Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2017-014 November 9, 2017

Typesafety for Explicitly-Coded Probabilistic Inference Procedures

Eric Atkinson, Michael Carbin

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Typesafety for Explicitly-Coded Probabilistic Inference Procedures

ERIC ATKINSON, MIT
MICHAEL CARBIN, MIT

Researchers have recently proposed several systems that ease the process of developing Bayesian probabilistic inference algorithms. These include systems for automatic inference algorithm synthesis as well as stronger abstractions for manual algorithm development. However, existing systems whose performance relies on the developer manually constructing a part of the inference algorithm have limited support for reasoning about the correctness of the resulting algorithm.

In this paper, we present Shuffle, a programming language for developing manual inference algorithms that enforces 1) the basic rules of probability theory and 2) statistical dependencies of the algorithm's corresponding probabilistic model. We have used Shuffle to develop inference algorithms for several standard probabilistic models. Our results demonstrate that Shuffle enables a developer to deliver performant implementations of these algorithms with the added benefit of Shuffle's correctness guarantees.

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 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 [18] 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) [8, 9], libraries of optimized inference programs for specific models [18], and handcoded inference programs [14, 19].

In cases where a developer can write some or all of the inference code, existing probabilistic inference systems have limited capability to help developers ensure that their inference programs are correct. Potential sources of errors include both 1) standard programming errors and 2) high-level inference errors in which the resulting inference code does not adhere to the rules of probability theory.

In this paper we present Shuffle, a programming language that provides developers with tools to reason about whether their programs 1) respect the statistical dependencies of their probabilistic model and 2) adhere to the basic rules of probability theory. Shuffle enables developers to explicitly specify their probabilistic model, which then serves as a specification from which Shuffle defines the semantics of terms in a program. Given the semantics of a program's terms, Shuffle can then ascribe a type for each term and verify that overall program is typesafe.

The contributions of this work are:

Language: Shuffle provides a set of operators that enable a developer to compose terms to produce an inference procedure. The types of Shuffle's terms are associate each term with random variables in a probabilistic model. For an 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.

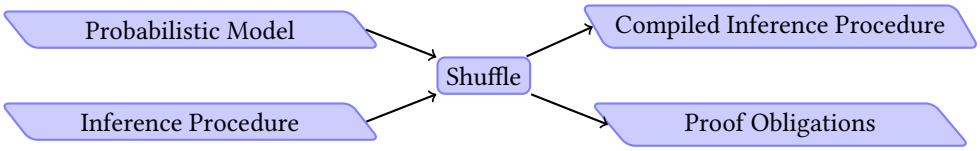

Fig. 1. Illustration of how a user interacts with Shuffle.

```
1  model GMM
2  {
3    domain Samples = {0:4};
4    domain Mus = {0:1};
5
6    variable R[Samples] obsSamples;
7    variable R[Mus] mu;
8    variable Mus[Samples] z;
9
10   def muPrior(j in Mus) : density(mu[j]) =
11     normal(mu[j],0,100);
12
13   def zPrior(i in Samples): density(z[i]) =
14     uniform(Mus,z[i]);
15
16   def obsDensity(i in Samples, j in Mus) :
17       density(obs[i] | mu[j], z[i], z[i] == j)
18       = normal(obs[i],mu[j],1)
19 }
```

Fig. 2. A Gaussian mixture model with four observations and two mixture components in Shuffle.

Correctness: We present a semantics of Shuffle objects, and we sketch a proof that Shuffle's type system is sound with respect to this semantics and can be used to ensure correctness of Shuffle-generated inference algorithms. The application of each Shuffle type rule may require a precondition. For example, a developer can coerce a density function with the type $\Pr(B \mid A)$ to a density function with the type $\Pr(B \mid A,\ C)$ if the random variable C is *statistically independent* of B given the dependencies of the model.

Shuffle Environment: Figure 1 shows how a user interacts with Shuffle. Shuffle takes as input a probabilistic model and an inference procedure written in Shuffle's language. It generates an executable *inference program*, as well as a set of *proof obligations*. The inference program is a Python program. The proof obligations are extra preconditions that Shuffle cannot verify internally. Shuffle assumes the user will verify these with external tools such as manual auditing, mechanical verification with a system such as Coq, or an SMT solver which can automatically produce a solution.

2 EXAMPLE: GAUSSIAN MIXTURE MODEL

To use Shuffle to create an inference program, a developer first specifies a probabilistic model. Figure 2 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. For simplicity of presentation, we fix the variance of each mixture component. In addition, the value of each mixture component has a Gaussian prior with mean 0.

Specifying Random Variables. The two-component GMM specified here has four observations contained in `obs` and two mixture components in `mu`. Figure 2 states what set each variable belongs to. Note that `R` refers to the domain of real numbers, and `{0:1}` refers to a finite set consisting of the values `0` and `1`. We model collections of random variables as functions from a *domain* to a *target set*. For example, `obs` represents all of the datapoints in the GMM, but `obs[0]` represents a single random variable corresponding to a single element of the domain `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 `obs[i]` has been attributed to mixture component `mu[0]` – and therefore its Gaussian has `mu[0]` as its mean. Alternatively, if `z[i] = 1`, then `obs[i]` is an observation of `mu[1]` with `mu[1]` as its mean.

Specifying Distributions. Figure 2 also specifies the probability densities for the random variables in the model via the `def` statement. A `def` statement specifies the type and implementation of a named term in the environment. For example, the definition of `muPrior` on Line 10 states that `muPrior` is a function – with a *quantified* variable `j` that ranges over all values of `Mus` (denoted by `j in Mus`) – with the type `density(mu[j])`. A type specification `density(`$A|B,\phi$`)` denotes that the term is a conditional probability density for the set of random variables A given, optionally, the set of *conditioned* random variables B under the optional *constraint* ϕ. In the case of `muPrior`, B and ϕ are the default values of the empty set and *true*. This means that for all values of `j`, `muPrior` computes the density of the random variable `mu[j]`. The implementation of `muPrior` computes the density of each mixture component as with the model that each mixture component is normally distributed with mean zero and variance 100. The function `normal` is a Shuffle provided primitive for computing the density

The definition of `zPrior` on Line 13 gives the density for each mixture assignment, `z[i]`, as uniformly distributed over the domain `Mus` (i.e., `z[i]` takes on any value in the domain of `Mus` with equal probability).

The definition of `obsDensity` on Line 16 gives the density of each observation, `obs[i]`. Unlike `muPrior` and `zPrior`, `obsDensity` has a non-empty set of condition variables as well as a constraint. Namely, the density of each `obs[i]`, is conditioned on the random variable `z[i]` (the observation's mixture component assignment) and the random variable `mu[j]` when `z[i] == j` (the mean for the observation's assigned mixture component). Constraints therefore enable a density function to express parameterized 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 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 analysis using a constraint solver. This stands in contrast to Turing-complete probabilistic programming languages in which precise dependency analysis is undecidable in general.

Inference. Shuffle enables a developer to soundly construct an *inference program*. An inference program computes a conditional distribution from the model. For our example two-component GMM, the two distributions we are interested in are 1) the distribution of the mixture component assignments given the observations (the basic clustering problem of mapping to observations to

clusters) – generally denoted by P(`z | obs`) and 2) the distribution of the mixture component means – generally denoted by P(`mu | obs`). Shuffle enables a developer to compute these distributions through both *exact* and *approximate* inference techniques.

2.1 Exact Inference

One approach to compute P(`z | obs`) is to create a probability density function that exactly computes the distribution. One implementation of this approach in Shuffle is to construct a function with type `density(z, obs)` that computes the joint density of `z` and `obs` and then divide that density by a function with type `density(obs)` that computes the density of `obs`. This implementation approach follows straightforwardly from Bayes' Rule in that for all random variables A and B, $P(A, B) = P(A|B) \cdot P(B)$ (Bayes' Rule) implies that $P(A|B) = P(A, B)$ / $P(B)$ (provided that $P(B) > 0$).

Figure 3 presents a Shuffle program that implements this strategy. Lines 1 to 43 implement the supporting density functions that are required to produce the joint density of `obs` and `z`, denoted by the definition `zJoint` on Line 45, as well as the density of `obs`, denoted by `obsPrior` on Line 48. We have deliberately unfolded most of this computation to make the types of the intermediate density objects clear.

Independence. The first three definitions (`muPriorZ` on Line 1, `zPriorI` on Line 4, and `obsDensityI` on Line 17) leverage the statistical independence relationships of the model to coerce the densities within the model to different types.

- `muPriorZ`: The definition of `muPriorZ` coerces `muPrior` from the model to the type `density(mu[j]|z)` using the `independent` annotation on a `def`. This coercion is sound under the assumption that for all `j`, `mu[j]` is independent of `z`. A manual inspection of the GMM model confirms that this is in fact true and therefore that this is a sound coercion. Shuffle does not verify this independence relationship. Shuffle instead emits to a log the fact that the inference program uses an independence assumption. Shuffle specifically emits the assertion

  $$\forall j.\ \texttt{mu[j]} \perp\!\!\!\perp \texttt{z}.$$

- `zPriorI`: The definition for `zPriorI` coerces `zPrior` from the model definition to the type `density(z[i] | z{i0 in Samples: i0 < i})` using the `independent` annotation. This usage of independence demonstrates that a user can assert the independence of a variable with quantified sets of variables. In this case, the developer asserts that each `z[i]` is independent of all other `z[i0]` where `i0` is less than `i` (given the standard total ordering on the integers given by <). Shuffle emits to its log the independence assertion

  $$\forall\ \texttt{i.}\ \ \texttt{z[i]} \perp\!\!\!\perp \texttt{z\{i0 in Samples: i0 < i\}}.$$

- `obsDensityZ`: The definition for `obsDensityZ` uses the `independent` annotation to coerce `obsDensity` from the model definition to new type that adds to the set of conditioned variables the variable group `z{i0 in Samples: i0 != i}`

  This new variable group represents the set of all cluster assignments except for the one that is at position `i`. Shuffle emits to its log the assertion that

  $$\forall \texttt{i,j.}\ \ \texttt{z\{i0 in Samples: i0 != i\}} \perp\!\!\!\perp \texttt{obs[i] | mu[j], z[i], z[i] == j}$$

  A key observation here is that Shuffle enables a developer to specify constrained independenices that are valid only under a given condition: namely, that `z[i] == j`. This independence assertion states that an observation is independent of other cluster assignments given its own cluster assignment.

```
def independent muPriorZ(j in Mus) : density(mu[j] | z) =
    muPrior(j);

def independent zPriorI (i in Samples) :
    density(z[i] | z{i0 : i0 < i}) =
    zPrior;

def independent obsDensityZ (i in Samples, j in Mus ):
    density(obs[i] | z{i0 in Samples: i0 != i},
                     mu[j], z[i], z[i] == j) =
    obsDensity(i,j);

def obsDensityZ1 (i in Samples, j in Mus):
    density(obs[i] | mu[j], z, z[i] == j) =
    obsDensityZ(i,j);

def independent obsDensityI(i in Samples, j in Mus) :
    density(obs[i] | obs{i0 in Samples: i0 < i && z[i0] == j},
                         mu[j], z ,z[i] == j) =
    obsDensityZ1(i,j);

def independent obsProd(j in Mus) :
    density(obs{i0 in Samples: z[i0] == j} | mu[j], z) =
    prod i in Samples where z[i] == j : obsDensityI(i,j);

def obsJoint(j in Mus) :
    density(obs{i0: z[i0] == j}, mu[j] | z) =
    obsProd(j) * muPriorZ(j);

def obsMarg(j in Mus) :
    density(obs{i0 in Samples: z[i0] == j} | z) =
    int obsJoint(j) by mu[j];

def muPost(j in Mus) :
    density(mu[j] | obs{i0 in samples: z[i0] == j}, z) =
    obsJoint(j) / obsMarg(j);

def independent obsMarg1(j in Mus) :
    density(obs{i0 in Samples: z[i0] == j} |
            obs{i0 in Samples: z[i0] < j}, z) =
    (obsJoint(j) / (muPost(j));

def obsAll : density(obs | z) =  prod j in Mus : (obsMarg1);

def zJoint : density(obs, z) =
    obsAll(j) * (prod i in Samples where true : zPriorI);

def obsPrior : density(obs) =  int zJoint z;

def export zPost : density(z | obs) =  zJoint / obsPrior;
```

Fig. 3. Inference program for computing density(z | obs) for a GMM.

Density Product. In the next step on Line 2, the developer constructs the density `obsProd`, which is the density of all observations that belong to the same cluster. The developer uses the `prod` operator to compute this density function. The product operator takes as input a density function that is indexed by a quantified variable and constructs a density function that multiples the results of that density function for each possible value of the quantified variable. The syntax `where z[i] == j` denotes that this product does not apply to the whole domain of the index variable `i`, but only to values of `i` that satisfy the predicate `z[i] == j`. In this example, given the quantifiers `i` and `j`, the resulting density multiplies all values of `obsDensityI(i, j)` for which `z[i] == j`.

Density Multiplication. Shuffle also enables a developer to multiply densities as demonstrated in the definition for `obsJoint` on Line 6. In this definition, the developer multiples together `obsProd` and `muPriorZ`. Density multiplication corresponds to Bayes' rule: $P(A, B) = P(A|B) \cdot P(B)$. The left and right operands of the multiplication correspond to the first and second probabilitiy distributions, respectively, on the right side of the equality. The resulting density for `obsProd` and `muPriorZ` therefore computes the joint density of the mean of the mixture component `mu[j]` and the observations that have been assigned to that mixture component.

Integration. Shuffle also enables a developer to marginalize out variables in a density via integration as demonstrated in the definition for `obsMarg` on Line 10. In this definition, the developer integrates `obsJoint(j)` with `mu[k]`. This has the effect of eliminating `mu[k]` from the type of `obsJoint`, thereby computing the density of the observations from one cluster center

```
obs{i0 in Samples: z[i0] == j}
```

unencumbered by the actual cluster center `mu[j]`. Shuffle enables a developer to specify arbitrary integrals (and corresponding sums) over continuous (and discrete) spaces. To provide an implementation for integrations, Shuffle contains simplification rules that encode analytic solutions to certain integral forms. In cases where Shuffle cannot produce an efficient executable implementation via its simplification rules, Shuffle reports an error.

Density Division. Shuffle also enables the developer to divide densities as demonstrated in the definition for `muPost` on Line 14, which is a density for a cluster center given the observations from that center. In this definition the developer constructs `muPost` using the operator `/`, which divides the values returned by the two input densities `obsJoint` and `obsMarg`. This has the opposite effect of multiplication, shifting a variable — in this case `obs{i0: i0 < i && z[i0] == j}` — from being part of the joint density to being conditioned on.

Summary. The remaining supporting definition `obsAll` leverages the operations discussed thus far to enable the developer to produce our desired densities of `zJoint : density(obs, z)` and `obsPrior : density(obs)`, which, through the division of the two densities, enable the developer to produce `zPost : density(z | obs)` (Line 50).

2.2 Approximate Inference

An alternative to exact inference is *approximate inference*. An approximate inference algorithm *estimates* the posterior distribution instead of computing it exactly. This may be more efficient for some models. In the exact inference algorithm for GMM, although Shuffle is able to generate an efficient implementation of the integration in `obsMarg` (Line 10), the integration over the discrete variable `z` in `obsPrior` has no simple solution and is tantamount to summing over all possible values of the variable group `z`. The variable group `z` is of the same size as the number of datapoints to the model and each variable may take on a value from `Mus`. The complexity of this summation is therefore $|Mus|^{|Samples|}$. In general, for large models, this summation is intractable.

```
1  def ziPost(i in Samples) : density(z[i] | z{k : k != i}, obs, true) =
2      ...;
3
4  def ziSample(i in Samples) :
5      sampler(z[i] | z{k : k != i}, obs) =
6      z[i] := sample ziPost(i);
7
8  def ziKernel(i in Samples) :
9      kernel(z[i] | z{k : k != i}, obs) =
10         lift zkSample(i);
11
12 def zKernel : kernel(z | obs) =
13     join i in Samples: zKernel(i);
14
15 def zSample : kernel(z | obs) =
16         fix zKernel();
17
18 def export zEst : estimator(z | obs) =
19         lift zSample();
```

Fig. 4. Approximate Inference Fragment for GMM. We present the full algorithm in Appendix B.

```
1  def muApprox(obs, count) :
2     sum = 0
3     total = 0
4     z = zeros(len(obs))
5     for i in range(num):
6       weight = zEst(obs,z)
7       sum += (weight if z[0] != z[1] else 0)
8           total += weight
9     return sum / total
```

Fig. 5. Python code for using the extracted code for `zEst` to estimate the probability that observation `0` and 1 are in different clusters. Note that `zeros(n)` returns a list of n zeros and `zEst` destructively updates the `z` variable

Figure 4 presents a fragment of an alternative approximate inference implementation in Shuffle for GMM that avoids executing the full summation. The primary result of this algorithm is an *estimator* for the distribution $P($`z | obs`$)$, `zEst` (Line 18). 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.

Figure 5 presents an example of how one would use an external program written in Python to use an estimator generated by Shuffle to estimate the probability that datapoint `0` and datapoint 1 are in different clusters. Specifically, repeatedly calling `zEst` produces a stream of weighted samples from the distribution of $P($`z | obs`$)$ and the resulting program computes the expectation of the indicator function that returns 1 when `z[0] != z[1]`.

Shuffle enables developers to implement *approximate* inference algorithms, which are often higher-performance than their exact counterparts, by exposing abstractions for *samplers*, *kernels*, and *estimators* as primitives in the language.

Samplers. A sampler with type denoted by `sampler(A | B)` is a function that produces a single sample of the random variable A given a value for the random variable B and a source of randomness. In Figure 5, the definition `ziSample : i. sampler(z[i] | z{k in Samples: k != i)}, obs)` implements a sampler that produces a value for `z[i]` given values for all differing `z[k]` and all of the observations. The developer implements this by directly sampling from the density `ziPost`, which computes the density for that distribution. We have elided the definition of `ziPost` for clarity of presentation, but a developer can produce `ziPost` using similar density arithmetic operations as those in the exact inference algorithm.

