disc observe expressions.

Fig. 9 interprets effectful MultiPPL terms using $\mathrm{Dist}_{\mathrm{w}}$. A Disc term $\Gamma;\Delta \vdash_{\mathrm{c}} M : A$ is interpreted as a measurable function $\mathcal{H}[\![M]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \to \mathrm{Dist}_{\mathrm{w}}[\![A]\!]$, and a Cont term $\Gamma;\Delta \vdash_{\mathrm{c}} e : \tau$ is interpreted as a measurable function $\mathcal{H}[\![e]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \to \mathrm{Dist}_{\mathrm{w}}[\![\tau]\!]$. To model the basic probabilistic operations, the interpretation additionally makes use of the following primitives:

- $\overline{(\bullet)} : \mathrm{Dist}(A) \to \mathrm{Dist}_{\mathrm{w}}(A)$ lifts distributions on A into $\mathrm{Dist}_{\mathrm{w}}$ by setting $w = 1$.
- $\mathrm{score} : [0,1] \to \mathrm{Dist}_{\mathrm{w}}\{\star\}$ sends a weight w to the Dirac distribution $\delta_{(w,\star)}$ centered at $(w, \star)$.
- For $p \in \mathbb{R}$, $\mathrm{flip}(p)$ is the Bernoulli distribution on $\{\top, \bot\}$ with parameter p if $p \in [0,1]$ and the Dirac distribution $\delta_{\bot}$ otherwise.
- For $a, b \in \mathbb{R}$, $\mathrm{uniform}(a,b)$ is the uniform distribution on $[a,b]$ if $a \leq b$ and $\delta_{\min(a,b)}$ otherwise.
- For $\lambda \in \mathbb{R}$, $\mathrm{poisson}(\lambda)$ is the Poisson distribution with rate λ if $\lambda > 0$ and δ_0 otherwise.

Boundaries have no effect under this interpretation, reflecting the idea that changing one's inference strategy should not change the inferred distribution; semantic values of Disc type are implicitly coerced into semantic values of Cont type and vice versa, thanks to the following lemma:

Lemma 3.1 (natural embedding). *If* $A \leftrightsquigarrow \tau$ *then* $[\![A]\!] = [\![\tau]\!]$.

Proof. By induction on $A \leftrightsquigarrow \tau$. □

3.4 Low-Level Model

This section presents a low-level model $\mathcal{L}[\![-]\!]$ of MultiPPL, capturing the particular details of our inference strategy.

The interpretations of types, typing contexts, and the pure fragment of MultiPPL are identical to the ones given in Section 3.3. Where $\mathcal{L}[\![-]\!]$ differs from $\mathcal{H}[\![-]\!]$ is in the interpretation of effectful terms. Key to this interpretation is the construction of a suitable semantic domain for interpreting effectful terms in a way that faithfully reflects the details of our implementation. We construct this semantic domain by combining models of exact and sampling-based inference.

Our model of sampling-based inference is entirely standard, making use of the monad $\mathrm{Dist}_{\mathrm{w}}$ of Section 3.3. This monad captures the fact that a sampler performs inference by drawing weighted samples from the distribution defined by a probabilistic program [55].

Our model of exact inference, on the other hand, is novel. As explained in Section 2.1.1 and documented in full detail in Holtzen et al. [23], exact inference via knowledge compilation performs inference by maintaining two pieces of state: a *weight map* w associating Boolean literals to probabilities, and a Boolean formula α, called the *accepting formula*, that encodes the paths through the program that do not violate observe statements. The final result of knowledge compilation is itself a Boolean formula φ; the posterior distribution can then be calculated by performing weighted model counting on $\varphi \wedge \alpha$ and α with respect to the weight map w.

The defining trait of this knowledge compilation strategy is that it *maintains an exact representation of the underlying probability space throughout probabilistic program execution.* At any given moment during knowledge compilation, there is an underlying *sample space*: the space of models over the collection of Boolean variables generated so far. The purpose of the weight map w is to represent a *distribution* over this sample space: the probability of a given model can be computed by multiplying the weights of all of its literals. Together, the sample space and the weight map form a *probability space*, which is statefully manipulated throughout the knowledge compilation process. Upon each encounter of a flip command, the probability space grows: this is implemented by generating a fresh Boolean variable to represent the result of the flip and extending w accordingly. The purpose of the accepting formula α is to represent an *event* in this probability space: the event consisting of those models that satisfy α. Upon each encounter of an observe command, this event shrinks: this is implemented by conjoining the condition being observed onto α. Finally, the purpose of the output formula φ is to represent a *random variable*, which is to say a Boolean-valued function out of the sample space: the formula φ represents the random variable that takes values $\top$ for those models that satisfy φ and $\bot$ otherwise.

What is essential about this setup is that it *maintains a conditional probability space* (Ω, μ, E), consisting of a sample space Ω (the space of models), a probability measure μ on it (represented by

$$\mathcal{L}[\![\text{ret } M]\!](\Omega)(\gamma, D) = \text{ret}(\Omega, \text{id}, [\![M]\!] \circ D)$$

$$\mathcal{L}[\![\text{let } X \text{ be } M \text{ in } N]\!](\Omega)(\gamma, D) = \begin{pmatrix} (\Omega_1, f_1, X) \leftarrow \mathcal{L}[\![M]\!](\Omega)(\gamma, D); \\ (\Omega_2, f_2, Y) \leftarrow \mathcal{L}[\![N]\!](\Omega_1)(\gamma, (D \circ f_1)[X \mapsto X]); \\ \text{ret}(\Omega_2, f_1 \circ f_2, Y) \end{pmatrix}$$

$$\mathcal{L}[\![\text{if } e \text{ then } M \text{ else } N]\!](\Omega)(\gamma, D) = \begin{pmatrix} \text{if } [\![e]\!](\gamma) \\ \text{then } \mathcal{L}[\![M]\!](\Omega)(\gamma, D) \\ \text{else } \mathcal{L}[\![N]\!](\Omega)(\gamma, D) \end{pmatrix}$$

$$\mathcal{L}[\![\text{flip } e]\!](\Omega)(\gamma, D) = \begin{pmatrix} p := \text{if } [\![e]\!](\gamma) \in [0,1] \text{ then } [\![e]\!](\gamma) \text{ else } 0; \\ \Omega_{\text{flip}} := (\{0,1\}, \mu, \{0,1\}) \text{ where } \mu(1) = p; \\ \Omega' := \Omega \otimes \Omega_{\text{flip}}; \\ X := \omega' \mapsto \text{if } \pi_2(\omega') = 1 \text{ then } \top \text{ else } \bot; \\ \text{ret}(\Omega', \pi_1, X) \end{pmatrix}$$

$$\mathcal{L}[\![\text{observe } M]\!](\Omega, \mu, E)(\gamma, D) = \begin{pmatrix} F := ([\![M]\!] \circ D)^{-1}(\top); \\ \text{score}(\mu|_E(F)); \\ \text{ret}((\Omega, \mu, E \cap F), \text{id}, \_ \mapsto \star) \end{pmatrix}$$

$$\mathcal{L}[\![(\!| e |\!)_E]\!](\Omega)(\gamma, D) = \begin{pmatrix} (\Omega', f, x) \leftarrow \mathcal{L}[\![e]\!](\Omega)(\gamma, D); \\ \text{ret}(\Omega', f, \_ \mapsto x) \end{pmatrix}$$

$$\mathcal{L}[\![\texttt{ret } e]\!](\Omega)(\gamma, D) = \text{ret}(\Omega, \text{id}, [\![e]\!](\gamma))$$

$$\mathcal{L}[\![\texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2]\!](\Omega)(\gamma, D) = \begin{pmatrix} (\Omega_1, f_1, x) \leftarrow \mathcal{L}[\![e_1]\!](\Omega)(\gamma, D) \\ (\Omega_2, f_2, y) \leftarrow \mathcal{L}[\![e_2]\!](\Omega_1)(\gamma[x \mapsto x], D \circ f_1) \\ \text{ret}(\Omega_2, f_1 \circ f_2, y) \end{pmatrix}$$

