Typesafety for Explicitly-Coded Probabilistic Inference Procedures

by

Eric Hamilton Atkinson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

Author ... Signature redacted .....
Department of Electrical Engineering and Computer Science
January 5, 2018

Certified by ... Signature redacted .....
Michael Carbin
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ..... Signature redacted .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Typesafety for Explicitly-Coded Probabilistic Inference Procedures

by

Eric Hamilton Atkinson

Submitted to the Department of Electrical Engineering and Computer Science
on January 5, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

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 thesis, I 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.

Thesis Supervisor: Michael Carbin
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank the members of the Programming Systems Group for their assistance with this project. I would like to thank Ben Sherman for helping me formalize Shuffle's estimator constructs, and Cambridge Yang for implementing Shuffle's optimizations and performance benchmarks and writing about them for Section 7 of this document. I would also like to thank my friends and family for their support during my graduate studies. This research was supported in part by the United States Department of Energy (Grants DE-SC0008923 and DE-SC0014204).

1 Introduction

Bayesian probabilistic inference is a process in which a developer specifies a generative probabilistic model and computes a posterior distribution for a set of variables in that model. Probabilistic models provide a flexible and well-studied formalism for uncertainty.

Researchers have recently proposed several systems that ease the process of developing inference procedures, or programs that implement posterior distributions. While inference procedures can be explicitly coded in general-purpose languages, automated systems [22, 10, 9, 7] provide stronger guarantees that the inference procedure correctly computes the posterior distribution for a given model. These systems take probabilistic models as input, and automatically produce inference procedures. The system, rather than the developer, ensures that the procedure is correct.

However, these systems can only generate a limited set of inference procedures. As a result, several recent systems [16, 21] allow developers to combine automatically-generated inference procedures with explicitly-coded components. In cases where a developer can write some or all of the inference code, existing systems have limited capability to help developers ensure that their inference procedures 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 thesis I 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.

1.1 Contributions

Language: Shuffle provides a set of operators that enable a developer to compose terms to produce an inference procedure. Terms of Shuffle's inference language include:

- **Densities.** Densities are functions which, given an assignment of random variables to values, return a real number representing the probability of that assignment – or, in the case of continuous random variables, the probability of an infinitesimal region around the assignment divided by the size of the region.

- **Samplers.** Samplers represent distributions by randomly choosing a new assignment of random variables to values. The chance that the sampler returns a particular value is the probability of that assignment.

- **Kernels.** Kernels represent distributions by being samplers that are invariant with respect to a distribution. This means that if the values of random variables are drawn from the desired distribution, then feeding this assignment into the kernel also results in an assignment that is drawn from the desired distribution.

- **Estimators.** Estimators represent distributions by producing a sample and a weight. By repeatedly calling the estimator, a developer can use the resulting list of weighted samples to estimate the expected value of any function under the distribution.

I present a semantics for each of these operators in Section 4. Inference procedures supported by Shuffle include variable elimination [23], Gibbs sampling [8], and likelihood weighting [6].

Type System: Shuffle's type system describes, for a given probabilistic model, which distribution an inference procedure represents in that model. For example, if d_1 is a density for the distribution $\Pr(A|B,C)$, where A,B, and C are sets of random variables, and d_2 is a density for the distribution $\Pr(B|C)$, then Shuffle's type system determines that the density $d_1 * d_2$ is a density for the distribution $\Pr(A,B|C)$. I

present Shuffle's type system and prove it correct with respect to Shuffle's semantics in Section 5.

Shuffle Environment: Shuffle takes as input a probabilistic model and an inference procedure written in Shuffle's language. It generates an executable inference procedure, as well as a set of *statistical assumptions.* The executable procedure is a Python program. The statistical assumptions are extra preconditions that Shuffle cannot verify internally. For example, the correctness of some inference algorithms relies on *statistical independence* relations between random variables in the model. These must be manually audited by the developer.

Summary. Altogether, Shuffle enables a developer to build a rich set of inference procedures. Furthermore, Shuffle's type system ensures that the procedures are correct with respect to a given probabilistic model, meaning they represent distributions in the model the developer wishes to compute.

2 Example: Gaussian mixture model

To use Shuffle to create an inference program, a developer first specifies a probabilistic model. Figure 1 presents a specification of a 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 GMM specified here has observations contained in `obs` and mixture components in `mu`. 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 real-valued random variable

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

Figure 1: A Gaussian mixture model with in Shuffle.

corresponding to a single element of the domain `Samples`. A domain is a named subset of the natural numbers, and a target set is either a domain or the real numbers, with the latter referred to by `R`.

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 1 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 7 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`,`ϕ`)` 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. This is inspired by the notation $\Pr(A|B)$ used in conventional descriptions of conditional probability. In the case of `muPrior`, 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 10 gives, as implied by its type the density `density(z[i])`, a 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 14 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 conditioning (as a function of each quantified variable) as well as dynamic conditioning (as a function of the observed value of other random variables in the model). Shuffle's constraint language supports equalities and inequalities over quantified parameters and observed random variables.

Inference. Shuffle enables a developer to soundly construct an *inference program*. An inference program computes a conditional distribution from the model. For our example 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 $\Pr(\texttt{z} \mid \texttt{obs})$ and 2) the distribution of the mixture component means – generally denoted by $\Pr(\texttt{mu} \mid \texttt{obs})$. Shuffle enables a developer to compute these distributions through both *exact* and *approximate* inference techniques. Whereas exact inference computes densities for

these distributions, approximate inference computes other representations of these distributions using Shuffle's sampler, kernel, and estimator types.

2.1 Exact Inference

One way Shuffle enables a developer to compute Pr(`z | obs`) is by providing the developer with a set of *density operators* that enable him or her to construct a function of type `density(z|obs)` that exactly computes this function. Figure 2 presents a Shuffle inference procedure that implements a density with the appropriate type. In this procedure, the developer constructs, as an intermediates step, a function with type `density(z, obs)` that computes the joint density of `z` and `obs`. It then divides 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 $P(B) > 0$). I have deliberately unfolded most of this computation to make the types of the intermediate density objects clear.

Independence. In the definition of `zPriorI` on Line 1, the developer leverages the statistical independence relationships of the model to coerce the density within the model to different types.

Density Multiplication. Shuffle also enables a developer to multiply densities. An example of this in Figure 2 is on Line 7. In the recursive definition of `zPriorH` on Line 5, the developer multiplies `zPriorI` with the previous iteration of `zPriorH`. 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. Shuffle's types closely correspond with this notation, as, for example, `zPriorH(i-1)` and `ziPriorI(i)` have types `density(z{i0 in Samples: i0 <= i - 1})` and `density(z[i] | z{i0 in Samples: i0 < i})`, respectively. Shuffle' type checker computes that the former type is equivalent to `density(z{i0 in Samples: i0 < i})`. Shuffle then computes

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

def rec zPriorH(i in Samples) :
    density(z{i0 in datAPoints: i0 <= i}) =
    ziPriorI(i) * zPriorH(i - 1);

def zPriorAll() : density(z) =
    zPriorH(max(Samples));

def independent obsDensI (i in Samples, j in Mus) :
    density(obs[i] | mu[j], z, z[i] == j) =
    obsDens(i,j);

def independent obsDensI2 (i in Samples, j in Mus) :
    density(
        obs[i] |
        obs{i0 in Samples: i0 < i && z[i0] == j}, mu[j],
        z, z[i] ==j
    ) =
    obsDensI(i,j);

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

def rec obsProdH(i in Samples, j in Mus) :
    density(obs{i0 in Samples: i0 <= i && z[i0] == j}
        | mu[j], z) =
    if (z[i] == j) {
        obsDensI2(i,j) * obsProdH(i-1,j)
    } else {
        obsProdH(i-1,j)
    };

def obsProd(j in Mus) :
    density(mu[j], obs{i0 in Samples: z[i0] == j} | z) =
    obsProdH(max(Samples),j) * muDensI(j);

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

```
43 def muPost (j in Mus) :
44     density(mu[j] | obs{i0 in Samples: z[i0] == j}, z)
45     = obsProd(j) / obsProdMarg(j);
46
47 def independent obsLike(j in Mus) :
48     density(obs{i0 in Samples: z[i0] == j} |
49         obs{i0 in Samples: z[i0] < j},
50         z
51     ) =
52     obsProd(j) / muPost(j);
53
54 def rec obsLikeAllH(j in Mus) :
55     density(obs{i0 in Samples: z[i0] <= j} | z) =
56     obsLike(j) * obsLikeAllH(j - 1);
57
58 def obsLikelihood() : density(obs | z) =
59     obsLikeAllH(max(Mus));
60
61 def obszJoint() : density(obs,z) =
62     obsLikelihood * zPriorAll();
63
64 def obsMarg() : density(obs) =
65     int obszJoint() by z;
66
67 def export zPost() : density(z | obs) =
68     obszJoint / obsMarg() by z
```

Figure 2: Inference program for computing `density(z | obs)` for a GMM.

the type of the product to be `density(z[i], z{i0 in Samples: i0 < i})`, and checks that this is the same as the type annotation for `zPriorH`, `density(z{i0 in Samples: i0 <= i})`.

Recursion. The definition of `zPriorH` on Line 5 illustrates Shuffle's handling of recursive procedures. This definition takes the product of `zPrior(i)` over all values of the variable `i`. The effect on the type is an inductive proof. Specifically, assuming that invocations of `zPriorH` have the annotated type within the body of `zPriorH`, Shuffle verifies that the body also has this type. Shuffle specifies default base cases for all of its objects. In this case, Shuffle defines that `zPriorH(i-1)`, where `i` is the smallest value in the domain `Samples`, yields the value 1. This is because `zPriorH`'s type, `density(zi0 in Samples: i0 <= i)`, is an empty set of variables when `i` falls below the minimum value in `Samples`. The constant function returning 1 is always a correct density for an empty set of random variables.

Integration. Shuffle also enables a developer to marginalize out variables in a density via integration. Figure 2 contains to examples of integration on Lines 42 and 65. In the definition for `obsProdMarg` on Line 43, the developer integrates `obsProd(j)` with `mu[j]`. This has the effect of eliminating `mu[j]` from the type of `obsProd(j)`. Likewise, in the definition for `obsMarg` on Line 64. In this definition, the developer integrates `obszJoint()` with `z`. This has the effect of eliminating `z` from the type of `obszJoint`. This corresponds to the marginalization operation in probability, wherein $P(B) = \int P(A, B)$. Shuffle eliminates through simplification integrals with known analytic solutions. Otherwise, for integrals over finite sets, Shuffle computes integrals using summation. In Figure 2, Shuffle eliminates the integral on Line 42 and treats the one on Line 65 as a summation. This behavior is due to the fact that `mu[j]` is continuous set, but `z` is finite.

Density Division. Shuffle also enables the developer to divide densities as demonstrated in the definition for `muPost` on Line 43. This follows the reverse form of Bayes' rule, wherein $P(A|B) = P(A, B)/P(B)$. In this definition the developer constructs

`muPost` using the operator `/`, which divides the values returned by the two input densities `obsProd(j)` and `obsProdMarg(j)`. This has the opposite effect of multiplication, shifting a variable — in this case `obs{i0 in Samples: z[i0] == j}` — from being part of the joint density to being conditioned on.

Summary. Shuffle enables developers to use arithmetic operations to compose probability densities with well-typed operations that correspond to the rules of probability.