Kernels. A kernel with type denoted by `kernel(A | B)` is a function that produces a single sample of the random variable A given values for both A and B and a source of randomness. Kernels enable an inference algorithm to explore the sample space of a distribution by generating a new sample sample point in the space given a starting point within the space. Shuffle enables a developer to directly create a kernel from a sampler using the `lift` statement as in the definition of `ziKernel` (Line 9).

Given a kernel, a developer can also instantiate and compose repeated applications of a kernel for all values of a quantified variable. For example, in the definition of `zKernel`, the developer uses `join` to produce samples for each `z[k]` in turn, thereby producing a kernel for all of `z` – denoted by the type `kernel(z | obs)`. Finally, a developer can create a sampler from a kernel. In the definition of `zSample` the developer uses the `fix` operator to convert a kernel into a sampler for a distribution. The key observation here is that the `fix` operator computes the fixpoint (via iterative self-application) of the `kernel` which, in the limit, is semantically equivalent to a sampler.

Estimator. An estimator with type denoted by `estimator(A | B)` is a function that given a value of the random variable B produces a random sample of A and a weight for that sample. In the definition of `zEst`, the developer directly lifts a sampler to be an estimator with the resulting estimator producing samples directly from the sampler with a weight of 1.

Summary. Together, Shuffle's abstractions for densities, samplers, kernels, and estimators enable developers to compose inference procedures with strongly-typed abstractions that 1) prevent developers from making common inference mistakes and 2) provide an audit trail for common modeling assumptions, such as independence. In the remaining sections, we present the full Shuffle language along with its semantics, type system, soundness proofs, and an evaluation of the performance of Shuffle on several models.

3 THE LANGUAGE

Figures 6 and 7 present Shuffle's syntax for the declarative specification of the model and the code that implements an inference program, respectively.

3.1 Model

A probabilistic model, M, defines the model's domain of values, the model's set of random variables, and the probability densities that relate them.

Model Domains. A *domain declaration*, *DDecl*, specifies a domain $\delta \in \Delta$ of values.

$$
\begin{array}{rl}
M \rightarrow & \mathit{DDecl}^{+}\ \mathit{VDecl}^{+}\ \mathit{DDef}^{+} \\
\mathit{DDecl} \rightarrow & \texttt{domain}\ \delta\ \texttt{=}\ \{n\texttt{:}n\} \\
\mathit{VDecl} \rightarrow & \texttt{variable}\ (\delta \mid \texttt{R})[\delta]\ v \\
\mathit{DDef} \rightarrow & \texttt{def}\ x\ (\texttt{(}q\ \texttt{in}\ \delta\texttt{)}^{*})\texttt{:}\ T\ \texttt{=}\ D_m
\end{array}
$$

$$
\begin{array}{rl}
D_m \rightarrow & r \mid n \mid q \\
& \mid v[D_m] \mid a(D_m^{*}) \\
& \mid D_m \texttt{ * } D_m \mid D_m \texttt{ / } D_m \\
& \mid D_m \texttt{ + } D_m \mid D_m \texttt{ - } D_m \\
& \mid \texttt{if (}\phi\texttt{) \{ }D_m\texttt{ \} else \{ }D_m\texttt{ \}}
\end{array}
$$

$$
n \in \mathbb{N},\ r \in \mathbb{R},\ x \in X,\ v \in V,\ q \in \boldsymbol{Q},\ \delta \in \Delta
$$

$$
a \in \mathcal{A}
$$

$$
\begin{array}{rl}
T \rightarrow & T_b(V_g^{+}\Big(\texttt{ , }V_g^{+}\texttt{(, }\phi\texttt{)}^{?}\Big)^{?}\texttt{)} \\
T_b \rightarrow & \texttt{density} \mid \texttt{sampler} \mid \texttt{kernel} \mid \texttt{estimator}
\end{array}
$$

$$
\begin{array}{rl}
V_g \rightarrow & V \mid v \mid v\{q : \phi\} \\
V \rightarrow & v[n] \mid v[q] \\
\mathit{Op} \rightarrow & \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{<} \\
\phi \rightarrow & A\ \mathit{Op}\ A \mid \phi\ \texttt{\&\&}\ \phi \mid \phi\ \texttt{||}\ \phi \\
A \rightarrow & n \mid q \mid V
\end{array}
$$

Fig. 6. The Syntax of Shuffle Models and Types

$$
\begin{array}{rl}
P \rightarrow & \texttt{def independent}^{?}\ x\ \texttt{(}(q\ \texttt{in}\ \delta)^{*}\texttt{)}\texttt{:}\ T\ \texttt{=}\ (D \mid S \mid K \mid E)\ \texttt{;}\ P \\
\mid & \texttt{def export}\ x\ \texttt{(}(q\ \texttt{in}\ \delta)^{*}\texttt{)}\texttt{:}\ T\ \texttt{=}\ (D \mid S \mid K \mid E)
\end{array}
$$

$$
\begin{array}{rl}
D \rightarrow & x(A^{*}) \mid D \texttt{ * } D \mid D \texttt{ / } D \\
\mid & \texttt{int}\ D\ \texttt{by}\ V_g \mid \texttt{prod}\ q\ \texttt{in}\ \delta\ \texttt{where}\ \phi\ \texttt{:}\ D
\end{array}
\qquad
\begin{array}{rl}
S \rightarrow & x(A^{*}) \mid V\ \texttt{:=}\ \texttt{sample}\ D \\
\mid & S\ \texttt{;}\ S \mid \texttt{join}\ q\ \texttt{in}\ \delta\ \texttt{:}\ S \mid \texttt{fix}\ K
\end{array}
$$

$$
\begin{array}{rl}
K \rightarrow & x(A^{*}) \mid \texttt{lift}\ S \mid \texttt{join}\ q\ \texttt{in}\ \delta\ \texttt{:}\ K \mid K\ \texttt{;}\ K \\
E \rightarrow & x(A^{*}) \mid \texttt{lift}\ S \mid \texttt{factor}\ E\ \texttt{by}\ D
\end{array}
$$

Fig. 7. The Syntax of Shuffle Inference Programs

Model Variables. A *variable declaration*, *VDecl*, specifies a random variable $v \in V$. A random variable is array-valued, and a domain δ specifies the *index space* of the array.

Model Densities. A *model probability density*, D_m, defines a probability distribution through *density operators*. A model probability density is either a real number, r, a natural number n, a quantified variable q, a model random variable indexed by a model density $v[D_m]$, an atomic density called with model-density arguments $a(D_m^{*})$, a multiplication of two densities, D_m `*` D_m, a division of a density by another density, D_m `/` D_m, an addition of two densities, D_m `+` D_m, a subtraction of a density from another density, D_m `-` D_m, or a conditional switch between densities,`if (`ϕ`) {` D_m `} else {` D_m `}`.

Model Density Declarations. A *model probability density declaration*, *DDef*, defines a mapping between a variable $x \in X$ and a model probability density. The definition specifies a set of quantified variables $q \in \boldsymbol{Q}$ that are bound within D_m. The definition also specifies a type, T, for the definition.

The language of types $T_b(V_g^+\Big(\mid V_g^+(,\ \phi)^?\Big)^?)$ denotes that a object is either a `density`, `sampler`, `kernel`, or `estimator` that computes the probability of a set of random variables conditioned on another set of random variables, while subject to a constraint on the conditioned random variables. The random variables within either set may be either a singular random variable v, a single random variable from an array of random variables, $v[n]$ or $v[q]$, or a constrained subset of the random variables within an array, $v\{q : \phi\}$.

A constraint, ϕ, that appears in either a type or a random variable subset notation is a boolean predicate (with conjunction and disjunction) of inequalities over 1) integers, 2) quantified variables from domains that are isomorphic to the integers, and a single random variable with a value from a domain that is isomorphic to the integers.

3.2 Inference Program

Densities. A *program probability density*, D, defines a probability distribution through density operators. The items and operators allowed in an inference program are different than those allowed in a model. Namely, a program probability density is either an invocation of a density function, a multiplication of two densities, a division of two densities, an integration, or a product. While the model density language supports a rich set of language constructs, the inference density language consists only of the operators which are supported by Shuffle's type system.

Samplers. A sampler, S, defines a probability distribution through *sampler operators*. Sampler operators include invoking a defined sampler, updating a value with a sample from a density, concatenating two such samplers together, joining over a quantified sampler, and to computing the fixed point of a kernel.

Kernels. A kernel, K, defines a probability distribution in terms of *kernel operators*. Kernel operators include lifting a sampler, composing two kernels together, and joining over a quantified kernel.

Estimators. An estimator, E, defines a probability distribution as a weighted sampler. A Shuffle user can construct an estimator out of a sampler, and use a density to reweight the samples.

3.3 Semantics

The semantics of a Shuffle term p, where p may be a density, sampler, estimator, constraint, or variable access is given by $[\![p]\!]$. The type of $[\![p]\!]$ varies depending on the kind of the term p. The following sections describe the behaviour of $[\![p]\!]$ for each kind of term p.

3.3.1 Preliminaries.

Variables. Our formalization relies on several disjoint variable spaces. A *quantified variable* $q \in \boldsymbol{Q}$ is drawn from the space $\boldsymbol{Q}$; a *named distribution* $x \in X$ drawn from X, the space of distribution names, a *random variable* $v \in V$ is drawn from V, the space of variable names, a *domain* $\delta \in \Delta$ is drawn from Δ, the space of domain names, and an *atomic density* is drawn from $\mathcal{A}$, the space of pre-defined density atoms. Our semantics for Shuffle leverages three types of variables:

(1) **Quantified Variables.** The denotation of a quantified variable q is the value the environment maps q to. If q is not in the environment, then the denotation is an error.
(2) **Random Variables.** The denotation of a random variable v `[` e `]`, where e is either a quantified variable q or a literal n, is the value that the environment maps v `[` e `]` to. We assume that the environment always maps every random variable to a value.

$$
\begin{array}{lcl}
[\![d_1 * d_2]\!](\sigma) & = & [\![d_1]\!](\sigma) * [\![d_2]\!](\sigma) \\
[\![d_1 \ / \ d_2]\!](\sigma) & = & \begin{cases} \frac{[\![d_1]\!](\sigma)}{[\![d_2]\!](\sigma)} & [\![d_2]\!](\sigma) \neq 0 \\ \bot_0 & \textbf{else} \end{cases} \\
[\![\texttt{int } d \texttt{ by } V_g]\!](\sigma) & = & \int_{[\![V_g]\!](\sigma)} [\![d]\!] \\
[\![\texttt{prod } q \texttt{ in } \delta \texttt{ where } \phi : d]\!] & = & \prod_{n \in [\![\delta]\!]} \begin{cases} [\![d]\!](\sigma[q \mapsto n]) & [\![\phi]\!](\sigma[q \mapsto n]) \\ 1 & \textbf{else} \end{cases}
\end{array}
$$

Fig. 8. The denotational semantics of densities

(3) **Distribution Variables.** A named distribution variable x may be *invoked* with a sequence of arguments $e_0, \ldots$. If x exists in the environment, then the denotation of the invocation $x\ (e_0, \ldots)$ is the denotation of the procedure x refers to, with the parameters $q_0, \ldots$ rebound to the invocation arguments.
(4) **Domains.** The denotation of a domain $\delta \in \Delta$ is a range of natrual numbers: $[\![\delta]\!] = [n_1, n_2] \subset \mathbb{N}$.

Environments. An environment, $\sigma \in \Sigma = (V \times \mathbb{N}) + Q + X \to (\mathbb{R}^+ + \mathbb{N})$ is a finite map from random variables, quantified variables, and bound distributions. The notation $\sigma(e)$ denotes the value to which e is mapped by σ, which can either be 1) a random variable access (v, n) where n is a natural number 2) a quantified variable access q or 3) a named distribution access x.

We use the notation $\sigma[a \mapsto b]$ to mean σ with a, which could be any of the above, remapped to b. We also use σ to refer to an element of $\Sigma_{\text{rv}} = (V \times Q) \to (\mathbb{R}^+ + \mathbb{N})$, the subset of Σ that only maps random variables.

Constraints. Constraints may be recursively composed out of conjunctions and disjunctions, as well as built out of equalities and inequalities of quantified variables and random variables. The meaning of a constraint is a boolean value corresponding to whether the constraint holds or not.

Source of Randomness. A *source of randomness*, denoted by sr is an infinite sequence of uniform distributed values on the interval $[0, 1] \subset \mathbb{R}^+$. We let the notation $\int_{\text{sr}} f(\text{sr})$ denote an integral over a set of finite prefixes of sr. We let the function *split* denote a function that returns two *identical* sources of randomness in that the integrals $\int_{\text{sr}}^0 f(\text{sr}^0)$ and $\int_{\text{sr}}^1 f(\text{sr}^1)$ are equal for any positive measurable function f.

Errors. A Shuffle inference procedure may produce one of two error values instead of a conventional value: 1) a procedure produces the error value $\bot_\sigma$ if and only if it requires access to an element of the environment that is not within the environments domain and 2) a procedure produces the error value $\bot_0$ if and only if it contains a division by 0. In the semantics below we elide explicit failure propagation rules. However, in general, if an operator requires the results of multiple operands and more than one operand yields an error value, then the operation returns the join over all all operands as given by the lattice of elements $\{v, \bot_\sigma, \bot_0\}$ with the reflexive total order $v \leq \bot_\sigma, v \leq \bot_0, \bot_\sigma \leq \bot_0$ where v denotes a standard value.

3.3.2 *Densities.*

The denotation of a density d, denoted by $[\![d]\!] \in \Sigma \to (\mathbb{R}^+ + \bot)$, is a function from an environment to a positive real number or an error value.

$$
\begin{array}{lcl}
[\![v \texttt{ [} e \texttt{] := sample } d]\!](\sigma, \mathrm{sr}) & = & \arg\min_r \left(\int_{x \in (-\infty, r]} [\![d]\!](\sigma[(v, [\![e]\!](\sigma)) \mapsto x]) \right) > \mathrm{sr} \\
[\![s_1 \texttt{ ; } s_2]\!](\sigma, \mathrm{sr}) & = & [\![s_2]\!]([\![s_1]\!](\sigma, \mathrm{sr}^1), \mathrm{sr}^0) \\
\mathrm{jImpl}(q, s, [n_1, n_1], \sigma, \mathrm{sr}) & = & [\![s]\!](\sigma[q \mapsto n_1], \mathrm{sr}) \\
\mathrm{jImpl}(q, s, [n_1, n_2], \sigma, \mathrm{sr}) & = & [\![s]\!](\mathrm{jImpl}(q, s, [n_1, n_2 - 1], \sigma, \mathrm{sr}^0)[q \mapsto n_2], \mathrm{sr}^1) \\
[\![\texttt{join } q \texttt{ in } \delta \texttt{ : } s]\!](\sigma, \mathrm{sr}) & = & \mathrm{jImpl}(q, s, [\![\delta]\!], \sigma, \mathrm{sr})
\end{array}
$$

Fig. 9. Denotational semantics for samplers.

Multiplication. The $*$ operator takes two densities and multiplies them together pointwise. Multiplication is the primary way information from multiple densities is combined in Shuffle. For example, if d_1 is a density for the distribution $\Pr(A|B, C)$, where A, B, and C are subsets of the random variables in σ, and d_2 is a density for $\Pr(B|C)$, then $d_1 * d_2$ is a density for the distribution $\Pr(A, B|C)$.

Division. The `/` operator divides the first density by the second pointwise. Division's serves as an inverse operation to multiplication. If d_1 is a density for the distribution $\Pr(A, B|C)$, where A, B, and C are subsets of the random variables in σ, and d_2 is a density for $\Pr(B|C)$, then d_1 `/` d_2 is a density for $\Pr(A|B, C)$. Division also serves to eliminate random variables. If d_1 is a density for the distribution $\Pr(A, B|C)$ and d_2 is a density for $\Pr(B|A, C)$, then d_1 `/` d_2 is a density for $\Pr(A|C)$.

Integration. The syntax `int` d `by` V_g computes the integral of a probability density, d. It computes the integral of its density parameter over all possible values of the random variables, V_g. These random variables may take values in any valid Shuffle space. A developer uses integration to turn a *joint density* into a *marginal* density. Specifically, if d is a joint density for the distribution $\Pr(A, B|C)$, where A, B, and C are sets of random variables, then `int` d `by` B is the marginal density for $\Pr(A|C)$.

A developer has a choice whether to use division or integration to eliminate a random variable. If the developer can easily find densities for the distributions $\Pr(B|A, C)$ and $\Pr(A, B|C)$, then he or she can divide them to build a density for the desired distribution $\Pr(A|C)$. If the developer can find a density for the distribution $\Pr(A, B|C)$, and knows that either B is finite or the integral `int` d `by` B can be simplified, then he or she can use integration to build the density $\Pr(A|C)$.

Products. The syntax `prod` q `in` δ `where` ϕ `:` d combines an indexed family of densities into a single density using the multiplicative product of a constrained subset of densities in the family. Specifically, it computes the product of over all instantiations of the index, q, for which the constraint ϕ evaluates to "true". For an example use-case, if for all $n \in [\![\delta]\!] = [n_1, n_2]$, d is a density for the distribution $\Pr(v_n | \{v_{n'} : (n_1 \leq n' < n) \wedge [\![\phi]\!](\sigma[q \mapsto n'])\}, C)$, where v_n is an indexed subset of the random variables in σ and C is another subset of the random variables that is disjoint from all v_n, then `prod` q `in` δ `where` ϕ `:` d is a density for the distribution $\Pr(\{v_n : [\![\phi]\!](\sigma[q \mapsto n])\} | C)$.

3.3.3 *Samplers.*

The denotation of a sampler s, denoted by the semantic function $[\![s]\!] \in (\Sigma \times \mathrm{SR}) \to (\Sigma + \bot)$, is a function that takes an environment and a source of randomness, and produces a new environment or an error value.