$$\mathcal{L}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!](\Omega)(\gamma, D) = \begin{pmatrix} \text{if } [\![e_1]\!](\gamma) \\ \text{then } \mathcal{L}[\![e_2]\!](\Omega)(\gamma, D) \\ \text{else } \mathcal{L}[\![e_3]\!](\Omega)(\gamma, D) \end{pmatrix}$$

$$\mathcal{L}[\![\texttt{flip } e]\!](\Omega)(\gamma, D) = \begin{pmatrix} x \leftarrow \overline{\text{flip}([\![e]\!](\gamma))} \\ \text{ret}(\Omega, \text{id}, x) \end{pmatrix}$$

$$\mathcal{L}[\![\texttt{uniform } e_1\ e_2]\!](\Omega)(\gamma, D) = \begin{pmatrix} x \leftarrow \overline{\text{uniform}([\![e_1]\!](\gamma), [\![e_2]\!](\gamma))} \\ \text{ret}(\Omega, \text{id}, x) \end{pmatrix}$$

$$\mathcal{L}[\![\texttt{poisson } e]\!](\Omega)(\gamma, D) = \begin{pmatrix} x \leftarrow \overline{\text{poisson}([\![e]\!](\gamma))} \\ \text{ret}(\Omega, \text{id}, x) \end{pmatrix}$$

$$\mathcal{L}[\![\texttt{obs}(e_o, \texttt{flip } e_1)]\!](\Omega)(\gamma, D) = \begin{pmatrix} \text{score}(\text{flip}([\![e_1]\!](\gamma))\ ([\![e_o]\!](\gamma))); \\ \text{ret}(\Omega, \text{id}, \star) \end{pmatrix}$$

$$\mathcal{L}[\![\texttt{obs}(e_o, \texttt{uniform } e_1\ e_2)]\!](\Omega)(\gamma, D) = \begin{pmatrix} \text{score}(\text{uniform}([\![e_1]\!](\gamma), [\![e_2]\!](\gamma))\ ([\![e_o]\!](\gamma))); \\ \text{ret}(\Omega, \text{id}, \star) \end{pmatrix}$$

$$\mathcal{L}[\![\texttt{obs}(e_o, \texttt{poisson } e_1)]\!](\Omega)(\gamma, D) = \begin{pmatrix} \text{score}(\text{poisson}([\![e_1]\!](\gamma))\ ([\![e_o]\!](\gamma))); \\ \text{ret}(\Omega, \text{id}, \star) \end{pmatrix}$$

$$\mathcal{L}[\![(\!| M |\!)_S]\!](\Omega)(\gamma, D) = \begin{pmatrix} ((\Omega', \mu', E'), f, X) \leftarrow \mathcal{L}[\![M]\!](\Omega)(\gamma, D) \\ x \leftarrow (\omega' \leftarrow \mu'|_{E'};\ \text{ret}(X(\omega'))) \\ \text{ret}((\Omega', \mu', E' \cap X^{-1}(x)), f, x) \end{pmatrix}$$

Fig. 10. Low-level interpretation of effectful MultiPPL terms. Parts crucial for sound inference interoperability are highlighted, appearing in the denotation of observe M and $(\!| M |\!)_S$. Best read in color.

the weight map), and an event E denoting the result of all observe statements so far (represented by the accepting formula), and that it *produces random variables*. The fact that these probability spaces, events, and random variables are represented via weighted Boolean formulas, while crucial for the efficiency of inference, are details of the implementation that are irrelevant to ensuring safe inference interoperability. Because of this, our low-level semantics abstracts over these representation concerns, choosing instead to work directly with probability spaces and random variables. Following Li et al. [27], we model Disc programs as statefully manipulating tuples (Ω, μ, E) and producing random variables X. For example, we model running the effectful Disc program

$$X : \mathsf{bool} \vdash_c \begin{pmatrix} \mathsf{let}\ Y\ \mathsf{be}\ \mathsf{flip}\ 1/2\ \mathsf{in} \\ \mathsf{observe}\ (X \wedge Y)\ \mathsf{in} \\ \mathsf{ret}\ X \end{pmatrix} : \mathsf{bool}$$

given input probability space (Ω, μ, E) and random variable $X : \Omega \to [\![\mathsf{bool}]\!]$ as follows:

- flip $1/2$ expands the probability space from (Ω, μ, E) to $(\Omega \times [\![\mathsf{bool}]\!], \mu \otimes \mathrm{Ber}\, 1/2, E \times [\![\mathsf{bool}]\!])$, and produces the Boolean random variable $Y : \Omega \times [\![\mathsf{bool}]\!] \to [\![\mathsf{bool}]\!]$ defined by $Y(\omega, b) = b$. This is implemented by generating a new Boolean variable representing Y. Note that Y is defined in terms of the new sample space $\Omega \times [\![\mathsf{bool}]\!]$. The function $\pi_1 : \Omega \times [\![\mathsf{bool}]\!] \to \Omega$ says how to convert between the old sample space Ω and the new sample space $\Omega \times [\![\mathsf{bool}]\!]$: the random variable X, defined in terms of the old space Ω, can be converted into a random variable $X \circ \pi_1$ defined in terms of the new space $\Omega \times [\![\mathsf{bool}]\!]$ by precomposition with π_1. Similarly, the conditioning set E, a subset of the old space Ω, can be converted into a conditioning set $\pi_1^{-1}(E) = E \times [\![\mathsf{bool}]\!]$ on the new sample space $\Omega \times [\![\mathsf{bool}]\!]$. In the implementation, these conversions are no-ops: they amount to the fact that a Boolean formula over Boolean variables Γ can be weakened to a Boolean formula over variables Γ, x.
- observe $(X \wedge Y)$ shrinks the new conditioning set $E \times [\![\mathsf{bool}]\!]$ by intersecting it with the subset $G := \{(\omega, b) \mid X(\omega) = Y(b) = \top\}$ of $\Omega \times [\![\mathsf{bool}]\!]$ on which X and Y are both $\top$; this produces a new conditioning set $(E \times [\![\mathsf{bool}]\!]) \cap G$. This is implemented by conjoining the Boolean formula representing $X \wedge Y$ onto the accepting formula.

In general, we will interpret MultiPPL programs in a semantic domain that combines this stateful approach to modelling exact inference with the standard Dist_w-based approach to modelling sampling-based inference: a MultiPPL program $\Gamma; \Delta \vdash_c M : A$ denotes a function that receives:

(1) a concrete instantiation $\gamma \in [\![\Gamma]\!]$ for free Cont variables
(2) a probability space Ω and a random variable $D \in \Omega \to [\![\Delta]\!]$ for free Disc variables,

and uses the monad Dist_w to produce a weighted sample consisting of a new probability space Ω' and a random variable $X \in \Omega' \to [\![A]\!]$ of outputs. The old and new probability spaces are connected by a function $f : \Omega' \to \Omega$, which says how to convert random variables and events defined in terms of the old space into random variables and events defined on the new one. The following definitions make this idea precise.

Definition 3.2. A *finite conditional probability space* is a triple (Ω, μ, E) where (1) Ω is a finite set; (2) $\mu : \Omega \to [0, 1]$ is a discrete probability distribution, and (3) E is a subset of Ω called the *conditioning set*. Let FCPS be the collection of finite conditional probability spaces.

Definition 3.3. A *map of finite conditional probability spaces* $f : (\Omega, \mu, E) \to (\Omega', \mu', E')$ is a measure-preserving map $f : (\Omega, \mu) \to (\Omega', \mu')$ such that $E \subseteq f^{-1}(E')$. For two finite conditional probability spaces (Ω, μ, E) and (Ω', μ', E'), let $(\Omega, \mu, E) \xrightarrow{\mathrm{FCPS}} (\Omega', \mu', E')$ be the set of maps from (Ω, μ, E) to (Ω', μ', E').

