Probabilistic Programming with Exact Conditions

DARIO STEIN, Radboud University Nijmegen, The Netherlands
SAM STATON, University of Oxford, United Kingdom

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.

1 INTRODUCTION

Probabilistic programming is a programming paradigm that uses code to formulate generative statistical models and perform inference on them [16, 45]. Techniques from programming language theory can be used to understand modelling assumptions such as conditional independence, as well as enable optimizations that improve the efficiency of various inference algorithms (e.g. [40]). We'll also elaborate this 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 $E_1 =:= E_2$ 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):

```
x = normal(μ=50, σ=10) # prior
y = normal(μ=x, σ=5)   # noisy measurement
y =:= 40               # make exact observation
```

Variants of exact conditioning are available in different frameworks: In Hakaru [37], certain exact conditioning queries can be addressed using symbolic disintegration, but (=:=) is not a first-class construct in Hakaru. Infer.NET [29] does allow exact conditioning on variables and employs an approximate inference algorithm to solve the resulting queries (e.g. [20]). 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 which allow us to simplify programs compositionally. Among the desired properties are the following: Program lines can be reordered as long as

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

dataflow is respected. That is, the *commutativity equation* remains valid for programs with conditioning

$$\begin{array}{l}\mathsf{let}\, x \,=\, u \,\mathsf{in} \\ \mathsf{let}\, y \,=\, v \,\mathsf{in}\, t\end{array} \quad \equiv \quad \begin{array}{l}\mathsf{let}\, y \,=\, v \,\mathsf{in} \\ \mathsf{let}\, x \,=\, u \,\mathsf{in}\, t\end{array} \tag{1}$$

where x not free in v and y 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.

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

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

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

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 particular 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.

Our goal is then to give denotational semantics to these languages, and prove that the desired properties (1),(2),(3) hold. We do this in an extensible way that is agnostic to the particular fragment of probability we use. *Categorical probability theory* is an abstract language to discuss notions such as determinism, independence and conditioning, which we use to give an abstract formulation of *inference problems* in Section 4. We prove a correspondence between a class of first-order probabilistic programming languages and mathematical structures called Markov and CD- categories (Section 3.1), which gives us a canonical route to denotational semantics. At the heart of this correspondence is the CD-calculus, which is the internal language of CD categories (Section 3.2).

The central contribution of this article is the Cond construction (Section 5), which extends a suitable Markov category $\mathbb{C}$ with effects $X \longrightarrow I$ for exact observations. Importantly, the Cond construction makes no mention of measure theory, densities or limits, which usually feature in discussions of conditional probability. It is instead 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

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

The desired properties of exact conditioning can be proved purely abstractly for the Cond construction. Our type-theoretic approach to conditioning will also help addressing counterintuitive behavior such as Borel's paradox (Section 7.1).

Lastly, 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). We too prove

the denotational semantics for the Gaussian language fully abstract in Section 6.1. An explicit characterization of Cond(Gauss) is future work though we gave a sound algebraic theory in [44].

Outline: To demonstrate the strengths and intricacies of the exact conditioning approach, this introduction proceeds with an extended discussion (Section 1.1) elaborating the noisy measurement example, and showcasing the power of program transformation using Gaussian random walks.

In Section 2, we formally introduce the Gaussian language and its operational semantics.

In Section 3, we review categorical probability theory using CD- and Markov categories, and introduce the CD-calculus (Section 3.1), which is the internal language of CD categories. This gives us a canonical way of understanding such categories as the semantic domains of first-order probabilistic programs. 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 points for this article are our paper in the Proceedings of LICS 2021 [44] and the first author's DPhil thesis [43]. We expose the Cond construction in a detailed and self-contained manner with emphasis on program equations and contextual equivalence. The characterizations in Section 6 and the recognition of the special role of branching in Section 6.3 are novel. A Python implementation of the Gaussian Language is available under [42].

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 paper. The later sections of the article are fully formal.

