



Probabilistic Programming with Exact Conditions

DARIO STEIN, Radboud University Nijmegen, The Netherlands

SAM STATON, University of Oxford, United Kingdom

2

We spell out the paradigm of *exact conditioning* as an intuitive and powerful way of conditioning on observations in probabilistic programs. This is contrasted with likelihood-based *scoring* known from languages such as STAN. We study exact conditioning in the cases of discrete and Gaussian probability, presenting prototypical languages for each case and giving semantics to them. We make use of categorical probability (namely Markov and CD categories) to give a general account of exact conditioning, which avoids limits and measure theory, instead focusing on restructuring dataflow and program equations. The correspondence between such categories and a class of programming languages is made precise by defining the internal language of a CD category.

CCS Concepts: • **Theory of computation** → **Denotational semantics; Categorical semantics; Probabilistic computation;**

Additional Key Words and Phrases: Probabilistic programming, exact conditioning, categorical probability theory

ACM Reference format:

Dario Stein and Sam Staton. 2024. Probabilistic Programming with Exact Conditions. *J. ACM* 71, 1, Article 2 (February 2024), 53 pages.

<https://doi.org/10.1145/3632170>

1 INTRODUCTION

Probabilistic programming is a programming paradigm that uses code to formulate generative statistical models and perform inference on them [19, 50]. Techniques from programming language theory can be used to understand modeling assumptions such as conditional independence, as well as enable optimizations that improve the efficiency of various inference algorithms (e.g., [45]). We'll also elaborate on this application in Section 1.1.

There are two different styles of conditioning on data in a probabilistic program: *scoring* and *exact conditioning*. *Scoring* features constructs to re-weight the current execution trace of the probabilistic program with a given likelihood. By contrast, *exact conditioning* focuses on a primitive operation $E_1 \coloneqq E_2$, which signifies that expressions E_1 and E_2 shall be conditioned to be exactly equal. A prototypical exact conditioning program looks as follows, where we infer some underlying value x from a noisy measurement y (see also Section 1.1):

This material is partly based on work supported by a Royal Society University Research Fellowship, ERC Project BLAST, and AFOSR Award No. FA9550-21-1-003.

Authors' addresses: D. Stein, Radboud University Nijmegen, Erasmusplein 1, Nijmegen, The Netherlands; e-mail: dario.stein@ru.nl; S. Staton, University of Oxford, Parks Road, OX1 3QD, Oxford, United Kingdom; e-mail: sam.staton@cs.ox.ac.uk.



This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2024 Copyright held by the owner/author(s).

0004-5411/2024/02-ART2

<https://doi.org/10.1145/3632170>

```

x = normal( $\mu=50$ ,  $\sigma=10$ ) # prior
y = normal( $\mu=x$ ,  $\sigma=5$ )    # noisy measurement
y ::= 40                    # make exact observation

```

Variants of exact conditioning are available in different frameworks: In HAKARU [42], certain exact conditioning queries can be addressed using symbolic disintegration, but ($::=$) is not a first-class construct in Hakaru. INFER.NET [33] does allow exact conditioning on variables and employs an approximate inference algorithm to solve the resulting queries (e.g., [23]). The intended formal meaning of exact conditioning is, however, far from obvious when continuous distributions such as Gaussians are involved (in our example, the observation $y == 40$ has probability zero). The goal of this article is to rigorously spell out the exact conditioning paradigm, give semantics to it, and analyze its properties.

We note that, even in this simple example, the use of exact conditioning is intuitive and allows the programmer to cleanly decouple the generative model from the data observation stage. As the example makes clear, exact conditioning lends itself to logical reasoning about programs. For example, after conditioning $s ::= t$, the expressions s and t are known to be equal and can be interchanged.

As we will show, exact conditioning also enjoys good formal properties that allow us to simplify programs compositionally. Among the desired properties are the following: Program lines can be reordered as long as dataflow is respected. That is, the *commutativity equation* remains valid for programs with conditioning

$$\begin{array}{l} \text{let } x = u \text{ in} \\ \text{let } y = v \text{ in } t \end{array} \equiv \begin{array}{l} \text{let } y = v \text{ in} \\ \text{let } x = u \text{ in } t \end{array} \quad (1)$$

where x not free in v and y is not free in u . We have a *substitution law*: if $t ::= u$ appears in a program, then later occurrences of t may be replaced by u .

$$(t ::= u); v[t/x] \equiv (t ::= u); v[u/x] \quad (2)$$

As a special base case, if we condition a normal variable on a constant \underline{c} , then the variable is simply *initialized* to this value:

$$\text{let } x = \text{normal}() \text{ in } (x ::= \underline{c}); t \equiv t[\underline{c}/x]. \quad (3)$$

We substantiate these claims further and give examples of applications of these laws in our extended introduction (Section 1.1).

In order to formally study exact conditioning, we focus on two concrete fragments of probabilistic computation:

- (1) Finite probability, which deals with finite sets and discrete distributions
- (2) Gaussian probability, which deals with multivariate Gaussian (normal) distributions and affine-linear maps

We give probabilistic languages for each fragment and extend them with an exact conditioning construct. The language for finite probability is well known, while the Gaussian language is novel. We give its formal description and operational semantics in Section 2.

METHODS. Our goal is to give denotational semantics to these languages and prove that the desired properties from Equations (1) and (3) hold. We wish to do this in an abstract way that relies as little as possible on the particular details of finite or Gaussian probability but instead treats them uniformly and is open to generalization. *Categorical probability theory* is such an abstract language

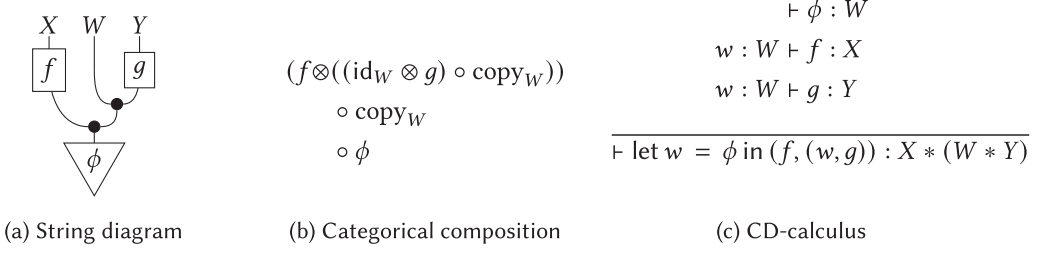


Fig. 1. Different formalisms for composition.

of mathematical models of probability, which allows us to discuss the relevant notions such as determinism, independence, and conditioning in a uniform way. We connect categorical probability theory to probabilistic programming using the following Curry-Howard style correspondence:

- (1) Probabilistic programs are the *internal languages* of categorical probability theories; program terms t can be interpreted as morphisms $\llbracket t \rrbracket$ in these categories.
- (2) For every probabilistic language, its *syntactic category* is a categorical model of probability theory. Objects are types and morphisms are terms of the language.

We prove a particular version of the correspondence in Section 3.3: the CD calculus is the internal language of CD categories, a widespread model of categorical probability theory discussed in Section 3.1. This language, which resembles a first-order OCaml, serves as a meta-language of which all other languages discussed in this article (finite or Gaussian, with or without conditioning) will be particular instances.

The Curry-Howard correspondence gives us *three equivalent formalisms* for describing probabilistic models (see Figure 1):

- (1) Terms in a probabilistic language
- (2) Morphisms in a categorical model
- (3) *String diagrams*, which are a well-known graphical notation to describe compositions of morphisms in an intuitive way [41]

We will frequently convert back and forth between the different formalisms for convenience and conciseness and to emphasize different mathematical or programming intuitions. We expect some familiarity of the reader for translating string-diagrammatic and algebraic categorical notation, though we will give a brief reminder in the introduction of Section 3. The correspondence between string diagrams and program terms is a novel technical contribution of this article. It is formally proved in Section 3.3, and we will aid the reader by spelling out examples in both programming and categorical terms. A reader not primarily interested in category theory may still view our string diagram manipulations as a concise graphical way of encoding program transformations.

To connect our development to more traditional probabilistic notions, we consider the formalism of directed graphical models (Bayesian networks). For example, the three expressions of Figure 1 can all be understood as encoding the generative structure of the following Bayesian network:



This Bayesian network indicates that there are random variables valued in spaces W , X , and Y , with the given conditional independence structure, but this conditional independence means that we can describe the situation by giving the distribution ϕ of the random variable valued in W , together with the distributions of the random variables valued in X and in Y , which both depend on the random choice for W , via f and g .

A systematic comparison of graphical models and string diagrams is given in [10]. We note that Bayesian networks are a strictly weaker formalism than the other three; i.e., not every string diagram or probabilistic program can be obtained from a Bayesian network.

Contributions. We build up this categorical machinery to first understand the semantics of probabilistic languages *without conditioning*. We then use the same framework to extend the language with an exact conditioning operator. On the side of categorical semantics, this amounts to extending a suitable category \mathbb{C} (for conditioning-free computation) with effects $X \rightarrow I$ for conditioning on observations. We call the extended category $\text{Cond}(\mathbb{C})$. Constructing this category (Section 5) and proving its desirable properties is the central contribution of this article.

Importantly, the Cond construction makes no mention of measure theory, densities, or limits, which usually feature in discussions of conditional probability, but is defined purely in terms of the categorical structure of \mathbb{C} . It is closely tied to reasoning in terms of program transformations: the Cond construction can be understood as giving normal forms for straight-line programs with exact observations, modulo contextual equivalence:

$$x : X \vdash \text{let } (y, k) : Y \otimes K = h \text{ in } (k := o); \text{return } y \quad : Y.$$

Our definition of *abstract inference problems* in Section 4 follows that intuition. The desired properties of exact conditioning can be proved purely abstractly for the Cond construction. Our type-theoretic approach to conditioning will also help address counterintuitive behavior such as Borel's paradox (Section 7.1).

We give concrete descriptions of the Cond construction applied to our main examples of discrete and Gaussian probability. This means analyzing contextual equivalence for our example languages in detail: For finite probability, we obtain substochastic kernels modulo *automatic renormalization* (Section 6.2). This fully characterizes contextual equivalence for straight-line inference with discrete probability and refines the semantics using the subdistribution monad. Our discussion reveals interesting connections between the admissibility of automatic normalization and the expressibility of branching in probabilistic programs (Section 6.3). For the Gaussian language we give a concrete analysis by proving that the denotational semantics is fully abstract in Section 6.1.

Outline. We give an overview of the structure of the article:

- To demonstrate the strengths and intricacies of the exact conditioning approach, we will follow up this introduction with an extended example (Section 1.1) elaborating the noisy measurement example and demonstrate the power of program transformations and compositional reasoning for a Gaussian random walk example.
- In Section 2, we formally introduce the Gaussian language and its operational semantics.
- In Section 3, we review string diagrams and categorical probability theory using CD and Markov categories. We then introduce the CD calculus (Section 3.1) and prove it to be internal language of CD categories, which gives us the central correspondence of probabilistic programs and string diagrams. In Section 4, we use the categorical notions to develop an abstract theory of inference problems in Markov categories.

- In Section 5, we present the Cond construction, which is the centerpiece of this article. We prove the well-definedness of its construction (Theorem 5.6) and verify the desired laws for conditioning in full generality (Section 5.3).
- In Section 6, we return to analyze the Cond construction in detail for our two example settings. This is tantamount to studying contextual equivalence for our exact conditioning languages: In Section 6.1, we show that the denotational semantics for the Gaussian language is fully abstract. In Section 6.2, we conduct a similar analysis for finite probability, arriving at an explicit characterization of Cond(FinStoch). Our discussion reveals interesting connections between the admissibility of automatic normalization and the availability of branching in probabilistic programs (Section 6.3).

Note. The starting point for this article is our paper in the Proceedings of LICS 2021 [49], where we introduced the Gaussian language and used the Cond construction to prove a full abstraction result. For this invited journal submission, we expand upon [49] with greater detail and background throughout Sections 1 through 4 and expose the Cond construction in a self-contained manner with an emphasis on program equations and graphical reasoning in Section 5. This submission also incorporates otherwise unpublished material that is part of the first author’s DPhil thesis [48], such as the formal presentation of the CD calculus. The treatment of finite probability in Section 6 and the recognition of the special role of branching in Section 6.3 are entirely novel contributions of this article. A Python implementation of the Gaussian language is available under [47].

1.1 Extended Discussion about Exact Conditioning

We proceed with an extended discussion on the differences between the scoring and exact conditioning paradigms and the strengths and difficulties related to exact conditioning. This discussion uses an informal Python-like language and is not technically essential for the rest of the article. The later sections of the article are fully formal.

Exact Conditioning versus Scoring. In the introduction, we considered a noisy measurement example: our *prior* assumption about the distribution of some quantity X is that it is normally distributed with mean $\mu = 50$ and standard deviation $\sigma = 10$. We only have access to a noisy measurement Y , which itself has standard deviation 5, and observe a value of $Y = 40$. Conditioned on that observation, the *posterior* distribution over X is now $\mathcal{N}(42, \sigma)$ with $\sigma = \sqrt{20} \approx 4.47$.

In probabilistic programming with scoring, the primitive **score**(r) re-weights the current execution trace of the probabilistic program with a *score* or *likelihood* $r \in \mathbb{R}_{>0}$. A derived operation **observe**(x, D) expresses an observation of a value x from some distribution D by scoring with the density $r = \text{pdf}_D(x)$. The scoring implementation of the noisy measurement example therefore looks like this:

```
x = normal(50, 10) # prior
observe(40, normal(x, 5)) # observation
```

The idea of Monte Carlo simulation is to run the program many times, picking different values for x from the normal distribution but preferring runs with a high likelihood. This makes execution traces more likely whose value of x lies closer to 40. Scoring constructs are widely available in popular probabilistic languages such as STAN [5] or WEBPPL [18]. Scoring with likelihoods from $\{0, 1\}$ is sometimes called a *hard constraint*, as opposed to more general *soft constraints*. The prototypical way of performing inference on scoring programs is by likelihood-weighted importance sampling. Hard constraints turn this into mere rejection sampling, because likelihood-zero traces

are discarded entirely. Replacing hard constraints by equivalent soft ones can thus be beneficial for inference efficiency.

Exact conditions are strictly more powerful than scoring, because we can express `observe(x,D)` in terms of conditioning on a freshly generated sample as `let y = sample(D) in y ::= x`. On the other hand, not every exact conditioning program can be expressed in terms of scoring:

In the special case of discrete probability, we can express an exact condition $E_1 ::= E_2$ by the hard constraint `score(if $E_1 == E_2$ then 1 else 0)` without issue. This causes an execution trace to be discarded whenever the condition is not met. This encoding is no longer viable for continuous distributions such as Gaussians: For example, the program

```
x = normal(0,1); x ::= 40
```

should return `x=40` deterministically, because `x` is conditioned to have that value. On the other hand, the following hard constraint:

```
x = normal(0,1); score(if x == 40 then 1 else 0)
```

will *reject every execution trace*, because the probability that `x==40` is true equals zero. It is important to distinguish exact conditioning (`::=`) from the Boolean equality test (`==`). This distinction is crucial to making sense of apparent paradoxes such as Borel's paradox (Section 7.1).

Compositional Reasoning about Conditions. To elaborate on the power of reasoning compositionally about conditioning programs, we consider the example of a simple Gaussian random walk with 100 steps, together with a table `obs` of exact observations (Figure 2). A straightforward implementation would be to first generate the entire random walk and then condition on the observations:

```
# generative model
for i in range(1,100):
    y[i] = y[i-1] + normal(0,1)
# observations
for j in obs:
    y[j] ::= obs[j]
```

The same program is more complicated to express without exact conditioning: using soft conditions, the observations would need to be known at the time of generation and `observe` commands need to be issued in place, breaking the decoupling between the model and the data.

On the other hand, rewriting the original model in such a way may improve the efficiency of inference. We can verify such a transformation using compositional reasoning: As we will show, it is consistent to reorder program lines as long as the dataflow is respected (Equation (1)), so the random walk program is equivalent to the following version with interleaved observations:

```
for i in range(1,100):
    y[i] = y[i-1] + normal(0,1)
    if i in obs: y[i] ::= obs[i]
```

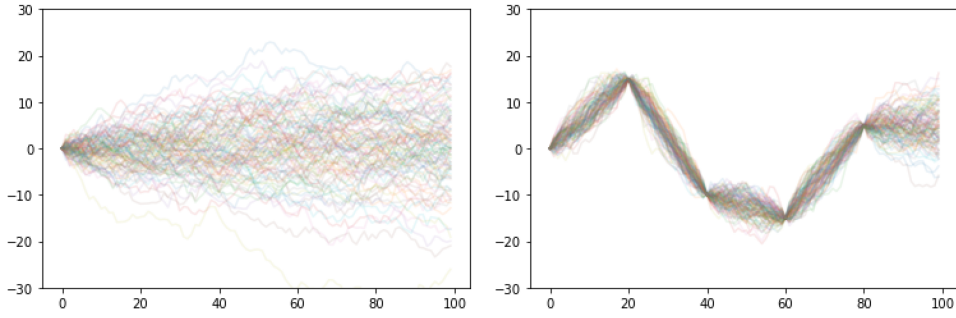


Fig. 2. Gaussian random walk (left) and conditioned posterior (right) with four exact observations at $i = 20, 40, 60, 80$.

In the observation branch, we can now use the initialization principle in Equation (3) to set $y[i]$ to its target value directly as $y[i] = \text{obs}[i]$. The remaining condition becomes $(y[i] - y[i-1]) ::= \text{normal}(0, 1)$, so we obtain

```

for i in range(1, 100):
    if i in obs:
        y[i] = obs[i]
        (y[i] - y[i-1]) ::= normal(0, 1)
    else:
        y[i] = y[i-1] + normal(0, 1)

```

In this version of the program, all exact conditions can now be replaced by **observe** statements:

```

for i in range(1, 100):
    if i in obs:
        y[i] = obs[i]
        observe(y[i] - y[i-1], normal(0, 1))
    else:
        y[i] = y[i-1] + normal(0, 1)

```

We can run this resulting program directly using a Monte Carlo simulation in Stan or WebPPL.

In Section 2, we formalize the operational semantics of our Gaussian language, in which there are two key commands: drawing from a standard normal distribution (`normal()`) and exact conditioning (`::=`). The operational semantics is defined in terms of configurations (t, ψ) , where t is a program and ψ is a state, which here is a Gaussian distribution. Each call to `normal()` introduces a new dimension into the state ψ , and conditioning (`::=`) alters the state ψ using a canonical form of conditioning for Gaussian distributions (Section 2.1).

In our first version of the random walk example, the operational semantics will first build up the prior distribution shown on the left in Figure 2, and then the second part of the program will condition to yield a distribution as shown on the right. But for the other programs above, the conditioning will be interleaved in the building of the model.

In stateful programming languages, the composition of programs is often complicated, and local transformations are difficult to reason about. This is what makes program transformations like the ones we used powerful and nontrivial to verify.

2 A LANGUAGE FOR GAUSSIAN PROBABILITY

In this section, after an overview of the mathematics of Gaussian probability (Section 2.1), we formally introduce a typed language (Section 2.2) for Gaussian probability and exact conditioning and provide an operational semantics for it (Section 2.3). Our operational semantics is straightforward, in that it maintains a symbolic description of the distribution over all latent variables as the program runs, expressed as a covariance matrix. Thus, the aim of this section is to formally describe language that we are studying in this article.

Looking beyond this section, the aim of the further sections of this article is to address the issue that, as usual, the simple operational semantics here is intensional and non-compositional. It is intensional in that if two different programs actually behave in the same way, that might be very unclear from the operational semantics; it is non-compositional in that the role of running subprograms is hidden in the overall run of the operational semantics. The aim of the remainder of the article, then, is to establish an equational and denotational framework for exact conditioning.

The language in this section is focused on Gaussian probability for concreteness, but to understand the equational framework, it will be helpful in future sections to move to a general setting (Sections 3.3 and 5 for the general case without and with exact conditioning, respectively). Once this general framework is established, we are able to offer a denotational explanation of exact conditioning, which specializes to this Gaussian language (Section 6.1).

2.1 Recap of Gaussian Probability

We briefly recall *Gaussian probability*, by which we mean the treatment of multivariate Gaussian distributions and affine-linear maps (e.g., [30]). A *Gaussian distribution* is the law of a random vector $X \in \mathbb{R}^n$ of the form $X = AZ + \mu$, where $A \in \mathbb{R}^{n \times m}$, $\mu \in \mathbb{R}^n$ are not random but the vector Z is a *multivariate standard normal* random vector. That is, its components $Z_1, \dots, Z_m \sim \mathcal{N}(0, 1)$ are independent and standard normally distributed, i.e., with the following probability density function:

$$\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}.$$

(Formally, this is a density that is regarded with respect to the Lebesgue measure, see Appendix A.1, but a high school knowledge of probability density is sufficient for this section.) The distribution of X is fully characterized by its *mean* μ and the positive semidefinite *covariance matrix* $\Sigma = AA^T$. Conversely, for any μ and positive semidefinite matrix Σ there is a unique Gaussian distribution of that mean and covariance, denoted $\mathcal{N}(\mu, \Sigma)$. The vector X takes values precisely in the affine subspace $S = \mu + \text{col}(\Sigma)$, where $\text{col}(\Sigma)$ denotes the column space of Σ . We call S the *support* of the distribution. We note that while it is common to consider Gaussian distributions with non-singular (positive definite) covariance matrix, it is convenient to allow the more general positive semidefinite case here, even including vanishing covariance. Natural programming constructs such as the copying of variables results in a singular covariance, and setting $\Sigma = 0$ lets us treat deterministic computation as a special case of probabilistic one. On the flipside, singular covariance requires us to carefully consider supports in our theory of conditioning.

