Compiling Probabilistic Programs for Variable Elimination with Information Flow

JIANLIN LI, University of Waterloo, Canada
ERIC WANG, University of Waterloo, Canada
YIZHOU ZHANG, University of Waterloo, Canada

A key promise of probabilistic programming is the ability to specify rich models using an expressive programming language. However, the expressive power that makes probabilistic programming languages enticing also poses challenges to inference, so much so that specialized approaches to inference ban language features such as recursion. We present an approach to variable elimination and marginal inference for probabilistic programs featuring bounded recursion, discrete distributions, and sometimes continuous distributions. A compiler eliminates probabilistic side effects, using a novel information-flow type system to factorize probabilistic computations and hoist independent subcomputations out of sums or integrals. For a broad class of recursive programs with dynamically recurring substructure, the compiler effectively decomposes a global marginal-inference problem, which may otherwise be intractable, into tractable subproblems. We prove the compilation correct by showing that it preserves denotational semantics. Experiments show that the compiled programs subsume widely used PTIME algorithms for recursive models and that the compilation time scales with the size of the inference problems. As a separate contribution, we develop a denotational, logical-relations model of information-flow types in the novel measure-theoretic setting of probabilistic programming; we use it to prove noninterference and consequently the correctness of variable elimination.

CCS Concepts: • **Theory of computation** → *Probabilistic computation*; *Program semantics*; *Program reasoning*; *Type theory*; • **Mathematics of computing** → *Bayesian computation*; *Statistical software*; • **Computing methodologies** → *Machine learning*; • **Software and its engineering** → *Compilers*; *Functional languages*; *Language features*; *Formal language definitions*.

Additional Key Words and Phrases: Probabilistic programming, probabilistic inference, information flow.

ACM Reference Format:
Jianlin Li, Eric Wang, and Yizhou Zhang. 2024. Compiling Probabilistic Programs for Variable Elimination with Information Flow. *Proc. ACM Program. Lang.* 8, PLDI, Article 218 (June 2024), 26 pages. https://doi.org/10.1145/3656448

1 INTRODUCTION

A probabilistic model describes a joint distribution $p(x, z)$ over latent variables z and observations x. Bayesian inference is concerned with computing $p(z|x) = p(x, z)/p(x)$, the posterior distribution of z conditioned on x. Typically, the hard work is in computing the *marginal likelihood* $p(x)$, also known as the *model evidence*. Computing the marginal may be intractable, as it generally requires integration over all possible values of the latent variables: $p(x) = \int p(x, z)\mathrm{d}z$.

Probabilistic programming languages (PPLs) are powerful means to specify probabilistic models and solve inference problems. A PPL allows for harnessing the expressivity of a high-level programming language to specify rich Bayesian models, as opposed to using more limiting formalisms such

Authors' addresses: Jianlin Li, jianlin.li@uwaterloo.ca, University of Waterloo, Canada; Eric Wang, e224wang@uwaterloo.ca, University of Waterloo, Canada; Yizhou Zhang, yizhou@uwaterloo.ca, University of Waterloo, Canada.

ACM 2475-1421/2024/6-ART218
https://doi.org/10.1145/3656448

as Bayesian networks. However, the expressive power that makes PPLs enticing makes inference even harder. Recent advances in PPL inference often specialize to a particular class of models and impose restrictions on expressible models, prohibiting useful features such as recursion.

Variable elimination (VE) is an effective approach to inference for probabilistic models with discrete random variables (r.v.s) [68]. It works by marginalizing out (i.e., eliminating) discrete r.v.s from a joint distribution, thus producing the marginal likelihood or reducing the inference problem to ones involving only continuous r.v.s.

VE has been generalized to PPLs, but the support for VE does not meet the desired level generality and scalability. For example, Factorie [42] is an interpreted PPL that supports VE for factor graphs, but its VE algorithms make specific, rigid assumptions on the factor-graph structure. SlicStan [29] and PERPL [15] are more recent, compiled approaches to VE and are designed to support broader classes of models. Unfortunately, SlicStan lacks support for recursion—recursion is a natural means to specify models in domains such as language modeling and computational biology. Another issue is that the time SlicStan takes to compile a program does not scale well with the size of the inference problem. PERPL, while supporting (unbounded) recursion, is designed to work in the absence of continuous r.v.s. For exact inference on certain recursive models involving only discrete r.v.s, PERPL does not empirically scale as well as the best algorithms known for the same models.

This paper presents a novel approach to VE for an expressive PPL. While acknowledging that it is an elusive, likely impossible goal for any single inference method to excel at all expressible programs, we aim to achieve good efficiency and scalability across a wide range of programs featuring bounded recursion, discrete distributions, and occasionally continuous distributions, all the while with provable correctness guarantees. We embody this approach in a PPL called Mappl.

VE as compilation. In Mappl, a probabilistic computation is compiled into a pure computation of the marginal likelihood. A discrete r.v. is eliminated by summing (over the variable's finite support) the product of all factors dependent on that variable. Control flow, namely branching and function calls, are compiled in continuation-passing style: the compiled branch or function takes as input a continuation representing the product of all factors dependent on the return value.

Decomposition, memoization, and amortization. We observe that many recursive probabilistic programs of interest enjoy the property that the exponentially many possible executions share substructure. For these programs, the VE compilation effectively decomposes, in a recursive manner, a global marginal inference problem into subproblems amenable to dynamic programming [9]. The same subproblem instance may be queried multiple times during inference, so the solution to the inference subproblem can be *memoized* and thus the cost of solving the subproblem *amortized*.

The subproblems are likely easier to solve than the global problem, because they have reduced dimensionality and are free of language constructs such as recursion that are difficult for inference. Some of these subproblems may be solved easily if they happen to contain no continuous r.v.s, some may be solved by existing approximate inference methods that specialize in straight-line programs with continuous r.v.s, and some may sometimes even be solved analytically by capitalizing on advances in symbolic integration. The upshot is that the VE compilation may render an otherwise intractable inference problem solvable in polynomial time.

Factorization by information-flow typing. It is well understood that the effectiveness of VE critically depends on exploiting independence to factorize joint distributions. With recursion, there is more reason for a VE compiler to exploit independence, as decomposition and memoization just would not be as effective if subproblem definitions were too coarse-grained.

To reason about independence, we design an information-flow type system for Mappl. To eliminate a variable x from a computation, the Mappl compiler consults information-flow typing to factorize the computation into two parts, the probabilistic side effects of which are respectively dependent and independent of x. The idea of using information-flow typing [21] to reason about independence is similar to that in SlicStan [29], but Mappl's type-system design and formal development differ substantially from SlicStan's. SlicStan is an imperative while-language where variables must be global and programs must have deterministic support, whereas Mappl is an expressive functional language allowing recursion and stochastic support. SlicStan is defined with an operational semantics, whereas we adopt a *compositional*, denotational treatment suited for reasoning about independence—and thus for factorizing computations—for open terms under binders.

Generality and scalability. Mappl generalizes VE compilation and information-flow typing to recursive probabilistic programs—for example, those expressing hidden Markov models (HMMs) and probabilistic context-free grammars (PCFGs).[1] Experiments show that Mappl's VE compiler can generate code that recovers widely used polynomial-time inference algorithms: the forward algorithm for HMMs [55] and the inside algorithm for PCFGs [5].

We consider it important to achieve good scalability of not only inference but also compilation. Notably, compared with SlicStan, the increased generality of Mappl to support recursion has implications for the scalability of compilation. In SlicStan, HMMs have to be expressed by unrolling recursion into a fixed number of iterations. For such models, SlicStan's compilation time does not scale well with the problem size (e.g., the length of the observed sequence). In Mappl, by contrast, compilation time stays constant as the problem size increases for such models, because Mappl can express them as probabilistic recursive functions and compile them to pure recursive functions, without unrolling. In addition, SlicStan uses a semilattice—as opposed to a lattice—of information-flow labels for factorization, which is considered to impede efficient label inference. By contrast, Mappl uses a simpler, better-behaved two-level lattice.

Correctness guarantees. We want to show that Mappl's VE compilation is correct by proving that the compiled program computes the marginal likelihood as defined by the denotational semantics. Since compilation uses information-flow typing to factorize computations, we need to show that our information-flow type system is sound with respect to the denotational semantics. To that end, we contribute a logical-relations model of information-flow types for proving noninterference in the novel, measure-theoretic setting of probabilistic programming.

2 KEY FEATURES, MAIN IDEAS, AND EXAMPLES

We use a simple hidden Markov model as a starting point to illustrate the key features and the main ideas of our approach. Figure 1a models a sequence of observations as being generated by a sequence of hidden states. The recursive function hmm takes as input the initial hidden state z_0 and a data sequence, which is assumed to be gathered by prepending newer observations to the front of the sequence. The return value of hmm is the next hidden state. The probability of transitioning from a state to the next state is given by a pure function step : $\mathbb{B} \to \mathsf{dist}(\mathbb{B})$. The probability of observing a data point in a state is given by an emission function emit : $\mathbb{B} \to \mathsf{dist}(\mathbb{R})$. The HMM is conditioned on observing the data sequence.

The inference problem is, given any given data sequence, to compute the marginal likelihood of observing it. The Mappl compiler translates the recursive, probabilistic hmm in Figure 1a into the recursive, pure hmm in Figure 1b. When the compiled hmm is called with the top-level continuation $\lambda\_.\ 0$ for the parameter k, it computes the desired marginal likelihood. This procedure for exact inference runs in time linear in the length of data, recovering the forward algorithm for HMMs.

[1]As a caveat, this paper does not address VE for almost surely terminating programs, which use unbounded recursion; we consider the restriction to deterministically bounded recursion to be a reasonable trade-off (Section 8).

source program: recursive and probabilistic | compiled program: recursive and pure

```
def hmm (z0, data) =                                                     (a)
 case data of
 | nil ⇒ ret(z0)
 | cons x xs ⇒
   z = hmm(z0, xs) // recursive call returns current state
   observe(emit(z); x)              // Bayesian conditioning
   sample(step(z))       // sample next state and return it
 end
```

➡

```
def hmm (k, z0, data) =                                                  (b)
 case data of
 | nil ⇒ k(z0)
 | cons x xs ⇒ hmm(
     λz. logPr(emit(z); x) +
         logsumexp𝔹(λy. logPr(step(z); y) + k(y)),
     z0, xs)
 end             // recovers linear-time forward algorithm
```

```
def pcfg (words) =                                                       (c)
 z = sample(Bern(0.5))
 case z, words of
 | true, cons "a" nil ⇒ factor(0)     // production S ⟶ a
 | false, cons _ _ ⇒                 // production S ⟶ SS
   sp = choose(len(words))
                       // split words at a randomly chosen point
   let left, right = split(words, sp) in
   pcfg(left)                               // recursive call
   pcfg(right)                              // recursive call
 | _, _ ⇒ factor(−∞)
 end
```

➡

```
def pcfg (k, words) =                                                    (d)
 logsumexp𝔹(
  λz. logPr(Bern(0.5); z) +
      case z, words of
      | true, cons "a" nil ⇒ 0 + k(unit)
      | false, cons _ _ ⇒
        logsumexpℕ(len(words),
         λsp. let left, right = split(words, sp) in
              pcfg(λ_. 0, left) +
              pcfg(k, right))
      | _, _ ⇒ −∞ + k(unit)
      end)         // recovers cubic-time inside algorithm
```

```
// a hybrid discrete–continuous HMM                                      (e)
def hmm′ (z0, data) =
 case data of
 | nil ⇒ ret(z0)
 | cons x xs ⇒
   z = hmm′(z0, xs)
   w = sample(Normal(0,1)) // sample a continuous r.v.
   observe(emit′(w,z); x)
   sample(step(z))
 end
```

➡

```
def hmm′ (k, z0, data) =                                                 (f)
 case data of
 | nil ⇒ k(z0)
 | cons x xs ⇒ hmm′(
     λz. logML(
           w = sample(Normal(0,1))
           factor(logPr(emit′(w,z); x))
         ) + logsumexp𝔹(λy. logPr(step(z); y) + k(y)),
     z0, xs)
 end
```

Figure 1. Examples of VE-compiling probabilistic programs in Mappl. The return value of a multi-line block of terms is that of the last term. The construct ret(e) is the monadic return that lifts a pure expression to a probabilistic term. The primitive logPr$(d; \cdot)$ is the log-probability density or mass function of the distribution d. logsumexp$_{\mathbb{B}}$: $(\mathbb{B} \to \mathbb{R}) \to \mathbb{R}$ and logsumexp$_{\mathbb{N}}$: $\mathbb{N} \to (\mathbb{N} \to \mathbb{R}) \to \mathbb{R}$ are the usual log-sum-exp functions for log-domain sums. The choose primitive randomly selects a natural number in a given range, but unlike the sample primitive, choose does not otherwise incur any probabilistic side effects.

Handling expressive language features. The hmm example uses recursion, which violates the assumptions of many existing approaches to PPL inference. Instead of using recursion to define the HMM, one could perform exact inference by unfolding the model into a fixed number of iterations and then applying existing inference methods good at nonrecursive programs. But this approach is awkward when the number of iterations is not known statically—namely, when data is dynamically sized. A key design goal of Mappl is to support a broad class of models definable with bounded recursion where the bound may not be known statically.

Unrolling recursion is even more awkward for models that are not iterative but properly recursive and for models where control flow is stochastic. A prime example is PCFGs. Figure 1c shows a PCFG model in Mappl that samples parse trees for the simple grammar $S \longrightarrow$ a $(0.5) \mid SS\,(0.5)$. The program is conditioned on it generating a parse tree for a sequence of words. The recursion pattern of pcfg is more complex than the HMMs. First, it is tree-structured rather than linear. Second, control flow is stochastic—which branch of case is taken depends on the sampled variable z in each recursive call. Yet, the Mappl compiler can still compile the program into a pure one that computes the marginal likelihood of observing words. The compilation applies to *mutually* recursive functions, too, which are useful for expressing more complex PCFGs in practice. The pure pcfg in Figure 1d recovers the cubic-time inside algorithm.

An information-flow type system. VE for Bayesian networks has a worst-case exponential running time, but tractable inference is possible for certain models by exploiting independence in the model structure: factors independent of a variable can be factored out of the summation that marginalizes out the variable.

Similarly, VE for recursive programs requires analyzing dependence and independence, for which we use a *static* information-flow analysis. In particular, we design an information-flow type system. Information-flow typing is a compositional, automatable means to reason about dependence, with applications in many different contexts [1], among which the most well-known is language-based security [60]. We repurpose the idea to our expressive PPL.

While information-flow typing for the pure fragment of Mappl is mostly standard, it is less obvious how to design typing rules for the probabilistic fragment. A principle is that the design of the type system should be guided by the denotational semantics. The denotational semantics of a probabilistic computation in Mappl is a measure over the space of its possible outcomes, which can be roughly thought of as a function that, given a set of values, produces the unnormalized probability that the computation returns a value in that set. Accordingly, our type system assigns to a probabilistic computation a labeled type A^ℓ: A types the return value of the computation, and the label ℓ classifies the level of information contained in the measure denoting the computation. For example, consider typing variable bindings in the probabilistic fragment. As expected, the rule requires the composed computation $\mathrm{x} = t;\ m$ to have a label no lower than the labels of the computations t and m being composed. However, it does not explicitly constrain the label ℓ_2 of the variable x being bound. In particular, it is *not* required that the label of x be at least as high as the label of t. This is in keeping with the denotation of $\mathrm{x} = t;\ m$, which is defined by composing the measures that denote t and m and *marginalizing* over the entire support of x.

$$\frac{\Delta;\Psi;\Xi;\mathrm{C};\Gamma \vdash t : A^{\ell_1} \qquad \Delta;\Psi;\Xi;\mathrm{C};\Gamma,\mathrm{x}:A^{\ell_2} \vdash m : B^{\ell_3}}{\Delta;\Psi;\Xi;\mathrm{C};\Gamma \vdash \mathrm{x} = t;\ m : B^{\ell_1 \sqcup \ell_3}}$$

Factorizing computations using information-flow typing. To eliminate a variable from a probabilistic computation, Mappl's VE compiler infers labels for the subcomputations, constraining that the variable being eliminated be labeled H (high). As many subcomputations are inferred to be labeled L (low) as possible. The larger computation can thus be factorized into a H partition and a L partition, and the L partition need not be involved in the elimination of the variable.

For example, in Figure 1a, to eliminate sample(step(z)) from the cons branch, the sampled variable is labeled H, and the information-flow analysis deduces that the probabilistic side effects of the recursive call hmm(z_0, xs) and the conditioning observe(emit(z); x) can be labeled L, while the side effects of sample(step(z)) and any computations in the caller that depend on the return value z must be labeled H. This factorization indicates that in the compiled program, only the H partition needs to be nested under the $\mathrm{logsumexp}_{\mathbb{B}}$ that marginalizes out sample(step(z)).

Compiling with continuations. Continuation-passing style (CPS) transformations [19] are an effective way to eliminate various forms of effects away from a program. It has found applications in the implementation of PPLs [28] and in the cost analysis of randomized algorithms [4, 35]. The Mappl compiler uses CPS to capture dependence in the presence of functions calls and branching.

For example, in the compiled hmm, a continuation of type $\mathbb{B} \to \mathbb{R}$ represents the dependencies of the recursive call's return value z: the continuation takes z as input and returns a log-likelihood that is the transformation of those caller terms whose probabilistic side effects depend on z. In the compiled pcfg, the continuations passed to recursive calls are less involved, as the information-flow analysis deduces that the return value has no nontrivial dependencies.