Sampling a Density. The syntax v `[` e `] := sample` d constructs a sampler from the density d. The sampler updates σ so that the mapped value of $(v, [\![e]\!](\sigma))$ is overwritten with the newly

$$
\begin{array}{lcl}
[\![\texttt{lift}\ s]\!] & = & [\![s]\!] \\
[\![k_1\ ;\ k_2]\!](\sigma, \mathrm{sr}) & = & [\![k_2]\!]([\![k_1]\!](\sigma, \mathrm{sr}^1), \mathrm{sr}^0) \\
\forall f.\ \int_{\mathrm{sr}} f([\![\ \texttt{fix}\ k]\!](\sigma, \mathrm{sr})) & = & \int_{\mathrm{sr}} f([\![s;\ \texttt{fix}\ s]\!](\sigma, \mathrm{sr})) \\
\mathrm{jImpl}(q, k, [n_1, n_1], \sigma, \mathrm{sr}) & = & [\![k]\!](\sigma[q \mapsto n_1], \mathrm{sr}) \\
\mathrm{jImpl}(q, k, [n_1, n_2], \sigma, \mathrm{sr}) & = & [\![k]\!](\mathrm{jImpl}(q, k, [n_1, n_2 - 1], \sigma, \mathrm{sr}^0)[q \mapsto n_2], \mathrm{sr}^1) \\
[\![\texttt{join}\ q\ \texttt{in}\ \delta\ :\ k]\!](\sigma, \mathrm{sr}) & = & \mathrm{jImpl}(q, k, [\![\delta]\!], \sigma, \mathrm{sr})
\end{array}
$$

Fig. 10. Denotational semantics for kernels

sampled value. The denotation of the sample command uses *inverse transform sampling* to produce a value. Inverse transform sampling requires solving a definite integral, which may not be feasible in general. Thus, while Shuffle defines a semantics for any sampler, it can only generate code for samplers of finite densities, or Gaussian densities using the Box-Muller transform[3].

Sampler Composition. A developer can compose two samplers s_1 and s_2 with the syntax $s_1\ ;\ s_2$. Specifically, if s_1 is a sampler for the distribution $\Pr(B|C)$ and s_2 is a sampler for the distribution $\Pr(A|B, C)$, then $s_1\ ;\ s_2$ is a sampler for the distribution $\Pr(A, B|C)$, where A, B, and C are subsets of the random variables in σ. Semantically, this feeds the output state of s_1 into s_2.

Join. The syntax $\texttt{join}\ q\ \texttt{in}\ \delta\ :\ d$ combines an indexed family of samplers into a single sampler that combines the results of each member of the family. Namely, it iterates over instantiations of the index q, passing the returned state of the previous iteration as the input state to the next one. For an example use-case, if for all $n \in [\![\delta]\!] = [n_1, n_2]$, d is a sampler for the distribution $\Pr(v_n|\{v_{n'} : n_1 \le n' < n\}, C)$, where v_n is an indexed subset of the random variables in σ and C is another subset of the random variables that is disjoint from all v_n, then $\texttt{join}\ q\ \texttt{in}\ \delta\ :\ d$ is a sampler for the distribution $\Pr(\{v_n : n \in [\![\delta]\!]\}|C)$.

3.3.4 *Kernels.*

The denotation of a kernel k, written $[\![k]\!] \in (\Sigma \times \mathrm{SR}) \to (\Sigma + \bot)$, is a function that takes an environment and a source of randomness, and produces a new environment or an error value.

Lift. A developer can lift a sampler to a kernel. The resulting kernel has exactly the same behavior as the original sampler, and is used to represent the same distribution.

Kernel Composition. A developer can compose two kernels with the ; operator. Specifically, if k_1 is a kernel for the distribution $\Pr(A|B, C)$ and k_2 is a kernel for the distribution $\Pr(B|A, C)$, where A, B, and C are subsets of the random variables in σ, then $k_1\ ;\ k_2$ is a kernel for the distribution $\Pr(A, B|C)$. Composing kernels feeds the environment that results form applying the first kernel as input to the application of the second kernel. Note that while kernel and sampler composition have the same semantics, kernel composition has a different type signature.

Join. The syntax $\texttt{join}\ q\ \texttt{in}\ \delta\ :\ d$ combines an indexed family of kernels into a single kernel that combines the results of each member of the family. Namely, it iterates over instantiations of the index q, passing the returned state of the previous iteration as the input state to the next one. For an example use-case, if for all $n \in [\![\delta]\!] = [n_1, n_2]$, d is a kernel for the distribution $\Pr(v_n|\{v_{n'} : n' \ne n\}, C)$, where v_n is an indexed subset of the random variables in σ and C is another subset of the random variables that is disjoint from all v_n, then $\texttt{join}\ q\ \texttt{in}\ \delta\ :\ d$ is a kernel for the distribution $\Pr(\{v_n : n \in [\![\delta]\!]\}|C)$.

$$
\begin{array}{lcl}
[\![\texttt{lift}\ s]\!](\sigma, \mathrm{sr}) & = & (1, [\![s]\!](\sigma, \mathrm{sr})) \\
[\![\texttt{factor}\ e\ \texttt{by}\ d]\!](\sigma, \mathrm{sr}) & = & \text{let } (w, \sigma') = [\![e]\!](\sigma, \mathrm{sr}) \text{ in } (w * [\![d]\!](\sigma'), \sigma')
\end{array}
$$

Fig. 11. Denotational semantics for estimators

$$
\begin{array}{lcl}
[\![q]\!](\sigma) & = & \begin{cases} \sigma(q) & q \in \sigma \\ \bot_\sigma & \textbf{else} \end{cases} \\
[\![v\ \texttt{[}\ e\ \texttt{]}]\!](\sigma) & = & \sigma(v, [\![e]\!](\sigma)) \\
[\![x\ \texttt{(}e_0, \ldots\texttt{)}]\!](\sigma) & = & \text{let } (q_0, \ldots), p = \sigma(x) \text{ in} \\
 & & \begin{cases} [\![p]\!](\sigma[q_0 \mapsto [\![e_0]\!](\sigma)][\ldots]) & x \in \sigma \\ \bot_\sigma & \textbf{else} \end{cases} \\
[\![\texttt{def}\ x\ \texttt{(}q_0\ \texttt{in}\ \delta_0\texttt{, ...):}\ t = p_1\ \texttt{;}\ p_2]\!](\sigma) & = & [\![p_2]\!](\sigma[x \mapsto ((q_0, \ldots), p_1)]) \\
[\![\texttt{def independent}\ x\ \texttt{(}q_0\ \texttt{in}\ \delta_0\texttt{, ...):}\ t = p_1\ \texttt{;}\ p_2]\!] & = & [\![\texttt{def}\ x\ \texttt{(}q_0\ \texttt{in}\ \delta_0\texttt{, ...):}\ t = p_1\ \texttt{;}\ p_2]\!] \\
[\![\texttt{def export}\ x\ \texttt{(}q_0\ \texttt{in}\ \delta_0\texttt{, ...):}\ t = p]\!](\sigma) & = & [\![p]\!](\sigma)
\end{array}
$$

Fig. 12. Semantics of Shuffle's structural constructs

Fixed points. For a given kernel for a distribution, a developer can produce a sampler for the same kernel via the `fix` operator. The denotational semantics of `fix` are declarative, as Figure 10 specifies that the operator must have the property that the sampled distribution is invariant under composition with the kernel. Shuffle type checks its code assuming an exact implementation of `fix`, but generates code that approximately implements it by running the kernel and passing its output back to itself in an iterative process. As the number of iterations grows large, the approximate distribution approaches the true distribution.

3.3.5 Estimators.

The denotation of an estimator e, denoted by $[\![e]\!] \in (\Sigma \times \mathrm{SR}) \to ((\mathbb{R}^+ \times \Sigma) + \bot)$, is a function that takes as input a source of randomness and an environment, and produces either a pair consisting of a new environment and a weight associated with that environment, or an error value.

Lift. A developer can lift a sampler to an estimator. The resulting estimator always returns the value 1 as the weight of a sampler.

Factor. The `factor` e `by` d takes a distribution and reweights it according to its density argument. A developer can use a `factor` statement to add additional conditions to an estimator. If e is an estimator for the distribution $\Pr(A|B)$, and d is a density for the distribution $\Pr(C|A, B)$, where A, B, and C are subsets of the random variables in σ, then `factor` e `by` d is an estimator for the distribution $\Pr(A|B, C)$. The semantics of the `factor` statement calls its estimator argument to return an initial weight, and then takes the resulting state and passes it through the density to get a new weight. The new weight is then multiplied with the old one.

3.4 Structural Constructs

Literals. The denotation of a literal real number r or natural number n is the number itself, regardless of the environment.

Definitions. The denotation of a definition `def` x `(`q_0 `in` δ_0`, ...):` $t = p_1$ `;` p_2 is the denotation of p_2 with x bound to the procedure p_1 with the parameters $(q_0, \dots)$. The denotation erases the type t and the domains $(\delta_0, \dots)$.

4 TYPE SYSTEM

In this section we present Shuffle's type system.

4.1 Model

Variable Sets. A *variable set* is a comma delimited list of random variables (V_g^+ in Figure 6) that we denote by the symbols A, B, and C. We specify the semantics of a variable set by the semantic function $[\![A]\!] : \Sigma \to \mathcal{P}(\mathcal{V})$ where $\mathcal{V} = V \times \mathbb{N}$. The denotation of a variable set is therefore a set of pairs that each consist of a random variable and the corresponding index within that variable. For each syntatic form, we give variable sets the following denotation:

- **Set Comprehensions.** For variable sets of the form $A = v$ `{`q_0 `in` δ_1`:` ϕ`}`, we let $[\![A]\!](\sigma) = \{(v, n) \mid [\![\phi]\!](\sigma[q_0 \mapsto n])\}$.
- **Indexed Variables.** We define the variable set v `[` e `]` as syntactic sugar for the set v `{`q_0 `in` δ`:` q_0 `==` e`}`, with the corresponding denotation given by that for set comprehensions.
- **Whole Variables.** The variable set v is syntactic sugar for the set v `{`q_0 `in` δ`: true}`, with the corresponding denotation given by that for set comprehensions.

Variable Set List. The comma operator A, B unions two *disjoint* variable sets. Namely, we therefore define the denotation of this operator by the function

$$[\![A, B]\!](\sigma) = \begin{cases} [\![A]\!](\sigma) \cup [\![B]\!](\sigma) & [\![A]\!](\sigma) \cap [\![B]\!](\sigma) = \emptyset \\ \bot_\sigma & \textbf{else} \end{cases}$$

Joint Density. We define the *joint density* of all variables in the model, $\mathcal{J}$, as follows. Let `def` x_i `(`q_0 `in` δ_0`, ...):` $T_{bi}(A_i|B_i, \phi_i) = p_i$ be a declaration from the model. Then,

$$\mathcal{J}(\sigma) = \prod_i \prod_{\hat{n} \in (\delta_0, \dots)} \begin{cases} [\![p_i]\!](\sigma[q_0 \mapsto n_0] \dots) & [\![\phi_i]\!](\sigma[q_0 \mapsto n_0] \dots) \\ 1 & \textbf{else} \end{cases}$$

We define the notation $\mathcal{J}(S_1|S_2)$, where $S_1 \subseteq \mathcal{V}$, $S_2 \subseteq \mathcal{V}$, where $S_1 \cap S_2 = \emptyset$, as

$$\mathcal{J}(S_1|S_2) = \frac{\int_{\mathcal{V} - (S_1 \cup S_2)} \mathcal{J}}{\int_{\mathcal{V} - S_2} \mathcal{J}}$$

The term $\mathcal{J}(S_1|S_2)$ is a function of the type $\Sigma_{rv} \to \mathbb{R}^+$.

4.2 Typing Judgment

A typing judgment is a logical proposition of the form $\mathcal{M}, \Gamma, \mathcal{L} \vdash p : t$ where $\mathcal{M}$ is a model, Γ is a type environment, $\mathcal{L}$ is an assumption log, p is a Shuffle inference program, and t is a type.

An independent typing judgment is a logical proposition of the form $\mathcal{M}, \Gamma, \mathcal{L} \vdash_I p : t$ where $\mathcal{M}$ is a model, Γ is a type environment, $\mathcal{L}$ is an assumption log, p is a Shuffle inference program, and t is a type. Independence typing judgments only hold under *independence assumptions* provided by the developer.

Type Environment. A type environment, Γ, is an element of the language defined by the grammar

$$\Gamma \to \emptyset \mid \Gamma :: \beta$$
$$\beta \to [x : q^*, \delta^*, T] \mid [v : \delta, \delta] \mid [q : \delta]$$

where x is a named distribution, q is a quantified variable, T is a Shuffle type (Figure 6), v is a random variable name, and δ is a domain.

Assumption Log. An assumption log, $\mathcal{L}$, records the set of model and inference program assumptions made by the developer during the construction of their inference program. An assumption log is of the form

$$\mathcal{L} \to \emptyset \mid \mathcal{L} :: \alpha$$
$$\alpha \to (\phi \Rightarrow A \perp\!\!\!\perp B \mid C) \mid \text{ReachesAll}(s).$$

An individual assumption is therefore either a statistical independence assertion or a reachability assertion. The entries in an assumption log are logical propositions. We denote the semantics of each entry by the semantic function $[\![\mathcal{L}]\!] : \Sigma \to \mathbb{B}$, which we give by

$$\begin{aligned}
[\![\mathcal{L} :: a]\!](\sigma) &= [\![\mathcal{L}]\!](\sigma) \wedge [\![a]\!](\sigma) \\
[\![\phi \Rightarrow A \perp\!\!\!\perp B \mid C]\!](\sigma) &= [\![\phi]\!](\sigma) \Rightarrow \Big(\mathcal{J}([\![A,B]\!](\sigma)|[\![C]\!](\sigma))(\sigma) \\
&\quad = \mathcal{J}([\![A]\!](\sigma)|[\![C]\!](\sigma))(\sigma) * \mathcal{J}([\![B]\!](\sigma)|[\![C]\!](\sigma))(\sigma)\Big) \\
[\![\text{ReachesAll}(s)]\!](\sigma) &= \forall v, n.\ \Big(\exists \text{sr}, r.\ s(\sigma[(v,n) \mapsto r], \text{sr})(v,n) \neq r \Rightarrow \forall r. \exists \text{sr}.\ s(\sigma, \text{sr})(v,n) = r\Big)
\end{aligned}$$

Model Relationsihp for Environments. An environment σ can *model*, written $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$ a Shuffle model $\mathcal{M}$, type environment Γ, and assumption log $\mathcal{L}$. This relation is defined as

$$\begin{aligned}
\sigma \vDash_x \Gamma, \mathcal{M}, \mathcal{L} &= \forall x.(\mathcal{M}, \Gamma, \mathcal{L} \vdash x : \forall \hat{q}.\ t) \Rightarrow (x \mapsto (\hat{q}, \hat{\delta}, p) \in \sigma) \\
\sigma \vDash_q \Gamma, \mathcal{M}, \mathcal{L} &= \forall q.(\mathcal{M}, \Gamma, \mathcal{L} \vdash q : \delta) \Rightarrow (q \mapsto n \in \sigma) \wedge (n \in [\![\delta]\!]) \\
\sigma \vDash \Gamma, \mathcal{M}.\mathcal{L} &= \sigma \vDash_x \Gamma, \mathcal{M}, \mathcal{L} \wedge \sigma \vDash_q \Gamma, \mathcal{M}, \mathcal{L} \wedge \mathcal{J}(\sigma) > 0
\end{aligned}$$

and note that, for any model, there is a dynamic and static environment which each contain mappings for all densities and random variables in in the model, and these environments satisfy $\vDash$.

4.3 Types

Densities. A density is a function that, under any substitution of the relevant quantified variables, computes appropriate distribution from the model. Specifically if,

(1) $\mathcal{M}, \Gamma, \mathcal{L} \vdash d : \forall \hat{q}.\ \texttt{density}(A|B, \phi)$
(2) $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$

then, for any environment $\sigma' = \sigma[\hat{q} \mapsto \hat{n}]$, $[\![\phi]\!](\sigma') \Rightarrow [\![d]\!](\sigma') = \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma'))(\sigma')$

Samplers. A term with sampler type is a function with it is possible to compute the expectation of any positive function f under the distribution $P(A|B)$. Specifically, if

(1) $\mathcal{M}, \Gamma, \mathcal{L} \vdash s : \forall \hat{q}.\ \texttt{sampler}(A|B, \phi)$
(2) $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$

then for any environment $\sigma' = \sigma[\hat{q} \mapsto \hat{n}]$ and any $f \in \Sigma_{rv} \to \mathbb{R}^+$

$$[\![\phi]\!](\sigma') \Rightarrow \int_{\text{sr}} f([\![s]\!](\sigma', \text{sr})) = \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma'))(\sigma')$$

Kernels. A kernel k is a surjective function such that for any sampler s for a given distribution and any positive function f, the expectation of f under s is the same as that under the composition of s with k. Specifically, if

(1) $\mathcal{M}, \Gamma, \mathcal{L} \vdash k : \forall \hat{q}.\ \texttt{kernel}(A|B, \phi)$
(2) $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$

then for any environment $\sigma' = \sigma[\hat{q} \mapsto \hat{n}]$, $f \in \Sigma_{rv} \to \mathbb{R}^+$, and sampler s such that

$$\int_{\text{sr}} f(s(\sigma', \text{sr})) = \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma'))(\sigma')$$

,

(1) $[\![\phi]\!](\sigma') \Rightarrow \exists \epsilon > 0.\ \int_{\text{sr}} f(k(\sigma', \text{sr})) > \epsilon \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma'))(\sigma')$
(2) $[\![\phi]\!](\sigma') \Rightarrow \int_{\text{sr}^0, \text{sr}^1} f(k(s(\sigma', \text{sr}^0), \text{sr}^1)) = \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma'))(\sigma')$

Estimators. An estimator is a function that produces a sample and corresponding weight such that the expectation of a positive function f under the estimator is correct. Specifically, if

(1) $\mathcal{M}, \Gamma, \mathcal{L} \vdash \text{est} : \forall \hat{q}.\ \texttt{estimator}(A|B, \phi)$
(2) $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$

then for any environment $\sigma' = \sigma[\hat{q} \mapsto \hat{n}]$, and any $f \in \Sigma_{rv} \to \mathbb{R}^+$,

$$[\![\phi]\!](\sigma') \Rightarrow \int_{\text{sr}} \frac{\pi_0([\![\text{est}]\!](\sigma', \text{sr})) * f(\pi_1([\![\text{est}]\!](\sigma', \text{sr})))}{\int_{\text{sr}} \pi_1([\![\text{est}]\!](\sigma', \text{sr}))} = \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma'))(\sigma')$$

where π_0 and π_1 are *projections* that extract the first and second elements, respectively, of their pair-valued arguments.