Gaussian probability defines a small convenient fragment of probability theory, with the following properties:

- Affine transformations of Gaussians remain Gaussian. That is, if an affine map f is written as $f(x) = Ax + b$ and X is a random vector with mean μ and covariance Σ , then $f(X)$ has mean $A\mu + b$ and covariance $A\Sigma A^T$. This operation is called the *pushforward* of distributions and is denoted f_* , i.e.,

$$f_*(\mathcal{N}(\mu, \Sigma)) = \mathcal{N}(A\mu + b, A\Sigma A^T). \quad (5)$$

- Conditional distributions of Gaussians are themselves Gaussian. If we decompose an $(m+n)$ -dimensional Gaussian vector $X \sim \mathcal{N}(\mu, \Sigma)$ into components X_1, X_2 with

$$X = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix}, \mu = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \Sigma = \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}, \text{ where } \Sigma_{21} = \Sigma_{12}^T,$$

there is a well-known explicit formula (e.g., [9, 3.13]) for the conditional distribution $X_1|(X_2 = a)$ of X_1 conditional on $X_2 = a$ for $a \in \text{supp}(X_2)$. Namely $X_1|(X_2 = a) \sim \mathcal{N}(\mu', \Sigma')$, where

$$\mu' = \mu_1 + \Sigma_{12}\Sigma_{22}^-(a - \mu_2) \quad \Sigma' = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^-\Sigma_{21}, \quad (6)$$

and Σ_{22}^- is any *generalized inverse* of Σ_{22} .

We elaborate on Equation (6) a bit: A generalized inverse of an $(m \times n)$ -matrix M is an $(n \times m)$ -matrix M^- such that $MM^-M = M$. Such inverses can be shown to always exist, but they need not be unique. If M is invertible, its unique generalized inverse is M^{-1} . The posterior covariance matrix $\Sigma' = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^-\Sigma_{21}$ is also known as the *Schur complement* and is independent of the choice of generalized inverse. The matrix $\Sigma_{12}\Sigma_{22}^-$ appearing in the calculation of μ' *does* depend on the choice of Σ_{22}^- . However, it takes uniquely defined values on the subspace $\text{col}(\Sigma_{22})$. Therefore, Equation (6) is only well defined if the observation a lies in the support of X_2 . This caveat is mirrored in our categorical treatment of Section 4, where conditionals are only unique on supports. A popular choice of generalized inverse is the Moore-Penrose pseudoinverse, which has connections to least squares optimization. For a detailed discussion of these concepts, we refer to [52, Section 1.6].

The formula for conditional probability becomes particularly simple if we condition on a single real-valued component of a vector: let $X \sim \mathcal{N}(\mu, \Sigma)$ and let $Z = uX$ for some $u \in \mathbb{R}^{n \times 1}$, and then the covariance of (X, Z) , regarded as a random $(n+1)$ -vector, decomposes as

$$\begin{pmatrix} \Sigma & \Sigma u^T \\ u \Sigma^T & \sigma_{22} \end{pmatrix}, \quad \text{where } \sigma_{22} = u \Sigma u^T,$$

and the conditional distribution of $X|(Z = a)$ is $\mathcal{N}(\mu', \Sigma')$ with

$$\mu' = \mu + \frac{a - u\mu}{\sigma_{22}} \Sigma u^T, \quad \Sigma' = \Sigma - \frac{1}{\sigma_{22}} \Sigma u^T u \Sigma \quad (7)$$

whenever $\sigma_{22} > 0$. If $\sigma_{22} = 0$ and $u\mu = a$, the condition is tautologously $0 = 0$ and we have $\mu' = \mu$, $\Sigma' = \Sigma$. Otherwise, $a \notin \text{supp}(Z)$, and the conditioning problem has no well-defined solution.

Example 2.1. Let $X, Y \sim \mathcal{N}(0, 1)$ be independent and $Z = X - Y$. The joint distribution of (X, Y, Z) is $\mathcal{N}(\vec{0}, \Sigma)$ with covariance matrix

$$\Sigma = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \\ 1 & -1 & 2 \end{pmatrix}.$$

By Equation (7), the conditional distribution of (X, Y) given $Z = 0$ has the following covariance matrix:

$$\Sigma' = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \frac{1}{2} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \cdot (1 \quad -1) = \begin{pmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{pmatrix}.$$

The posterior distribution is thus equivalent to the model

$$X \sim \mathcal{N}(0, 0.5), Y = X$$

with one univariate Normal distribution having mean 0 and variance 0.5.

Borel's Paradox. Borel's paradox is an important subtlety that occurs when conditioning on the equality of random variables $X = Y$. The original formulation involves conditioning a uniform point on a sphere to lie on a great circle, but we will use Borel's paradox to refer to any situation where conditioning on equivalent equations leads to different outcomes (e.g., [42]). For example, if instead of the condition $X - Y = 0$ in Example 2.1 we had chosen the seemingly equivalent equations $X/Y = 1$ or even $[X = Y] = 1$ (using Iverson bracket notation), we would have obtained different posteriors:

Example 2.2. If $X, Y \sim \mathcal{N}(0, 1)$, then conditioned on $(X/Y = 1)$, the variable X can be shown to have density $|x|e^{-x^2}$ [40]. Under the Boolean condition $[X = Y] = 1$, the inference problem has no solution because the model $X, Y \sim \mathcal{N}(0, 1), Z = [X = Y]$ is measure-theoretically equal to $X, Y \sim \mathcal{N}(0, 1), Z = 0$ (since independent Gaussian random variables are almost surely different), and conditioning on $0 = 1$ is inconsistent.

We will address Borel's paradox and posit that a careful type-theoretic phrasing (Section 4) helps alleviate its seemingly paradoxical nature (Section 7.1).

2.2 Types and Terms of the Gaussian Language

We now describe a language for Gaussian probability and conditioning. The core language resembles first-order OCaml with a construct `normal()` to sample from a standard Gaussian and conditioning denoted as $(::)$. Types τ are generated from a basic type R denoting *real number* or *random variable*, pair types, and unit type I :

$$\tau ::= R \mid I \mid \tau * \tau.$$

Terms of the language are

$$\begin{aligned} e ::= & x \mid e + e \mid \alpha \cdot e \mid \underline{\beta} \mid (e, e) \mid () \\ & \mid \text{let } x = e \text{ in } e \mid \text{let } (x, y) = e \text{ in } e \\ & \mid \text{normal}() \mid e :: e, \end{aligned}$$

where α, β range over real numbers. Typing judgements are

$$\begin{array}{c} \frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \quad \frac{}{\Gamma \vdash () : I} \quad \frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t : \tau}{\Gamma \vdash (s, t) : \sigma * \tau} \\[10pt] \frac{\Gamma \vdash s : R \quad \Gamma \vdash t : R}{\Gamma \vdash s + t : R} \quad \frac{\Gamma \vdash t : R}{\Gamma \vdash \alpha \cdot t : R} \quad \frac{}{\Gamma \vdash \underline{\beta} : R} \\[10pt] \frac{}{\Gamma \vdash \text{normal}() : R} \quad \frac{\Gamma \vdash s : R \quad \Gamma \vdash t : R}{\Gamma \vdash (s :: t) : I} \\[10pt] \frac{\Gamma \vdash s : \sigma \quad \Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \text{let } x = s \text{ in } t : \tau} \\[10pt] \frac{\Gamma \vdash s : \sigma * \sigma' \quad \Gamma, x : \sigma, y : \sigma' \vdash t : \tau}{\Gamma \vdash \text{let } (x, y) = s \text{ in } t : \tau}. \end{array}$$

In Section 3.3 we will introduce the general CD calculus, and our Gaussian language is an instance of this,¹ with base type R and signature:

$$(+) : R * R \rightarrow R, \quad \alpha \cdot (-) : R \rightarrow R, \quad \underline{\beta} : I \rightarrow R, \quad \text{normal} : I \rightarrow R, \quad (::) : R * R \rightarrow I. \quad (8)$$

¹In the CD calculus, we use projection maps rather than pattern-matching `let`, but those constructs are interdefinable.

This will give us a clear path to denotational semantics: in Section 6.1, we will indeed identify our language as the internal language of an appropriate CD category with an exact conditioning morphism.

We use standard syntactic sugar for sequencing $s; t$. We identify the type $R^n = R * (R * \dots)$ with vectors $x = (x_1, \dots, x_n)$ and write matrix-vector multiplication $A \cdot x$ in an informal manner. For $\sigma \in \mathbb{R}$ and $e : R$, we define $\text{normal}(x, \sigma^2) \equiv x + \sigma \cdot \text{normal}()$. More generally, for a covariance matrix Σ and $x : R^n$, we write $\text{normal}(x, \Sigma) = x + A \cdot (\text{normal}(), \dots, \text{normal}())$, where A is any matrix such that $\Sigma = AA^T$. By the simple nature of the typing rules, we can identify any context and type with R^n for suitable n .

For example, referring to Example 2.1, the tuple (X, Y, Z) can be written in our language as

$$\text{let } (x, y) = (\text{normal}(), \text{normal}()) \text{ in } (x, y, x - y).$$

The full example with conditioning can be written

$$\text{let } (x, y, z) = (\text{let } (x, y) = (\text{normal}(), \text{normal}()) \text{ in } (x, y, x - y)) \text{ in } z ::= 0; (x, y).$$

This program is contextually equivalent (Definition 2.5) to

$$\text{let } x = \sqrt{0.5} * \text{normal}() \text{ in let } y = x \text{ in } (x, y).$$

2.3 Operational Semantics

Informally, our operational semantics works as follows: calling $\text{normal}()$ allocates a latent random variable, and a prior distribution over all latent variables is maintained; calling $(::=)$ updates this prior by symbolic inference according to Equation (6).

Formally, we define a reduction relation over configurations. A *configuration* is either a dedicated failure symbol \perp or a pair

$$(e, \psi),$$

where ψ is a Gaussian distribution on \mathbb{R}^r (i.e., a mean vector and covariance matrix) and $z_1 : R, \dots, z_r : R \vdash e$. Thus, a running term e may have free variables; these stand for dimensions in a given multivariate Gaussian distribution ψ , reminiscent of a closure in a higher-order language.

To define a reduction relation, we first introduce values, redexes, and reduction contexts. *Values* v, w and *redexes* ρ are defined as

$$\begin{aligned} v, w &::= x \mid (v, w) \mid v + w \mid \alpha \cdot v \mid \underline{\beta} \mid () \\ \rho &::= \text{normal}() \mid v ::= w \mid \text{let } x = v \text{ in } e \mid \text{let } (x, y) = v \text{ in } e. \end{aligned}$$

A *reduction context* C with hole $[-]$ is of the form

$$\begin{aligned} C &::= [-] \mid (C, e) \mid (v, C) \mid C + e \mid v + C \mid \alpha \cdot C \mid C ::= e \mid v ::= C \\ &\quad \mid \text{let } x = C \text{ in } e \mid \text{let } (x, y) = C \text{ in } e. \end{aligned}$$

Perhaps the only thing to note is that, in keeping with the call-by-value tradition of most probabilistic programming languages, we do reduce before a let assignment; i.e. $\text{let } x = C \text{ in } e$ is a reduction context. It is easy to show by induction that every term is either a value or decomposes uniquely as $C[\rho]$. The latent variables $(z_1 \dots z_r)$ are taken from a distinct supply of variable names $\{z_i : i \in \mathbb{N}\}$. We first define reduction on redexes (1–3) and then reduction contexts (4):

- (1) Calling $\text{normal}()$ allocates a fresh latent variable and adds an independent dimension to the prior:

$$(\text{normal}(), \psi) \triangleright (z_{r+1}, \psi \otimes \mathcal{N}(0, 1)),$$

where ψ is a Gaussian distribution on \mathbb{R}^r with mean μ and covariance Σ ; here $(\psi \otimes \mathcal{N}(0, 1))$ is notation for the Gaussian distribution on \mathbb{R}^{r+1} with mean $(\mu, 0)$ and covariance matrix $\begin{pmatrix} \Sigma & 0 \\ 0 & 1 \end{pmatrix}$.

- (2) To define conditioning, note that every value $z_1 : \mathbb{R}, \dots, z_r : \mathbb{R} \vdash v : \mathbb{R}$ defines an affine function $\mathbb{R}^r \rightarrow \mathbb{R}$. In order to reduce $(v ::= w, \psi)$, we consider an independent random variable $X \sim \psi$ and define the auxiliary real random variable $Z = v(X) - w(X)$. If 0 lies in the support of Z , we denote by $\psi|_{v=w}$ the outcome of conditioning X on $Z = 0$ and reduce:

$$(v ::= w, \psi) \triangleright ((), \psi|_{v=w}).$$

Otherwise $(v ::= w, \psi) \triangleright \perp$, indicating that the inference problem has no solution. To be completely precise, since v and w are affine, the function $(v - w)$ is affine too, so we can find $u \in \mathbb{R}^{1 \times r}$ and $b \in \mathbb{R}$ such that $(v(x) - w(x)) = ux + b$ and then we condition on $uX = -b$ using Equation (7).

- (3) Let bindings are standard:

$$(\text{let } x = v \text{ in } e, \psi) \triangleright (e[v/x], \psi)$$

$$(\text{let } (x, y) = (v, w) \text{ in } e, \psi) \triangleright (e[v/x, w/y], \psi).$$

- (4) Lastly, under reduction contexts, if $(\rho, \psi) \triangleright (e, \psi')$, we define $(C[\rho], \psi) \triangleright (C[e], \psi')$. If $(\rho, \psi) \triangleright \perp$, then $(C[\rho], \psi) \triangleright \perp$.

PROPOSITION 2.3. *For every closed typed program $\vdash e : \tau$ either there is a unique value configuration (v, ψ) such that $(e, ()) \triangleright^* (v, \psi)$ with v a value or $(e, ()) \triangleright^* \perp$. (Here $()$ is the unique 0-dimensional Gaussian distribution, and \triangleright^* is the reflexive transitive closure of \triangleright .)*

PROOF NOTES. First, the \triangleright relation is deterministic and satisfies progress and type preservation lemmas. These are all shown by induction on typing derivations. Next, all reduction sequences terminate, because the number of steps is bounded by the number of symbols from $\{\text{normal}, ::=, \text{let } = \text{in}\}$ in an expression. \square

We consider the observable result of this execution either failure or the pushforward distribution $v_*\psi$ on \mathbb{R}^n , as this distribution could be sampled from empirically.

Example 2.4. The program

$$\text{let } (x, y) = (\text{normal}(), \text{normal}()) \text{ in } x ::= y; x + y$$

reduces to $(z_1 + z_2, \psi)$, where

$$\psi = \mathcal{N}\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{pmatrix}\right).$$

The observable outcome of the run is the pushforward distribution $(1 \ 1)_*\psi = \mathcal{N}(0, 2)$ on \mathbb{R} .

One goal of this article is to study properties of this language compositionally and abstractly, without relying on any specific properties of Gaussians. From the operational semantics, we can define an extensional and compositional contextual equivalence.

Definition 2.5. We say $\Gamma \vdash e_1, e_2 : \tau$ are *contextually equivalent*, written $e_1 \approx e_2$, if for all closed contexts $K[-]$ and $i, j \in \{1, 2\}$:

- (1) when $(K[e_i], !) \triangleright^* (v_i, \psi_i)$, then $(K[e_j], !) \triangleright^* (v_j, \psi_j)$ and $(v_i)_*\psi_i = (v_j)_*\psi_j$;
- (2) when $(K[e_i], !) \triangleright^* \perp$, then $(K[e_j], !) \triangleright^* \perp$.

Here $(v_i)_*\psi_i$ denotes the pushforward distribution as defined in Equation (5).

We later study contextual equivalence by developing a denotational semantics for the Gaussian language (Section 6.1) and proving it fully abstract (Theorem 6.2). We also note that nothing conceptually limits the language in this section to only Gaussians. We are running with this example for concreteness, but any family of distributions that can be sampled and conditioned can be used. So we will take care to establish properties of the semantics in a general setting.

3 CATEGORICAL SEMANTICS FOR PROBABILISTIC PROGRAMMING

This section introduces denotational semantics for the probabilistic language in Section 2, at first without conditioning. We will construct this semantics using a general and reusable strategy, namely:

- (1) understanding the structure of mathematical models of probability theory and
- (2) introducing a metalanguage (the CD calculus) that acts as the internal language of these categorical models.

The Gaussian language will be a particular instance of the metalanguage, and its semantics will take place in the Markov category Gauss (defined in Definition 3.8). The language for finite probability will have semantics in the Markov category FinStoch (Definition 3.4).

This is a general section on the relationship between probabilistic languages and categorical models. We will recall symmetric monoidal categories and string diagrams, and define CD and Markov categories, which are the relevant categorical models of probability theory. We then introduce the CD calculus as a metalanguage for first-order probabilistic programs and prove the desired correspondence:

- (1) Every program term can be interpreted as a morphism in a CD category.
- (2) Every string diagram can be encoded as a program term.
- (3) Every valid manipulation of string diagrams translates to a provable equality of programs, and vice versa.

This is formally stated and proven in terms of internal language and syntactic category (Proposition 3.14 and Definition 3.18). This unlocks the equivalent formalisms discussed in the introduction (Figure 1). These formalisms form the basis of our study of conditioning in the later Sections 4 and 5.

Monoidal Categories and String Diagrams. Recall that a category comprises objects and morphisms between the objects. In this context, the objects are to be thought of as generalized spaces and the morphisms as stochastic functions. That is, a morphism $X \rightarrow Y$ is thought of roughly as something that takes an argument from X and makes some random choices before returning an element of Y . (The reader familiar with probability theory can regard them as probability kernels or parameterized measures.) In particular, our categories will be monoidal, which means we have the following constructions (see, e.g., [32] for full definitions):

- Monoidal structure: There is a distinguished object I (thought of as the one-point space), and for any objects X and Y there is an object $X \otimes Y$ (thought of as the product space). The morphisms $I \rightarrow X$ are thought of as probability distributions on X , and so the morphisms $Y \rightarrow X$ can be thought of as distributions on X with parameters from Y .
- Categorical composition: For any morphisms $f: X \rightarrow Y$ and $g: Y \rightarrow Z$, there is a composite morphism $g \circ f: X \rightarrow Z$. This represents running stochastic computation *in sequence*. Mathematically, in several of the examples (Section 3.2), composition is calculated by a form of integration or summation (integrating over Y), and so we can regard this composition as an abstract account of integration. We will often abbreviate the composition $g \circ f$ as gf . In the category of sets and functions, morphisms $x: 1 \rightarrow X$ can be identified with elements

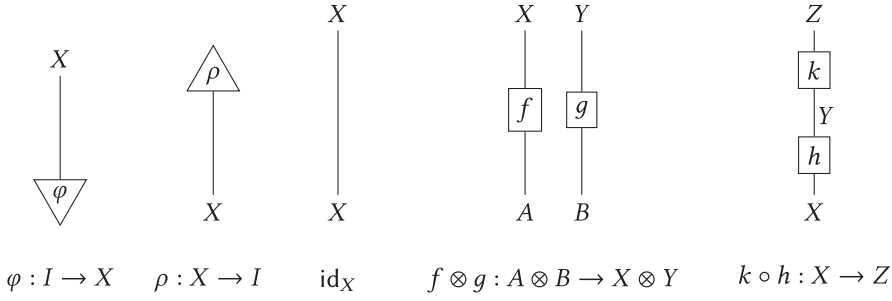


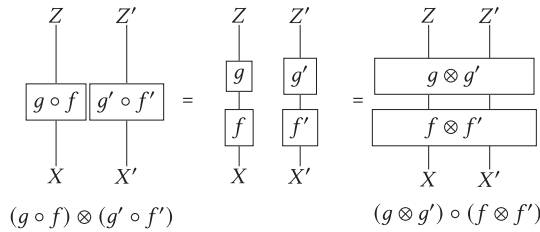
Fig. 3. Overview of string diagram notation.

$x \in X$. If $f : X \rightarrow Y$ is a function, the composite $f \circ x = fx$ agrees with function application $f(x)$. This notation will be convenient in Section 4 when applied to deterministic states.

- Monoidal composition: For any morphisms $f : A \rightarrow B$ and $g : X \rightarrow Y$, there is a composite morphism $(f \otimes g) : A \otimes X \rightarrow B \otimes Y$. This can be understood as running stochastic computations *in parallel*: informally, given a pair (a, x) , we randomly produce b from a and y from x , returning the pair (b, y) . Mathematically, in the examples (Section 3.2), this monoidal composition amounts to product probabilities. In the measure theoretic example (Definition 3.10), the interchange law of the tensor \otimes encodes Fubini’s theorem (see Appendix A.1).

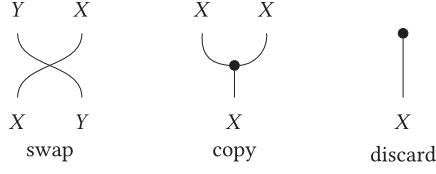
The notation for composing morphisms (with \circ and \otimes) can quickly become cryptic, and it is important to find good notations. *String diagrams* are one such widely used and intuitive notation (e.g., [27, 41]). In the string diagram, the objects of the category become wires and morphisms are boxes; sequential composition (\circ) is simply joining wires together, and monoidal composition (\otimes) is juxtaposition. States ($I \rightarrow X$) and effects ($X \rightarrow I$) have special notations, because the unit object I need not be drawn in string diagrams (see Figure 3 for an overview).