Decomposition into subproblems with memoization. The compiled hmm and pcfg run in polynomial time, despite the exponentially many possible executions of the input programs. This algorithmic efficiency is because the recursive programs have dynamically recurring substructure, on which the VE compilation capitalizes to generate recurring subproblems and memoize their solutions. For example, the compiled hmm corresponds to the following recursive equations for computing, in log domain, the marginal likelihood $L(\mathrm{k}, \mathrm{z}_0, \mathrm{data})$ of observing data:

$$L(\mathrm{k}, \mathrm{z}_0, \mathrm{nil}) = \mathrm{k}(\mathrm{z}_0)$$
$$L(\mathrm{k}, \mathrm{z}_0, \mathrm{cons\ x\ xs}) = L\Big(\lambda \mathrm{z}.\ \overbrace{\log \Pr(\mathrm{emit}(\mathrm{z}); \mathrm{x})}^{\phi_1} + \overbrace{\log \sum_{\mathrm{y}} \exp\Big(\log \Pr(\mathrm{step}(\mathrm{z}); \mathrm{y}) + \mathrm{k}(\mathrm{y})\Big)}^{\phi_2\text{: nested inference subproblem (summing out y)}}, \mathrm{z}_0, \mathrm{xs}\Big)$$

The generated inference subproblem ϕ_2 eliminates the discrete r.v. sample(step(z)), denoted by y, by summing over its finite support. Although the subproblem is nested inside a continuation, inference time is linear in the length of data: the subproblem needs to be solved at most once for each of the two possible values of z and for each continuation k created—there can only be as many as the length of data. The solution to a subproblem instance, once computed, can be memoized and reused whenever the same subproblem instance is encountered again. This decomposition and memoization is what recovers the dynamic-programming algorithms for HMMs and PCFGs [55, 5]. Contrast this with solving the global problem directly (say, with an enumeration-based approach to exact inference) without first compiling the recursive program to decompose it into subproblems:

$$L(\mathrm{z}_0, \mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{n}) = \log \sum_{\mathrm{z}_1} \sum_{\mathrm{z}_2} \ldots \sum_{\mathrm{z}_\mathrm{n}} \prod_{i=0}^{n-1} \Pr(\mathrm{emit}(\mathrm{z}_\mathrm{i}); \mathrm{x}_{\mathrm{i}+1}) \Pr(\mathrm{step}(\mathrm{z}_\mathrm{i}); \mathrm{z}_{\mathrm{i}+1})$$

The sums over the Boolean variables $\mathrm{z}_1, \mathrm{z}_2, \ldots, \mathrm{z}_\mathrm{n}$ enumerate all possible execution traces of the program. Hence, the inner product has to be computed $O(2^n)$ times, where n is the length of data.

Continuous parameters. Consider hmm′ in Figure 1e, a hybrid discrete–continuous HMM. It is largely the same as hmm except that the emission function takes as input an additional, freshly sampled Gaussian variable w. Directly solving the marginal-inference problem using a general-purpose inference method such as importance sampling would be intractable. Instead, Figure 1f shows that hmm′ is compiled similarly to hmm, with the factor ϕ_1 replaced by a nested inference problem marginalizing out w. Marginal inference with the compiled hmm′ is efficient, even when the inference subproblem logML(...) is solved using general-purpose Monte Carlo methods. The key is that with memoization, this subproblem generated by Mappl's compiler needs to be solved only $O(n)$ times, once for each of the two values of z and for each of the at most n values of x:

$$L(\mathrm{k}, \mathrm{z}_0, \mathrm{cons\ x\ xs}) = L\Big(\lambda \mathrm{z}.\ \overbrace{\log \int_{\mathrm{w}} \exp\Big(\log \Pr(\textsc{Normal}(0,1); \mathrm{w}) + \log \Pr(\mathrm{emit}'(\mathrm{w}, \mathrm{z}); \mathrm{x})\, \mathrm{dw}\Big)}^{\phi_1'\text{: nested inference subproblem (integrating out w)}} + \overbrace{\log \sum_{\mathrm{y}} \exp\Big(\cdots\Big)}^{\phi_2}, \mathrm{z}_0, \mathrm{xs}\Big)$$

A semantic model of information-flow types. Factoring out independent factors is akin to *loop-invariant code motion*, a compiler optimization that moves code outside a loop if it is independent of the loop index. The dependence analysis VE entails is more sophisticated, though, due to the measure-theoretic nature of the *semantics* of probabilistic programs. Fortunately, information-flow typing provides a *syntactic*, principled means to reason about independence.

How can we argue that the syntactic approach of information-flow typing to reasoning about independence is semantically sound in this probabilistic setting? For an information-flow type system, the usual notion of soundness is noninterference [27]. In our novel setting, we take noninterference to mean that the measure denoting a probabilistic computation of a type labeled low behaves irrespective of substitutions for its high-labeled free variables. To prove noninterference, we adapt the semantic, logical-relations proof technique [61, 1], interpreting labeled types as partial equivalence relations on measures indistinguishable to an observer. We believe that our semantic model and its metatheories are the first to introduce observer-sensitive equational reasoning to a measure-theoretic setting and thus are of independent interest.

3 SYNTAX, TYPE SYSTEM, AND DENOTATIONAL SEMANTICS

Syntax. Figure 2 defines the syntax of Mappl programs. Local variables and global variables are notated in blue. An overline denotes a sequence of zero or more elements.

Mappl has a pure, deterministic fragment and a monadic, probabilistic fragment, similar to some prior PPL formalisms [67, 37, 36]. Pure computations take the form of *expressions*. The pure fragment is a simply typed λ-calculus equipped with real numbers, pairs, sums, iso-recursive types, n-ary operations, and two primitive distributions (Bernoulli and normal, representative of discrete and continuous distributions). A special binary operation $\mathsf{logPr}(d; e)$ gives the log-probability density or mass of a distribution d at a point e.

Recursive types, sum types, and product types together enable the expression of algebraic data types, including Booleans and lists: $\mathbb{B} \overset{\text{def}}{=} \mathbb{U} + \mathbb{U}$, $\mathsf{list}_\tau \overset{\text{def}}{=} \mu\alpha.\, \mathbb{U} + (\tau \times \alpha)$. Distributions have type $\mathsf{dist}(\sigma)$, where $\sigma = \mathbb{B}$ for a Bernoulli distribution and $\sigma = \mathbb{R}$ for a normal distribution.

Probabilistic computations are in the forms of *terms* and *commands*. A command sequences terms. The return value of a command is that of its last term. We will write t for $t\,\$$ when it is clear from context that t is being used as the last term of a command. Terms have the following forms: $\mathsf{ret}(e)$ returns the value of a pure expression e, $\mathsf{sample}(d)$ samples from a distribution d, $\mathsf{factor}(e)$ conditions the program using a log-domain expression e, $\mathsf{case}(e; \mathrm{x}.m_1; \mathrm{x}.m_2)$ branches on a sum-typed expression e, and $\mathrm{f}(\overline{e})$ invokes a global function f with arguments $\overline{e}$. The factor form supports soft constraints, which subsume conditioning on continuous observations—that is, $\mathsf{observe}(d; e)$ can be encoded as $\mathsf{factor}(\mathsf{logPr}(d; e))$. For brevity of presentation, we omit hard constraints (i.e., $\mathsf{factor}(-\infty)$), but they are straightforward to incorporate in both the syntax and the semantics. We sometimes write $\mathsf{sample}_\sigma(d)$ and $\mathsf{logPr}_\sigma(d; e)$, where σ is $\mathbb{B}$ or $\mathbb{R}$, to indicate the type of the support of the distribution d.

The pure fragment supports nested marginal inference via the form $\mathsf{logML}(m)$. While the command m is probabilistic, $\mathsf{logML}(m)$ is a pure expression, since inference handles the probabilistic effects of m. It returns the log-marginal likelihood of the probabilistic computation m.

A program in Mappl is composed of a set of global definitions and a *main command*. The global definitions can be either pure ($\mathcal{G}$) or probabilistic ($\mathcal{F}$). Importantly, Mappl allows mutual recursion among pure globals and among probabilistic globals.

Recursion leaves open the possibility of nontermination, but for VE in this paper, we do not concern ourselves with programs that have possibility of not terminating.

Type system. The base type system for Mappl is standard. Figure 2 shows selected rules. Expression typing judgments have the form $\Delta; \Gamma \vdash e : \tau$, where Δ is a context mapping names of pure global definitions to their types, and Γ is a context mapping local variables to their types. Term and command typing judgments have the forms $\Delta; \Psi; \Gamma \vdash t : \tau$ and $\Delta; \Psi; \Gamma \vdash m : \tau$. Computations in the probabilistic fragment can use probabilistic globals, whose types are provided by the context Ψ.

Denotational semantics. We use ω-quasi Borel spaces (ωqbses) [64] as the semantic domain. ωqbses are as a drop-in replacement for measurable spaces. They enable carrying out measure theory in the presence of higher-order types and recursive types, by providing well-behaved

$$
\begin{array}{llll}
\text{expressions} & e, d & ::= & \mathrm{x} \mid \mathsf{unit} \mid r \mid \lambda \mathrm{x}.\, e \mid e_1\, e_2 \mid \langle e_1, e_2 \rangle \mid \pi_1(e) \mid \pi_2(e) \\
 & & & \mid \mathsf{inl}\, e \mid \mathsf{inr}\, e \mid \mathsf{case}(e; \mathrm{x}.e_1; \mathrm{x}.e_2) \mid \mathsf{let}(e_1; \mathrm{x}.e_2) \\
 & & & \mid \mathsf{roll}(e) \mid \mathsf{unroll}(e_1; \mathrm{x}.e_2) \mid \mathsf{op}(e_1, \ldots, e_n) \\
 & & & \mid D(e_1, \ldots, e_n) \mid \mathsf{logPr}(d; e) \mid \mathsf{logML}(m) \\
\text{distributions} & D & ::= & \textsc{Bern} \mid \textsc{Normal} \\
\text{terms} & t & ::= & \mathsf{ret}(e) \mid \mathsf{sample}(d) \mid \mathsf{factor}(e) \mid \mathrm{f}(e_1, \ldots, e_n) \\
 & & & \mid \mathsf{case}(e; \mathrm{x}.m_1; \mathrm{x}.m_2) \\
\text{commands} & m & ::= & t\,\$ \mid \mathrm{x} = t;\, m \\
\text{pure globals} & \mathcal{G} & ::= & \mathsf{def}\ \mathrm{x} = e \\
\text{prob. globals} & \mathcal{F} & ::= & \mathsf{def}\ \mathrm{f}(\mathrm{x}_1, \ldots, \mathrm{x}_n) = m \\
\text{programs} & \mathcal{P} & ::= & \overline{\mathcal{G}}; \overline{\mathcal{F}}; m
\end{array}
$$

$$
\begin{array}{lll}
\text{types} & & \\
\tau, \sigma & ::= & \mathbb{U} \mid \mathbb{R} \mid \alpha \mid \mathsf{dist}(\tau) \mid \mu\alpha.\, \tau \\
 & & \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \\
\text{contexts of locals} & & \\
\Gamma & ::= & \varnothing \mid \Gamma, \mathrm{x} : \tau \\
\text{contexts of pure globals} & & \\
\Delta & ::= & \varnothing \mid \Delta, \mathrm{x} : \tau \\
\text{types of probabilistic globals} & & \\
F & ::= & (\tau_1, \ldots, \tau_n) \to \tau \\
\text{contexts of probabilistic globals} & & \\
\Psi & ::= & \varnothing \mid \Psi, \mathrm{f} : F
\end{array}
$$

$$
\frac{\Gamma(\mathrm{x}) = \tau}{\Delta; \Gamma \vdash \mathrm{x} : \tau}
\qquad
\frac{\mathrm{x} \notin \mathrm{dom}(\Gamma) \quad \Delta(\mathrm{x}) = \tau}{\Delta; \Gamma \vdash \mathrm{x} : \tau}
\qquad
\frac{\Delta; \Gamma \vdash e : \mathbb{R}}{\Delta; \Gamma \vdash \textsc{Bern}(e) : \mathsf{dist}(\mathbb{B})}
\qquad
\frac{\Delta; \Gamma \vdash e_1 : \mathbb{R} \quad \Delta; \Gamma \vdash e_2 : \mathbb{R}}{\Delta; \Gamma \vdash \textsc{Normal}(e_1, e_2) : \mathsf{dist}(\mathbb{R})}
$$

$$
\frac{\Delta; \Gamma \vdash d : \mathsf{dist}(\tau) \quad \Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \mathsf{logPr}(d; e) : \mathbb{R}}
\qquad
\frac{\Delta; \varnothing; \Gamma \vdash m : \mathbb{U}}{\Delta; \Gamma \vdash \mathsf{logML}(m) : \mathbb{R}}
\qquad
\frac{\Delta; \Gamma \vdash e : \tau}{\Delta; \Psi; \Gamma \vdash \mathsf{ret}(e) : \tau}
\qquad
\frac{\Delta; \Gamma \vdash e : \mathbb{R}}{\Delta; \Psi; \Gamma \vdash \mathsf{factor}(e) : \mathbb{U}}
$$

$$
\frac{\Delta; \Gamma \vdash e : \tau_1 + \tau_2 \quad \Delta; \Psi; \Gamma, \mathrm{x} : \tau_i \vdash m_i : \tau \ \mathit{for}\ i \in \{1, 2\}}{\Delta; \Psi; \Gamma \vdash \mathsf{case}(e; \mathrm{x}.m_1; \mathrm{x}.m_2) : \tau}
\qquad
\frac{\Psi(\mathrm{f}) = (\tau_1, \ldots, \tau_n) \to \tau \quad \Delta; \Gamma \vdash e_i : \tau_i \ \mathit{for}\ i \in \{1, \ldots, n\}}{\Delta; \Psi; \Gamma \vdash \mathrm{f}(e_1, \ldots, e_n) : \tau}
$$

$$
\frac{\Delta; \Gamma \vdash d : \mathsf{dist}(\sigma)}{\Delta; \Psi; \Gamma \vdash \mathsf{sample}(d) : \sigma}
\qquad
\frac{\Delta; \Psi; \Gamma \vdash t : \tau}{\Delta; \Psi; \Gamma \vdash t\,\$: \tau}
\qquad
\frac{\Delta; \Psi; \Gamma \vdash t : \tau \quad \Delta; \Psi; \Gamma, \mathrm{x} : \tau \vdash m : \tau'}{\Delta; \Psi; \Gamma \vdash \mathrm{x} = t;\, m : \tau'}
$$

$$
\frac{\begin{array}{c}\Psi(\mathrm{f}) = (\tau_1, \ldots, \tau_n) \to \tau \\ \Delta; \Psi; \mathrm{x}_1 : \tau_1, \ldots, \mathrm{x}_n : \tau_n \vdash m : \tau\end{array}}{\Delta; \Psi \vdash \mathsf{def}\ \mathrm{f}(\mathrm{x}_1, \ldots, \mathrm{x}_n) = m : \Psi(\mathrm{f})}
\qquad
\frac{\begin{array}{c}\mathrm{dom}(\Psi) = \{\mathrm{f_i} \mid \mathcal{F}_i \in \overline{\mathcal{F}} \wedge \mathrm{name}(\mathcal{F}_i) = \mathrm{f_i}\} \\ \Delta; \Psi \vdash \mathcal{F}_i : \Psi(\mathrm{f_i}) \ \mathit{for}\ \mathcal{F}_i \in \overline{\mathcal{F}} \ \mathit{s.t.}\ \mathrm{name}(\mathcal{F}_i) = \mathrm{f_i}\end{array}}{\Delta; \Psi \vdash \overline{\mathcal{F}} : \Psi}
\qquad
\frac{\begin{array}{c}\Delta \vdash \overline{\mathcal{G}} : \Delta \quad \Delta; \Psi \vdash \overline{\mathcal{F}} : \Psi \\ \Delta; \Psi; \varnothing \vdash m : \tau\end{array}}{\vdash \overline{\mathcal{G}}; \overline{\mathcal{F}}; m}
$$

Figure 2. Syntax of Mappl and selected typing rules.

function spaces and ωcpos. Thus, in this paper, we use standard measure-theory notations with the understanding that we are working with ωqbses. For a measurable function $f : X \to \mathbb{R}^{+}$, the Lebesgue integral of f with respect to a measure μ on X is denoted $\int f \mathrm{d}\mu$ or $\int f(x)\mu(\mathrm{d}x)$.

The semantic interpretation $[\![\tau]\!]$ of each type τ is an ωqbs. Types in Mappl are similar to those in the SFPC calculus of Vákár et al. [64], which has function types and iso-recursive types; we refer the reader to their paper for detailed constructions of the ωqbses. For an ωqbs X, we write $X_{\bot}$ for the lifting of $[\![\tau]\!]$ to another ωqbs with an extra element $\bot$ signifying partiality. There is a commutative strong monad of measures on ωqbses [64]; for an ωqbs X, we write Meas X for the ωqbs of measures on X.

Figure 3 shows selected interpretations of expressions, terms, and commands. The definitions use the operator $\gg\!=$ to handle partiality. In addition to $\gg\!= : [\![\tau_1]\!]_{\bot} \to ([\![\tau_1]\!] \to [\![\tau_2]\!]_{\bot}) \to [\![\tau_2]\!]_{\bot}$, we overload $\gg\!=$ on $[\![\tau_1]\!]_{\bot} \to ([\![\tau_1]\!] \to \mathrm{Meas}\,[\![\tau_2]\!]_{\bot}) \to \mathrm{Meas}\,[\![\tau_2]\!]_{\bot}$ such that $v \gg\!= f \overset{\text{def}}{=}$ if $v = \bot$ then $\lambda E.\, \mathbb{1}_E(\bot)$ else $f(v)$. Here, $\mathbb{1}_E(v)$ is the indicator function that is 1 if $v \in E$ and 0 otherwise.

The contract for interpreting expressions is that an expression of type τ is interpreted as an element of $[\![\tau]\!]_{\bot}$. The denotation $[\![e]\!]_{\delta;\gamma}$ of an expression e typed in contexts Δ and Γ is interpreted under substitutions $\delta \in [\![\Delta]\!]$ and $\gamma \in [\![\Gamma]\!]$ for the bindings in Δ and Γ. That is, δ and γ provide semantic interpretations for the bound global and local variables. The denotation of a primitive distribution is its probability density or mass function.

