Towards Correct-by-Construction Probabilistic Inference

Eric Atkinson
MIT CSAIL
eatkinson@csail.mit.edu

Michael Carbin
MIT CSAIL
mcarbin@csail.mit.edu

Abstract

Researchers have recently proposed several systems that ease the process of Bayesian probabilistic inference. In this process, a developer must first devise a probabilistic model, and then build a program to perform inference over the model. However, existing systems have only limited support for reasoning about the correctness of the resulting inference program. In this paper, we present Shuffle, a programming language for probabilistic inference with correctness guarantees. We show how building an efficient inference program can be accomplished by applying proof rules that compose Shuffle programs.

1 Introduction

Researchers have recently proposed several systems that ease the process of Bayesian probabilistic inference wherein a developer specifies a generative probabilistic model and, through a combination of both automated and manual method methods, computes a posterior probability distribution for a set of variables in the model. These systems support specification approaches that range from declarative specifications of limited classes of graphical models [9] to Turing-complete stochastic programs. The inference strategies supported by these systems include general automated inference (using a single or small set of inference algorithms) [3, 4], libraries of optimized inference programs for specific models [9], and handcoded inference programs [6].

In cases where a developer can write some or all of the inference code, existing probabilistic inference systems do not enable developers to ensure that their inference programs are correct. Potential sources of errors include both 1) standard programming errors that may cause an inference algorithm to perform incorrect arithmetic or mistakenly refer to the wrong random variable and 2) high-level inference errors such as inference steps that are unsound given the dependencies between random variables in the model. In this paper we present Shuffle, the first programming system for manual development of inference programs with correctness as a primary concern.

Language:  Shuffle provides a programming language that enables the developer to specify both models and inference procedures with the same language by including conditional probability distributions as first-class objects. Specifically, an object in Shuffle may be a density function – which computes the probability of an event in the model – a sampler – that computes a sample from a distribution – or an estimator that computes a weighed stream of samples from a distribution.

Type System:  The types of Shuffle’s objects are rich in that a type includes the random variables over which the object operates. For example, a function $f$ can be explicitly typed as a density function for the distribution $\Pr(A \mid B)$ where $A$ and $B$ are random variables in the model.

Combinator Library:  Shuffle also provides a library of functions or combinators that enable developers to manipulate and compose distributions objects into new distribution objects. For example, Shuffle provides a multiplication operator that given a function $f(a)$ that computes the density of $\Pr(A)$ and a function $g(b, a)$ that computes the density of $\Pr(B \mid A)$, infers that the function $\lambda a. \lambda b. g(b, a) * f(a)$ is a function that computes the density of $\Pr(A, B)$.
Figure 1: A Gaussian mixture model with four observations and two mixture components in Shuffle.

**Correctness:** Each application of a combinator may require a precondition on the structure of the model. For example, a developer can coerce a function that computes a density for \( \Pr(B \mid A) \) to one that is a density for \( \Pr(B \mid A, C) \) if the random variable \( C \) is independent of \( B \) given the dependencies of the model. For each required precondition in an inference program, Shuffle generates a *proof obligation* that the developer can either manually audit, mechanically verify with a system such as Coq, or automatically solve with an SMT solver.

**Inference Compilation.** Given a model specification and inference program, Shuffle produces an implementation of each definition in the inference program. Shuffle maps each operation in the inference specification to a sequence of operations that execute in a simple lambda calculus-like intermediate representation. Shuffle relies on this generic inference program representation because it makes it possible to translate the generated inference procedures to either direct implementations in a standard language with random primitives (e.g., Python, Java, or C++) or to a probabilistic programming language, such as Venture [6].

**Summary of Contributions:** Shuffle enables a developer to build well-typed inference algorithms using a compositional set of primitives that makes plain an algorithm’s correctness specification.

## 2 Example: Gaussian mixture model

To use Shuffle to create an inference program, a developer first specifies a probabilistic model. Figure 1 presents a specification of a two-component Gaussian Mixture Model (GMM), a model for representing clustering relationships. In general, an \( n \)-component GMM models a set of real-valued datapoints as a set of noisy observations, each coming from one of \( n \) real-valued quantities termed *mixture components*. Each observation is Gaussian distributed with the value of one of the \( n \) mixture components as its mean. In addition, the value of each mixture component has a Gaussian prior with mean 0. We assume the variances of these Gaussians are fixed.

**Specifying Random Variables.** The two-component GMM specified here has four observations contained in \( \text{obs} \) and two mixture components in \( \text{mu} \). Figure 1 states what set each variable belongs to. Note that \( \mathbb{R} \) refers to a real number, \( \{0,1\} \) refers to a finite set, and \( \rightarrow \) refers to a function set. We model collections of random variables as random functions from a *domain* to a *target set*. For instance, \( \text{obs} \) represents all of the datapoints in the GMM, but \( \text{obs}(0) \) represents a single random variable corresponding to a single element of the domain \( \text{Samples} \).