Note: For readability, finite conditional probability spaces (Ω, μ, E) will be written Ω unless disambiguation is needed.

With these two definitions in hand, we can give a precise description to the semantic domains used to construct our low-level model of effectful MultiPPL terms. Given a finite conditional probability space Ω as input, an effectful Disc term $\Gamma; \Delta \vdash_c M : A$ sends a pair of substitutions for free Disc and Cont variables to a distribution over weighted samples consisting of a new finite conditional probability space Ω' and a random variable $\Omega' \to [\![A]\!]$ of outputs:

$$\mathcal{L}[\![\Gamma; \Delta \vdash_c M : A]\!](\Omega) \ : \ [\![\Gamma]\!] \times (\Omega \to [\![\Delta]\!]) \to \mathrm{Dist}_w\left(\coprod_{\Omega' \in \mathrm{FCPS}} (\Omega' \xrightarrow{\mathrm{FCPS}} \Omega) \times (\Omega' \to [\![A]\!])\right)$$

The notation $\coprod_{\Omega' \in \mathrm{FCPS}} (\Omega' \xrightarrow{\mathrm{FCPS}} \Omega) \times (\Omega' \to [\![A]\!])$ denotes an indexed coproduct: an element of this set is a tuple (Ω', f, X) consisting of a new finite conditional probability space Ω', a map of finite conditional probability spaces $f : \Omega' \to \Omega$ connecting the old and new sample spaces, and a random variable X defined on the new sample space.

Analogously, an effectful Cont term $\Gamma; \Delta \vdash_c e : \tau$ sends a pair of substitutions to a distribution over weighted samples consisting of a new finite conditional probability space Ω' and a value $v \in [\![\tau]\!]$:

$$\mathcal{L}[\![\Gamma; \Delta \vdash_c e : \tau]\!](\Omega) \ : \ [\![\Gamma]\!] \times (\Omega \to [\![\Delta]\!]) \to \mathrm{Dist}_w\left(\coprod_{\Omega' \in \mathrm{FCPS}} (\Omega' \xrightarrow{\mathrm{FCPS}} \Omega) \times [\![\tau]\!]\right)$$

The semantic equations defining $\mathcal{L}[\![\Gamma; \Delta \vdash_c M : A]\!](\Omega)$ and $\mathcal{L}[\![\Gamma; \Delta \vdash_c e : \tau]\!](\Omega)$ are given in Fig. 10. As in Fig. 9, we continue to use Haskell-style syntactic sugar for the Dist_w monad operations. The interpretation of effectful Cont programs is largely similar to the one given by $\mathcal{H}[\![-]\!]$; the primary difference is the plumbing of probability spaces Ω and maps f throughout. The interpretation of effectful Disc programs statefully manipulates the probability space as sketched earlier: flip e expands the probability space from Ω to $\Omega \otimes \Omega_{\mathrm{flip}}$, where Ω_{flip} is a freshly-generated probability space supporting a Bernoulli-distributed random variable with parameter e, and observe M shrinks the conditioning set from E to $E \cap F$, where F is the subset of the sample space on which M is $\top$. Maps of conditional probability spaces f are used to convert random variables from old to new sample spaces throughout.

The interpretation of the Cont-to-Disc boundary $(\!|\, e \,|\!)_E$ is to draw a weighted sample x from e and return the constant random variable at x. Conversely, the interpretation of the Disc-to-Cont boundary $(\!|\, M \,|\!)_S$ is to compute the random variable X denoted by M and then return a sample x drawn from the distribution of X. The parts of Fig. 10 shown in bold ensure sound inference interoperability: in the interpretation of $(\!|\, M \,|\!)_S$, the event $X^{-1}(x)$ is added to the conditioning set to ensure *sample consistency*; in the interpretation of observe M, the statement $\mathrm{score}(\mu|_E(F))$ performs *importance weighting*, to ensure the weight of the current execution remains correct relative to other possible executions.[1]

3.5 Soundness

This section presents our main theoretical result: the low-level model $\mathcal{L}[\![-]\!]$ capturing our inference strategy soundly refines the high-level model $\mathcal{H}[\![-]\!]$; that is, given a complete MultiPPL program e, weighted samples drawn from e according to our knowledge-compilation- and importance-sampling-based inference strategy follow the same distribution as samples drawn according to

[1] Here, $\mu|_E$ is the distribution μ conditioned on the event E.

$\mathcal{H}[\![-]\!]$. To make this precise, we first define what it means to run a complete MultiPPL program, and what it means for two distributions over weighted samples to be equivalent.

Definition 3.4. For a closed program $\bullet;\bullet \vdash_c e : \tau$, let $\mathrm{eval}_{\mathcal{L}}(e)$ be the computation

$$\begin{pmatrix} (\_,\_,x) \leftarrow \mathcal{L}[\![e]\!](\mathrm{emp})(\emptyset,\emptyset); \\ \mathrm{ret}\ x \end{pmatrix} : \mathrm{Dist}_{\mathrm{w}}[\![\tau]\!]$$

where $\emptyset$ is the empty substitution, emp is the unique 1-point probability space. Let $\mathrm{eval}_{\mathcal{H}}(e)$ be the computation $\mathcal{H}[\![e]\!](\emptyset,\emptyset) : \mathrm{Dist}_{\mathrm{w}}[\![\tau]\!]$.

Definition 3.5. Two computations $\mu, \nu : \mathrm{Dist}_{\mathrm{w}} A$ are *equal as importance samplers*, written $\mu \simeq \nu$, if for all bounded integrable $k : A \to \mathbb{R}$ it holds that $\mathbb{E}_{(a,x)\sim\mu}[a \cdot k(x)] = \mathbb{E}_{(b,y)\sim\nu}[b \cdot k(y)]$.

With these definitions in hand, our soundness theorem states that our inference strategy agrees with the high-level model up to equality of importance samplers.

Theorem 3.6 (soundness). *If* $\bullet;\bullet \vdash_c e : \tau$ *then* $\mathrm{eval}_{\mathcal{L}}(e) \simeq \mathrm{eval}_{\mathcal{H}}(e)$.

Theorem 3.6 is proved by induction on typing, after suitable strengthening of the theorem statement from closed to open terms. The essence of the proof boils down to two key lemmas. The first lemma allows swapping the order of sampling and scoring, and is crucial to the correctness of our importance reweighting scheme in interpreting observe:

Lemma 3.7. *If* $(\Omega,\mu,E) \in \mathrm{FCPS}$ *then* $\begin{pmatrix} \omega \leftarrow \mu; \\ \mathrm{score}(\mathbf{1}_{\omega\in E}); \\ \mathrm{ret}\ \omega \end{pmatrix} \simeq \begin{pmatrix} \mathrm{score}(\mu(E)); \\ \omega \leftarrow \mu|_E; \\ \mathrm{ret}\ \omega \end{pmatrix}$.

The second lemma says that sampling twice — from a marginal on X to get a sample x, then from the conditional distribution given $X = x$ — is the same as sampling once from the joint distribution, and is crucial to ensuring sample consistency in our implementation of the boundary $(\!|\, M \,|\!)_S$.

Lemma 3.8. *If* $(\Omega,\mu,E) \in \mathrm{FCPS}$ *and* $X : \Omega \to A$ *with* A *finite, then*

$$\begin{pmatrix} x \leftarrow (\omega \leftarrow \mu; \mathrm{ret}(X\omega)); \\ \omega' \leftarrow \mu|_{X^{-1}(x)}; \\ \mathrm{ret}(x,\omega) \end{pmatrix} = \begin{pmatrix} \omega' \leftarrow \mu; \\ \mathrm{ret}(X\omega',\omega') \end{pmatrix}.$$

The full details can be found in Appendix A.7.