Exact Conditioning versus Scoring: In the introduction, we considered an 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* $\mathtt{r} \in \mathbb{R}_{>0}$. A derived operation `observe(x,D)` expresses an observation of a value x from some distribution D by scoring with the density $r = \mathrm{pdf}_D(x)$. 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 [4] or WebPPL [15]. 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 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 1). 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 (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]
```

In the observation branch, we can now use initialization principle (3) to set `y[i]` to its target value directly as `y[i] = obs[i]`. The remaining condition becomes `(y[i] − y[i−1]) =:= 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.

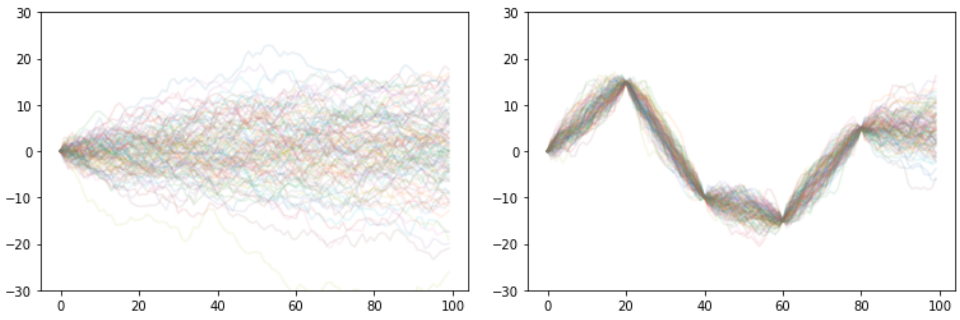


Fig. 1. Gaussian random walk (left) and conditioned posterior (right) with four exact observations at $i = 20, 40, 60, 80$

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 1, 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, composition of programs is often complicated and local transformations are difficult to reason about. This is what makes programs 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 essentially maintains a record of exact conditions 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 paper.

Looking beyond this section, the aim of the future sections of this paper is to address that 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 paper, 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 (Section 3.2 and Section 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. [26]). 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 $Z_1, \ldots, Z_m \sim \mathcal{N}(0, 1)$ are independent and standard normally distributed, i.e. with the following density function with respect to the Lebesgue measure:

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

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 $S = \mu + \mathrm{col}(\Sigma)$ where $\mathrm{col}(\Sigma)$ denotes the column space of Σ. We call S the *support* of the distribution.

This defines a small convenient fragment of probability theory.

- 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 $A\Sigma A^T$.
- Furthermore, 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. [8, 3.13]) for the conditional distribution $X_1|(X_2 = a)$ of X_1 conditional on $X_2 = a$ for $a \in \operatorname{supp}(X_2)$. Namely $X_1|(X_2 = a) \sim \mathcal{N}(\mu', \Sigma')$ where

$$\mu' = \mu_1 + \Sigma_{12}\Sigma_{22}^-(a - \mu_2) \quad \Sigma' = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^-\Sigma_{21} \tag{4}$$

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

We elaborate on formula (4) 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 *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 $\operatorname{col}(\Sigma_{22})$. Therefore, formula (4) 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 an detailed discussion of these concepts, we refer to [47, Section 1.6].

The formula for conditional probability becomes particular 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 $u \in \mathbb{R}^{n\times 1}$, then the covariance of (X, Z), regarded as a random $(n+1)$-vector, decomposes as

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

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

$$\mu' = \mu + \frac{a - u\mu}{\sigma_{22}}\Sigma u^T, \quad \Sigma' = \Sigma - \frac{1}{\sigma_{22}}\Sigma u^T u \Sigma \tag{5}$$

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, $a \notin \operatorname{supp}(Z)$, 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 $\mathcal{N}(0, \Sigma)$ with covariance matrix

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

By (5), the conditional distribution of (X, Y) given $Z = 0$ has the following covariance matrix

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

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. [37]). 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 $[X = Y] = 1$, 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 $|x|e^{-x^2}$ [36]. 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 $X, Y \sim \mathcal{N}(0,1), Z = 0$ 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.

$$\tau ::= \mathsf{R} \mid \mathsf{I} \mid \tau * \tau$$

Terms of the language are

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

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

$$\frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \qquad \frac{}{\Gamma \vdash () : \mathsf{I}} \qquad \frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t : \tau}{\Gamma \vdash (s, t) : \sigma * \tau}$$

$$\frac{\Gamma \vdash s : \mathsf{R} \quad \Gamma \vdash t : \mathsf{R}}{\Gamma \vdash s + t : \mathsf{R}} \qquad \frac{\Gamma \vdash t : \mathsf{R}}{\Gamma \vdash \alpha \cdot t : \mathsf{R}} \qquad \frac{}{\Gamma \vdash \underline{\beta} : \mathsf{R}}$$

$$\frac{}{\Gamma \vdash \mathsf{normal}() : \mathsf{R}} \qquad \frac{\Gamma \vdash s : \mathsf{R} \quad \Gamma \vdash t : \mathsf{R}}{\Gamma \vdash (s =:= t) : \mathsf{I}}$$

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \mathsf{let}\, x \;=\; s \,\mathsf{in}\, t : \tau}$$

$$\frac{\Gamma \vdash s : \sigma * \sigma' \quad \Gamma, x : \sigma, y : \sigma' \vdash t : \tau}{\Gamma \vdash \mathsf{let}\, (x, y) \;=\; s \,\mathsf{in}\, t : \tau}$$

In Section 3.2 we will introduce the general CD-calculus, and our Gaussian language is an instance of this[1], with base type R and signature

$$(+) : \mathsf{R} * \mathsf{R} \to \mathsf{R}, \quad \alpha \cdot (-) : \mathsf{R} \to \mathsf{R}, \quad \underline{\beta} : \mathsf{I} \to \mathsf{R}, \quad \mathsf{normal} : \mathsf{I} \to \mathsf{R}, \quad (=:=) : \mathsf{R} * \mathsf{R} \to \mathsf{I} \tag{6}$$

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 $s; t$, identifying the type $\mathsf{R}^n = \mathsf{R} * (\mathsf{R} * \ldots)$ with vectors, and for matrix-vector multiplication $A \cdot \vec{x}$. For $\sigma \in \mathbb{R}$ and $e : \mathsf{R}$, we define $\mathsf{normal}(x, \sigma^2) \equiv x + \sigma \cdot \mathsf{normal}()$. More

[1] in the CD-calculus, we use projection maps rather than pattern-matching let, but those constructs are interdefinable

generally, for a covariance matrix Σ, we write $\mathsf{normal}(\vec{x}, \Sigma) = \vec{x} + A \cdot (\mathsf{normal}(), \ldots, \mathsf{normal}())$ where A is any matrix such that $\Sigma = AA^T$. 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

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

The full example with conditioning can be written

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

This program is contextually equivalent (Definition 2.5) to

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

2.3 Operational Semantics

Informally, our operational semantics works as follows: calling 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 the formula (4).

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

$$(e, \psi)$$

where ψ is a Gaussian distribution on $\mathbb{R}^r$ (i.e. a mean vector and covariance matrix) and $z_1 : \mathsf{R}, \ldots, z_r : \mathsf{R} \vdash e$. 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

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

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

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

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. $\mathsf{let}\ x\ =\ C\ \mathsf{in}\ e$ 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 $(z_1 \ldots z_r)$ 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 normal() allocates a fresh latent variable and adds an independent dimension to the prior

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

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 $\left(\begin{smallmatrix} \Sigma & 0 \\ 0 & 1 \end{smallmatrix}\right)$.

(2) To define conditioning, note that every value $z_1 : \mathsf{R}, \ldots, z_r : \mathsf{R} \vdash v : R$ defines an affine function $\mathbb{R}^r \to \mathbb{R}$. In order to reduce $(v \mathrel{::=} w, \psi)$, 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

$$(v \mathrel{::=} w, \psi) \rhd ((), \psi|_{v=w})$$

Otherwise $(v ::= w, \psi) \rhd \bot$, 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 formula (5).

(3) Let bindings are standard

$$(\mathsf{let}\, x \;=\; v \,\mathsf{in}\, e, \psi) \rhd (e[v/x], \psi)$$
$$(\mathsf{let}\, (x, y) \;=\; (v, w) \,\mathsf{in}\, e, \psi) \rhd (e[v/x, w/y], \psi)$$

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

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

Proof notes. First, the $\rhd$ 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 $\{\mathsf{normal}, ::=, \mathsf{let}\;=\;\mathsf{in}\}$ in an expression. □

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

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

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 paper 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 $(K[e_i], !) \rhd^* (v_i, \psi_i)$ then $(K[e_j], !) \rhd^* (v_j, \psi_j)$ and $(v_i)_*\psi_i = (v_j)_*\psi_j$
(2) when $(K[e_i], !) \rhd^* \bot$ then $(K[e_j], !) \rhd^* \bot$

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 nothing conceptually limits the language in this section to only Gaussians. We are running with this example for concreteness, but any family of distributions which 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 FOUNDATIONS FOR PROBABILISTIC PROGRAMMING

This section introduces a categorical generalization of the probabilistic language in Section 2, but without conditioning. Conditioning will be analyzed categorically in Section 5, following the abstract account of inference problems in Section 4.