The contract for interpreting terms and commands is that a term or command of type τ is interpreted as a measure on $[\![\tau]\!]_{\bot}$—i.e., an element of Meas $[\![\tau]\!]_{\bot}$. The interpretations $[\![t]\!]_{\delta;\psi;\gamma}$ and

$$\llbracket \mathrm{x} \rrbracket_{\delta;\gamma} \overset{\text{def}}{=} \gamma(\mathrm{x}) \text{ if } \mathrm{x} \in \mathrm{dom}(\gamma)$$
$$\llbracket \mathrm{x} \rrbracket_{\delta;\gamma} \overset{\text{def}}{=} \delta(\mathrm{x}) \text{ if } \mathrm{x} \notin \mathrm{dom}(\gamma) \wedge \mathrm{x} \in \mathrm{dom}(\delta)$$
$$\llbracket \mathsf{inl}\ e \rrbracket_{\delta;\gamma} \overset{\text{def}}{=} \mathsf{inl}\ \llbracket e \rrbracket_{\delta;\gamma}$$
$$\llbracket \textsc{Bern}(e) \rrbracket_{\delta;\gamma} \overset{\text{def}}{=} \llbracket e \rrbracket_{\delta;\gamma} \gg\!= \lambda p.\ \mathsf{if}\ 0 \le p \le 1\ \mathsf{then}\ (\lambda v.\ \mathsf{if}\ v\ \mathsf{then}\ p\ \mathsf{else}\ 1-p)\ \mathsf{else}\ \bot$$
$$\llbracket \mathsf{logPr}(d;e) \rrbracket_{\delta;\gamma} \overset{\text{def}}{=} \llbracket d \rrbracket_{\delta;\gamma} \gg\!= \lambda f.\ \llbracket e \rrbracket_{\delta;\gamma} \gg\!= \lambda v.\ \log f(v)$$
$$\llbracket \mathsf{logML}(m) \rrbracket_{\delta;\gamma} \overset{\text{def}}{=} \mathsf{let}\ r = \llbracket m \rrbracket_{\delta;\varnothing;\gamma}(\llbracket \mathbb{U} \rrbracket)\ \mathsf{in}\ \mathsf{if}\ 0 < r < \infty\ \mathsf{then}\ \log r\ \mathsf{else}\ \bot$$
$$\llbracket \mathsf{ret}(e) \rrbracket_{\delta;\psi;\gamma} \overset{\text{def}}{=} \lambda E.\ \mathbb{1}_E(\llbracket e \rrbracket_{\delta;\gamma})$$
$$\llbracket \mathsf{case}(e;\mathrm{x}.m_1;\mathrm{x}.m_2) \rrbracket_{\delta;\psi;\gamma} \overset{\text{def}}{=} \llbracket e \rrbracket_{\delta;\gamma} \gg\!= \lambda v.\ \mathsf{case}\ v\ \mathsf{of}\ \mathsf{inl}\ u \Rightarrow \llbracket m_1 \rrbracket_{\delta;\psi;\gamma[\mathrm{x}\mapsto u]} \mid \mathsf{inr}\ u \Rightarrow \llbracket m_2 \rrbracket_{\delta;\psi;\gamma[\mathrm{x}\mapsto u]}\ \mathsf{end}$$
$$\llbracket \mathsf{factor}(e) \rrbracket_{\delta;\psi;\gamma} \overset{\text{def}}{=} \llbracket e \rrbracket_{\delta;\gamma} \gg\!= \lambda r.\ \lambda E.\ \exp(r) \cdot \mathbb{1}_E(\mathsf{unit})$$
$$\llbracket \mathsf{sample}_\sigma(d) \rrbracket_{\delta;\psi;\gamma} \overset{\text{def}}{=} \llbracket d \rrbracket_{\delta;\gamma} \gg\!= \lambda f.\ \lambda E.\ \int_E f \, \mathrm{d}\nu_\sigma$$
where ν_σ is the reference measure over $\llbracket \sigma \rrbracket$
$$\llbracket \mathrm{f}(e_1,\dots,e_n) \rrbracket_{\delta;\psi;\gamma} \overset{\text{def}}{=} \llbracket e_1 \rrbracket_{\delta;\gamma} \gg\!= \lambda v_1.\ \dots\ \llbracket e_n \rrbracket_{\delta;\gamma} \gg\!= \lambda v_n.\ \psi(\mathrm{f})(v_1,\dots,v_n)$$
$$\llbracket t\ \$ \rrbracket_{\delta;\psi;\gamma} \overset{\text{def}}{=} \llbracket t \rrbracket_{\delta;\psi;\gamma}$$
$$\llbracket \mathrm{x} = t;\ m \rrbracket_{\delta;\psi;\gamma} \overset{\text{def}}{=} \lambda E.\ \int_u \left(u \gg\!= \lambda v.\ \llbracket m \rrbracket_{\delta;\psi;\gamma[\mathrm{x}\mapsto v]} \right)(E) \llbracket t \rrbracket_{\delta;\psi;\gamma}(\mathrm{d}u)$$
$$\llbracket \mathsf{def}\ \mathrm{x} = e \rrbracket_{\delta} \overset{\text{def}}{=} \llbracket e \rrbracket_{\delta;\varnothing}$$
$$\llbracket \mathsf{def}\ \mathrm{f}(\overline{\mathrm{x}}) = m \rrbracket_{\delta;\psi} \overset{\text{def}}{=} \lambda \overline{v}.\ \llbracket m \rrbracket_{\delta;\psi;\{\overline{\mathrm{x}\mapsto v}\}}$$
$$\llbracket \mathcal{G}_1 \dots \mathcal{G}_n \rrbracket_{\delta} \overset{\text{def}}{=} \{\mathrm{g}_1 \mapsto \llbracket \mathcal{G}_1 \rrbracket_{\delta}, \dots, \mathrm{g}_n \mapsto \llbracket \mathcal{G}_n \rrbracket_{\delta}\}$$
where $\mathrm{name}(\mathcal{G}_i) = \mathrm{g}_i$ for $i \in \{1, \dots, n\}$
$$\llbracket \mathcal{F}_1 \dots \mathcal{F}_n \rrbracket_{\delta;\psi} \overset{\text{def}}{=} \{\mathrm{f}_1 \mapsto \llbracket \mathcal{F}_1 \rrbracket_{\delta;\psi}, \dots, \mathrm{f}_n \mapsto \llbracket \mathcal{F}_n \rrbracket_{\delta;\psi}\}$$
where $\mathrm{name}(\mathcal{F}_i) = \mathrm{f}_i$ for $i \in \{1, \dots, n\}$

Figure 3. Selected definitions of the denotational semantics for Mappl.

$\llbracket m \rrbracket_{\delta;\psi;\gamma}$ are additionally indexed by a semantic substitution $\psi \in \llbracket \Psi \rrbracket$ for the global variables bound in Ψ. The denotation of $\mathsf{sample}_\sigma(d)$ is a measure over $\llbracket \sigma \rrbracket$, obtained by integrating the density function denoting d with respect to either the Lebesgue measure or the counting measure, depending on whether σ is $\mathbb{R}$ or $\mathbb{B}$. The denotation of $\mathrm{x} = t;\ m$ composes the denotations of t and m, integrating the measure denoting m with respect to the measure denoting t. The pure expression $\mathsf{logML}(m)$ is denoted by the logarithm of the measure denoting the $\mathbb{U}$-typed m on $\llbracket \mathbb{U} \rrbracket$.

Global definitions are mutually recursive and thus are interpreted under semantic substitutions too. For a program $\overline{\mathcal{G}};\overline{\mathcal{F}};m$, the index-free denotations are given by the fixpoints $\llbracket \overline{\mathcal{G}} \rrbracket = \delta_* \overset{\text{def}}{=} \mathsf{fix}\,\delta.\ \llbracket \overline{\mathcal{G}} \rrbracket_\delta$ and $\llbracket \overline{\mathcal{F}} \rrbracket = \psi_* \overset{\text{def}}{=} \mathsf{fix}\,\psi.\ \llbracket \overline{\mathcal{F}} \rrbracket_{\delta_*;\psi}$ We shall write $\llbracket m \rrbracket_\gamma$ for $\llbracket m \rrbracket_{\delta_*;\psi_*;\gamma}$; similarly for $\llbracket t \rrbracket_\gamma$ and $\llbracket e \rrbracket_\gamma$.

4 INFORMATION-FLOW TYPE SYSTEM

Syntax. Figure 4 shows the syntax of labels, types, and contexts of the information-flow type system. The type system is parameterized by a join semilattice $\mathcal{L}$ of labels, although in this work, we will need only the two-level lattice $\{\mathsf{L}, \mathsf{H}\}$ with $\mathsf{L} \sqsubseteq \mathsf{H}$.

Types are of the form A^ℓ, where A is an unlabeled type that is further constructed from labeled types. Metavariables τ range over (labeled) types. We differentiate them from types τ in the base type system by typesetting labeled types in upright font and in purple. Similarly, we typeset metavariables Δ, Ψ, and Γ differently than Δ, Ψ, and Γ, as Δ, Ψ, and Γ now contain labeled types.

The type system supports label polymorphism, as well as ordering constraints on labels, for functions—that is, the type of a function can be parameterized by label variables η and ordering constraints of the form $\ell_1 \sqsubseteq \ell_2$. Label polymorphism allows more reusable code [45, 44].

Typing the pure fragment. Typing judgments for expressions have the form $\Delta;\Xi;\mathrm{C};\Gamma \vdash e : \tau$, where Ξ records the label variables in scope and C the label ordering constraints. These rules are largely standard. In particular, introduction rules (e.g., those for unit, lambdas, pairs, and distributions) do not constrain the label of the expression—the label can be arbitrarily low. A subsumption rule exists to allow weakening (i.e., increasing) the label of an expression. Subtyping rules are standard and thus omitted. For a labeled type A^ℓ, subtyping $\le$ is covariant in both A and ℓ.

The type system uses fine-grained labeling, in that every type, top-level or nested, is labeled [56]. For example, the type of a pair takes the form $(A_1^{\ell_1} \times A_2^{\ell_2})^{\ell_3}$. The nested labels ℓ_1 and ℓ_2 classify the

$$
\begin{array}{lll}
\text{labels} & \ell, O & ::= \eta \mid l \in \mathcal{L} \mid \ell_1 \sqcup \ell_2 \\
\text{unlabeled types} & A, B & ::= \mathbb{U} \mid \mathbb{R} \mid \alpha \mid \mathsf{dist}(\tau) \mid \mu\alpha.\,\tau \\
 & & \mid \forall[\Xi|C].\tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \\
\text{labeled types} & \tau & ::= A^{\ell} \\
\text{types of prob. globals} & F & ::= \forall[\Xi|C].(\tau_1, \ldots, \tau_n) \to \tau_0
\end{array}
\qquad
\begin{array}{lll}
\text{contexts of label variables} & \Xi & ::= \varnothing \mid \Xi, \eta \\
\text{label constraints} & C & ::= \varnothing \mid C, \ell_1 \sqsubseteq \ell_2 \\
\text{contexts of locals} & \Gamma & ::= \varnothing \mid \Gamma, x : \tau \\
\text{contexts of pure globals} & \Delta & ::= \varnothing \mid \Delta, x : \tau \\
\text{contexts of prob. globals} & \Psi & ::= \varnothing \mid \Psi, f : F
\end{array}
$$

$$
\frac{\Delta;\Xi;C;\Gamma \vdash e : \tau \quad \Xi;C;\mathcal{L} \vdash \tau \le \tau'}{\Delta;\Xi;C;\Gamma \vdash e : \tau'}
\qquad
\frac{\Delta;\Psi;\Xi;C;\Gamma \vdash t : \tau \quad \Xi;C;\mathcal{L} \vdash \tau \le \tau'}{\Delta;\Psi;\Xi;C;\Gamma \vdash t : \tau'}
\qquad
\frac{\Delta;\Psi;\Xi;C;\Gamma \vdash m : \tau \quad \Xi;C;\mathcal{L} \vdash \tau \le \tau'}{\Delta;\Psi;\Xi;C;\Gamma \vdash m : \tau'}
$$

$$
\frac{}{\Delta;\Xi;C;\Gamma \vdash \mathsf{unit} : \mathbb{U}^{\ell}}
\qquad
\frac{\Gamma(x) = \tau}{\Delta;\Xi;C;\Gamma \vdash x : \tau}
\qquad
\frac{x \notin \mathrm{dom}(\Gamma) \quad \Delta(x) = \tau}{\Delta;\Xi;C;\Gamma \vdash x : \tau}
\qquad
\frac{\Delta;\Xi;C;\Gamma \vdash d : \mathsf{dist}(A^{\ell})^{\ell} \quad \Delta;\Xi;C;\Gamma \vdash e : A^{\ell}}{\Delta;\Xi;C;\Gamma \vdash \mathsf{logPr}(d;e) : \mathbb{R}^{\ell}}
$$

$$
\frac{\Delta;\Xi;C;\Gamma \vdash e : \mathbb{R}^{\ell_1}}{\Delta;\Xi;C;\Gamma \vdash \textsc{Bern}(e) : \mathsf{dist}(\mathbb{B}^{\ell_1})^{\ell_2}}
\qquad
\frac{\Delta;\Xi;C;\Gamma \vdash e_1 : \mathbb{R}^{\ell_1} \quad \Delta;\Xi;C;\Gamma \vdash e_2 : \mathbb{R}^{\ell_1}}{\Delta;\Xi;C;\Gamma \vdash \textsc{Normal}(e_1, e_2) : \mathsf{dist}(\mathbb{R}^{\ell_1})^{\ell_2}}
\qquad
\frac{\Delta;\varnothing;\Xi;C;\Gamma \vdash m : \mathbb{U}^{\ell}}{\Delta;\Xi;C;\Gamma \vdash \mathsf{logML}(m) : \mathbb{R}^{\ell}}
$$

$$
\frac{\Delta;\Xi;C;\Gamma \vdash e : \tau}{\Delta;\Psi;\Xi;C;\Gamma \vdash \mathsf{ret}(e) : \tau}
\qquad
\frac{\Delta;\Xi;C;\Gamma \vdash e : \mathbb{R}^{\ell}}{\Delta;\Psi;\Xi;C;\Gamma \vdash \mathsf{factor}(e) : \mathbb{U}^{\ell}}
\qquad
\frac{\Delta;\Xi;C;\Gamma \vdash d : \mathsf{dist}(A^{\ell})^{\ell}}{\Delta;\Psi;\Xi;C;\Gamma \vdash \mathsf{sample}(d) : A^{\ell}}
$$

$$
\frac{\begin{array}{c}\Delta;\Xi;C;\Gamma \vdash e : (\tau_1 + \tau_2)^{\ell} \\ \Delta;\Psi;\Xi;C;\Gamma, x : \tau_i \vdash m_i : A^{\ell} \ \ \textit{for}\ i \in \{1,2\}\end{array}}{\Delta;\Psi;\Xi;C;\Gamma \vdash \mathsf{case}(e; x.m_1; x.m_2) : A^{\ell}}
\qquad
\frac{\begin{array}{c}\Psi(f) = \forall[\overline{\eta}|C_f].(\tau_1, \ldots, \tau_n) \to \tau_0 \quad \Xi;C;\mathcal{L} \vdash C_f\{\overline{\ell}/\overline{\eta}\} \\ \Delta;\Xi;C;\Gamma \vdash e_i : \tau_i\{\overline{\ell}/\overline{\eta}\}\ \textit{for}\ i \in \{1, \ldots, n\}\end{array}}{\Delta;\Psi;\Xi;C;\Gamma \vdash f(e_1, \ldots, e_n) : \tau_0\{\overline{\ell}/\overline{\eta}\}}
$$

$$
\frac{\Delta;\Psi;\Xi;C;\Gamma \vdash t : \tau}{\Delta;\Psi;\Xi;C;\Gamma \vdash t\,\$: \tau}
\qquad
\frac{\begin{array}{c}\Delta;\Psi;\Xi;C;\Gamma \vdash t : A^{\ell} \\ \Delta;\Psi;\Xi;C;\Gamma, x : A^{\ell'} \vdash m : B^{\ell}\end{array}}{\Delta;\Psi;\Xi;C;\Gamma \vdash x = t;\ m : B^{\ell}}
\qquad
\frac{\begin{array}{c}\Psi(f) = \forall[\Xi|C].(\tau_1, \ldots, \tau_n) \to \tau \\ \Delta;\Psi;\Xi;C;x_1 : \tau_1, \ldots, x_n : \tau_n \vdash m : \tau\end{array}}{\Delta;\Psi \vdash \mathsf{def}\ f(x_1, \ldots, x_n) = m : \Psi(f)}
$$

Figure 4. Syntax of information-flow types and selected rules of the information-flow type system.

contents of the pair, while the top-level label ℓ_3 classifies the reference to the pair. The distinction enables fine-grained control over the flow of information.

The introduction rules for primitive distributions use the type $\mathsf{dist}(A^{\ell_1})^{\ell_2}$, where A is either $\mathbb{B}$ or $\mathbb{R}$, ℓ_1 classifies the contents of the distribution (i.e., how the r.v. is distributed), and ℓ_2 classifies the reference to the distribution. For instance, given $x : \mathbb{B}^{\mathsf{H}}$, the expression $\textsc{Bern}(\mathsf{case}(x; \_.0.7; \_.0.1))$ can be typed at $\mathsf{dist}(\mathbb{B}^{\mathsf{H}})^{\mathsf{L}}$. This fine-grained labeling allows the distribution to be stored in a data structure that can only hold L references, while controlling that when the distribution is eventually retrieved and sampled, the probabilistic effects are classified at H. Whereas coarser-grained type systems trade fine-grained control for reduced label annotation burden, label annotation is not a concern in our setting, because programmers do not specify security policies through labels as they would in a security-typed language. Instead, labels are automatically inferred.

Typing the probabilistic fragment. The design of the type system is guided by the denotational semantics: while the label of an expression e is designed to classify the information the semantic value $[\![e]\!]_{\delta;\gamma}$ contains, the label of a term t or a command m should classify the information the measure $[\![t]\!]_{\delta;\psi;\gamma}$ or $[\![m]\!]_{\delta;\psi;\gamma}$ contains.

Consider typing $\mathsf{sample}(d)$, where d has type $\mathsf{dist}(A^{\ell_1})^{\ell_2}$. Since the contents of the distribution d, as well as the identity of it, determine the measure denoting $\mathsf{sample}(d)$, the term should be typed at a level no lower than $\ell_1 \sqcup \ell_2$. In Figure 4, the typing rule for $\mathsf{sample}(d)$ handles distributions that can be typed at $\mathsf{dist}(A^{\ell})^{\ell}$. This rule suffices, as the type $\mathsf{dist}(A^{\ell_1})^{\ell_2}$ is covariant in both ℓ_1 and ℓ_2: both labels can be weakened to a label $\ell \sqsupseteq \ell_1 \sqcup \ell_2$ by subsumption. Consider typing $\mathsf{case}(e; x.m_1; x.m_2)$, where e has type $(\tau_1 + \tau_2)^{\ell}$. Information flows, via a control structure, from e to the measure over the possible outcomes of evaluating the term. So the term should be classified at a level no lower than ℓ.