2.2 Approximate Inference

In the exact inference algorithm for GMM, although Shuffle is able to generate an efficient implementation of the integration in `obsMarg` (Line 64), the integration over the discrete variable `z` in `obszJoint` 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. 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.

Figure 3 presents an alternative approximate inference implementation in Shuffle for GMM that avoids executing the full summation. The result of this algorithm is an *estimator* for the distribution $P($`z` $\mid$ `obs`$)$, `zEst` (Line 112). 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 4 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` $\mid$ `obs`$)$ and thus `muApprox` computes the expectation of the indicator function for `z[0] != z[1]`.

```
def independent obsDens1 (i in Samples, j in Mus) :
    density(obs[i] | mu[j], z, z[i] == j) =
    obsDens(i,j);

def obsDens2 (i in Samples, j in Mus, k in Samples) :
    density(obs[i] | mu[j], z, z[i] == j && i != k) =
    obsDens1(i,j);

def independent obsDens3 (i in Samples, j in Mus,
    k in Samples) :
    density(
        obs[i] |
        obs{i0 in Samples:
            i0 < i && (z[i0] == j && i0 != k)
        }, mu[j], z ;
        z[i] ==j && i != k
    ) =
    obsDens2(i,j,k);

def independent muDens1(j in Mus) : density(mu[j] | z)
    = muPrior(j);

def rec obsProdHelper(i in Samples, j in Mus,
    k in Samples) :
    density(obs{i0 in Samples:
        i0 <= i && z[i0] == j && i0 != k
    } | mu[j], z) =
    if (z[i] == j && i != k) {
        obsDens3(i,j,k) * obsProdHelper(i-1,j,k)
    } else {
        obsProdHelper(i-1,j,k)
    };

def obsProd(j in Mus, k in Samples) :
    density(mu[j], obs{i0 in Samples:
        z[i0] == j && i0 != k
    } | z) =
    obsProdHelper(max(Samples),j,k) * muDens1(j);

def muPost (j in Mus, k in Samples) :
    density(mu[j] | obs{i0 in Samples:
        z[i0] == j && i0 != k
    }, z, z[k] == j) =
    obsProd(j,k) / int obsProd(j,k) by mu[j];
```

```
def independent obsDensNew (j in Mus, k in Samples) :
    density(
        obs[k] |
        obs{i0 in Samples:
            z[i0] == j && i0 != k
        }, mu[j], z, z[k] == j
    ) = obsDens(k,j);

def muJoint (k in Samples, j in Mus) :
    density(
        mu[j], obs[k] |
        obs{i0 in Samples:
            z[i0] == j && i0 != k
        }, z, z[k] == j
    ) = obsDensNew(j,k) * muPost(j,k);

def independent obsPred(k in Samples, j in Mus) :
    density(
        obs[k] | obs{i0 in Samples:
            i0 != k
        }, z, z[k] == j
    ) =
    int muJoint(k,j) by mu[j];

def obsPred1(k in Samples) :
    density(
        obs[k] | z[k],
        z{i0 in Samples : i0 != k},
        obs{i0 in Samples: i0 != k}
    ) =
    obsPred(k, z[k]);

def independent zDensNew(i in Samples) :
    density(z[i] |
        z{i0 in Samples: i0 != i},
        obs{i0 in Samples: i0 != i}) =
    zPrior(i);

def zJoint(k in Samples) :
    density(
        z[k], obs[k] |
        z{i0 in Samples: i0 != k},
        obs{i0 in Samples: i0 != k}
    ) = obsPred1(k) * zDensNew(k);
```

```
89  def ziPost(k in Samples) :
90      density(z[k] | z{i0 in Samples: i0 != k}, obs) =
91      zJoint(k) / int zJoint(k) by z[k];
92
93  def ziSample(k in Samples) :
94      sampler(z[k] | z{i0 in Samples: i0 != k}, obs) =
95      z[k] := sample ziPost(k);
96
97  def ziKernel(k in Samples) :
98      kernel(z[k] | z{i0 in Samples: i0 != k}, obs) =
99      lift ziSample(k);
100
101 def rec zKernelHelper(k in Samples) :
102     kernel(z{i0 in Samples: i0 <= k} |
103         z{i0 in Samples: k < i0}, obs) =
104     zKernelHelper(k-1); ziKernel(k);
105
106 def zKernel() : kernel(z | obs) =
107     zKernelHelper(max(Samples));
108
109 def zSample() : sampler(z | obs) =
110     fix zKernel();
111
112 def export zEst() : estimator(z | obs) =
113     lifte zSample()
```

Figure 3: Approximate Inference for GMM

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

Figure 4: 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

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 languages. In addition to the density operators presented in the previous section, the inference procedure in Figure 3 makes use of operators over these extra primitives.

Samplers. A sampler with type denoted by `sampler(A | B)` is a function that assigns new values to the random variable A according to the distribution $\Pr(A|B)$. In Figure 3, the definition `ziSample` 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.

Kernels. A kernel with type denoted by `kernel(A | B)` is a sampler that is *invariant* with respect to the distribution $\Pr(A|B)$. Thus, given a true sampler s for the distribution $\Pr(A|B)$, composing s with a kernel of type `kernel(A | B)` is still a sampler for $\Pr(A|B)$. Shuffle enables a developer to directly create a kernel from a sampler using the `lift` statement as in the definition of `ziKernel` (Line 97).

A developer can combine kernels together using the ";" operator, which executes each of its arguments in succession. For example, the definition of `zKernel` combines the recursively defined `zKernelHelper` with `ziKernel`. By combining kernels for each `z[i]` given the remainder of the `z` variables and all of the `obs` variables, `zKernel` constructs a kernel `z` given `obs` – 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.

Estimators. An estimator with type denoted by the notation `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. Shuffle enables a developer to adjust the weight of the sample to implement other approximate inference algorithms such as likelihood weighting [6].

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, I present the full Shuffle language along with its semantics, type system, soundness proofs, and an evaluation of the performance of Shuffle on several inference procedures, including Gibbs sampling and likelihood weighting.

3 The Language

Figures 5 and 6 present Shuffle's syntax for the declarative specification of the model and the code that implements an inference procedure, 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. A *domain declaration*, *DDecl*, specifies a domain $\delta \in \Delta$ of values. 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 $x(D_m^*)$, a multiplication of two densities, $D_m * D_m$, a division of a density by another density,

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

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

$$
\begin{aligned}
\oplus_m &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\prec &\rightarrow \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{<}
\end{aligned}
$$

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

$$
\begin{aligned}
D_m &\rightarrow r \mid n \mid q \mid v\texttt{[}D_m\texttt{]} \mid x(D_m^{*}) \mid D_m \oplus_m D_m \\
&\quad \mid \texttt{if}\ (\phi)\ \texttt{\{}\ D_m\ \texttt{\}}\ \texttt{else}\ \texttt{\{}\ D_m\ \texttt{\}}
\end{aligned}
$$

$$
\begin{aligned}
V_g &\rightarrow V_s \mid v \mid v\{q : \phi\} \\
V_s &\rightarrow v\texttt{[}n\texttt{]} \mid v\texttt{[}q\texttt{]}
\end{aligned}
$$

$$
\begin{aligned}
\phi &\rightarrow A \prec A \mid \neg\phi \mid \phi\ \texttt{\&\&}\ \phi \mid \phi\ \texttt{||}\ \phi \\
A &\rightarrow n \mid q \mid q\ \texttt{-}\ n \mid V_s \mid \texttt{max(}\delta\texttt{)} \mid \texttt{min(}\delta\texttt{)}
\end{aligned}
$$

Figure 5: The Syntax of Shuffle Models and Types

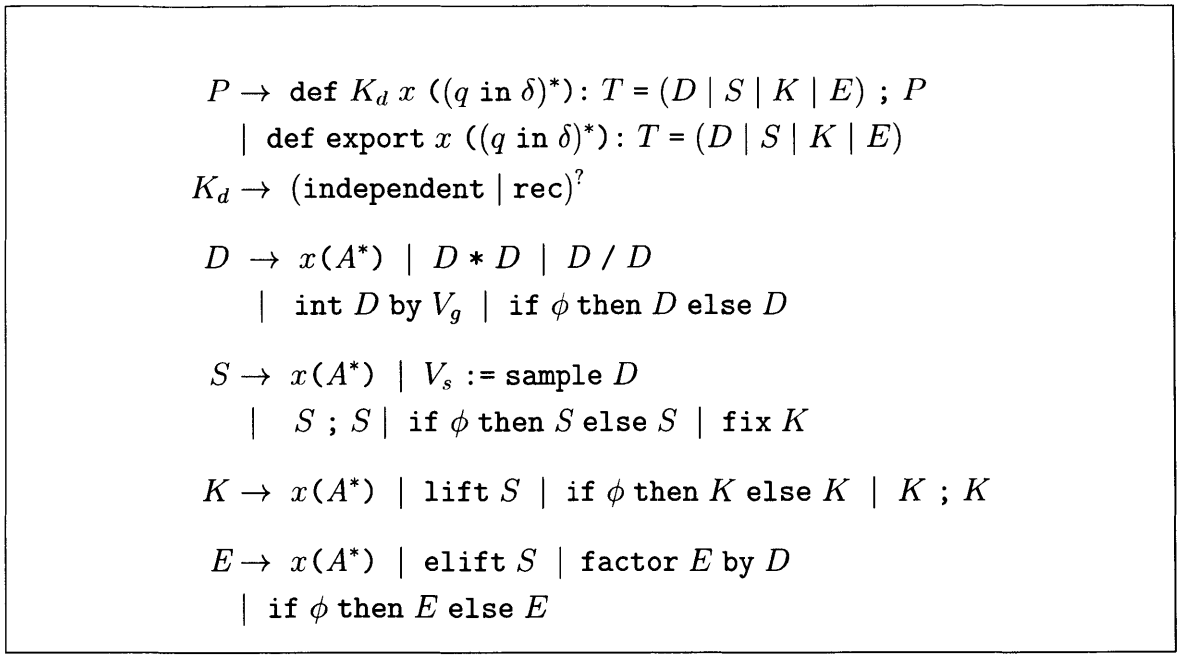

Figure 6: The Syntax of Shuffle Inference Procedures

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.

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 $q \in \mathcal{Q}$ that are bound within D_m. The definition also specifies a type, T, for the definition.

Types. The language of types $T_b(V_g^+\left(\mid V_g^+(, \phi)^? \right)^?)$ 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, disjunction, and negation) of inequalities over 1) integers, 2) quantified variables from domains that are isomorphic to the integers,

and 3) a single random variable with a value from a domain that is isomorphic to the integers.

3.2 Inference Procedure

Densities. An *inference probability density*, D, defines a probability distribution through density operators. The operators for an inference density are different from those of a model density. An inference procedure may *integrate* a density using the syntax `int` D `by` V_g and *invoke* a density with name x with the syntax x`(`A^*`)`, operations which are not available to model densities. However, an inference procedure may not add or subtract densities, and may only use constants, quantified variables, or random variables as arguments to invocations. These differences facilitate type checking of inference procedures, whereas model densities are flexible enough to support a range of probabilistic models.

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, and 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 and composing two kernels together.

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.

4 Semantics

I denote the semantics of a Shuffle term ρ, where ρ may be a density, sampler, estimator, constraint, or variable access by $[\![\rho]\!]$. The type of $[\![\rho]\!]$ is a function of the type of the term ρ. The following sections describe the behavior of $[\![\rho]\!]$ for each kind of term ρ.