4.4 Auxiliary Definitions.

We also define the semantics of the notation $\mathcal{M}, \mathcal{L} \vDash PR$, for the logical proposition PR, as follows:

$$\begin{array}{lcl} \mathcal{M}, \mathcal{L} \vDash \phi \Rightarrow A \perp\!\!\!\perp B \mid C & = & \forall \sigma.\ [\![\mathcal{L}]\!](\sigma) \Rightarrow [\![\phi \Rightarrow A \perp\!\!\!\perp B \mid C]\!](\sigma) \\ \mathcal{M}, \mathcal{L} \vDash \text{ReachesAll}(s) & = & \forall \sigma.\ [\![\mathcal{L}]\!](\sigma) \Rightarrow [\![\text{ReachesAll}(s)]\!](\sigma) \end{array}$$

During type checking, Shuffle checks a number of side predicates that are necessary to ensure safety properties.

(1) **Free variables.** The notation FV(e) means the *free variables* of the term e, which can either be a variable set or a constraint. The set of free variables contains the names of any quantified or random variables, with the exception of a variable set defined by $v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ \phi\}$, whose free variables to not include q_0. The notation FRV(e) has the same definition, but refers only to the random variables. Likewise, the notation FQV(e) refers to the free quantified variables.
(2) **Variable Subsets** If S is a set of variable names of the form $S = v_0, v_1, \ldots, v_n$, then the predicate $S \subseteq A$, for a variable set A, means that A must be of the form $A_0, A_1, \ldots, A_m$, and furthermore $S \subseteq \{A_0, A_1, \ldots, A_m\}$.
(3) **Quantified Variable Subsets.** The notation $\hat{q} \subseteq' \hat{q}'$, where q and q' are lists of quantified variable definitions, means that if $q_0 \in q$, then $q_0 \in q'$
(4) **Dynamic Dependence Checking.** If A and B are variable sets, the dynamic dependence predicate DDep(A, B) holds if for any random variable set $v\ \{q\ \texttt{in}\ \delta\texttt{:}\ \phi\}$ appears in either A or B, and $v'\ [\ q'\ ]$ appears in ϕ, then if $q \neq q'$, either v' or $v'\ [\ q'\ ]$ must appear in B. If $q = q'$, then v' must appear in B.
(5) **Valid Types.** A type $T_b(A|B, \phi)$ satisfies the predicate Valid($T_b(A|B, \phi)$) if

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 : \forall \hat{q}.\ \texttt{density}(A|B,C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash d_2 : \forall \hat{q}.\ \texttt{density}(B|C,\phi) \quad \text{DDep}(A,C)}{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 * d_2 : \forall \hat{q}.\ \texttt{density}(A,B|C,\phi)}\ \text{DMUL}$$

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 : \forall \hat{q}.\ \texttt{density}(A,B|C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash d_2 : \forall \hat{q}.\ \texttt{density}(B|C,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1\ /\ d_2 : \forall \hat{q}.\ \texttt{density}(A|B,C,\phi)}\ \text{DDIV}$$

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 : \forall \hat{q}.\ \texttt{density}(A,B|C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash d_2 : \forall \hat{q}.\ \texttt{density}(A|B,C,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1\ /\ d_2 : \forall \hat{q}.\ \texttt{density}(B|C,\phi)}\ \text{DDIV2}$$

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d : \forall \hat{q}.\ \texttt{density}(A,B|C,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{int}\ d\ \texttt{by}\ B : \forall \hat{q}.\ \texttt{density}(A|C,\phi)}\ \text{DINT}$$

$$\frac{\begin{array}{c}\mathcal{M},\Gamma :: [q:\delta],\mathcal{L} \vdash d : \forall q,\hat{q}'.\ \texttt{density}(v\ [\ q\ ]| \\ v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < q\ \texttt{\&\&}\ \phi[q_0/q]\},C,\phi) \qquad q \notin \text{FV}(C)\end{array}}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{prod}\ q\ \texttt{in}\ \delta\ \texttt{where}\ \phi : d : \forall \hat{q}'.\ \texttt{density}(v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ \phi[q_0/q]\}|C)}\ \text{DPROD}$$

$$\frac{\begin{array}{l}\mathcal{M},\Gamma : [q:\delta],\mathcal{L} \vdash d : \forall q,\hat{q}'.\ \texttt{density}(\\ \quad v\ \{q_0\ \texttt{in}\ \delta_0\texttt{:}\ v'\ [\ q_0\ ]\ \texttt{==}\ q\}| \\ \quad v\ \{q_0\ \texttt{in}\ \delta_0\texttt{:}\ v'\ [\ q_0\ ] < q\},C) \qquad \mathcal{M},\Gamma,\mathcal{L} \vdash v : (\delta_0,\delta) \quad q \notin \text{FV}(C)\end{array}}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{prod}\ q\ \texttt{in}\ \delta\ \texttt{where}\ \phi : d : \forall \hat{q}'.\ \texttt{density}(v|C)}\ \text{DPROD2}$$

Fig. 13. Type rules for probability densities

(a) For every random variable name v such that $v\ [\ q\ ]$ is referenced in ϕ, either $v\ [\ q\ ]$ or v must appear in the random variable set B, and
(b) $\text{DDep}(A,B)$.

Capture-avoiding Substitution. A basic type t can have a quantified variable q subsituted with another term e if e is another quantified variable q' or a random variable V (Figure 6). The substitution is written $t[e/q]$ The semantics of this is to replace every instance of q with e, unless e *captures* q. This means that for a variable definition $v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ \phi\}$, if $q_0 = q$, then no substitution takes place for this term. If $e = q' = q_0$, then e captures q and q_0 is given a new name q_0' that does not conflict with the existing quantified variable names. We note here that renaming q_0 does not change the semantics of the variable set.

4.5 Densities

DMUL. This rule takes two densities and multiplies them together pointwise. It converts one of the conditioned variables in the first density to an output variable. This assumes that the converted variable is an output variable in the second density. The rule also ensures the removed condition variable does not render the constraints invalid.

DDIV. This rule divides the first density by the second. This has the effect of converting output variables in the first density to a conditioned variable in the new density. The moved variables must be output variables in the second density.

DDIV2. This rule allows a developer to convert a joint density over two variables to a density over only one of the variables. The developer must have the density for the variable we want to eliminate, conditioned on the variable we want to keep. We then divide the densities pointwise.

DINT. This rule provides a method for *eliminating* a set of variables from a joint distribution. The resulting density has an integral expression over the eliminated variables.

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d : \forall \hat{q}.\ \texttt{density}(v\ [\ e\ ]|B,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash v\ [\ e\ ] := \texttt{sample}\ d : \forall \hat{q}.\ \texttt{sampler}(v\ [\ e\ ]|B,\phi)}\ \text{SLIFT}$$

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash s_1 : \forall \hat{q}.\ \texttt{sampler}(B|C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash s_2 : \forall \hat{q}.\ \texttt{sampler}(A|B,C,\phi) \quad \text{DDep}(A,C)}{\mathcal{M},\Gamma,\mathcal{L} \vdash s_1\ ;\ s_2 : \forall \hat{q}.\ \texttt{sampler}(A,B|C,\phi)}\ \text{SBIND}$$

$$\frac{\begin{array}{c}\mathcal{M},\Gamma :: [q:\delta],\mathcal{L} \vdash s : \forall q,\hat{q}'.\ \texttt{sampler}(\ v_0\ [\ q\ ],\ldots| \\ v_0\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < q\},\ldots,C) \end{array} \qquad q \notin \text{FV}(C)}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{join}\ q\ \texttt{in}\ \delta : s : \forall \hat{q}'.\ \texttt{sampler}(v_0,\ldots|C)}\ \text{SALL}$$

Fig. 14. Type rules for samplers

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash s : \forall \hat{q}.\ \texttt{sampler}(A|B,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash v : \delta_1,\delta_2 \quad \mathcal{L} \vDash \text{ReachesAll}(s)}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{lift}\ s : \forall \hat{q}.\ \texttt{kernel}(A|B,\phi)}\ \text{KLIFT}$$

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash k_1 : \forall \hat{q}.\ \texttt{kernel}(A|B,C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash k_2 : \forall \hat{q}.\ \texttt{kernel}(B|A,C,\phi) \quad \text{DDep}((A,B),C)}{\mathcal{M},\Gamma,\mathcal{L} \vdash k_1\ ;\ k_2 : \forall \hat{q}.\ \texttt{kernel}(A,B|C,\phi)}\ \text{K2}$$

$$\frac{\mathcal{M},\Gamma :: [q:\delta],\mathcal{L} \vdash k : \forall q,\hat{q}'.\ \texttt{kernel}(v\ [\ q\ ]|v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0\ \texttt{!=}\ q\},C) \quad q \notin \text{FV}(C)}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{join}\ q\ \texttt{in}\ \delta : k : \forall \hat{q}'.\ \texttt{kernel}(v|C)}\ \text{KALL}$$

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash k : \forall \hat{q}.\ \texttt{kernel}(A|B,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{fix}\ k : \forall \hat{q}.\ \texttt{sampler}(A|B,\phi)}\ \text{KFIX}$$

Fig. 15. Type rules for kernels

DPROD. The DPROD rule transforms a density for an individual element of a set into a density over the whole set. It does this by taking the product over all possible instantiations of the quantified variable that defines an individual element in the set. If constraints restrict the size of the set, it only includes those members for which the constraints are satisfied. The DPROD2 rule is similar, but combines densities for disjoint subsets of a random variable set into a density for the whole set.

4.6 Samplers

SLIFT. This rule states that a density for a particular distribution can be used to build a sampler for the same distribution. It samples from one random variable at a time, with the random variable being indexed by an arbitrary quantified variable or constant.

SBIND. This rule allows for reasoning about composed samplers. Its type signature is the same as that of DMUL.

SALL. This rule builds a sampler for an entire set of variables given a sampler for an individual member of that set. It does this by joining them together. The input sampler must sample one variable at a time, and be unconstrained. Any conditions in the input sampler, other than the set of random variables previously sampled, must not depend on the quantified variable.

4.7 Kernels

KLIFT. This rule constructs a kernel out of a sampler. In order to do so, the sampler's output must be a finite random variable. The ReachesAll predicate means that the input sampler must

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash e : \forall \hat{q}.\ \texttt{sampler}(A|B, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{lift}\ e : \forall \hat{q}.\ \texttt{estimator}(A|B, \phi)}\ \text{ELIFT}$$

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash e : \forall \hat{q}.\ \texttt{estimator}(A|B, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash d : \forall \hat{q}.\ \texttt{density}(C|A, B, \phi) \quad \text{DDep}(C, B)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{factor}\ e\ \text{by}\ d : \forall \hat{q}.\ \texttt{estimator}(A|B, C, \phi)}\ \text{EFACT}$$

Fig. 16. Type rules for estimators

have some probability of reaching any part of its output space. The constructed kernel's behavior is the same as the input sampler's.

K2. This rule combines two kernels for conditional densities into a kernel for a joint density. Specifically, the two kernels must be conditioned on each other's output variables. The sampler returned represents the joint density of each kernel's output conditioned on any global conditions both kernels have. Functionally, the rule composes the kernels together using sampler composition.

KALL. This rule is the inductive variant of KERN-COMBINE. Given a kernel for an individual variable conditioned on all other variables in the set, KERN-ALL returns a kernel for the whole set of of variables. The rule uses the `join` syntax to construct the new kernel.

KFIX. This rule transforms a kernel for a distribution into a sampler for the same distribution using the `fix` operator. The `fix` operator has well-defined behavior, but is not implemented exactly in Shuffle. In practice, we use an iterative scheme that approximates its true value.

4.8 Estimators

ELIFT. This rule enables us to build an estimator for a distribution out of a sampler for precisely the same distribution. The estimator calls the sampler to generate a sample, and assigns each sample a weight of 1 regardless of its value.

EFACT. This rule performs *likelihood weighting*. It requires an estimator for a distribution and a density whose conditional variables — A and B — also appear in the estimator. The density's conditions must include the conditional and resulting variables in the estimator. The rule produces an estimator which has the resulting variable of the density as an added condition. It does this by re-weighting the samples from the original estimator according to the density.

4.9 Structural Rules

DEF. The DEF and DEF-IND rules allows a user to introduce a new distribution with a `def` statement. These rules also include type assertions. These assert that the checked type of the internal distribution can be coerced to the user-supplied type included in the `def` statement, possibly appending a sequence of assumptions to the assumption log in the process. There are two rules for the two different kinds of coercions: one for regular coercions, and one for independence coercions.

MODEL. This rule gives a type for all densities established by the model. It serves as the set of axioms for the type system.

INV. The INV rule types invocations of distributions defined with one of the DEF rules. To find the new type, each quantified variable from the definition is substituted with its invocation argument. For each invocation argument, if it has a free quantified variable, then this variable is added to the type. If the invocation argument has no free quantified variables, then the argument is ignored for the purposes of constructing the new quantified variable list $\hat{q}'$. An additional side

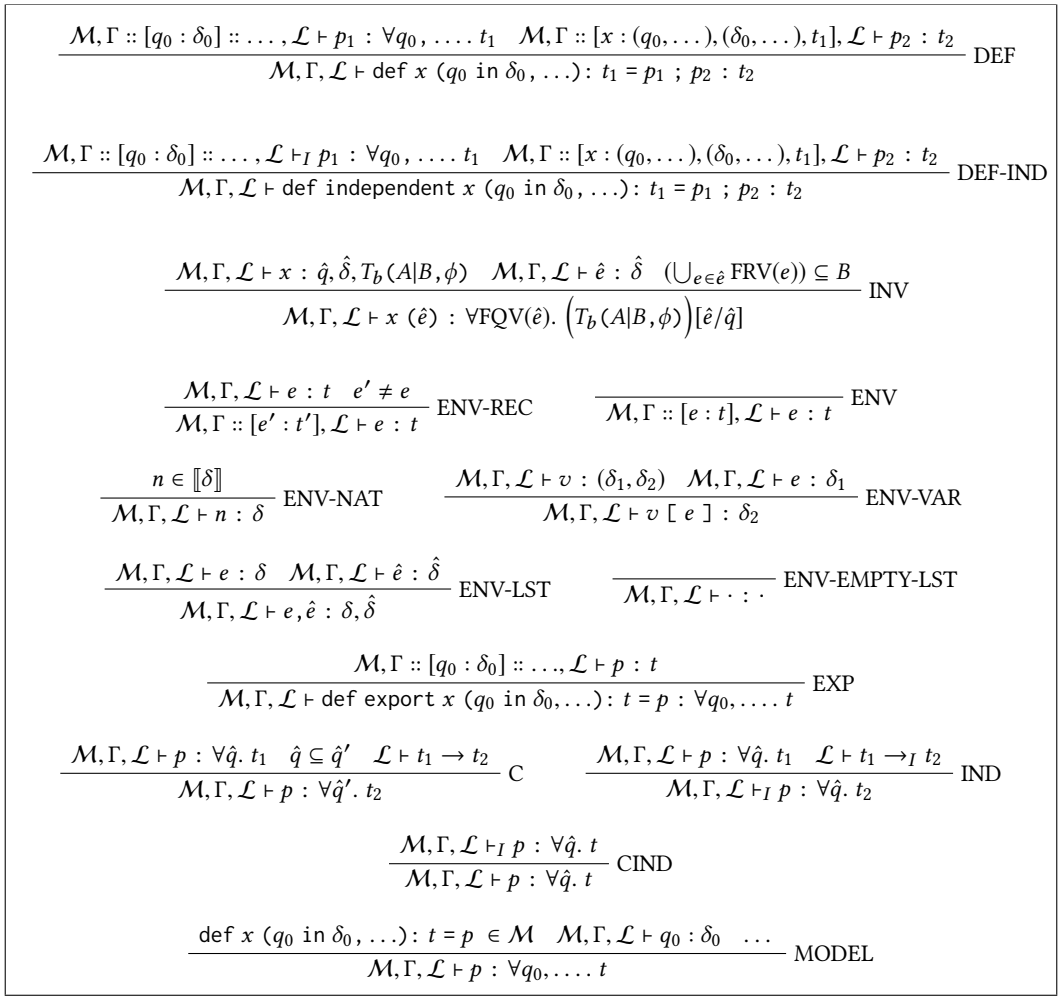

Fig. 17. Type rules for defining and calling distribution objects.

predicate states that every random variable which any of the invocation arguments depend on must be contained in the conditions of the type.

C, IND, and CIND. The C, IND, and CIND rules ensure that coercions introduced by the DEF rules maintain the soundness of the system. specifically, they state that the new type appended to Γ in DEF and DEF-IND, respectively, can be deduced within the type system. Each rule makes appendages to the assumption log as required.