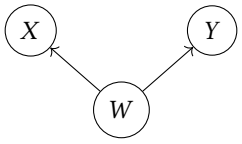
We begin this section with an overview of categorical methods for building new foundations for measure theory, a key method in this paper. Recall that a category comprises objects and morphisms between the objects. In this context, the objects are to be thought of as generalized spaces, and the morphisms as parameterized measures. In particular, our categories will be monoidal, which means we have the following constructions (see e.g. [28] 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 $I \to X$ are thought of as distributions or measures on X, and so the morphisms $Y \to X$ are thought of as measures on X with parameters from Y.
- Categorical composition: for any morphisms $f\colon X \to Y$ and $g\colon Y \to Z$, there is a composite morphism $gf : X \to Z$. This can be understood as a form of integration (integrating over Y).
- Monoidal composition: for any morphisms $f\colon A \to B$ and $g\colon X \to Y$, there is a composite morphism $(f \otimes g) : A \otimes X \to B \otimes Y$. This can be understood as a form of product measure.

Monoidal categories are an important general concept, but to discuss measures and probabilities in this axiomatic way, it is appropriate to impose further conditions, resulting in copy-delete categories and Markov categories as we discuss in Section 3.1.

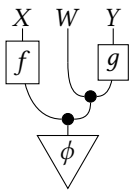
The notation for composing morphisms (with $\circ$ and $\otimes$) can be cryptic and it is important to find good notations. Here we focus on two notations: string diagrams (e.g. [24]) and a new probabilistic programming language called CD-calculus (Section 3.2). In the string diagram, the objects of the category become wires and morphisms are boxes; category composition ($\circ$) is simply joining wires together, and monoidal composition ($\otimes$) is juxtaposition. Even without categorical machinery, string diagrams carry an intuitive meaning as dataflow diagrams. The CD calculus is an instance of an internal language, that is the objects of the category become the types of the CD calculus, morphisms are terms and composition is let-binding.

For illustration, let us consider a graphical model (Bayesian network) of the shape



(7)

meaning that X, Y, W are random variables such that X and Y are *conditionally independent given* W. Conditional independence can be characterized purely abstractly in Markov categories [10, Section 12] by the existence of morphisms $f : W \to X, g : W \to Y$ which generate X, Y conditional on W. Figure 2 contrasts the different formalisms for recovering the joint distribution over $X \otimes (W \otimes Y)$ from the marginal $\phi : I \to W$.



(a) String diagram

$$(f \otimes ((\mathrm{id}_W \otimes g) \circ \mathrm{copy}_W)) \circ \mathrm{copy}_W \circ \phi$$

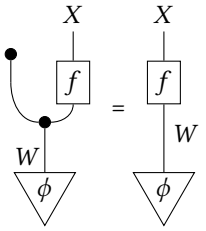
(b) Categorical composition

$$w : W \vdash f : X$$
$$w : W \vdash g : Y$$
$$\vdash \mathsf{let}\ w\ =\ \phi\ \mathsf{in}\ (f, (w, g))$$

(c) CD-calculus

Fig. 2. Different languages for composition

A systematic comparison of graphical models and string diagrams is given in [9]. The three languages, string diagrams, categorical and CD calculus have the same level of expressive power but offer different advantages, so we will frequently use them alongside each other to use the appropriate notation in our discussion of Section 5. In the programming syntax, all copying is conveniently handled in the nonlinear use of variables (Figure 2c). String diagrams let us be precise about the scope of random variables by explicit marginalization, e.g.



In the programming terms, this is usually expressed via nested let

$$(\mathsf{let}\, x \;=\; (\mathsf{let}\, w \;=\; \phi\, \mathsf{in}\, f(w))\, \mathsf{in}\, x) = f(\phi)$$

Both string diagrams and the CD calculus are intuitive and economical languages for reasoning about large composites.

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, and ask whether they are equal there (Section 3.3). 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 3.4.

3.1 Copy-Delete Categories and Markov Categories

We will recall two closely related notions, namely:

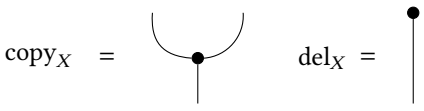
Markov categories to model purely stochastic computation [10] and
CD categories which model potentially *unnormalized* stochastic computation [5].

Markov categories have been used to formalize various theorems of probability and statistics, such as sufficient statistics (Fisher-Neyman, Basu, Bahadur) [10], stochastic dominance (Blackwell-Sherman-Stein) [11] and zero-one laws [12]. 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 (**??**). 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.

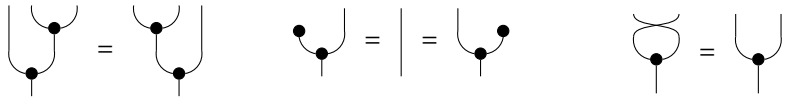
Definition 3.1 (CD category). A *copy-delete category* (CD category) is a symmetric monoidal category $(\mathbb{C}, \otimes, I)$ where every object X is equipped with the structure of a commutative comonoid

$$\mathrm{copy}_X : X \to X \otimes X \quad \mathrm{del}_X : X \to I$$

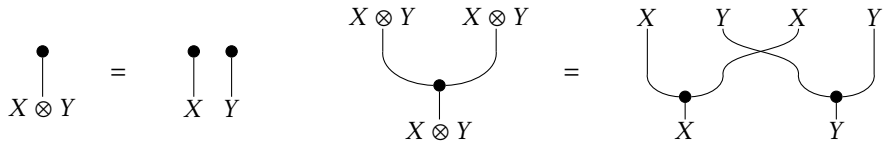
graphically depicted as



satisfying the axioms

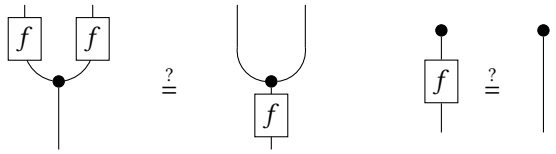


We require that the comonoid structure be compatible with the monoidal structure as follows



Here we are using string diagram notation: the morphisms should be read from bottom to top, vertical composition is the categorical composition of morphisms, and horizontal juxtaposition is monoidal product. For example the last diagram on the right describes a morphism $(X \otimes Y) \to (X \otimes Y \otimes X \otimes Y)$.

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



(8)

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

Definition 3.2 (Copyable and discardable morphism). A morphism $f : X \to Y$ is called *copyable* if the first equation of (8) holds: $\mathrm{copy}_Y \circ f = (f \otimes f) \circ \mathrm{copy}_X$. A morphism $f : X \to Y$ is called *discardable* if the second equation of (8) holds: $\mathrm{del}_Y \circ f = \mathrm{del}_X$.

Definition 3.3 (Markov category). 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. The interchange law of the tensor $\otimes$ encodes exchangeability of stochastic computation (Fubini's theorem), and the discardability condition in Markov categories means that probability measures have total mass 1.

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

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

and define projection maps

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

via discarding. Recall that a *state* in a symmetric monoidal category is a morphism $\psi : I \to X$. An *effect* is a morphism $\rho : X \to I$. Note that by terminality of the unit I in a Markov category, all effects $X \to I$ 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 $\psi : I \to X$ *distributions*, and if $f : A \to X \otimes Y$, we define its *marginal* (of X) $f_X : A \to X$ 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 Markov category $\mathbb{C}$, the wide subcategory $\mathbb{C}_{\text{det}}$, which consists of only the deterministic morphisms, is cartesian.

We now give examples of the relevant categories for this article.

Definition 3.4. The Markov category FinStoch has as objects finite sets X, and morphisms $X \to Y$ 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*

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

To give a morphism in $\mathsf{FinStoch}(X, Y)$ is to give a Kleisli arrow $X \to D(Y)$ for the distribution monad on Set. This is a general receipe for constructing CD- and Markov categories.

Proposition 3.5. *Let $\mathbb{C}$ be a category with finite products and $T : \mathbb{C} \to \mathbb{C}$ be a strong, commutative monad. Then the Kleisli category $\mathsf{Kl}(T)$ is a CD category, which is furthermore Markov if and only if T is* affine*, i.e. $T1 \cong 1$.*

We modify FinStoch as follows to allow for unnormalized computation:

Definition 3.6. The CD category FinSubStoch has as objects finite sets X, and morphisms $X \to Y$ are *subprobability* channels $p(y|x)$, that is $p(y|x) \in [0,1]$ and for all $x \in X$,

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

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

We now define the Markov category which captures Gaussian probability.

Definition 3.7 ([10, §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 $m \to n$ are tuples (A, b, Σ) where $A \in \mathbb{R}^{n \times m}, b \in \mathbb{R}^n$ and $\Sigma \in \mathbb{R}^{n \times n}$ is a positive semidefinite matrix. The tuple represents a stochastic map $f : \mathbb{R}^m \to \mathbb{R}^n$ 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

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

Proposition 3.8. *A morphism (A, b, Σ) in* Gauss *is deterministic iff $\Sigma = 0$, i.e. there is no randomness involved.*

Proof. Write $f = (A, b, \Sigma)$, 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 $\Sigma = 0$. □

It follows that the deterministic subcategory $\mathrm{Gauss}_{\mathrm{det}}$ 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 rather comprehensive Markov category.

Definition 3.9. The Markov category BorelStoch has as objects standard Borel spaces X, and morphisms $X \to Y$ are probability kernels $\Sigma_X \times Y \to [0, 1]$.

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

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

Definition 3.10. 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 $\mathcal{P} : \mathsf{Set} \to \mathsf{Set}$ and the nonempty powerset monad $\mathcal{P}^+ : \mathsf{Set} \to \mathsf{Set}$ respectively. The category Rel^+ is referred to as SetMulti in [10].