Typing a call to a probabilistic global function checks, with $\Xi; \mathsf{C}; \mathcal{L} \vdash \mathsf{C}_f\{\overline{\ell}/\overline{\eta}\}$, that the constraints specified in the function's type (after substitution) are satisfied under the current context.

Now consider typing $\mathrm{x} = t;\ m$. Since the denotation is defined by composing the two measures $[\![t]\!]$ and $[\![m]\!]$, the label of $\mathrm{x} = t;\ m$ is required to be no lower than the labels of t and m. It is perhaps surprising that in the typing rule, the label ℓ' of x is not required to be at least as high as the label ℓ of t. An explanation is that denotationally, t merely defines a measure on the possible values x can take; it does not determine the value of x in any given run.

This wrinkle has implications for the precision of the information-flow analysis. Consider the example below. The unnormalized joint density of x and y consists of three factors: $\phi_1(\mathrm{x})\phi_2(\mathrm{x},\mathrm{y})\phi_3(\mathrm{y})$.

```
x = sample(Bern(.5))
y = sample(Bern(case(x; _.0.7; _.0.1)))
factor(case(y; _.8; _.2))
```

Suppose that we want to marginalize out x. Labeling x at H, we hope to type the third term at L, to justify that it need not be involved in the marginalization of x: $\sum_{\mathrm{x}} \phi_1(\mathrm{x})\phi_2(\mathrm{x},\mathrm{y})\phi_3(\mathrm{y}) = \phi_3(\mathrm{y}) \sum_{\mathrm{x}} \phi_1(\mathrm{x})\phi_2(\mathrm{x},\mathrm{y})$. The typing rule for $\mathrm{x} = t;\ m$ allows y to be labeled L, despite that its right-hand side term needs to be typed at H. In contrast, if the typing rule required y to be labeled H, then the third term would also have to be typed at H, which would disallow the third term to be factored out of the sum.

Finally, the expression $\mathsf{logML}(m)$ is typed at a label no lower than the command m's label, since the measure denoting m determines the model evidence of the probabilistic computation m.

Remarks. In the presence of side effects such as mutable state, information-flow type systems often use a program-counter label [22] to *lower*-bound information leaked through side effects. Indeed, in the imperative while-language of SlicStan [29], typing judgments of commands do use a label to lower-bound the write effects of commands. In Mappl, however, the label of a command is an *upper* bound that directly classifies the information that can flow into the command's measure denotation, just as an expression's label upper-bounds the information that can flow into the expression's denotation. In SlicStan, by contrast, factorization must first produce an upper bound by joining the labels of its subexpressions.

5 NONINTERFERENCE VIA A LOGICAL-RELATIONS MODEL

Semantic types. We now establish the soundness of the information-flow type system, by constructing a semantic model of the types. Figure 5 defines our logical-relations model.

The definition uses a function $\lfloor\cdot\rfloor$ that strips a type of all labels occurring in it; $\lfloor\cdot\rfloor$ sends a type in the information-flow type system (Section 4) to a type in the base type system (Section 3). The function is overloaded on labeled types τ, unlabeled types A, and contexts Δ, Ψ, Γ.

The main idea behind our model is to interpret each type τ as two binary relations: a value relation $\mathcal{V}[\![\tau]\!]^{O}_{\xi}$ for semantic typing of pure expressions, and a measure relation $\mathcal{M}[\![\tau]\!]^{O}_{\xi}$ for semantic typing of probabilistic computations. Both relations are parameterized by a label O that stands for the "security clearance" of an observer, and by a substitution ξ for the label variables occurring free in τ. The model is constructed by first defining, by induction on types, the interpretations $\mathcal{V}[\![A]\!]^{O}_{\xi;\theta}$, $\mathcal{V}[\![\tau]\!]^{O}_{\xi;\theta}$, and $\mathcal{M}[\![\tau]\!]^{O}_{\xi;\theta}$ parameterized by a semantic substitution θ for free type variables, with fixpoints taken in the case of recursive types. The relations $\mathcal{V}[\![A]\!]^{O}_{\xi}$, $\mathcal{V}[\![\tau]\!]^{O}_{\xi}$, and $\mathcal{M}[\![\tau]\!]^{O}_{\xi}$ are then defined on those types without free type variables.

The value relation $\mathcal{V}[\![A]\!]^{O}_{\xi}$ relates two semantic values in the semantic domain $[\![\lfloor A\rfloor]\!]$ if they are indistinguishable to the observer. What is considered indistinguishable is determined by the observer's label O and the type A. At a ground type $\mathbb{U}$ or $\mathbb{R}$, only identical values are related. At a function type, two functions are related if they send related inputs to related outputs. The relations at product, sum, and recursive types are standard as well. The relation at a distribution type $\mathsf{dist}(A^{\ell})$ relates two density functions.

$$\mathcal{V}\llparenthesis \mathbb{U} \rrparenthesis^{O}_{\xi;\theta}(u_1,u_2) \overset{\text{def}}{=} u_1 = u_2 \in \llbracket \mathbb{U} \rrbracket$$
$$\mathcal{V}\llparenthesis \mathbb{R} \rrparenthesis^{O}_{\xi;\theta}(r_1,r_2) \overset{\text{def}}{=} r_1 = r_2 \in \llbracket \mathbb{R} \rrbracket$$
$$\mathcal{V}\llparenthesis \mathsf{dist}(A^{\ell}) \rrparenthesis^{O}_{\xi;\theta}(f_1,f_2) \overset{\text{def}}{=} \forall v \in \llbracket \lfloor A \rfloor \rrbracket.\ \mathcal{V}\llparenthesis \mathbb{R}^{\ell} \rrparenthesis^{O}_{\xi;\theta}(f_1(v),f_2(v)) \quad \text{where } A = \mathbb{R} \text{ or } \mathbb{B}$$
$$\mathcal{V}\llparenthesis \forall[\Xi|\mathrm{C}].\tau \to \tau' \rrparenthesis^{O}_{\xi;\theta}(f_1,f_2) \overset{\text{def}}{=} \forall \xi' \in \llparenthesis \Xi|\xi\, \mathrm{C} \rrparenthesis.\ \forall(v_1,v_2) \in \mathcal{V}\llparenthesis \tau \rrparenthesis^{O}_{\xi,\xi';\theta}.\ \mathcal{V}_{\perp}\llparenthesis \tau' \rrparenthesis^{O}_{\xi,\xi';\theta}(f_1(v_1),f_2(v_2))$$
$$\mathcal{V}\llparenthesis \tau + \tau' \rrparenthesis^{O}_{\xi;\theta}(v_1,v_2) \overset{\text{def}}{=} \big(v_1 = \mathsf{inl}\ u_1 \wedge v_2 = \mathsf{inl}\ u_2 \wedge \mathcal{V}\llparenthesis \tau \rrparenthesis^{O}_{\xi;\theta}(u_1,u_2)\big) \vee \big(v_1 = \mathsf{inr}\ u_1 \wedge v_2 = \mathsf{inr}\ u_2 \wedge \mathcal{V}\llparenthesis \tau' \rrparenthesis^{O}_{\xi;\theta}(u_1,u_2)\big)$$
$$\mathcal{V}\llparenthesis \tau \times \tau' \rrparenthesis^{O}_{\xi;\theta}(v_1,v_2) \overset{\text{def}}{=} v_1 = (u_1,u_1') \wedge v_2 = (u_2,u_2') \wedge \mathcal{V}\llparenthesis \tau \rrparenthesis^{O}_{\xi;\theta}(u_1,u_1') \wedge \mathcal{V}\llparenthesis \tau' \rrparenthesis^{O}_{\xi;\theta}(u_2,u_2')$$
$$\mathcal{V}\llparenthesis \alpha \rrparenthesis^{O}_{\xi;\theta}(v_1,v_2) \overset{\text{def}}{=} \theta(\alpha)(v_1,v_2)$$
$$\mathcal{V}\llparenthesis \mu\alpha.\,\tau \rrparenthesis^{O}_{\xi;\theta}(v_1,v_2) \overset{\text{def}}{=} \mu R.\ \mathcal{V}\llparenthesis \tau \rrparenthesis^{O}_{\xi;\theta[\alpha\mapsto R]}(v_1,v_2)$$

$$\mathcal{V}\llparenthesis A^{\ell} \rrparenthesis^{O}_{\xi;\theta}(v_1,v_2) \overset{\text{def}}{=} \begin{cases} \mathcal{V}\llparenthesis A \rrparenthesis^{O}_{\xi;\theta}(v_1,v_2) & \text{if } \xi\,\ell \sqsubseteq O, \\ v_1, v_2 \in \llbracket \lfloor A \rfloor \rrbracket & \text{otherwise} \end{cases}$$
$$\mathcal{E}\llparenthesis A \rrparenthesis^{O}_{\xi;\theta}(E_1,E_2) \overset{\text{def}}{=} E_1, E_2 \in \Sigma_{\llbracket \lfloor A \rfloor \rrbracket} \wedge \forall(v_1,v_2) \in \mathcal{V}\llparenthesis A \rrparenthesis^{O}_{\xi;\theta}.\ v_1 \in E_1 \Leftrightarrow v_2 \in E_2$$
$$\mathcal{M}\llparenthesis A^{\ell} \rrparenthesis^{O}_{\xi;\theta}(\mu_1,\mu_2) \overset{\text{def}}{=} \begin{cases} \forall(E_1,E_2) \in \mathcal{E}\llparenthesis A \rrparenthesis^{O}_{\xi;\theta}.\ \mu_1(E_1) = \mu_2(E_2) & \text{if } \xi\,\ell \sqsubseteq O, \\ \mu_1, \mu_2 \in \mathrm{Meas}\,\llbracket \lfloor A \rfloor \rrbracket & \text{otherwise} \end{cases}$$

$$\mathcal{V}\llparenthesis A \rrparenthesis^{O}_{\xi} \overset{\text{def}}{=} \mathcal{V}\llparenthesis A \rrparenthesis^{O}_{\xi;\varnothing} \qquad \mathcal{E}\llparenthesis A \rrparenthesis^{O}_{\xi} \overset{\text{def}}{=} \mathcal{E}\llparenthesis A \rrparenthesis^{O}_{\xi;\varnothing}$$
$$\mathcal{V}\llparenthesis \tau \rrparenthesis^{O}_{\xi} \overset{\text{def}}{=} \mathcal{V}\llparenthesis \tau \rrparenthesis^{O}_{\xi;\varnothing} \qquad \mathcal{M}\llparenthesis \tau \rrparenthesis^{O}_{\xi} \overset{\text{def}}{=} \mathcal{M}\llparenthesis \tau \rrparenthesis^{O}_{\xi;\varnothing}$$

$$\mathcal{V}_{\perp}\llparenthesis A \rrparenthesis^{O}_{\xi}(v_1,v_2) \overset{\text{def}}{=} (v_1 = v_2 = \perp) \vee (v_1,v_2) \in \mathcal{V}\llparenthesis A \rrparenthesis^{O}_{\xi}$$
$$\mathcal{V}_{\perp}\llparenthesis A^{\ell} \rrparenthesis^{O}_{\xi}(v_1,v_2) \overset{\text{def}}{=} \begin{cases} \mathcal{V}_{\perp}\llparenthesis A \rrparenthesis^{O}_{\xi}(v_1,v_2) & \text{if } \xi\,\ell \sqsubseteq O, \\ v_1, v_2 \in \llbracket \lfloor A \rfloor \rrbracket_{\perp} & \text{otherwise} \end{cases}$$
$$\mathcal{E}_{\perp}\llparenthesis A \rrparenthesis^{O}_{\xi}(E_1,E_2) \overset{\text{def}}{=} E_1, E_2 \in \Sigma_{\llbracket \lfloor A \rfloor \rrbracket_{\perp}} \wedge \forall(v_1,v_2) \in \mathcal{V}_{\perp}\llparenthesis A \rrparenthesis^{O}_{\xi}.\ v_1 \in E_1 \Leftrightarrow v_2 \in E_2$$
$$\mathcal{M}_{\perp}\llparenthesis A^{\ell} \rrparenthesis^{O}_{\xi}(\mu_1,\mu_2) \overset{\text{def}}{=} \begin{cases} \forall(E_1,E_2) \in \mathcal{E}_{\perp}\llparenthesis A \rrparenthesis^{O}_{\xi}.\ \mu_1(E_1) = \mu_2(E_2) & \text{if } \xi\,\ell \sqsubseteq O, \\ \mu_1, \mu_2 \in \mathrm{Meas}\,\llbracket \lfloor A \rfloor \rrbracket_{\perp} & \text{otherwise} \end{cases}$$

$$\llparenthesis \Xi|\mathrm{C} \rrparenthesis(\xi) \overset{\text{def}}{=} \mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\xi) \wedge \varnothing;\varnothing;\mathcal{L} \vdash \xi\,\mathrm{C}$$
$$\llparenthesis \forall[\Xi|\mathrm{C}].\,(\tau_1,\ldots,\tau_n) \to \tau \rrparenthesis^{O}(f_1,f_2) \overset{\text{def}}{=} \forall \xi \in \llparenthesis \Xi|\mathrm{C} \rrparenthesis.\ \forall(v_{11},v_{12}) \in \mathcal{V}\llparenthesis \tau_1 \rrparenthesis^{O}_{\xi}.\ \ldots\ .\ \forall(v_{n1},v_{n2}) \in \mathcal{V}\llparenthesis \tau_n \rrparenthesis^{O}_{\xi}.\ \mathcal{M}_{\perp}\llparenthesis \tau \rrparenthesis^{O}_{\xi}(f_1(v_{11},\ldots,v_{n1}),f_2(v_{12},\ldots,v_{n2}))$$
$$\llparenthesis \Delta \rrparenthesis^{O}(\delta_1,\delta_2) \overset{\text{def}}{=} \forall \mathrm{x} \in \mathrm{dom}(\Delta).\ \mathcal{V}_{\perp}\llparenthesis \Delta(\mathrm{x}) \rrparenthesis^{O}_{\varnothing}(\delta_1(\mathrm{x}),\delta_2(\mathrm{x}))$$
$$\llparenthesis \Psi \rrparenthesis^{O}(\psi_1,\psi_2) \overset{\text{def}}{=} \forall \mathrm{f} \in \mathrm{dom}(\Psi).\ \llparenthesis \Psi(\mathrm{f}) \rrparenthesis^{O}(\psi_1(\mathrm{f}),\psi_2(\mathrm{f}))$$
$$\llparenthesis \Gamma \rrparenthesis^{O}_{\xi}(\gamma_1,\gamma_2) \overset{\text{def}}{=} \forall \mathrm{x} \in \mathrm{dom}(\Gamma).\ \mathcal{V}\llparenthesis \Gamma(\mathrm{x}) \rrparenthesis^{O}_{\xi}(\gamma_1(\mathrm{x}),\gamma_2(\mathrm{x}))$$

$$\Delta;\Xi;\mathrm{C};\Gamma \vDash e_1 \approx e_2 : \tau \overset{\text{def}}{=} \forall O \in \mathcal{L}.\ \forall(\delta_1,\delta_2) \in \llparenthesis \Delta \rrparenthesis^{O}.\ \forall \xi \in \llparenthesis \Xi|\mathrm{C} \rrparenthesis.\ \forall(\gamma_1,\gamma_2) \in \llparenthesis \Gamma \rrparenthesis^{O}_{\xi}.\ \mathcal{V}_{\perp}\llparenthesis \tau \rrparenthesis^{O}_{\xi}(\llbracket e_1 \rrbracket_{\delta_1;\gamma_1},\llbracket e_2 \rrbracket_{\delta_2;\gamma_2})$$
$$\Delta;\Psi;\Xi;\mathrm{C};\Gamma \vDash t_1 \approx t_2 : \tau \overset{\text{def}}{=} \forall O \in \mathcal{L}.\ \forall(\delta_1,\delta_2) \in \llparenthesis \Delta \rrparenthesis^{O}.\ \forall(\psi_1,\psi_2) \in \llparenthesis \Psi \rrparenthesis^{O}.\ \forall \xi \in \llparenthesis \Xi|\mathrm{C} \rrparenthesis.\ \forall(\gamma_1,\gamma_2) \in \llparenthesis \Gamma \rrparenthesis^{O}_{\xi}.\ \mathcal{M}_{\perp}\llparenthesis \tau \rrparenthesis^{O}_{\xi}(\llbracket t_1 \rrbracket_{\delta_1;\psi_1;\gamma_1},\llbracket t_2 \rrbracket_{\delta_2;\psi_2;\gamma_2})$$
$$\Delta;\Psi;\Xi;\mathrm{C};\Gamma \vDash m_1 \approx m_2 : \tau \overset{\text{def}}{=} \forall O \in \mathcal{L}.\ \forall(\delta_1,\delta_2) \in \llparenthesis \Delta \rrparenthesis^{O}.\ \forall(\psi_1,\psi_2) \in \llparenthesis \Psi \rrparenthesis^{O}.\ \forall \xi \in \llparenthesis \Xi|\mathrm{C} \rrparenthesis.\ \forall(\gamma_1,\gamma_2) \in \llparenthesis \Gamma \rrparenthesis^{O}_{\xi}.\ \mathcal{M}_{\perp}\llparenthesis \tau \rrparenthesis^{O}_{\xi}(\llbracket m_1 \rrbracket_{\delta_1;\psi_1;\gamma_1},\llbracket m_2 \rrbracket_{\delta_2;\psi_2;\gamma_2})$$

Figure 5. Semantic information-flow types $\llparenthesis\,\rrparenthesis$ and semantic information-flow typing $\vDash$.