4 Evaluation

In Section 3 we described the theoretical underpinnings of MultiPPL and proved it sound. In this section we provide implementation details and empirical evidence for the utility of MultiPPL by measuring its scalability on well-known inference tasks and comparing its performance against existing probabilistic programming systems. We conclude with a discussion of our evaluation and how these programs relate to the design space of MultiPPL programs.

4.1 Lightweight Extensions to MultiPPL

The semantics described in Section 3 provide a minimal model of multi-language interoperation that is simple and correct. In our implementation we extend the semantics of Disc and Cont to support more features, resulting in a practical and flexible language.

4.1.1 Extensions to Cont. Importance-sampling languages often include more features than those described in Cont. The grammar for Cont, shown in Fig. 2, supports three base distributions: Bernoulli, Uniform, and Poisson distributions. In our implementation we include many more distributions including Normal, Beta, and Dirichlet distributions, as well as their corresponding observation expressions. We also extend Cont with unbounded loops and list data structures.

4.1.2 Extensions to Disc. Our implementation of Disc directly leverages the BDD library of Dice [23] and includes support for integers as described in Holtzen et al. [23]. Integers can be introduced into a Disc program either by embedding a Cont integer or through new syntax in Disc representing a discrete distribution. Both terms are translated into one-hot encoded tuples of Boolean variables: Cont integers are translated dynamically, while discrete categorical distributions are translated by the compiler statically into the Disc grammar shown in Fig. 2.

4.2 Empirical Evaluation

MultiPPL programs encompass a vast design space, including both Cont and Disc programs as well any interleaving of these two languages. To investigate the efficacy of our implementation and characterize this landscape, we ask the following questions:

(1) *Does MultiPPL capture enough expressive power to represent interesting and practical probabilistic structure while maintaining competitive performance?* We consider four benchmarks with complex conditional independence structures to illustrate the design space of MultiPPL programs. We draw on models in the domains of network analysis [18, 25] and Bayesian networks [5, 7].
(2) *How does MultiPPL compare with contemporary PPLs in using exact and approximate inference with respect to wall-clock time and distance from the exact distribution?* To answer this question, we benchmark against state-of-the-art PPLs which handle both discrete and continuous variables: PSI [19], performing exact inference by compilation, and Pyro [8], using its importance sampling infrastructure for approximate inference.

4.2.1 Experimental Setup. For exact inference, PSI is a best-in-class language that encodes both discrete and continuous variables using its compiled symbolic approach. For approximate inference we leverage Pyro's importance sampling infrastructure. MultiPPL is written in Rust and performs both knowledge compilation and sampling during runtime evaluation when it encounters a Disc or Cont program, respectively. To unify the comparison between these disparate settings, we delineate our evaluation criteria along two metrics of sample *efficiency* and sample *quality*.

The sample *efficiency* of each inference strategy is defined as the wall-clock time to draw 1000 samples; measured in seconds and recorded in "Time(s)" column of the following figures. Comparing the performance of inference algorithms implemented in different languages is a general challenge. To account for the difference in overhead, we treat Cont as our baseline in the approximate setting.

Sample *quality* is also important and we computed the *L1-distance* (i.e., the difference of absolute values) between a ground-truth answer, derived for each task, and the estimated quantity from sampling. Tasks that only evaluate exact inference always yield an L1-distance of 0: for these tasks we only report wall-clock time, and we only draw one sample from the MultiPPL program.

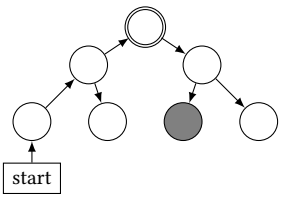
Heuristically, our aim in writing MultiPPL programs is to achieve high *quality* samples using Disc while maintaining reasonable wall-clock *efficiency* with Cont. While this guides the design of our evaluation, users must decide how this trade-off effects their models on a case-by-case basis.

All benchmarks involving approximate inference are performed using a fixed budget of 1000 samples and all statistics collected are averaged over 100 independent trials.[2]

[2]All evaluations are run on a single thread of an AMD EPYC 7543 Processor with 2.8GHz and 500 GiB of RAM. A software artifact is available on Zenodo[56] and GitHub (https://github.com/stites/multippl)

Table 1. Empirical results of our benchmarks of the arrival, discrete Bayesian network, and gossip tasks. "MultiPPL (Cont)" shows the evaluation of a baseline Cont program with no boundary crossings into Disc, evaluations under the "MultiPPL" column performs interoperation. "t/o" indicates a timeout beyond 30 minutes, and "—" indicates that the problem is not expressible in PSI because of an unbounded loop.

Model	PSI		Pyro		MultiPPL (Cont)		MultiPPL	
	L1	Time(s)	L1	Time(s)	L1	Time(s)	L1	Time(s)
arrival/tree-15	—	—	0.365	12.713	0.355	**0.247**	**0.337**	0.349
arrival/tree-31	—	—	0.216	26.366	0.218	**0.561**	**0.179**	0.754
arrival/tree-63	—	—	0.118	53.946	0.120	**1.469**	**0.093**	1.912
alarm	t/o	t/o	1.290	16.851	1.173	**0.433**	**0.364**	14.444
insurance	t/o	t/o	0.149	13.724	0.144	**1.104**	**0.099**	11.406
gossip/4	—	—	**0.119**	6.734	**0.119**	**0.720**	0.118	0.812
gossip/10	—	—	0.533	6.786	0.531	1.561	**0.524**	**1.373**
gossip/20	—	—	0.747	7.064	**0.745**	3.565	0.750	**2.888**



(a) The arrival network topology.

$n \sim \text{Poisson}(\lambda = 3)$
$q \leftarrow 0$
while $n > 0$ **do**
  $q \leftarrow q + \texttt{network}()$
  $n \leftarrow n - 1$
end while
return q

(b) Pseudocode describing arrival task.

Fig. 11. Implementation-generic details for the packet-arrival task. Shown in 11a, a packet traverses the network by entering the bottom-left most node, annotated by the start arrow. We observe a successful traversal to the gray-filled node, and we query the double-circle node for its posterior distribution. The PSI, Pyro, and MultiPPL programs all follow pseudocode shown in 11b, where `network` models the topology.

4.2.2 Estimating Packet Arrival. Our first evaluation comes from the motivating example of Fig. 1. For this arrival task we are interested in modeling packets traversing a router network and observe the presence, or absence, of packets at their destination. Our main interest is in some unobservable router that lives along the traversal path, and we query the expected number of packets which pass through this node. The router network in our evaluation has a tree-based topology that uses an equal-cost multipath (ECMP) protocol [24], as shown in Fig. 11a. The ECMP protocol dictates that a packet is forwarded with uniform probability to all neighboring routers with equal distance to the goal, as shown in Fig. 11a. In this scenario, n packets traverse the network where n is drawn from a Poisson distribution with a rate of 3, as described in Fig. 11b. The presence of this Poisson random variable makes this example quite challenging for many existing PPL inference strategies because the resulting loop has a statically unbounded number of iterations. We made the following additional design decisions in making this task:

(1) **Evidence:** We observe the gray node of the network topology depicted in Fig. 11a.
(2) **Query:** We query the expected probability that the packet traverses through a central node of the tree-topology, depicted by the twice-circled node of Fig. 11a.
(3) **Boundary decisions:** Cont models the Poisson distribution and outer loop. One boundary call is made to the network, defined in Disc.

(4) **Scaling:** We scale this model to topologies of 15, 31, and 63 nodes.
(5) **Ground truth:** Our ground truth is defined by writing a Dice program for the network, and analytically solving for the expected number of packets.