4.1 Preliminaries

Errors. A Shuffle inference procedure may produce one of two error values instead of a conventional value: 1) a procedure produces the error value $\perp_\sigma$ 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 $\perp_0$ if and only if it contains a division by 0. In the semantics below I use $\perp = \{\perp_\sigma, \perp_0\}$ to refer to the domain of errors, and 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 $\{v, \perp_\sigma, \perp_0\}$ with the reflexive total order $v \leq \perp_\sigma, v \leq \perp_0, \perp_\sigma \leq \perp_0$ where v denotes a standard value.

Environments. An environment, $\sigma \in \Sigma = (V \times \mathbb{N}) + \mathcal{Q} + X \rightarrow (\mathbb{R}^+ + \mathbb{N})$ is a finite map from random variables, quantified variables, and bound distributions. The notation $\sigma(\rho)$ denotes the value to which ρ 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.

I use the notation $\sigma[\rho_1 \mapsto \rho_2]$ to mean σ with ρ_1, which could be any of the above, remapped to ρ_2. I use the notation π_q, π_{rv}, and π_x to denote projections that return environments with bindings only for quantified variables, random variables, and bound distributions, respectively. I use the notation $\sigma_1 + \sigma_2$ to refer to the environment that contains the combined bindings from σ_1 and σ_2. If a value is bound in both σ_1 and σ_2, it is bound to $\perp_\sigma$ in $\sigma_1 + \sigma_2$. I also use σ to refer to an element of $\Sigma_{\text{rv}} = (V \times \mathbb{N}) \rightarrow (\mathbb{R}^+ + \mathbb{N})$, the subset of Σ that only maps random variables.

Variables. Our formalization relies on several disjoint variable spaces. A *quantified variable* $q \in \mathcal{Q}$ is drawn from the space $\mathcal{Q}$; 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; and a *domain* $\delta \in \Delta$ is drawn from Δ, the space of domain names. 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 $v\texttt{[}\rho\texttt{]}$, where ρ is either a quantified variable q or a literal n, is the value that the environment maps $v\texttt{[}\rho\texttt{]}$ to. Shuffle's semantics assume that the environment always maps every random variable to a value.

3. **Distribution Variables.** A named distribution variable x may be *invoked* with a sequence of arguments $a_0, \ldots, a_n$. If x exists in the environment, then the denotation of the invocation $x\texttt{(}a_0\texttt{, } \ldots \texttt{, }a_n\texttt{)}$ is the denotation of the procedure x refers to, with the parameters $q_0, \ldots, q_n$ rebound to the invocation arguments.

4. **Domains.** The denotation of a domain $\delta \in \Delta$ is a range of natural numbers: $[\![\delta]\!] = [n_1, n_2] \subset \mathbb{N}$.

Variable Sets. A *variable set* is a comma delimited list of random variables (V_g^+ in Figure 5) that I denote by the symbols A, B, and C. I 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 syntactic form, I give variable sets the following denotation:

- **Set Comprehensions.** For variable sets of the form $A = v\texttt{\{}q_0\texttt{ in }\delta_1\texttt{: }\phi\texttt{\}}$, let $[\![A]\!](\sigma) = \{(v, n) \mid [\![\phi]\!](\sigma[q_0 \mapsto n])\}$.

- **Indexed Variables.** The single variable $v\texttt{[}\rho\texttt{]}$, in the context of a variable set is syntactic sugar for the set $v\texttt{\{}q_0\texttt{ in }\delta\texttt{: }q_0\texttt{ == }\rho\texttt{\}}$, with the corresponding denotation given by that for set comprehensions.

- **Whole Variables.** The variable set v is syntactic sugar for the set $v\texttt{\{}q_0\texttt{ in }\delta\texttt{: true\}}$, with the corresponding denotation given by that for set comprehensions.

$$[\![\texttt{def } x \texttt{ (}q_0 \texttt{ in } \delta_0, \ldots\texttt{): } t = \rho_1 \texttt{ ; } p_2]\!](\sigma) = [\![p_2]\!](\sigma[x \mapsto ((q_0, \ldots), \sigma, \rho_1)])$$

$$[\![\texttt{def independent } x \texttt{ (}q_0 \texttt{ in } \delta_0, \ldots\texttt{): } t = \rho_1 \texttt{ ; } p_2]\!] =$$

$$[\![\texttt{def } x \texttt{ (}q_0 \texttt{ in } \delta_0, \ldots\texttt{): } t = \rho_1 \texttt{ ; } p_2]\!]$$

$$[\![\texttt{def rec } x \texttt{ (}q_0 \texttt{ in } \delta_0, \ldots\texttt{): } t = \rho_1 \texttt{ ; } p_2]\!] = [\![p_2]\!](\sigma[x \mapsto ((q_0, \ldots), \sigma, \rho_1, \texttt{rec})])$$

$$[\![\texttt{def export } x \texttt{ (}q_0 \texttt{ in } \delta_0, \ldots\texttt{): } t = \rho]\!](\sigma) = [\![\rho]\!](\sigma)$$

Figure 7: Semantics of Shuffle's structural constructs

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

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

Source of Randomness. A *source of randomness*, denoted by "sr" is an infinite sequence of uniform distributed values on the interval $[0, 1] \subset \mathbb{R}^+$. Let the notation $\int_{\text{sr}} f(\text{sr})$ denote an integral over a set of finite prefixes of "sr". For a given source of randomness "sr", I use the notation $\text{sr}^0, \text{sr}^1 = \text{split}(\text{sr})$ to mean "sr" split into two identical sources of randomness sr^0 and sr^1 such 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.

4.2 Structural Constructs

Figure 7 presents the semantics of Shuffle's definition and invocation operators.

Definitions. The denotation of an inference procedure definition `def` x `(`q_0 `in` $\delta_0, \ldots$`):` t `=` ρ_1 `;` p_2 is the denotation of p_2 with x bound to the procedure ρ_1 with the parameters $(q_0, \ldots)$. The denotation erases the type t and the domains $(\delta_0, \ldots)$.

Recursive Definitions. The denotation of a recursive definition `def rec` x `(`q_0 `in` δ_0`, ...):` t `=` ρ_1 `;` p_2 is the same as that of an ordinary definition, except that the program ρ_1 is tagged as a recursive procedure.

4.3 Densities

Figure 8 presents the denotation of a density. 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.

Multiplication and Division. The * operator takes two densities and multiplies them together pointwise. The / operator divides the first density by the second pointwise.

Conditionals. The syntax `if` ϕ `then` d_t `else` d_f returns the value of the density d_t if the constraint ϕ is true, and that of d_f if the constraint is false.

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.

Invocation. The syntax $x(\hat{a})$ invokes a density named x that exists in the environment. The call evaluates the density in an environment where the quantified variables are rebound to their parameters $\hat{a}$. For recursive procedures, if the first argument falls outside the domain, the denotation of call is the default value 1. Shuffle's type system enforces that the set of random variables in this case is empty (see Section 5.5), and the constant function returning 1 is always a valid density for an empty variable set.

4.4 Samplers

Figure 9 presents the denotation of a sampler. 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

$$[\![d_1 * d_2]\!](\sigma) = [\![d_1]\!](\sigma) * [\![d_2]\!](\sigma) \qquad [\![d_1 \ / \ d_2]\!](\sigma) = \begin{cases} \frac{[\![d_1]\!](\sigma)}{[\![d_2]\!](\sigma)} & [\![d_2]\!](\sigma) \neq 0 \\ \bot_0 & \textbf{else} \end{cases}$$

$$[\![\texttt{if } \phi \texttt{ then } d_1 \texttt{ else } d_2]\!](\sigma) = \begin{cases} [\![d_1]\!](\sigma) & [\![\phi]\!](\sigma) \\ [\![d_2]\!](\sigma) & \textbf{else} \end{cases} \qquad [\![\texttt{int } d \texttt{ by } V_g]\!](\sigma) = \int_{[\![V_g]\!](\sigma)} [\![d]\!]$$

$$[\![x(a_0, \dots)]\!](\sigma) = \text{ let } (q_0, \dots), \sigma_p, \rho = \sigma(x) \text{ in } \begin{cases} [\![\rho]\!](\pi_q(\sigma_p)[q_0 \mapsto [\![a_0]\!](\sigma)] \dots [q_n \mapsto [\![a_n]\!](\sigma)] + \pi_{\text{rv}}(\sigma) + \pi_x(\sigma_p)) & x \in \sigma \\ \bot_\sigma & \textbf{else} \end{cases}$$

$$[\![x(a_0, \dots)]\!](\sigma) = \text{ let } (q_0, \dots), \sigma_p, \rho, \texttt{rec} = \sigma(x) \text{ in}$$

$$\begin{cases} 1 & [\![a_0]\!](\sigma) < \min([\![\delta_0]\!]) \\ [\![\rho]\!](\pi_q(\sigma_p)[q_0 \mapsto [\![a_0]\!](\sigma)] \dots [q_n \mapsto [\![a_n]\!](\sigma)] + \pi_{\text{rv}}(\sigma) + \pi_x(\sigma_p)) & [\![a_0]\!](\sigma) \geq \min([\![\delta_0]\!]) \wedge x \in \sigma \\ \bot_\sigma & \textbf{else} \end{cases}$$

Figure 8: Denotational semantics of densities

value. The new environment will have one or more random variables assigned to new values randomly chosen according to the sampler's distribution and the value of the source of randomness.

Sampling. The syntax $v[\rho]$:= `sample` d constructs a sampler from the density d. The sampler updates σ so that the mapped value of $(v, [\![\rho]\!](\sigma))$ is overwritten with the newly sampled value. I specify the denotation of the sample command via *inverse transform sampling.*

Composition. A developer can compose two samplers s_1 and s_2 with the syntax s_1 ; s_2. This feeds the output state of s_1 into s_2.

Conditionals. The syntax `if` ϕ `then` s_t `else` s_f returns the value of the sampler s_t if the constraint ϕ is true, and that ofs_f if the constraint is false.

Invocation. The syntax $x(\hat{a})$ invokes a sampler named x that exists in the environment. The invocation evaluates the sampler in an environment where the quantified variables are rebound to their parameters $\hat{a}$. For recursive procedures, if the first argument falls outside the domain, the denotation of call is the default value σ. Shuffle's type system enforces that the set of random variables in this case is empty (see Section 5.5), and the identity function returning σ is always a valid sampler for an empty variable set.