3.2 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. [27, 30, 31], but the commutativity of the tensor allow for some convenient simplifications and a concise equational presentation. To this extent it is a novel calculus.

Definition. 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

$$A ::= \tau \mid \mathsf{unit} \mid A * A$$

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

$$t ::= x \mid () \mid (t, t) \mid \pi_i\, t \mid f\, t \mid \mathsf{let}\, x \;=\; t\, \mathsf{in}\, t \qquad (i = 1, 2)$$

subject to the typing rules $x_1 : A_1, \ldots, x_n : A_n \vdash t : B$ given in Figure 3.

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{\Gamma \vdash s : A \quad \Gamma \vdash t : B}{\Gamma \vdash (s, t) : A * B} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash f\, t : B}\ (f : A \to B)$$

$$\frac{\Gamma \vdash t : A_1 * A_2}{\Gamma \vdash \pi_1\, t : A_1} \qquad \frac{\Gamma \vdash t : A_1 * A_2}{\Gamma \vdash \pi_2\, t : A_2} \qquad \frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash e : A}{\Gamma \vdash \mathsf{let}\, x \;=\; e\, \mathsf{in}\, t : B}$$

Fig. 3. Typing rules for the CD calculus

We employ some standard syntactic sugar, for example sequencing

$$s; t \stackrel{\mathrm{def}}{=} \mathsf{let}\, x \;=\; s\, \mathsf{in}\, t \qquad (x \notin \mathsf{fv}(t))$$

We also define a pattern-matching let as syntactic sugar

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

Conversely, we can provably recover the projection constructs from this sugar (in a sense made precise by the equational theory in Section 3.4):

$$(\pi_1\,s)=(\mathsf{let}\,(x,y)\;=\;s\,\mathsf{in}\,x)\qquad(\pi_2\,s)=(\mathsf{let}\,(x,y)\;=\;s\,\mathsf{in}\,y)$$

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

3.3 Semantics

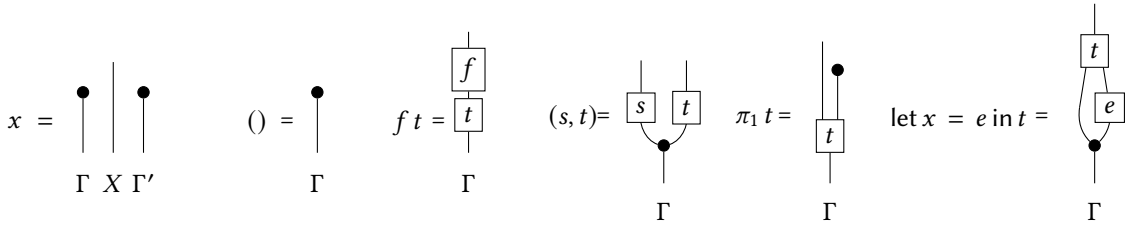
We now explain how the CD calculus can be interpreted in CD categories. Types will be interpreted as objects, and terms interpreted as morphisms. Formally, a *model* of signature (τ,ω) is a CD category $\mathbb{C}$ together with an assignment of objects $[\![A]\!]\in\mathbb{C}$ for each basic type and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

$$[\![\mathsf{unit}]\!]=I\quad[\![A_1*A_2]\!]=[\![A_1]\!]\otimes[\![A_2]\!]\quad[\![A_1,\dots,A_n]\!]=[\![A_1]\!]\otimes(\cdots\otimes[\![A_n]\!])$$

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

- $[\![x]\!]$ is the discarding map $[\![\Gamma,A,\Gamma']\!]\cong[\![\Gamma]\!]\otimes[\![A]\!]\otimes[\![\Gamma']\!]\to I\otimes[\![A]\!]\otimes I\cong[\![A]\!]$
- $[\![()]\!]$ is the discarding map $\mathrm{del}_{[\![\Gamma]\!]}:[\![\Gamma]\!]\to I$
- $[\![(s,t)]\!]$ is the map $[\![\Gamma]\!]\xrightarrow{\mathrm{copy}_{[\![\Gamma]\!]}}[\![\Gamma]\!]\otimes[\![\Gamma]\!]\xrightarrow{[\![s]\!]\otimes[\![t]\!]}[\![A]\!]\otimes[\![B]\!]=[\![A*B]\!]$
- $[\![\pi_i t]\!]$ is marginalization $[\![\Gamma]\!]\xrightarrow{[\![t]\!]}[\![A_1]\!]\otimes[\![A_2]\!]\to[\![A_i]\!]$
- $[\![f\,t]\!]$ is the composite $[\![\Gamma]\!]\xrightarrow{[\![t]\!]}[\![A]\!]\xrightarrow{[\![f]\!]}[\![B]\!]$
- $[\![\mathsf{let}\,x\;=\;e\,\mathsf{in}\,t]\!]$ is given by $[\![\Gamma]\!]\xrightarrow{\mathrm{copy}_{[\![\Gamma]\!]}}[\![\Gamma]\!]\otimes[\![\Gamma]\!]\xrightarrow{\mathrm{id}_{[\![\Gamma]\!]}\otimes[\![e]\!]}[\![\Gamma]\!]\otimes[\![A]\!]\xrightarrow{[\![t]\!]}[\![B]\!]$

The semantics can be seen as a procedure for translating every term of the CD calculus into a string diagram as follows (where we omit $[\![-]\!]$ for readability).



Proposition 3.11 (Structural rules). *Weakening and exchange correspond to discarding of variables, and swap isomorphisms respectively.*

Proof. Straightforward induction using the comonoid axioms. □

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

$$(+)\mathsf{R}*\mathsf{R}\to\mathsf{R}\qquad\beta\cdot(-):\mathsf{R}\to\mathsf{R}\qquad\underline{\beta}:\mathsf{unit}\to\mathsf{R}\qquad\mathsf{normal}:\mathsf{unit}\to\mathsf{R}$$

The Markov category Gauss models this signature using $[\![\mathsf{R}]\!] = 1$ 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 Cond(Gauss). That category will need to interpret the additional function symbol $(::) : \mathsf{R} * \mathsf{R} \to \mathsf{R}$.

3.4 Equational Theory

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

In call-by-value languages, the substitution

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

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 $(\mathsf{let}\, x \;=\; e\, \mathsf{in}\, u) \equiv u[e/x]$ 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 4. 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.

Congruence laws:

$$\equiv \text{ is reflexive, symmetric and transitive} \qquad \text{(equiv)}$$

$$\frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{(\mathsf{let}\, x \;=\; e_1\, \mathsf{in}\, e_2) \equiv (\mathsf{let}\, x \;=\; e_1'\, \mathsf{in}\, e_2')} \qquad (\mathsf{let}.\xi)$$

A *value expression* is a term of the form

$$V ::= x \mid () \mid (V, V) \mid \pi_i V \mid \mathsf{let}\, x \;=\; V\, \mathsf{in}\, V$$

The axioms of the CD calculus are:

$$(\mathsf{let}\, x \;=\; e\, \mathsf{in}\, t) \equiv t[e!x] \qquad (\mathsf{let.lin})$$

$$(\mathsf{let}\, x \;=\; V\, \mathsf{in}\, t) \equiv t[V/x] \qquad (\mathsf{let.val})$$

$$\pi_i\,(x_1, x_2) \equiv x_i \qquad (*.\beta)$$

$$(\pi_1\, x, \pi_2\, x) \equiv x \qquad (*.\eta)$$

$$x \equiv () \qquad (\mathsf{unit}.\eta)$$

where we write $t[x!e]$ for substituting a unique free occurrence of x. For the internal language of *Markov categories*, extend (let.lin) to all substitutions targeting *at most one* free occurrence of x.

Fig. 4. Axioms of the CD-calculus

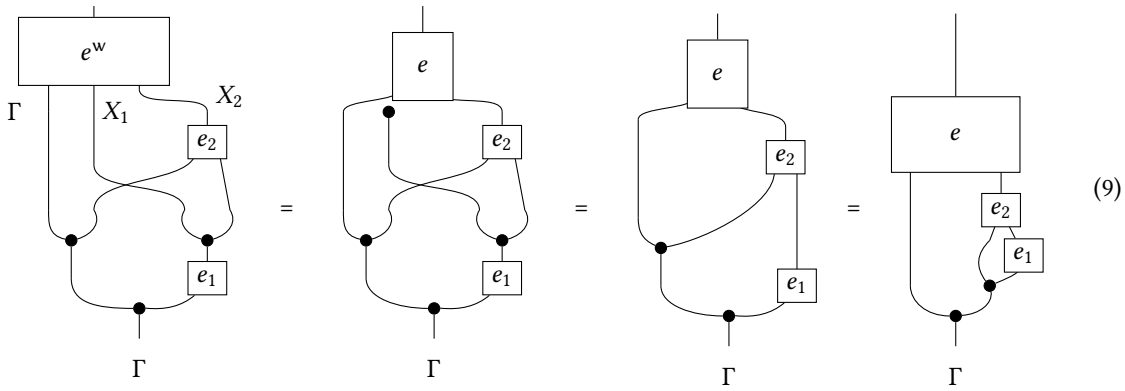
Proposition 3.12 (Soundness). *Every CD model validates the axioms of the CD calculus. That is if* $\Gamma \vdash e_1 \equiv e_2 : A$ *then* $[\![e_1]\!] = [\![e_2]\!] : [\![\Gamma]\!] \to [\![A]\!]$.

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 (Section 8.1). 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

$$[\![(\mathsf{let}\, x_1 \;=\; e_1 \,\mathsf{in}\, \mathsf{let}\, x_2 \;=\; e_2 \,\mathsf{in}\, e^{\mathsf{w}})]\!] \equiv [\![(\mathsf{let}\, x_2 \;=\; (\mathsf{let}\, x_1 \;=\; e_1 \,\mathsf{in}\, e_2) \,\mathsf{in}\, e)]\!]$$

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 3.11. □

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

Proposition 3.13. *All axioms of the ground λ_c-calculus [31, Tables 6,7] and commutativity are derivable.*

$$(\mathsf{let}\, x_2 \;=\; (\mathsf{let}\, x_1 \;=\; e_1 \,\mathsf{in}\, e_2) \,\mathsf{in}\, e) \equiv (\mathsf{let}\, x_1 \;=\; e_1 \,\mathsf{in}\, \mathsf{let}\, x_2 \;=\; e_2 \,\mathsf{in}\, e) \quad x_1 \notin \mathsf{fv}(e) \tag{assoc}$$

$$(\mathsf{let}\, x_1 \;=\; e_1 \,\mathsf{in}\, \mathsf{let}\, x_2 \;=\; e_2 \,\mathsf{in}\, e) \equiv (\mathsf{let}\, x_2 \;=\; e_2 \,\mathsf{in}\, \mathsf{let}\, x_1 \;=\; e_1 \,\mathsf{in}\, e) \quad x_1 \notin \mathsf{fv}(e_2), x_2 \notin \mathsf{fv}(e_1) \tag{comm}$$

$$(\mathsf{let}\, x \;=\; e \,\mathsf{in}\, x) \equiv e \tag{id}$$

$$(\mathsf{let}\, x_1 \;=\; x_2 \,\mathsf{in}\, e) \equiv e[x_2/x_1] \tag{let.β}$$

$$f e \equiv (\mathsf{let}\, x \;=\; e \,\mathsf{in}\, f x) \tag{let.f}$$

$$(s, t) \equiv (\mathsf{let}\, x \;=\; s \,\mathsf{in}\, \mathsf{let}\, y \;=\; t \,\mathsf{in}\, (x, y)) \tag{let.$*$}$$

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

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 $x_1, \ldots, x_n$ (the order does not matter). By repeated application of (let.val), we can derive

$$t \equiv \mathsf{let}\, x_1 \;=\; x \,\mathsf{in}\, \cdots \mathsf{let}\, x_n \;=\; x \,\mathsf{in}\, \hat{t} \tag{10}$$

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

$$t[e/x] \equiv \hat{t}[e!x_1] \cdots [e!x_n] \equiv \mathsf{let}\, x_1 \;=\; e \,\mathsf{in}\, \cdots \mathsf{let}\, x_n \;=\; e \,\mathsf{in}\, \hat{t} \tag{11}$$

This means we can reduce questions about substitution to the copying behavior of the term e. We adapt the definitions from [13, 25].

Definition 3.14. A term e is called *copyable* if

$$(\mathsf{let}\, x \;=\; e \,\mathsf{in}\, (x, x)) \equiv (e, e) \tag{12}$$

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

$$(\mathsf{let}\, x \;=\; e \,\mathsf{in}\, ()) \equiv () \tag{13}$$

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

Proposition 3.15. *The substitution equation*

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

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*

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, also every string diagram can be parsed into a term, and the theory of $\equiv$ proves all ways of reading a diagram equivalent.

Definition 3.16. Fix a CD signature $\mathfrak{S}$. The *syntactic category* Syn has

(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
$$x : A \vdash \mathsf{let}\, x \;=\; e \,\mathsf{in}\, t$$
(5) Tensor on objects is defined as $A \otimes B = A * B$ with unit type unit. The tensor on morphisms of $x_1 : A_1 \vdash s_1 : B_1$, $x_2 : A_2 \vdash s_1 : B_2$ is
$$x : A_1 * A_2 \vdash \mathsf{let}\, x_1 \;=\; \pi_1\, x \,\mathsf{in}\, \mathsf{let}\, x_2 \;=\; \pi_2\, x \,\mathsf{in}\, (s_1, s_2)$$
(6) CD structure is given by nonlinear use of variables, that is
$$\mathrm{copy}_A = x : A \vdash (x, x) : A * A$$
$$\mathrm{del}_A = x : A \vdash () : \mathsf{unit}$$

The verification of the CD category axioms is tedious but standard. Note that we can build on existing work [27] because our axioms prove all equations of the ground fragment of λ_c (Proposition 3.13). We expect that the syntactic category is an initial model over a given signature and the definition of the semantics $[\![-]\!]$ 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. In brief, Markov categories are semicartesian monoidal categories where every object is equipped with a comonoid structure. We now present an abstract theory of *inference problems* in Markov categories with sufficient structure. We begin by recalling categorical rephrasings of core notations from probability: conditional probability (Section 4.1), and almost-sure equality, absolute continuity and support (Section 4.2), mostly from [5, 10]. 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;