A GMM models the uncertainty in the attribution of each observation to a mixture component with an explicit set of random variables \( z \) (one for each observation). If \( z(i) = 0 \), then \( \text{obs}(i) \) is an observation of mixture component \( \text{mu}(0) \) – and therefore its Gaussian has \( \text{mu}(0) \) as its center. Alternatively, if \( z(i) = 1 \), then \( \text{obs}(i) \) is an observation of \( \text{mu}(1) \).
Figure 2: Fragment of Exact Inference Program for GMM

Specifying Distributions. Figure 1 also specifies the probability densities for the variables in the model. The notation \( f : \text{density}(A \mid B) \) denotes the judgment “\( f \) is a conditional probability density for \( A \) given \( B \)”, where \( f \) is a Shuffle density function. Each function may also be quantified with meta-variable that takes on a value from a domain. In the definition of \( \mu_{\text{Prior}} \), \( j \) quantifies over all mixture component indices (\( \text{Mus} \)) and provides a density for each corresponding \( \mu(j) \). The unused first parameter in \( \mu_{\text{Prior}} \) refers to the quantified variable \( j \), and the second refers to the value of \( \mu(j) \). If this distribution were conditioned on other variables, they would appear as further parameters to the density function.

The type of a density function can also include constraints. The density function \( \text{obsDensity} \) models mixture assignments with the constraint that the function computes the density of \( \text{obs}(i) \) given that its assignment \( \text{z}(i) \) equals \( j \) and given the value of \( \mu(j) \). Constraints therefore enable a density function to express parametrized dependencies (as a function of each quantified variable) as well as dynamic dependencies (as a function of the observed value of other random variables in the model). We restrict the language of constraints to equalities and inequalities over linear expressions of quantified parameters and observed random variables. A key motivation for this design is that Shuffle models can express dynamic dependencies while still being amenable to static dependency analyses. This stands in contrast to Turing-complete probabilistic programming languages in which precise dependency analysis is undecidable in general.

Random Primitives. The notation \( \text{normalD}(x, \mu, \text{var}) \) refers to the Normal, or Gaussian, probability density of some variable \( x \) given mean \( \mu \) and variance \( \text{var} \). The notation \( \text{flip}(\text{Mus}, \text{zi}) \) refers to a density for the two-element domain \( \text{Mus} \) that assigns equal probability to each possible value of \( \text{zi} \). It is equivalent to the constant function that always returns 0.5.

Inference. Shuffle enables a developer to soundly construct a Shuffle inference program. An inference program represents a specific distribution in the model. For our example two-component GMM, the distribution we would like to compute is the distribution of the mixture components given the set of observations, \( \Pr(\mu \mid \text{obs}) \).

2.1 Exact Inference

One approach to compute the distribution \( \Pr(\mu \mid \text{obs}) \) is to create a probability density function that exactly computes the distribution. One implementation of this approach is to compute the joint density of \( \mu \) and \( z \) given \( \text{obs} \) (\( \Pr(\mu, z \mid \text{obs}) \)) and marginalize over \( z \). Figure 2 presents a fragment of a Shuffle program that implements this strategy. Shuffle’s primary benefit is that it provides a developer with primitives that enable him or her to manually compose and transform the functions specified in Figure 1 to produce the sound implementation presented in Figure 2.

Density Multiply. In this figure, \( \mu_{\text{Joint}} \) is the joint density function we desire to build. The developer constructs the density function using the \( \text{multiply} \) construct, which multiples two density functions together. In this case, the implementation of \( \mu_{\text{Joint}} \) multiplies together \( \mu_{\text{PostCond}} \) and \( \text{zPost} \), which each compute the density of \( \Pr(\mu(j) \mid \text{obs}, z) \) (where \( j \) is a mixture component)
and \( \Pr(z \mid \text{obs}) \), respectively. For clarity of presentation, we elide the implementation of \( z\text{Post} \) and only present the implementation of \( \mu\text{PostCond} \).

**Bayes’ Rule.** The implementation of \( \mu\text{PostCond} \) leverages the \texttt{bayes\_rule} operation to compute a density function of the appropriate type. The operator \texttt{bayes\_rule} takes two densities as input: a density function for \( \Pr(B \mid A, C) \) and a density function for \( \Pr(A \mid C) \) and constructs a density function for \( \Pr(A \mid B, C) \) using Bayes’ Rule. Namely,

\[
\Pr(A \mid B, C) = \frac{\Pr(B \mid A, C) \Pr(A \mid C)}{\int \Pr(B \mid A=a, C) \Pr(A=a \mid C) \, da}
\]

In this case, the implementation constructs \( \mu\text{PostCond} \) from the density functions \texttt{obsDensityAll} and \( \mu\text{Prior}' \). The internal implementation of the function that Shuffle generates uses symbolic integration to implement the denominator. In general, Shuffle may introduce intractable integrals. However, in practice, Shuffle includes program transformation and simplification rules that can eliminate these integrals when more efficient implementations are known (e.g., conjugate priors).