4.5 Kernels

Figure 10 presents the denotation of a kernel. 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.

$$[\![v\,\texttt{[}\rho\texttt{]}\ \texttt{:=}\ \texttt{sample}\ d]\!](\sigma, \mathrm{sr}) = \sigma[(v, [\![\rho]\!](\sigma)) \mapsto \arg\min_r \left(\int_{x \in (-\infty, r]} [\![d]\!](\sigma[(v, [\![\rho]\!](\sigma)) \mapsto x]) \right) > \mathrm{sr}]$$

$$[\![s_1\ ;\ s_2]\!](\sigma, \mathrm{sr}) = [\![s_2]\!]([\![s_1]\!](\sigma, \mathrm{sr}^1), \mathrm{sr}^0) \qquad [\![\texttt{if}\ \phi\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!](\sigma) = \begin{cases} [\![s_1]\!](\sigma) & [\![\phi]\!](\sigma) \\ [\![s_2]\!](\sigma) & \textbf{else} \end{cases}$$

$$[\![x(a_0, \dots)]\!](\sigma) = \text{ let } (q_0, \dots), \sigma_p, \rho = \sigma(x) \text{ in } \begin{cases} [\![\rho]\!](\pi_q(\sigma_p)[q_0 \mapsto [\![a_0]\!](\sigma)] \dots [q_n \mapsto [\![a_n]\!](\sigma)] + \pi_{\mathrm{rv}}(\sigma) + \pi_x(\sigma_p)) & x \in \sigma \\ \bot_\sigma & \textbf{else} \end{cases}$$

$$[\![x(a_0, \dots)]\!](\sigma) = \text{ let } (q_0, \dots), \sigma_p, \rho, \texttt{rec} = \sigma(x) \text{ in}$$

$$\begin{cases} \sigma & [\![a_0]\!](\sigma) < \min([\![\delta_0]\!]) \\ [\![\rho]\!](\pi_q(\sigma_p)[q_0 \mapsto [\![a_0]\!](\sigma)] \dots [q_n \mapsto [\![a_n]\!](\sigma)] + \pi_{\mathrm{rv}}(\sigma) + \pi_x(\sigma_p)) & [\![a_0]\!](\sigma) \geq \min([\![\delta_0]\!]) \wedge x \in \sigma \\ \bot_\sigma & \textbf{else} \end{cases}$$

Figure 9: Denotational semantics of samplers.

$$
\begin{array}{lcl}
[\![\texttt{lift}\ s]\!] & = & [\![s]\!] \\
\forall f.\ \int_{\mathrm{sr}} f([\![\texttt{fix}\ k]\!](\sigma, \mathrm{sr})) & = & \int_{\mathrm{sr}} f([\![s;\ \texttt{fix}\ s]\!](\sigma, \mathrm{sr}))
\end{array}
$$

Figure 10: Denotational semantics of kernels (Abbreviated)

Composition. A developer can compose two kernels with the syntax k_1 ; k_2. This feeds the output state of k_1 into k_2.

Conditionals. The syntax `if` ϕ `then` k_t `else` k_f returns the value of the kernel k_t if the constraint ϕ is true, and that ofk_f if the constraint is false.

Fixed Point. 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.

4.6 Estimators

Figure 11 presents the semantics of 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 modifies the weight of the estimator e.

$$[\![\texttt{factor}\ e\ \texttt{by}\ d]\!](\sigma, \mathrm{sr}) = \text{ let } (w, \sigma') = [\![e]\!](\sigma, \mathrm{sr}) \text{ in } (w * [\![d]\!](\sigma'), \sigma') \qquad [\![\texttt{if}\ \phi\ \texttt{then}\ e_t\ \texttt{else}\ e_f]\!](\sigma) = \begin{cases} [\![e_t]\!](\sigma) & [\![\phi]\!](\sigma) \\ [\![e_f]\!](\sigma) & \textbf{else} \end{cases}$$

$$[\![\texttt{elift}\ s]\!](\sigma, \mathrm{sr}) = (1, [\![s]\!](\sigma, \mathrm{sr}))$$

$$[\![x(a_0, \dots)]\!](\sigma) = \text{ let } (q_0, \dots), \sigma_p, \rho = \sigma(x) \text{ in } \begin{cases} [\![\rho]\!](\pi_q(\sigma_p)[q_0 \mapsto [\![a_0]\!](\sigma)] \dots [q_n \mapsto [\![a_n]\!](\sigma)] + \pi_{\mathrm{rv}}(\sigma) + \pi_x(\sigma_p)) & x \in \sigma \\ \bot_\sigma & \textbf{else} \end{cases}$$

$$[\![x(a_0, \dots)]\!](\sigma) = \text{ let } (q_0, \dots), \sigma_p, \rho, \texttt{rec} = \sigma(x) \text{ in}$$

$$\begin{cases} (1, \sigma) & [\![a_0]\!](\sigma) < \min([\![\delta_0]\!]) \\ [\![\rho]\!](\pi_q(\sigma_p)[q_0 \mapsto [\![a_0]\!](\sigma)] \dots [q_n \mapsto [\![a_n]\!](\sigma)] + \pi_{\mathrm{rv}}(\sigma) + \pi_x(\sigma_p)) & [\![a_0]\!](\sigma) \geq \min([\![\delta_0]\!]) \wedge x \in \sigma \\ \bot_\sigma & \textbf{else} \end{cases}$$

Figure 11: Denotational semantics for estimators

Conditionals. The syntax `if` ϕ `then` e_t `else` e_f returns the value of the estimator e_t if the constraint ϕ is true, and that ofe_f if the constraint is false.

Invocation. The syntax $x(\hat{a})$ invokes an estimator named x that exists in the environment. The call evaluates the estimator in an environment where the quantified variables are rebound to their parameters $\hat{a}$. For recursive procedures, if the first argument falls outside the domain, the denotation of call is the default value $(1, \sigma)$. Shuffle's type system enforces that the set of random variables in this case is empty (see Section 5.5), and the function returning $(1, \sigma)$ is always a valid estimator for an empty variable set.

5 Type System

In this section I present Shuffle's type system. A typing judgment is a logical proposition of the form $\mathcal{M}, \Gamma, \mathcal{L} \vdash \rho : \tau$ where $\mathcal{M}$ is a model, Γ is a type environment, $\mathcal{L}$ is an *assumption log*, ρ is a Shuffle inference program, and τ is a type from the following grammar:

$$\tau \to \delta \mid (\delta, \delta) \mid \mathbb{B} \mid T \mid (q^*, \delta^*, T) \mid (q^*, \delta^*, T, \texttt{rec})$$

In this grammar, δ is a domain, q is a quantified variable, $\mathbb{B}$ is the boolean type, and T is a Shuffle type as specified in Figure 5.

For example, the type judgment $\mathcal{M}, \Gamma, \mathcal{L} \vdash d$: `density(`$A|B$`,`ϕ`)` states that under the model $\mathcal{M}$, type environment Γ, and assumption log $\mathcal{L}$, the Shuffle inference procedure d is, when ϕ is true, a density for the conditional distribution $\Pr(A|B)$.

5.1 Model

A Shuffle model $\mathcal{M} = (\mathit{DDecls}, \mathit{VDecls}, \mathit{DDefs})$ is a set of (optional) domain declarations, variable declarations, and density definitions.

Joint Density. Define the *joint density* of all variables in the model, $\mathcal{J}$, as follows. Let

$$\texttt{def } x_i \texttt{ (}q_0^i \texttt{ in } \delta_0^i, \ldots\texttt{)}: T_{bi}(A_i|B_i, \phi_i) = \rho_i$$

be a declaration from the model. Then, defining

$$\mathcal{J}(\sigma) = \prod_{i \in |DDefs, \hat{n} \in (\delta_0^i, \ldots)} \begin{cases} [\![\rho_i]\!](\sigma[\hat{q}^i \mapsto \hat{n}]) & [\![\phi_i]\!](\sigma[\hat{q}^i \mapsto \hat{n}]) \\ 1 & \textbf{else} \end{cases}$$

I 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 $\Sigma_{\text{rv}} \to \mathbb{R}^+$.

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

$$\texttt{def } x_i \texttt{ (}q_0^i \texttt{ in } \delta_0^i, \ldots, q_n^i \texttt{ in } \delta_n^i\texttt{)}: t_{bi}(A_i|B_i, \phi_i) = \rho_i$$

assuming the predicate $\text{WF}_\text{i}(\sigma)$ is defined as follows:

$$\text{WF}_i(\sigma) = (q_0^i \in \text{dom}(\sigma) \wedge \sigma(q_0^i) \in [\![\delta_0^i]\!]) \wedge \cdots \wedge (q_n^i \in \text{dom}(\sigma) \wedge \sigma(q_n^i) \in [\![\delta_n^i]\!])$$

1. For every variable (v, n) in the model's space, there exists a σ and exactly one i such that $\text{WF}_\text{i}(\sigma) \wedge [\![\phi_i]\!](\sigma) \wedge ((v, n) \in [\![A_i]\!](\sigma))$

2. There exists a strict partial order $\prec$ such that

$$\forall \sigma, \text{WF}_\text{i}(\sigma) \Rightarrow (v_1, n_1) \in [\![A]\!](\sigma), (v_2, n_2) \in [\![B]\!](\sigma).\ (v_1, n_1) \prec (v_2, n_2)$$

3. $\forall \sigma, \text{WF}_\text{i}(\sigma) \wedge [\![\phi_i]\!](\sigma) \Rightarrow \int_{[\![A_i]\!](\sigma)} [\![\rho_i]\!](\sigma) = 1$

4. $\forall i. \text{Valid}(A_i|B_i, \phi_i)$

5. For all σ such that $\mathrm{WF}_i(\sigma)$ and $[\![\phi_i]\!](\sigma)$, $[\![A_i]\!](\sigma) \neq \bot_\sigma$ and $[\![B_i]\!](\sigma) \neq \bot_\sigma$

5.2 Type Environment

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

$$\Gamma \to \emptyset \mid \Gamma :: [\beta : \tau]$$
$$\beta \to x \mid q \mid v$$

where x is a named distribution, q is a quantified variable, τ is a type from the language described above.

5.3 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

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

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

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

Reachability. The predicate ReachesAll(s) states that a given sampler s reaches every value in its output space with positive probability. In other words, for any variable $(v, n) \in \mathcal{V}$, if s modifies v, n, then there must be some positive probability of reaching every value of n in n's domain.

$$\mathcal{M}, \mathcal{L} \vDash \text{ReachesAll}(s) = \forall \sigma.\ [\![\mathcal{L}]\!](\sigma) \Rightarrow [\![\text{ReachesAll}(s)]\!](\sigma)$$

5.4 Types

Model Relation. An environment σ *models*, 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

$$\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L} =$$

$$\mathcal{J}(\sigma) > 0 \wedge$$

$$\forall v.\ \mathcal{M}, \Gamma, \mathcal{L} \vdash v : (\delta_1, \delta_2) \Rightarrow (\forall n \in [\![\delta_1]\!].\ \sigma((v, n)) \in [\![\delta_2]\!]) \wedge$$

$$\forall q.\ \mathcal{M}, \Gamma, \mathcal{L} \vdash q : \delta \Rightarrow (q \in \text{dom}(\sigma) \wedge \sigma(q) \in [\![\delta]\!]) \wedge$$

$$\Big(\forall x\ \exists \Gamma_p.\ \mathcal{M}, \Gamma, \mathcal{L} \vdash x : ((q_0, \ldots, q_n), (\delta_0, \ldots, \delta_n), t) \Rightarrow \sigma(x) =$$

$$((q_0, \ldots, q_n), \sigma_p, \rho) \wedge \sigma_p \vDash \Gamma_p, \mathcal{M}, \mathcal{L} \wedge \mathcal{M}, \Gamma_p :: [q_0 : \delta_0] :: \cdots :: [q_n : \delta_n], \mathcal{L} \vDash \rho : t\Big) \wedge$$

$$\Big(\forall x\ \exists \Gamma_p.\ \mathcal{M}, \Gamma, \mathcal{L} \vdash x : ((q_0, \ldots, q_n), (\delta_0, \ldots, \delta_n), t, \texttt{rec}) \Rightarrow \sigma(x) =$$

$$((q_0, \ldots, q_n), \sigma_p, \rho, \texttt{rec}) \wedge \sigma_p \vDash \Gamma_p, \mathcal{M}, \mathcal{L} \wedge$$

$$\mathcal{M}, \Gamma_p :: [x : ((q_0, \ldots, q_n), (\delta_0, \ldots, q_n), t, \texttt{rec})] :: [q_0 : \delta_0] :: \cdots :: [q_n : \delta_n], \mathcal{L} \vDash \rho : t\Big)$$

and note that, for any model $\mathcal{M}$, there is an environment and a type environment which each contain mappings for all densities and random variables in in the model, and these environments satisfy $\vDash$.