The rows labeled by "arrival" in Table 1 summarize the evaluation for this section. This table shows that MultiPPL's samples are significantly higher quality than Pyro's and the Cont program in this experiment. As the topology increases in size, we see that the MultiPPL program is able to produce increasingly higher quality samples with respect to L1 distance. This is because MultiPPL is able to exactly compute packet reachability of a single traversal in increasingly larger networks using Disc, while still able to express sampling from the Poisson distribution in Cont. The MultiPPL program performing interoperation is an order of magnitude more efficient than Pyro, however the Cont alternative is still most efficient with regard to wall-clock time and yields similar quality samples as Pyro. PSI, using its symbolic inference procedure, fails to model the unbounded loop.

4.2.3 Querying Discrete Bayesian Networks. Bayesian networks [41] provide a challenging and practical source of programs with widespread adoption across numerous domains, including medicine [1, 38], healthcare [5], and actuarial sciences [7]. Even in the purely-discrete setting, Bayesian networks remain a practical challenge when evaluating exact inference strategies due to the complex independence structures intrinsic to this domain.

In this task we study interoperation of our language by modeling two discrete Bayesian networks: ALARM [5] and Insurance [7]. These networks pose a scaling challenge for exact inference, and form the largest models described in our evaluation: ALARM contains 509 flip primitives and the Insurance network contains 1008 flip primitives. Modeling the entirety of the network in Disc and sampling 1000 times will result in a time-out for our evaluation, and we must use interoperation to increase sample quality while keeping sample efficiency competitive in our benchmark.

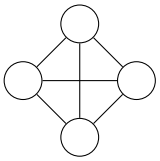
The ALARM network models a system for patient monitoring, while the Insurance network estimates the expected costs for a car insurance policyholder. We summarize these tasks as follows:

(1) **Evidence:** In both models, we observed one or more leaf nodes.
(2) **Query:** We query all root nodes for both of the Bayesian networks.
(3) **Boundary decisions:** Variables are defined in Disc or Cont to heuristically maximize the degree of exact inference permitted while keeping the wall-clock time within 60 seconds.
(4) **Scaling:** ALARM contains 509 flip primitives and Insurance contains 1008 flip primitives.
(5) **Ground truth:** The ground truth is defined by an equivalent Dice program.

The Cont model, with similar sample quality to Pyro, is more efficient than its MultiPPL counterpart in this evaluation. It is also significantly more efficient than Pyro and PSI (which timed out on this benchmark). As an importance sampler, Cont and Pyro simply sample each distribution directly, and we see the Python overhead slowing down the Pyro model.

Our MultiPPL programs demonstrate superior sample quality to the Pyro and Cont models. We achieve this by declaring boundaries that split the ALARM and Insurance networks into sub-networks that are modeled exactly with Disc and keep compiled BDD sizes small. However it should be noted that placement of boundaries tips the scales in a tradeoff between quality and efficiency. Optimal interleaving between Cont and Disc is task-sensitive, and the MultiPPL programs evaluated only demonstrate a best-effort approach to modeling.

4.2.4 Network Takeover with a Gossip Protocol. The gossip protocol is a common peer-to-peer communication method for distributed systems. In our setting each packet traverses an undirected, fully-connected network using a FIFO scheduler for transport. At each time step, indicated by a tick in the scheduler, a server will schedule two additional packets to all of its neighbors with each destination drawn i.i.d. from a uniform distribution. This task initializes with a *compromised* node



(a) Topology of the gossip network.

```
init ← 0
steps ~ Uniform(4, 8)
state ←
      (true, false, false, false)
deque ← [ ]
for n = 1, 2 do
    s ~ Discrete(1/3, 1/3, 1/3);
    deque ← s+1 :: deque
end for
```

```
while steps > 0 do
    cur ← head(deque);
    deque ← tail(deque);
    state[cur] ← state[cur] ∨ true
    for n = 1, 2 do
        s ~ Discrete(1/3, 1/3, 1/3);
        ix ← if (s < cur) { s } else { s + 1 };
        deque ← deque ++ [ix]
    end for
    steps ← steps - 1
end while
return state
```

(b) Pseudocode for a gossip network task.

Fig. 12. Implementation-generic details of the gossip network task. The 4-node topology of the undirected network is shown in 12a. Pseudocode to iterate over each time step is provided in 12b.

which sends two *infected* packets to its neighbors. When a server receives an infected packet, it becomes compromised and can only propagate infected packets for the remainder of the evaluation. Taken from Gehr et al. [18], we sample n time steps from a uniform distribution, step n times, then query the model for the expected number of compromised servers.

This evaluation poses an expressivity challenge to the Disc sublanguage, which cannot define the dynamic-length FIFO queue without interoperation with Cont. To handle this requirement, we extend Cont to support lists and define all discrete random variables in Disc. At each end of the loop we update our queue in Cont, collapsing any compiled BDDs.

(1) **Evidence:** This task defines a direct sampler and no evidence is given.
(2) **Query:** The model queries for the expected number of compromised servers after n steps.
(3) **Boundary decisions:** Discrete variables are in Disc, the loop and FIFO queue live in Cont.
(4) **Scaling:** This network scales from 4- to 10- and 20- nodes.
(5) **Ground truth:** The PSI model from Gehr et al. [18] was used to generate the ground truth for a statically defined set of time steps. An enumerative model was also defined to count the number of states. The expectation of these models over the loop is derived analytically.

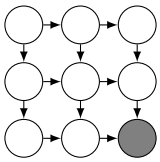
In Table 1, we see that all terminating evaluations have similar L1-distances, with Pyro and Cont programs producing slightly better quality samples. The MultiPPL model produces more efficient samples, on average, which speaks to the minimal overhead of interoperation when knowledge compilation plays a small role in inference. There is also the possibility that BDDs are cached and reused, resulting in a small speedup for some intermediate samples drawn from Disc.

As this benchmark comes with a PSI implementation from Gehr et al. [18], we provided a best-effort attempt at getting this to run including limiting the number of time steps to make the task more tractable, but we were unable reproduce their results within our 30m evaluation window.

4.2.5 Estimating Network Reliability. The *network reliability* task is interested in a single packet's traversal through a network using a probabilistic routing protocol that is embedded in a larger network. As a model only involving discrete random variables, we can observe how interoperation effects sample quality and efficiency by looking at programs defined in Cont, Disc, and in an optimal interoperation configuration. Consider, again the ECMP protocol from Section 4.2.2. In this task we modify each router with non-uniform probabilities, as a packet can traverse out of the sub-network. The sub-network itself is a directed grid, shown in Fig. 13a, with the probability of traversal being dependent on the packet's origin. Pseudocode for the model is presented in Fig. 13b.

Table 2. Exact and approximate results for models performing approximate inference

# Nodes	PSI	MultiPPL (Disc)	Pyro		MultiPPL (Cont)		MultiPPL	
	Time(s)	Time(s)	L1	Time(s)	L1	Time(s)	L1	Time(s)
9	546.748	**0.001**	0.080	3.827	0.079	**0.067**	**0.033**	0.098
36	t/o	**0.089**	1.812	14.952	0.309	**0.277**	**0.055**	1.169
81	t/o	**40.728**	7.814	33.199	0.680	**0.887**	**0.079**	81.300



(a) Topology of a 9-node network

$n_{00} \sim \text{Bern } \frac{1}{3}$
$n_{01} \sim \text{if } \quad n_{00} \text{ then Bern } \frac{1}{4} \text{ else Bern } \frac{1}{5}$
$n_{10} \sim \text{if } \neg n_{00} \text{ then Bern } \frac{1}{4} \text{ else Bern } \frac{1}{5}$
$p \leftarrow \text{if } \quad n_{10} \vee \quad n_{01} \text{ then } \frac{1}{6} \text{ else}$
$\qquad\qquad n_{10} \vee \neg n_{01} \text{ then } \frac{1}{7} \text{ else}$
$\qquad\qquad \neg n_{10} \vee \quad n_{01} \text{ then } \frac{1}{8} \text{ else } \frac{1}{9}$
$n_{11} \sim \text{Bern } p$
observe $n_{11} = \top$
return $(n_{00}, n_{01}, n_{10}, n_{11})$

(b) Pseudocode of a 4-node model

Fig. 13. An overview of the reliability task, with the topology of the 9-node network in 13a: a packet is observed in the node shaded gray and all nodes are queried for their posterior distribution. In 13b we show the pseudocode for a 4-node reliability task, similar structure is used for networks with 9-, 36-, and 81-nodes.