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.

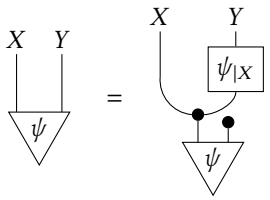
Looking forward, we will develop a notion of *open inference problem* in Section 5 as part of a compositional framework for collecting conditions, so as to give a compositional semantics to the kind of programming with conditioning demonstrated in Section 2.

For the rest of this section, we fix a Markov category $\mathbb{C}$.

4.1 Conditionals

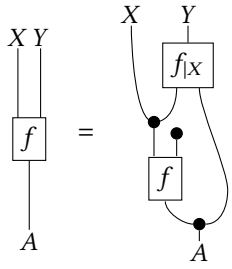
In essence, conditioning is a way of recovering a joint distribution only given access to part of its information. In programming terms, conditionals are a powerful way of restructuring dataflow to our liking: Given a joint distribution over (X, Y), we can always form a generative story where the value 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-Jacobs.

Definition 4.1 ([10, 11.1]). A *conditional distribution* for $\psi : I \to X \otimes Y$ (given X) is a morphism $\psi|_X : X \to Y$ such that



(14)

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

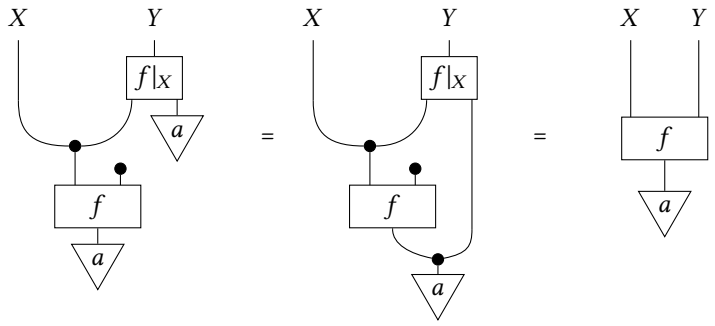


(15)

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

Proposition 4.2. *If $f : A \to X \otimes Y$ has conditional $f|_X : X \otimes A \to Y$ and $a : I \to A$ is a* deterministic *state, then $f|_X(\mathrm{id}_X \otimes a)$ is a conditional distribution for fa.*

Proof. Using determinism of a, we check that



□

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

Proof. In BorelStoch, the definition of conditionals instantiates to *regular conditional distributions* which are known to exist for standard Borel spaces. As a special case, conditionals in FinStoch are given by the traditional conditional distribution [10, 11.2]

$$\psi|_X(y|x) = \frac{\psi(x,y)}{\psi_X(x)} \quad \text{when } \pi_X(x) > 0 \tag{16}$$

Conditionals in Gauss exist and can be given using an explicit formula generalizing (4) [10, 11.8]. The property that conditionals of Gaussians are again Gaussian is sometimes called *self-conjugacy* [21].

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 unique. For example, the formula (16) 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 (4) depends on a choice of generalized inverse, which is only unique on the appropriate support. It turns out that these notions have an elegant abstract formulation which lets us address the non-uniqueness of conditionals in general Markov categories.

Definition 4.4 ([5, 5.1], [10, 13.1]). Let $\mu : I \to X$ be a distribution. Two morphisms $f, g : X \to Y$ are called *μ-almost surely equal* (written $f =_\mu g$) if

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

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

Proposition 4.5. *If $\psi : I \to X \otimes Y$ is a distribution and $\psi|_X, \psi|'_X$ are two morphisms satisfying* (14)*, then*

$$\psi|_X =_{\psi_X} \psi|'_X \tag{17}$$

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

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

Proposition 4.6. (1) *In* FinStoch, $f, g : X \to D(Y)$ *are μ-almost surely equal iff the distributions $f(x) = g(x)$ agree for all x with $\mu(x) > 0$*
(2) *In* BorelStoch, $f, g : X \to \mathcal{G}(Y)$ *are μ-almost surely equal iff $f(x) = g(x)$ as measures for μ-almost all x.*
(3) *In* Gauss, *if $\mu : I \to m$ is a distribution with support S (in the sense of Section 2.1), then $f, g : m \to n$ are μ-almost surely equal iff $fx = gx$ for all $x \in S$, seen as deterministic states $x : 0 \to m$.*
(4) *In* Rel^+, *if $M \subseteq X$ and $R, S : X \to Y$ 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 [10, 13.2] and [11, 3.19]. The result for Gauss is a strengthening of the result for BorelStoch. The morphisms $f, g : m \to n$ can be faithfully considered BorelStoch maps $f, g : \mathbb{R}^m \to \mathcal{G}(\mathbb{R}^n)$, 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. □

Absolute continuity can now be formulated naturally in terms of almost-sure equality:

Definition 4.7 ([11, 2.8]). Given two distributions $\mu, \nu : I \to X$, we say that μ is *absolutely continuous* with respect to ν, written $\mu \ll \nu$, if for all $f, g : X \to Y$ we have

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

Importantly, absolute continuity lets us strengthen statements about almost-sure equality to actual equality.

Lemma 4.8. *If $\mu : I \to X$, $f =_\mu g$ 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* $\mathrm{supp}(\mu) \subseteq \mathrm{supp}(\nu)$.
(4) *In* Rel^+, $R \ll S$ *iff* $R \subseteq S$.

Proof. The claim for FinStoch follows immediately from Proposition 4.6, and the result for BorelStoch is given in [11, 2.9]. Proposition 4.6 implies that the support condition for Gauss is sufficient. To see that it is also necessary, let $x \in \mathrm{supp}(\mu) \setminus \mathrm{supp}(\nu)$. Then we can find two affine functions f, g which agree on $\mathrm{supp}(\nu)$ but $f(x) \neq g(x)$. Now $f =_\nu g$ but not $f =_\mu g$, hence $\mu \not\ll \nu$. □

In FinStoch, we can also rephrase $\mu \ll \nu$ as $\mathrm{supp}(\mu) \subseteq \mathrm{supp}(\nu)$ if we define $\mathrm{supp}(\mu) = \{x : \mu(x) > 0\}$. 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 $x : I \to X$ 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* $x \in \mathrm{supp}(\mu)$
(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 [37].

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* $o : I \to K$.
(2) An *inference problem* over X is a tuple (K, ψ, o) of an object K, a joint distribution $\psi : I \to X \otimes K$ called the model and an observation $o : I \to K$.

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 $\psi|_K \circ o : I \to X$ where $\psi|_K : K \to X$ 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 4.5. □

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

To a programmer, the inference problem (K, ψ, o) represents a closed program of the form

$$\mathsf{let}\ (x, k)\ =\ \psi\ \mathsf{in}\ (k{:=}\, o); x \tag{18}$$

where $k{:=}\, o$ denotes the exact observation we want to make. Our treatment directly follows from desirable program equations like (2) and (3), or the symbolic approach of [37]: Finding a conditional for ψ amounts to restructuring the dataflow of (18) as

$$\mathsf{let}\ k\ =\ \psi_K\ \mathsf{in}\ \mathsf{let}\ x\ =\ \psi|_K(k)\ \mathsf{in}\ (k{:=}\, o); x$$

which by commutativity is the same as

$$\mathsf{let}\ k\ =\ \psi_K\ \mathsf{in}\ (k{:=}\, o); \psi|_K(k)$$

From the initialization principle, if $k \ll \psi_K$, the problem simplifies to

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

which is the posterior $\psi|_K(o)$. If on the other hand $k \not\ll \psi_K$, the computation must return failure.

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

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

which corresponds to the inference problem $(1, \mu, 0)$ where $\mu : 0 \to 2 \otimes 1$ 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 (14). 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 conditioning as a morphism. The key step is to move from a closed inference problem $(K, \psi\colon I \to X \otimes K, o\colon I \to K)$, to *open inference problems* or *conditioning channels*, where ψ is replaced with a morphism with 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.

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

$$x : X \vdash \mathsf{let}\ (y, k) : Y \otimes K\ =\ f(x)\ \mathsf{in}\ (k \coloneqq o); y \tag{19}$$

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 5). Two representations (19) are deemed equivalent if

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

For modularity, we present the construction in two stages: In the first stage (Section 5.1) we form a category $\mathrm{Obs}(\mathbb{C})$ on the same objects as $\mathbb{C}$ consisting of the data (19) but without any quotienting. This adds, purely formally, for every observation $o : I \to X$ an observation effect $(\coloneqq o) : X \rightsquigarrow I$. 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 $\mathrm{Cond}(\mathbb{C})$. 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 Section 5.3 and Section 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 [10, 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 $\mathrm{Obs}(\mathbb{C})$:

- the object part of $\mathrm{Obs}(\mathbb{C})$ is the same as $\mathbb{C}$
- morphisms $X \rightsquigarrow Y$ are tuples (K, f, o) where $K \in \mathrm{ob}(\mathbb{C})$, $f \in \mathbb{C}(X, Y \otimes K)$ and $o \in \mathbb{C}_{\mathrm{det}}(I, K)$, representing (19)
- The identity on X is $\mathrm{Id}_X = (I, \mathrm{id}_X, !)$ where $! = \mathrm{id}_I$.
- Composition is defined by

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

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

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

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

Recall that a symmetric premonoidal category (due to [35]) 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 $\mathrm{Obs}(\mathbb{C})$ does not yet identify observations arriving in different order. This will be remedied automatically later when passing to the quotient $\mathrm{Cond}(\mathbb{C})$. Composition and tensor can be depicted graphically as in Figure 5, where dashed wires indicate condition wires K and their attached observations o. For an observation $o : I \to K$, the conditioning effect $(\coloneqq o) : K \rightsquigarrow I$ is given by (I, id_K, o).