4.10 Coercions

Normal Coercions. Shuffle uses a normal coercion of the form $t_1 \rightarrow t_2$ to assert that a type judgment t_1 implies another type judgment t_2. These require additional predicates which encode logical formulae. Shuffle employs the Z3 theorem prover [12] to verify that these predicates are true. These predicates are:

$$\frac{\mathcal{M} \vDash \phi_2 \Rightarrow A \equiv C \quad \mathcal{M} \vDash \phi_2 \Rightarrow B \equiv D \quad \mathcal{M} \vDash \phi_2 \Rightarrow \phi_1}{\mathcal{M} \vdash T_b(A|B, \phi_1) \rightarrow T_b(C|D, \phi_2)} \text{ LOG-TYPE-C}$$

$$\frac{\mathcal{M}, \mathcal{L} \vDash \phi \Rightarrow A \perp\!\!\!\perp C \mid B \quad \mathcal{M} \vDash \phi \Rightarrow (A \cap C = \emptyset)}{\mathcal{M}, \mathcal{L} \vdash T_b(A|B, \phi) \rightarrow_I T_b(A|B, C, \phi)} \text{ LOG-TYPE-CIND}$$

Fig. 18. Rules for coercion side predicates

- $\mathcal{M} \vDash \phi \Rightarrow A \equiv B$. This predicate states that whenever, ϕ is true, the variable sets A and B must be equivalent. Shuffle checks this by constructing, for each random variable v specified by the model $\mathcal{M}$, the formulas ϕ_{vA} and ϕ_{vB} which specify the set of indices n such that $(v, n) \in [\![A]\!](\sigma)$ or $(v, n) \in [\![A]\!](\sigma)$, respectively. Shuffle then checks whether $\phi \Rightarrow (\phi_{vA} \iff \phi_{vB})$.
- $\mathcal{M} \vDash \phi_1 \Rightarrow \phi_2$. This predicate states that the constraints ϕ_1 imply the constraints ϕ_2.
- $\mathcal{M} \vDash \phi \Rightarrow (A \cap B) = \emptyset$. This predicate determines that the variable groups A and B are disjoint.

The semantics of each predicate are defined as follows

$$\begin{array}{lcl} \mathcal{M} \vDash \phi \Rightarrow A \equiv B & = & \forall \sigma.\ [\![\phi]\!](\sigma) \Rightarrow ([\![A]\!](\sigma) = [\![B]\!](\sigma)) \\ \mathcal{M} \vDash \phi_1 \Rightarrow \phi_2 & = & \forall \sigma.\ [\![\phi_1]\!](\sigma) \Rightarrow [\![\phi_2]\!](\sigma) \\ \mathcal{M} \vDash \phi \Rightarrow (A \cap B) = \emptyset & = & \forall \sigma.\ [\![\phi]\!](\sigma) \Rightarrow [\![A]\!](\sigma) \cap [\![B]\!](\sigma) = \emptyset \end{array}$$

Independence Coercions. Shuffle uses a normal coercion of the form $t_1 \rightarrow_I t_2$ to assert that a type judgment t_1 implies another type judgment t_2. This requires the assumption log $\mathcal{L}$ to entail independence amongst certain variables present in t_1 and t_2.

5 SOUNDNESS

5.1 Preliminaries

Integration by Substitution. The soundness theorems presented in this section rely on a property of integrals known as the *substitution rule*. For measurable functions f and g,

$$\psi(r) = \int_{x \in (-\inf, r]} g(x) \Rightarrow \int_{x \in \psi[S]} f(x) = \int_{x \in S} f(\psi(x)) * g(x)$$

where the notation $\psi[S]$ means the set obtained by mapping the function ψ over S.

Valid Models. A model $\mathcal{M}$ is considered *valid* if for every definition in $\mathcal{M}$ of the form

$$\texttt{def}\ x_i(\ \hat{q}\ )\ :\ t_{bi}\ \ (A_i \,|\, B_i, \phi)\ =\ p_i$$

(1) For every variable (v, n) in the model's space, and for any σ there is exactly one value of i such that $[\![\phi]\!]\sigma \wedge ((v, n) \in [\![A_i]\!]\sigma)$
(2) There exists a strict partial order $\prec$ such that $\forall \sigma, (v_1, n_1) \in [\![A]\!](\sigma), (v_2, n_2) \in [\![B]\!](\sigma).\ (v_1, n_1) \prec (v_2, n_2)$
(3) $\int_{A_i} [\![p_i]\!] = 1$
(4) $\forall i.\text{Valid}(t_{bi}(A_i|B_i, \phi))$
(5) $[\![A_i]\!](\sigma) \neq \perp_\sigma$ and $[\![B_i]\!](\sigma) \neq \perp_\sigma$

Equivalence of Substitution and Environment Mapping. We make use of the following lemma in the proof sketches below. Let p be a Shuffle term that is either a variable set or a constraint.

We have the following equivalences, for any environment σ:

$$\begin{array}{lcl}
[\![p]\!](\sigma[q \mapsto n]) & = & [\![p[n/q]]\!](\sigma) \\
[\![p]\!](\sigma[q_1 \mapsto n]) & = & [\![p[q_2/q_1]]\!](\sigma[q_2 \mapsto n]) \\
[\![p]\!](\sigma[q \mapsto n_1]) & = & [\![v\ [\ n_2\ ]/q]\!](\sigma[(v, n_2) \mapsto n_1]) \\
[\![p]\!](\sigma[q_1 \mapsto n_1]) & = & [\![p[v\ [\ q_2\ ]/q_1]]\!](\sigma[(v, n_2) \mapsto n_1][q_2 \mapsto n_2])
\end{array}$$

5.2 Progress

Progress Theorem. Assuming

(1) $\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall \hat{q}.\ T_b(A|B, \phi)$
(2) $\hat{q} = q_0, q_1, \ldots, q_m$ is a list of quantified variables, and $n_0, n_1, \ldots, n_m$ is a list of values for these variables drawn from the appropriate domains
(3) $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$

and defining $\sigma' = \sigma[q_0 \mapsto n_0][q_1 \mapsto n_1] \ldots [q_m \mapsto n_m]$we will show that

(1) $[\![p]\!](\sigma')$ is never $\perp_\sigma$ or $\perp_0$
(2) $[\![\phi]\!](\sigma') \Rightarrow [\![A]\!](\sigma') \cap [\![B]\!](\sigma') = \emptyset$
(3) Neither $[\![A]\!](\sigma')$ nor $[\![B]\!](\sigma')$ is ever $\perp_\sigma$.
(4) Valid$(\forall q.\ T_b(A|B, \phi))$

Proof Sketch. We proceed by induction on the structure of derivations for types. Specific cases are outlined below.

DMUL. Recall the DMUL rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash d_1 : \forall \hat{q}.\ \texttt{density}(A|B, C, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash d_2 : \forall \hat{q}.\ \texttt{density}(B|C, \phi) \quad \text{DDep}(A, C)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash d_1 * d_2 : \forall \hat{q}.\ \texttt{density}(A, B|C, \phi)}\ \text{DMUL}$$

The density $d_2 * d_2$ must be well-defined (i.e. $d_1 * d_2$ is not $\perp_\sigma$ or $\perp_0$) if d_1 and d_2 are well-defined. If B, C is not $\perp_\sigma$, then the denotations of B and C are disjoint, and, combined with the inductive hypothesis that A's denotation is disjoint from B, C's , this means that the denotation of A, B is disjoint from that of C. Furthermore, A must be disjoint from B so A, B is never $\perp_\sigma$. The type validity follows directly from the inductive assumptions and the fact that $\text{DDep}(A, C) \wedge \text{DDep}(B, C) \Rightarrow \text{DDep}((A, B), C)$.

DDIV and DDIV2. Recall the DDIV and DDIV2 rules

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash d_1 : \forall \hat{q}.\ \texttt{density}(A, B|C, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash d_2 : \forall \hat{q}.\ \texttt{density}(B|C, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash d_1\ /\ d_2 : \forall \hat{q}.\ \texttt{density}(A|B, C, \phi)}\ \text{DDIV}$$

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash d_1 : \forall \hat{q}.\ \texttt{density}(A, B|C, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash d_2 : \forall \hat{q}.\ \texttt{density}(A|B, C, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash d_1\ /\ d_2 : \forall \hat{q}.\ \texttt{density}(B|C, \phi)}\ \text{DDIV2}$$

The density $d_2\ /\ d_2$ must be well-defined if d_1 and d_2 are well-defined and d_2 is nonzero, which must be the case if $\mathcal{J}(\sigma) > 0$ as prescribed in the assumption $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$. The disjointness of A and B, C (or B and C in the case of DIV2) follows from the same reasoning as in DMUL, as does the argument that none of the variable sets' denotation is $\perp_\sigma$. The final conclusion for progress results from the fact that $\text{DDep}((A, B), C) \Rightarrow \text{DDep}(A, (B, C))$ and $\text{DDep}((A, B), C) \Rightarrow \text{DDep}(B, C)$

DINT. Recall the DINT rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash d : \forall \hat{q}.\ \texttt{density}(A, B|C, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{int}\ d\ \texttt{by}\ B : \forall \hat{q}.\ \texttt{density}(A|C, \phi)}\ \text{DINT}$$

For any quantified variable q that B depends on, we can show by structural induction on the type rules that q must appear in $\hat{q}$, which means that assuming d is well-defined `int` d `by` B must be well-defined. The remaining conclusions follow straightforwardly from the properties of A, B.

DPROD. Recall the DPROD and DPROD2 rules

$$\frac{\begin{array}{c}\mathcal{M},\Gamma :: [q:\delta],\mathcal{L} \vdash d : \forall q,\hat{q}'.\ \texttt{density}(v\ \texttt{[}\ q\ \texttt{]}| \\ v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < q\ \texttt{\&\&}\ \phi[q_0/q]\},C,\phi) \qquad q \notin \text{FV}(C)\end{array}}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{prod}\ q\ \texttt{in}\ \delta\ \texttt{where}\ \phi : d : \forall \hat{q}'.\ \texttt{density}(v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ \phi[q_0/q]\}|C)}\ \text{DPROD}$$

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d : \forall q\ \texttt{in}\ \delta,\hat{q}'.\ \texttt{density}(v\ \texttt{[}\ q\ \texttt{]}|v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < q\},C) \quad q \notin FV(C)}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{prod}\ q\ \texttt{in}\ \delta\ \texttt{where}\ \phi : d : \forall \hat{q}'.\ \texttt{density}(v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ \phi[q_0/q]\}|C)}\ \text{DPROD2}$$

For any quantified variable q that ϕ depends on, we can show by structural induction on the type rules that q must appear in $\hat{q}'$, which means that assuming d is well-defined $\texttt{prod}\ q\ \texttt{in}\ \delta\ \texttt{where}\ \phi : d$ must be well-defined. The disjointness and variable well-definedness conclusions follow from the dijointness properties of the "," operator, and the remaining conclusion follows from inlining the definitions of FRV and DDep.

SLIFT. Recall the SLIFT rule

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d : \forall \hat{q}.\ \texttt{density}(v\ \texttt{[}\ e\ \texttt{]}|B,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash v\ \texttt{[}\ e\ \texttt{]}\ \texttt{:=}\ \texttt{sample}\ d : \forall \hat{q}.\ \texttt{sampler}(v\ \texttt{[}\ e\ \texttt{]}|B,\phi)}\ \text{SLIFT}$$

For any quantified variable q that $v\ \texttt{[}\ e\ \texttt{]}$ depends on, we can show by structural induction on the type rules that q must appear in $\hat{q}$, which means that assuming d is well-defined $v\ \texttt{[}\ e\ \texttt{]}\ \texttt{:=}\ \texttt{sample}\ d$ must be well-defined. The remaining conclusions follow straightforwardly because the parameters of the type are the same.

SBIND. Recall the SBIND rule

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash s_1 : \forall \hat{q}.\ \texttt{sampler}(B|C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash s_2 : \forall \hat{q}.\ \texttt{sampler}(A|B,C,\phi) \quad \text{DDep}(A,C)}{\mathcal{M},\Gamma,\mathcal{L} \vdash s_1\ ;\ s_2 : \forall \hat{q}.\ \texttt{sampler}(A,B|C,\phi)}\ \text{SBIND}$$

The sampler $s_1\ ;\ s_2$ is well-defined if s_1 and s_2 are well-defined. The remaining conclusions follow from the same reasoning as that of DMUL.

SALL. Recall the SALL rule

$$\frac{\begin{array}{c}\mathcal{M},\Gamma :: [q:\delta],\mathcal{L} \vdash s : \forall q,\hat{q}'.\ \texttt{sampler(}\ v_0\ \texttt{[}\ q\ \texttt{]},\ldots| \\ v_0\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < q\},\ldots,C) \qquad q \notin \text{FV}(C)\end{array}}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{join}\ q\ \texttt{in}\ \delta : s : \forall \hat{q}'.\ \texttt{sampler}(v_0,\ldots|C)}\ \text{SALL}$$

The sampler $\texttt{join}\ q\ \texttt{in}\ \delta : s$ is well-defined if s is well-defined. The disjointness and variable well-definedness conclusions follow from the disjointness properties of the "," operator, and the remaining conclusion follows from properties of FRV and DDep.

KLIFT. Recall the KLIFT rule

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash s : \forall \hat{q}.\ \texttt{sampler}(A|B,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash v : \delta_1,\delta_2 \quad \mathcal{L} \vDash \text{ReachesAll}(s)}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{lift}\ s : \forall \hat{q}.\ \texttt{kernel}(A|B,\phi)}\ \text{KLIFT}$$

The kernel $\texttt{lift}\ s$ is well-defined if s is well-defined. The remaining conclusions follow straightforwardly because the parameters of the type are the same.

K2. Recall the K2 rule

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash k_1 : \forall \hat{q}.\ \texttt{kernel}(A|B,C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash k_2 : \forall \hat{q}.\ \texttt{kernel}(B|A,C,\phi) \quad \text{DDep}((A,B),C)}{\mathcal{M},\Gamma,\mathcal{L} \vdash k_1\ ;\ k_2 : \forall \hat{q}.\ \texttt{kernel}(A,B|C,\phi)}\ \text{K2}$$

The kernel $k_1\ ;\ k_2$ is well-defined if k_1 and k_2 are well-defined. Using the semantics of the "," operator, we show that if $[\![A]\!](\sigma') \cap [\![B,C]\!](\sigma') = \emptyset$ and $[\![B]\!](\sigma') \cap [\![A,C]\!](\sigma') = \emptyset$, then $[\![A,B]\!](\sigma') \cap [\![C]\!](\sigma') = \emptyset$ and $[\![A]\!](\sigma') \cap [\![B]\!](\sigma') = \emptyset$. The final conclusion follows from the definition of Valid and inductive validity assumptions.

KALL. Recall the KALL rule

$$\frac{\mathcal{M}, \Gamma :: [q : \delta], \mathcal{L} \vdash k \ : \ \forall q, \hat{q}'.\ \texttt{kernel}(v\ [\ q\ ]|v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0\ \texttt{!=}\ q\}, C) \quad q \notin \text{FV}(C)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{join}\ q\ \texttt{in}\ \delta\ :\ k\ :\ \forall \hat{q}'.\ \texttt{kernel}(v|C)}\ \text{KALL}$$

The kernel `join` q `in` δ : k is well-defined if k is well-defined. The disjointness and variable well-definedness conclusions follow from the disjointness requirement of the "," operator, and the remaining conclusion follows from inlining the definitions of FRV and DDep.

KFIX. Recall the KFIX rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash k \ : \ \forall \hat{q}.\ \texttt{kernel}(A|B, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{fix}\ k\ :\ \forall \hat{q}.\ \texttt{sampler}(A|B, \phi)}\ \text{KFIX}$$

Since the semantics of `fix` k is defined declaratively, we must show that a solution exists. Due to the first property of kernel soundness, we can apply a theorem from **(author?)** [5] which states that the fixed point can be approximated to arbitrary precision with an iterative algorithm. This means that a solution must exist and furthermore, the solution is unique. The remaining conclusions follow straightforwardly because the parameters of the type are the same.

ELIFT. Recall the ELIFT rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash e \ : \ \forall \hat{q}.\ \texttt{sampler}(A|B, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{lift}\ e\ :\ \forall \hat{q}.\ \texttt{estimator}(A|B, \phi)}\ \text{ELIFT}$$

The estimator `lift` s is well-defined if s is well-defined. The remaining conclusions follow straightforwardly because the parameters of the type are the same.

EFACT. Recall EFACT rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash e \ : \ \forall \hat{q}.\ \texttt{estimator}(A|B, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash d \ : \ \forall \hat{q}.\ \texttt{density}(C|A, B, \phi) \quad \text{DDep}(C, B)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{factor}\ e\ \texttt{by}\ d\ :\ \forall \hat{q}.\ \texttt{estimator}(A|B, C, \phi)}\ \text{EFACT}$$

The estimator `factor` e `by` d is well-defined if e and d are well-defined. According to the semantics of the ",", if $[\![A]\!](\sigma') \cap [\![B]\!](\sigma') = \emptyset$ and $[\![C]\!](\sigma') \cap [\![A, B]\!](\sigma') = \emptyset$, then $[\![A]\!] \cap [\![B, C]\!] = \emptyset$ and $[\![B]\!] \cap [\![C]\!] = \emptyset$. Finally, the validity of the type follows from the fact that $\text{DDep}(A, B) \wedge \text{DDep}(C, B) \implies \text{DDep}(A, (B, C))$.