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

Statement 1 (Density). *If* $\mathcal{M}, \Gamma, \mathcal{L} \vDash d : \texttt{density}(A|B, \phi)$*, then for any environment* σ *such that* $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$*,*

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

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 $P(A|B)$. A sampler must also preserve the model relation on its output.

Statement 2 (Sampler). *If* $\mathcal{M}, \Gamma, \mathcal{L} \vDash s : \texttt{sampler}(A|B, \phi)$*, then for all* σ *such that* $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$ *and all* $f \in \Sigma_{\text{rv}} \to \mathbb{R}^+$*,*

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

and

$$[\![\phi]\!](\sigma) \Rightarrow [\![s]\!](\sigma, \text{sr}) \vDash \Gamma, \mathcal{M}, \mathcal{L}$$

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. A kernel must also preserve the model relation on its output.

Statement 3 (Kernel). *If* $\mathcal{M}, \Gamma, \mathcal{L} \vDash k : \texttt{kernel}(A|B, \phi)$*, then for all* σ *such that* $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$*, for any* $f \in \Sigma_{\text{rv}} \to \mathbb{R}^+$*, and* s *such that* $\mathcal{M}, \Gamma, \mathcal{L} \vDash s : \texttt{sampler}(A|B, \phi)$*,*

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. $[\![\phi]\!](\sigma) \Rightarrow \int_{\text{sr}^0, \text{sr}^1} f(k(s(\sigma, \text{sr}^0), \text{sr}^1)) = \int_{[\![A]\!](\sigma)} f(\sigma) * \mathcal{J}([\![A]\!](\sigma)|[\![B]\!](\sigma))(\sigma)$
3. $[\![\phi]\!](\sigma) \Rightarrow [\![s]\!](\sigma, \text{sr}) \vDash \Gamma, \mathcal{M}, \mathcal{L}$

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. An estimator must also preserve the model relation on the sample portion of its output.

Statement 4 (Estimator). *if* $\mathcal{M}, \Gamma, \mathcal{L} \vDash e$: `estimator`$(A|B, \phi)$ *then for all* σ *such that* $\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$*, and any* $f \in \Sigma_{rv} \to \mathbb{R}^+$,

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

and

$$[\![\phi]\!](\sigma) \Rightarrow \pi_0([\![e]\!](\sigma, \text{sr})) \vDash \Gamma, \mathcal{M}, \mathcal{L}$$

Quantified Types. Shuffle exports inference procedures that are functions of quantified variables. The function is function is correct if instantiations of the function body are correct. This relationship is defined by the equality

$$\mathcal{M}, \Gamma, \mathcal{L} \vDash \rho : (q_0, \ldots, q_n), (\delta_0, \ldots, \delta_n), t \iff$$
$$\mathcal{M}, \Gamma :: [q_0 : \delta_0] :: \cdots :: [q_n : \delta_n], \mathcal{L} \vDash \rho : t$$

Valid Types. The variable sets A and B and constraint ϕ in a type $t_b(A|B, \phi)$ can depend on the values of random variables. A type is *valid*, written Valid$(A|B, \phi)$ if all of these variables are contained in the set of conditioned random variables B. Using the notation $v \cap B$ to mean the variables $(v, n) \in B$ that have a given variable name $v \in V$, a type is valid if

1. There exists a sequence $v_0, v_1, v_2, \ldots, v_n$ on the random variable names $v_i \in \mathcal{V}$ such that $\forall \sigma_1, \sigma_2, \Big(\forall q.\ \sigma_1(q) = \sigma_2(q)\Big) \Rightarrow [\![v_0 \cap B]\!](\sigma_1) = [\![v_0 \cap B]\!](\sigma_2)$, and for any $n > 0$,

$$\forall \sigma_1, \sigma_2, \Big(\forall q.\ \sigma_1(q) = \sigma_2(q)\Big) \wedge$$
$$\forall n_2 < n.\ [\![v_{n_2} \cap B]\!](\sigma_1) = [\![v_{n_2} \cap B]\!](\sigma_2) \wedge$$
$$\Big(\forall (v, n_3) \in [\![v_{n_2} \cap B]\!](\sigma_1).\ \sigma_1(v_{n_2}, n_3) = \sigma_2(v_{n_2}, n_3)\Big) \Rightarrow$$
$$[\![v_n \cap B]\!](\sigma_1) = [\![v_n \cap B]\!](\sigma_2)$$

2. For all σ_1, σ_2,

$$\Big(\forall q.\ \sigma_1(q) = \sigma_2(q)\Big) \wedge [\![B]\!](\sigma_1) = [\![B]\!](\sigma_2) \wedge \Big(\forall (v,n) \in [\![B]\!](\sigma_1), \sigma_1(v,n) = \sigma_2(v,n)\Big)$$

$$\Rightarrow [\![A]\!](\sigma_1) = [\![A]\!](\sigma_2)$$

3. For all σ_1, σ_2,

$$\Big(\forall q.\ \sigma_1(q) = \sigma_2(q)\Big) \wedge [\![B]\!](\sigma_1) = [\![B]\!](\sigma_2) \wedge \Big(\forall (v,n) \in [\![B]\!](\sigma_1), \sigma_1(v,n) = \sigma_2(v,n)\Big)$$

and

$$\Rightarrow [\![\phi]\!](\sigma_1) = [\![\phi]\!](\sigma_2)$$

4. For all σ, $[\![A]\!](\sigma) \cap [\![B]\!](\sigma) = \emptyset$

These enforce the following properties with respect to the type $t_b(A|B, \phi)$:

1. **Valid Conditions.** For the set of conditioned variables B, there must be an order $\prec$ on the set of variable names $\mathcal{V}$ such that $v_1 \prec v_2$ means that the subset of variables in B with name v_1 can be computed without reference to the subset with name v_2. Thus, for example, choosing B as

```
obs{i in Samples:  z[i] == z[k] && i != k}, z{i in Samples:  i !=
                                k}
```

would have the order `z` $\prec$ `obs`. By contrast, choosing B as any of the following would be invalid

```
obs{i in S: z[i] == z[k]}, z{i in S: i != k}
z{i in S: z[i] == j}
a{i in D: b[i] == b[k]}, b{i in D: a[i] == a[k]}
```

2. **Valid Variables.** The random variable set A must be well defined given the conditioned variables B. This means that any variable that the value of A depends on must be contained in B.

3. **Valid Constraints.** The constraint ϕ must be well defined given the conditioned variables B. This means that any variable that the value of ϕ depends on must be contained in B.

4. **Disjointness.** The variable sets A and B must be disjoint.

5.5 Type Rules

Substitution. I denote standard capture-avoiding substitution on the free quantified variables of a type t by the notation $t[\rho/q]$ where ρ is a term and q is a quantified variable.

Variable Restriction. The notation $A \downarrow \phi$ refers to a random variable set A can be restricted to the subset of A such that the constraint ϕ holds. The semantics of this is defined with respect to a desugaring of A to a the following form:

$$A = v_0\{q_0 \texttt{ in } \delta_0\texttt{: } \phi_0\}, \ldots, v_n\{q_n \texttt{ in } \delta_n\texttt{: } \phi_n\}$$

The definition of $A \downarrow \phi$ is then

$$A \downarrow \phi = v_0\{q_0 \texttt{ in } \delta_0\texttt{: } \phi_0 \texttt{ \&\& } \phi\}, \ldots, v_n\{q_n \texttt{ in } \delta_n\texttt{: } \phi_n \texttt{ \&\& } \phi\}$$

Densities, Figure 12 shows the typing rules for probability densities. These follow the rules of conditional probability, and give the developer the ability to multiply, integrate, and divide densities in a typesafe manner. The DMUL rule requires a check that the type is valid because it might be the case that, for instance, the set of random variables A depends on the values of random variables in B, which would render the type `density(`A`,`$B|C$`,`ϕ`)` invalid.

Samplers. Figure 13 shows the typing rules for samplers. These give the developer the ability to sample from one or more random variables in the model and verify the sampling operations are correct. The SBIND rule requires a check that the type

DMUL

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 : \texttt{density}(A|B,C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash d_1 : \texttt{density}(B|C,\phi) \quad \mathcal{M},\Gamma \vDash \text{Valid}(A,B|C,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 * d_2 : \texttt{density}(A,B|C,\phi)}$$

DDIV

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 : \texttt{density}(A,B|C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash d_2 : \texttt{density}(B|C,\phi) \quad \mathcal{M},\Gamma \vDash \text{Valid}(A|B,C,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash d_1 \,/\, d_2 : \texttt{density}(A|B,C,\phi)}$$

DDIV2

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

DINT

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

Figure 12: Type rules for probability densities

SLIFT

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash d : \texttt{density}(v[\rho]|B,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash v[\rho]\ :=\ \texttt{sample}\ d : \texttt{sampler}(v[\rho]|B,\phi)}$$

SBIND

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash s_1 : \texttt{sampler}(B|C,\phi) \quad \mathcal{M},\Gamma,\mathcal{L} \vdash s_2 : \texttt{sampler}(A|B,C,\phi) \quad \mathcal{M},\Gamma \vDash \text{Valid}(A,B|C,\phi)}{\mathcal{M},\Gamma,\mathcal{L} \vdash s_1\ ;\ s_2 : \texttt{sampler}(A,B|C,\phi)}$$

Figure 13: Type rules for samplers

KLIFT

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

KCOMBINE

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash k_1 : \texttt{kernel}(A|B, C, \phi) \quad \mathcal{M}, \Gamma, \mathcal{L} \vdash k_2 : \texttt{kernel}(B|A, C, \phi) \qquad \mathcal{M}, \Gamma \vDash \text{Valid}(A, B|C, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash k_1\ ;\ k_2 : \texttt{kernel}(A, B|C, \phi_1\ \texttt{\&\&}\ \phi_2)}$$

KFIX

$$\frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash k : \texttt{kernel}(A|B, \phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{fix}\ k : \texttt{sampler}(A|B, \phi)}$$

Figure 14: Type rules for kernels

is valid because it might be the case that, for instance, the set of random variables A depends on the values of random variables in B, which would render the type $\texttt{density}(A, B|C, \phi)$ invalid.

Kernels. Figure 14 shows the typing rules for kernels. These give the developer the ability to perform Markov-chain Monte Carlo sampling for subset of the random variables by providing the guarantee that the kernel, if repeatedly applied to an environment, converges to a sampler for the distribution described by the type. The K2 rule requires a check that the type is valid because it might be the case that, for instance, the set of random variables A depends on the values of random variables in B, which would render the type $\texttt{kernel}(A, B|C, \phi)$ invalid. The preconditions of the KLIFT rule maintain that 1) Any kernel in Shuffle must sample over a finite distribution and 2) The kernel must produce every value in its output space with positive probability. This means that every kernel representable in Shuffle must admit an approximate implementation of `fix` [4].