5.2 Cond – Contextual Equivalence of Inference Programs

Let us now assume that $\mathbb{C}$ has all conditionals. We wish to quotient Obs-morphisms, relating them to the conditionals which 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

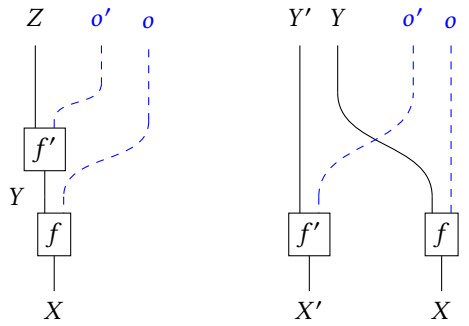


Fig. 5. Composition and tensoring of morphisms in Obs

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 of observational equivalence (Definition 4.14) defines an equivalence relation on states $I \rightsquigarrow X$ in $\mathsf{Cond}(\mathbb{C})$. 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

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

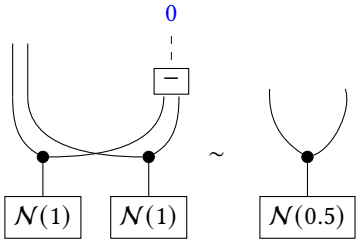


Fig. 6. Example 4.15 describes observationally equivalent states $0 \rightsquigarrow 2$ in $\mathsf{Obs}(\mathsf{Gauss})$

We now give a general recipe to extend an equivalence relation on states to a congruence on arbitrary morphisms $f : X \to Y$.

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 $X \to Y$ via

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

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

We show now that under good assumptions, the quotient by conditioning (Definition 5.2) on $\mathbb{X} = \mathsf{Obs}(\mathbb{C})$ is functorial, and induces a quotient category $\mathsf{Cond}(\mathbb{C})$. 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 $x : I \to X$, $y : I \to Y$, and arbitrary $f : X \to Y$ and $\mu : I \to X$.

(1) $x \otimes y \ll \langle \mathrm{id}_X, f \rangle \mu$
(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 $\langle \mathrm{id}, f \rangle \mu$ is the image space $\{(x, Ax + c) : x \in S, c \in T\}$. Hence $(x, y) \ll \langle \mathrm{id}, f \rangle \mu$ 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 $\langle \mathrm{id}, f \rangle \mu$ iff x has positive probability under μ, and y has positive probability under $f(-|x)$.

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

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

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

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

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

$$(H \otimes K, (f \otimes \mathrm{id}_K)\psi, v \otimes o) \sim (H \otimes K', (f \otimes \mathrm{id}_{K'})\psi', v \otimes o') \tag{20}$$

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 (20) fail and are thus equivalent.

The composite fails. Assume from now that the states succeed and thus also have equal posteriors

$$\psi|_K(o) = \psi'|_K(o') \tag{21}$$

We first show that the success conditions on both sides of (20) are the same, so if the LHS fails so does the RHS. 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) $v \otimes o \ll (f_H \otimes \mathrm{id}_K)\psi$
(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

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

It is clear that condition 2 agrees for both sides of (20). Hence so does 1.

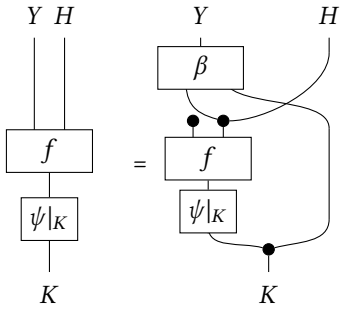
The composite succeeds. We are left with the case that both sides of (20) succeed, and need to show that the composite posteriors agree

$$[(f \otimes \mathsf{id}_K)\psi]|_{H\otimes K}(v \otimes o) = [(f \otimes \mathsf{id}_{K'})\psi']|_{H\otimes K'}(v \otimes o') \tag{22}$$

We use a variant of the argument from [10, 11.11] that double conditionals can be replaced by iterated conditionals. Consider the parameterized conditional

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

with universal property

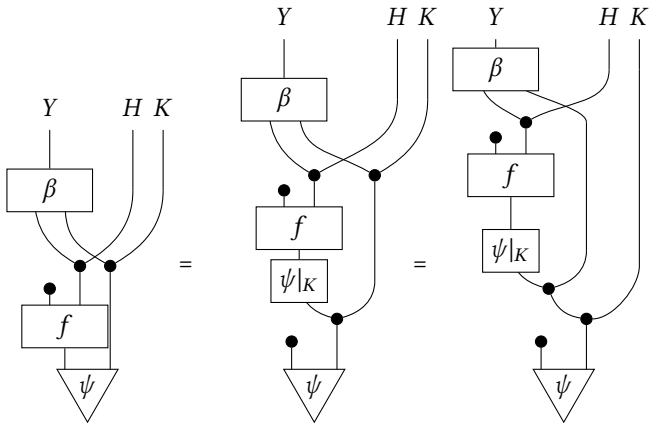


(23)

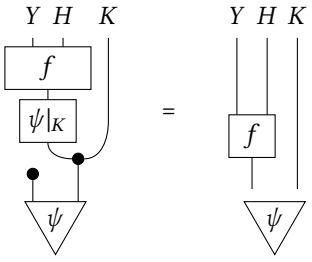
Some string diagram manipulation shows that β too has the universal property of the double conditional

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

We check



which further reduces using (23) to the desired



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

$$\beta(\mathrm{id}_H \otimes o) = (f \circ \psi|_K(o))|_H \tag{24}$$

But this conditional agrees with $(f \circ \psi'|_K(o'))|_H$ by assumption (21). Hence we can evaluate the joint posterior successively,

$$\begin{aligned}
[(f \otimes \mathrm{id}_K)\psi]|_{H\otimes K}(v \otimes o) &= \beta(\mathrm{id}_H \otimes o) \circ v \\
&\overset{(24)}{=} (f \circ \psi|_K(o))|_H \circ v \\
&\overset{(21)}{=} (f \circ \psi'|_{K'}(o'))|_H \circ v \\
&\overset{\text{symmetric}}{=} [(f \otimes \mathrm{id}_{K'})\psi']|_{H\otimes K'}(v \otimes o)
\end{aligned}$$

establishing (22). □

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

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

(1) *$o \ll f_K\psi_X$ and $o' \ll f'_{K'}\psi'_X$ and $[(\mathrm{id}_A \otimes f)\psi]|_K(o) = [(\mathrm{id}_A \otimes f')\psi']|_{K'}(o')$*
(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 $\mathrm{copy}_X \circ \phi$.

By Definition 5.3, the quotient $\mathrm{Obs}(\mathbb{C})/\!\approx$ 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*

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

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

Proof. Let $\psi : I \to A \otimes X$. 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

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

This requires the fact that isomorphisms are deterministic in a Markov category with conditionals [10, 11.26]. The proof more generally works if α is deterministic and split monic. □

We can now give the Cond construction:

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

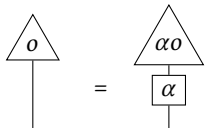
$$\mathrm{Cond}(\mathbb{C}) = \mathrm{Obs}(\mathbb{C})/\!\approx$$

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

5.3 Laws for Conditioning

We will now establish convenient properties of $\mathsf{Cond}(\mathbb{C})$ 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 $\mathsf{Cond}(\mathbb{C})$ is a well-defined CD category already implies that commutativity equation holds for such programs (Proposition 3.13).

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



Proposition 5.10. *The functor J is faithful for common Markov categories. Concretely, morphisms $f, g : X \to Y$ are equated via $J(f) \approx J(g)$ if and only if*

$$\forall \psi : I \to A \otimes X, (\mathsf{id}_A \otimes f)\psi = (\mathsf{id}_A \otimes g)\psi \tag{25}$$

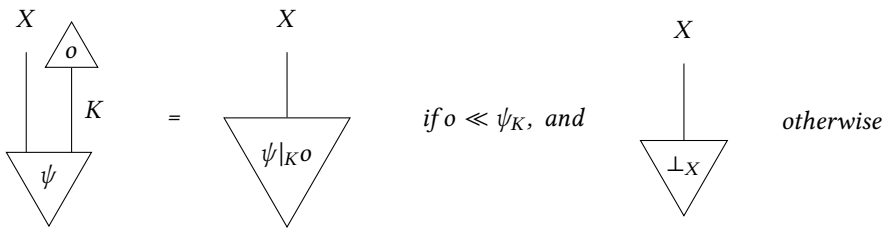
In particular, J is faithful whenever I is a separator. This is the case for Gauss, FinStoch, BorelStoch *and* Rel^+.

Proof. Directly from the definition of $\approx$. □

By construction, the states in $\mathsf{Cond}(\mathbb{C})$ are precisely to 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 $\mathsf{Cond}(\mathbb{C})$ are of the following form:*

(1) *There exists a unique failure state $\bot_X : I \rightsquigarrow X$ 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*

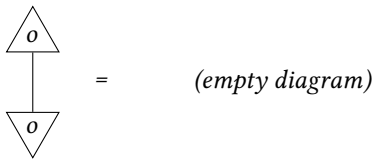


(3) *Failure is "strict" in the sense that any composite or tensor with $\bot$ gives $\bot$.*
(4) *The only scalars $I \rightsquigarrow I$ are id_I and $\bot_I$. Both are copyable, but $\bot_I$ is not discardable.*

Proof. By definition of $\sim$. □

[2] it 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

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



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 (2). Categorically, we are asking how the conditioning effect interacts with copying:

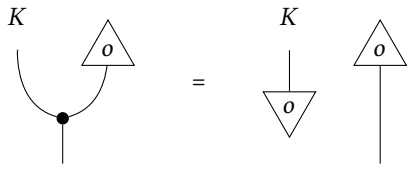
Proposition 5.13 (Enforcing conditions). *We have*

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

In programming notation, this is

$$(x \coloneqq o); x \approx (x \coloneqq o); o$$

and in string diagrams



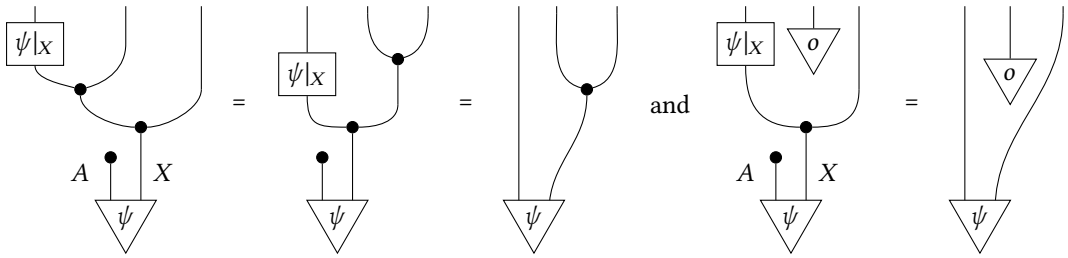
(26)

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 $\psi : I \to A \otimes X$; 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

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

as evidenced by the following string diagrams



Composing with o, we obtain the desired equal posteriors

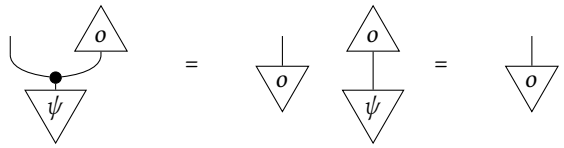
$$\langle \psi|_X, \mathrm{id}_X \rangle o = \psi|_X(o) \otimes o = (\psi|_X \otimes o)(o)$$

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*

$$(\mathsf{let}\, x \;=\; \psi \,\mathsf{in}\, (x \coloneqq o); x) \approx o$$

Proof. Combining Proposition 5.13 and Corollary 5.12, we have



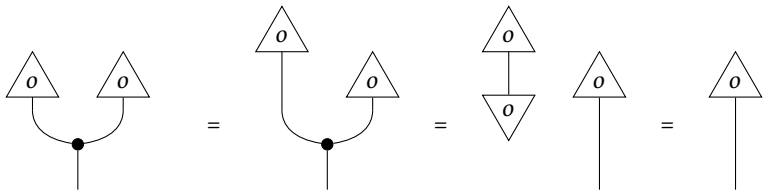
□

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).

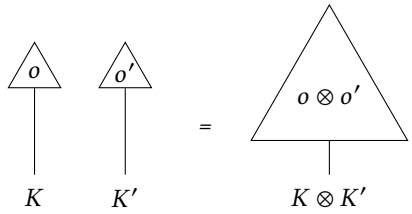
Proof. Again by Proposition 5.13 and Corollary 5.12 we obtain



□

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

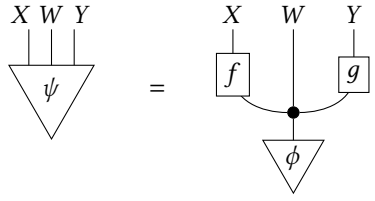
Proposition 5.16 (Aggregation). *Conditions can be aggregated*



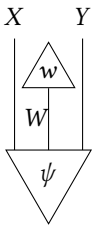
Proof. By definition of the monoidal structure of Obs. □

5.4 Example: Graphical Models and Conditioning

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 (7) of variables X, Y which are conditionally independent on W, the joint distribution ψ can be factored as follows

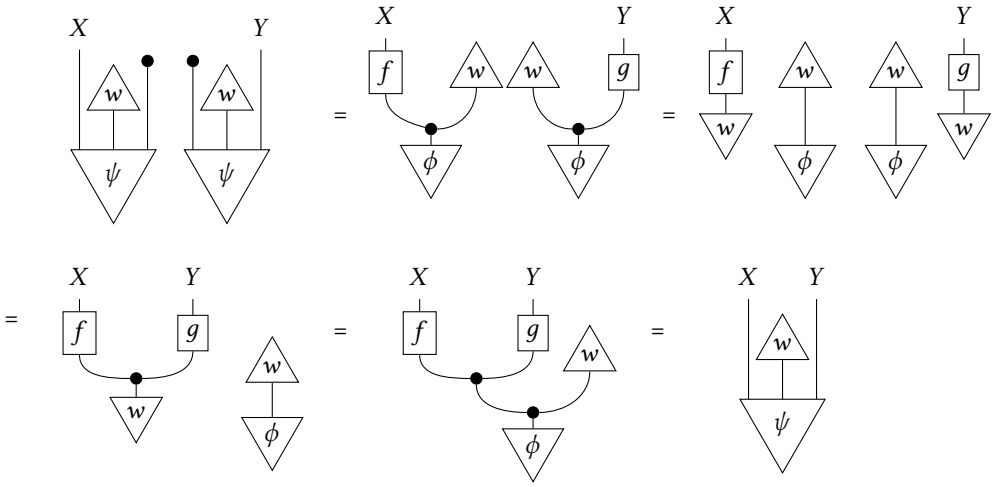


Using the conditioning effects in Cond(ℂ), we can now incorporate *observed nodes* into this language.



(27)

We want to argue that once the 'common cause' W has been observed, X and Y become independent: We can show this purely using graphical reasoning: Applying repeatedly Proposition 5.13, idempotence of scalars and determinism of w, we obtain that (27) is the product of its marginals:



6 DENOTATIONAL SEMANTICS AND CONTEXTUAL EQUIVALENCE

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 Cond(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 Cond(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 Cond(Gauss), where $x :=: y$ is translated as $(x-y) := 0$. A term $\vec{x} : R^m \vdash e : R^n$ denotes a conditioning channel $[\![e]\!] : m \rightsquigarrow n$.