We read such diagrams from bottom to top. Even without categorical machinery, string diagrams carry an intuitive meaning as dataflow diagrams. We can manipulate string diagrams using intuitive rules (sliding around of boxes, formally “planar isotopy”), and the axioms of monoidal categories ensure that all such manipulations result in the same overall composite. Even more, if there are different ways of parsing a string diagram into a sequence of composites, then all these ways have provably equal meanings. This result is known as *coherence* for monoidal categories. We demonstrate this for the so-called *interchange law* $(g \circ f) \otimes (g' \circ f') = (g \otimes g') \circ (f \otimes f')$, which is derivable from the axioms of monoidal categories. It corresponds to the unambiguous reading of the string diagram



Symmetry, Copying, and Discarding. Monoidal categories are an important general concept, but to discuss probabilities in this axiomatic way, it is appropriate to require further structure, resulting in copy-delete categories and Markov categories as we discuss in Section 3.1. The crucial operations

are swapping, copying, and discarding of data, which we depict as follows:



The same building blocks are necessary to interpret terms of a programming language in which variables can be used without linearity restriction. For example, the term-in-context $x, y, z \vdash (y, x, x)$ requires us to swap and copy x , as well as discard z . The same could be achieved by first copying x and then swapping both copies with y . These two diagrams are provably equal from the axioms of CD categories:

$$\llbracket (y, x, x) \rrbracket = \text{Diagram 1} = \text{Diagram 2}$$

The equation shows two equivalent string diagrams for the term (y, x, x) .
 Diagram 1: A strand for Y and a strand for X (labeled Z at the bottom) are present. The X strand is copied into two strands, both labeled X at the top. One of these X strands is then swapped with the Y strand. The final output strands at the top are labeled Y, X, X .
 Diagram 2: Similar to Diagram 1, but the X strand is first swapped with the Y strand, and then the resulting X strand is copied into two strands, both labeled X at the top. The final output strands at the top are also labeled Y, X, X .

In defining the CD calculus (Section 3.3), we extend the syntax with let-binding, tuples, projections, and function calls. The calculus has equivalent expressive power to string diagrams, as showcased in the introduction (Figure 1). In Section 3.4, we give the systematic method to associate to every term t a string diagram $\llbracket t \rrbracket$. In particular, manipulations of the string diagrams correspond to valid program transformations. For example, using the definition of the semantics of let-bindings in Figure 5, the validity of the commutativity Equation (1) corresponds to the following string diagram manipulation (where $x \notin \text{fv}(t)$, $y \notin \text{fv}(s)$):

$$\llbracket \text{let } x = s \text{ in } \text{let } y = t \text{ in } (x, y) \rrbracket = \text{Diagram 1} = \text{Diagram 2} = \text{Diagram 3} = \llbracket \text{let } y = t \text{ in } \text{let } x = s \text{ in } (x, y) \rrbracket$$

The equation shows a sequence of four string diagrams representing the commutativity of let-bindings.
 Diagram 1: A box labeled $\llbracket s \rrbracket$ has two output strands. One strand is copied into two strands, both labeled X at the top. The other strand from the box is labeled Y at the top. The final output strands at the top are labeled Y, X, X .
 Diagram 2: Two separate boxes, $\llbracket s \rrbracket$ and $\llbracket t \rrbracket$, are shown. The $\llbracket s \rrbracket$ box has two output strands, both labeled X at the top. The $\llbracket t \rrbracket$ box has one output strand labeled Y at the top. The final output strands at the top are labeled Y, X, X .
 Diagram 3: A box labeled $\llbracket t \rrbracket$ has one output strand labeled Y at the top. This strand is copied into two strands, both labeled Y at the top. One of these Y strands is then swapped with a strand from the $\llbracket s \rrbracket$ box. The final output strands at the top are labeled Y, X, X .
 Diagram 4: A box labeled $\llbracket s \rrbracket$ has two output strands, both labeled X at the top. One of these X strands is then swapped with a strand from the $\llbracket t \rrbracket$ box. The final output strands at the top are labeled Y, X, X .

In these opening remarks so far, we have started from categorical notions and moved to notations. It is also helpful to follow the opposite route: we can regard notation as primal—be it string diagrams, graphical models, or probabilistic programs. Now to decide whether two composites are equal (two diagrams, two programs), we regard them as morphisms in a category (called the *syntactic category*) and ask whether they are equal there (Section 3.4). In this way, category theory is merely a formalism for compositional theories of equality, which are useful from a foundational perspective as well as for understanding valid program manipulations. We axiomatize this equality from the programming perspective in Section 3.5.

We will now formally introduce CD categories and Markov categories and define the relevant mathematical models for finite and Gaussian probability, before returning to the CD calculus in Section 3.3.

3.1 Copy-delete Categories and Markov Categories

We will recall two closely related notions, namely:

- Markov categories** to model purely stochastic computation [11] and
- CD categories**, which model potentially *unnormalized* stochastic computation [6].

Markov categories have been used to formalize various theorems of probability and statistics, such as sufficient statistics (Fisher-Neyman, Basu, Bahadur) [11], stochastic dominance (Blackwell-Sherman-Stein) [12], and zero-one laws [13]. Both types of category admit a convenient graphical language in terms of string diagrams. We will also define the CD calculus, which is the internal language of CD categories and reminiscent of first-order OCaml (Section 3.3). This makes CD categories a natural foundation for probabilistic programming. Denotational semantics will be given to our Gaussian language by recognizing it as the internal language of an appropriate CD category.

Definition 3.1 (CD Category [6]). A *copy-delete category* (CD category) is a symmetric monoidal category (\mathbb{C}, \otimes, I) , where every object X is equipped with the structure of a commutative comonoid:

$$\text{copy}_X : X \rightarrow X \otimes X \quad \text{del}_X : X \rightarrow I,$$

graphically depicted as

$$\text{copy}_X = \begin{array}{c} \text{---} \backslash \text{---} \\ \text{---} \bullet \text{---} \\ \text{---} / \text{---} \\ | \end{array} \quad \text{del}_X = \begin{array}{c} \bullet \\ | \end{array}$$

satisfying the axioms

$$\begin{array}{c} \text{---} \backslash \text{---} \\ \text{---} \bullet \text{---} \\ \text{---} / \text{---} \\ | \end{array} = \begin{array}{c} \text{---} \backslash \text{---} \\ \text{---} \bullet \text{---} \\ \text{---} / \text{---} \\ | \end{array} \quad \begin{array}{c} \bullet \\ | \end{array} = \begin{array}{c} | \end{array} = \begin{array}{c} \bullet \\ | \end{array} \quad \begin{array}{c} \text{---} \backslash \text{---} \\ \text{---} \bullet \text{---} \\ \text{---} / \text{---} \\ | \end{array} = \begin{array}{c} \text{---} \backslash \text{---} \\ \text{---} \bullet \text{---} \\ \text{---} / \text{---} \\ | \end{array}$$

We require that the comonoid structure be compatible with the monoidal structure as follows:

$$\begin{array}{c} \bullet \\ | \end{array} = \begin{array}{c} \bullet \quad \bullet \\ | \quad | \end{array} \quad \begin{array}{c} X \otimes Y \quad X \otimes Y \\ \text{---} \backslash \text{---} \\ \text{---} \bullet \text{---} \\ \text{---} / \text{---} \\ X \otimes Y \end{array} = \begin{array}{c} X \quad Y \quad X \quad Y \\ \text{---} \backslash \text{---} \quad \text{---} / \text{---} \\ \text{---} \bullet \text{---} \quad \text{---} \bullet \text{---} \\ \text{---} / \text{---} \quad \text{---} \backslash \text{---} \\ X \quad Y \end{array}$$

It is important that copy and del are *not* assumed to be natural; explicitly the equations

$$\begin{array}{c} \boxed{f} \quad \boxed{f} \\ \text{---} \backslash \text{---} \\ \text{---} \bullet \text{---} \\ | \end{array} \stackrel{?}{=} \begin{array}{c} \text{---} \backslash \text{---} \\ \text{---} \bullet \text{---} \\ \text{---} / \text{---} \\ \boxed{f} \end{array} \quad \begin{array}{c} \bullet \\ | \end{array} \stackrel{?}{=} \begin{array}{c} \bullet \\ | \end{array} \quad (9)$$

need *not* hold in general. We give special names to situations where they do hold.

Definition 3.2 (Copyable, Discardable, Deterministic). A morphism $f : X \rightarrow Y$ is called *copyable* if the first equation of Equation (9) holds: $\text{copy}_Y \circ f = (f \otimes f) \circ \text{copy}_X$. A morphism $f : X \rightarrow Y$ is called *discardable* if the second equation of Equation (9) holds: $\text{del}_Y \circ f = \text{del}_X$. A morphism is called *deterministic* if it is copyable and discardable.

Definition 3.3 (Markov Category [11]). A Markov category is a CD category \mathbb{C} in which the following equivalent properties hold:

- (1) \mathbb{C} is semicartesian; i.e., the unit I is terminal.
- (2) Every morphism is discardable.
- (3) del is natural.

The definitions of CD and Markov categories encode a significant amount of properties of stochastic computation, as discussed informally at the beginning of Section 3. The discardability condition in Markov categories informally means that probabilities sum to 1, as is the case in the examples (Section 3.2).

Notation. The presence of explicit copying and discarding maps lets us apply a product-like syntax for CD categories: if $f : A \rightarrow X$, $g : A \rightarrow Y$, we write a tupling

$$\langle f, g \rangle \stackrel{\text{def}}{=} (f \otimes g) \circ \text{copy}_A$$

and define projection maps

$$\pi_X : X \otimes Y \rightarrow X \quad \pi_Y : X \otimes Y \rightarrow Y$$

via discarding. Recall that a *state* in a symmetric monoidal category is a morphism $\psi : I \rightarrow X$. An *effect* is a morphism $\rho : X \rightarrow I$. Note that by terminality of the unit I in a Markov category, all effects $X \rightarrow I$ must be trivial; in CD categories, effects will be of interest.

In Markov categories, we will furthermore employ the following probabilistic terminology: we call states $\psi : I \rightarrow X$ *distributions*, and if $f : A \rightarrow X \otimes Y$, we define its *marginal* (of X) $f_X : A \rightarrow X$ to be $\pi_X f$. Of course, we generally have $f \neq \langle f_X, f_Y \rangle$ unless \mathbb{C} is Cartesian. For every CD category \mathbb{C} , the wide subcategory \mathbb{C}_{det} , which consists of only the deterministic morphisms, is Cartesian [11, Remark 10.13].

3.2 Examples of CD Categories

In this subsection, we will briefly introduce the relevant examples of Markov and CD categories we will be working with, in particular finite and Gaussian probability. All examples in this subsection are standard material and covered in [11]. More mathematical detail and the theory of conditioning are given in Section 4. Readers primarily interested in syntax can proceed with Section 3.3.

Definition 3.4 ([11, 2.5]). The Markov category FinStoch has as objects finite sets X , and morphisms $X \rightarrow Y$ are probability channels, that is, stochastic matrices $p \in [0, 1]^{Y \times X}$, which are sometimes written in the notation $p(y|x)$. Composition in FinStoch takes the form of the *Kolmogorov-Chapman equation*:

$$(pq)(z|x) = \sum_y p(z|y)q(y|x). \quad (10)$$

We modify FinStoch as follows to allow for unnormalized (“sub-stochastic”) computation:

Definition 3.5. The CD category FinSubStoch has as objects finite sets X , and morphisms $X \rightarrow Y$ are *subprobability channels* $p(y|x)$, that is, $p(y|x) \in [0, 1]$, and for all $x \in X$,

$$\sum_y p(y|x) \leq 1.$$

Composition is again given by Equation (10).

A convenient way to understand composition in these categories is using the theory of monads.

Definition 3.6. The distribution monad D and subdistribution monad $D_{\leq 1}$ on the category of sets are defined as the sets of all (sub)probability distributions on a given set X :

$$D(X) = \left\{ p : X \rightarrow [0, 1] \text{ finitely supported} : \sum_x p(x) = 1 \right\},$$

$$D_{\leq 1}(X) = \left\{ p : X \rightarrow [0, 1] \text{ finitely supported} : \sum_x p(x) \leq 1 \right\}.$$

The unit of both monads is given by taking Dirac distribution $x \mapsto \delta_x$ with

$$\delta_x(y) = \begin{cases} 1, & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

and the monad multiplication μ is defined as

$$\mu(\rho)(x) = \sum_p \rho(p) \cdot p(x).$$

Kleisli composition for these monads recovers the Kolmogorov-Chapman equation (Equation (10)). That is, morphisms in $\text{FinStoch}(X, Y)$ are Kleisli arrow $X \rightarrow D(Y)$ for D , and morphisms in $\text{FinSubStoch}(X, Y)$ are Kleisli arrows for $X \rightarrow D_{\leq 1}(Y)$. The following result shows that Kleisli categories are a general source of CD categories.

PROPOSITION 3.7 ([11, 3.2]). *Let \mathbb{C} be a category with finite products and $T : \mathbb{C} \rightarrow \mathbb{C}$ be a strong, commutative monad. Then the Kleisli category $\text{Kl}(T)$ is a CD category, which is furthermore Markov if and only if T is affine, i.e., $T1 \cong 1$.*

Again, a morphism in $\text{FinSubStoch}(X, Y)$ is a Kleisli arrow $X \rightarrow D_{\leq 1}(Y)$ for the subprobability monad on Set .

We now define the Markov category that captures the Gaussian probability of Section 2.1.

Definition 3.8 ([11, Section 6]). The Markov category **Gauss** has objects $n \in \mathbb{N}$, which represent the affine space \mathbb{R}^n , and $m \otimes n = m + n$. Morphisms $m \rightarrow n$ are tuples (A, b, Σ) , where $A \in \mathbb{R}^{n \times m}$, $b \in \mathbb{R}^n$, and $\Sigma \in \mathbb{R}^{n \times n}$ are a positive semidefinite matrix. The tuple represents a stochastic map $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ that is affine-linear, perturbed with multivariate Gaussian noise of covariance Σ , informally written

$$f(x) = Ax + b + \mathcal{N}(\Sigma) \text{ or } Ax + \mathcal{N}(b, \Sigma).$$

Such morphisms compose sequentially and in parallel in the expected way, with noise accumulating independently:

$$(A, b, \Sigma) \circ (C, d, \Xi) = (AC, Ad + b, A\Xi A^T + \Sigma)$$

$$(A, b, \Sigma) \otimes (C, d, \Xi) = \left(\begin{pmatrix} A & 0 \\ 0 & C \end{pmatrix}, \begin{pmatrix} b \\ d \end{pmatrix}, \begin{pmatrix} \Sigma & 0 \\ 0 & \Xi \end{pmatrix} \right).$$

Copy and discard structure are given using the affine maps:

$$\text{copy}_n : \mathbb{R}^n \rightarrow \mathbb{R}^{n+n}, x \mapsto (x, x) \quad \text{del}_n : \mathbb{R}^n \rightarrow \mathbb{R}^0, x \mapsto ().$$

Note that we explicitly allow zero covariance in the definition of **Gauss**. This way, the category is able to encode deterministic computation as a special case.

PROPOSITION 3.9. *A morphism (A, b, Σ) in **Gauss** is deterministic (Definition 3.2) iff $\Sigma = 0$; i.e., there is no randomness involved.*

PROOF. Write $f = (A, b, \Sigma)$, and then the covariance matrices of $f \circ \text{copy}$ and $\text{copy} \circ f$ are

$$\begin{pmatrix} \Sigma & 0 \\ 0 & \Sigma \end{pmatrix} \text{ and } \begin{pmatrix} \Sigma & \Sigma \\ \Sigma & \Sigma \end{pmatrix},$$

respectively. Thus, f is copyable iff $\Sigma = 0$. \square

It follows that the deterministic subcategory $\text{Gauss}_{\text{det}}$ is the category Aff consisting of the spaces \mathbb{R}^n and affine maps between them.

Note that this definition of Gauss involves no measure theory at all; a Gaussian is fully described by its mean and covariance matrix. Measure theory can, however, be used to build a Markov category that is rather comprehensive in that it includes the previous two examples. We briefly state the definition here and refer to the appendix (Appendix A.1) for details. (In this article, we will not use this measure-theoretic Markov category in a crucial way; we will only use it in illustrative examples.)

Definition 3.10 ([11, Section 4]). The Markov category BorelStoch has as objects standard Borel spaces X , and morphisms $X \rightarrow Y$ are probability kernels $\Sigma_X \times Y \rightarrow [0, 1]$.

BorelStoch arises as the Kleisli category of the Giry monad \mathcal{G} on standard Borel spaces [16]. Both FinStoch and Gauss are subcategories of BorelStoch ; i.e., there are faithful inclusion functors that preserve all CD structure.

Lastly, we give an example to show that the formalism of CD categories can encompass not only probabilistic situations but also nondeterminism.

Definition 3.11 (e.g., [11, Example 2.6 and Section 8.1]). We denote by Rel the CD category of sets X, Y and relations $R \subseteq X \times Y$ between them. We denote by Rel^+ the Markov subcategory of sets and left-total relations between them, i.e., $\forall x \in X \exists y \in Y, (x, y) \in R$.

The two categories are obtained as the Kleisli categories of the powerset monad $\mathcal{P} : \text{Set} \rightarrow \text{Set}$ and the nonempty powerset monad $\mathcal{P}^+ : \text{Set} \rightarrow \text{Set}$, respectively. The category Rel^+ is referred to as SetMulti in [11].

3.3 Internal Languages and Denotational Semantics

We present the CD calculus, which is the internal language CD categories. It is reminiscent of the first-order fragment of fine-grained call-by-value or the computational λ -calculus (e.g., [31, 34, 35]), but the commutativity of the tensor allows for some convenient simplifications and a concise equational presentation. To this extent it is a novel calculus.

Definition 3.12. A CD signature $\mathfrak{S} = (\tau, \omega)$ consists of sets τ of base types and function symbols ω . A *type* is recursively defined by closing the base types under tuple formation:

$$A ::= \tau \mid \text{unit} \mid A * A.$$

Each function symbol $f \in \omega$ is equipped with a unary *arity* of types, written $f : A \rightarrow B$. The terms of the CD-calculus are given by

$$t ::= x \mid () \mid (t, t) \mid \pi_i t \mid f t \mid \text{let } x = t \text{ in } t \quad (i = 1, 2)$$

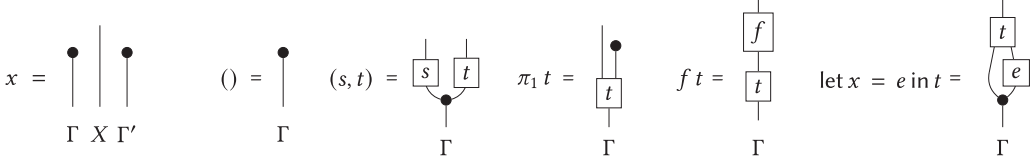
subject to the typing rules $x_1 : A_1, \dots, x_n : A_n \vdash t : B$ given in Figure 4.

We employ some standard syntactic sugar, for example, sequencing:

$$s; t \stackrel{\text{def}}{=} \text{let } x = s \text{ in } t \quad (x \notin \text{fv}(t)).$$

$$\begin{array}{c}
\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \quad \frac{}{\Gamma \vdash () : \text{unit}} \quad \frac{\Gamma \vdash s : A \quad \Gamma \vdash t : B}{\Gamma \vdash (s, t) : A * B} \quad \frac{\Gamma \vdash t : A}{\Gamma \vdash ft : B} \quad (f : A \rightarrow B) \\
\\
\frac{\Gamma \vdash t : A_1 * A_2}{\Gamma \vdash \pi_1 t : A_1} \quad \frac{\Gamma \vdash t : A_1 * A_2}{\Gamma \vdash \pi_2 t : A_2} \quad \frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash e : A}{\Gamma \vdash \text{let } x = e \text{ in } t : B}
\end{array}$$

Fig. 4. Typing rules for the CD calculus.

Fig. 5. Translating terms into string diagrams (brackets $\llbracket - \rrbracket$ omitted for readability).

We also define a pattern-matching let as syntactic sugar:

$$(\text{let } (x, y) = s \text{ in } t) \stackrel{\text{def}}{=} (\text{let } p = s \text{ in let } x = \pi_1 p \text{ in let } y = \pi_2 p \text{ in } t).$$

Conversely, we can provably recover the projection constructs from this sugar (in a sense made precise by the equational theory in Section 3.5):

$$(\pi_1 s) = (\text{let } (x, y) = s \text{ in } x) \quad (\pi_2 s) = (\text{let } (x, y) = s \text{ in } y).$$

We prefer the projections over pattern matching when presenting the equational theory, because this means one less binding construct.