Having defined the value relation at unlabeled types A, we can define the relation $\mathcal{V}\llparenthesis A^{\ell} \rrparenthesis^{O}_{\xi}$ at a labeled type A^{ℓ}, which depends on the labels ℓ and O. If $\xi\,\ell \sqsubseteq O$ ($\xi\,\ell$ means applying the substitution function ξ to the label ℓ), the observer is cleared to see values at label $\xi\,\ell$, so $\mathcal{V}\llparenthesis A^{\ell} \rrparenthesis^{O}_{\xi}$ contains exactly those values related by $\mathcal{V}\llparenthesis A \rrparenthesis^{O}_{\xi}$. Otherwise, $\xi\,\ell \not\sqsubseteq O$ and thus the observer does not have the clearance to see values at $\xi\,\ell$, so $\mathcal{V}\llparenthesis A^{\ell} \rrparenthesis^{O}_{\xi}$ is the full relation $\llbracket \lfloor A \rfloor \rrbracket \times \llbracket \lfloor A \rfloor \rrbracket$.

The measure relation $\mathcal{M}\llparenthesis A^{\ell} \rrparenthesis^{O}_{\xi}$ relates two measures in the semantic domain $\mathrm{Meas}\,\llbracket \lfloor A \rfloor \rrbracket$ if they are indistinguishable to the observer. Recall that for a probabilistic computation, its label ℓ represents the information contained in the measure denoting it. Hence the following two cases:

- If $\xi\,\ell \not\sqsubseteq O$, the observer is not classified at a high enough level to differentiate between two measures at $\xi\,\ell$. So $\mathcal{M}\llparenthesis A^{\ell} \rrparenthesis^{O}_{\xi}$ is the full relation $\mathrm{Meas}\,\llbracket \lfloor A \rfloor \rrbracket \times \mathrm{Meas}\,\llbracket \lfloor A \rfloor \rrbracket$.
- Indistinguishability is subtler to define for the case $\xi\,\ell \sqsubseteq O$. Here, we consider two measures indistinguishable if they agree on related measurable sets. The relation $\mathcal{E}\llparenthesis A \rrparenthesis^{O}_{\xi}$ defines the notion of relatedness for measurable sets. Two measurable sets E_1 and E_2 are related when they are closed to one another under the value relation $\mathcal{V}\llparenthesis A \rrparenthesis^{O}_{\xi}$—that is, $v_1 \in E_1 \Leftrightarrow v_2 \in E_2$ for all $(v_1,v_2) \in \mathcal{V}\llparenthesis A \rrparenthesis^{O}_{\xi}$.

The relations $\mathcal{V}_{\perp}\llparenthesis A \rrparenthesis^{O}_{\xi}$, $\mathcal{V}_{\perp}\llparenthesis \tau \rrparenthesis^{O}_{\xi}$, $\mathcal{E}_{\perp}\llparenthesis A \rrparenthesis^{O}_{\xi}$, and $\mathcal{M}_{\perp}\llparenthesis \tau \rrparenthesis^{O}_{\xi}$ then lift $\mathcal{V}\llparenthesis A \rrparenthesis^{O}_{\xi}$, $\mathcal{V}\llparenthesis \tau \rrparenthesis^{O}_{\xi}$, $\mathcal{E}\llparenthesis A \rrparenthesis^{O}_{\xi}$, and $\mathcal{M}\llparenthesis \tau \rrparenthesis^{O}_{\xi}$ to account for partiality.

Semantic typing. To define semantic typing, we first define the semantic interpretation of contexts. In particular, two substitutions γ_1 and γ_2 are related at context Γ when for every variable in the domain of Γ, γ_1 and γ_2 map them to related values. And similarly for the interpretation of Δ and Ψ.

Semantic typing judgments have the forms $\Delta; \Xi; \mathrm{C}; \Gamma \vdash e_1 \approx e_2 : \tau$, $\Delta; \Psi; \Xi; \mathrm{C}; \Gamma \vdash t_1 \approx t_2 : \tau$, and $\Delta; \Psi; \Xi; \mathrm{C}; \Gamma \vdash m_1 \approx m_2 : \tau$. Intuitively, they are defined to hold when the denotations are related under any observer and any related substitutions.

We prove the fundamental property of logical relations, which states that syntactically well-typed expressions, terms, and commands are semantically well-typed.

Theorem 5.1 (fundamental property).
(1) $\Delta; \Xi; \mathrm{C}; \Gamma \vdash e : \tau$ *implies* $\Delta; \Xi; \mathrm{C}; \Gamma \vDash e \approx e : \tau$.
(2) $\Delta; \Psi; \Xi; \mathrm{C}; \Gamma \vdash t : \tau$ *implies* $\Delta; \Psi; \Xi; \mathrm{C}; \Gamma \vDash t \approx t : \tau$.
(3) $\Delta; \Psi; \Xi; \mathrm{C}; \Gamma \vdash m : \tau$ *implies* $\Delta; \Psi; \Xi; \mathrm{C}; \Gamma \vDash m \approx m : \tau$.

The proof is by induction on the (syntactic) typing derivations, with each case proving that semantic typing is compatible with some syntactic typing rule. As with a typical logical-relations proof, the challenge is in setting up the logical-relations model (think of them as induction hypotheses) and the proof is routine. Of special note about the proof is that it involves showing that two integrals are equal $\int f_1(x)\, \mu_1(\mathrm{d}x) = \int f_2(x)\, \mu_2(\mathrm{d}x)$ when the integrands and the measures are not equal point-wise. Lemma 5.2 is a convenient result [18] that enables proving equivalence using a *coarser* structure than point-wise equality.

Lemma 5.2 (Coarsening). *Let (X, Σ_X) be a measurable space, μ_1, μ_2 be measures on X, and $f_1, f_2 : X \to \mathbb{R}^+$ be measurable functions. Let $R \subseteq \Sigma_X \times \Sigma_X$ be a binary relation on measurable sets. If (1) μ_1 and μ_2 agree on R-related sets, i.e., $\mu_1(E_1) = \mu_2(E_2)$ for all $(E_1, E_2) \in R$, and (2) if f_1 and f_2 have R-related preimages, i.e., $\left(f_1^{-1}(S), f_2^{-1}(S)\right) \in R$ for all $S \in \Sigma_{\mathbb{R}}$, then $\int f_1(x)\, \mu_1(\mathrm{d}x) = \int f_2(x)\, \mu_2(\mathrm{d}x)$.*

The lemma allows proving integrals equal by picking a suitable relation R on measurable sets, for which the relations $\mathcal{E}_\perp \llbracket A \rrbracket_\xi^O$ on measurable sets will fit the bill.

Noninterference. Noninterference follows from the fundamental property. It guarantees that the measure denoting a L-typed term behaves irrespective of the H-labeled variables in the context.

Theorem 5.3 (Noninterference). *Let Γ_{H} be a context that binds only* H*-labeled variables—that is, for all $\mathrm{x} \in dom(\Gamma_{\mathsf{H}})$, there is some A such that $\Gamma_{\mathsf{H}}(\mathrm{x}) = A^{\mathsf{H}}$. Let $f : \llbracket \lfloor \tau \rfloor \rrbracket_\perp \to \mathbb{R}^+$ be a measurable function. If $\vdash \tau : \mathsf{L}$, $\vdash \Gamma_{\mathsf{L}} : \mathsf{L}$, and $\Delta; \Psi; \varnothing; \varnothing; \Gamma_{\mathsf{H}}, \Gamma_{\mathsf{L}} \vdash t : \tau$, then for all $\gamma_1, \gamma_2 \in \llbracket \lfloor \Gamma_{\mathsf{H}} \rfloor \rrbracket$ and $\gamma_{\mathsf{L}} \in \llbracket \lfloor \Gamma_{\mathsf{L}} \rfloor \rrbracket$,*

$$\int f(x)\, \llbracket t \rrbracket_{\gamma_1, \gamma_{\mathsf{L}}}(\mathrm{d}x) = \int f(x)\, \llbracket t \rrbracket_{\gamma_2, \gamma_{\mathsf{L}}}(\mathrm{d}x).$$

Here, $\vdash \tau : \mathsf{L}$ is defined to mean that all labels occurring in τ, including those nested labels, are L. And $\vdash \Gamma : \mathsf{L}$ means that $\vdash \Gamma(\mathrm{z}) : \mathsf{L}$ for all $\mathrm{z} \in \mathrm{dom}(\Gamma)$. It is not a sufficient condition that the *outermost* label of τ and those in Γ are L. For example, the typing $\Psi; \Delta; \varnothing; \varnothing; \mathrm{x} : \mathbb{R}^{\mathsf{H}} \vdash \mathrm{ret}(\langle \mathrm{x}, \mathrm{x} \rangle) : (\mathbb{R}^{\mathsf{H}} \times \mathbb{R}^{\mathsf{H}})^{\mathsf{L}}$ is valid, but it would be absurd if it implied that $\langle \mathrm{x}, \mathrm{x} \rangle$ behaved irrespective of x. Similarly, $\mathrm{x} : \mathbb{R}^{\mathsf{H}}, \mathrm{y} : (\mathbb{R}^{\mathsf{H}} \to \mathbb{R}^{\mathsf{L}})^{\mathsf{L}} \vdash \mathrm{ret}(\mathrm{y}\ \mathrm{x}) : \mathbb{R}^{\mathsf{L}}$ is valid typing, but it would be absurd if it implied that the application of y to x behaved irrespective of x for a γ_{L} such that $\gamma_{\mathsf{L}}(\mathrm{y}) = \lambda x. x \in \llbracket \lfloor (\mathbb{R}^{\mathsf{H}} \to \mathbb{R}^{\mathsf{L}})^{\mathsf{L}} \rfloor \rrbracket$. Theorem 5.3 is a corollary of a more general version of noninterference (given in an appendix [39]) that relaxes the condition $\vdash \Gamma_{\mathsf{L}} : \mathsf{L}$ and allows the integrands on the two sides of the equation to be different.

6 VARIABLE-ELIMINATION TRANSFORMATION

Main idea. To generate a pure program, the transformation must compile away all probabilistic-fragment constructs. In particular, (1) it eliminates random variables (r.v.s) by summation or

$\boxed{\mathbf{T}[\![\mathcal{P}]\!] = \mathcal{P}'}$ *CPS-translate a probabilistic program to a pure program*

$$\frac{\mathbf{K}[\![m]\!]\,(\lambda \mathrm{x}.\,0) = e \qquad \mathbf{T}[\![\mathcal{F}]\!] = \mathcal{G}' \text{ for } \mathcal{F} \in \overline{\mathcal{F}} \qquad \text{def logsumexp} = \lambda \mathrm{x}.\,\log(\exp(\mathrm{x}\ \text{true}) + \exp(\mathrm{x}\ \text{false})) \in \overline{\mathcal{G}}}{\mathbf{T}\left[\!\!\left[\overline{\mathcal{G}};\overline{\mathcal{F}};m\right]\!\!\right] = \overline{\mathcal{G}},\overline{\mathcal{G}'};\varnothing;\text{ret}(e)}$$

$\boxed{\mathbf{T}[\![\mathcal{F}]\!] = \mathcal{G}}$ *CPS-translate a probabilistic global function to a pure one*

$$\frac{\mathbf{K}[\![m]\!]\ \mathrm{k} = e}{\mathbf{T}[\![\text{def f}(\mathrm{x}_1, \ldots, \mathrm{x}_n) = m]\!] = \text{def f} = \lambda \mathrm{k}.\,\lambda \mathrm{x}_1.\,\ldots\,.\,\lambda \mathrm{x}_n.\,e}$$

$\boxed{\mathbf{K}[\![m]\!]\ e_k = e}$ *CPS-translate a command to a pure expression*

$$\frac{\mathbf{D}^{+}[\![\varnothing \vdash \mathrm{x}_1 = t_1;\ \ldots;\ \mathrm{x}_n = t_n;\ \text{factor}(e_k\ \mathrm{x}_n)]\!] = e}{\mathbf{K}[\![\mathrm{x}_1 = t_1;\ \ldots;\ \mathrm{x}_{n-1} = t_{n-1};\ t_n]\!]\ e_k = e}$$

$\boxed{\mathbf{D}^{+}[\![\overline{\mathrm{z}} \vdash m]\!] = e}$ *Accumulate discrete r.v.s into worklist* $\overline{\mathrm{z}}$

$$\frac{\mathbf{D}^{+}[\![\overline{\mathrm{z}}, \mathrm{y} \vdash \overline{\mathrm{x} = t};\ \text{factor}(\text{logPr}_{\mathbb{B}}(d; \mathrm{y}));\ m]\!] = e}{\mathbf{D}^{+}[\![\overline{\mathrm{z}} \vdash \overline{\mathrm{x} = t};\ \mathrm{y} = \text{sample}_{\mathbb{B}}(d);\ m]\!] = e}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ t_i \neq \text{sample}_{\mathbb{B}}(d) \qquad \mathbf{D}^{-}[\![\overline{\mathrm{z}} \vdash \mathrm{x}_1 = t_1;\ \ldots;\ \mathrm{x}_n = t_n;\ t_{n+1}]\!] = e}{\mathbf{D}^{+}[\![\overline{\mathrm{z}} \vdash \mathrm{x}_1 = t_1;\ \ldots;\ \mathrm{x}_n = t_n;\ t_{n+1}]\!] = e}$$

$\boxed{\mathbf{D}^{-}[\![\overline{\mathrm{z}} \vdash m]\!] = e}$ *Eliminate discrete r.v.s in worklist* $\overline{\mathrm{z}}$

$$\frac{\Delta;\Psi;\mathrm{y} : \mathbb{B}^{\mathsf{H}};\Gamma \vdash m \rightsquigarrow_{\Gamma'} m_{\mathsf{H}} * m_{\mathsf{L}} \qquad \vdash \Gamma : \mathsf{L} \qquad \mathbf{C}[\![m_{\mathsf{H}}]\!] = e_{\mathsf{H}} \qquad \mathbf{D}^{-}[\![\overline{\mathrm{z}} \vdash m_{\mathsf{L}};\ \text{factor}(\text{logsumexp}\ (\lambda \mathrm{y}.\,e_{\mathsf{H}}))]\!] = e}{\mathbf{D}^{-}[\![\overline{\mathrm{z}}, \mathrm{y} \vdash m]\!] = e}$$

$$\frac{\mathbf{C}[\![m]\!] = e}{\mathbf{D}^{-}[\![\varnothing \vdash m]\!] = e}$$

$\boxed{\mathbf{C}[\![m]\!] = e}$ *Eliminate probabilistic-level control flow (calls & branching) with CPS*

$$\frac{\Delta;\Psi;\mathrm{y} : A^{\mathsf{H}};\Gamma \vdash m \rightsquigarrow_{\Gamma'} m_{\mathsf{H}} * m_{\mathsf{L}} \qquad \vdash \Gamma : \mathsf{L} \qquad \mathbf{C}[\![m_{\mathsf{H}}]\!] = e_{\mathsf{H}} \qquad \mathbf{C}[\![\overline{\mathrm{x} = t};\ m_{\mathsf{L}};\ \text{factor}(\mathrm{f}\ (\lambda \mathrm{y}.\,e_{\mathsf{H}})\ \overline{e})]\!] = e'}{\mathbf{C}[\![\overline{\mathrm{x} = t};\ \mathrm{y} = \mathrm{f}(\overline{e});\ m]\!] = e'}$$

$$\frac{\Delta;\Psi;\mathrm{y} : A^{\mathsf{H}};\Gamma \vdash m \rightsquigarrow_{\Gamma'} m_{\mathsf{H}} * m_{\mathsf{L}} \qquad \vdash \Gamma : \mathsf{L} \qquad \mathbf{C}[\![m_{\mathsf{H}}]\!] = e_{\mathsf{H}} \qquad \mathbf{K}[\![m_i]\!]\ (\lambda \mathrm{y}.\,e_{\mathsf{H}}) = e_i \text{ for } i \in \{1, 2\} \qquad \mathbf{C}[\![\overline{\mathrm{x} = t};\ m_{\mathsf{L}};\ \text{factor}(\text{case}(e; \mathrm{z}.e_1; \mathrm{z}.e_2))]\!] = e'}{\mathbf{C}[\![\overline{\mathrm{x} = t};\ \mathrm{y} = \text{case}(e; \mathrm{z}.m_1; \mathrm{z}.m_2);\ m]\!] = e'}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ t_i \neq \mathrm{f}(\ldots) \wedge t_i \neq \text{case}(\ldots) \qquad \mathbf{R}[\![\mathrm{x}_1 = t_1;\ \ldots;\ \mathrm{x}_{n-1} = t_{n-1};\ t_n]\!] = e}{\mathbf{C}[\![\mathrm{x}_1 = t_1;\ \ldots;\ \mathrm{x}_{n-1} = t_{n-1};\ t_n]\!] = e}$$

$\boxed{\mathbf{R}[\![m]\!] = e}$ *Transform away any remaining probabilistic-level terms*

$$\mathbf{R}[\![\text{ret}(e)]\!] = \text{let}(e; \_.0) \qquad \frac{\mathbf{R}[\![m]\!] = e'}{\mathbf{R}[\![\mathrm{x} = \text{ret}(e);\ m]\!] = \text{let}(e; \mathrm{x}.e')}$$

$$\mathbf{R}[\![\text{factor}(e)]\!] = e \qquad \frac{\mathbf{R}[\![m\{\text{unit}/\mathrm{x}\}]\!] = e'}{\mathbf{R}[\![\mathrm{x} = \text{factor}(e);\ m]\!] = e + e'}$$

$$\frac{\Delta;\Psi;\mathrm{y} : \mathbb{R}^{\mathsf{H}};\Gamma \vdash m \rightsquigarrow_{\Gamma'} m_{\mathsf{H}} * m_{\mathsf{L}} \qquad \vdash \Gamma : \mathsf{L} \qquad \mathbf{R}[\![m_{\mathsf{L}};\ \text{factor}(\text{logML}(\mathrm{y} = \text{sample}_{\mathbb{R}}(d);\ m_{\mathsf{H}}))]\!] = e}{\mathbf{R}[\![\mathrm{y} = \text{sample}_{\mathbb{R}}(d);\ m]\!] = e}$$

$\boxed{\Delta;\Psi;\Gamma_{\mathsf{H}};\Gamma_{\mathsf{L}} \vdash m \rightsquigarrow_{\Gamma'} m_{\mathsf{H}} * m_{\mathsf{L}}}$ *Factorize m into m_{H} and m_{L} with information-flow typing*