DEF. Recall the DEF and DEF-IND rules

$$\frac{\mathcal{M}, \Gamma :: [q_0 : \delta_0] :: \ldots, \mathcal{L} \vdash p_1 \ : \ \forall q_0, \ldots.\ t_1 \quad \mathcal{M}, \Gamma :: [x : (q_0, \ldots), (\delta_0, \ldots), t_1], \mathcal{L} \vdash p_2 \ : \ t_2}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{def}\ x\ (q_0\ \texttt{in}\ \delta_0, \ldots)\texttt{:}\ t_1 = p_1\ ;\ p_2\ :\ t_2}\ \text{DEF}$$

$$\frac{\mathcal{M}, \Gamma :: [q_0 : \delta_0] :: \ldots, \mathcal{L} \vdash_I p_1 \ : \ \forall q_0, \ldots.\ t_1 \quad \mathcal{M}, \Gamma :: [x : (q_0, \ldots), (\delta_0, \ldots), t_1], \mathcal{L} \vdash p_2 \ : \ t_2}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{def independent}\ x\ (q_0\ \texttt{in}\ \delta_0, \ldots)\texttt{:}\ t_1 = p_1\ ;\ p_2\ :\ t_2}\ \text{DEF-IND}$$

Taking $\hat{q} = q_0, \ldots$ and $\hat{\delta} = \delta_0, \ldots$, and assuming p_2 is well-defined under the environment $\sigma[x \mapsto (\hat{q}, p_1)]$, the program `def` x $(q_0$ `in` $\delta_0, \ldots)$`:` $t_1 = p_1$; p_2 is well-defined. The program p_2 must be well-defined by inductive assumption because $\sigma \vDash \mathcal{M}, \Gamma, \mathcal{L} \implies \sigma[x \mapsto (\hat{q}, p_1)] \vDash \mathcal{M}, \Gamma :: [x : \hat{q}, \hat{\delta}, t_1], \mathcal{L}$. The remaining conclusions follow from the fact that the side predicate $\mathcal{L} \vdash t_a \rightarrow t_1$ enforces that the variable sets in t_a and t_1 are the same, and the predicate $\mathcal{L} \vdash t_a \rightarrow_I t_1$ enforces that all modifications to the variables preserve disjointness.

INV. Recall the INV rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash x \ : \ \hat{q}, \hat{\delta}, T_b(A|B, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash \hat{e} \ : \ \hat{\delta} \quad (\bigcup_{e \in \hat{e}} \text{FRV}(e)) \subseteq B}{\mathcal{M}, \Gamma, \mathcal{L} \vdash x\ (\hat{e})\ :\ \forall \text{FQV}(\hat{e}).\ \Big(T_b(A|B, \phi)\Big)[\hat{e}/\hat{q}]}\ \text{INV}$$

From the assumption that $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$, we know that $\sigma(x) = (q_0 \texttt{ in } \delta_0, \dots), p$ is defined, which means that $x\ (e_0, \dots)$ must be well-defined. To prove the disjointness condition, we apply the environment-substitution lemma. Also because of the assumption that $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$, we know that $\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall q_0 \texttt{ in } \delta_0, \dots.\ T_b(A|B, \phi)$, which means our inductive assumption yields $\text{Valid}(T_b(A|B, \phi))$. Combined with the assumption that $(\text{FRV}(e_0) \cup \dots) \subseteq B$, this means that $\text{DDep}(A[e_0/q_0][\dots], B[e_0/q_0][\dots])$, $\text{DDep}(B[e_0/q_0][\dots], B[e_0/q_0][\dots])$, and furthermore $\text{FRV}(\phi[e_0/q_0][\dots]) \subseteq B$. This means that $\text{Valid}(T_b(A|B, \phi))$ is true.

MODEL. See the model validity assumptions above.

C and CIND. Recall the C, IND, and CIND rules

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall \hat{q}.\ t_1 \quad \hat{q} \subseteq \hat{q}' \quad \mathcal{L} \vdash t_1 \to t_2}{\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall \hat{q}'.\ t_2}\ \text{C} \qquad \frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall \hat{q}.\ t_1 \quad \mathcal{L} \vdash t_1 \to_I t_2}{\mathcal{M}, \Gamma, \mathcal{L} \vdash_I p : \forall \hat{q}.\ t_2}\ \text{IND}$$

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash_I p : \forall \hat{q}.\ t}{\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall \hat{q}.\ t}\ \text{CIND}$$

The program p must be well defined by assumption. The remaining conclusions follow from the fact that the side predicate $\mathcal{L} \vdash t_a \to t_1$ enforces that the variable sets in t_a and t_1 are the same, and the predicate $\mathcal{L} \vdash t_a \to_I t_1$ enforces that all modifications to the variables preserve disjointness.

5.3 Densities

Density Soundness Theorem. A density is considered sound if, under any substitution of the relevant quantified variables, the density evaluates to the appropriate distribution from the model. Specifically, given the assumptions

(1) $\mathcal{M}, \Gamma, \mathcal{L} \vdash d : \forall \hat{q}.\ \texttt{density}(A|B, \phi)$
(2) $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$

it must be true that, for any environment $\sigma' = \sigma[q_0 \mapsto n_0][q_1 \mapsto n_1] \dots [q_m \mapsto n_m]$ that binds values for the set of quantified variables $\hat{q} = q_0 \texttt{ in } \delta_0, q_1 \texttt{ in } \delta_1, \dots, q_m \texttt{ in } \delta_m$,

$$[\![\phi]\!](\sigma') \Rightarrow [\![d]\!](\sigma') = \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma'))(\sigma')$$

Proof Sketch. Proceed by induction on the derivations. Specific rules are outlined below:

MODEL. Recall the MODEL rule

$$\frac{\texttt{def } x\ (q_0 \texttt{ in } \delta_0, \dots) \texttt{: } t = p \in \mathcal{M} \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash q_0 : \delta_0 \quad \dots}{\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall q_0, \dots.\ t}\ \text{MODEL}$$

From the definition of $\mathcal{J}$, we know that

$$\mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma')) = \frac{\int_{\mathcal{V} - ([\![A]\!](\sigma') \cup [\![B]\!](\sigma'))} \prod_i \begin{cases} [\![d_i]\!](\sigma') & [\![\phi_i]\!](\sigma') \\ 1 & \textbf{else} \end{cases}}{\int_{\mathcal{V} - [\![B]\!](\sigma')} \prod_i \begin{cases} [\![d_i]\!](\sigma') & [\![\phi_i]\!](\sigma') \\ 1 & \textbf{else} \end{cases}}$$

There exists some i^* such that $d_{i^*} = p$ and $\phi_{i^*} = \phi$. Furthermore, because A and B are disjoint, $[\![p]\!](\sigma')$ does not depend on any variables outside of A and B, and $\int_{[\![A]\!](\sigma')} [\![p]\!](\sigma') = 1$, we can

simplify the above equation to

$$\mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma')) = \begin{cases} [\![p]\!](\sigma') & [\![\phi]\!] \\ 1 & \textbf{else} \end{cases} * \frac{\int_{\mathcal{V}-([\![A]\!](\sigma')\cup[\![B]\!](\sigma')} \prod_{i\neq i^*} \begin{cases} [\![d_i]\!](\sigma') & [\![\phi_i]\!](\sigma') \\ 1 & \textbf{else} \end{cases}}{\int_{\mathcal{V}-[\![B]\!](\sigma')} \prod_{i\neq i^*} \begin{cases} [\![d_i]\!](\sigma') & [\![\phi_i]\!](\sigma') \\ 1 & \textbf{else} \end{cases}}$$

Given that the product in the numerator is independent of the values of variables in $[\![A]\!](\sigma')$, we can further simplify this equation to

$$\mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma')) = \begin{cases} [\![p]\!](\sigma') & [\![\phi]\!] \\ 1 & \textbf{else} \end{cases}$$

which means $[\![\phi]\!](\sigma') \Rightarrow \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma')) = [\![p]\!](\sigma')$ as required.

DMUL. Recall the DMUL rule

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 : \forall \hat{q}.\ \texttt{density}(A|B,C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash d_2 : \forall \hat{q}.\ \texttt{density}(B|C,\phi) \quad \text{DDep}(A,C)}{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 * d_2 : \forall \hat{q}.\ \texttt{density}(A,B|C,\phi)}\ \text{DMUL}$$

The basic premise of the rule is based on the following property of $\mathcal{J}$

$$\mathcal{J}(S_1|S_2,S_3)\mathcal{J}(S_2|S_3) = \left(\frac{\int_{\mathcal{V}-(S_1\cup S_2\cup S_3)}\mathcal{J}}{\int_{\mathcal{V}-(S_2\cup S_3)}\mathcal{J}}\right)\left(\frac{\int_{\mathcal{V}-(S_2\cup S_3)}\mathcal{J}}{\int_{\mathcal{V}-S_3}\mathcal{J}}\right) = \frac{\int_{\mathcal{V}-(S_1\cup S_2\cup S_3)}\mathcal{J}}{\int_{\mathcal{V}-S_3}\mathcal{J}} = \mathcal{J}(S_1,S_2|S_3)$$

DDIV. Recall the DDIV rule

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 : \forall \hat{q}.\ \texttt{density}(A,B|C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash d_2 : \forall \hat{q}.\ \texttt{density}(B|C,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 \ / \ d_2 : \forall \hat{q}.\ \texttt{density}(A|B,C,\phi)}\ \text{DDIV}$$

Applying the identity from DMUL in reverse, we see that

$$\mathcal{J}(S_1|S_2,S_3)\mathcal{J}(S_2|S_3) = \mathcal{J}(S_1,S_2|S_3) \Rightarrow \mathcal{J}(S_1|S_2,S_3) = \frac{\mathcal{J}(S_1,S_2|S_3)}{\mathcal{J}(S_2|S_3)}$$

which justifies the basic rule construct.

DDIV2. Recall the DDIV2 rule

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 : \forall \hat{q}.\ \texttt{density}(A,B|C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash d_2 : \forall \hat{q}.\ \texttt{density}(A|B,C,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 \ / \ d_2 : \forall \hat{q}.\ \texttt{density}(B|C,\phi)}\ \text{DDIV2}$$

From the identity in DMUL, we can deduce that

$$\mathcal{J}(S_1|S_2,S_3)\mathcal{J}(S_2|S_3) = \mathcal{J}(S_1,S_2|S_3) \Rightarrow \mathcal{J}(S_2|S_3) = \frac{\mathcal{J}(S_1,S_2|S_3)}{\mathcal{J}(S_1|S_2,S_3)}$$

which justifies the basic rule construct.

DINT. Recall the DINT rule

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d : \forall \hat{q}.\ \texttt{density}(A,B|C,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{int}\ d\ \texttt{by}\ B : \forall \hat{q}.\ \texttt{density}(A|C,\phi)}\ \text{DINT}$$

This rule relies on the following simplification of $\mathcal{J}$:

$$\int_{S_1}\mathcal{J}(S_1,S_2|S_3) = \int_{S_1}\frac{\int_{\mathcal{V}-(S_1\cup S_2\cup S_3)}\mathcal{J}}{\int_{\mathcal{V}-S_3}\mathcal{J}} = \frac{\int_{S_1}\int_{\mathcal{V}-(S_1\cup S_2\cup S_3)}\mathcal{J}}{\int_{\mathcal{V}-S_3}\mathcal{J}} = \frac{\int_{\mathcal{V}-(S_2\cup S_3)}\mathcal{J}}{\int_{\mathcal{V}-S_3}\mathcal{J}} = \mathcal{J}(S_2|S_3)$$

The second step above is justified by the fact that $S_1 \cap S_3 = \emptyset$, so the denominator is a constant with respect to the outer integral.

DPROD. Recall the DPROD rule

$$\frac{\begin{array}{c}\mathcal{M},\Gamma :: [q:\delta],\mathcal{L} \vdash d\ :\ \forall q,\hat{q}'.\ \texttt{density}(v\ [\ q\ ]| \\ v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < q\ \texttt{\&\&}\ \phi[q_0/q]\},C,\phi) \qquad\qquad q \notin \text{FV}(C)\end{array}}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{prod}\ q\ \texttt{in}\ \delta\ \texttt{where}\ \phi\ :\ d\ :\ \forall\hat{q}'.\ \texttt{density}(v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ \phi[q_0/q]\}|C)}\ \text{DPROD}$$

In this rule, q is a value from a domain δ consisting an ordered set of integers $[n_1, n_2]$. Define the set $\delta_{n_2'} = [n_1, n_2']$ for $n_2' \in [n_1, n_2]$ We proceed by induction over n_2'. Our inductive hypothesis is that

$\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{prod}\ q\ \texttt{in}\ \delta_{n_2'}\ \texttt{where}\ \phi\ :\ d\ :\ \forall\hat{q}'.\ \texttt{density}(v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 <= n_2'\ \texttt{\&\&}\ \phi[q_0/q]\}|C,\phi)$ is sound for any $n_2' \in [n_1, n_2]$ with $n_2' = n_1$ as the base case.

- **Base case.** If the constraints are satisfied, then the definition of semantics for densities defines that the product over a set of size 1 simply returns the inner density d. We also use the equivalences between variable sets

  $$v\ \{q_0\ \texttt{in}\ D\texttt{:}\ q_0 <= n_1\} = v\ [\ n_1\ ], \quad v\ \{q_0\ \texttt{in}\ D\texttt{:}\ f\ [\ q_0\ ] < n_1\} = \emptyset$$

  to justify the type judgment. If the constraints are not satisfied, then the hypothesis still holds because $\mathcal{J}(\emptyset|S_2) = 1$ by definition.
- **Inductive step.** Assuming the inductive hypothesis is true for $n_2' - 1$, we will show it is true for n_2'. First consider the case where ϕ is true for $q_0 = n_2'$. In this case, we show that the following programs are semantically equivalent:

  $$\texttt{prod}\ q\ \texttt{in}\ \delta_{n_2'}\ \texttt{where}\ \phi\ :\ d$$
  $$\texttt{def}\ x_I\ (q\ \texttt{in}\ \delta)\texttt{:}\ t = d\ ;\ x\ (n_2')\ *\ \texttt{prod}\ q\ \texttt{in}\ \delta_{n_2'-1}\ \texttt{where}\ \phi\ :\ d$$

  We will now give an appropriate type to the latter program, and show that this type is sound. We note that choosing t as

  $$\texttt{density}(v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 == q\}|v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < q\ \texttt{\&\&}\ \phi[q_0/q]\},C,\phi)$$

  is always a valid coercion. The inductive hypothesis is equivalent to stating that the following type is sound for the product in the above program:

  $$\forall\hat{q}'.\ \texttt{density}(v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < n_2'\ \texttt{\&\&}\ \phi[q_0/q]\}|C,\phi)$$

  Using the soundness of the DEF, CALL-LIT, and DMUL rules, we show that

  $$\forall\hat{q}'.\ \texttt{density}(v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 == q\},v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < n_2'\ \texttt{\&\&}\ \phi[q_0/q]\}|C,\phi)$$

  is a sound type for this program. Using the semantics of variables sets, we show that this type is semantically equivalent to the type

  $$\forall\hat{q}'.\ \texttt{density}(v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0\} <= n_2'\ \texttt{\&\&}\ \phi|C,\phi)$$

  completing the proof.

  If ϕ is not true for $q_0 = n_2'$, then the value of the product remains unchanged, as do the variable sets in the type. Therefore, the type is sound in this case as well.

The conclusion of the rule follows from choosing $n_2' = n_2$, since $\delta_{n_2} = \delta$. An analogous proof holds for the DPROD2 rule.

5.4 Samplers

Sampler Soundness Theorem. For a sampler to be sound, it must be able to properly compute the expectation of any positive function f. A sampler computes the expectation by feeding its output into f, and this must be equal to the model expectation defined by multiplying with the appropriate distribution and integrating. Specifically, under the assumptions

(1) $\mathcal{M},\Gamma,\mathcal{L} \vdash s\ :\ \forall\hat{q}.\ \texttt{sampler}(A|B,\phi)$

(2) $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$

it must be true that, for any environment $\sigma' = \sigma[q_0 \mapsto n_0][q_1 \mapsto n_1] \ldots [q_m \mapsto n_m]$ that binds values for the set of quantified variables $\hat{q} = q_0 \texttt{ in } \delta_0, q_1 \texttt{ in } \delta_1, \ldots, q_m \texttt{ in } \delta_m$, and any $f \in \Sigma_{rv} \to \mathbb{R}^+$

$$[\![\phi]\!](\sigma') \Rightarrow \int_{\text{sr}} f([\![s]\!](\sigma', \text{sr})) = \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma') | [\![B]\!](\sigma'))(\sigma')$$

Proof sketch. We proceed by structural induction on the rules which may produce samplers. Individual cases are outlined below.

SLIFT. Recall the SLIFT rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash d : \forall \hat{q}.\ \texttt{density}(v\ [\ e\ ] | B, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash v\ [\ e\ ] := \texttt{sample}\ d : \forall \hat{q}.\ \texttt{sampler}(v\ [\ e\ ] | B, \phi)}\ \text{SLIFT}$$

In the discrete case, notice that the size of the set

$$\{\text{sr} | [\![d := \texttt{sample}\ ]\!](\sigma'[(v, e) \mapsto n])\}$$

is exactly $[\![d]\!]((v, e) \mapsto n)$, and furthermore each such set is disjoint for different values of a. Therefore, we can write the integral over sr as a linear combination over these different cases:

$$\int_{\text{sr}} f([\![s]\!](\sigma)) = \sum_n [\![d]\!](\sigma'[(v, e) \mapsto n]) * f(n)$$

According to the assumptions in the rule, this is equal to $\int_{[\![} A]\!](\sigma') f * \mathcal{J}([\![A]\!](\sigma') | [\![B]\!](\sigma')$ as required.

In the continuous case, the sample returned must be a real value r such that

$$\text{sr} = \int_{x \in [-\infty, r]} [\![d]\!](v, [\![p]\!](\sigma') \mapsto x) = g(r) \Rightarrow \int_{\text{sr}} f([\![s]\!](\sigma', \text{sr})) = \int_{\text{sr}} f(g^{-1}(\text{sr}))$$

Substituting g for ψ and $f \circ g^{-1}$ for f in the definition for the substitution rule, we see that

$$\int_{\text{sr}} f([\![s]\!](\sigma', \text{sr})) = \int_{[\![A]\!](\sigma')} f(\sigma') * [\![d]\!](\sigma') = \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma') | [\![B]\!](\sigma'))(\sigma')$$

where the above step is due to the soundness theorem for densities.