3.4 Semantics

We now explain how the CD calculus can be interpreted in CD categories. Types will be interpreted as objects, and terms interpreted as morphisms. Formally, a *model* of signature (τ, ω) is a CD category \mathbb{C} together with an assignment of objects $\llbracket A \rrbracket \in \mathbb{C}$ for each basic type and morphisms $\llbracket f \rrbracket : \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket$ for each function symbol $f : A \rightarrow B$. Here we extend $\llbracket - \rrbracket$ to arbitrary types and contexts by

$$\llbracket \text{unit} \rrbracket = I \quad \llbracket A_1 * A_2 \rrbracket = \llbracket A_1 \rrbracket \otimes \llbracket A_2 \rrbracket \quad \llbracket A_1, \dots, A_n \rrbracket = \llbracket A_1 \rrbracket \otimes (\dots \otimes \llbracket A_n \rrbracket).$$

For any model, the interpretation of a term $\Gamma \vdash t : A$ is defined recursively as:

- $\llbracket x \rrbracket$ is the discarding map $\llbracket \Gamma, A, \Gamma' \rrbracket \cong \llbracket \Gamma \rrbracket \otimes \llbracket A \rrbracket \otimes \llbracket \Gamma' \rrbracket \rightarrow I \otimes \llbracket A \rrbracket \otimes I \cong \llbracket A \rrbracket$.
- $\llbracket () \rrbracket$ is the discarding map $\text{del}_{\llbracket \Gamma \rrbracket} : \llbracket \Gamma \rrbracket \rightarrow I$.
- $\llbracket (s, t) \rrbracket$ is the map $\llbracket \Gamma \rrbracket \xrightarrow{\text{copy}_{\llbracket \Gamma \rrbracket}} \llbracket \Gamma \rrbracket \otimes \llbracket \Gamma \rrbracket \xrightarrow{\llbracket s \rrbracket \otimes \llbracket t \rrbracket} \llbracket A \rrbracket \otimes \llbracket B \rrbracket = \llbracket A * B \rrbracket$.
- $\llbracket \pi_i t \rrbracket$ is marginalization $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket t \rrbracket} \llbracket A_1 \rrbracket \otimes \llbracket A_2 \rrbracket \rightarrow \llbracket A_i \rrbracket$.
- $\llbracket ft \rrbracket$ is the composite $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket t \rrbracket} \llbracket A \rrbracket \xrightarrow{\llbracket f \rrbracket} \llbracket B \rrbracket$.
- $\llbracket \text{let } x = e \text{ in } t \rrbracket$ is given by $\llbracket \Gamma \rrbracket \xrightarrow{\text{copy}_{\llbracket \Gamma \rrbracket}} \llbracket \Gamma \rrbracket \otimes \llbracket \Gamma \rrbracket \xrightarrow{\text{id}_{\llbracket \Gamma \rrbracket} \otimes \llbracket e \rrbracket} \llbracket \Gamma \rrbracket \otimes \llbracket A \rrbracket \xrightarrow{\llbracket t \rrbracket} \llbracket B \rrbracket$.

The semantics can be seen as a procedure for translating every term of the CD calculus into a string diagram, as shown in Figure 5.

PROPOSITION 3.13 (STRUCTURAL RULES FOR CONTEXTS). *The weakening and exchange rules*

$$\frac{\Gamma \vdash t : B}{\Gamma, x : A \vdash t : B} \quad \frac{\Gamma, \Delta \vdash t : B}{\Delta, \Gamma \vdash t : B}$$

Congruence laws:		
	\equiv is reflexive, symmetric and transitive	(equiv)
	$\frac{e_1 \equiv e'_1 \quad e_2 \equiv e'_2}{(\text{let } x = e_1 \text{ in } e_2) \equiv (\text{let } x = e'_1 \text{ in } e'_2)}$	(let.ξ)
A <i>value expression</i> is a term of the form		
	$V ::= x \mid () \mid (V, V) \mid \pi_i V \mid \text{let } x = V \text{ in } V$	
The axioms of the CD calculus are:		
	$(\text{let } x = e \text{ in } t) \equiv t[e!x]$	(let.lin)
	$(\text{let } x = V \text{ in } t) \equiv t[V/x]$	(let.val)
	$\pi_i(x_1, x_2) \equiv x_i$	(*.β)
	$(\pi_1 x, \pi_2 x) \equiv x$	(*.η)
	$x \equiv ()$	(unit.η)
where we write $t[e!x]$ for substituting a unique free occurrence of x . For the internal language of <i>Markov categories</i> , extend (let.lin) to all substitutions targeting <i>at most one</i> free occurrence of x .		

Fig. 6. Axioms of the CD calculus.

are derivable, and their semantics corresponds to precomposition with the discard and swap morphisms.

PROOF. Straightforward induction using the comonoid axioms. \square

As an example, we use the CD calculus to give straightforward denotational semantics to the conditioning-free fragment of the Gaussian language in Gauss. We notice that this fragment is precisely the CD calculus for the signature \mathfrak{S} with base type R and function symbols

$$(+): R * R \rightarrow R \quad \beta \cdot (-): R \rightarrow R \quad \underline{\beta}: \text{unit} \rightarrow R \quad \text{normal}: \text{unit} \rightarrow R.$$

The Markov category Gauss models this signature using $\llbracket R \rrbracket = 1$ and the obvious interpretations of the function symbols.

The goal of Section 5 will be to interpret the full Gaussian language in a CD category $\text{Cond}(\text{Gauss})$. That category will need to interpret the additional function symbol $(::): R * R \rightarrow R$.

3.5 Equational Theory

We now give a sound and complete equational theory with respect to CD models.

In call-by-value languages, the substitution

$$(\text{let } x = e \text{ in } u) \equiv u[e/x]$$

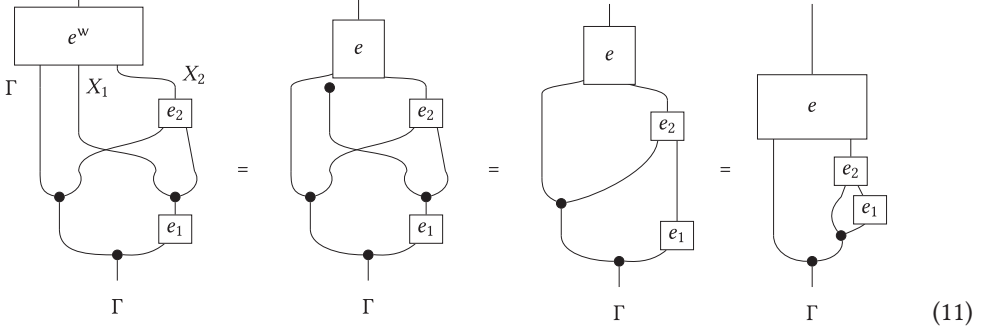
is generally only admissible if e is a *value expression*; that is, it does not produce effects. In the CD calculus, another powerful substitution scheme is valid: we can replace $(\text{let } x = e \text{ in } u) \equiv u[e/x]$ whenever u uses x *linearly*, i.e., exactly once, *even if* e is an effectful computation. Using the linear and value substitution schemes, the theory of the CD calculus can be presented concisely as in Figure 6. Note that we omit the context of equations when unambiguous and identify bound variables up to α -equivalence. Whenever we say “use” or “occurrence,” we mean free use and occurrence, and substitution is always capture avoiding.

PROPOSITION 3.14 (SOUNDNESS). *Every CD model validates the axioms of the CD calculus. That is, if $\Gamma \vdash e_1 \equiv e_2 : A$, then $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$.*

PROOF. The proofs are straightforward if tedious string diagram manipulations. We showcase the validation of one interesting equation, ([assoc](#)), here and move the remaining derivations to the appendix (Appendix [A.2](#)). Let $\Gamma \vdash e_1 : X_1$, $\Gamma, x_1 : X_1 \vdash e_2 : X_2$ and $\Gamma, x_2 : X_2 \vdash e : Y$. Then showing

$$\llbracket (\text{let } x_1 = e_1 \text{ in let } x_2 = e_2 \text{ in } e^w) \rrbracket \equiv \llbracket (\text{let } x_2 = (\text{let } x_1 = e_1 \text{ in } e_2) \text{ in } e) \rrbracket$$

translates to the following manipulation of string diagrams:



Note that we formally write e^w to be fully explicit about weakening e ; its denotation discards the unused X_1 -wire as per Proposition [3.13](#). \square

The equational theory lets us derive many useful program equations, including commutativity.

PROPOSITION 3.15. *All axioms of the ground λ_c -calculus [[35](#), Tables 6 and 7] and commutativity are derivable.*

$$\begin{aligned} (\text{let } x_2 = (\text{let } x_1 = e_1 \text{ in } e_2) \text{ in } e) &\equiv (\text{let } x_1 = e_1 \text{ in let } x_2 = e_2 \text{ in } e) && x_1 \notin \text{fv}(e) && (\text{assoc}) \\ (\text{let } x_1 = e_1 \text{ in let } x_2 = e_2 \text{ in } e) &\equiv (\text{let } x_2 = e_2 \text{ in let } x_1 = e_1 \text{ in } e) && x_1 \notin \text{fv}(e_2), x_2 \notin \text{fv}(e_1) && (\text{comm}) \\ (\text{let } x = e \text{ in } x) &\equiv e && && (\text{id}) \\ (\text{let } x_1 = x_2 \text{ in } e) &\equiv e[x_2/x_1] && && (\text{let.}\beta) \\ fe &\equiv (\text{let } x = e \text{ in } fx) && && (\text{let.f}) \\ (s, t) &\equiv (\text{let } x = s \text{ in let } y = t \text{ in } (x, y)) && && (\text{let.*}) \end{aligned}$$

Note that by commutativity ([comm](#)), the order of evaluation in ([let.*](#)) does not matter.

PROOF. In the appendix (Appendix [A.2](#)). \square

We proceed with some syntactic remarks about the CD calculus on the relationship between linear substitution to general nonlinear substitutions: If t is a term with n free occurrences of the variable x , let \hat{t} denote the term t with those occurrences replaced with distinct fresh variables x_1, \dots, x_n (the order does not matter). By repeated application of ([let.val](#)), we can derive

$$t \equiv \text{let } x_1 = x \text{ in } \dots \text{let } x_n = x \text{ in } \hat{t}. \quad (12)$$

We can now substitute some or all occurrences of x using ([let.lin](#)) as follows:

$$t[e/x] \equiv \hat{t}[e!x_1] \dots [e!x_n] \equiv \text{let } x_1 = e \text{ in } \dots \text{let } x_n = e \text{ in } \hat{t}. \quad (13)$$

This means we can reduce questions about substitution to the copying behavior of the term e . We adapt the definitions from [[14](#), [29](#)].

Definition 3.16. A term e is called *copyable* if

$$(\text{let } x = e \text{ in } (x, x)) \equiv (e, e) \quad (14)$$

is derivable. A term e is called *discardable* if

$$(\text{let } x = e \text{ in } ()) \equiv () \quad (15)$$

is derivable. We call e *deterministic* if it is both copyable and discardable.

PROPOSITION 3.17. *The substitution equation*

$$(\text{let } x = e \text{ in } t) \equiv t[e/x]$$

is derivable in any of the following circumstances:

- (1) t uses x exactly once.
- (2) t uses x at least once, and e is copyable.
- (3) t uses x at most once, and e is discardable.
- (4) e is deterministic (combining the previous two points).

Finally, we remark that the CD calculus is complete with respect to CD models. We employ the usual construction of a *syntactic category* or free CD category over a given CD signature. Not only can every term be translated into a string diagram, but also every string diagram can be parsed into a term, and the theory of \equiv proves all ways of reading a diagram equivalent.

Definition 3.18. Fix a CD signature \mathfrak{S} . The *syntactic category* Syn is given as follows:

- (1) Objects are types A .
- (2) Morphisms are equivalence classes of terms $x : A \vdash t : B$ modulo \equiv .
- (3) Identities are variables $x : A \vdash x : A$.
- (4) Composition is let binding; if $x : A \vdash s : B$ and $x : B \vdash t : C$, their composite is

$$x : A \vdash \text{let } x = s \text{ in } t.$$
- (5) Tensor on objects is defined as $A \otimes B = A * B$ with unit type unit . The tensor on morphisms of $x_1 : A_1 \vdash s_1 : B_1$, $x_2 : A_2 \vdash s_2 : B_2$ is

$$x : A_1 * A_2 \vdash \text{let } x_1 = \pi_1 x \text{ in let } x_2 = \pi_2 x \text{ in } (s_1, s_2).$$
- (6) CD structure is given by nonlinear use of variables, that is

$$\begin{aligned} \text{copy}_A &= x : A \vdash (x, x) : A * A \\ \text{del}_A &= x : A \vdash () : \text{unit}. \end{aligned}$$

The verification of the CD category axioms is tedious but standard. Note that we can build on existing work [31] because our axioms prove all equations of the ground fragment of λ_c (Proposition 3.15). We expect that the syntactic category is an initial model over a given signature and the definition of the semantics $\llbracket - \rrbracket$ is forced by preserving CD structure, but we won't formalize this here.

4 AN ABSTRACT ACCOUNT OF INFERENCE

In Section 3, we recalled Markov categories (Definition 3.3) as abstract formulations of probability theory, equipped with multiple notational formalisms: string diagram as well as programming notations. We now present an abstract theory of *inference problems* in Markov categories with sufficient structure. We approach the topic from the programming languages side. An informal outline is as follows: roughly, an *inference problem* is a closed program of the form

$$\text{let } (x, k) = \psi \text{ in } (k := o); \text{return } x \quad \text{for some } \psi \text{ and } o, \quad (16)$$

where $k := o$ is, at this point, simply a notation for recording an exact condition we wish to make—that the second marginal k of ψ is equal to the observation o (see Section 4.3). A *solution* to this problem is a contextually equivalent program that no longer mentions $k := o$. Our treatment is now guided by program equations like Equations (2) and (3) and also the symbolic approach of [42]: finding a conditional for ψ amounts to restructuring the dataflow of Equation (16) in a way that the return value x is expressed in terms of the observation k (where $\psi_K = \pi_2 \psi$):

$$\text{let } k = \psi_K \text{ in let } x = \psi|_K(k) \text{ in } (k := o); \text{return } x \quad \text{for some } \psi|_K.$$

By commutativity in Equation (1) and (comm), this is the same as

$$\text{let } k = \psi_K \text{ in } (k := o); \text{return } \psi|_K(k).$$

We can regard the initialization principle in Equation (3) as a fundamental property of exact conditions $(k := o)$ and then use this. Informally, if it is possible for k to be o ($k \ll \psi_K$, Definition 4.7), we may simply substitute the observation for the variable k :

$$\text{let } k = o \text{ in } \psi|_K(k),$$

which finally results in the solution $\psi|_K(o)$, which is equivalent to Equation (16). If on the other hand it is impossible for k to be equal to o ($k \not\ll \psi_K$), then the inference problem has no solution and is infeasible.

For the rest of this chapter, we formally express this idea of conditioning via program transformation in terms of Markov categories. We begin by recalling categorical rephrasings of core notations from measure-theoretic probability: conditional probability (Section 4.1) and almost-sure equality, absolute continuity, and support (Section 4.2), mostly due to [6, 11]. We then formulate precisely what an inference problem is and when it succeeds (Section 4.3). For now, we summarize informally: an inference problem is a pair (ψ, o) of a distribution ψ on a compound space $X \otimes K$ and a deterministic observation o about K , regarded in the spirit of Equation (16); it succeeds if we are able to infer a conditional distribution, or posterior, about X , and fails otherwise. This failure is not sought in practice but happens for instance if one attempts to record two different exact observations about the same data point, or in general if the observation o is outside the support of ψ .

Looking forward, we will develop a notion of *open inference problem* in Section 5 as part of a compositional framework for collecting conditions, so as to give a compositional semantics to the kind of programming with conditioning demonstrated in Section 2.

For the rest of this section, we fix a Markov category \mathbb{C} (Definition 3.3).

4.1 Conditionals

In essence, conditioning is a way of recovering a joint distribution only given access to part of its information. Given a joint distribution over (X, Y) , we can always form a generative story where the value of X is sampled first, and then Y is computed depending (or conditional) on X . The categorical formulations of conditioning trace back to Golubtsov and Cho and Jacobs [6, 17].

Definition 4.1 ([11, 11.1]). A *conditional distribution* for $\psi : I \rightarrow X \otimes Y$ (given X) is a morphism $\psi|_X : X \rightarrow Y$ such that

$$\begin{array}{c} X \quad Y \\ \downarrow \quad \downarrow \\ \psi \end{array} = \begin{array}{c} X \quad Y \\ \downarrow \quad \downarrow \\ \psi|_X \\ \downarrow \quad \downarrow \\ \psi \end{array} \quad (17)$$

More generally, a (*parameterized*) *conditional* for $f : A \rightarrow X \otimes Y$ is a morphism $f|_X : X \otimes A \rightarrow Y$ such that

$$\begin{array}{c} X \quad Y \\ \parallel \\ \boxed{f} \\ \downarrow \\ A \end{array} = \begin{array}{c} X \quad Y \\ \parallel \\ \boxed{f|_X} \\ \downarrow \quad \downarrow \\ \bullet \quad \bullet \\ \parallel \\ \boxed{f} \\ \downarrow \\ A \end{array} \quad (18)$$

It is worth spelling out that Equation (18), expressed in the CD calculus, corresponds precisely to the type of restructuring of dataflow that was discussed in the introduction to this chapter. Equation (18) simply becomes

$$\psi \equiv (\text{let } x = \pi_1 \psi \text{ in } (x, \psi|_X(x))).$$

Parameterized conditionals can again be specialized to conditional distributions by fixing a parameter.

PROPOSITION 4.2. *If $f : A \rightarrow X \otimes Y$ has conditional $f|_X : X \otimes A \rightarrow Y$ and $a : I \rightarrow A$ is a deterministic state, then $f|_X(\text{id}_X \otimes a)$ is a conditional distribution for the composite $f \circ a$.*

PROOF. Using determinism of a , we check that

$$\begin{array}{c} X \quad Y \\ \parallel \\ \boxed{f|_X} \\ \downarrow \quad \downarrow \\ \bullet \quad \bullet \\ \parallel \\ \boxed{f} \\ \downarrow \\ \triangle a \end{array} = \begin{array}{c} X \quad Y \\ \parallel \\ \boxed{f|_X} \\ \downarrow \quad \downarrow \\ \bullet \quad \bullet \\ \parallel \\ \boxed{f} \\ \downarrow \\ \triangle a \end{array} = \begin{array}{c} X \quad Y \\ \parallel \\ \boxed{f} \\ \downarrow \\ \triangle a \end{array}$$

□

PROPOSITION 4.3. *FinStoch, BorelStoch, Gauss, and Rel^+ have all conditionals.*

PROOF. In BorelStoch, the definition of conditionals instantiates to *regular conditional distributions* that are known to exist under the assumptions of the category (that is on *standard Borel spaces*) (see [11, 11.7], [2, Theorem 3.5]). As a special case, conditionals in FinStoch are given by the traditional conditional distribution [11, 11.2]:

$$\psi|_X(y|x) = \frac{\psi(x, y)}{\psi_X(x)} \quad \text{when } \pi_X(x) > 0. \quad (19)$$

Conditionals in Gauss exist and can be given using an explicit formula generalizing Equation (6) [11, 11.8]. The property that conditionals of Gaussians are again Gaussian is sometimes called *self-conjugacy* [24].

In Rel^+ , the conditional of the state $R \subseteq X \times Y$ with respect to X is given by “slicing” the relation

$$R|_X(x) = \{y \in Y : (x, y) \in R\},$$

which is nothing but R itself. □

4.2 Almost-sure Equality, Absolute Continuity, and Supports

Conditionals in Markov categories are generally not uniquely determined, although there is still a sense in which they are unique. For example, Equation (19) only determines $\psi|_X$ on the set $\{x : \pi(x) > 0\}$, which we call the *support* of ψ_X . Outside of the support, the conditional may be modified arbitrarily. Similarly, the formula for conditionals of Gaussian distributions (Equation (6)) depends on a choice of generalized inverse, which is only unique on the appropriate support.

If the reader is familiar with measure-theoretic probability, they will recall that the essential uniqueness of conditionals, when they exist, is usually stated in terms of “almost-sure equality” and “absolute continuity.” These have established generalizations to Markov categories in general, as we now recall (Definitions 4.4 and 4.7, respectively). These definitions specialize to the measure-theoretic concepts by fixing a Markov category (Propositions 4.5 and 4.9), but for a reader not familiar with measure-theoretic probability, the definitions can be taken as basic. A reference of measure-theoretic terminology is given in Appendix A.1.

Definition 4.4 ([6, 5.1], [11, 13.1]). Let $\mu : I \rightarrow X$ be a distribution. Two morphisms $f, g : X \rightarrow Y$ are called μ -almost surely equal (written $f =_\mu g$) if

$$\langle \text{id}_X, f \rangle \mu = \langle \text{id}_X, g \rangle \mu.$$

PROPOSITION 4.5. *For our example categories, the abstract definition of almost sure equality recovers the familiar meaning:*

- (1) In *FinStoch*, $f, g : X \rightarrow D(Y)$ are μ -almost surely equal iff the distributions $f(x) = g(x)$ agree for all x with $\mu(x) > 0$.
- (2) In *BorelStoch*, $f, g : X \rightarrow \mathcal{G}(Y)$ are μ -almost surely equal iff $f(x) = g(x)$ as measures for μ -almost all x ; that is, the set $\{x : f(x) \neq g(x)\}$ has μ -measure 0.
- (3) In *Gauss*, if $\mu : I \rightarrow m$ is a distribution with support S (in the sense of Section 2.1), then $f, g : m \rightarrow n$ are μ -almost surely equal iff $f \circ x = g \circ x$ for all elements $x \in S$, seen as deterministic states $x : 0 \rightarrow m$.
- (4) In *Rel*⁺, if $M \subseteq X$ and $R, S : X \rightarrow \mathcal{P}^+(Y)$ are two left-total relations, then $R =_M S$ iff $R(x) = S(x)$ for all $x \in M$.