Invariants: $\Delta;\Psi;\varnothing;\varnothing;\Gamma_{\mathsf{H}},\Gamma_{\mathsf{L}},\Gamma' \vdash m_{\mathsf{H}} : \mathbb{U}^{\mathsf{H}}$ *and* $\Delta;\Psi;\varnothing;\varnothing;\Gamma_{\mathsf{L}} \vdash m_{\mathsf{L}} : \mathbb{U}^{\mathsf{L}}$

$$\frac{\Delta;\Psi;\varnothing;\varnothing;\Gamma_{\mathsf{H}},\Gamma_{\mathsf{L}} \vdash t : \mathbb{U}^{\mathsf{L}} \qquad \mathit{Canonicalize}(t, \Gamma_{\mathsf{H}}) = t'}{\Delta;\Psi;\Gamma_{\mathsf{H}};\Gamma_{\mathsf{L}} \vdash t \rightsquigarrow_{\varnothing} \text{ret}(\text{unit}) * t'} \qquad \frac{\Delta;\Psi;\varnothing;\varnothing;\Gamma_{\mathsf{H}},\Gamma_{\mathsf{L}} \vdash t : \mathbb{U}^{\mathsf{H}}}{\Delta;\Psi;\Gamma_{\mathsf{H}};\Gamma_{\mathsf{L}} \vdash t \rightsquigarrow_{\varnothing} t * \text{ret}(\text{unit})}$$

$$\frac{\Delta;\Psi;\varnothing;\varnothing;\Gamma_{\mathsf{H}},\Gamma_{\mathsf{L}} \vdash t : A^{\mathsf{L}} \qquad \vdash A : \mathsf{L} \qquad \mathit{Canonicalize}(t, \Gamma_{\mathsf{H}}) = t' \qquad \Delta;\Psi;\Gamma_{\mathsf{H}};\Gamma_{\mathsf{L}}, \mathrm{x} : A^{\mathsf{L}} \vdash m \rightsquigarrow_{\Gamma'} m_{\mathsf{H}} * m_{\mathsf{L}} \qquad t = \text{sample}_{\mathbb{R}}(d) \Rightarrow \mathrm{x} \notin FV(m_{\mathsf{H}})}{\Delta;\Psi;\Gamma_{\mathsf{H}};\Gamma_{\mathsf{L}} \vdash \mathrm{x} = t;\ m \rightsquigarrow_{\Gamma', \mathrm{x}:A^{\mathsf{L}}} m_{\mathsf{H}} * (\mathrm{x} = t';\ m_{\mathsf{L}})}$$

$$\frac{\Delta;\Psi;\varnothing;\varnothing;\Gamma_{\mathsf{H}},\Gamma_{\mathsf{L}} \vdash t : A^{\ell} \qquad \Delta;\Psi;\Gamma_{\mathsf{H}}, \mathrm{x} : A^{\mathsf{H}};\Gamma_{\mathsf{L}} \vdash m \rightsquigarrow_{\Gamma'} m_{\mathsf{H}} * m_{\mathsf{L}}}{\Delta;\Psi;\Gamma_{\mathsf{H}};\Gamma_{\mathsf{L}} \vdash \mathrm{x} = t;\ m \rightsquigarrow_{\Gamma'} (\mathrm{x} = t;\ m_{\mathsf{H}}) * m_{\mathsf{L}}}$$

Figure 6. Variable-elimination transformation. Typing contexts are omitted for brevity (except in factorization judgments). A version of the transformation with complete context information is given in an appendix.

$$\mathbf{T}\llbracket \mathsf{def}\ \mathrm{hmm}(\mathrm{z}_0, \mathrm{data}) = m \rrbracket \overset{\textcircled{1}}{=} \mathsf{def}\ \mathrm{hmm} = \lambda \mathrm{k}.\ \lambda \mathrm{z}_0.\ \lambda \mathrm{data}.\ \mathbf{K}\llbracket m \rrbracket\ \mathrm{k}$$

$$\mathbf{K}\llbracket m \rrbracket\ \mathrm{k} = \mathbf{K}\llbracket \mathsf{case}\ \mathrm{data}\ \ldots \rrbracket\ \mathrm{k} \overset{\textcircled{2}}{=} \mathbf{D}^{+}\left\llbracket \varnothing \vdash \begin{array}{l} \mathrm{y} = \mathsf{case}\ \mathrm{data}\ \ldots \\ \mathsf{factor}(\mathrm{k}(\mathrm{y})) \end{array} \right\rrbracket \overset{\textcircled{3}}{=} \mathbf{D}^{-}\left\llbracket \varnothing \vdash \begin{array}{l} \mathrm{y} = \mathsf{case}\ \mathrm{data}\ \ldots \\ \mathsf{factor}(\mathrm{k}(\mathrm{y})) \end{array} \right\rrbracket \overset{\textcircled{4}}{=} \mathbf{C}\left\llbracket \begin{array}{l} \mathrm{y} = \mathsf{case}\ \mathrm{data}\ \ldots \\ \mathsf{factor}(\mathrm{k}(\mathrm{y})) \end{array} \right\rrbracket$$

$$\overset{\textcircled{5}}{=} \mathbf{C}\left\llbracket \begin{array}{l} \mathsf{factor}(\mathsf{case}\ \mathrm{data}\ \mathsf{of} \\ \quad \mid \mathsf{nil} \Rightarrow \mathbf{K}\llbracket m_{\mathrm{nil}} \rrbracket\ (\lambda \mathrm{y}.\ \mathbf{C}\llbracket \mathsf{factor}(\mathrm{k}(\mathrm{y})) \rrbracket) \\ \quad \mid \mathsf{cons}\ \mathrm{x}\ \mathrm{xs} \Rightarrow \mathbf{K}\llbracket m_{\mathrm{cons}} \rrbracket\ (\lambda \mathrm{y}.\ \mathbf{C}\llbracket \mathsf{factor}(\mathrm{k}(\mathrm{y})) \rrbracket) \\ \mathsf{end}) \end{array} \right\rrbracket \overset{\textcircled{6}}{=} \begin{array}{l} \mathsf{case}\ \mathrm{data}\ \mathsf{of} \\ \mid \mathsf{nil} \Rightarrow \mathbf{K}\llbracket m_{\mathrm{nil}} \rrbracket\ \mathrm{k} \\ \mid \mathsf{cons}\ \mathrm{x}\ \mathrm{xs} \Rightarrow \mathbf{K}\llbracket m_{\mathrm{cons}} \rrbracket\ \mathrm{k} \\ \mathsf{end} \end{array}$$

$$\mathbf{K}\llbracket m_{\mathrm{nil}} \rrbracket\ \mathrm{k} = \mathbf{K}\llbracket \mathsf{ret}(\mathrm{z}_0) \rrbracket\ \mathrm{k} \overset{\textcircled{7}}{=} \mathbf{D}^{+}\llbracket \varnothing \vdash \mathsf{factor}(\mathrm{k}(\mathrm{z}_0)) \rrbracket \overset{\textcircled{8}}{=} \mathbf{D}^{-}\llbracket \varnothing \vdash \mathsf{factor}(\mathrm{k}(\mathrm{z}_0)) \rrbracket \overset{\textcircled{9}}{=} \mathbf{C}\llbracket \mathsf{factor}(\mathrm{k}(\mathrm{z}_0)) \rrbracket \overset{\textcircled{10}}{=} \mathrm{k}(\mathrm{z}_0)$$

$$\mathbf{K}\llbracket m_{\mathrm{cons}} \rrbracket\ \mathrm{k} = \mathbf{K}\left\llbracket \begin{array}{l} \mathrm{z} = \mathrm{hmm}(\mathrm{z}_0, \mathrm{xs}) \\ \mathsf{observe}(\mathrm{emit}(\mathrm{z}); \mathrm{x}) \\ \mathsf{sample}(\mathrm{step}(\mathrm{z})) \end{array} \right\rrbracket \mathrm{k} \overset{\textcircled{11}}{=} \mathbf{D}^{+}\left\llbracket \varnothing \vdash \begin{array}{l} \mathrm{z} = \mathrm{hmm}(\mathrm{z}_0, \mathrm{xs}) \\ \mathsf{observe}(\mathrm{emit}(\mathrm{z}); \mathrm{x}) \\ \mathrm{y} = \mathsf{sample}(\mathrm{step}(\mathrm{z})) \\ \mathsf{factor}(\mathrm{k}(\mathrm{y})) \end{array} \right\rrbracket \overset{\textcircled{12}}{=} \mathbf{D}^{+}\left\llbracket \mathrm{y} \vdash \begin{array}{l} \mathrm{z} = \mathrm{hmm}(\mathrm{z}_0, \mathrm{xs}) \\ \mathsf{observe}(\mathrm{emit}(\mathrm{z}); \mathrm{x}) \\ \mathsf{factor}(\mathsf{logPr}(\mathrm{step}(\mathrm{z}); \mathrm{y})) \\ \mathsf{factor}(\mathrm{k}(\mathrm{y})) \end{array} \right\rrbracket$$

$$\overset{\textcircled{13}}{=} \mathbf{D}^{-}\left\llbracket \mathrm{y} \vdash \begin{array}{l} \mathrm{z} = \mathrm{hmm}(\mathrm{z}_0, \mathrm{xs}) \\ \mathsf{observe}(\mathrm{emit}(\mathrm{z}); \mathrm{x}) \\ \mathsf{factor}(\mathsf{logPr}(\mathrm{step}(\mathrm{z}); \mathrm{y})) \\ \mathsf{factor}(\mathrm{k}(\mathrm{y})) \end{array} \right\rrbracket \overset{\textcircled{14}}{=} \mathbf{D}^{-}\left\llbracket \varnothing \vdash \begin{array}{l} \mathrm{z} = \mathrm{hmm}(\mathrm{z}_0, \mathrm{xs}) \\ \mathsf{observe}(\mathrm{emit}(\mathrm{z}); \mathrm{x}) \\ \mathsf{factor}(\mathrm{logsumexp}(\lambda \mathrm{y}. \\ \quad \mathbf{C}\left\llbracket \begin{array}{l} \mathsf{factor}(\mathsf{logPr}(\mathrm{step}(\mathrm{z}); \mathrm{y})) \\ \mathsf{factor}(\mathrm{k}(\mathrm{y})) \end{array} \right\rrbracket \\ )) \end{array} \right\rrbracket \overset{\textcircled{15}}{=} \mathbf{C}\left\llbracket \begin{array}{l} \mathrm{z} = \mathrm{hmm}(\mathrm{z}_0, \mathrm{xs}) \\ \mathsf{observe}(\mathrm{emit}(\mathrm{z}); \mathrm{x}) \\ \mathsf{factor}(\mathrm{logsumexp}(\lambda \mathrm{y}. \\ \quad \mathsf{logPr}(\mathrm{step}(\mathrm{z}); \mathrm{y}) + \\ \quad \mathrm{k}(\mathrm{y}) \\ )) \end{array} \right\rrbracket$$

$$\overset{\textcircled{16}}{=} \mathbf{C}\left\llbracket \begin{array}{l} \mathsf{factor}(\mathrm{hmm}(\\ \quad \lambda \mathrm{z}.\ \mathbf{C}\left\llbracket \begin{array}{l} \mathsf{observe}(\mathrm{emit}(\mathrm{z}); \mathrm{x}) \\ \mathsf{factor}(\mathrm{logsumexp}(\lambda \mathrm{y}.\ \mathsf{logPr}(\mathrm{step}(\mathrm{z}); \mathrm{y}) + \mathrm{k}(\mathrm{y}))) \end{array} \right\rrbracket, \\ \mathrm{z}_0, \mathrm{xs})) \end{array} \right\rrbracket \overset{\textcircled{17}}{=} \begin{array}{l} \mathrm{hmm}(\\ \quad \lambda \mathrm{z}.\ \mathsf{logPr}(\mathrm{emit}(\mathrm{z}); \mathrm{x}) + \\ \qquad \mathrm{logsumexp}(\lambda \mathrm{y}.\ \mathsf{logPr}(\mathrm{step}(\mathrm{z}); \mathrm{y}) + \mathrm{k}(\mathrm{y})), \\ \mathrm{z}_0, \mathrm{xs}) \end{array}$$

Figure 7. Compiling the hmm function in Figure 1a to that in Figure 1b. The calculation largely follows the rules in Figure 6. To simplify presentation, we use the equality $\mathbf{C}\llbracket \mathsf{factor}(e_1); \ldots; \mathsf{factor}(e_n) \rrbracket = e_1 + \ldots + e_n$ and standard λ-calculus conversions without detailing the intermediate steps.

integration, creating inference subproblems as a result, and (2) it compiles stochastic control flow to deterministic control flow in continuation-passing style. In both cases, the transformation uses information-flow typing to factorize a command into a H partition and a L partition. The measure denotation of the L partition is guaranteed to be independent of the H-labeled variable—be it bound to a sample, case, or a call term—being eliminated. So the L partition can be factored out of the summation, integration, or continuation indexed by the H-labeled variable.

Program transformation. Figure 6 formalizes the VE transformation as a set of mutually recursive translation functions. The translation is defined for well typed programs (with respect to the base type system in Section 3), so it additionally takes typing contexts as input, but for brevity, we omit them in Figure 6; a version with complete context information can be found in an appendix. As a running example, the step-by-step translation of hmm is shown in Figure 7. We now describe the translation rules in Figure 6, referring to steps in Figure 7 as concrete instantiations of the rules.

$\mathbf{T}\llbracket \mathcal{F} \rrbracket$ translates a probabilistic function $\mathcal{F}$ to a pure function additionally parameterized by a continuation k, with the body m of $\mathcal{F}$ translated as $\mathbf{K}\llbracket m \rrbracket\ \mathrm{k}$ ($\textcircled{1}$).

$\mathbf{K}\llbracket m \rrbracket\ e_k$ translates a command m given a continuation e_k, which is the log-factor dependent on the return value of m. At the top level, the main command m is translated with the top-level continuation $\lambda \mathrm{x}.\ 0$. $\mathbf{K}\llbracket m \rrbracket\ e_k$ works by applying e_k to m's return value, appending the resulting factor to m, and then uses $\mathbf{D}^{+}$ to further translate the resulting $\mathbb{U}$-typed command ($\textcircled{2}\textcircled{7}\textcircled{11}$).

While $\mathbf{K}$ can be applied to any well typed command, going forward, the other translations ($\mathbf{D}^{+}$, $\mathbf{D}^{-}$, $\mathbf{C}$, and $\mathbf{R}$) are only defined on commands of the unit type $\mathbb{U}$, as the commands will already have been CPS-translated.

$\mathbf{D}^+$ does preparatory work for eliminating discrete r.v.s, of which the real work is done by $\mathbf{D}^-$. $\mathbf{D}^+$ accumulates bindings of discrete r.v.s into a worklist, turning all $\mathsf{sample}_{\mathbb{B}}$ terms into factor terms (⑫), so that $\mathbf{D}^-$ can eliminate the discrete r.v.s in the worklist in one go without having to worry about factorization possibly creating unbound references to the variables.

$\mathbf{D}^-[\![\overline{z} \vdash m]\!]$ eliminates from m the discrete r.v.s stored in the worklist $\overline{z}$, one at a time, until the worklist is empty (⑭). Elimination of a variable y from a command m involves factorizing m into m_{H} and m_{L}, using the $\rightsquigarrow$ judgment to be defined shortly. $\mathbf{D}^-$ eliminates y by summing over y the factor $\lambda y.\, e_{\mathsf{H}}$ contributed by m_{H}. Importantly, m_{L} can be left out of this sum, because it is guaranteed that the measure denotation of m_{L} is independent of the H-labeled y. The formalized translation does not make memoization explicit, but solutions to this sum may be memoized, with the memo table indexed by the free (discrete) variables in the sum $\mathrm{logsumexp}\ (\lambda y.\, e_{\mathsf{H}})$. These free variables can be thought of as the Markov blanket [49] of y, conditioned on which all other variables are uncorrelated with y. The order in which the variables are eliminated is left unspecified; it is a well studied, orthogonal problem. It is NP-hard to find the optimal ordering that has an elimination width equal to the tree width [20]. In practice, heuristics (e.g., eliminating variables with fewer neighbors first) are effective in giving good orderings with low elimination widths.

$\mathbf{C}$ eliminates probabilistic-level control flow, namely call and branching terms. To translate a function call $y = f(\overline{e})$, the variable y is labeled H, and the continuation m to the function call is factorized into m_{H} and m_{L}. The factor $\lambda y.\, e_{\mathsf{H}}$, representing the probabilistic effects of m_{H}, is passed to the translated pure function f as the continuation argument. The pure function call $f\ (\lambda y.\, e_{\mathsf{H}})\ \overline{e}$ returns a $\mathbb{R}$-valued factor, which is then appended to the command to be translated further (⑯). Notice that the measure denotation of m_{L} is independent of the return value y. So m_{L} can be left out of the continuation passed to the CPS-translated function call to f.

$\mathbf{C}$ translates a branching term $y = \mathsf{case}(e; x.m_1; x.m_2)$ in a similar manner. It factorizes the probabilistic continuation into two partitions, constructs a pure continuation using the H partition only, and CPS-translates the branches by passing this pure continuation (⑤).

When there is no more control-flow terms to eliminate, $\mathbf{R}$ takes over to eliminate any remaining probabilistic-level terms, namely $\mathsf{sample}_{\mathbb{R}}$ and factor. A $\mathsf{sample}_{\mathbb{R}}$ term is eliminated by integration (i.e., applying logML), which can be any marginal-inference method of choice. Solutions to this integration may be memoized, with the memo table indexed by the free (discrete) variables in the integral. Unlike $\mathsf{sample}_{\mathbb{B}}$ terms, $\mathsf{sample}_{\mathbb{R}}$ terms are not first converted to factor terms before being eliminated, as integration in general requires *sampling* from prior distributions of continuous r.v.s.