**Subset Density Product.** The function \texttt{obsDensityAll} computes the joint density of all observations with \( j \) as the assigned mixture component given \( z \) and \( \mu(j) \). The developer uses the \texttt{subst\_set\_prod} function to compute this density function.

The function \texttt{subst\_set\_prod} takes as input a density function that is indexed by a quantifier (in this case \texttt{obsDensity} is indexed by the sample identifier) a set of objects from the quantifier’s domain (in this case the set of sample identifiers, \( os \)), and constructs a density function that multiplies the results of \texttt{obsDensity} for each sample identifier. The result is a density function for all variables for which the quantified value satisfies the constraints, conditioned on any variables arising from the original density (\texttt{obsDensity} in this case) and any variables arising from the constraints themselves.

**Add Independent Parameters.** The function \( \mu\text{Prior}' \) computes the density of the mixture component \( j \) given \( z \). In the model, the distribution of each \( \mu(j) \) is independent of \( z \). The developer can therefore construct the appropriate density function by coercing a function that computes the prior distribution of \( \mu(j) \) to one that computes the distribution of \( \mu(j) \) given \( z \). The developer achieves this via the \texttt{add\_independent\_parameters} function which, \( z \) to the set of observed random variables in \( \mu(j) \)'s prior, \( \mu\text{Prior} \).

To ensure that the use of \texttt{add\_independent\_parameters} is sound, Shuffle emits a proof obligation that asserts that \( \mu(j) \) must be independent of \( z \). Such proof obligations provide a verifiable record of the model properties that must be true for the inference program to be sound.

**Summation.** The final step to compute the posterior distribution of \( \mu \) (\( \mu\text{Post} \)) is to sum over \( \mu\text{Joint} \) for all possible values of \( z \). The developer achieves this step through the \texttt{sum\_all} construct. This construct takes as input a joint distribution and an argument that specifies the position of the variable to sum over. In this case, the developer specifies a sum over \( \mu\text{Joint} \) on the variable at index 1 (i.e., \( z \) in zero-based index notation).

### 2.2 Approximate Inference

An alternative to exact inference is approximate inference. An approximate inference program estimates the posterior distribution instead of computing it exactly. Shuffle supports approximate inference programs by enabling a developer to specify and implement an estimator.

An estimator produces a list of weighted samples that can be used to approximately answer questions about the distribution the estimator represents. As the number of samples increases, the approximation becomes more accurate. For example, given an estimator \( \mu\text{Estimate} \) for the distribution \( \Pr(\mu(0), \mu(1) \mid \text{obs}) \), a program external to Shuffle could estimate the probability that the data points came from sources that are far apart:

```python
def muApprox(obs):
    sum = 0
    total = 0
    for (mu0Sample, mu1Sample, weight) in muEstimate(obs):
        sum = sum + (if abs(mu0Sample-mu1Sample)>10 then weight else 0)
    total = total + weight
    sum / total
```
def muSampler(os) : sampler (mu | obs)
{
    zs = zSample(os);
    return muSample(os, zs);
}

def zSample : sampler(z | obs) =
    combine_gibbs((map zSampleJ Samples))

def zSampleJ :
    forall j : Samples. sampler(z(j) | obs, z{i : Samples | i != j}) =
    categorical(zCond);

def zCond :
    forall j : Samples. density(z(j) | obs, z{i : Samples | i != j}) =
    ...

def muSample(os, zs) : sampler(mu | obs, z)
{
    s0 = muSampleK(Mus.0, os, zs);
    s1 = muSampleK(Mus.1, os, zs);
    return (s0, s1);
}

def muSampleK : forall k : Mus. sampler(mu(k) | obs, z) =
    normal_sample(muPostCond);

def muEstimate : estimator(mu | obs) =
    repeat(muSampler);

Figure 3: Fragment of Approximate Inference Program for GMM

In Shuffle, a developer builds an estimator out of a small set of operators. Figure 3 presents a fragment of an approximate inference implementation for GMM.

**Composition.** Shuffle enables a developer to compose two samplers together. Shuffle specifically enables a developer to implement a sampler via a sequence of assignment statements that each assign the output of a sampler invocation to a local variable. For example, the sampler muSampler (a sampler for the target distribution Pr(mu | obs)) composes the sampler zSample with the sampler muSample to produce its result. The function zSample produces a sample of the mixture component assignments according to Pr(z | obs). Given those assignments, muSampler produces a sample of the mixture components according to Pr(mu | z, obs).