This benchmark observes a packet arriving at the final node in the sub-network, and queries the probability that this packet passes through each router in the model. As there are no continuous random variables involved, we can model this task using either exact or approximate inference.

(1) **Evidence:** The final node in the network topology observes a successful packet traversal.
(2) **Query:** The model queries for the marginal probability on all nodes in the network.
(3) **Boundary decisions:** MultiPPL programs (in column "MultiPPL" of table Table 2), model the minor upper- and lower- triangles of the network topology in Disc and perform interoperation along the minor diagonal to break the exact inference task into two parts. This maximizes the size of compiled BDDs while providing orders of magnitude improvement in sample efficiency.
(4) **Scaling:** This network scales in the size of the grid, scaling from 9- to 36- to 81- nodes.
(5) **Ground truth:** An equivalent Dice model was used as the ground truth for this model.

The first two columns of Table 2 show the results of exact compilation; comparing PSI to Disc programs (column "MultiPPL (Disc)"). Because of the nature of this evaluation, Disc can represent the exact posterior of the model and produce perfect samples with competitive efficiency for small programs. As the program grows in size, producing samples take considerably longer: scaling with the size of the underlying logical formula.

The partially-collapsed and fully-sampled MultiPPL programs are compared to Pyro in the remaining columns of Table 2. MultiPPL programs (column "MultiPPL") are defined in Disc and model the minor diagonal of the network's grid in Cont. Programs fully defined in Cont (column "MultiPPL (Cont)") sample each node individually in the same manner as Pyro.

In this evaluation Cont is more efficient and MultiPPL programs effectively leverage Disc's knowledge compilation to produce higher-quality samples. For smaller models, the defined MultiPPL programs have efficiency competitive to Cont. As the model scales, the overhead of knowledge compilation increases. This can be seen by noting the single-sample efficiency of Disc programs

from our exact evaluation. As the MultiPPL program scales to 81 nodes the sample efficiency decreases, suggesting an alternative collapsing scheme may be preferable for larger programs.

This network reliability evaluation, alongside prior evaluations, demonstrates that MultiPPL consistently produces higher quality samples compared to alternatives in the approximate setting. Through these evaluations, we find that MultiPPL does capture enough expressive power to represent interesting and practical probabilistic structure while remaining competitive with other languages. That said, the performance of MultiPPL's inference poses a nuanced landscape and we leave a full characterization of this design space to future work.

5 Related Work

Multi-language interoperation between probabilistic programming languages builds on a wide body of work spanning the programming language and the machine learning communities. We situate our research in four categories: heterogeneous inference, programmable inference, multi-language semantics, and the monadic semantics of probabilistic programming languages.

Heterogeneous Inference in Probabilistic Programming Languages. There are existing probabilistic programming languages and systems that enable users to blend different kinds of inference algorithms when performing inference on a single probabilistic program. Particularly relevant are approaches that leverage *Rao-Blackwellization* in order to combine exact and approximate inference strategies into a single system. Within this vein, Atkinson et al. [2] introduced *semi-symbolic inference*, where the idea is to perform exact marginalization over distributions whose posteriors can be determined to have some closed-form solution. Other works that use variations of Rao-Blackwellization [21, 33, 37] all seek to explicitly marginalize out portions of the distribution by using closed-form exact posteriors when feasible. The main difference between these approaches to Rao-Blackwellization and our proposed approach is that these systems do not expose *separate languages* that correspond to different phases of the inference algorithm: they provide a single unified syntax in which the user programs. As a consequence, they all rely on (semi-)automated means of automatically discovering which portions can be feasibly Rao-Blackwellized; this process can be difficult to control and lead to unpredictable performance. Our multi-language approach has the following benefits: (1) predictable and interpretable performance due to the explicit choice of inference algorithm that is exposed to the user; and (2) amenability to modular formalization, since we can verify the correctness of each inference strategy and verify the correctness of their composition on the boundary. We hope to incorporate the interesting ideas of these related works into MultiPPL, in particular closed-form approaches to exact marginalization of continuous distributions.

There is a broad literature on heterogeneous inference that we hope to eventually draw on to build a richer vocabulary of sublanguages to add to MultiPPL. Friedman and Van den Broeck [17] described an approach to collapsed approximate inference that dynamically blends exact inference via knowledge compilation and approximate inference via sampling; we are curious if this can be integrated with our system. We also look towards incorporating more stateful inference algorithms such as Markov-Chain Monte Carlo into MultiPPL, and aim to investigate this in future work.

Programmable Inference. Programmable inference (or inference (meta-)programming) provide probabilistic programmers with a meta-language for defining new inference algorithms within a single PPL by offering language primitives that give direct access to the inference runtime [29]. Cusumano-Towner et al. [12] provides a black-box interface to underlying inference algorithms alongside combinators to operate on these interfaces, Stites et al. [58] designs a domain specific language (DSL) for inference which produces correct-by-construction importance weights.

We see programmable inference as a viable means of designing new inference algorithms which we can incorporate into a multi-language. Furthermore, a multi-language setting can offer inference

programmers the ability to abstract away the nuances of the inference process, lowering the barrier to entry for this type of development. One common thread through much of the work on inference programming is core primitives which encapsulate the building blocks for inference algorithms including resample-move sequential Monte Carlo, variational inference, many other Markov chain Monte Carlo methods. These primitives could be designed formally as DSLs, which would be a great addition to a multi-language and something we look forward to developing in future work.

Nested Inference. Nested inference enriches a probabilistic programming language with a first-class `infer` or `normalize` construct that enables the programmer to query for the probability of an event inside their probabilistic programs [3, 42, 54, 59, 64]. Nested inference is a useful programming construct that enables a variety of new applications, such as in cognitive science where one agent may wish to reason about the intent of another [59]. Nested inference is similar in spirit to our multi-language approach in that it gives the programmer control over when inference is performed on their program and what inference algorithm is used. A key difference between nested inference and our multi-language approach is that the former provides access to the inference result whereas MultiPPL's boundary forms do not. This difference is essential. In our view, there is the following analogy to non-probabilistic programming: performing nested inference is like invoking a compiler and inspecting the resulting binary, whereas performing multi-language inference is like interoperating through an FFI. In the non-probabilistic setting, these two situations require distinct semantic models — compare, for example, formal models of introspection and dynamic code generation [6, 15, 28, 30, 52] with formal models of FFI-based interoperability [22, 26, 31, 39, 45] — and we believe the same is likely true of our probabilistic setting.

In the future, it would be interesting to consider integrating nested inference within a multi-language setting and exploring the consequences of this new feature on language interoperation. It would also be quite interesting to investigate whether our multi-language inference strategy could be compiled to, or expressed in terms of, rich nested inference constructs. A preliminary analysis reveals a number of basic differences between MultiPPL's inference strategy and standard models of nested inference, so such a compilation scheme would likely require significant modifications to nested inference — for a detailed technical discussion, see Appendix B.

Multi-Language Semantics. Today, it is often the case that probabilistic programming languages are embedded in a host, non-probabilistic language [4]. However, these PPLs assume their host semantics will not interfere with the semantics of the PPL's inference process. This work is the first of its kind to build on top of multi-language semantics to reason about inference algorithms.