Like in $\mathbf{D}^-$ and $\mathbf{C}$, the probabilistic continuation is factorized in $\mathbf{R}$ to allow irrelevant terms to be left out of the integration. Unlike in $\mathbf{D}^-$ or $\mathbf{C}$, continuous r.v.s are eliminated not always one at a time, but likely simultaneously, to avoid creating unnecessary nested integrals. For instance, the command $m \overset{\text{def}}{=} x = \mathsf{sample}(\textsc{Normal}(0,1));\ y = \mathsf{sample}(\textsc{Normal}(3,1));\ \mathsf{factor}(\mathsf{logPr}(\textsc{Normal}(x^2 + y^2, 1); 6))$ is translated as $\mathbf{R}[\![m]\!] = \mathbf{R}[\![\mathsf{factor}(\mathsf{logML}(m))]\!] = \mathsf{logML}(m)$. Factorization makes sure that all three terms belong to the same H partition, despite that information-flow typing does not demand so, so that x and y can be marginalized out simultaneously.

Command factorization via information-flow typing. The judgment $\Delta; \Psi; \Gamma_{\mathsf{H}}; \Gamma_{\mathsf{L}} \vdash m \rightsquigarrow_{\Gamma'} m_{\mathsf{H}} * m_{\mathsf{L}}$ factorizes a command m into m_{H} and m_{L}. Here, Γ' is the variable bindings declared in m_{L} and available for use in m_{H}, such that $\Delta; \Psi; \varnothing; \varnothing; \Gamma_{\mathsf{L}} \vdash m_{\mathsf{L}} : \mathbb{U}^{\mathsf{L}}$ and $\Delta; \Psi; \varnothing; \varnothing; \Gamma_{\mathsf{H}}, \Gamma_{\mathsf{L}}, \Gamma' \vdash m_{\mathsf{H}} : \mathbb{U}^{\mathsf{H}}$. In broad strokes, factorization works by partitioning all H-labeled terms in the command to m_{H} and all L-labeled terms to m_{L} (⑤⑭⑯). Label inference is implemented by solving unification constraints.

Step ⑭ performs factorization to eliminate y. With y labeled H, the two factor terms must be typed at H, while the other two terms can be typed at L and left out of the sum over y. Step ⑯ performs factorization to eliminate the recursive call $z = \mathrm{hmm}(z_0, xs)$. With z labeled H, both terms in the continuation must be typed at H, so their translation is passed to the pure hmm as the continuation argument. Step ⑤ performs factorization to eliminate y = case data ..., which is similar.

Because noninterference guarantees that a L-labeled term behaves irrespective of the value of the H-labeled variables in Γ_{H}, factorization additionally *canonicalizes* a L-labeled term: it replaces the H-labeled variables with default values. This substitution ensures that m_{L} does not refer to variables bound in Γ_{H} and thus is well typed under the context Γ_{L}. Finally, factorization specially treats $\mathsf{sample}_{\mathbb{R}}$ terms: it partitions x = $\mathsf{sample}_{\mathbb{R}}(d)$ into m_{L} only if x is not needed in the m_{H}, thus scoping the integral over x to one partition only and avoiding unnecessary nested integrals.

Lemma 6.1. *Let Γ_{H} be a context that binds only* H*-labeled variables and Γ_{L} be a context such that* $\vdash \Gamma_{\mathsf{L}} : \mathsf{L}$. *If $\Delta; \Psi; \Gamma_{\mathsf{H}}; \Gamma_{\mathsf{L}} \vdash m \rightsquigarrow_{\Gamma'} m_{\mathsf{H}} * m_{\mathsf{L}}$, then for any $\gamma_{\mathsf{H}} \in [\![\lfloor \Gamma_{\mathsf{H}} \rfloor]\!]$ and $\gamma_{\mathsf{L}} \in [\![\lfloor \Gamma_{\mathsf{L}} \rfloor]\!]$, it holds that*

$$[\![m]\!]_{\gamma_{\mathsf{H}},\gamma_{\mathsf{L}}}([\![\mathbb{U}]\!]) = \oint_{[\![\lfloor \Gamma' \rfloor]\!]} [\![m_{\mathsf{H}}]\!]_{\gamma_{\mathsf{H}},\gamma_{\mathsf{L}},\gamma'}([\![\mathbb{U}]\!]) \, [\![m_{\mathsf{L}}]\!]_{\gamma_{\mathsf{L}}}(\mathrm{d}\gamma').$$

Lemma 6.1 assures that factorizing a command m produces two partitions that together preserve the semantics of m. The notation $\oint_{[\![\lfloor \Gamma' \rfloor]\!]} f(\gamma') \, [\![m_{\mathsf{L}}]\!]_{\gamma_{\mathsf{L}}}(\mathrm{d}\gamma')$, defined in an appendix with the proof, is a shorthand for the multiple integral with respect to the measures each denoting an $\mathrm{x}_i = t_i$ in m_{L}. The lemma is a consequence of noninterference (Theorem 5.3).

Correctness. We prove that the variable-elimination transformation is correct. The theorem states that the transformed pure expression, when it terminates, computes the log model evidence of the original probabilistic program. As expected, the proof depends on Lemma 6.1.

Theorem 6.2. *Let $\overline{\mathcal{G}}; \overline{\mathcal{F}}; m$ be a well typed program where the main command m has type τ. If* $\mathbf{T}\left[\!\left[\overline{\mathcal{G}}; \overline{\mathcal{F}}; m\right]\!\right] = \overline{\mathcal{G}}, \overline{\mathcal{G}'}; \varnothing; \mathsf{ret}(e)$ *and* $[\![e]\!]_{\varnothing} \neq \bot$, *then* $\log [\![m]\!]_{\varnothing}([\![\tau]\!]) = [\![e]\!]_{\varnothing}$.

7 EXPERIMENTAL EVALUATION

Scalability of VE compilation. We compare Mappl and SlicStan [29]. Stan [13], while a popular PPL, does not support discrete parameters. In response, SlicStan features a state-of-the-art VE compiler that performs information-flow analysis and emits variable-eliminated Stan code. As a benchmark, we consider a simple HMM, for which both compilers can generate code whose running time scales linearly with the length of the observed sequence. But compilation time differs.

Figure 8 shows how compilation time scales as the size of the inference problem increases. (All experiments in Section 7 were run on a server with a 3.6GHz CPU and 12GB of RAM.) In SlicStan, models such as HMMs are expressed by unrolling recursion into a fixed number of iterations, so it is expected that compilation time increases as the size of the inference problem increases. Figure 8 confirms this behavior and further shows that SlicStan struggles with large problem sizes: compiling the model conditioned on generating a sequence of length 60 takes over 30 minutes. In contrast, because Mappl can express the HMM as a recursive program, the compilation time is constant with respect to the problem size.

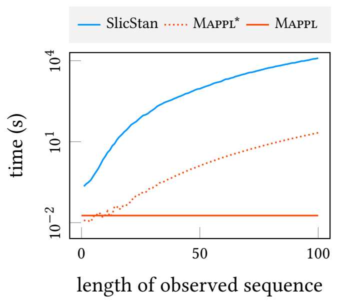


Figure 8. Scaling of compilation time.

We also report the time Mappl takes to compile a version of the HMM with recursion unrolled. Figure 8 (Mappl*) suggests that the Mappl compiler exhibits better scalability than SlicStan on the same unrolled model. A probable reason for this speedup is that Mappl uses a simple two-level lattice in the information-flow analysis, whereas SlicStan uses a meet semilattice, which, as discussed by Gorinova et al. [29], hinders efficient constraint solving.

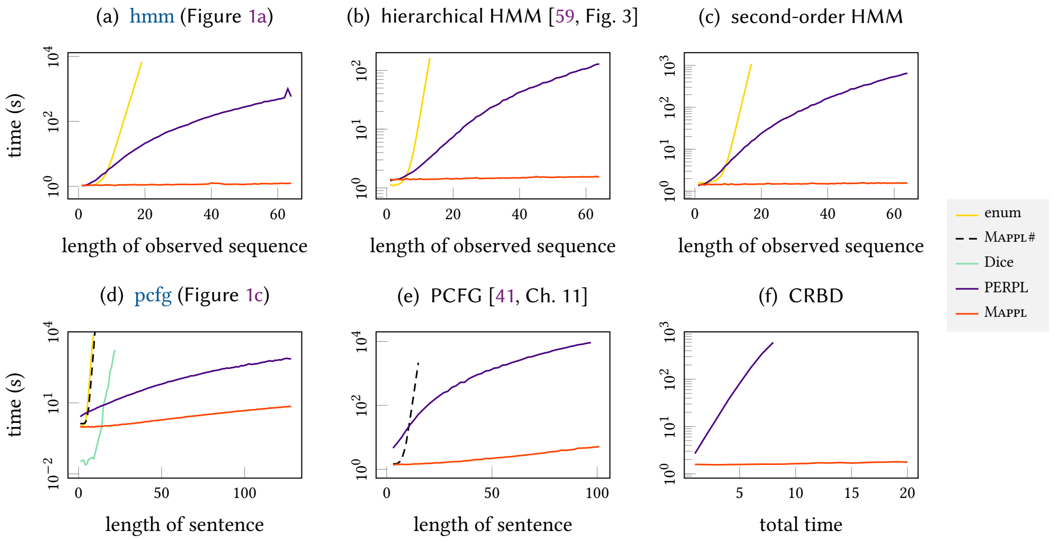


Figure 9. Scaling plots comparing exact-inference methods on recursive programs.

Scalability of exact inference: HMMs, PCFGs, and CRBD. We compare Mappl and PERPL [15] on recursive programs. PERPL represents a state-of-the-art approach to exact inference for recursive programs, compiling them to factor graph grammars [16] and then to systems of equations.

The benchmarks are the HMM in Figure 1a, a hierarchical HMM, a second-order HMM, the PCFG in Figure 1c, a PCFG with 6 nonterminals and 12 productions, and a discrete-time phylogenetic model. The phylogenetic model generates phylogenetic trees under the constant-rate birth–death (CRBD) assumption. This CRBD model is similar to the PCFGs in that it uses recursion (as opposed to iteration) and exhibits stochastic control flow [58].

Figure 9 shows how the inference running time scales as the size of the inference problem increases. Compilation time is not measured, as it does not vary with the problem size for either Mappl or PERPL. As PERPL uses a Python back end, to allow a fair comparison, compiled Mappl programs are further compiled to Python. We use enumeration-based exact inference implemented in Pyro [10] as an additional baseline on some benchmarks; it leads to exponentially increasing running time and runs out of memory quickly on all benchmarks.

On the two HMMs, PERPL leads to running time superlinear in the problem size, whereas Mappl recovers the linear-time forward algorithm for HMMs. For an observed sequence of length 30, PERPL inference takes over 1 minute, while Mappl inference takes 1.5 seconds. On the two PCFGs and the CRBD model, PERPL also scales less favorably than Mappl. We note that PERPL supports unbounded recursion and thus allows the PCFG and CRBD models to be specified in a more declarative way. For example, the CRBD model in PERPL, though complicated by PERPL's linearity restriction, uses an almost surely terminating function to generate the waiting time until the next speciation or extinction event, whereas the Mappl version uses a geometric distribution truncated at the remaining time steps to ensure termination.

We also assess, with the PCFGs, the performance implications of the information-flow analysis. Specifically, we evaluate the performance of a version of Mappl with command factorization disabled (Mappl#). Disabling factorization means that the VE compilation has to assume correlation between the subparses of a nonterminal, thereby hindering the discovery of recurring substructure

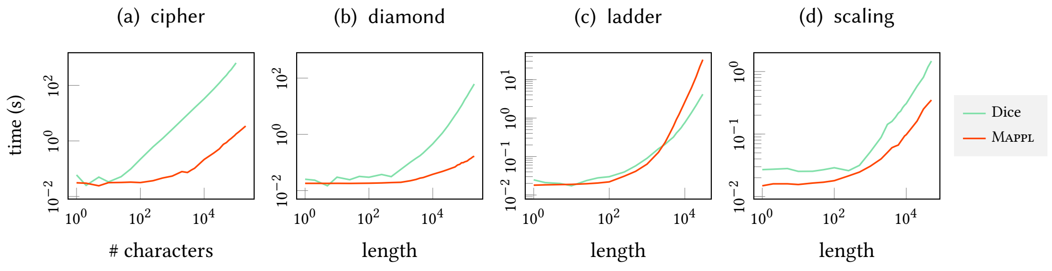


Figure 10. Scaling plots comparing Dice and Mappl on benchmarks from Holtzen et al. [31, Fig. 10].

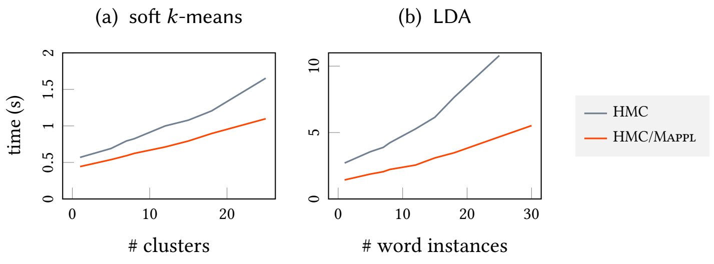


Figure 11. Ahead-of-time VE compilation using Mappl speeds up HMC in NumPyro. Time per chain is reported.

amenable to dynamic programming. Figures 9d and 9e confirm that without factorization, VE-based inference is intractable.

Scalability of exact inference: Dice benchmarks. We compare Mappl and Dice [31]. Dice is a state-of-the-art approach to exact inference for discrete, nonrecursive programs. It casts inference to weighted model counting (WMC) on binary decision diagrams (BDDs), exploiting independence structure in programs to create compact BDDs for factorized inference. We use benchmarks [31, Fig. 10] on which Dice has been shown to demonstrate superior scalability over other PPLs that support exact inference. As Dice uses a C library for WMC on BDDs, to allow a fair comparison, compiled Mappl programs are further compiled to Rust. This Rust back end of Mappl is not yet full-featured but is sufficient for these Dice benchmarks.

Figure 10 shows how the running time scales as the problem size increases. Given that the Dice running time reported by Holtzen et al. [31] includes the time required for BDD generation, the Mappl running time reported here includes that for VE compilation. Mappl is competitive with Dice on these scaling benchmarks, in fact outperforming Dice in three out of four cases. A possible explanation is that since Mappl can express these benchmarks as recursive programs, compilation time does not increase with the problem size.

We also run Dice on a PCFG. Dice does not support recursion, so we follow the recipe of Chiang et al. [15, App. E] in expressing a PCFG in Dice by manually unfolding a loop that generates a parse from subparses. Figure 9d suggests that PCFGs in this encoding are intractable for Dice.

These benchmarks all contain conditional independence structure as a result of function abstractions. While Mappl may do better on such programs, we note that Dice performs better on large Bayesian networks (BN). For example, for a BN with ~40 nodes, Dice solves the inference problem within 30 ms, while Mappl takes over 1 s. We conjecture that this is due to the known result that WMC can significantly outperform VE when models contain substantial *local structure* [14].

Approximate inference: Hamiltonian Monte Carlo (HMC). HMC [23] is a powerful sampling method for differentiable models. Discrete latent variables introduce nondifferentiability, posing

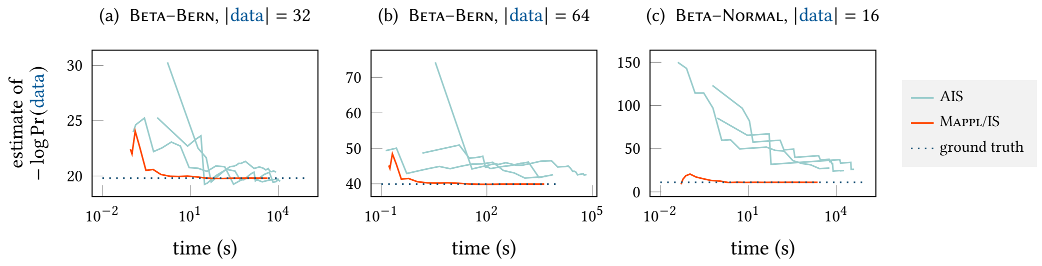


Figure 12. Performance of ML estimation on a family of hybrid discrete–continuous HMM models (Figure 1e), measured by how the negative log ML estimate changes as allowable inference time increases.

challenges to applying HMC to hybrid discrete–continuous models. We consider two such models: a soft k-means clustering model and a latent Dirichlet allocation (LDA) model. One way to handle them is by marginalizing out the discrete variables using VE. For example, Pyro's support for HMC can handle these models by performing VE on plated factor graphs [48].

Pyro performs VE at run time. So we examine whether the performance of Pyro's HMC can be improved by ahead-of-time VE compilation through Mappl. Specifically, we use NumPyro [54], which supports fast HMC inference on top of JAX [24]. We run NumPyro's HMC on the original model and on the Mappl-compiled model with necessary syntax adjustments applied (including replacing the top-level logML with an invocation of NumPyro's HMC).

Figure 11 displays the running time of sampling a single chain consisting of 10,000 samples and 2,500 burn-in samples using the No-U-Turn sampler [32], while varying the number of discrete latent variables in the models. Time is measured after JIT is warmed up. As expected, ahead-of-time VE compilation leads to improved run-time performance.

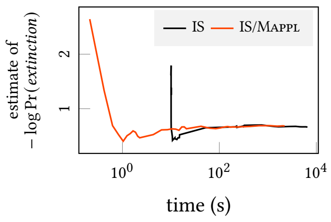
Approximate inference: marginal-likelihood estimation. We examine the performance implications of Mappl's VE compilation for marginal likelihood (ML) estimation, a key task in Bayesian learning and inference. We use a family of hybrid discrete–continuous HMMs (hmm′ in Figure 1e) as benchmarks. For the VE-compiled programs, we use importance sampling (IS) with Pyro to solve the inference subproblems (i.e., nested integrals). As a baseline, we use annealed importance sampling (AIS) [47] to solve the global inference problems directly. AIS, generalizing IS, is a widely used sampling method for ML estimation. We assess the convergence rate of the ML estimate as allowable running time increases. Running time roughly translates to the number of importance samples. We experiment with multiple hyperparameter settings for AIS.

Figures 12a and 12b show the results for an HMM with input data of length 32 and 64, respectively. Mappl/IS converges within tens of seconds in both cases. In contrast, AIS either takes thousands of seconds to converge or does not show signs of convergence even after thousands of seconds, for each hyperparameter setting tested. The quick convergence with Mappl/IS is a consequence of the VE compilation eliminating discrete r.v.s and generating single-dimensional subproblems easily solvable by a Monte-Carlo method.