SBIND. Recall the SBIND rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash s_1 : \forall \hat{q}.\ \texttt{sampler}(B | C, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash s_2 : \forall \hat{q}.\ \texttt{sampler}(A | B, C, \phi) \quad \text{DDep}(A, C)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash s_1\ ;\ s_2 : \forall \hat{q}.\ \texttt{sampler}(A, B | C, \phi)}\ \text{SBIND}$$

First, we apply the soundness assumption for s_1 to find the expectation of the function $s_1 \circ f$. Then, we use the assumption soundness assumption on s_2. This yields the equation

$$\int_{\text{sr}^0, \text{sr}^1} f([\![s_2]\!]([\![s_1]\!](\sigma', \text{sr}^0), \text{sr}^1))$$

$$= \int_{[\![A]\!](\sigma'), [\![B]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma') | [\![B, C]\!](\sigma'))(\sigma') * \mathcal{J}([\![B]\!](\sigma') | [\![C]\!](\sigma'))(\sigma')$$

Using the identity from DMUL, we can simplify this to

$$= \int_{[\![A]\!](\sigma'), [\![B]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A, B]\!](\sigma') | [\![C]\!](\sigma'))(\sigma')$$

SALL. Recall the SALL rule

$$\frac{\mathcal{M}, \Gamma :: [q : \delta], \mathcal{L} \vdash s \ : \ \forall q, \hat{q}'.\ \texttt{sampler(}\ v_0\ \texttt{[}\ q\ \texttt{]}, \ldots | \\ \quad v_0\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < q\}, \ldots, C) \qquad q \notin \mathrm{FV}(C)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{join}\ q\ \texttt{in}\ \delta\ :\ s\ :\ \forall \hat{q}'.\ \texttt{sampler(}v_0, \ldots | C)}\ \text{SALL}$$

In this rule, δ represents an ordered set of integers $[n_1, n_2]$. Define the set $\delta_{n_2'} = [n_1, n_2']$ for $n_2' \in [n_1, n_2]$ We proceed by induction over n_2'. Our inductive hypothesis is that

$$\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{join}\ q\ \texttt{in}\ \delta_{n_2'}\ :\ s\ :\ \forall \hat{q}'.\ \texttt{sampler(}v_0\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 <= n_2'\}, \ldots | C, \phi)$$

is sound for any $n_2' \in [n_1, n_2]$ with $n_2' = n_1$ as the base case.

- **Base case.** The definition of semantics for samplers defines that the product over a set of size 1 simply returns the inner sampler s. We also note the following equivalences between variable sets

  $$\forall i. \quad v_i\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 <= n_1\} = v_i\ \texttt{[}\ n_1\ \texttt{]}, \quad v_i\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < n_1\} = \emptyset$$

  justifying the type judgment.
- **Inductive step.** Assuming the inductive hypothesis is true for $n_2' - 1$, we will show it is true for n_2'. We first note that the following programs are semantically equivalent:

  $$\texttt{join}\ q\ \texttt{in}\ \delta_{n_2'}\ :\ s$$
  $$\texttt{def}\ x_I\ (q\ \texttt{in}\ \delta)\texttt{:}\ t = s\ \texttt{;}\ (\texttt{join}\ q\ \texttt{in}\ \delta_{n_2'-1}\ :\ s)\ \texttt{;}\ x\ (n_2')$$

  We will now give an appropriate type to the latter program, and show that this type is sound. We note that choosing t as

  $$\texttt{sampler(}v_0\ \texttt{[}\ q\ \texttt{]}, \ldots | v_0\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < q]\}, \ldots, C, \texttt{true)}$$

  is always a valid coercion. The inductive hypothesis is equivalent to stating that the following type is sound for the product in the above program:

  $$\forall \hat{q}'.\ \texttt{sampler(}v_0\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < n_2'\}, \ldots | C, \phi)$$

  Using the soundness of the DEF, CALL, and SBIND rules, we show that

  $$\forall \hat{q}'.\ \texttt{sampler(}v_0\ \texttt{[}\ n_2'\ \texttt{]}, \ldots, v_0\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 < n_2'\}, \ldots | C, \texttt{true)}$$

  is a sound type for this program. Using the semantics of variables sets, we show that this type is semantically equivalent to the type

  $$\forall \hat{q}'.\ \texttt{sampler(}v_0\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 <= n_2'\}, \ldots | C, \texttt{true)}$$

  completing the proof.

The conclusion of the rule follows from choosing $n_2' = n_2$, since $\delta_{n_2} = \delta$

5.5 Kernels

Kernel Soundness Theorem. Kernels require two properties in order two be sound. The first property states that the kernel can reach any state a sampler for the same distribution could reach, and the second property says that a sampler for the distribution is invariant under the kernel. Specifically, assuming

(1) $\mathcal{M}, \Gamma, \mathcal{L} \vdash k\ :\ \forall \hat{q}.\ \texttt{kernel(}A | B, \phi\texttt{)}$
(2) $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$

it must be true that, for any environment $\sigma' = \sigma[q_0 \mapsto n_0][q_1 \mapsto n_1] \ldots [q_m \mapsto n_m]$ that binds values for the set of quantified variables $\hat{q} = q_0 \texttt{ in } \delta_0, q_1 \texttt{ in } \delta_1, \ldots, q_m \texttt{ in } \delta_m$, any $f \in \Sigma_{rv} \to \mathbb{R}^+$, and any s such that

$$\int_{\text{sr}} f(s(\sigma', \text{sr})) = \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma') | [\![B]\!](\sigma'))(\sigma')$$

it must be true that there exists an $\epsilon > 0$ such that

(1) $\int_{\text{sr}} f(k(\sigma', \text{sr})) > \epsilon \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma') | [\![B]\!](\sigma'))(\sigma')$

(2) $\int_{\text{sr}^0, \text{sr}^1} f(k(s(\sigma', \text{sr}^0), \text{sr}^1)) = \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma') | [\![B]\!](\sigma'))(\sigma')$

Proof sketch. we proceed by structural induction on the rule derivations for kernels. The specific cases are outlined below.

KLIFT. Recall the KLIFT rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash s : \forall \hat{q}.\ \texttt{sampler}(A|B, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash v : \delta_1, \delta_2 \quad \mathcal{L} \vDash \text{ReachesAll}(s)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{lift } s : \forall \hat{q}.\ \texttt{kernel}(A|B, \phi)} \text{ KLIFT}$$

To prove the first property, we can choose $\epsilon = 1$ and inline the definition of the sampler. For the second condition, we must deduce that if s is a sampler, then $\texttt{fix } s \equiv s$. In otherwords, for any measurable function f over the output space, the equation

$$\int_{\text{sr}} f(\text{fix}(\sigma, \text{sr})) = \int_{\text{sr}, \text{sr}'} f(\text{fix}(s(\sigma, \text{sr}'), \text{sr}))$$

is satisfied for fix = s. To see this, we inline the definition for a sampler, which reduces the second property to the equation

$$\int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma') | [\![B]\!](\sigma'))(\sigma')$$

$$= \int_{[\![A]\!](\sigma')} \mathcal{J}([\![A]\!](\sigma') | [\![B]\!](\sigma'))(\sigma') \int_{[\![A]\!](\sigma')} f(\sigma') \mathcal{J}([\![A]\!](\sigma') | [\![B]\!](\sigma'))(\sigma')$$

which must hold because $\int_{S_1} \mathcal{J}(S_1 | S_2) = 1$.

K2. Recall the K2 rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash k_1 : \forall \hat{q}.\ \texttt{kernel}(A|B, C, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash k_2 : \forall \hat{q}.\ \texttt{kernel}(B|A, C, \phi) \quad \text{DDep}((A, B), C)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash k_1 \ ; \ k_2 : \forall \hat{q}.\ \texttt{kernel}(A, B|C, \phi)} \text{ K2}$$

First, we will show that k_1 is invariant for the distribution $A, B|C$. This means that k_1 satisfies the *second* property for $\mathcal{M}, \Gamma, \mathcal{L} \vdash k_1 : \forall \hat{q}.\ \texttt{kernel}(A, B|C, \phi)$ to be sound, even though k_1 does not satisfy the first property. This is true because, for a sampler s such that

$$\int_{\text{sr}} f(s(\sigma', \text{sr})) = \int_{[\![A, B]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A, B]\!](\sigma') | [\![C]\!](\sigma'))(\sigma')$$

because of the property from DMUL, we must have that

$$\int_{\text{sr}} f(s(\sigma', \text{sr})) = \int_{[\![A, B]\!](\sigma')} (f(\sigma') * \mathcal{J}([\![B]\!](\sigma') | [\![C]\!](\sigma'))(\sigma')) * \mathcal{J}([\![A]\!](\sigma') | [\![B, C]\!](\sigma'))(\sigma')$$

Substituting in $f * \mathcal{J}([\![B]\!](\sigma') | [\![C]\!](\sigma'))$ for f in the soundness assumption for k_1, we can deduce that

$$\int_{\text{sr}^0, \text{sr}^1} f(k_1(s(\sigma', \text{sr}^0), \text{sr}^1)) = \int_{[\![A, B]\!](\sigma')} (f(\sigma') * \mathcal{J}([\![B]\!](\sigma') | [\![C]\!](\sigma'))(\sigma')) * \mathcal{J}([\![A]\!](\sigma') | [\![B, C]\!](\sigma'))(\sigma')$$

Using again the identity from DMUL, this means that

$$\int_{\mathrm{sr}^0,\mathrm{sr}^1} f(k_1(s(\sigma',\mathrm{sr}^0),\mathrm{sr}^1)) = \int_{[\![A,B]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A,B]\!](\sigma')|[\![C]\!](\sigma'))(\sigma')$$

completing the proof of the invariance property of k_1. By a similar logic, k_2 is also invariant for the distribution $A,B|C$. This means that $k_1 \ ; \ k_2$ is invariant for the distribution $A,B|C$. This proves the second property of kernel soundness. For the first property, we note that the ReachesAll condition applies transitively to any kernel that can be generated with the kernel rules.

KALL. Recall the KALL rule

$$\frac{\mathcal{M},\Gamma :: [q:\delta],\mathcal{L} \vdash k : \forall q,\hat{q}'.\ \texttt{kernel}(v\ \texttt{[}\ q\ \texttt{]}|v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0\ \texttt{!=}\ q\},C) \quad q \notin \mathrm{FV}(C)}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{join}\ q\ \texttt{in}\ \delta : k : \forall \hat{q}'.\ \texttt{kernel}(v|C)}\ \text{KALL}$$

In this rule, δ represents an ordered set of integers $[n_1,n_2]$. Define the set $\delta_{n_2'} = [n_1,n_2']$ for $n_2' \in [n_1,n_2]$ We proceed by induction over n_2'. Our inductive hypothesis is that

$$\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{join}\ q\ \texttt{in}\ \delta_{n_2'} : k :$$
$$\forall \hat{q}'.\ \texttt{kernel}(v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 \texttt{<=} n_2'\}|v\ \{q_0\ \texttt{in}\ D\texttt{:}\ n_2' \texttt{<}\ q_0\},C,\texttt{true})$$

is sound for any $n_2' \in [n_1,n_2]$ with $n_2' = n_1$ as the base case.

- **Base case.** The definition of semantics for samplers defines that the product over a set of size 1 simply returns the inner sampler s. We also note the following equivalences between variable sets

  $$v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 \texttt{<=} n_1\} = v\ \texttt{[}\ n_1\ \texttt{]}, \quad v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ n_1 \texttt{<}\ q_0\} = v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0\ \texttt{!=}\ n_1\}$$

  justifying the type judgment.
- **Inductive step.** Assuming the inductive hypothesis is true for $n_2' - 1$, we will show it is true for n_2'. We first note that the following programs are semantically equivalent:

  $$\texttt{join}\ q\ \texttt{in}\ \delta_{n_2'} : k$$
  $$\texttt{def}\ x_I\ \texttt{(}q\ \texttt{in}\ \delta\texttt{):}\ t = k\ \texttt{;}\ \texttt{(join}\ q\ \texttt{in}\ \delta_{n_2'-1} : k\texttt{)}\ \texttt{;}\ x\ \texttt{(}n_2'\texttt{)}$$

  We will now give an appropriate type to the latter program, and show that this type is sound. We note that choosing t as

  $$\texttt{kernel(}v\ \texttt{[}\ q\ \texttt{]}|v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 \texttt{<}\ q\},v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q \texttt{<}\ q_0\},C,\texttt{true)}$$

  is always a valid coercion. The inductive hypothesis is equivalent to stating that the following type is sound for the product in the above program:

  $$\forall \hat{q}'.\ \texttt{sampler(}v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 \texttt{<}\ n_2'\}|C,\phi\texttt{)}$$

  Using the soundness of the DEF, CALL, and K2 rules, we show that

  $$\forall \hat{q}'.\ \texttt{kernel(}v\ \texttt{[}\ n_2'\ \texttt{]},v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 \texttt{<}\ n_2'\}|v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ n_2' \texttt{<}\ q_0\},C,\texttt{true)}$$

  is a sound type for this program. Using the semantics of variables sets, we show that this type is semantically equivalent to the type

  $$\forall \hat{q}'.\ \texttt{kernel(}v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ q_0 \texttt{<=} n_2'\}|v\ \{q_0\ \texttt{in}\ \delta\texttt{:}\ n_2' \texttt{<}\ q_0\},C,\texttt{true)}$$

  completing the proof.

The conclusion of the rule follows from choosing $n_2' = n_2$, since $\delta_{n_2} = D$

KFIX. Recall the KFIX rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash k : \forall \hat{q}.\ \texttt{kernel}(A|B, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{fix}\ k : \forall \hat{q}.\ \texttt{sampler}(A|B, \phi)}\ \text{KFIX}$$

The semantics of `fix` state that the sampler `fix` k must be consistent with the second proposition of kernel soundness, and we know from the progress theorem that this sampler's denotation must be unique.

5.6 Estimators

Estimator Soundness Theorem. An estimator is considered sound if the sample it produces, when fed through a reweighted version of an arbitrary positive function f, computes the expectation of that function. Specifically, under the assumptions

(1) $\mathcal{M}, \Gamma, \mathcal{L} \vdash s : \forall \hat{q}.\ \texttt{estimator}(A|B, \phi)$
(2) $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$

it must be true that, for any environment $\sigma' = \sigma[q_0 \mapsto n_0][q_1 \mapsto n_1] \ldots [q_m \mapsto n_m]$ that binds values for the set of quantified variables $\hat{q} = q_0\ \texttt{in}\ \delta_0, q_1\ \texttt{in}\ \delta_1, \ldots, q_m\ \texttt{in}\ \delta_m$, and any $f \in \Sigma_{rv} \to \mathbb{R}^+$

$$[\![\phi]\!](\sigma') \Rightarrow \int_{\text{sr}} \frac{\pi_0([\![e]\!](\sigma', \text{sr})) * f(\pi_1([\![e]\!](\sigma', \text{sr})))}{\int_{\text{sr}} \pi_1([\![e]\!](\sigma', \text{sr}))} = \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma'))(\sigma')$$

Proof sketch. We proceed by structural induction on the rules which may produce estimators. Individual cases are outlined below.

ELIFT. Recall the ELIFT rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash e : \forall \hat{q}.\ \texttt{sampler}(A|B, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{lift}\ e : \forall \hat{q}.\ \texttt{estimator}(A|B, \phi)}\ \text{ELIFT}$$

Since the first element of e is defined to be 1 in all cases the expression

$$[\![\phi]\!](\sigma') \Rightarrow \int_{\text{sr}} \frac{\pi_0([\![e]\!](\sigma', \text{sr})) * f(\pi_1([\![e]\!](\sigma', \text{sr})))}{\int_{\text{sr}} \pi_1([\![e]\!](\sigma', \text{sr}))}$$

can be simplified to $\frac{\int_{\text{sr}} f([\![s]\!]([q \mapsto a], \text{sr}))}{\int_{\text{sr}} 1} = \int_{\text{sr}} f([\![s]\!]([q \mapsto a], \text{sr}))$ which, according to the correctness of the sampler, must equal the expression $\int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma'))(\sigma')$ as required.

EFACT. Recall the EFACT rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash e : \forall \hat{q}.\ \texttt{estimator}(A|B, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash d : \forall \hat{q}.\ \texttt{density}(C|A, B, \phi) \quad \text{DDep}(C, B)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{factor}\ e\ \texttt{by}\ d : \forall \hat{q}.\ \texttt{estimator}(A|B, C, \phi)}\ \text{EFACT}$$

Using the definitions from the semantics, the expression

$$[\![\phi]\!](\sigma') \Rightarrow \int_{\text{sr}} \frac{\pi_0([\![e]\!](\sigma', \text{sr})) * f(\pi_1([\![e]\!](\sigma', \text{sr})))}{\int_{\text{sr}} \pi_1([\![e]\!](\sigma', \text{sr}))}$$

becomes

$$\frac{\int_{\text{sr}} [\![d]\!]([\![s]\!](\sigma', \text{sr})) f([\![s]\!](\sigma', \text{sr}))}{\int_{\text{sr}} [\![d]\!]([\![s]\!](\sigma', \text{sr}))} = \int_{[\![A]\!](\sigma')} f(\sigma') * \mathcal{J}([\![A]\!](\sigma')|[\![B, C]\!](\sigma'))(\sigma')$$

where the last step requires inlining the soundness theorems for d and s, and on properties of $\mathcal{J}$ established in the proof for DMUL.

5.7 Structural Rules

For clarity, we have omitted the cases for structural rules in the above theorems. The cases are symmetric for each theorem.