Estimators. Figure 15 shows the typing rules for estimators. These give the developer the ability to to conduct *likelihood weighting*. The EFACT rule requires a check that the type is valid because it might be the case that, for instance, the set

$$\text{ELIFT}\quad \frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash s : \texttt{sampler}(A|B,\phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{elift}\ s : \texttt{estimator}(A|B,\phi)}$$

$$\text{EFACT}\quad \frac{\mathcal{M}, \Gamma, \mathcal{L} \vdash e : \texttt{estimator}(A|B,\phi) \qquad \mathcal{M}, \Gamma, \mathcal{L} \vdash d : \texttt{density}(C|A,B,\phi) \qquad \mathcal{M}, \Gamma \vDash \text{Valid}(A|B,C,\phi)}{\mathcal{M}, \Gamma, \mathcal{L} \vdash \texttt{factor}\ e\ \texttt{by}\ d : \texttt{estimator}(A|B,C,\phi)}$$

Figure 15: Type rules for estimators

of random variables C depends on the values of random variables in A, which would render the type $\texttt{estimator}(A|B,C,\phi)$ invalid.

5.6 Structural Rules

Conditionals. Figure 16 shows Shuffle's IF rule for handling conditionals. This rule enables the developer to construct inference procedures whose behavior differs based on whether a constraint is true or false. The system must check that the resulting type is valid, because the constraint may introduce an illegal dependency in the random variables or the constraint in the resulting type.

Definition and Invocation. Figure 16 shows Shuffle's rules for defining and invoking inference procedures. These give the developer the ability to encapsulate components of the inference procedure while ensuring these components are correctly defined and invoked. Shuffle checks type validity on invocations because for two reasons: 1) A type written by the developer must be valid, and 2) the substitution in the invocation rules (INV and INV-REC in Figure 16) may yield an invalid type if, for example, it results in A depending on the value of a random variable that is not in B.

Recursion. Shuffle's type system imposes restrictions on recursive programs. The INV-REC rule in Figure 16 enforces that procedures may only recurse on their first argument, and must pass all other arguments through unchanged. The DEF-REC rule

IF

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash \rho_t : t_b(A_t|B_t, \phi_t) \qquad \mathcal{M},\Gamma,\mathcal{L} \vdash \rho_f : t_b(A_f|B_f, \phi_f) \qquad \mathcal{M},\Gamma,\mathcal{L} \vdash \phi_i : \mathbb{B}}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{if } \phi_i \texttt{ then } \rho_t \texttt{ else } \rho_f : t_b(A_t \downarrow \phi_i, A_f \downarrow \neg\phi_i \mid B_t \downarrow \phi_i, B_f \downarrow \neg\phi_i, (\phi_i \texttt{ \&\& } \phi_t) \texttt{ || } (\neg\phi_i \texttt{ \&\& } \phi_f))}$$

DEF

$$\frac{\begin{array}{c}\mathcal{M},\Gamma :: [q_0 : \delta_0] :: \cdots :: [q_n : \delta_n], \mathcal{L} \vdash \rho_1 : t_1 \\ \mathcal{M},\Gamma :: [x : ((q_0,\ldots,q_n),(\delta_0,\ldots,\delta_n),t_1)], \mathcal{L} \vdash p_2 : (\hat{qv}, \hat{\delta}, t_2)\end{array}}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{def } x \texttt{ (}q_0 \texttt{ in } \delta_0, \ldots, q_n \texttt{ in } \delta_n\texttt{) : } t_1 = \rho_1 \texttt{ ; } p_2 : (\hat{qv}, \hat{\delta}, t_2)}$$

DEF-REC

$$\frac{\begin{array}{c}\mathcal{M},\Gamma :: [x : ((q_0,\ldots,q_n),(\delta_0,\ldots,\delta_n),t_1,\texttt{rec})] :: [q_0 : \delta_0] :: \cdots :: [q_n : \delta_n], \mathcal{L} \vdash \rho_1 : t_1 \\ \forall\sigma.\ [\![q_0]\!](\sigma) < \min([\![\delta_0]\!]) \Rightarrow [\![A]\!](\sigma) = \emptyset \qquad \mathcal{M},\Gamma \vDash \text{Valid}(A|B,\phi) \\ \mathcal{M},\Gamma :: [x : ((q_0,\ldots,q_n),(\delta_0,\ldots,\delta_n),t_1)], \mathcal{L} \vdash p_2 : (\hat{q},\hat{\delta},t_2)\end{array}}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{def rec } x \texttt{ (}q_0 \texttt{ in } \delta_0, \ldots, q_n \texttt{ in } \delta_n\texttt{) : } t_1 = \rho_1 \texttt{ ; } p_2 : (\hat{q},\hat{\delta},t_2)}$$

DEF-IND

$$\frac{\begin{array}{c}\mathcal{M},\Gamma :: [q_0 : \delta_0] :: \cdots :: [q_n : \delta_n], \mathcal{L} \vdash_I \rho_1 : t_1 \\ \mathcal{M},\Gamma :: [x : (q_0,\ldots,q_n),(\delta_0,\ldots,\delta_n),t_1], \mathcal{L} \vdash p_2 : (\hat{q},\hat{\delta},t_2)\end{array}}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{def independent } x \texttt{ (}q_0 \texttt{ in } \delta_0, \ldots, q_n \texttt{ in } \delta_n\texttt{) : } t_1 = \rho_1 \texttt{ ; } p_2 : (\hat{q},\hat{\delta},t_2)}$$

EXP

$$\frac{\mathcal{M},\Gamma :: [q_0 : \delta_0] :: \cdots :: [q_n : \delta_n], \mathcal{L} \vdash \rho : t}{\mathcal{M},\Gamma,\mathcal{L} \vdash \texttt{def export } x \texttt{ (}q_0 \texttt{ in } \delta_0, \ldots, q_n \texttt{ in } \delta_n\texttt{) : } t = \rho : ((q_0,\ldots,q_n),(\delta_0,\ldots,\delta_n),t)}$$

INV

$$\frac{\begin{array}{c}\mathcal{M},\Gamma,\mathcal{L} \vdash x : \hat{q},\hat{\delta},t_b(A|B,\phi) \qquad \mathcal{M},\Gamma,\mathcal{L} \vdash \hat{a} : \hat{\delta} \\ \text{Valid}(A[\hat{a}/\hat{q}]|B[\hat{a}/\hat{q}], \phi[\hat{a}/\hat{q}])\end{array}}{\mathcal{M},\Gamma,\mathcal{L} \vdash x(\hat{a}) : t_b(A|B,\phi)[\hat{a}/\hat{q}]}$$

INV-REC

$$\frac{\mathcal{M},\Gamma,\mathcal{L} \vdash x : (q_0, q_1, \ldots, q_n), \hat{\delta}, t, \texttt{rec} \qquad n > 0}{\mathcal{M},\Gamma,\mathcal{L} \vdash x(q_0 - n, q_1, \ldots, q_n) : t[q_0 - n/q_0]}$$

Figure 16: Type rules for if statements, definitions, and invocations.

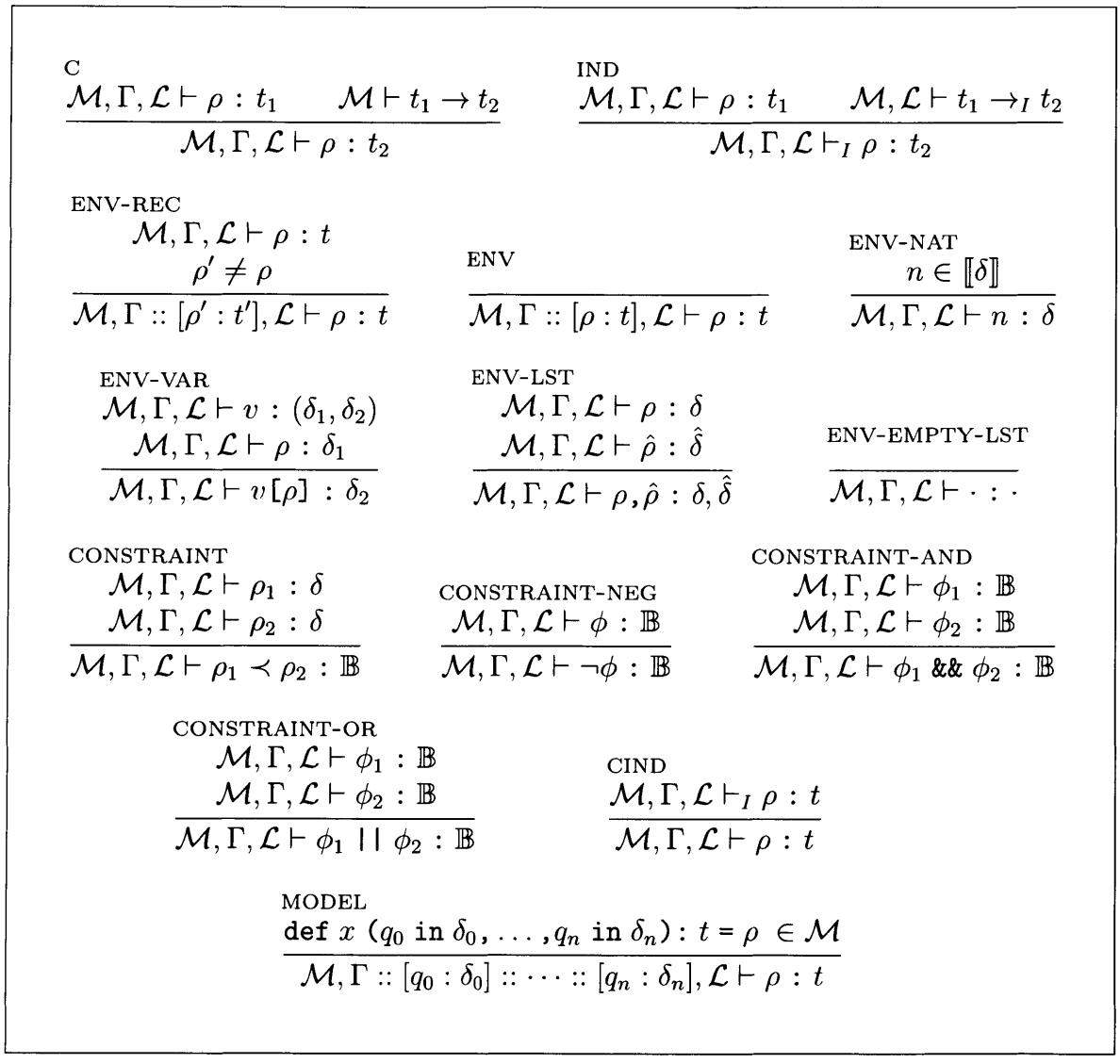

Figure 17: Type rules for coercions and the type environment.

$$\text{L1}\quad \frac{\mathcal{M} \vDash \phi_2 \Rightarrow A \equiv C \qquad \mathcal{M} \vDash \phi_2 \Rightarrow B \equiv D \qquad \mathcal{M} \vDash \phi_2 \Rightarrow \phi_1}{\mathcal{M} \vdash t_b(A|B, \phi_1) \rightarrow t_b(C|D, \phi_2)}$$

$$\text{L2}\quad \frac{\mathcal{M}, \mathcal{L} \vDash \phi \Rightarrow A \perp\!\!\!\perp C \mid B \qquad \mathcal{M} \vDash \phi \Rightarrow (A \cap C = \emptyset)}{\mathcal{M}, \mathcal{L} \vdash t_b(A|B, \phi) \rightarrow_I t_b(A|B, C, \phi)}$$

Figure 18: Rules for coercion side predicates

enforces that a recursive procedure's base case always corresponds to the set of result random variables A being empty. This justifies the default base cases in Shuffle's semantics.