In fact, the compiled program reveals that the inference problems of this experiment have exact solutions: we are able to use Mathematica to obtain closed-form solutions to the generated subproblem integrals. The dotted lines in Figures 12a and 12b represent the exact solutions to the global problems as computed from those to the subproblems. Our approach enables potentially harnessing the power of symbolic-integration engines by translating a recursive program into subproblems that have readily available closed-form solutions.

We also consider a similar HMM where the subproblems in the VE-compiled program are not known to have closed-form solutions—though they can be approximated with arbitrary precision using numerical methods. Figure 12c shows that Mappl/IS quickly converges to this approximate solution (the dotted line), whereas AIS fails to converge for the relatively small problem size of 16.

VE does not always lead to improved performance, however. As another benchmark, we consider the aforementioned CRBD model extended with two continuous latent variables for the birth and death rates. The plot to the right shows that IS with the VE-compiled program does not converge faster than IS with the original program. The VE-compiled program consists in a top-level integral that nests an exact-inference problem of the kind addressed in Figure 9f, so each importance sample requires solving a new exact-inference problem and is thus more expensive than an importance sample of the original program.



8 RELATED WORK AND DISCUSSION

VE for probabilistic programs. VE is supported by many PPLs. Here we focus the discussion on those on the more expressive end of the spectrum. Early versions of IBAL support VE for programs with unbounded recursion and use lazy evaluation [51]. This approach allows models such as PCFGs to be specified more declaratively, but it seems to have been abandoned in favor of bounded recursion for correctness and efficiency concerns [50] in later versions of IBAL [52] and Figaro [53].

PERPL [15] supports exact inference for programs with unbounded recursion, by compiling them to monotone systems of polynomial equations. Infinite data types, such as integers and strings, pose challenges to equation solving, as they would lead to infinite systems of equations. In response, PERPL uses whole-program transformations (de- and re-functionalization) to eliminate infinite data types. These transformations further necessitate a linear type system to ensure correctness.

Mappl shares the restriction of bounded recursion with a few other PPLs that support VE on PCFG-like models. Bounded recursion, while unable to express PCFGs as almost surely terminating programs, is expressive enough for Bayesian-inference queries on these models (see Figure 1c and another encoding given in an appendix), as the observed data is finite. Koller et al. [34] call such queries *evidence-finite*. We consider our choice to restrict attention to bounded recursion a sweet spot in the design space: it aligns well with the evidence-finite nature of many Bayesian inference problems, does not require the programmer to reason about linearity, leads to provably correct VE-compiled code with performance matching the best known PTIME algorithms, and still allows reasonably concise, readable programs.

SlicStan [29] supports VE for an imperative PPL where programs have deterministic support and variables are global. It supports loops but not recursion, and HMMs expressed via loops do not seem to type-check in SlicStan's information-flow type system. Mappl, in contrast, is a functional PPL with a wider range of features. The denotational treatment is compositional by nature: it enables a noninterference result on open terms, crucial for eliminating variables in subterms under binders. As Section 7 shows, Mappl's support for recursion, as well as its use of a two-level lattice rather than a meet semilattice, avoids the limitation of the SlicStan compiler in scaling to large models.

Solving probabilistic inference problems analytically. Exact, analytical solutions to inference problems are welcome whenever computationally efficient. Some PPLs support exact inference for nonrecursive programs with no or very restricted form of continuous variables, by compiling them into finite graph representations for efficient inference [12, 17, 31, 59]. FSPN [63] and PERPL [15] support exact inference for recursive programs, though they are known to work only for programs with discrete variables. Hakaru [46, 65] and Psi [25, 26] enable exact inference for programs with continuous variables (though still omitting recursion) using computer-algebra solvers.

Delayed sampling in Birch [43] and ProbZelus [8, 3] allow partial analytical solutions to subprograms by exploiting conjugacy. The similarity to our approach is that both are forms of automatic Rao–Blackwellization [57, §4.2] that analytically reduce an inference problem to a better-behaved one. The distinctions are that delayed sampling derives closed-form posteriors for conjugate priors whereas our approach compiles away discrete variables, that delayed sampling is an *inference-time* approach based on dynamic dependence graphs whereas ours is a *compile-time* transformation, and that delayed sampling is not known to work with recursion.

In practice, no single inference technology is likely to excel at all problems; our approach and existing inference methods are complementary. Identifying independence is generally useful for compile-time Rao–Blackwellization; for example, our information-flow analysis can potentially be used in gradient-based methods to reduce variance for gradient estimators. On the other hand, our approach can potentially capitalize on advances in symbolic integration to solve generated subproblems analytically.

Reasoning about independence in probabilistic programs. Verifying randomized algorithms may require reasoning about independence, for which program logics have been developed [7, 6, 40]. While these program logics enable calculational, largely manual proofs of functional correctness, an information-flow type analysis is more amenable to automation through type inference. Hur et al. [33] and Amtoft and Banerjee [2] study program slicing for probabilistic while-languages. Their reasoning is concerned with determining if two variables are correlated conditioned on the observe statements in a program. Conditional independence can sometimes be determined syntactically for Bayesian networks through the notion of active trails [49]. The idea has been adapted to probabilistic programs [33, 37], though a full soundness result is lacking.

Semantics of probabilistic programs. As a Cartesian-closed alternative to measurable spaces, QBSes are introduced to handle higher-order types [30, 62]. It is further shown that QBSes can be equipped with compatible ω-cpo structures to handle term-level recursion and recursive types [64]. Mappl's denotational semantics uses these constructions. Another way to give semantics to PPLs is by first defining a deterministic operational semantics indexed by a randomness source and then integrating over randomness to obtain a measure semantics [11]. Prior work constructs logical relations for program equivalence in this operational setting [18, 66, 69], while we construct logical relations for noninterference in a denotational setting.

9 CONCLUSION

Our approach to variable elimination and marginal inference, presented in the context of Mappl, represents a generalization and synthesis of several important ideas.

- A compiler eliminates probabilistic effects, generalizing variable elimination from graphical models to a richly expressive PPL with recursion.
- It decomposes a global inference problem into subproblems, recovering and generalizing widely used dynamic-programming algorithms for recursive models.
- It factorizes computations into independent partitions, repurposing information-flow typing to probabilistic programs.
- Its correctness result relies on a logical-relations argument, adapting semantic models for noninterference to a measure-theoretic setting.

The payoff is that Mappl allows useful recursive models to be expressed in a functional, recursive style, while enabling sound, scalable inference for a broad class of these programs. Future work could explore ways to enable programmer control over the decomposition into inference subproblems and to exploit local structure in certain models to further speed up inference.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We thank Oghenevwogaga Ebresafe, Edward Lee, and Cong Ma for discussions and help. This work was supported by the Natural Sciences and Engineering Research Council of Canada. The views and opinions expressed are those of the authors and do not necessarily reflect the position of any funding agency.

DATA AVAILABILITY STATEMENT

The Mappl compiler implementation is available [38].

REFERENCES

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A core calculus of dependency. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL).* https://doi.org/10.1145/292540.292555

[2] Torben Amtoft and Anindya Banerjee. 2020. A theory of slicing for imperative probabilistic programs. *ACM Tran. on Programming Languages and Systems (TOPLAS)* 42, 2 (April 2020). https://doi.org/10.1145/3372895

[3] Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022. Semi-symbolic inference for efficient streaming probabilistic programming. *Proc. of the ACM on Programming Languages (PACMPL)* 6, OOPSLA2 (Oct. 2022). https://doi.org/10.1145/3563347

[4] Martin Avanzini, Gilles Barthe, and Ugo Dal Lago. 2021. On continuation-passing transformations and expected cost analysis. *Proc. of the ACM on Programming Languages (PACMPL)* 5, ICFP (Aug. 2021). https://doi.org/10.1145/3473592

[5] J. K. Baker. 1979. Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America* 65, S1 (1979). https://doi.org/10.1121/1.2017061

[6] Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. 2021. A bunched logic for conditional independence. In *ACM/IEEE Symp. on Logic In Computer Science (LICS).* https://doi.org/10.1109/LICS52264.2021.9470712

[7] Gilles Barthe, Justin Hsu, and Kevin Liao. 2019. A probabilistic separation logic. *Proc. of the ACM on Programming Languages (PACMPL)* 4, POPL (Dec. 2019). https://doi.org/10.1145/3371123

[8] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive probabilistic programming. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3385412.3386009

[9] Richard E. Bellman. 1957. *Dynamic Programming.* Princeton University Press.

[10] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research (JMLR)* 20, 1 (2019). arXiv:1810.09538

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

[12] Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. 2011. Measure transformer semantics for Bayesian machine learning. In *European Symp. on Programming (ESOP).* https://doi.org/10.1007/978-3-642-19718-5_5

[13] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017). https://doi.org/10.18637/jss.v076.i01

[14] Mark Chavira and Adnan Darwiche. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence* 172, 6 (2008). https://doi.org/10.1016/j.artint.2007.11.002

[15] David Chiang, Colin McDonald, and Chung-chieh Shan. 2023. Exact recursive probabilistic programming. *Proc. of the ACM on Programming Languages (PACMPL)* 7, OOPSLA1 (April 2023). https://doi.org/10.1145/3586050 arXiv:2210.01206

[16] David Chiang and Darcey Riley. 2020. Factor graph grammars. In *Conf. on Neural Information Processing Systems (NeurIPS).* arXiv:2010.12048

[17] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. In *Proc. of the 9th Joint Meeting of the European Software Engineering Conf. and the*

ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE). https://doi.org/10.1145/2491411.2491423

[18] Ryan Culpepper and Andrew Cobb. 2017. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In *European Symp. on Programming (ESOP).* https://doi.org/10.1007/978-3-662-54434-1_14

[19] Oliver Danvy and Andrzex Filinski. 1992. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science* 2, 4 (1992). https://doi.org/10.1017/S0960129500001535

[20] Adnan Darwiche. 2009. *Modeling and Reasoning with Bayesian Networks.* Cambridge University Press. https://doi.org/10.1017/CBO9780511811357

[21] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Comm. of the ACM (CACM)* 20, 7 (July 1977). https://doi.org/10.1145/359636.359712

[22] Dorothy Elizabeth Robling Denning. 1982. *Cryptography and Data Security.* Addison-Wesley Reading.

[23] Simon Duane, A.D. Kennedy, Brian J. Pendleton, and Duncan Roweth. 1987. Hybrid Monte Carlo. *Physics Letters B* 195, 2 (1987). https://doi.org/10.1016/0370-2693(87)91197-X

[24] Roy Frostig, Matthew Johnson, and Chris Leary. 2018. Compiling machine learning programs via highllevel tracing. https://mlsys.org/Conferences/doc/2018/146.pdf

[25] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *Int'l Conf. on Computer Aided Verification (CAV).* https://doi.org/10.1007/978-3-319-41528-4_4

[26] Timon Gehr, Samuel Steffen, and Martin Vechev. 2020. λPSI: Exact inference for higher-order probabilistic programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3385412.3386006

[27] J. A. Goguen and J. Meseguer. 1982. Security policies and security models. In *IEEE Symp. on Security and Privacy.* https://doi.org/10.1109/SP.1982.10014

[28] Noah D. Goodman and Andreas Stuhlmüller. 2014. The design and implementation of probabilistic programming languages. http://dippl.org.

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

[30] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *ACM/IEEE Symp. on Logic In Computer Science (LICS).* https://doi.org/10.5555/3329995.3330072 arXiv:1701.02547

[31] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proc. of the ACM on Programming Languages (PACMPL)* 4, OOPSLA (Nov. 2020). https://doi.org/10.1145/3428208 arXiv:2005.09089

[32] Matthew D. Homan and Andrew Gelman. 2014. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research (JMLR)* 15, 1 (Jan. 2014). https://doi.org/10.5555/2627435.2638586

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

[34] Daphne Koller, David McAllester, and Avi Pfeffer. 1997. Effective Bayesian inference for stochastic programs. In *Proc. of the 14th National Conf. on Artificial Intelligence and the 9thConf. on Innovative Applications of Artificial Intelligence (AAAI/IAAI).*

[35] Satoshi Kura. 2023. Higher-order weakest precondition transformers via a CPS transformation. (2023). arXiv:2301.09997

[36] Alexander K. Lew, Matin Ghavamizadeh, Martin C. Rinard, and Vikash K. Mansinghka. 2023. Probabilistic programming with stochastic probabilities. *Proc. of the ACM on Programming Languages (PACMPL)* 7, PLDI (June 2023). https://doi.org/10.1145/3591290

[37] Jianlin Li, Leni Ven, Pengyuan Shi, and Yizhou Zhang. 2023. Type-preserving, dependence-aware guide generation for sound, effective amortized probabilistic inference. *Proc. of the ACM on Programming Languages (PACMPL)* POPL. https://doi.org/10.1145/3571243

[38] Jianlin Li, Eric Wang, and Yizhou Zhang. 2024. Compiling probabilistic programs for variable elimination with information flow (artifact). https://doi.org/10.5281/zenodo.10951893

[39] Jianlin Li, Eric Wang, and Yizhou Zhang. 2024. *Compiling Probabilistic Programs for Variable Elimination with Information Flow (Extended Version).* Technical Report CS-2024-03. School of Computer Science, University of Waterloo. https://cs.uwaterloo.ca/sites/ca.computer-science/files/uploads/files/cs-2024-03.pdf

[40] John M. Li, Amal Ahmed, and Steven Holtzen. 2023. Lilac: A modal separation logic for conditional probability. *Proc. of the ACM on Programming Languages (PACMPL)* 7, PLDI (June 2023). https://doi.org/10.1145/3591226 arXiv:2304.01339

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

[42] Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. FACTORIE: Probabilistic programming via imperatively defined factor graphs. In *Conf. on Neural Information Processing Systems (NIPS).*

[43] Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas Schön. 2018. Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. In *Int'l Conf. on Artificial Intelligence and Statistics (AISTATS).* arXiv:1708.07787

[44] Andrew C. Myers. 1999. JFlow: Practical mostly-static information flow control. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL).* https://doi.org/10.1145/292540.292561

[45] Andrew C. Myers and Barbara Liskov. 1997. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP).* https://doi.org/10.1145/268998.266669

[46] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *Functional and Logic Programming.* https://doi.org/10.1007/978-3-319-29604-3_5

[47] Radford M. Neal. 2001. Annealed importance sampling. *Statistics and Computing* 11 (2001). https://doi.org/10.1023/A:1008923215028

[48] Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Neeraj Pradhan, Justin Chiu, Alexander Rush, and Noah Goodman. 2019. Tensor variable elimination for plated factor graphs. In *Int'l Conf. on Machine Learning (ICML).* arXiv:1902.03210

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

[50] Avi Pfeffer. [n.d.]. IBAL: An expressive, functional probabilistic modeling language. ([n. d.]). https://www.cs.tufts.edu/~nr/cs257/archive/avi-pfeffer/ibal-journal.pdf

[51] Avi Pfeffer. 2001. IBAL: A probabilistic rational programming language. In *Int'l Joint Conf. on Artificial Intelligence (IJCAI).*

[52] Avi Pfeffer. 2007. The design and implementation of IBAL: A general-purpose probabilistic language. In *Introduction to Statistical Relational Learning.* The MIT Press. https://doi.org/10.7551/mitpress/7432.003.0016

[53] Avi Pfeffer. 2016. *Practical Probabilistic Programming.* Manning Publications.

[54] Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable effects for flexible and accelerated probabilistic programming in NumPyro. (2019). arXiv:1912.11554

[55] Lawrence R. Rabiner and Biing-Hwang Juang. 1986. An introduction to hidden Markov models. *IEEE ASSP Magazine* 3, 1 (1986). https://doi.org/10.1109/MASSP.1986.1165342

[56] Vineet Rajani and Deepak Garg. 2018. Types for information flow control: Labeling granularity and semantic models. In *IEEE Computer Security Foundations Symp. (CSF).* https://doi.org/10.1109/CSF.2018.00024

[57] Christian P. Robert and George Casella. 1999. *Monte Carlo Statistical Methods.* Springer. https://doi.org/10.1007/978-1-4757-4145-2

[58] Fredrik Ronquist, Jan Kudlicka, Viktor Senderov, Johannes Borgström, Nicolas Lartillot, Daniel Lundén, Lawrence Murray, Thomas B. Schön, and David Broman. 2021. Universal probabilistic programming offers a powerful approach to statistical phylogenetics. *Communications Biology* 4, 1 (2021). https://doi.org/10.1038/s42003-021-01753-7

[59] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: Probabilistic programming with fast exact symbolic inference. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/3453483.3454078

[60] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003). https://doi.org/10.1109/JSAC.2002.806121

[61] Andrei Sabelfeld and David Sands. 2001. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation* 14 (2001). https://doi.org/10.1023/A:1011553200337

[62] Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2017. Denotational validation of higher-order Bayesian inference. *Proc. of the ACM on Programming Languages (PACMPL)* 2, POPL (Dec. 2017). https://doi.org/10.1145/3158148

[63] Andreas Stuhlmüller and Noah D. Goodman. 2012. A dynamic programming algorithm for inference in recursive probabilistic programs. (2012). arXiv:1206.3555

[64] Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. *Proc. of the ACM on Programming Languages (PACMPL)* 3, POPL (2019). https://doi.org/10.1145/3290349

[65] Rajan Walia, Praveen Narayanan, Jacques Carette, Sam Tobin-Hochstadt, and Chung-chieh Shan. 2019. From high-level inference algorithms to efficient code. *Proc. of the ACM on Programming Languages (PACMPL)* 3, ICFP (July 2019). https://doi.org/10.1145/3341702

[66] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. 2018. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proc. of the ACM on Programming Languages (PACMPL)* 2, ICFP (2018). https://doi.org/10.1145/3236782

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

[68] Nevin Lianwen Zhang and David Poole. 1994. A simple approach to Bayesian network computations. In *Proc. of the 10th Canadian Conference on Artificial Intelligence*.

[69] Yizhou Zhang and Nada Amin. 2022. Reasoning about “reasoning about reasoning”: Semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. of the ACM on Programming Languages (PACMPL)* 6, POPL (2022). https://doi.org/10.1145/3498677

Received 2023-11-16; accepted 2024-03-31