**Gibbs Sampling.** In this example, the developer uses Shuffle’s support for Gibbs sampling to construct zSample. The combine_gibbs primitive takes as input a list of samplers over a domain, where each sampler generates a sample conditioned on all other elements in the domain. In this case, the developer constructs this list by mapping zSampleJ over the domain of Samples.

**Sampling Discrete Distributions.** The developer constructs zSampleJ using the categorical primitive, which automatically samples a discrete distribution given a density for the distribution. We elide the definition of zCond for clarity of presentation.

**Sampling Continuous Distributions.** The sampler muSample – composes together a sampler for each mu(k). Each sampler generates samples from mu’s distribution via the normal_sample primitive. This primitive takes as input a density function with a normal distribution as its underlying distribution produces a sample. In this case, the developer uses muPostCond from Figure 2 as the density function.

**Estimators.** In the last step, the developer builds the estimator Pr(mu | obs). The function muEstimate implements this estimator via the repeat construct that builds a list of samples by repeatedly invoking the specified sampler.
3 The Shuffle System

Shuffle provides developers with a suite of strongly typed combinators for manipulating densities, samplers, and estimators.

**Densities.** A density is a function from a parameter space to the positive real numbers. Developers can combine densities using a set of arithmetic operators including addition, multiplication, and integration. For each of these operations, Shuffle directly tracks the type of the resulting values, thereby ensuring that the resulting terms implement their intended distributions.

**Samplers.** A sampler is a stochastic function that draws a single sample from a distribution. A developer can compose multiple samplers using assignment statements as well as self-compose a sampler using a fixpoint construct that computes the sampler’s fixpoint.

For example, the implementation of `combine_gibbs` in Figure 3 uses Shuffle’s composition and fixpoint statements to compose the correct Gibbs update and then compute the fixpoint – via repeated application of the update – of the implied Markov chain.

**Estimators.** An estimator approximates a distribution with a weighted list of samples. As shown in Figure 3, the `repeat` operator builds an estimator out of a sampler and specifies that each sample has weight 1. Shuffle includes additional language constructs to manipulate each sample’s weight as well as remap weight-sample pairs from an existing estimator into a new estimator. For example, we have used these more advanced estimator constructs to implement a Rao-Blackwellized Particle Filter for the Simultaneous Localization and Mapping problem [1]. Shuffle’s additional estimator constructs make it possible to soundly implement the importance sampling methods required by this technique.

**Type Checking and Verification Condition Generation.** Shuffle infers the type of each operation in a program. As a result, for any variable in the program, a developer may optionally provide a type annotation to assert the intended type of the term. In addition to type checking, Shuffle also generates verification conditions when combinators in the language require variable independence assertions or assertions that assert that a sampler’s output spans its entire range. This latter condition is important for some MCMC approximate inference techniques. For example, Shuffle generates 11 independence assertions and 4 reachability assertions for our approximate inference implementation for GMM.

4 Related Work

There are several systems that enable their users to specify and perform probabilistic inference.

Church [4] and WebPPL [3] enable a user to specify Turing-complete stochastic programs as models, but restrict inference algorithms to all-purpose algorithms such as Metropolis-Hastings [7, 5]. JAGS [9] provides a notation for expressing graphical models and automatically performs Gibbs sampling for a fixed set of distributions. JAGS therefore provides automated support for a subset of Shuffle’s rules. For example, JAGS can automatically generate a collapsed sampler for GMM. However, it can do so only if the model is specified with a monolithic GMM primitive. This stands in contrast to Shuffle, which, via its compositional nature, enables a user to prove the correctness of collapsed sampling approaches for a wide class of models.

Other systems enable a user to augment the system’s inference approach with arbitrary code. Venture [6] enables a user to manually add code to the system’s inference program. However, when the user augments the inference program with arbitrary code, there is no guarantee that the resulting inference program is correct. In contrast, the code that a user generates with Shuffle is in accordance with the Shuffle’s proof rules and therefore enjoys Shuffle’s correctness claims.

Finally, there are languages that have built-in systems for simplifying probability densities. Hakaru [8] employs the Maple computer algebra system and PSI [2] is a solver dedicated specifically to this problem. These systems provide correct densities for a larger class of distributions than those support by Shuffle. However, these systems are not compositional in the same sense as Shuffle. PSI only handles densities, and cannot reason about composing samplers and estimators. Hakaru does have analogies to all objects in Shuffle, but only transforms whole programs. Shuffle’s proof rules can build a whole inference program out of smaller programs for sub-components of the model.
References