Additional Structural Rules. Figure 17 shows additional structural rules that connect different pieces of Shuffle's type system together. These include 1) ENV-REC, ENV, ENV-NAT, ENV-VAR, ENV-LST, and ENV-EMPTY-LST rules that instantiate types from the environment and determine whether Shuffle terms belong to a named domain 2) C, IND, and CIND rules which apply *normal* and *independent* coercions (see Section 5.7 3) CONSTRAINT, CONSTRAINT-AND, CONSTRAINT-NEG, and CONSTRAINT-OR rules which ensure constraints have the boolean type $\mathbb{B}$, and 3) the MODEL rule instantiating the types in the model $\mathcal{M}$, which serve as axioms for Shuffle's type system.

5.7 Coercions

Figure 18 shows Shuffle's rules for type 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 that encode logical formulae. Shuffle employs the Z3 theorem prover [14] to verify that these predicates are true. These predicates are:

- $\mathcal{M} \vDash \phi \Rightarrow A \equiv B$. 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

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

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.

5.8 Properties

Integration by Substitution. The following proof sketches make use of 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.

Equivalence of Substitution and Environment Mapping. The following proof sketches make use of a lemma. Let ρ be a Shuffle term that is either a variable set or a constraint. We have the following equivalences, for any environment σ:

$$[\![\rho[a/q]]\!](\sigma) = [\![\rho]\!](\sigma[q \mapsto [\![a]\!](\sigma))$$

I will now show these properties are sufficient to prove that Shuffle's type system guarantees that derivable types and programs are computable.

Theorem 1 (Type Validity). *Assume that*

1. $\mathcal{M}, \Gamma, \mathcal{L} \vdash \rho : t_b(A|B, \phi)$
2. $\sigma \vDash \mathcal{M}, \Gamma, \mathcal{L}$

Then, it must be true that

1. *Neither* $[\![A]\!](\sigma)$ *nor* $[\![B]\!](\sigma)$ *is ever* $\perp_\sigma$.
2. Valid$(A|B, \phi)$

Proof Sketch. The proof follows from induction on the structure of derivations for types. Specific cases are outlined below.

DMUL. 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 assumptions.

DDIV and DDIV2. 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 $\perp_\sigma$. Checking type validity is not necessary in DDIV2 because DDIV2 adds no variables to the type.

DINT. The conclusions follow straightforwardly from the properties of A, B. Checking type validity is not necessary because DINT adds no variables to the type.

SLIFT. The conclusions follow straightforwardly because the parameters of the type are the same.

SBIND. The conclusions follow from the same reasoning as that of DMUL.

KLIFT. The conclusions follow straightforwardly because the parameters of the type are the same.

K2. If A, C and B, C are not $\bot_\sigma$, then A, B, and C must all be mutually disjoint, which means that A, B is not $\bot_\sigma$ and A, B is disjoint from C. The final conclusion follows from the assumptions.

KFIX. The conclusions follow straightforwardly because the parameters of the type are the same.

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. According to the semantics of ",", if A and B are disjoint, and A, B is well-defined and disjoint from C, then A is disjoint from B, C and B, C is well-defined. The validity of the type follows from the assumptions.

IF. The disjointness follows from analyzing separately the cases where ϕ is true and where ϕ is false. The variable $A_t \downarrow \phi_i$ must be disjoint from the variable $A_e \downarrow \neg\phi_i$, so the variable $A_t \downarrow \phi_i, A_e \downarrow \phi_i$ is never $\bot_\sigma$. Because of the assumption that $\mathcal{M}, \Gamma, \mathcal{L} \vdash \phi_i : \mathbb{B}$, $[\![\phi_i]\!](\sigma)$ is not $\bot_\sigma$. A similar argument holds for $B_t \downarrow \phi_i, B_e \downarrow \phi_i$. The validity conclusion results from rule assumptions.

DEF. It must be true that

$$\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L} \Rightarrow \sigma[x \mapsto (\hat{q}, \hat{\delta}, \rho_1)] \vDash \mathcal{M}, \Gamma :: [x : (\hat{q}, \hat{\delta}, t_1)], \mathcal{L}$$

Therefore, the conclusions are true for t_2 by inductive assumption. A similar line of reasoning holds for DEF-IND and DEF-REC.

INV. To prove the disjointness condition, apply the environment-substitution lemma. The validity conclusion follows from the rule assumptions. The INV-REC rule follows similar reasoning, except that the type validity is guaranteed by inductive assumption.

MODEL. See the model validity assumptions above.

C and CIND. The conclusions follow from the fact that the side predicate $\mathcal{M}, \mathcal{L} \vdash t_1 \rightarrow t_2$ enforces that the variable sets in t_1 and t_2 are the same, and the predicate $\mathcal{M}, \mathcal{L} \vdash t_1 \rightarrow_I t_2$ enforces that all modifications to the variables preserve disjointness.

Theorem 2 (Progress). *Assume that*

1. $\mathcal{M}, \Gamma, \mathcal{L} \vdash \rho : t_b(A|B, \phi)$
2. $\sigma \vDash \mathcal{M}, \Gamma, \mathcal{L}$

Then, it must be true that

1. $[\![\rho]\!](\sigma)$ *is uniquely defined and never* $\perp_\sigma$ *or* $\perp_0$

Proof Sketch. The proof follows from induction on the structure of derivations for types. Most rules follow straightforwardly from applying the inductive assumptions and the semantic definitions. Cases that do not follow this reasoning are detailed below.

DDIV and DDIV2. For d_1 / d_2 to be well defined (unique and not $\perp_\sigma$ or $\perp_0$, d_2 must never be 0. This is guaranteed by well-formedness because it enforces that $\mathcal{J}(\sigma) > 0$.

DINT. For `int` d `by` B to be well-defined, B must never be $\perp_\sigma$. This is enforced by the type validity property above.

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 [4] 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.

DEF. It must be true that

$$\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L} \Rightarrow \sigma[x \mapsto (\hat{q}, \hat{\delta}, \rho_1)] \vDash \mathcal{M}, \Gamma :: [x : (\hat{q}, \hat{\delta}, t_1)], \mathcal{L}$$

Therefore, the conclusions are true for p_2 by inductive assumption. A similar line of reasoning holds for DEF-IND and DEF-REC.

Furthermore, Shuffle's type system guarantees that the type for a program correctly specifies the program's functional correctness:

Theorem 3 (Density Soundness).
if $\mathcal{M}, \Gamma, \mathcal{L} \vdash d$: `density`$(A|B, \phi)$ *then* $\mathcal{M}, \Gamma, \mathcal{L} \vDash d$: `density`$(A|B, \phi)$, *and*
if $\mathcal{M}, \Gamma, \mathcal{L} \vdash d : \hat{q}, \hat{\delta},$ `density`$(A|B, \phi)$, *then* $\mathcal{M}, \Gamma, \mathcal{L} \vDash d : \hat{q}, \hat{\delta},$ `density`$(A|B, \phi)$

Proof Sketch. The proof follows from induction on the derivations. Specific rules are outlined below:

MODEL. From the definition of $\mathcal{J}$, it must be true 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 $d_{i^*} = \rho$ and $\phi_{i^*} = \phi$. Furthermore, because A and B are disjoint, $[\![\rho]\!](\sigma)$ does not depend on any variables outside of A and B, and $\int_{[\![A]\!](\sigma)} [\![\rho]\!](\sigma) = 1$, the above equation simplifies to

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

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

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

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

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

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

Applying the identity from DMUL in reverse, it must be true 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. From the identity in DMUL, it must be true 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. 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.

Theorem 4 (Sampler Soundness).
if $\mathcal{M},\Gamma,\mathcal{L} \vdash s : \texttt{sampler}(A|B,\phi)$ *then* $\mathcal{M},\Gamma,\mathcal{L} \vDash s : \texttt{sampler}(A|B,\phi)$, *and*
if $\mathcal{M},\Gamma,\mathcal{L} \vdash s : \hat{q},\hat{\delta},\texttt{sampler}(A|B,\phi)$ *then* $\mathcal{M},\Gamma,\mathcal{L} \vDash s : \hat{q},\hat{\delta},\texttt{sampler}(A|B,\phi)$

Proof sketch. The proof follows from structural induction on the rules which may produce samplers. Individual cases are outlined below.

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

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

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

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

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

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

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

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

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

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

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

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

Using the identity from DMUL, this simplifies to

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

Theorem 5 (Kernel Soundness).
if $\mathcal{M}, \Gamma, \mathcal{L} \vdash k : \texttt{kernel}\,(A|B, \phi)$ *then* $\mathcal{M}, \Gamma, \mathcal{L} \vDash k : \texttt{kernel}\,(A|B, \phi)$, *and*
if $\mathcal{M}, \Gamma, \mathcal{L} \vdash k : \hat{q}, \hat{\delta}, \texttt{kernel}\,(A|B, \phi)$ *then* $\mathcal{M}, \Gamma, \mathcal{L} \vDash k : \hat{q}, \hat{\delta}, \texttt{kernel}\,(A|B, \phi)$

Proof sketch. The proof follows from structural induction on the rule derivations for kernels. The specific cases are outlined below.

KLIFT. To prove the first property, choose $\epsilon = 1$ and inline the definition of the sampler. The second condition is equivalent to the statement that if s is a sampler, then $\texttt{fix}\ s \equiv s$. In other words, 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 $\text{fix} = s$. To see this, 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. First, I will show that k_1 is invariant for the distribution $A, B|C$. This means that k_1 satisfies the *second* property for $\mathcal{M}, \Gamma, \mathcal{L} \vdash k_1 : \texttt{kernel}(A, B|C, \phi)$ 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, it must be true 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, it must hold 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, note that the ReachesAll condition applies transitively to any kernel that can be generated with the kernel rules.

Theorem 6 (Estimator Soundness).
if $\mathcal{M},\Gamma,\mathcal{L} \vdash e : \texttt{estimator}(A|B,\phi)$ *then* $\mathcal{M},\Gamma,\mathcal{L} \vDash e : \texttt{estimator}(A|B,\phi)$, *and*
if $\mathcal{M},\Gamma,\mathcal{L} \vdash e : \hat{q},\hat{\delta},\texttt{estimator}(A|B,\phi)$ *then* $\mathcal{M},\Gamma,\mathcal{L} \vDash e : \hat{q},\hat{\delta},\texttt{estimator}(A|B,\phi)$

Proof sketch. The proof follows from structural induction on the rules which may produce estimators. Individual cases are outlined below.

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

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

can be simplified to $\frac{\int_{\mathrm{sr}} f([\![s]\!]([q\mapsto a],\mathrm{sr}))}{\int_{\mathrm{sr}} 1} = \int_{\mathrm{sr}} f([\![s]\!]([q\mapsto a],\mathrm{sr}))$ 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. Using the definitions from the semantics, the expression

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

becomes

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

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.