DEF. Recall the DEF rule

$$\frac{\mathcal{M}, \Gamma :: [q_0 : \delta_0] :: \ldots, \mathcal{L} \vdash p_1 : \forall q_0, \ldots. \, t_1 \quad \mathcal{M}, \Gamma :: [x : (q_0, \ldots), (\delta_0, \ldots), t_1], \mathcal{L} \vdash p_2 : t_2}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{def}\ x\ (q_0\ \texttt{in}\ \delta_0, \ldots) : t_1 = p_1\ ;\ p_2 : t_2}\ \text{DEF}$$

According to the definition of $\vDash$, we can show that.

$$\sigma' \vDash \Gamma, \mathcal{M}, \mathcal{L} \Rightarrow \sigma'[x \mapsto ((q_0, \ldots), p_1)] \vDash \Gamma :: [x : (q_0, \ldots), (\delta_0, \ldots), t_1], \mathcal{M}, \mathcal{L}$$

which means we can inductively apply the soundness assumption for p_2. The proof for the DEF-IND rule is similar.

INV. Recall the INV rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash x : \hat{q}, \hat{\delta}, T_b(A|B, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash \hat{e} : \hat{\delta} \quad (\bigcup_{e \in \hat{e}} \text{FRV}(e)) \subseteq B}{\mathcal{M}, \Gamma, \mathcal{L} \vdash x\ (\hat{e}) : \forall \text{FQV}(\hat{e}).\ \left(T_b(A|B, \phi)\right)[\hat{e}/\hat{q}]}\ \text{INV}$$

Due to the assumption that $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$, we know that $\sigma(x) = (q_0\ \texttt{in}\ \delta_0, \ldots), p$ is defined and furthermore $\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall q_0\ \texttt{in}\ \delta_0, \ldots. \, T_b(A|B, \phi)$. The soundness of the INV rule follows straightforwardly from applying the inductive assumption and the substitution-environment lemma.

C. Recall the C rule

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall \hat{q}. \, t_1 \quad \hat{q} \subseteq \hat{q}' \quad \mathcal{L} \vdash t_1 \to t_2}{\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall \hat{q}'. \, t_2}\ \text{C}$$

Writing the types t_1 and t_2 as $T_b(A_1|B_1, \phi_1)$ and $T_b(A_2|B_2, \phi_2)$, respectively, the added assumptions mean that, for any σ', $[\![A_1]\!](\sigma') = [\![A_2]\!](\sigma')$, $[\![B_1]\!](\sigma') = [\![B_2]\!](\sigma')$, and $[\![\phi_1]\!](\sigma') \Rightarrow [\![\phi_2]\!](\sigma')$. This means that the soundness of the judgment $\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall q. \, t_1$ *implies* the soundness of the judgment $\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall q'. \, t_2$.

CIND. Recall the IND and CIND rules

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall \hat{q}. \, t_1 \quad \mathcal{L} \vdash t_1 \to_I t_2}{\mathcal{M}, \Gamma, \mathcal{L} \vdash_I p : \forall \hat{q}. \, t_2}\ \text{IND} \qquad \frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash_I p : \forall \hat{q}. \, t}{\mathcal{M}, \Gamma, \mathcal{L} \vdash p : \forall \hat{q}. \, t}\ \text{CIND}$$

Writing the types t_1 and t_2 as $T_b(A|B, \phi)$ and $T_b(A|B, C, \phi)$, respectively, the added assumptions mean that, for any σ', $\mathcal{J}([\![C, A]\!](\sigma')|[\![B]\!]) = \mathcal{J}([\![C]\!](\sigma')|[\![B]\!](\sigma')) * \mathcal{J}([\![A]\!](\sigma')|[\![B]\!])$ which, applying the identity from the DMUL case, means $\mathcal{J}([\![A]\!](\sigma')|[\![B, C]\!](\sigma')) = \mathcal{J}([\![A]\!](\sigma')|[\![B]\!](\sigma')$.

6 THE SHUFFLE SYSTEM

Shuffle as a system performs type checking, assumption log generation, and inference program extraction. A developer therefore receives a concrete executable inference program that has been type checked against the program's specified types as well as an auditable list of assumptions about the probabilistic model

6.1 Type Checking

The Shuffle system implements the type checking rules presented in Section 4.

Assumption Log. Shuffle's type checking algorithm emits an assumption log. The assumption log includes assumed independence assertions generated by The rule DEF-IND in Figure 17 and assumed reachability assertions generated by rule KFIX in Figure 15.

6.2 Inference Program Extraction.

Shuffle extracts a python program for a given type checked Shuffle program. Shuffle's program extraction is by and large a straightforward, syntax-directed recursive procedure that produces a python program that implements the denotational semantics presented in Figure 8, Figure 9, Figure 10, Figure 11, and Figure 12 . Shuffle's extraction procedure differs operationally from the denotational semantics in that it 1) simplifies integral expressionr 2) fails to compile integral expressions it cannot simplify and 3) uses a representation for probabilities that ensures numerical stability.

To illustrate the extraction process, consider the following fragment from the GMM inference procedure in Figure 3.

```
1  def independent obsProd(j in Mus) :
2      density(obs{i0 in Samples: z[i0] == j} | mu[j], z) =
3      prod i in Samples where z[i] == j : obsDensityI(i,j);
4
5  def obsJoint(j in Mus) :
6      density(obs{i0: z[i0] == j}, mu[j] | z) =
7      obsProd(j) * muPriorZ(j);
8
9  def obsMarg(j in Mus) :
10     density(obs{i0 in Samples: z[i0] == j} | z) =
11     int obsJoint(j) by mu[j];
12
13 def muPost(j in Mus) :
14     density(mu[j] | obs{i0 in samples: z[i0] == j}, z) =
15     obsJoint(j) / obsMarg(j);
16
17 def export obsMarg1(j in Mus):
18     density(obs{i0 in Samples: z[i0] == j} |
19             obs{i0 in Samples: z[i0] < j}, z) =
20     (obsJoint(j) / muPost(j));
```

Simplification. Shuffle first simplifies the inference procedure by removing any type coercions and inlining any `def` statements. For the running example, this yields the following procedure:

```
1  def export obsMarg1(j in Mus) =
2      (
3          prod i in Samples where z[i] == j: normal(obs[i],mu[j],1) *
4          normal(mu[j],0,100)
5      )
6      /
7      (
8          (
9              prod i in Samples where z[i] == j:
10                 normal(obs[i],mu[j],1)
11             *
12             normal(mu[j],0,100)
13         ) /
14         (int (
15             prod i in Samples where z[i] == j:
16                 normal(obs[i],mu[j],1)
17             *
18             normal(mu[j],0,100)
19         ) by mu[j])
20     )
```

Next, Shuffle simplifies integrals with known closed-form solutions. Shuffle can currently simplify conjugate and posterior-predictive distributions for Gaussian and Dirichlet distributions. In the running example, the code from Line 7 to Line 20 is the Gaussian conjugate distribution. Shuffle recognizes this, and simplifies this sub-procedure. The running example becomes

```
1  def export obsMarg1(j in Mus):
2      (
3          prod i in Samples where z[i] == j: normal(obs[i],mu[j],1) *
4          normal(mu[j],0,100)
5      )
6      /
7      (
8          let tmp1 = (sum i in Samples where z[i] == j: obs[i]) in
9          let tmp2 = (sum i in Samples where z[i] == j: 1) in
10         normal (
11             mu[j],
12            (1 /. 100 + tmp1 /. 1) / (1 /. 100 + tmp2 /. 1),
13            1 / (1 /. 100 + tmp2 /. 1)
14         )
15     )
```

Code Generation. Shuffle translates the simplified procedure to Python code. Any operations involving probabilities become logarithmic space-operation for numerical stability, meaning multiplication becomes addition and division becomes subtraction. If there are any unsimplified integrals over real-valued random variables, the code generator produces an error. The running example generates the following Python code:

```
1  from samplelib import *
2
3  def obsMarg1(j,z,obs,mu):
4      tmp1 = 0
5      for i in Samples:
6          if z[i] == j:
7              tmp1 += obs[i]
8          else:
9              tmp1 = tmp1
10     tmp2 = 0
11     for i in Samples:
12         if z[i] == j:
13             tmp2 += 1
14         else:
15             tmp2 = tmp2
16     tmp3 = 0
17     for i in Samples:
18         if z[i] == j:
19             tmp3 += normald(obs[i],mu[j],100)
20         else:
21             tmp3 = tmp3
22     return (tmp3 + normald(mu[j],0,100)) - normald(
23             mu[j],
24             (1/100 + tmp1/1) / (1/100 + tmp2/1),
25             1/(1/100 + tmp2/1)
26     )
```

Here, the function `normald` on Line 22 is assumed to be a library function imported from in `samplelib` on Line 1.

7 EVALUATION

In this section we evaluate the performance of extracted Shuffle inference procedures for several models. A key goal is demonstrate that Shuffle's abstractions do not reduce the performance of an algorithm when compared to a standardly handcoded solution.

7.1 Methodology

Performance. The performance of an approximate inference algorithm can be broken down into the *approximation amount* and the *computational efficiency*. The approximation amount is simply the number of samples required to reduce the Monte Carlo approximation error to an acceptable amount, or similarly the number of iterations in the iterative implementation of `fix` that are required to achieve the desired level of accuracy. The computational efficiency is the amount of time required to generate a sample or run an iteration. In this section, we focus on the computational efficiency of Shuffle relative to other state-of-the-art probabilistic inference systems.

Baseline. To determine Shuffle's computational efficiency independently of approximation amount, we compared against an existing probabilistic programming system called Venture [14]. Venture makes a good comparison point for Shuffle because unlike other probabilistic programming systems, Venture provides flexible built-in support for advanced handcoded inference procedures (however without a guarantee of safety). Also, Venture is written in Python, the same language

	Shuffle	Venture
GMM	0.349 ± 0.007s	2.16 ± 0.04s
SLAM	0.138 ± 0.002s	3.10 ± 0.04s
LDA	573 ± 13s	5.07 ± 0.05s

(a) Shuffle vs. Venture run time on three different models. The format is "average" ± "standard deviation".

GMM Size	Shuffle	Venture	Speedup (average)
$N = 20, K = 5$	0.067 ± 0.001s	0.45 ± 0.02s	6.7x
$N = 20, K = 10$	0.237 ± .003s	0.86 ± 0.04s	3.6x
$N = 20, K = 15$	0.500 ± 0.004s	1.43 ± 0.05s	2.86x
$N = 20, K = 20$	0.89 ± 0.01s	1.94 ± 0.06s	2.2x

(b) Speedup of Shuffle vs. Venture for various sizes of GMM. N is the number of data points and K is the number of clusters. The runtimes of Shuffle and Venture each are reported in the format "average" ± "standard deviation", and the speedup is the average Venture time divided by the average Shuffle time.

Table 1

that Shuffle extracts to, so any performance difference is due differing internal abstractions and not the underlying sampling and mathematical primitives.

Case Study. As a case study, we considered inference algorithms that employ *collapsing*. Collapsing is the task of using density arithmetic to remove a random variable from consideration during the inference algorithm. As an example, the approximate inference algorithm for a GMM in Figure 4 collapses out the `mu` variable by means of analytic solutions to integrals over Gaussian probability densities.This GMM inference algorithm approach is known as collapsed Gibbs sampling, for which there are theoretical results in the literature on how collapsing affects the approximation amount[13]. In each of the benchmarks below, we consider some form of sampling that is made more efficency through collapsing.

Benchmarks. The three benchmarks we used were:

(1) **GMM.** A Gaussian mixture model similar to the one in Figure 2. This model contained 150 datapoints and 3 cluster centers. Inference in this model uses an approximate sampler similar to the one in Figure 4.
(2) **SLAM.** A small instance of the Simultaneous Localization And Mapping problem [6]. Inference is performed with a Rao-Blackwellized particle filter [6] without any "resampling" or "rejuvenation" steps. This benchmark used 50 samples or *particles*.
(3) **LDA.** A Latent Dirichlet Allocation model [2], using 100 words, 5 topics, 5 documents and an alphabet size of 100. Words are assumed to be distributed evenly across the documents. Inference is performed using a collapsed Gibbs sampler [10].

For the Gibbs samplers for GMM and LDA, we ran each sampler for 100 iterations and found the average iteration time. For SLAM, we ran both Shuffle's and Venture's inference procedures 100 times and computed the average run-time.

7.2 Results

Performance Results. Table 1a shows the results of the experiments. Shuffle performed well in most cases, but was slower on the LDA benchmark. Suspecting scalability issues, we ran tests that varied the size of the model.

Assumption Type	GMM	SLAM	LDA
Independence	7	5	13
Reachability	1	0	1

Table 2. The number of each type of assumption in each case study.

Table 1b shows the results of varying the size of the GMM. For this test, we used 100 iterations at each model size, with data generated from the prior, and recorded the average speedup of Shuffle over Venture. This shows that while Shuffle enjoys a considerable advantage, it is reduced at larger scales. We suspect Shuffle is faster because it has leaner abstractions compared to Venture's internal data structures. Venture performs run-time type checking, interprets rather than compiles inference procedures, and indirectly implements Gibbs sampling as a type of Metropolis-Hastings[11, 15] sampler, all of which contribute to overhead. We are exploring program optimizations to improve the scalability of Shuffle's inference procedures.

Assumption Logs. For each case study, we measured the number of each type of external assumption that Shuffle generated. Table 2 shows the results. Shuffle generates on the order of 10 independence assumptions on each case study, and at most one reachability assumption. We are investigating whether these assumptions can be verified automatically.

8 RELATED WORK

Shuffle builds on **(author?)** [1] by providing a novel semantics and programming language for typesafe probabilistic programming. There are several systems that enable their users to perform probabilistic inference.

Automated Inference. Church [8] and WebPPL [9] enable a user to specify Turing-complete stochastic programs as models, but restrict inference algorithms to all-purpose algorithms such as Metropolis-Hastings [11, 15]. JAGS [18] provides a notation for expressing graphical models and automatically performs 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 for a wide class of models.

Manual Unverified Inference. Other systems, suchas Venture [14] and PyMC [17] enable a user to augment the system's inference procedure with arbitrary code. However, when the user augments the inference algorithm with arbitrary code, there is no guarantee that the resulting inference algorithm 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 guarantees.

Compiled Inference. AugurV2 [4] provides a language of coarse-grained operators to build inference procedures out of, like Shuffle. AugurV2 supports a richer set of kernels than Shuffle, but does not support estimators. AugurV2 also provides more support for parallelism and alternative compilation targets. However, AugurV2 does not provide correctness guarantees as strong as Shuffle's. In particular, AugurV2's kernels are not guaranteed to converge iteratively to the target distribution. AugurV2 also does not have density operations to support collapsed Gibbs samplers. Thus AugurV2 does not support any of the benchmarks from Section 7, although it does support other inference procedures for the GMM and LDA models.

Automatic Simplification. There are languages that have built-in systems for simplifying probability densities. For example, Hakaru [16] employs the Maple computer algebra system and PSI [7] 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, kernels, and estimators. Hakaru provides tools for composing simplified densities, but only transforms whole programs. Shuffle's proof rules, by contrast, can build a whole inference program out of smaller programs for sub-components of the model.

9 CONCLUSION

In this paper we presented Shuffle, a system for typesafe programming with probability distributions. Shuffle's language of distributions is rich enough to support several complicated inference algorithms. The terms in this language are densities, samplers, kernels, and estimators, and we have developed operators over these terms as well as type rules that associate each term with part of a probabilistic model. We have proven Shuffle's type system is sound with respect to the semantics we have provided.

Shuffle supports extracting inference algorithms to Python, and the performance of extracted code compares favorably with probabilistic programming systems using the same base language. Shuffle also has the ability to simplify some integral expressions. Most importantly, Shuffle can generate *proof obligations* that are necessary for an inference algorithm's correctness. These encapsulate parts of the verification process that are external to Shuffle itself.

The aim of Shuffle is to explore the relationship between program verification and probabilistic inference. Probabilistic models provide good specifications for situations where there is uncertainty, such as with inference algorithms. However, inference algorithms have resisted compositional analysis and verification due to their randomized and uncertain nature. Shuffle provides developers with the ability to develop inference algorithms with confidence that they are correct, and hopefully, similar techniques could result in a suite of programming tools for developers to handle uncertainty effectively.

REFERENCES

[1] Eric Atkinson and Michael Carbin. Towards correct-by-construction probabilistic inference. In *LearningSys*, 2016.

[2] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. In *J. Mach. Learn. Res.*, volume 3, 2003.

[3] G. E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. In *Annals of Mathematical Statistics*, 1958.

[4] Greg Morisett Daniel Huang, Jean-Baptiste Tristan. Compiling markov chain monte carlo algorithms for probabilistic modeling. In *PLDI*, 2017.

[5] Elizabeth M. Wilmer David A. Levin, Yuval Peres. Markov chains and mixing times. 2008.

[6] Arnaud Doucet, Nando de Freitas, Kevin Murphy, and Stuart Russell. Rao-blackwellised particle filtering for dynamic bayesian networks. In *UAI*, 2000.

[7] Timon Gehr, Sasa Misailovic, and Martin Vechev. PSI: Exact symbolic inference for probabilistic programs. In *CAV*, 2016.

[8] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *UAI*, 2008.

[9] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. 2014. Accessed: 2016-10-7.

[10] T. Griffiths and M. Steyvers. Finding scientific topics. In *PNAS*, volume 101, 2004.

[11] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. In *Biometrika*, volume 57, 1970.

[12] Nikolaj Bjørner Leonardo De Moura. Z3: An efficient smt solver. In *TACAS / ETAPS*, 2008.

[13] Jun S. Liu. The collapsed gibbs sampler in bayesian computations with applications to a gene regulation problem. In *Journal of the American Statistical Association*, volume 89, 1994.

[14] V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. In *ArXiv e-prints*, 2014.
[15] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. In *Journal of Chemical Physics*, volume 21, 1953.
[16] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in hakaru (system description). In *FLOPS*, 2016.
[17] Anand Patil, David Huard, and Christopher Fonnesbeck. Pymc: Bayesian stochastic modelling in python. 35, 2010.
[18] Martyn Plummer. *JAGS Version 4.0.0 user manual*. Addison-Wesley, Reading, Massachusetts, 2015.
[19] D. Tran, M. D. Hoffman, R. A. Saurous, E. Brevdo, K. Murphy, and D. M. Blei. Deep probabilistic programming. In *ICLR*, 2017.