PROOF. The results for *FinStoch* and *BorelStoch* are given in [11, 13.2] and [12, 3.19]. The result for *Gauss* is a strengthening of the result for *BorelStoch*. The morphisms $f, g : m \rightarrow n$ can be faithfully considered *BorelStoch* maps $f, g : \mathbb{R}^m \rightarrow \mathcal{G}(\mathbb{R}^n)$, so we have $f(x) = g(x)$ for μ -almost all x . Because f, g are furthermore continuous functions and μ is equivalent to the Lebesgue measure on the support S , the equality almost everywhere can be strengthened to equality on all of S . \square

It follows directly from the definitions that conditional distributions are almost surely unique:

PROPOSITION 4.6. *If $\psi : I \rightarrow X \otimes Y$ is a distribution and $\psi|_X, \psi|'_X$ are two morphisms satisfying Equation (18), then*

$$\psi|_X = \psi_X \psi|'_X. \quad (20)$$

That is, conditional distributions are unique almost surely with respect to the marginal ψ_X .

The important notion of *absolute continuity* can now be formulated naturally in terms of almost-sure equality:

Definition 4.7 ([12, 2.8]). Given two distributions $\mu, \nu : I \rightarrow X$, we say that μ is *absolutely continuous* with respect to ν , written $\mu \ll \nu$, if for all $f, g : X \rightarrow Y$ we have

$$f =_\nu g \text{ implies } f =_\mu g.$$

Absolute continuity lets us strengthen statements about almost-sure equality to actual equality.

LEMMA 4.8. *If $\mu : I \rightarrow X$, $f =_\mu g$, and $x \ll \mu$, then $fx = gx$.*

On the other hand, \ll recovers the usual notion of absolute continuity in our example categories.

- PROPOSITION 4.9. (1) *In FinStoch, $\mu \ll \nu$ iff $\nu(x) = 0$ implies $\mu(x) = 0$.*
 (2) *In BorelStoch, $\mu \ll \nu$ iff for all measurable sets A , $\nu(A) = 0$ implies $\mu(A) = 0$.*
 (3) *In Gauss, $\mu \ll \nu$ iff $\text{supp}(\mu) \subseteq \text{supp}(\nu)$ (in the sense of Section 2.1).*
 (4) *In Rel⁺, $R \ll S$ iff $R \subseteq S$.*

PROOF. The claim for FinStoch follows immediately from Proposition 4.5, and the result for BorelStoch is given in [12, 2.9]. Proposition 4.5 implies that the support condition for Gauss is sufficient. To see that it is also necessary, let $x \in \text{supp}(\mu) \setminus \text{supp}(\nu)$. Then we can find two affine functions f, g that agree on $\text{supp}(\nu)$ but $f(x) \neq g(x)$. Now $f =_\nu g$ but not $f =_\mu g$; hence, $\mu \not\ll \nu$. \square

In FinStoch, we can also rephrase $\mu \ll \nu$ as $\text{supp}(\mu) \subseteq \text{supp}(\nu)$ if we define $\text{supp}(\mu) = \{x : \mu(x) > 0\}$. For the purposes of our development, it will suffice to consider the special case of the absolute continuity relation restricted to deterministic states and distributions. We take this as the categorical *definition* of supports:

Definition 4.10. *If $x : I \rightarrow X$ is a deterministic state, we say that x lies in the support of μ if $x \ll \mu$.*

We obtain the following characterization:

- PROPOSITION 4.11. (1) *In FinStoch, $x \ll \mu$ iff $\mu(x) > 0$.*
 (2) *In BorelStoch, $x \ll \mu$ iff $\mu(\{x\}) > 0$.*
 (3) *In Gauss, $x \ll \mu$ iff $x \in \text{supp}(\mu)$.*
 (4) *In Rel⁺, $x \ll R$ if $x \in R$.*

It is crucial that the support of a distribution can change with the surrounding Markov category:

Example 4.12. Let $\mu = \mathcal{N}(0, 1)$ be the standard normal distribution. When considered in Gauss, its support is \mathbb{R} , and in particular for all $x_0 \in \mathbb{R}$ we have $x_0 \ll \mu$. In BorelStoch, we have $x_0 \not\ll \mu$ because $\mu(\{x_0\}) = 0$.

This means that smaller Markov categories like Gauss have a stronger notion of support, which in turn allows more interesting conditions to be evaluated. This is reminiscent of the tradeoff between expressiveness and well-behavedness discussed under the notion of “well-behaved disintegrations” in [42].

By combining the notions of conditionals and support, we can now present an abstract theory of inference problems.

4.3 Abstract Inference Problems

Let \mathbb{C} be a Markov category with all conditionals. In order to describe statistical inference categorically, we introduce the following terminology:

- (1) An *observation* is a constant piece of data, that is, a *deterministic state* $o : I \rightarrow K$.
- (2) An *inference problem* over X is a tuple (K, ψ, o) of an object K , a joint distribution $\psi : I \rightarrow X \otimes K$ called the model, and an observation $o : I \rightarrow K$.

The problem is then to infer the posterior distribution over X conditioned on the observation o . An inference problem can either succeed or fail if the observation o is inconsistent with the model.

- (1) We say (K, ψ, o) *succeeds* if the observation lies in the support of the model, i.e., $o \ll \psi_K$. In that case, a *solution* to the inference problem is the composite $\psi|_K \circ o : I \rightarrow X$, where $\psi|_K : K \rightarrow X$ is a conditional to ψ with respect to K . The solution is also referred to as a *posterior* for the problem.
- (2) If $o \not\ll \psi_K$, we say that the inference problem *fails* or is infeasible.

PROPOSITION 4.13. *Solutions to inference problems are unique; i.e., if (K, ψ, o) succeeds and $\psi|_K, \psi|'_K$ are two conditionals, then $\psi|_K(o) = \psi|'_K(o)$.*

PROOF. Combine Lemma 4.8 and Proposition 4.6. □

Definition 4.14. We call two inference problems *observationally equivalent* if they either both fail or both succeed with equal posteriors.

For the rest of this section, we will rederive Example 2.1 in terms of the categorical machinery and show that it matches the conditioning procedure from Section 2.

Example 4.15. The Example 2.1 can be written as

$$\begin{aligned} X &\sim \mathcal{N}(0, 1) \\ Y &\sim \mathcal{N}(0, 1) \\ (X - Y) &:= 0, \end{aligned}$$

which corresponds to the inference problem $(1, \mu, 0)$, where $\mu : 0 \rightarrow 2 \otimes 1$ has covariance matrix

$$\Sigma = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \\ 1 & -1 & 2 \end{pmatrix}.$$

A conditional with respect to the third coordinate Z is

$$\mu|_Z(z) = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} z + \mathcal{N} \begin{pmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{pmatrix},$$

which can be verified by calculating Equation (18). The marginal $\mu_Z = \mathcal{N}(2)$ is supported on all of \mathbb{R} ; hence, $0 \ll \mu_Z$ and by Proposition 4.13 the composite

$$\mu|_Z(0) = \mathcal{N} \begin{pmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{pmatrix}$$

is the uniquely defined solution to the inference problem.

The same inference problem would not have a solution when interpreted in BorelStoch instead of Gauss. This is because $0 \not\ll \mu_Z$ (Example 4.12). In BorelStoch, we can only condition on observations of positive probability; this agrees with the classical definition of conditional probability:

$$P(A|B) \stackrel{\text{def}}{=} \frac{P(A \cap B)}{P(B)} \text{ if } P(B) > 0.$$

In Gauss, we can also condition on probability zero observations in a principled way because the notion of support is better behaved.

5 COMPOSITIONAL CONDITIONING: THE COND CONSTRUCTION

In Section 4 we have seen that Markov categories with conditionals allow a general recipe for conditioning. In order to give *compositional* semantics to a language with conditioning, we need to internalize the conditioning operation as a morphism. The key step is to move from a closed inference problem $(K, \psi : I \rightarrow X \otimes K, o : I \rightarrow K)$ to *open inference problems* or *conditioning channels*, where ψ is replaced with a morphism with a more general domain, so that it can be composed.

With some care, we can turn these conditioning channels into a CD category (Definition 3.1, a monoidal category where every object has a comonoid structure). This allows us to give a denotational semantics to a CD calculus with conditioning, and in particular a denotational semantics for the Gaussian language with conditioning of Section 2. Looking forward, in Section 6 we will show that this denotational semantics is fully abstract: it precisely captures the contextual equivalence from the operational semantics.

The construction presented in this section is rather involved, but its well-definedness and properties are the central technical contribution of this article. From Section 5.3 onward, we can harness the good properties the construction enjoys to return to a higher-level picture and use string diagrams for studying a graphical language of conditioning.

Let \mathbb{C} be a Markov category; then a conditioning channel $X \rightsquigarrow Y$ is given by a morphism $X \rightarrow Y \otimes K$ together with an observation (i.e., deterministic state) $o : I \rightarrow K$. This represents an intensional open program of the form

$$x : X \vdash \text{let } (y, k) : Y \otimes K = f(x) \text{ in } (k := o); y. \quad (21)$$

We think of K as an additional hidden output wire, to which we attach the observation o . Such programs compose in the obvious way, by aggregating observations (Figure 7). Two representations (Equation (21)) are deemed equivalent if they are contextually equivalent; that is, roughly they compute the same posteriors in all contexts.

An important caveat is that the primary operation we formalize is that of an *exact observation* $(:= o)$, where o is a deterministic state. Binary *exact conditioning*, $(:=)$ between two expressions may be encoded in terms of $(:=)$, for example, as $(x == y) := \text{true}$ for finite sets or as $(x - y) := 0$ for Gaussians. Generally, the choice of encoding does matter (Example 2.2), so we consider this choice an additional structure and focus on formalizing $(:=)$.

For modularity, we present the construction in two stages: In the first stage (Section 5.1) we form a category $\text{Obs}(\mathbb{C})$ on the same objects as \mathbb{C} consisting of the data in Equation (21) but without any quotienting. This adds, purely formally, for every observation $o : I \rightarrow X$ an observation effect $(:= o) : X \rightsquigarrow I$. In the second stage (Section 5.2)—this is the core of the construction—we relate these morphisms to the conditionals present in \mathbb{C} ; that is, we quotient by contextual equivalence. The resulting quotient is called $\text{Cond}(\mathbb{C})$. Under mild assumptions, this will have the good properties of a CD category, showing that conditioning stays commutative. We demonstrate the resulting reasoning methods in Sections 5.3 and 5.4.

5.1 Obs: Open Programs with Observations

For ease of notation, we will assume \mathbb{C} is a strictly monoidal category; that is, all associators and unitors are identities (this poses no restriction by [11, 10.17]). We note that all constructions can instead be performed purely string-diagrammatically.

Definition 5.1. The following data define a symmetric premonoidal category called $\text{Obs}(\mathbb{C})$:

- The object part of $\text{Obs}(\mathbb{C})$ is the same as \mathbb{C} .
- Morphisms $X \rightsquigarrow Y$ are tuples (K, f, o) , where K is an object of \mathbb{C} , $f \in \mathbb{C}(X, Y \otimes K)$, and $o \in \mathbb{C}_{\text{det}}(I, K)$, representing Equation (21).

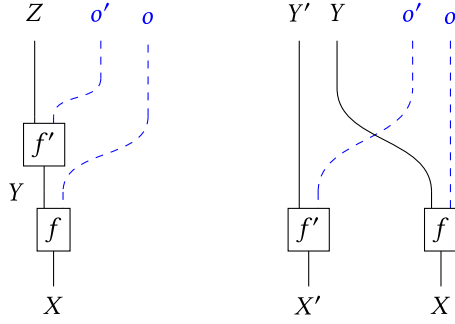


Fig. 7. Composition and tensoring of morphisms in Obs.

– The identity on X is $\text{Id}_X = (I, \text{id}_X, !)$, where $! = \text{id}_I$.

– Composition is defined by

$$(K', f', o') \bullet (K, f, o) = (K' \otimes K, (f' \otimes \text{id}_K)f, o' \otimes o).$$

– If $(K, f, o) : X \rightsquigarrow Y$ and $(K', f', o') : X' \rightsquigarrow Y'$, their (premonoidal) tensor product is defined as

$$(K' \otimes K, (\text{id}_{Y'} \otimes \text{swap}_{K', Y} \otimes \text{id}_K)(f' \otimes f), o' \otimes o).$$

– There is an identity-on-objects functor $J : \mathbb{C} \rightarrow \text{Obs}(\mathbb{C})$ that sends $f : X \rightarrow Y$ to $(I, f, !) : X \rightsquigarrow Y$. This functor is strict premonoidal and its image central.

– $\text{Obs}(\mathbb{C})$ inherits symmetry and comonoid structure.

Recall that a symmetric premonoidal category (due to [39]) is like a symmetric monoidal category where the interchange law $(f_1 \otimes f_2) \circ (g_1 \otimes g_2) = f_1 g_1 \otimes f_2 g_2$ need not hold. This is the case because $\text{Obs}(\mathbb{C})$ does not yet identify observations arriving in different order. This will be remedied automatically later when passing to the quotient $\text{Cond}(\mathbb{C})$. For an observation $o : I \rightarrow K$, the conditioning effect $(:=o) : K \rightsquigarrow I$ is defined by (I, id_K, o) .

Notation. A morphism $F : X \rightsquigarrow Y$ in $\text{Obs}(\mathbb{C})$ consists of a morphism $f : X \rightarrow Y \otimes K$ in \mathbb{C} with an extra datum $o : I \rightarrow K$. It can be convenient to describe morphisms in $\text{Obs}(\mathbb{C})$ by their underlying morphisms in \mathbb{C} , with extra “conditioning wires” into the object K and observations attached to them. Informally, we will highlight these wires with dashed lines in blue, simply to distinguish the codomain object Y from the condition object K . Using this notation, composition and tensor in $\text{Obs}(\mathbb{C})$ take the form shown in Figure 7. In Section 5.3, we will use an actual string diagram in $\text{Cond}(\mathbb{C})$ as opposed to string diagrams in \mathbb{C} , which will obviate the need for blue wires.

5.2 Cond: Contextual Equivalence of Inference Programs

Let us now assume that \mathbb{C} has all conditionals. We wish to quotient Obs-morphisms by contextual equivalence, relating them to the conditionals that can be computed in \mathbb{C} . We know how to interpret *closed* programs, because a state $(K, \psi, o) : I \rightsquigarrow X$ is precisely an inference problem as in Section 3: If $o \not\ll \psi_K$, the observation does not lie in the support of the model and conditioning fails. If not, we form the conditional $\psi|_K$ in \mathbb{C} and obtain a well-defined posterior $\mu|_K \circ o$.

The observational equivalence (Definition 4.14) defines an equivalence relation on states $I \rightsquigarrow X$ in $\text{Cond}(\mathbb{C})$. We will extend this relation to a congruence on arbitrary morphisms $X \rightsquigarrow Y$ by a general categorical construction.

Definition 5.2. Given two states $I \rightsquigarrow X$, we define $(K, \psi, o) \sim (K', \psi', o')$ if they are observationally equivalent as inference problems; that is, either

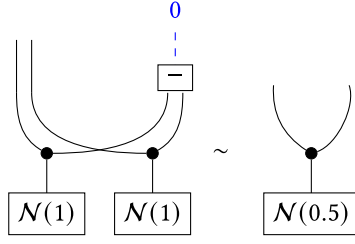


Fig. 8. Example 4.15 describes observationally equivalent states $0 \leadsto 2$ in $\text{Obs}(\text{Gauss})$.

- (1) $o \ll \psi_K$ and $o' \ll \psi'_{K'}$, and $\psi|_K(o) = \psi'|_{K'}(o')$, or
- (2) $o \not\ll \psi_K$ and $o' \not\ll \psi'_{K'}$.

Figure 8 gives an example of observationally equivalent states using Gaussian probability distributions. We now give a general recipe to extend an equivalence relation on states to a congruence on arbitrary morphisms $f : X \rightarrow Y$.

Definition 5.3. Let \mathbb{X} be a symmetric premonoidal category. An equivalence relation \sim on states $\mathbb{X}(I, -)$ is called *functorial* if $\psi \sim \psi'$ implies $f\psi \sim f\psi'$. We can extend such a relation to a congruence \approx on all morphisms $X \rightarrow Y$ via

$$f \approx g \Leftrightarrow \forall A, \psi : I \rightarrow A \otimes X, (\text{id}_A \otimes f)\psi \sim (\text{id}_A \otimes g)\psi.$$

The quotient category \mathbb{X}/\approx is symmetric premonoidal.

We show now that under good assumptions, the quotient by conditioning (Definition 5.2) on $\mathbb{X} = \text{Obs}(\mathbb{C})$ is functorial and induces a quotient category $\text{Cond}(\mathbb{C})$. The technical condition is that supports interact well with dataflow:

Definition 5.4. A Markov category \mathbb{C} has *precise supports* if the following are equivalent for all deterministic $x : I \rightarrow X$, $y : I \rightarrow Y$, and arbitrary $f : X \rightarrow Y$ and $\mu : I \rightarrow X$:

- (1) $x \otimes y \ll \langle \text{id}_X, f \rangle \mu$.
- (2) $x \ll \mu$ and $y \ll fx$.

The word “support” here refers to Definition 4.10.

PROPOSITION 5.5. Gauss, FinStoch, BorelStoch, and Rel^+ have precise supports.

PROOF. This follows from the characterizations of \ll in Proposition 4.11. For Gauss, let μ have support S and $f(x) = Ax + \mathcal{N}(b, \Sigma)$. Let T be the support of $\mathcal{N}(b, \Sigma)$. The support of $\langle \text{id}, f \rangle \mu$ is the image space $\{(x, Ax + c) : x \in S, c \in T\}$. Hence, $(x, y) \ll \langle \text{id}, f \rangle \mu$ iff $x \ll \mu$ and $y \ll fx$. Similarly, for Rel^+ , we readily verify

$$x \in \{(x, y) : x \in \mu, (x, y) \in f\} \Leftrightarrow x \in \mu \wedge y \in f(x).$$

For FinStoch, an outcome (x, y) has positive probability under $\langle \text{id}, f \rangle \mu$ iff x has positive probability under μ and y has positive probability under $f(-|x)$.

For BorelStoch, the measure $\psi = \langle \text{id}, f \rangle \mu$ is given by

$$\psi(A \times B) = \int_{x \in A} f(B|x) \mu(dx).$$

Hence, $\psi(\{(x_0, y_0)\}) = f(\{y_0|x\})\mu(\{x\})$, which is positive exactly if $\mu(\{x_0\}) > 0$ and $f(\{y_0|x\}) > 0$. \square

THEOREM 5.6. *Let \mathbb{C} be a Markov category that has conditionals and precise supports. Then \sim is a functorial equivalence relation on $\text{Obs}(\mathbb{C})$.*

PROOF. Let $(K, \psi, o) \sim (K', \psi', o') : I \rightsquigarrow X$ be equivalent states and $(H, f, v) : X \rightsquigarrow Y$ be any morphism. We need to show that the composites

$$(H \otimes K, (f \otimes \text{id}_K)\psi, v \otimes o) \sim (H \otimes K', (f \otimes \text{id}_{K'})\psi', v \otimes o') \quad (22)$$

are equivalent. We analyze different cases.

The States Fail. If a state (K, ψ, o) fails because $o \not\ll \psi_K$, then any composite must fail too. So both sides of Equation (22) fail and are thus equivalent.

The Composite Fails. Assume from now on, that the states succeed and thus also have equal posteriors:

$$\psi|_K(o) = \psi'|_K(o'). \quad (23)$$

We first show that the success conditions on both sides of Equation (22) are the same, so if the left-hand side fails, so does the right-hand side. The “precise supports” axiom lets us split the success condition into two statements; that is, the following are equivalent (and analogous for ψ', o'):

- (1) $v \otimes o \ll (f_H \otimes \text{id}_K)\psi$.
- (2) $o \ll \psi_K$ and $v \ll f_H \psi|_K(o)$.

To see this, we instantiate Definition 5.4 with the morphisms $\mu = \psi_K$ and $g = f_H \circ \psi|_K$, because the definition of the conditional $\psi|_K$ lets us recover

$$\langle g, \text{id}_K \rangle \mu = (f_H \otimes \text{id}_K)\psi.$$

It is clear that condition 2 agrees for both sides of Equation (22). Hence, so does condition 1.