Multi-language semantics, while new to the domain of probabilistic programming, has had a large impact on the broader programming language community. They play a fundamental role in reasoning about interoperation [39], gradual typing [35, 60], and compositional compiler verification [45]. There are two styles of calculi which represent the current approaches to multi-language interoperation. These are the *multi-languages* approach from Matthews and Findler [31] and the a more fine-grained approach by Siek and Taha [49] using a *gradually typed lambda calculus*.

Ye et al. [63] takes a traditional programming language approach to the gradual typing of PPLs and defines a gradually typed probabilistic lambda calculus which allows a user to migrate a PPL from an untyped- to typed- language — a nontrivial task involving a probabilistic coupling argument for soundness. In contrast, our work centers on how multi-languages can help the interoperation of inference algorithms across semantic domains.

Baydin et al. [4] establishes, informally, a common interface for PPLs to interact with scientific simulators across language boundaries. In this work, the semantics of the simulator is a black-box distribution defined in some language, which may or may not be probabilistic, and a separate PPL may interact with the simulator during the inference process. While Baydin et al. [4] works

across language boundaries, they do not reason about interoperation — they only involve one inference algorithm — and they do not provide any soundness guarantees. That said, Baydin et al. [4] demonstrates a simple boundary allowing for rapid integration of many practical probabilistic programming languages, something we also strive for.

Monadic Semantics of PPLs. Numerous monads have been developed for use as semantic domains that capture the various notions of computation used in probabilistic inference. The fundamental building block for each of these models is the probability monad Dist, along with its generalizations to monads of subdistributions and measures [20]. Using this probability monad to give semantics to probabilistic programs goes back to at least Ramsey and Pfeffer [43], who further build on this basic setup by introducing *measure terms* to efficiently answer expectation-based queries. Staton et al. [55] make use of the writer monad transformer applied to the monoid of weights to obtain a monad suitable for modelling probabilistic programs with score-based conditioning; we have made essential use of this monad to define the two semantic models of MultiPPL presented in Section 3. Ścibior [48] use monad transformer stacks, implemented in Haskell, to obtain a variety of sampling-based inference algorithms in a compositional manner, with each layer of the stack encompassing a different component of an inference algorithm. Our semantics of MultiPPL builds on this line of work in giving monadic semantics to probabilistic computations by providing a model of exact inference via knowledge compilation in terms of stateful manipulation of finite conditional probability spaces and random variables. In future work, we intend to investigate whether this state-passing semantics can be packaged into a monad of its own, capturing the notion of computation carried out when performing knowledge compilation, by making use of recent constructions in categorical probability [50, 51].

6 Conclusion

Performing inference on models with a mix of continuous and discrete random variables is an important modeling challenge for practical systems and MultiPPL offers a multi-language approach to tackle this problem. In this work, we provide a sound denotational semantics that generalizes for all exact inference algorithms and sampling-based approximate inference that satisfy our semantic domains. We identify two requirements to establish the correctness of the interoperation described: that the exact PPL must maintain *sample consistency* and that the approximate sampling-based PPL must perform *importance weighting*. We demonstrate that our implementation of MultiPPL benefits from the expressiveness of Cont and makes practical problems representable and additionally provides tractable inference from Disc for complex discrete-structured probabilistic programs.

Ultimately, we hope that our multi-language perspective can lead to a clean formal unification of many probabilistic program semantics and inference strategies. For future work, we hope to extend our semantics to incorporate local-search inference strategies such as sequential and Markov-Chain Monte Carlo. With enough coverage across semantics, we also gain the opportunity to look at probabilistic interoperation by inspecting a shared core calculus for inference, and would draw on work from Patterson [40]. Finally, by providing a syntactic approach to inference interoperation, we also open up opportunities to use static analysis to see when and how we might automatically insert boundaries to further specialize a model's inference algorithm.

Acknowledgements

We thank the reviewers for their thoughtful and clarifying feedback. This project was supported by the National Science Foundation under grant #2220408.

References

[1] Steen Andreassen, Roman Hovorka, Jonathan Benn, Kristian G. Olesen, and Ewart R. Carson. 1991. A Model-Based Approach to Insulin Adjustment. In *AIME 91*, Vol. 44. Springer Berlin Heidelberg, Berlin, Heidelberg, 239–248. doi:10.1007/978-3-642-48650-0_19

[2] Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022. Semi-Symbolic Inference for Efficient Streaming Probabilistic Programming. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1668–1696. doi:10.1145/3563347

[3] Chris L Baker, Rebecca Saxe, and Joshua B Tenenbaum. 2009. Action understanding as inverse planning. *Cognition* 113, 3 (2009), 329–349. doi:10.1016/j.cognition.2009.07.005

[4] Atılım Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Victor Lee, Kyle Cranmer, Prabhat, and Frank Wood. 2019. Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver, Colorado, Article 29. doi:10.1145/3295500.3356180

[5] Ingo A Beinlich, Henri Jacques Suermondt, R Martin Chavez, and Gregory F Cooper. 1989. The ALARM Monitoring System: A Case Study with Two Probabilistic Inference Techniques for Belief Networks. In *AIME 89*. Springer, 247–256. doi:10.1007/978-3-642-93437-7_28

[6] Nick Benton and Chung-Kil Hur. 2010. Step-Indexing: The Good, the Bad and the Ugly. In *Modelling, Controlling and Reasoning about State, Proceedings of Dagstuhl Seminar 10351* (modelling, controlling and reasoning about state, proceedings of dagstuhl seminar 10351 ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. https://www.microsoft.com/en-us/research/publication/step-indexing-the-good-the-bad-and-the-ugly/

[7] John Binder, Daphne Koller, Stuart Russell, and Keiji Kanazawa. 1997. Adaptive Probabilistic Networks with Hidden Variables. 29, 2-3 (1997), 213–244. doi:10.1023/A:1007421730016

[8] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 1 (Jan. 2019), 973–978. https://dl.acm.org/doi/abs/10.5555/3322706.3322734

[9] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of statistical software* 76 (2017). doi:10.18637/jss.v076.i01

[10] Mark Chavira and Adnan Darwiche. 2008. On Probabilistic Inference by Weighted Model Counting. *Artificial Intelligence* 172, 6 (April 2008), 772–799. doi:10.1016/j.artint.2007.11.002

[11] Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *Programming Languages and Systems (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 368–392. doi:10.1007/978-3-662-54434-1_14

[12] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-Purpose Probabilistic Programming System with Programmable Inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, Phoenix, AZ, USA, 221–236. doi:10.1145/3314221.3314642

[13] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. 2018. Foundations of Dependent Interoperability. *Journal of Functional Programming* 28 (Jan. 2018), e9. doi:10.1017/S0956796818000011

[14] A. Darwiche and P. Marquis. 2002. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research* 17 (Sept. 2002), 229–264. doi:10/ghjsq6

[15] Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (May 2001), 555–604. doi:10.1145/382780.382785

[16] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and Learning in Probabilistic Logic Programs Using Weighted Boolean Formulas. *Theory and Practice of Logic Programming* 15, 3 (2015), 358–401. doi:10.1017/S1471068414000076

[17] Tal Friedman and Guy Van den Broeck. 2018. Approximate Knowledge Compilation by Online Collapsed Importance Sampling. In *Advances in Neural Information Processing Systems*, Vol. 31. Curran Associates, Inc., 15. https://dl.acm.org/doi/10.5555/3327757.3327898

[18] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: Probabilistic Inference for Networks. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 586–602. doi:10.1145/3296979.3192400

[19] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. *Proc. of ESOP/ETAPS* 9779 (2016), 62–83. doi:10/gmq8ks

[20] Michele Giry. 2006. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis: Proceedings of an International Conference Held at Carleton University, Ottawa, August 11–15, 1981*. Springer, 68–85. doi:10.1007/BFb0092872

[21] Maria I Gorinova, Andrew D Gordon, Charles Sutton, and Matthijs Vákár. 2021. Conditional independence by typing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 1 (2021), 1–54. doi:10.1145/3490421