Structural Rules

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

IF. The soundness of conditionals follows from the fact that, if σ is such that $[\![\phi]\!](\sigma)$ is true,

$$\mathcal{M}, \Gamma, \mathcal{L} \vDash \rho : T_b(\Phi(\phi_i, A_t, A_e) | \Phi(\phi_i, A_t, B_t), \Phi(\phi_i, \phi_t, \phi_e))$$
$$\iff \mathcal{M}, \Gamma, \mathcal{L} \vDash \rho : T_b(A_t | B_t, \phi_t)$$

Otherwise, since σ is such that $[\![\phi]\!](\sigma)$ is false, it must be true that

$$\mathcal{M}, \Gamma, \mathcal{L} \vDash \rho : T_b(\Phi(\phi_i, A_t, A_e) | \Phi(\phi_i, A_t, B_t), \Phi(\phi_i, \phi_t, \phi_e)) \iff$$
$$\mathcal{M}, \Gamma, \mathcal{L} \vDash \rho : T_b(A_e | B_e, \phi_e)$$

DEF. According to the definitions of $\vDash$ it must hold that

$$\sigma \vDash \Gamma, \mathcal{M}, \mathcal{L}$$
$$\Rightarrow \sigma[x \mapsto (\hat{q}, \hat{\delta}, \rho_1)] \vDash \mathcal{M}, \Gamma :: [x : (\hat{q}, \hat{\delta}, t_1)], \mathcal{L}$$

which means inductive soundness assumption for p_2 holds. The proof for the DEF-IND and DEF-REC rules are similar.

INV. Due to the assumption that $\sigma \vDash \mathcal{M}, \Gamma, \mathcal{L}$, it must hold that $\sigma(x) = (\hat{q}, \hat{\delta}, \sigma_p, \rho)$ is defined and furthermore there exists a Γ_p such that $\mathcal{M}, \Gamma_p, \mathcal{L} \vdash \rho : t$. The soundness of the INV rule follows straightforwardly from applying the inductive assumption and the substitution-environment lemma.

For INV-REC, there are two cases. If $[\![q_0]\!](\sigma) < [\![\delta_0]\!]$, then $[\![A]\!](\sigma) = \emptyset$, then because $\mathcal{J}(\emptyset | S) = 1$, it must be true that $1, \sigma, and(1, \sigma)$ are the correct values for

densities, samplers and kernels, and estimators, respectively. These are exactly the values the semantics prescribes in this case. If $[\![q_0]\!](\sigma) \geq [\![\delta_0]\!]$, then similar reasoning to INV holds.

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 $\mathcal{M}, \Gamma, \mathcal{L} \vdash \rho : t_1$ *implies* the soundness of the judgment $\mathcal{M}, \Gamma, \mathcal{L} \vdash \rho : t_2$.

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 procedure that has been type checked against the program's specified types as well as an auditable list of assumptions about the probabilistic model that must be true for the inference procedure to be correct.

6.1 Type Checking

The Shuffle system implements the type checking rules presented in Section 5. Shuffle uses the Z3 theorem prover to check assertions over sets of quantified and random variables. Shuffle models quantified variable values, random variable values, and domain bounds as 64-bit bitvectors, and constraints using a combination of bitvector comparisons and boolean operations. Shuffle checks equality and implication relations between constraints using quantifier-free bitvector theories, and checks type validity assertions through Z3's quantified bit-vector formulas.

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 7 . Shuffle's extraction procedure differs operationally from the denotational semantics in that it 1) simplifies integral expressions 2) fails to compile integral expressions it cannot simplify and 3) uses a representation for probabilities that ensures numerical stability.

For example, under the GMM presented in Figure 1, Shuffle translates the inference procedure

```
1 def export f(i in Samples, j in Mus):...=
2   obsDens(i,j) * muPrior(j)
```

to the Python code

```
1 import samplelib
2 def f(obsSamples, mu, z, i, j):
3   return samplelib.normald(obs[i],mu[j],1)
4     + samplelib.normald(mu[j],0,100)
```

Simplification Shuffle first simplifies the inference procedure by removing any type coercions and inlining any `def` statements. Shuffle translates recursive definitions to loops. After eliminating `def` statements, Shuffle simplifies integrals with known closed-form solutions. Shuffle can currently simplify conjugate and posterior-predictive distributions for Gaussian and Dirichlet distributions.

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 the code contains any unsimplified integrals over real-valued random variables, Shuffle produces an error at this stage.

7 Evaluation

In this section I evaluate the performance of extracted Shuffle inference procedures for several models.

7.1 Methodology

Shuffle sits within a landscape of probabilistic programming tools and approaches that range from fully automated systems, to systems that encourage handcoded implementations via a built-in library of primitives and – in the extreme – direct implementations in a standard programming language or implementations that leverage deep learning. Of these approaches, I evaluate Shuffle's performance in comparison to equivalent implementations in Venture.

Venture serves as an appropriate baseline whose design encourages developers to handcode implementations that directly implement density arithmetic and sampling operation with the support of coarse-grained sampling primitives that Venture provides. For example, Venture enables a programmer to use a coarse-grained primitive that performs a Gibbs update to a set of variables. Further, Venture is implemented in Python, which therefore equalizes the underlying execution platform of the two approaches.

Research Questions: Our comparison with Venture seeks to answer the following research questions:

- **Flexibility:** Is Shuffle flexible? Specifically, can Shuffle support a set of approximate inference algorithms for a set of standard models?
- **Performance:** Do Shuffle's abstractions increase or decrease performance when compared to implementations of the same algorithms within the Venture?

Benchmarks. I evaluate Shuffle and Venture on the following benchmarks:

1. **GMM.** A Gaussian mixture model similar to the one in Figure 1. This model contains 10000 datapoints and 10 cluster centers. Inference in this model uses an approximate sampler similar to the one in Figure 3.

2. **SLAM.** A scaled-up version of the Simultaneous Localization And Mapping problem [5]. This model has 500 map states, 1000 time steps, and an observation space of size 100. The inference procedure is a Rao-Blackwellized particle filter [5] with 100 particles.

3. **LDA.** A Latent Dirichlet Allocation model [2] with approximately 466k words, 50 topics, 3430 documents and an alphabet size of 6.9k. Words are assumed to be distributed evenly across the documents. Inference is performed using a collapsed Gibbs sampler [11].

For these benchmarks I consider 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 3 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 [15]. In each of the benchmarks, our use of collapsing makes sampling more efficient.

Dataset. Each benchmark uses synthetically generated observed data from the prior distribution of the statistical model. Because the comparison is between computational efficiencies of identical algorithms, I do not anticipate that changes in the data values alone will have a large impact on performance. The size of the datasets are similar in scale to real-world examples of GMM [3], SLAM [12], and LDA [19] models.

Measurement Methodology. Each test measures wall clock time (in seconds) for Shuffle and Venture. The GMM and LDA tests measure the average time for one

Benchmark	LOC		Runtime Performance (s)			Type Checking	
	Shuffle	Venture	Shuffle	Venture	Speedup	Indep.	Reach.
GMM	130	91	$1.87 * 10^{-3}$	$6.34 * 10^{-2}$	33.9x	7	1
SLAM	94	126	1579.6	2066.1	1.3x	5	0
LDA	190	25/50	$3.2 * 10^{-1}$	$1.63 * 10^{1}$	50.1x	13	1

Table 1: Shuffle vs. Venture on the benchmarks. The "Indep." column refers to the number of independence assumptions a benchmark generates, and the "Reach." column refers to the number of reachability assumptions.

Gibbs update to a single random variable. The SLAM test measures the average time to perform inference on 100 particles.

7.2 Results

Table 1 presents the results of the experiments.

Flexibility: Each benchmark, includes a lines of code (LOC) count in Table 1 for both Shuffle and Venture implementations. The GMM and SLAM benchmarks require extensions to Venture to implement custom stochastic procedures. The LDA benchmark use Venture's builtin procedure; I was able to implement LDA in Venture with 25 LOC, while the builtin procedure in Venture is 50 LOC. In benchmarks requiring custom Venture stochastic procedures, the programming effort is comparable for both systems. For LDA, Venture benefits from the ability to reuse stochastic procedures while Shuffle does not support such reuse.

Performance: Shuffle performs better than Venture on all three of the benchmarks, but this advantage is considerably reduced for SLAM because Venture can take advantage of the fact that some sampling operations in this model are deterministic.

7.3 Discussion

Shuffle is faster than Venture because Shuffle statically reasons about dependencies whereas Venture reasons about dependencies dynamically. Like Shuffle, Venture maintains, at runtime, values for each of the random variables in the model. To implement a kernel, Venture randomly resamples a subset of the variables in the state. For example, for the kernel `lift z[i] := sample ziPost(i)`, Venture resamples the random variable `z[i]` according to the *likelihood* density – which would be described in Shuffle by the type `density(obs | z)` – under each potential new value of `z[i]`. To avoid re-evaluating the entire likelihood for each value, Venture incrementalizes the value of the likelihood over the the update to `z[i]`. Venture builds a dynamic dependence graph to identify sub-expressions in the likelihood that must be updated to reflect the new value. In contrast, Shuffle statically verifies that the code for the sub-procedure `ziPost` computes the new sample for `z[i]` correctly.

8 Related Work

Shuffle builds on [1] by providing a novel semantics and programming language for typesafe probabilistic programming.

Automated Inference. Church [9] and WebPPL [10] enable a user to specify Turing-complete stochastic programs as models, but restrict inference algorithms to all-purpose algorithms such as Metropolis-Hastings [17, 13]. JAGS [22] 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, such as Venture [16] and PyMC [21] 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. Park et al. developed a language that includes samplers and other objects as first-class primitives [20]. The type of a term in their language communicates the base type of the object (e.g., a sampler). While, Shuffle shares its base operations with their language, Shuffle's novel contribution is to extend the types to describe the conditional distribution that the object represents.

Compiled Inference. AugurV2 [3] 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.

Program Transformation. Hakaru [18] enables developers to perform probabilistic inference by applying transformations to a program that specifies the underlying probabilistic model. The resulting transformed program implements an executable inference procedure for the query of interest. Shuffle's approach is complementary in that it advocates ground-up composition of inference algorithms from base primitives. In addition, Shuffle's type system can serve as a strong, well-typed interface for composing inference procedures generated by both strategies with the resulting guarantee that the overall inference procedure is correct.

The PSI solver [7] transforms probabilistic models into densities representing inference procedures. PSI can find densities for a larger class of models than Shuffle, but doesn not support samplers, kernels, or estimators.

9 Conclusion

In this thesis I 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 I have developed operators over these terms as well as type rules that associate each term with part of a probabilistic model. I have proven Shuffle's type system is sound with respect to the semantics I 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. Shuffle generates *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.

Bibliography

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

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

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

[6] Robert M. Fung and Kuo-Chu Chang. Weighing and integrating evidence for stochastic simulation on bayesian networks. In *UAI*, 1989.

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

[8] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1984.

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

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

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

[12] Jose E. Guivant and Eduardo Mario Nebot. Optimization of the simultaneous localization and map-building algorithm for real-time implementation. In *Transactions on Robotics and Automation*, 2001.

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

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

[15] 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.

[16] V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. In *ArXiv e-prints*, 2014.

[17] 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.

[18] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in hakaru (system description). In *FLOPS*, 2016.

[19] David Newman. Bag of words dataset. In *UCI Machine Learning Respository.*

[20] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 171–182, New York, NY, USA, 2005. ACM.

[21] Anand Patil, David Huard, and Christopher Fonnesbeck. Pymc: Bayesian stochastic modelling in python. 35, 2010.

[22] Martyn Plummer. *JAGS Version 4.0.0 user manual.* Addison-Wesley, Reading, Massachusetts, 2015.

[23] Nevin Lianwen Zhang and David Poole. A simple approach to bayesian network computations. In *Canadian Conference on Artificial Intelligence*, 1994.