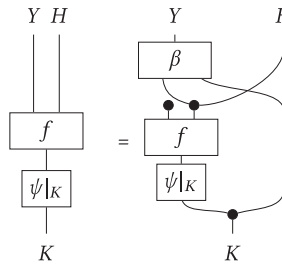
The Composite Succeeds. We are left with the case that both sides of Equation (22) succeed and need to show that the composite posteriors agree:

$$[(f \otimes \text{id}_K)\psi]|_{H \otimes K}(v \otimes o) = [(f \otimes \text{id}_{K'})\psi']|_{H \otimes K'}(v \otimes o'). \quad (24)$$

We use a variant of the argument from [11, 11.11] that double conditionals can be replaced by iterated conditionals. Consider the parameterized conditional

$$\beta \stackrel{\text{def}}{=} (f \circ \psi|_K)|_H : H \otimes K \rightarrow Y$$

with universal property

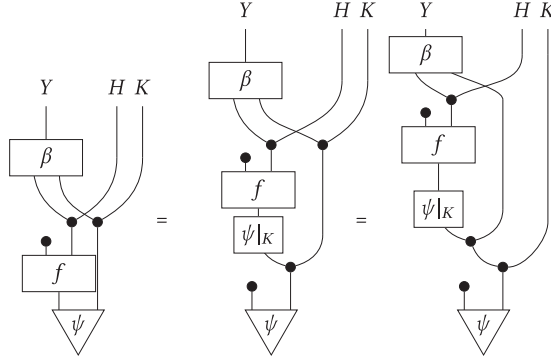


$$(25)$$

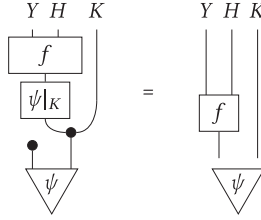
Some string diagram manipulation shows that β too has the universal property of the double conditional

$$\beta = [(f \otimes \text{id}_K)\psi]|_{H \otimes K}.$$

We check



which further reduces using Equation (25) to the desired



By specialization (Proposition 4.2), we can fix one observation o in β to obtain a conditional:

$$\beta(\text{id}_H \otimes o) = (f \circ \psi|_K(o))|_H. \quad (26)$$

But this conditional agrees with $(f \circ \psi'|_K(o'))|_H$ by Equation (23). Hence, we can evaluate the joint posterior successively:

$$\begin{aligned} [(f \otimes \text{id}_K)\psi]|_{H \otimes K}(v \otimes o) &= \beta(\text{id}_H \otimes o) \circ v \\ &\stackrel{(26)}{=} (f \circ \psi|_K(o))|_H \circ v \\ &\stackrel{(23)}{=} (f \circ \psi'|_K(o'))|_H \circ v \\ &\stackrel{\text{symmetric}}{=} [(f \otimes \text{id}_{K'})\psi']|_{H \otimes K'}(v \otimes o), \end{aligned}$$

establishing Equation (24). \square

We can spell out the induced congruence \approx on $\text{Obs}(X, Y)$ as follows:

PROPOSITION 5.7. *We have $(K, f, o) \approx (K', f', o') : X \rightsquigarrow Y$ if and only if for all $\psi : I \rightarrow A \otimes X$, either*

- (1) $o \ll f_K \psi_X$ and $o' \ll f'_{K'} \psi'_X$ and $[(\text{id}_A \otimes f)\psi]|_K(o) = [(\text{id}_A \otimes f')\psi']|_{K'}(o')$, or
- (2) $o \not\ll f_K \psi_X$ and $o' \not\ll f'_{K'} \psi'_X$.

Furthermore, because \mathbb{C} has conditionals, it is sufficient to check these conditions for $A = X$ and ψ of the form $\text{copy}_X \circ \phi$.

By Definition 5.3, the quotient $\text{Obs}(\mathbb{C})/\approx$ is a well-defined symmetric premonoidal category. We argue now that it is in fact monoidal. Checking the interchange means showing that the order of observations does not matter modulo \approx . We can derive this from a general statement about isomorphic conditions.

PROPOSITION 5.8 (ISOMORPHIC CONDITIONS). *Let $(K, f, o) : X \rightsquigarrow Y$ and $\alpha : K \cong K'$ be an isomorphism. Then*

$$(K, f, o) \approx (K', (\text{id}_Y \otimes \alpha)f, \alpha o).$$

In programming terms, the observations $(k := o)$ and $(\alpha k := \alpha o)$ are contextually equivalent.

PROOF. Let $\psi : I \rightarrow A \otimes X$. We first notice that $o \ll \psi_K$ if and only if $\alpha o \ll \alpha \psi_K$, so the success conditions coincide. It is now straightforward to check the universal property:

$$(\text{id}_A \otimes f)\psi|_K = (\text{id}_A \otimes ((\text{id}_X \otimes \alpha)f))\psi|_{K'} \circ \alpha.$$

This requires the fact that isomorphisms are deterministic in a Markov category with conditionals [11, 11.26]. The proof more generally works if α is deterministic and split monic. \square

We can now give the Cond construction by means of quotienting $\text{Obs}(\mathbb{C})$ modulo contextual equivalence.

Definition 5.9. Let \mathbb{C} be a Markov category that has conditionals and precise supports. We define $\text{Cond}(\mathbb{C})$ as the quotient category:

$$\text{Cond}(\mathbb{C}) = \text{Obs}(\mathbb{C}) / \approx.$$

This quotient is a CD category, and the functor $J : \mathbb{C} \rightarrow \text{Cond}(\mathbb{C})$ preserves CD structure.

5.3 Laws for Conditioning

We will now establish convenient properties of $\text{Cond}(\mathbb{C})$ in a purely abstract way. In terms of the internal language, those are the desired program equations for a language with exact conditioning. For example, the fact that $\text{Cond}(\mathbb{C})$ is a well-defined CD category already implies that the commutativity equation holds for such programs (Proposition 3.15).

Secondly, we can draw string diagrams in the category $\text{Cond}(\mathbb{C})$. These look like diagrams in \mathbb{C} to which we add effects $(:= o) : X \rightarrow I$ for every observation $o : I \rightarrow X$. For example, Proposition 5.8 states diagrammatically that for all isomorphisms α and observations o , we have

$$\begin{array}{c} \triangle o \\ | \end{array} = \begin{array}{c} \triangle \alpha o \\ \square \alpha \\ | \end{array}$$

We begin by showing that passing to $\text{Cond}(\mathbb{C})$ generally does not collapse morphisms that were distinct in \mathbb{C} ; that is, the functor J is faithful for common Markov categories.

PROPOSITION 5.10. *Two morphisms $f, g : X \rightarrow Y$ are equated via $J(f) \approx J(g)$ if and only if*

$$\forall \psi : I \rightarrow A \otimes X, (\text{id}_A \otimes f)\psi = (\text{id}_A \otimes g)\psi. \quad (27)$$

In particular, J is faithful whenever I is a separator. This is the case for Gauss, FinStoch, BorelStoch, and Rel^+ .

PROOF. Directly from the definition of \approx . \square

By construction, the states in $\text{Cond}(\mathbb{C})$ are precisely inference problems up to observational equivalence. Any such problem either fails or computes a well-defined posterior, which gives rise to the following classification:

PROPOSITION 5.11 (STATES IN Cond). *The states $I \rightsquigarrow X$ in $\text{Cond}(\mathbb{C})$ are of the following form:*

- (1) There exists a unique failure state $\perp_X : I \rightsquigarrow X$ given by the equivalence class of any (K, ψ, o) with $o \not\ll \psi_K$.²
- (2) Any other state is equal to a conditioning-free posterior, namely $(K, \psi, o) \approx J(\psi|_K \circ o)$. That is, diagrammatically,

$$\begin{array}{c} X \\ | \\ \triangleup_o \\ | \\ \psi \\ \triangle \end{array} = \begin{array}{c} X \\ | \\ \psi|_K o \\ \triangle \end{array} \quad \text{if } o \ll \psi_K, \text{ and} \quad \begin{array}{c} X \\ | \\ \perp_X \\ \triangle \end{array} \quad \text{otherwise}$$

- (3) Failure is “strict” in the sense that any composite or tensor with \perp gives \perp .
- (4) The only scalars $I \rightsquigarrow I$ are id_I and \perp_I . Both are copyable, but \perp_I is not discardable.

PROOF. By definition of \sim . □

COROLLARY 5.12. If $o \ll \psi$, then $(\psi := o)$ succeeds without observable effect; in particular, because $o \ll o$, we can always eliminate tautological conditions:

$$\begin{array}{c} \triangleup_o \\ | \\ \triangle_o \end{array} = \text{(empty diagram)}$$

The central law of conditioning states that after we enforce a condition, it will hold with exactness. In programming terms, this is the substitution principle (Equation (2)). Categorically, we are asking how the conditioning effect interacts with copying:

PROPOSITION 5.13 (ENFORCING CONDITIONS). We have

$$(X, \text{copy}_X, o) \approx (X, o \otimes \text{id}_X, o).$$

In programming notation, this is

$$(x := o); x \approx (x := o); o,$$

and in string diagrams,

$$\begin{array}{c} K \\ \curvearrowright \bullet \\ | \end{array} \triangleup_o = \begin{array}{c} K \\ | \\ \triangle_o \end{array} \triangleup_o \quad (28)$$

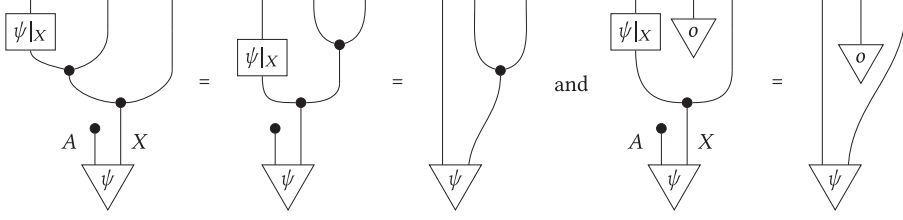
Note that the conditioning effect *cannot* be eliminated; however, after the condition takes place, the other wire can be assumed to now contain o .

PROOF. Let $\psi : I \rightarrow A \otimes X$; the success condition reads $o \ll \psi_X$ both cases. Now let $o \ll \psi_X$ and let $\psi|_X$ be a conditional distribution for ψ . The following maps give the required conditionals:

$$[(\text{id}_A \otimes \text{copy}_X)\psi]|_X = \langle \psi|_X, \text{id}_X \rangle \quad [(\text{id}_A \otimes o \otimes \text{id}_X)\psi]|_X = \psi|_X \otimes o$$

²It is a minor extra assumption that there exists a non-instance $o \not\ll \mu$ in \mathbb{C} ; this should be the case in any Markov category of practical interest.

as evidenced by the following string diagrams:



Composing with o , we obtain the desired equal posteriors:

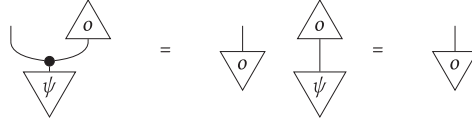
$$\langle \psi|_X, \text{id}_X \rangle o = \psi|_X(o) \otimes o = (\psi|_X \otimes o)(o)$$

from determinism of o . □

COROLLARY 5.14 (INITIALIZATION). *Conditioning a fresh variable on a feasible observation makes it assume that observation. Formally, if $o \ll \psi$, then*

$$(\text{let } x = \psi \text{ in } (x := o); x) \approx o.$$

PROOF. Combining Proposition 5.13 and Corollary 5.12, we have



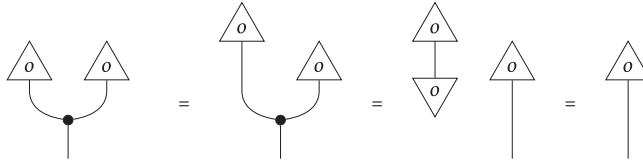
□

COROLLARY 5.15 (IDEMPOTENCE). *Conditioning is idempotent; that is,*

$$(x := o); (x := o) \approx (x := o).$$

In other words, the conditioning effect is copyable (but not discardable).

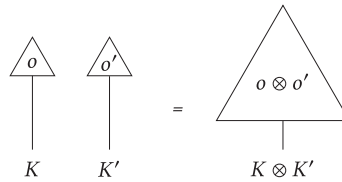
PROOF. Again by Proposition 5.13 and Corollary 5.12, we obtain



□

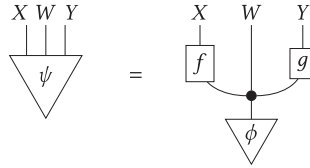
We note that this does not imply that *every* effect in $\text{Cond}(\mathbb{C})$ is copyable, only that exact observations are.

PROPOSITION 5.16 (AGGREGATION). *Conditions can be aggregated:*



PROOF. By definition of the monoidal structure of Obs . □

We demonstrate the power of our conditioning laws by briefly revisiting graphical models as mentioned in the introduction of Section 3: every graphical model can be turned into a string diagram, where the independence structure of the graphical model translates into a factorization of the diagram. For example, in the model of Equation (??) of variables X, Y , which are conditionally independent on W , the joint distribution ψ can be factored as follows:



(29)

The diagrammatic proof shows the equality of two expressions for the tensor product of two linear maps f and g . The top row shows the initial expression as a tensor product of two separate components. The middle row shows the first component with a dot on the line from f to the ϕ triangle, and the second component with a dot on the line from g to its ϕ triangle. The bottom row shows the final simplified expression where the two components are connected by a dot on the line from f to a shared ϕ triangle, and the second component is now a single w triangle connected to the ϕ triangle.

In Section 5, we introduced the Cond construction (Definition 5.9) as a way of building a category that accommodates the abstract inference for Markov categories (Section 4). As we have seen, we can interpret the CD calculus (Section 3) in categories built from the Cond construction, and this forms a probabilistic programming language with exact conditioning. In this final section, we will work out in detail what the Cond construction does when applied to our specific example settings of finite and Gaussian probability.

In Section 6.1, we show that the Gaussian language (Section 2) has fully abstract denotational semantics in $\text{Cond}(\text{Gauss})$: equality in the category coincides with the operational contextual equivalence from Section 2.3.

In Section 6.2, we conduct the same analysis for finite probability and show that $\text{Cond}(\text{FinStoch})$ consists of substochastic kernels up to *automatic normalization*. In Section 6.3, we spell out the relationship between the admissibility of automatic normalization and the expressibility of branching in the language.

6.1 Full Abstraction for the Gaussian Language

The Gaussian language embeds into the internal language of $\text{Cond}(\text{Gauss})$, where $x \coloneqq y$ is translated as $(x - y) \coloneqq 0$. A term $\vec{x} : R^m \vdash e : R^n$ denotes a conditioning channel $\llbracket e \rrbracket : m \leadsto n$.

PROPOSITION 6.1 (CORRECTNESS). *If $(e, \psi) \triangleright (e', \psi')$, then $\llbracket e \rrbracket \psi = \llbracket e' \rrbracket \psi'$. If $(e, \psi) \triangleright \perp$, then $\llbracket e \rrbracket = \perp$.*

PROOF. We can faithfully interpret ψ as a state in both Gauss and $\text{Cond}(\text{Gauss})$. If $x \vdash e$ and $(e, \psi) \triangleright (e', \psi')$, then e' has potentially allocated some fresh latent variables x' . We show that

$$\text{let } x = \psi \text{ in } (x, \llbracket e \rrbracket) = \text{let } (x, x') = \psi' \text{ in } (x, \llbracket e' \rrbracket). \quad (30)$$

This notion is stable under reduction contexts.

Let C be a reduction context. Then

$$\begin{aligned} \text{let } x &= \psi \text{ in } (x, \llbracket C[e] \rrbracket(x)) \\ &= \text{let } x = \psi \text{ in let } y = \llbracket e \rrbracket(x) \text{ in } (x, \llbracket C \rrbracket(x, y)) \\ &= \text{let } (x, x') = \psi' \text{ in let } y = \llbracket e' \rrbracket(x, x') \text{ in } (x, \llbracket C \rrbracket(x, y)) \\ &= \text{let } (x, x') = \psi' \text{ in } (x, \llbracket C[e'] \rrbracket). \end{aligned}$$

Now for the redexes:

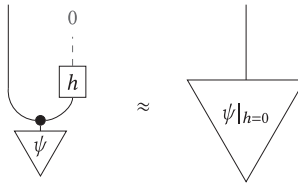
- (1) The rules for `let` follow from the general axioms of value substitution in the internal language.
- (2) For `normal()`, we have $(\text{normal}(), \psi) \triangleright (x', \psi \otimes \mathcal{N}(0, 1))$ and verify

$$\begin{aligned} \text{let } x &= \psi \text{ in } (x, \llbracket \text{normal}() \rrbracket) \\ &= \psi \otimes \mathcal{N}(0, 1) \\ &= \text{let } (x, x') = \psi \otimes \mathcal{N}(0, 1) \text{ in } (x, \llbracket x' \rrbracket). \end{aligned}$$

- (3) For conditioning, we have $(v \coloneqq w, \psi) \triangleright ((), \psi|_{v=w})$. We need to show

$$\text{let } x = \psi \text{ in } (x, \llbracket v \coloneqq w \rrbracket) = \text{let } x = \psi|_{v=w} \text{ in } (x, ()).$$

Let $h = v - w$; then we need to show that the following morphisms are equivalent in $\text{Cond}(\text{Gauss})$:



Applying Proposition 5.11 to the left-hand side requires us to compute the conditional $\langle \text{id}, h \rangle \psi|_2 \circ 0$, which is exactly how $\psi|_{h=0}$ is defined. \square

THEOREM 6.2 (FULL ABSTRACTION). $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket$ if and only if $e_1 \approx e_2$ (where \approx is contextual equivalence, Definition 2.5).

PROOF. For \Rightarrow , let $K[-]$ be a closed context. Because $\llbracket - \rrbracket$ is compositional, we obtain $\llbracket K[e_1] \rrbracket = \llbracket K[e_2] \rrbracket$. By Proposition 2.3, if both succeed, we have reductions $(K[e_i], !) \triangleright^* (v_i, \psi_i)$, and by correctness, $v_1\psi_1 = \llbracket K[e_1] \rrbracket = \llbracket K[e_2] \rrbracket = v_2\psi_2$, as desired. If $\llbracket K[e_1] \rrbracket = \llbracket K[e_2] \rrbracket = \perp$, then both $(K[e_i], !) \triangleright^* \perp$.

For \Leftarrow , we note that Cond quotients by contextual equivalence, but all Gaussian contexts are definable in the language. \square

6.2 Contextual Equivalence for Finite Probability

With programs over a finite domain, we can understand conditioning in terms of rejection sampling. This means that we run a program N times, with different random choices each time. We reject those runs that violate the conditions, and then we resample from among the acceptable results. As $N \rightarrow \infty$, this random distribution converges to the probability distribution that the program describes.

The following reformulation is semantically equivalent: For a closed program, suppose the program would return some value x_1 with probability p_1 , value x_2 with probability p_2 , and so on. Then the probability that the program will not fail is $Z = \sum_i p_i$. The result of rejection sampling is a program that actually returns x_i with probability $\frac{p_i}{Z}$, so that we have a normalized probability distribution over $\{x_1 \dots x_n\}$, i.e., $\sum \frac{p_i}{Z} = 1$. The quantity Z is called the *normalization constant* of the program, or sometimes *model evidence*.

For example, the program

$$\text{let } x = \text{bernoulli}(0.4) \text{ in let } y = \text{bernoulli}(0.4) \text{ in } x :: y; x$$

will fail the condition with probability $2 \cdot 0.4 \cdot 0.6 = 0.48$, return true with probability $0.4^2 = 0.16$, and return false with probability $0.6^2 = 0.36$. Under rejection sampling, once we renormalize, the program is equivalent to $\text{bernoulli}(\frac{0.16}{0.36}) \approx \text{bernoulli}(0.44)$.

Rejection sampling makes sense for closed programs. For programs with free variables, we can still understand a program that rejects runs that violate the conditions, but normalization is more subtle. For example, in the program

$$\text{let } y = \text{bernoulli}(0.4) \text{ in } x :: y; x, \tag{31}$$

the normalizing constant is either 0.4 or 0.6, depending on the value of x . If we normalize regardless of the value of x , then the meaning of the program must change, because it would simply return x , and the context

$$\text{let } x = \text{bernoulli}(0.4) \text{ in } [-]$$

distinguishes this.

There is nonetheless some normalization that can be done in straight-line programs, since, e.g., the meaning is *not* changed by prefixing a program with a closed program. It is, for example, safe to regard Equation (31) as equivalent to

$$\text{let } z = \text{bernoulli}(0.2) \text{ in } z :: \text{false}; \text{let } y = \text{bernoulli}(0.4) \text{ in } x :: y; x, \tag{32}$$

because the difference in normalizing constant will be the same for both values of x .

Semantically, the interpretation of a program with free variables is a stochastic kernel, and one involving rejection too is a substochastic kernel (Definition 3.5). As we show, we can accommodate multiplication by a constant if it is uniform across all arguments; this is what we call “projectivized” substochastic kernels, by analogy with the construction of a projective space from a vector space.

Definition 6.3. The CD category FinProjStoch of *projectivized substochastic kernels* is a quotient of the CD category of FinSubStoch of substochastic maps:

- (1) Objects are finite sets X .
- (2) Morphisms $X \rightarrow Y$ are equivalence classes $[p]$ of substochastic kernels $p(y|x)$ up to a scalar. That is, we identify p and q if there exists a number $\lambda > 0$ such that $p(y|x) = \lambda \cdot q(y|x)$ for all $x \in X, y \in Y$. In this circumstance we write $p \propto q$.

It is routine to verify that the monoidal and CD category structure are preserved by this quotient.