[22] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Artifact for "Melocoton: A Program Logic for Verified Interoperability Between OCaml and C"* 7, OOPSLA2 (Oct. 2023), 247:716–247:744. doi:10.1145/3622823

[23] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31. doi:10.1145/3428208

[24] Christian Hopp. 2000. Analysis of an Equal-Cost Multi-Path Algorithm. https://doi.org/10.17487/RFC2992

[25] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (Oct. 2011), 1765–1775. doi:10.1109/JSAC.2011.111002

[26] Joomy Korkut, Kathrin Stark, and Andrew W. Appel. 2025. A Verified Foreign Function Interface between Coq and C. *Proc. ACM Program. Lang.* 9, POPL (Jan. 2025), 24:687–24:717. doi:10.1145/3704860

[27] John M. Li, Amal Ahmed, and Steven Holtzen. 2023. Lilac: A Modal Separation Logic for Conditional Probability. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 112:148–112:171. doi:10.1145/3591226

[28] Jacques Malenfant, Christophe Dony, and Pierre Cointe. 1996. A Semantics of Introspection in a Reflective Prototype-Based Language. *LISP and Symbolic Computation* 9, 2 (May 1996), 153–179. doi:10.1007/BF01806111

[29] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: A Higher-Order Probabilistic Programming Platform with Programmable Inference. *arXiv* (March 2014), 78–78. arXiv:1404.0099

[30] Jacob Matthews and Robert bruce Findler. 2008. An Operational Semantics for Scheme1. *J. Funct. Program.* 18, 1 (Jan. 2008), 47–86. doi:10.1017/S0956796807006478

[31] Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. *ACM SIGPLAN Notices* 42, 1 (2007), 3–10. doi:10.1145/1190215.1190220

[32] Torben Meisling. 1958. Discrete-Time Queuing Theory. *Operations Research* 6, 1 (1958), 96–105. doi:10.1287/opre.6.1.96

[33] Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain (Proceedings of Machine Learning Research, Vol. 84)*. PMLR, 1037–1046. http://proceedings.mlr.press/v84/murray18a.html

[34] Radford M. Neal. 2011. MCMC Using Hamiltonian Dynamics. *Handbook of Markov Chain Monte Carlo* 54 (2011), 113–162.

[35] Max Stewart New. 2020. *A Semantic Foundation for Sound Gradual Typing*. Ph. D. Dissertation. Northeastern University, USA. https://dl.acm.org/doi/book/10.5555/AAI28263083

[36] Max S New, William J Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 103–116. doi:10.1145/2951913.2951941

[37] Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Neeraj Pradhan, Justin Chiu, Alexander Rush, and Noah Goodman. 2019-06-09/2019-06-15. Tensor Variable Elimination for Plated Factor Graphs. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*. 4871–4880. https://proceedings.mlr.press/v97/obermeyer19a.html

[38] A. Onisko, Marek J. Druzdzel, H. Wasyluk, and A. Onisko. 2005. A Probabilistic Causal Model for Diagnosis of Liver Disorders.

[39] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic Soundness for Language Interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, San Diego CA USA, 609–624. doi:10.1145/3519939.3523703

[40] Daniel Baker Patterson. 2022. *Interoperability through Realizability: Expressing High-Level Abstractions Using Low-Level Code*. Ph. D. Dissertation. Northeastern University. doi:10.17760/D20467221

[41] Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[42] Tom Rainforth, Robert Cornish, Hongseok Yang, and Andrew Warrington. 2018. On Nesting Monte Carlo Estimators. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*. PMLR, 4264–4273. http://proceedings.mlr.press/v80/rainforth18a.html

[43] Norman Ramsey and Avi Pfeffer. 2002. Stochastic Lambda Calculus and Monads of Probability Distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 154–165. doi:10.1145/503272.503288

[44] Christian P Robert, George Casella, and George Casella. 1999. *Monte Carlo statistical methods*. Vol. 2. Springer.

[45] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-language Semantics and Verification. *Proceedings of the*

ACM on Programming Languages 7, POPL (Jan. 2023), 27:775–27:805. doi:10.1145/3571220

[46] Tian Sang, Paul Bearne, and Henry Kautz. 2005. Performing Bayesian Inference by Weighted Model Counting. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1 (AAAI'05)*. AAAI Press, Pittsburgh, Pennsylvania, 475–481. https://dl.acm.org/doi/10.5555/1619332.1619409

[47] Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. 2018. Fabous Interoperability for ML and a Linear Language. In *Foundations of Software Science and Computation Structures*. Springer International Publishing, Cham, 146–162. doi:10.1007/978-3-319-89366-2_8

[48] Adam Michał Ścibior. 2018. *Formally Justified and Modular Bayesian Inference for Probabilistic Programs*. Ph. D. Dissertation. Apollo - University of Cambridge Repository.

[49] Jeremy G Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the Scheme and Functional Programming Workshop*.

[50] Alex Simpson. 2017. Probability Sheaves and the Giry Monad. In *7th Conference on Algebra and Coalgebra in Computer Science (CALCO 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Schloss-Dagstuhl-Leibniz Zentrum für Informatik. doi:10.4230/LIPIcs.CALCO.2017.1

[51] Alex Simpson. 2018. Category-Theoretic Structure for Independence and Conditional Independence. *Electronic Notes in Theoretical Computer Science* 336 (2018), 281–297. doi:10.1016/j.entcs.2018.03.028

[52] Brian Cantwell Smith. 1984. Reflection and Semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. Association for Computing Machinery, New York, NY, USA, 23–35. doi:10.1145/800017.800513

[53] Steffen Smolka, Praveen Kumar, David M Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable Verification of Probabilistic Networks. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 190–203. doi:10.1145/3314221.3314639

[54] Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Programming Languages and Systems (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 855–879. doi:10.1007/978-3-662-54434-1_32

[55] Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. 2016. Semantics for Probabilistic Programming: Higher-order Functions, Continuous Distributions, and Soft Constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. ACM, New York, NY, USA, 525–534. doi:10.1145/2933575.2935313

[56] Sam Stites, John M. Li, and Steven Holtzen. 2025. Artifact: Multi-Language Probabilistic Programming. Zenodo. doi:10.5281/zenodo.14593465

[57] Sam Stites, John M. Li, and Steven Holtzen. 2025. Multi-Language Probabilistic Programming. doi:10.48550/arXiv.2502.19538 arXiv:2502.19538 [cs]

[58] Sam Stites, Heiko Zimmermann, Hao Wu, Eli Sennesh, and Jan-Willem van de Meent. 2021. Learning Proposals for Probabilistic Programs with Inference Combinators. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*. PMLR, 1056–1066. https://proceedings.mlr.press/v161/stites21a.html

[59] A. Stuhlmüller and N. D. Goodman. 2014. Reasoning about Reasoning by Nested Conditioning: Modeling Theory of Mind with Probabilistic Programs. *Cognitive Systems Research* 28 (June 2014), 80–99. doi:10.1016/j.cogsys.2013.07.003

[60] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 395–406. doi:10.1145/1328438.1328486

[61] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as Libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 132–141. doi:10.1145/1993498.1993514

[62] Jesse A. Tov and Riccardo Pucella. 2010. Stateful Contracts for Affine Types. In *Programming Languages and Systems*. Springer, Berlin, Heidelberg, 550–569. doi:10.1007/978-3-642-11957-6_29

[63] Wenjia Ye, Matías Toro, and Federico Olmedo. 2023. A Gradual Probabilistic Lambda Calculus. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 84:256–84:285. doi:10.1145/3586036

[64] Yizhou Zhang and Nada Amin. 2022. Reasoning about "Reasoning about Reasoning": Semantics and Contextual Equivalence for Probabilistic Programs with Nested Queries and Recursion. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 16:1–16:28. doi:10/gqqdp8

Received 2024-10-15; accepted 2025-02-18; revised 2024-10-15; accepted 2025-02-18