THEOREM 6.4. *The CD categories $\text{Cond}(\text{FinStoch})$ and FinProjStoch are equivalent.*

PROOF. Sketch. Given a conditioning channel $Q : X \rightsquigarrow Y$ presented by a finite set of observations K , a probability kernel $q(y, k|x)$, and an observation $k_0 \in K$, we associate to it the subprobability kernel $\rho_Q : X \rightarrow D_{\leq 1}(Y)$ given by the likelihood function

$$\rho_Q(y|x) = q(y, k_0|x).$$

Conversely, we associate to every subprobability kernel $\rho : X \rightarrow D_{\leq 1}(Y)$ a conditioning channel $Q_\rho = (\{0, 1\}, q_\rho, 1)$ with a single Boolean observation $b := 1$, defined as

$$q_\rho(y, b|x) = b \cdot \rho(y|x) + (1 - b) \cdot (1 - \rho(y|x)).$$

We recover the subprobability kernel ρ_Q from the conditioning channel Q that way: given any distribution $p(x)$, the posterior in Proposition 5.7 is given by

$$\frac{p(x)q(y, k_0|x)}{\sum_{x,y} p(x)q(y, k_0|x)}.$$

We see that q_ρ computes the same posterior, namely

$$\frac{p(x)q_\rho(y, 1|x)}{\sum_{x,y} p(x)q_\rho(y, 1|x)} = \frac{p(x)\rho(y|x)}{\sum_{x,y} p(x)\rho(y|x)} = \frac{p(x)q(y, k_0|x)}{\sum_{x,y} p(x)q(y, k_0|x)}.$$

On the other hand, the subprobability kernel ρ can be recovered from Q_ρ up to a constant. For a uniform prior $p(x) = 1/|X|$, the posterior under Q_ρ in Proposition 5.7 becomes

$$\frac{\rho(y|x)}{\sum_{x,y} \rho(x, y)},$$

from which we can read off $\rho(y|x)$ up to the constant in the denominator. \square

Identifying scalar multiples is necessary because the Cond construction by definition “normalizes automatically.” That is, it considers two conditioning channels equivalent if they compute the same posterior distributions for all priors. We will explore the relationship with model evidence and branching in Section 6.3.

We briefly showcase some of the structure of this category by characterizing the discardable morphisms and observing that conditioning gives a commutative monoid structure. The latter gives a characterization for the finite uniform distributions as units for conditioning.

Example 6.5. A projectivized subprobability kernel $p : X \rightarrow Y$ is discardable (Definition 3.2) if and only if there exists a constant $\lambda \neq 0$ such that

$$\forall x, \sum_y p(y|x) = \lambda.$$

As an instance of Proposition 5.11, in particular, to give a state in $\text{FinProjStoch}(1, Y)$ is to give either a normalizable distribution $p \in \text{FinStoch}(1, Y)$ or the failure kernel $\perp_Y = 0$.

Definition 6.6 (Conditioning Product). In $\text{Cond}(\text{FinStoch})$, we define an exact conditioning operation $x \text{ ::= } y$ by exactly observing true from the Boolean equality test $(x == y)$. We define a morphism $\bullet : X \times X \leadsto X$ by

$$x \bullet y \stackrel{\text{def}}{=} (x \text{ ::= } y); x.$$

In terms of projectivized subprobability kernels, this is

$$\bullet(z|x, y) = \begin{cases} 1 & z = x = y \\ 0 & \text{otherwise.} \end{cases}$$

We call \bullet the *conditioning product*.

Concretely, for subdistributions $p(x), q(x)$, the subdistribution $(p \bullet q)$ has the product of mass functions

$$(p \bullet q)(x) = p(x) \cdot q(x).$$

PROPOSITION 6.7. *The conditioning product defines a commutative monoid structure on X , where the unit is given by the uniform distribution $u_X : 1 \leadsto X$.*

PROOF. The operation \bullet is commutative and associative already in FinSubStoch ; however, it does not have a unit. Conditioning with the uniform distribution produces a global factor of $1/|X|$, which is cancelled by the proportionality relation. Therefore, u_X is a unit for \bullet in $\text{Cond}(\text{FinStoch})$. \square

This is intuitive in programming terms: observing from a uniform distribution gives no new information. Such a conditioning statement can thus be discarded.

Aside on non-determinism. We show the analogous version of Theorem 6.4 for non-determinism. The Cond construction here does nothing more than add the possibility for failure (zero outputs) in a systematic way.

PROPOSITION 6.8. $\text{Cond}(\text{Rel}^+) \cong \text{Rel}$.

PROOF. Given a conditioning channel (K, R, k_0) with $R \subseteq X \times Y \times K$ left-total in X , we define a possibly non-total relation $R' \subseteq X \times Y$ by $R' = \{(x, y) : (x, y, k_0) \in R\}$. On the other hand, given R' , we form the conditioning channel $(2, R'', 1)$ with left-total relation $R'' \subseteq X \times Y \times 2$ defined as

$$(x, y, b) \in R'' \stackrel{\text{def}}{\Leftrightarrow} ((x, y) \in R' \Leftrightarrow (b = 1)).$$

These constructions are easily seen to be inverses. Any relation R' is recovered from R'' , and we have $(K, R, k_0) \approx (2, R'', 1)$ because Proposition 5.7 boils down to checking that $(x, y, k_0) \in R \Leftrightarrow (x, y, 1) \in R''$. \square

The conditioning product \bullet in Rel is the relation $\{(x, x, x) : x \in X\}$, and on states we have $R \bullet S = R \cap S$. The conditioning product has a unit $v_X : 1 \rightarrow X$ given by the maximal subset $v_X = X$.

6.3 Automatic Normalization and Straight-line Inference

By *automatic normalization*, we mean that two (open) probabilistic programs that differ by an overall normalization constant Z are considered equivalent, as formalized in Section 6.2. As a consequence, the precise value of the normalization constant cannot be extracted operationally from such programs, which is a limitation whenever Z is itself a quantity of interest. On the other hand, auto-normalization is a convenient optimization, as seen in Equation (32) or Proposition 6.7.

In this section, we argue that validity of auto-normalization is tied to the form of branching available in language under consideration. We distinguish *straight-line inference programs* with a

static structure of conditions from programs where we can dynamically choose whether to execute conditions or not. This is sufficient for inference in Bayesian networks, in which the observed nodes are determined statically. For example, the Bayesian network depicted in Equation (??) corresponds to the straight-line program in Figure 1(c). Auto-normalization is valid for straight-line programs but not for those with more general branching. The quotient from Section 6.2 arises naturally from studying contextual equivalence of a straight-line inference language. In semantical terms, this means $\text{Cond}(\text{FinStoch})$ does not have coproducts.

We consider the CD calculus (Section 3.3) with base types X for all finite sets and function symbols \underline{f} for all subprobability kernels $f : X \rightarrow D_{\leq 1}(Y)$:

$$t ::= x \mid () \mid (t, t) \mid \pi_i t \mid \underline{f}(t) \mid \text{let } x = t_1 \text{ in } t_2.$$

This language can express scoring and exact conditioning because there exist suitable subprobability kernels:

$$\text{score}_p \in D_{\leq 1}(1) \text{ and } (:=) : X \times X \rightarrow D_{\leq 1}(1).$$

We denote this calculus **PSL**, for *straight-line inference*. Following the development in Section 3.13, the language **PSL** has canonical denotational semantics in FinSubStoch . (In fact, **PSL** is precisely the internal language of FinSubStoch as a CD category.)

We also consider a richer language, **P**, which contains the syntax of **PSL** and also if-then-else branching:

$$t ::= \dots \mid \text{if } t_1 \text{ then } t_2 \text{ else } t_3$$

with typing rule

$$\frac{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : A}{\Gamma \vdash t_1 : 2 \quad \Gamma \vdash t_2 : A \quad \Gamma \vdash t_3 : A}.$$

This language can also be interpreted in FinSubStoch , via

$$\llbracket \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rrbracket(a|\gamma) = \llbracket t_2 \rrbracket(a|\gamma) \cdot \llbracket t_1 \rrbracket(\text{true}|\gamma) + \llbracket t_3 \rrbracket(a|\gamma) \cdot \llbracket t_1 \rrbracket(\text{false}|\gamma).$$

Categorically, this makes use of the distributive coproducts in FinSubStoch [38, 45]. **P** is a commonly considered probabilistic language, and subprobability kernel semantics are already known to be fully abstract, though we rederive this here.

As explained in the introduction to this section, a program in the language **P** or **PSL** is typically executed by some sort of inference engine, which tries to sample (usually approximately) from the posterior distribution it defines. In the semantics, we express this top-level normalization for a finite set X using the function $\text{normalize} : D_{\leq 1}(X) \rightarrow D_{\leq 1}(X)$, which is defined as

$$\text{normalize}(\varphi)(x) = \begin{cases} \frac{1}{Z} \cdot \varphi(x) & \text{where } Z = \sum_{x \in X} \varphi(x) \neq 0 \\ 0 & \text{where } \forall x. \varphi(x) = 0. \end{cases}$$

The zero distribution is mapped to itself, signaling failure of normalization.

Definition 6.9. Two closed **P** programs $s, t : X$ are called *observationally equivalent*, written $s \approx t$, if the normalized distributions they define are equal, that is, $\text{normalize}(\llbracket s \rrbracket) = \text{normalize}(\llbracket t \rrbracket)$.

We say that two open programs $\Gamma \vdash s, t : X$ are contextually equivalent if under every closed context $C[-]$ we have $C[s] \approx C[t]$. The distinguishing power crucially depends on the fragment of the language we are allowed to use in the contexts $C[-]$.

Definition 6.10. The terms s, t are called *straight-line equivalent*, written $s \approx_{\text{PSL}} t$, if for every closed context $C[-]$ in **PSL** (without branching) we have $C[s] \approx C[t]$. The terms s, t are called *branching equivalent*, written $s \approx_{\text{P}} t$, if for every closed context $C[-]$ in **P** (possibly involving branching) we have $C[s] \approx C[t]$.

The following proposition shows that straight-line equivalence can distinguish subprobability kernels up to a constant.

PROPOSITION 6.11. *Two open programs are straight-line equivalent iff their denotations are proportional:*

$$s \approx_{\text{PSL}} t \Leftrightarrow \llbracket s \rrbracket \propto \llbracket t \rrbracket.$$

That is, FinProjStoch is fully abstract for the language PSL.

PROOF. \Leftarrow It is easy to show that the semantics of all PSL constructs are linear or bilinear and hence respect the relation \propto . \Rightarrow Let $s \approx_{\text{PSL}} t$ and consider the straight-line context

$$C[t] \stackrel{\text{def}}{=} \text{let } x = u_X \text{ in } (x, t),$$

where again u_X denotes the uniform distribution on the finite set X . Its denotation is

$$\llbracket C[t] \rrbracket(x, y) = \frac{1}{|X|} \llbracket t \rrbracket(y|x).$$

By assumption, $\llbracket C[s] \rrbracket \propto \llbracket C[t] \rrbracket$, so we have $\llbracket s \rrbracket \propto \llbracket t \rrbracket$. \square

This gives a clear interpretation of Theorem 6.4. In our current terminology, the Cond construction aims to give a canonical semantics for straight-line inference programs: the construction presents a normal form for straight-line programs up to straight-line equivalence. The lack of branching is reflected in the fact that unlike FinSubStoch, FinProjStoch does not have coproducts.

Branching Inference. Up to straight-line equivalence, programs that differ by a global constant are not distinguishable; that is, auto-normalization is valid. This changes if we allow branching in the contexts, because branching can be used to extract the normalization constant. This trick is fundamental to so-called “Bayesian model selection.” If y is a closed program, then the Boolean program

if bernoulli(0.5) then y ; true else false

normalizes to a distribution that returns true with probability

$$p = \frac{0.5 \cdot Z}{0.5 \cdot Z + 0.5} = \frac{Z}{Z + 1} \quad \text{where } Z = \sum_x \llbracket y \rrbracket(x)$$

and false with probability $\frac{1}{Z+1}$. Because the assignment $Z \mapsto Z/(Z+1)$ is a bijection $[0, \infty) \rightarrow [0, 1)$, we can recover Z from the probability p . It follows that FinSubStoch is fully abstract for \mathbf{P} .

PROPOSITION 6.12. *Two open programs s, t are branching equivalent if their denotations are equal as subprobability kernels:*

$$s \approx_{\mathbf{P}} t \Leftrightarrow \llbracket s \rrbracket = \llbracket t \rrbracket : \llbracket \Gamma \rrbracket \rightarrow D_{\leq 1}(\llbracket X \rrbracket).$$

That is, FinSubStoch is fully abstract for the language \mathbf{P} .

PROOF. From straight-line equivalence, we know that $\llbracket s \rrbracket = \lambda \cdot \llbracket t \rrbracket$. We then use the “Bayesian model selection” trick to show $\lambda = 1$. \square

The tradeoff between straight-line inference and branching inference is an interesting design decision: Branching inference is more general and allows us to extract the normalization constant. On the other hand, restricting ourselves to straight-line inference, we are free to normalize at any point, which leads to an appealing equational theory. For example, an “uninformative” observation from a uniform distribution can be eliminated (Proposition 6.7).

We emphasize that the crucial difference between **PSL** and **P** lies in putting conditions in branches. Even in **PSL**, we can still implement if-then-else when the branches do not involve conditions, because we can make use of the probability kernel

$$\text{ite} : 2 \times X \times X \rightarrow D(X)$$

with $\text{ite}(1, x, y) = \delta_x$ and $\text{ite}(0, x, y) = \delta_y$. If t_1, t_2 are discardable (condition-free) terms, we can define

$$(\text{if } c \text{ then } t_1 \text{ else } t_2) \stackrel{\text{def}}{=} \text{ite}(c, t_1, t_2).$$

In **PSL**, the structure of conditions is static, while it is dynamic in **P**.

7 CONTEXT, RELATED WORK, AND OUTLOOK

7.1 Symbolic Disintegration, Consistency, and Paradoxes

Our line of work can be regarded as a synthetic and axiomatic counterpart of the symbolic disintegration of [42] (see also [15, 36, 37, 51]). That work provides in particular verified program transformations to convert an arbitrary probabilistic program of type $R \otimes \tau$ to an equivalent one that is of the form

$$\text{let } x = \text{lebesgue}() \text{ in let } y = M \text{ in } (x, y).$$

Now the exact conditioning $x := o$ can be carried out by substituting o for x in M . We emphasize the similarity to our treatment of *inference problems* in Section 3, as well as the role that coordinate transformations play in both of our work, [49] and [42]. One language novelty in our work is that exact conditioning is a first-class construct in our language, as opposed to a whole-program transformation, which makes the consistency of exact conditioning more apparent.

Consistency is a fundamental concern for exact conditioning. *Borel's paradox* is an example of an inconsistency that arises if one is careless with exact conditioning ([26, Ch. 15], [25, Section 3.3]): it arises when naively substituting equivalent equations within $(::=)$. For example, the equation $x - y = 0$ is equivalent to $x/y = 1$ over the (non-zero) real numbers. Yet, in a hypothetical extension of our language that allows division, the following programs would not be contextually equivalent, as discussed in Example 2.2:

$$\begin{array}{ll} x = \text{normal}(0, 1) & x = \text{normal}(0, 1) \\ y = \text{normal}(0, 1) & \neq y = \text{normal}(0, 1) \\ x - y ::= 0 & x/y ::= 1 \end{array}$$

For that reason, we make it clear in our treatment of inference problems (Section 4.3) that conditioning on a deterministic observation $(:=)$ is the fundamental notion. Binary conditioning $(::=)$ is a derived notion that involves further choices, and those choices are not equivalent. Our approach also makes it clear that we should always condition on random variables directly, and not on (Boolean) predicates: by presenting conditioning as an algebraic effect, the expressions $(s ::= t) : I$ and $(s == t) : \text{bool}$ have a different formal status and can no longer be confused.

7.2 Contextual Equivalence for Exact Conditioning Languages

In this article, we have emphasized the role of program equations for manipulating probabilistic programs and based the *Cond* construction on an analysis of contextual equivalence of straight-line inference (Section 6).

While we have fully characterized the case of finite probability (Section 6.2), a corresponding explicit characterization of contextual equivalence for the Gaussian language is still outstanding. We have given partial results in that direction (see [49]) in the form of a sound equational theory for contextual equivalence. The classification of effects $n \rightarrow 0$ in *Cond*(Gauss) is not straightforward:

it is not true that every effect is observing from a unique distribution $0 \rightarrow n$, as, for example, $(\vdash) : 2 \rightarrow 0$ is not of that form. We believe that by passing to an extension category of Gaussians $\text{Gauss} \rightarrow \text{GaussEx}$, we can obtain the desired duality and achieve a more explicit characterization.

It is a further challenge to find semantics for exact conditioning with branching. Automatic normalization is no longer valid here (Section 6.3), and the subtleties of [25] have to be accounted for. Mathematically, this would be an extension of the Cond construction that produces a distributive Freyd category [38]. An example of a categorical model of the Beta-Bernoulli process with branching (but no first-class conditioning) is in [46].

7.3 Other Directions

Categorical Tools. Once a foundation is in algebraic or categorical form, it is easy to make connections to and draw inspiration from a variety of other works: the Obs construction (Definition 5.1) that we considered here is reminiscent of lenses [7] and the Oles construction [20]. These have recently been applied to probability theory [43], quantum theory [22], and reversible computing [21]. The details and intuitions are different, but a deeper connection or generalization may be profitable in the future.

Probabilistic Logic Programming. The concept of exact conditioning is reminiscent of unification in Prolog-style logic programming. Our presentation in [49] is partly inspired by the algebraic presentation of predicate logic of [44], which has a similar signature and axioms. Logic programming is also closely related to relational programming, and we note that our laws for conditioning are reminiscent of graphical presentations of categories of linear relations [1, 3, 4].

PROBLOG [8] supports both logic variables and random variables within a common formalism. We have not considered logic variables in conjunction with the Gaussian language, but a challenge for future work is to bring the ideas of exact conditioning closer to the ideas of unification, both practically and in terms of the semantics. This is again related to the extension GaussEx by “improper priors,” which are a unit for the conditioning product in the same way uniform distributions are in finite probability (Proposition 6.7). The connections with logic programming are spelled out in more detail in [48, Section 20.2].

Implementation. The purpose of our Gaussian language was to give a minimalistic calculus in which to study the novel effect of conditioning in isolation. The close fit of the denotational semantics to the language was thus expected and can be seen as an instance of letting semantics inspire language design. To extend our calculus to a full-blown programming language, one can make use of the general framework of algebraic effects to combine conditioning with other effects like memory or recursion. For example, we can treat higher-order functions by modeling the language on a presheaf category, which is Cartesian closed. The operational semantics easily extend to a full language, for which we have given implementations in Python and F# [47].

A APPENDIX

A.1 Overview of Measure Theory

In order to formalize probability distributions with uncountable support such as Gaussians, one traditionally employs measure theory. While measure theory is not central to the understanding of this article, it is needed to make certain arguments and intuitions rigorous, so we provide a short reference and refer to [28] for a comprehensive introduction.

Measurable Spaces. A σ -algebra on a set X is a collection of subsets of X that contains \emptyset and is closed under complementation and countable union. A *measurable space* is a pair (X, Σ_X) of a set X and a σ -algebra Σ_X on X . The subsets $U \in \Sigma_X$ are called *measurable subsets*. A function

$f : X \rightarrow Y$ between measurable spaces is called *measurable* if for all measurable subsets $A \subseteq Y$, the preimage $f^{-1}(A)$ is measurable in X . The product space $X \times Y$ is naturally equipped with the product- σ -algebra, which is generated by the rectangles $A \times B$ for $A \in \Sigma_X$, $B \in \Sigma_Y$.

Every topological space X comes with a natural σ -algebra, namely the *Borel σ -algebra* $\mathcal{B}(X)$ generated by its open sets. Continuous functions between topological spaces become measurable under this definition.

All examples of measurable spaces in this article will be of a particularly well-behaved type, namely *standard Borel spaces*.

Definition A.1. A *Polish space* is a topological space homeomorphic to a complete metric space with a countable dense subset. A *standard Borel space* is a measurable space that is isomorphic to $(X, \mathcal{B}(X))$ for some Polish space X .

We write \mathbf{Sbs} for the category of standard Borel spaces and measurable maps. This category includes all spaces that will be relevant for real-valued probability, for example, \mathbb{R}^n , the interval $[0, 1]$, and countable discrete spaces.

Definition A.2. Let X be a measurable space. A *measure* on X is a function $\mu : \Sigma_X \rightarrow [0, \infty]$ such that

$$\mu(\emptyset) = 0 \quad \text{and} \quad \mu\left(\sum_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \mu(A_i), \quad (33)$$

where $\sum_i A_i$ denotes disjoint union.

A *probability measure* is a measure satisfying $\mu(X) = 1$. If μ is a measure on X and $f : X \rightarrow Y$ is measurable, the *pushforward measure* $f_*\mu$ is defined by $f_*\mu(A) = \mu(f^{-1}(A))$. For $x \in X$, the *Dirac measure* δ_x is the probability measure on X defined, using Iverson bracket notation, by $\delta_x(A) = [x \in A]$. The Borel-Lebesgue measure is the unique measure on $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$ assigning every interval its length; that is, $\ell([a, b]) = b - a$ for all $a \leq b$. For two probability measures μ, ν , the product probability measure $\mu \otimes \nu$ is uniquely defined via $(\mu \otimes \nu)(A \times B) = \mu(A) \cdot \nu(B)$ for all measurable A, B . A subset that is assigned measure zero can be seen as negligible; this informs the following terminology:

Definition A.3. If μ is a measure on X and $\phi(x)$ some measurable property, we say ϕ holds μ -almost everywhere if $\mu(\{x \in X : \neg\phi(x)\}) = 0$. We say μ is *absolutely continuous* with respect to ν , written $\mu \ll \nu$, if for all measurable sets A , $\nu(A) = 0$ implies $\mu(A) = 0$.

Integration. For a non-negative measurable function $f : X \rightarrow [0, \infty)$ and a measure μ , its integral is defined as

$$\int_X f(x) \mu(dx) \stackrel{\text{def}}{=} \sup_{\{A_i\}} \sum_i \mu(U_i) \cdot \inf_{x \in U_i} f(x) \quad \in \quad [0, \infty],$$

where the supremum ranges over finite measurable partitions of X . The integral extends to measurable functions $f : X \rightarrow \mathbb{R}$ that are integrable, i.e., satisfy $\int |f(x)| \mu(dx) < \infty$. Fubini's theorem states that the order of integration can be interchanged; for all probability measures μ, ν and integrable $f : X \times Y \rightarrow \mathbb{R}$, we have

$$\int \int f(x, y) \mu(dx) \nu(dy) = \int \int f(x, y) \nu(dy) \mu(dx). \quad (34)$$

Density functions are important practical tools for defining measures: if μ is a measure on X and $f : X \rightarrow [0, \infty)$ is measurable, then

$$\nu(A) = \int_A f(x) \mu(dx) \stackrel{\text{def}}{=} \int f(x) [x \in A] \mu(dx)$$

defines another measure. For example, the standard normal distribution on \mathbb{R} is defined as having the density function φ with respect to the Lebesgue measure, where

$$\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

Definition A.4. A probability kernel $X \rightsquigarrow Y$ between measurable spaces is a function $f : X \times \Sigma_Y \rightarrow [0, 1]$ such that $f(-, A)$ is measurable for all $A \in \Sigma_Y$ and $f(x, -)$ is a probability measure for all $x \in X$. Kernels $f : X \rightsquigarrow Y$ and $g : Y \rightsquigarrow Z$ compose using integration

$$(g \bullet f)(x, A) = \int_Y g(y, A) f(x, dy) \quad (35)$$

and any measurable map $f : X \rightarrow Y$ induces a Dirac kernel $\delta_f : X \rightsquigarrow Y$ via $\delta_f(x, A) = [f(x) \in A]$.

Definition A.5 ([11]). The category **BorelStoch** consists of standard Borel spaces and probability kernels between them. Identities are given by δ_{id_X} and composition is kernel composition.

BorelStoch has the structure of a Markov category, whose tensor is given by $X \otimes Y = X \times Y$ on objects and by product measures on morphisms. Copying and discarding are given by the Dirac kernels for the canonical maps $\Delta : X \rightarrow X \times X$, $! : X \rightarrow 1$ in **Sbs**.

Our example Markov categories **FinStoch** and **Gauss** faithfully embed in **BorelStoch**, where we interpret

- (1) the finite set X as the discrete standard Borel space $(X, \mathcal{P}(X))$,
- (2) the object n of **Gauss** as $(\mathbb{R}^n, \mathcal{B}(\mathbb{R}^n))$,

and the morphisms of these categories as actual probability kernels.

Definition A.6 (Giry Monad). There is a monad $\mathcal{G} : \mathbf{Sbs} \rightarrow \mathbf{Sbs}$ due to Giry [16] that assigns to X the space of probability measures $\mathcal{G}X$, endowed with the least σ -algebra, making all evaluations $\text{ev}_A : \mathcal{G}X \rightarrow [0, 1], \mu \mapsto \mu(A)$ measurable for $A \in \Sigma_X$. The unit of this monad takes the Dirac measure $x \mapsto \delta_x$. Kleisli composition takes the average measure via integration; that is, for $f : X \rightarrow \mathcal{G}Y$ and $\mu \in \mathcal{G}X$, we have the Kleisli extension

$$f^\dagger(\mu)(A) = \int_X f(x)(A) \mu(dx). \quad (36)$$

For $h : X \rightarrow Y$, the functorial action $\mathcal{G}(h)(\mu)$ is given by the pushforward measure $h_*\mu$.

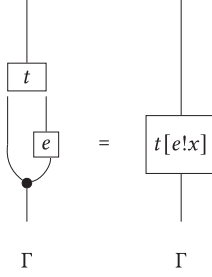
The Giry monad is strong, affine, and commutative, where commutativity follows from Fubini's theorem. A Kleisli arrow $X \rightarrow \mathcal{G}Y$ is the same as a Markov kernel $X \rightsquigarrow Y$, and Kleisli composition (Equation (36)) agrees with kernel composition (Equation (35)).

A.2 CD Calculus

We verify that the remaining axioms of the CD calculus hold in any CD model, completing the proof from Section 3.5.

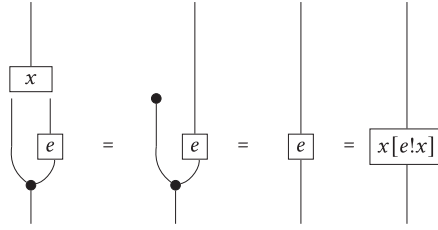
PROOF OF PROPOSITION 3.14. (**let.ξ**) follows from the compositionality of the semantics. (***,β**), (***,η**), and (**unit.η**) are immediate. We proceed to prove that (**let.lin**) is valid; that is, if t uses x

exactly once, then

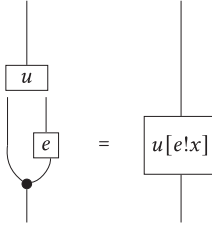


We argue by induction over the term structure of t .

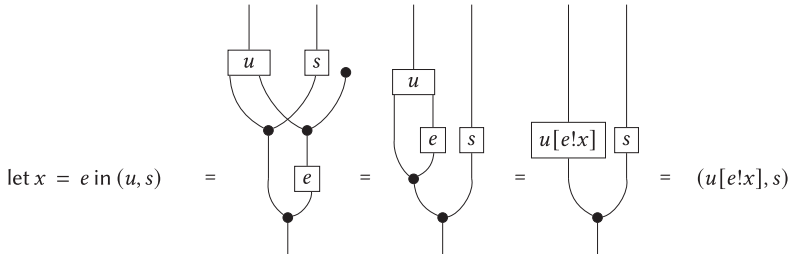
Variable. If $t = x$, then



Pairing. Let $t = (u, s)$, where wlog x occurs freely exactly once in u and zero times in s . By inductive hypothesis, we have $(\text{let } x = e \text{ in } u) = u[e!x]$, i.e.,



from which we derive by the comonoid laws and weakening of s



The case for (s, u) is symmetric.

Function Application. If $t = f u$, we obtain immediately from the inductive hypothesis

$$\text{let } x = e \text{ in } f u \quad = \quad \begin{array}{c} \text{f} \\ \text{u} \\ \text{e} \end{array} \quad = \quad \begin{array}{c} \text{f} \\ \text{u}[e!x] \end{array} \quad = \quad (f u)[e!x]$$

The proof for the projection case $t = \pi_i u$ is analogous.

Let-binding I. Let $t = (\text{let } y = u \text{ in } s)$ with u, s as before; then

$$(\text{let } x = e \text{ in let } y = u \text{ in } s) \equiv (\text{let } y = (\text{let } x = e \text{ in } u) \text{ in } s) \equiv (\text{let } y = u[e!x] \text{ in } s)$$

is a special case of ([assoc](#)), which was proved in Equation (11).

Let-binding II. Let $t = (\text{let } y = s \text{ in } u)$; then

$$(\text{let } x = e \text{ in let } y = s \text{ in } u) \equiv (\text{let } y = s \text{ in let } x = e \text{ in } u) \equiv (\text{let } y = s \text{ in } u[e!x])$$

is a special case of ([comm](#)). The inductive hypothesis on u involves both weakening and exchange and reads

$$u[e!x] \quad = \quad \begin{array}{c} \text{u} \\ \text{e} \end{array}$$

from which we derive

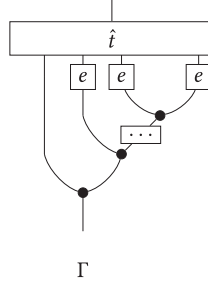
$$\text{let } x = e \text{ in let } y = s \text{ in } u \quad = \quad \begin{array}{c} \text{u} \\ \text{s} \\ \text{e} \end{array} \quad = \quad \begin{array}{c} \text{u} \\ \text{e} \\ \text{s} \end{array} \quad = \quad \text{let } y = s \text{ in } u[e!x]$$

This finishes the validation for linear substitution.

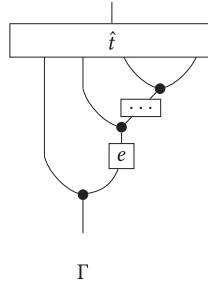
For ([let.val](#)), we carry out the semantic analog of Proposition 3.17 and consider sequences of let-bindings:

$$\text{let } x_1 = e \text{ in } \cdots \text{let } x_n = e \text{ in } \hat{t},$$

whose denotation is



This can be replaced by $\text{let } x = e \text{ in let } x_1 = x \text{ in } \dots \text{let } x_n = x \text{ in } \hat{t}$, i.e.,



whenever the denotation of e is deterministic in the CD category sense. It remains to note that the denotations of all values of the CD calculus are always deterministic. \square

PROOF. The first case is (let.lin) and the last case follows from the combination of the previous ones.

Copyable. We begin with the special case that t has precisely two occurrences of x . Then

$$\begin{aligned}
 & t[e/x] \\
 & \stackrel{(13)}{\equiv} \text{let } x_1 = e \text{ in let } x_2 = e \text{ in } \hat{t} \\
 & \stackrel{(let.val),(*,\beta)}{\equiv} \text{let } p = (e, e) \text{ in let } x_1 = \pi_1 p \text{ in let } x_2 = \pi_2 p \text{ in } \hat{t} \\
 & \stackrel{(14)}{\equiv} \text{let } p = (\text{let } x = e \text{ in } (x, x)) \text{ in let } x_1 = \pi_1 p \text{ in let } x_2 = \pi_2 p \text{ in } \hat{t} \\
 & \stackrel{(assoc)}{\equiv} \text{let } x = e \text{ in let } p = (x, x) \text{ in let } x_1 = \pi_1 p \text{ in let } x_2 = \pi_2 p \text{ in } \hat{t} \\
 & \stackrel{(let.val),(*,\beta)}{\equiv} \text{let } x = e \text{ in let } x_1 = x \text{ in let } x_2 = x \text{ in } \hat{t} \\
 & \stackrel{(13)}{\equiv} \text{let } x = e \text{ in } t.
 \end{aligned}$$

Repeating this process, any chain of repeated let bindings of a copyable term e

$$\text{let } x_1 = e \text{ in } \dots \text{let } x_n = e \text{ in } \dots$$

can be replaced by

$$\text{let } x = e \text{ in let } x_1 = x \text{ in } \dots \text{let } x_n = x \text{ in } \dots$$

Discardable. Let t have no free occurrence of x and e be discardable. Then

$$\begin{aligned}
 & \text{let } x = e \text{ in } t \\
 \stackrel{(\text{let.val})}{\equiv} & \text{let } x = e \text{ in let } y = () \text{ in } t \\
 \stackrel{(\text{assoc})}{\equiv} & \text{let } y = (\text{let } x = e \text{ in } ()) \text{ in } t \\
 \stackrel{(15)}{\equiv} & \text{let } y = () \text{ in } t \\
 \stackrel{(\text{let.val})}{\equiv} & t.
 \end{aligned}$$

□

PROOF OF PROPOSITION 3.15. We will invoke $(\text{let.}\xi)$ implicitly throughout. $(\text{let.}\beta)$ follows immediately from (let.val) because x_2 is a value. (id) , (let.f) , and (let.*) follow by applying (let.lin) one or two times.

For (comm) , we notice that because $x_2 \notin \text{fv}(e_1)$, the expression $\text{let } x_1 = e_2 \text{ in let } x_2 = x_2 \text{ in } e$ has a unique free occurrence of x_2 ; hence, by linear substitution:

$$\begin{aligned}
 & \text{let } x_2 = e_2 \text{ in let } x_1 = e_1 \text{ in } e \\
 \stackrel{(\text{let.}\beta)}{\equiv} & \text{let } x_2 = e_2 \text{ in let } x_1 = e_1 \text{ in let } x_2 = x_2 \text{ in } e \\
 \stackrel{(\text{let.lin})}{\equiv} & \text{let } x_1 = e_1 \text{ in let } x_2 = e_2 \text{ in } e.
 \end{aligned}$$

For (assoc) , if $x_1 \notin \text{fv}(e)$, then $\text{let } x_2 = (\text{let } x_1 = x_1 \text{ in } e_2) \text{ in } e$ has a unique free occurrence of x_1 ; hence,

$$\begin{aligned}
 & \text{let } x_1 = e_1 \text{ in let } x_2 = e_2 \text{ in } e \\
 \stackrel{(\text{let.}\beta)}{\equiv} & \text{let } x_1 = e_1 \text{ in let } x_2 = (\text{let } x_1 = x_1 \text{ in } e_2) \text{ in } e \\
 \stackrel{(\text{let.lin})}{\equiv} & \text{let } x_2 = (\text{let } x_1 = e_1 \text{ in } e_2) \text{ in } e.
 \end{aligned}$$

□

ACKNOWLEDGMENTS

One starting point for this work was a question from Ohad Kammar, and we thank him for that. It has been helpful to discuss developments in this work with many people over the years, and we are also grateful to our anonymous reviewers for helpful suggestions.

REFERENCES

- [1] John C. Baez and Jason Erbele. 2015. Categories in control. *Theory Appl. Categ.* 30 (2015), 836–881. arXiv:1405.6881 [math.CT]
- [2] Vladimir I. Bogachev and Ilya I. Malofeev. 2020. Kantorovich problems and conditional measures depending on a parameter. *J. Math. Anal. Appl.* (2020).
- [3] Filippo Bonchi, Robin Piedeleu, Pawel Sobocinski, and Fabio Zanasi. 2019. Graphical affine algebra. *34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*, Vancouver, BC, Canada, 1–12. DOI: [10.1109/LICS.2019.8785877](https://doi.org/10.1109/LICS.2019.8785877)
- [4] Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. 2017. The calculus of signal flow diagrams I: Linear relations on streams. *Information and Computation* 252 (2017), 2–29.
- [5] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017).
- [6] Kenta Cho and B. Jacobs. 2019. Disintegration and Bayesian inversion via string diagrams. *Mathematical Structures in Computer Science* 29 (2019), 938–971.

- [7] Bryce Clarke, Derek Elkins, Jeremy Gibbons, Fosco Loregian, Bartosz Milewski, Emily Pillmore, and Mario Roman. 2020. Profunctor Optics, a Categorical Update. (2020). arxiv:2001.07488.
- [8] L. De Raedt and A. Kimming. 2015. Probabilistic (logic) programming concepts. *Machine Learning* 100 (2015), 5–47.
- [9] Morris L. Eaton. 2007. Multivariate statistics: A vector space approach. *Institute of Mathematical Statistics Lecture Notes - Monograph Series* 53 (2007), 512.
- [10] Brendan Fong. 2012. *Causal Theories: A Categorical Perspective on Bayesian Networks*. Ph.D. Dissertation. University of Oxford.
- [11] T. Fritz. 2020. A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics. *Advances in Mathematics* 370 (2020).
- [12] T. Fritz, Tomáš Gonda, P. Perrone, and Eigil Fjeldgren Rischel. 2020. Representable Markov Categories and Comparison of Statistical Experiments in Categorical Probability, 107–239. <https://arxiv.org/abs/2010.07416>
- [13] Tobias Fritz and Eigil Fjeldgren Rischel. 2020. Infinite products and zero-one laws in categorical probability. *Compositionality* 2, 3 (Aug. 2020). <https://doi.org/10.32408/compositionality-2-3>
- [14] Carsten Führmann. 2002. Varieties of effects. In *Foundations of Software Science and Computation Structures: 5th International Conference Proceedings*. 144–159.
- [15] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *Computer Aided Verification 2016*.
- [16] Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis. Lecture Notes in Mathematics*, 68–85.
- [17] Peter V. Golubtsov. 2002. Monoidal Kleisli category as a background for information transformers theory. *Information Theory and Information Processing* 2, 1 (2002), 62–84.
- [18] Noah D. Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. Retrieved August 3, 2021, from <http://dippl.org>
- [19] Noah D. Goodman and Joshua B. Tenenbaum. 2016. Probabilistic Models of Cognition. Retrieved March 26, 2021, from <http://probbmods.org>
- [20] C. Hermida and R. D. Tennent. 2012. Monoidal indeterminates and categories of possible worlds. *Theoretical Computer Science* 430 (2012), 3–22.
- [21] Chris Heunen and Robin Kaarsgaard. 2021. Bennett and Stinespring, together at last. In *Proceedings of the 18th International Conference on Quantum Physics and Logic (QPL '21) (Electronic Proceedings in Theoretical Computer Science, 343)*. EPTCS, 102–118. <https://doi.org/10.4204/EPTCS.343>
- [22] Mathieu Huot and Sam Staton. 2018. Universal properties in quantum theory. In *Proceedings of the 15th International Conference on Quantum Physics and Logic 2018*.
- [23] Infernet [n.d.]. Infer.NET Tutorial 3: Learning a Gaussian. <https://dotnet.github.io/infer/userguide/Learning%20a%20Gaussian%20tutorial.html>
- [24] B. Jacobs. 2020. A channel-based perspective on conjugate priors. *Mathematical Structures in Computer Science* 30, 1 (2020), 44–61. <https://doi.org/10.1017/S0960129519000082>
- [25] Jules Jacobs. 2021. Paradoxes of probabilistic programming: And how to condition on events of measure zero with infinitesimal probabilities. In *Proceedings of the ACM on Programming Languages* 5. (POPL '21), 1–26.
- [26] E. T. Jaynes. 2003. *Probability Theory: The Logic of Science*. CUP.
- [27] André Joyal and Ross Street. 1991. The geometry of tensor calculus, I. *Advances in Mathematics* 88 (1991), 55–112.
- [28] Olav Kallenberg. 1997. *Foundations of Modern Probability*. Springer, New York.
- [29] Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 349–360.
- [30] Steffen Lauritzen and Frank Jensen. 1999. Stable local computation with conditional Gaussian distributions. *Statistics and Computing* 11 (2001), 191–203. <https://doi.org/10.1023/A:1008935617754>
- [31] Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210. [https://doi.org/10.1016/S0890-5401\(03\)00088-9](https://doi.org/10.1016/S0890-5401(03)00088-9)
- [32] Saunders MacLane. 1971. *Categories for the Working Mathematician*. Springer-Verlag. ix+262 pages. Graduate Texts in Mathematics, Vol. 5.
- [33] T. Minka, J. M. Winn, J. P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. 2018. Infer.NET 0.3. Microsoft Research, Cambridge. <http://dotnet.github.io/infer>.
- [34] E. Moggi. 1989. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science*. IEEE Press, 14–23.
- [35] Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. [https://doi.org/10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4) Selections from 1989 IEEE Symposium on Logic in Computer Science.
- [36] Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas Schön. 2018. Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. In *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics*. 1037–1046.

- [37] Praveen Narayanan and Chung-chieh Shan. 2019. Applications of a disintegration transformation. In *Workshop on Program Transformations for Machine Learning*.
- [38] John Power. 2006. Generic models for computational effects. *Theoretical Computer Science* 364, 2 (2006), 254–269. <https://doi.org/10.1016/j.tcs.2006.08.006>
- [39] J. Power and E. Robinson. 1997. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science* 7 (1997), 453–468.
- [40] Michael A. Prosschan and Brett Presnell. 1998. Expect the unexpected from conditional expectation. *The American Statistician* 52, 3 (1998), 248–252.
- [41] P. Selinger. 2011. *A Survey of Graphical Languages for Monoidal Categories*. Springer, Berlin, 289–355. https://doi.org/10.1007/978-3-642-12821-9_4
- [42] Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian inference by symbolic disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 130–144.
- [43] Toby St Clere Smithe. 2020. Bayesian Updates Compose Optically. arXiv:2006.01631 [math.CT] <https://arxiv.org/abs/2006.01631>
- [44] Sam Staton. 2013. An algebraic presentation of predicate logic. In *Foundations of Software Science and Computation Structures*, Frank Pfenning (Ed.). Springer, Berlin, 401–417.
- [45] Sam Staton. 2017. Commutative semantics for probabilistic programming. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer, Berlin, 855–879.
- [46] Sam Staton, Dario Stein, Hongseok Yang, Nathanael Ackerman, Cameron Freer, and Daniel Roy. 2018. The Beta-Bernoulli process and algebraic effects. In *Proceedings of 45th International Colloquium on Automata, Languages and Programming (ICALP '18)*. <http://arxiv.org/abs/1802.09598>
- [47] Dario Stein. 2021. GaussianInfer. <https://github.com/damast93/GaussianInfer>
- [48] Dario Stein. 2021. *Structural Foundations for Probabilistic Programming Languages*. Ph. D. Dissertation. University of Oxford.
- [49] Dario Stein and Sam Staton. 2021. *Compositional Semantics for Probabilistic Programs with Exact Conditioning*. Association for Computing Machinery, New York, NY.
- [50] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. arXiv:1809.10756 [stat.ML]
- [51] Rajan Walia, Praveen Narayanan, Jacques Carette, Sam Tobin-Hochstadt, and Chung-chieh Shan. 2019. From high-level inference algorithms to efficient code. In *Proceedings of the ACM on Programming Languages* 3 (ICFP'19), 1–30.
- [52] Zhang Z. 2005. *The Schur Complement and Its Applications*. <https://doi.org/10.1007/b105056>

Received 12 July 2022; revised 6 October 2023; accepted 20 October 2023