Proposition 6.1 (Correctness). *If* $(e, \psi) \rhd (e', \psi')$ *then* $[\![e]\!]\psi = [\![e']\!]\psi'$. *If* $(e, \psi) \rhd \bot$ *then* $[\![e]\!] = \bot$.

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

$$\mathsf{let}\, x \;=\; \psi \,\mathsf{in}\, (x, [\![e]\!]) = \mathsf{let}\, (x, x') \;=\; \psi' \,\mathsf{in}\, (x, [\![e']\!]). \tag{28}$$

This notion is stable under reduction contexts.

Let C be a reduction context. Then

$$\begin{aligned}
&\mathsf{let}\, x \;=\; \psi \,\mathsf{in}\, (x, [\![C[e]]\!](x)) \\
&= \mathsf{let}\, x \;=\; \psi \,\mathsf{in}\, \mathsf{let}\, y \;=\; [\![e]\!](x) \,\mathsf{in}\, (x, [\![C]\!](x, y)) \\
&= \mathsf{let}\, (x, x') \;=\; \psi' \,\mathsf{in}\, \mathsf{let}\, y \;=\; [\![e']\!](x, x') \,\mathsf{in}\, (x, [\![C]\!](x, y)) \\
&= \mathsf{let}\, (x, x') \;=\; \psi' \,\mathsf{in}\, (x, [\![C[e']]\!])
\end{aligned}$$

Now for the redexes

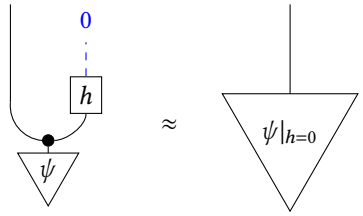
(1) The rules for let follow from the general axioms of value substitution in the internal language
(2) For normal() we have $(\mathsf{normal}(), \psi) \rhd (x', \psi \otimes \mathcal{N}(0,1))$ and verify

$$\begin{aligned}
&\mathsf{let}\, x \;=\; \psi \,\mathsf{in}\, (x, [\![\mathsf{normal}()]\!]) \\
&= \psi \otimes \mathcal{N}(0,1) \\
&= \mathsf{let}\, (x, x') \;=\; \psi \otimes \mathcal{N}(0,1) \,\mathsf{in}\, (x, [\![x']\!])
\end{aligned}$$

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

$$\mathsf{let}\, x \;=\; \psi \,\mathsf{in}\, (x, [\![v ::: w]\!]) = \mathsf{let}\, x \;=\; \psi|_{v=w} \,\mathsf{in}\, (x, ())$$

Let $h = v - w$, then we need to the following morphisms are equivalent in Cond(Gauss):



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

Theorem 6.2 (Full abstraction). *$[\![e_1]\!] = [\![e_2]\!]$ 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 $[\![-]\!]$ is compositional, we obtain $[\![K[e_1]]\!] = [\![K[e_2]]\!]$. If both succeed, we have reductions $(K[e_i], !) \rhd^* (v_i, \psi_i)$ and by correctness $v_1\psi_1 = [\![K[e_1]]\!] = [\![K[e_2]]\!] = v_2\psi_2$ as desired. If $[\![K[e_1]]\!] = [\![K[e_2]]\!] = \bot$ then both $(K[e_i], !) \rhd^* \bot$.

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

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 the among the acceptable results. As $N \to \infty$, 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

$$\mathsf{let}\, x \;=\; \mathsf{bernoulli}(0.4)\, \mathsf{in}\, \mathsf{let}\, y \;=\; \mathsf{bernoulli}(0.4)\, \mathsf{in}\, x \mathrel{::=} y; x$$

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 $\mathsf{bernoulli}(\frac{0.16}{0.36}) \approx \mathsf{bernoulli}(0.44)$.

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

$$\mathsf{let}\, y \;=\; \mathsf{bernoulli}(0.4)\, \mathsf{in}\, x \mathrel{::=} y; x \tag{29}$$

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

$$\mathsf{let}\, x \;=\; \mathsf{bernoulli}(0.4)\, \mathsf{in}\, [-]$$

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 program (29) as equivalent to

$$\mathsf{let}\, z \;=\; \mathsf{bernoulli}(0.2)\, \mathsf{in}\, z \mathrel{::=} \mathsf{false}; \mathsf{let}\, y \;=\; \mathsf{bernoulli}(0.4)\, \mathsf{in}\, x \mathrel{::=} y; x \tag{30}$$

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 (Def. **??**). 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 $X \to Y$ 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* Cond(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 $\rho_Q : X \to D_{\leq 1}(Y)$ given by the likelihood function

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

Conversely, we associate to every subprobability kernel $\rho : X \to D_{\leq 1}(Y)$ 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

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

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. □

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 explore 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.

Example 6.5. A projectivized subprobability kernel $p : X \to Y$ 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 $\mathsf{FinProjStoch}(1, Y)$ is to give either a normalizable distribution $p \in \mathsf{FinStoch}(1, Y)$ or the failure kernel $\bot_Y = 0$.

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

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

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 $u_X : 1 \rightsquigarrow X$.*

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 $\mathsf{Cond}(\mathsf{FinStoch})$. □

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 nondeterminism. The Cond construction here does nothing more than add the possibility for failure (zero outputs) in a systematic way.

Proposition 6.8. $\mathsf{Cond}(\mathsf{Rel}^+) \cong \mathsf{Rel}$.

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

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

This 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 $(x, y, k_0) \in R \Leftrightarrow (x, y, 1) \in R''$. □

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 $v_X : 1 \to X$ 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 which 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 (30) 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. 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 $\mathsf{Cond}(\mathsf{FinStoch})$ does not have coproducts.

We consider the CD-calculus (Section 3.2) with base types X for all finite sets and function symbols $\underline{f}$ for all subprobability kernels $f : X \to D_{\leq 1}(Y)$:

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

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

$$\mathrm{score}_p \in D_{\leq 1}(1) \text{ and } (\coloneqq) : X \times X \to D_{\leq 1}(1)$$

We denote the this calculus **PSL**, for *straight-line inference* Following the development in Section 3.11, 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:

$$t ::= \ldots \mid \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$$

with typing rule

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

This language can also be interpreted in FinSubStoch, via

$$[\![\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3]\!](a|\gamma) = [\![t_2]\!](a|\gamma) \cdot [\![t_1]\!](\mathrm{true}|\gamma) + [\![t_3]\!](a|\gamma) \cdot [\![t_1]\!](\mathrm{false}|\gamma)$$

Categorically, this makes use of the distributive coproducts in FinSubStoch [34, 40]. **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 $\mathsf{normalize} : D_{\leq 1}(X) \to D_{\leq 1}(X)$ which is defined as

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

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 $\mathsf{normalize}([\![s]\!]) = \mathsf{normalize}([\![t]\!])$.

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 $s \approx_{\mathbf{PSL}} t$, 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 $s \approx_{\mathbf{P}} t$, 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.*

$$s \approx_{\mathbf{PSL}} t \Leftrightarrow [\![s]\!] \propto [\![t]\!]$$

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

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

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

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

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

By assumption $[\![C[s]]\!] \propto [\![C[t]]\!]$, so we have $[\![s]\!] \propto [\![t]\!]$. □

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 which differ by a global constant are not distinguishable, that is auto-normalization is valid. Note that this does not imply that t and $\mathsf{normalize}(t)$ are straight-line equivalent, see (29). It does however mean that if programs only differ in their normalization constant, then this fact cannot be observed.

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

$$\mathsf{if}\ \mathsf{bernoulli}(0.5)\ \mathsf{then}\ y; \mathsf{true}\ \mathsf{else}\ \mathsf{false}$$

normalizes to a distribution which returns true with probability

$$p = \frac{0.5 \cdot Z}{0.5 \cdot Z + 0.5} = \frac{Z}{Z+1} \qquad \text{where } Z = \sum_x [\![y]\!](x)$$

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

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

$$s \approx_{\mathbf{P}} t \Leftrightarrow [\![s]\!] = [\![t]\!] : [\![\Gamma]\!] \to D_{\leq 1}([\![X]\!])$$

That is FinSubStoch *is fully abstract for the language* **P**.

Proof. From straight-line equivalence, we know that $[\![s]\!] = \lambda \cdot [\![t]\!]$. We then use the 'Bayesian model selection' trick to show $\lambda = 1$. □

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 form 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

$$\mathsf{ite} : 2 \times X \times X \to D(X)$$

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

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

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 [37] (see also [14, 32, 33, 46]). That work provides in particular verified program transformations to convert an arbitrary probabilistic program of type $\mathsf{R} \otimes \tau$ to an equivalent one that is of the form

$$\mathsf{let}\, x \;=\; \mathsf{lebesgue}()\, \mathsf{in}\, \mathsf{let}\, y \;=\; M\, \mathsf{in}\, (x, y)$$

Now the exact conditioning $x \coloneqq o$ 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 our work [44] and [37]. 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 ([23, Ch. 15], [22, §3.3]): It arises when naively substituting equivalent equations within $(\coloneqq)$. For example, the equation $x - y = 0$ is equivalent to $x/y = 1$ over the (nonzero) real numbers. Yet, in a hypothetical extension of our language which allows division, the following programs would not contextually equivalent, as discussed in Example 2.2:

```
x = normal(0,1)        x = normal(0,1)
y = normal(0,1)   ≢    y = normal(0,1)
x−y =:= 0              x/y =:= 1
```

For that reason, we make it clear in our treatment of inference problems (Section 4.3) that conditioning on a deterministic observation $(\coloneqq)$ is the fundamental notion. Binary conditioning $(\coloneqq)$ is a derived notion which 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 $(s \coloneqq t) : \mathsf{I}$ and $(s == t) : \mathsf{bool}$ 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 explict characterization of contextual equivalence for the Gaussian language is still outstanding. We have given partial results in that direction [44] in the form of a sound equational theory for contextual equivalence. The classification of effects $n \to 0$ in $\mathsf{Cond}(\mathsf{Gauss})$ is not straightforward: it is not true that every effect is observing from a unique distribution $0 \to n$, as for example $(\coloneqq) : 2 \to 0$ is not of that form. We believe that by passing to an extension category of Gaussians $\mathsf{Gauss} \to \mathsf{GaussEx}$, 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 [22] have to be accounted for. Mathematically, this would be an extension of the Cond construction which produces a distributive Freyd category [34]. An example of a categorical model of the Beta-Bernoulli process with branching (but no first-class conditioning) is in [41].

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 work: The Obs construction (Definition 5.1) that we considered here is reminiscent of lenses [6] and the Oles construction [17]. These have recently been applied to probability theory [38], quantum theory [19] and reversible computing [18]. 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 [44] is partly inspired by the algebraic presentation of predicate logic of [39], 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].

ProbLog [7] supports both logic variables as well as 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 [43, 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 modelling 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# [42].

REFERENCES

[1] Baez, J. C., and Erbele, J. Categories in control. *Theory Appl. Categ. 30* (2015), 836–881.

[2] Bonchi, F., Piedeleu, R., Sobocinski, P., and Zanasi, F. Graphical affine algebra. In *Proc. LICS 2019* (2019).

[3] Bonchi, F., Sobocinski, P., and Zanasi, F. The calculus of signal flow diagrams I: linear relations on streams. *Inform. Comput. 252* (2017).

[4] Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. Stan: A probabilistic programming language. *Journal of statistical software 76*, 1 (2017).

[5] Cho, K., and Jacobs, B. Disintegration and Bayesian inversion via string diagrams. *Mathematical Structures in Computer Science 29* (2019), 938 – 971.

[6] Clarke, B., Elkins, D., Gibbons, J., Loregian, F., Milewski, B., Pillmore, E., and Roman, M. Profunctor optics, a categorical update. arxiv:2001.07488, 2020.

[7] De Raedt, L., and Kimming, A. Probabilistic (logic) programming concepts. *Mach. Learn. 100* (2015).

[8] Eaton, M. L. Multivariate statistics: A vector space approach. *Lecture Notes-Monograph Series 53* (2007), i–512.

[9] Fong, B. *Causal Theories: A Categorical Perspective on Bayesian Networks (MSc thesis).* PhD thesis, University of Oxford, 2012.

[10] Fritz, T. A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics. *Adv. Math. 370* (2020).

[11] Fritz, T., Gonda, T., Perrone, P., and Rischel, E. F. Representable Markov categories and comparison of statistical experiments in categorical probability, 2020.

[12] Fritz, T., and Rischel, E. F. Infinite products and zero-one laws in categorical probability. *Compositionality 2* (Aug. 2020).

[13] Führmann, C. Varieties of effects. In *Proc. FOSSACS 2002* (2002), pp. 144–159.

[14] Gehr, T., Misailovic, S., and Vechev, M. PSI: Exact symbolic inference for probabilistic programs. In *Proc. CAV 2016* (2016).

[15] Goodman, N. D., and Stuhlmüller, A. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2021-8-3.

[16] Goodman, N. D., and Tenenbaum, J. B. Probabilistic Models of Cognition. http://probmods.org, 2016. Accessed: 2021-3-26.

[17] Hermida, C., and Tennent, R. D. Monoidal indeterminates and categories of possible worlds. *Theoret. Comput. Sci. 430* (2012).
[18] Heunen, C., and Kaarsgaard, R. Bennett and Stinespring, together at last. In *Proceedings of the 18th International Conference on Quantum Physics and Logic (QPL 2021)* (Sept. 2021), no. 343 in Electronic Proceedings in Theoretical Computer Science, EPTCS, pp. 102–118.
[19] Huot, M., and Staton, S. Universal properties in quantum theory. In *Proc. QPL 2018* (2018).
[20] Infer.net tutorial 3: Learning a gaussian. https://dotnet.github.io/infer/userguide/Learning%20a%20Gaussian%20tutorial.html.
[21] Jacobs, B. A channel-based perspective on conjugate priors. *Mathematical Structures in Computer Science 30*, 1 (2020), 44–61.
[22] Jacobs, J. Paradoxes of probabilistic programming. In *Proc. POPL 2021* (2021).
[23] Jaynes, E. T. *Probability Theory: The Logic of Science*. CUP, 2003.
[24] Joyal, A., and Street, R. The geometry of tensor calculus, i. *Advances in Mathematics 88* (1991), 55–112.
[25] Kammar, O., and Plotkin, G. D. Algebraic foundations for effect-dependent optimisations. In *Proc. POPL 2012* (2012), pp. 349–360.
[26] Lauritzen, S., and Jensen, F. Stable local computation with conditional Gaussian distributions. *Statistics and Computing 11* (11 1999).
[27] Levy, P. B., Power, J., and Thielecke, H. Modelling environments in call-by-value programming languages. *Information and Computation 185*, 2 (2003), 182–210.
[28] MacLane, S. *Categories for the Working Mathematician*. Springer-Verlag, 1971. Graduate Texts in Mathematics, Vol. 5.
[29] Minka, T., Winn, J., Guiver, J., Zaykov, Y., Fabian, D., and Bronskill, J. Infer.NET 0.3, 2018. Microsoft Research Cambridge.
[30] Moggi, E. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science* (1989), IEEE Press, p. 14–23.
[31] Moggi, E. Notions of computation and monads. *Information and Computation 93*, 1 (1991), 55 – 92. Selections from 1989 IEEE Symposium on Logic in Computer Science.
[32] Murray, L., Lundén, D., Kudlicka, J., Broman, D., and Schön, T. Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics* (2018), pp. 1037–1046.
[33] Narayanan, P., and Shan, C. Applications of a disintegration transformation. In *Workshop on program transformations for machine learning* (2019).
[34] Power, J. Generic models for computational effects. *Theor. Comput. Sci. 364*, 2 (2006), 254–269.
[35] Power, J., and Robinson, E. Premonoidal categories and notions of computation. *Math. Struct. Comput. Sci. 7* (1997), 453–468.
[36] Proschan, M. A., and Presnell, B. Expect the unexpected from conditional expectation. *The American Statistician 52*, 3 (1998).
[37] Shan, C.-c., and Ramsey, N. Exact Bayesian inference by symbolic disintegration. In *Proc. POPL 2017* (2017).
[38] St Clere Smithe, T. Bayesian updates compose optically, 2020.
[39] Staton, S. An algebraic presentation of predicate logic. In *Foundations of Software Science and Computation Structures* (Berlin, Heidelberg, 2013), F. Pfenning, Ed., Springer Berlin Heidelberg, pp. 401–417.
[40] Staton, S. Commutative semantics for probabilistic programming. In *Programming Languages and Systems* (Berlin, Heidelberg, 2017), H. Yang, Ed., Springer Berlin Heidelberg, pp. 855–879.
[41] Staton, S., Stein, D., Yang, H., Ackerman, N., Freer, C., and Roy, D. The beta-bernoulli process and algebraic effects. *Proceedings of 45th International Colloquium on Automata, Languages and Programming (ICALP '18)* (02 2018).
[42] Stein, D. GaussianInfer. https://github.com/damast93/GaussianInfer, 2021.
[43] Stein, D. *Structural Foundations for Probabilistic Programming Languages*. PhD thesis, University of Oxford, 2021.
[44] Stein, D., and Staton, S. *Compositional Semantics for Probabilistic Programs with Exact Conditioning*. Association for Computing Machinery, New York, NY, USA, 2021.
[45] van de Meent, J.-W., Paige, B., Yang, H., and Wood, F. An introduction to probabilistic programming, 2018.
[46] Walia, R., Narayanan, P., Carette, J., Tobin-Hochstadt, S., and Shan, C.-c. From high-level inference algorithms to efficient code. *Proc. ACM Program. Lang. 3*, ICFP (July 2019).
[47] Z., Z. *The Schur Complement and Its Applications*. 01 2005.

8 APPENDIX

8.1 CD-calculus

Proof. We will invoke (let.ξ) implicitly throughout. (let.β) follows immediately from (let.val) because x_2 is a value. (id), (let.f), (let.$*$) follow by applying (let.lin) one or two times.

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

$$\begin{aligned} & \mathsf{let}\, x_2 = e_2 \,\mathsf{in}\, \mathsf{let}\, x_1 = e_1 \,\mathsf{in}\, e \\ \overset{(\mathrm{let}.\beta)}{\equiv}\ & \mathsf{let}\, x_2 = e_2 \,\mathsf{in}\, \mathsf{let}\, x_1 = e_1 \,\mathsf{in}\, \mathsf{let}\, x_2 = x_2 \,\mathsf{in}\, e \\ \overset{(\mathrm{let.lin})}{\equiv}\ & \mathsf{let}\, x_1 = e_1 \,\mathsf{in}\, \mathsf{let}\, x_2 = e_2 \,\mathsf{in}\, e \end{aligned}$$

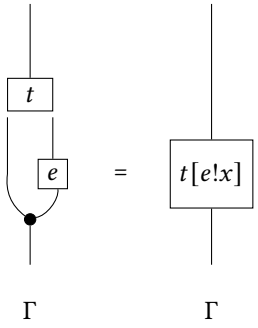
For (assoc), if $x_1 \notin \mathrm{fv}(e)$ then $\mathsf{let}\, x_2 = (\mathsf{let}\, x_1 = x_1 \,\mathsf{in}\, e_2) \,\mathsf{in}\, e$ has a unique free occurrence of x_1, hence

$$\begin{aligned} & \mathsf{let}\, x_1 = e_1 \,\mathsf{in}\, \mathsf{let}\, x_2 = e_2 \,\mathsf{in}\, e \\ \overset{(\mathrm{let}.\beta)}{\equiv}\ & \mathsf{let}\, x_1 = e_1 \,\mathsf{in}\, \mathsf{let}\, x_2 = (\mathsf{let}\, x_1 = x_1 \,\mathsf{in}\, e_2) \,\mathsf{in}\, e \\ \overset{(\mathrm{let.lin})}{\equiv}\ & \mathsf{let}\, x_2 = (\mathsf{let}\, x_1 = e_1 \,\mathsf{in}\, e_2) \,\mathsf{in}\, e \end{aligned}$$

□

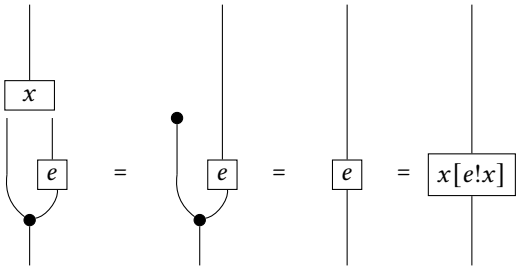
We verify the remaining axioms of the CD calculus.

of Proposition 3.12. (let.ξ) follows from the compositionality of the semantics. ($*.\beta$), ($*.\eta$) and (unit.η) are are immediate. We proceed to prove that (let.lin) is valid, that is if t uses x exactly once, then

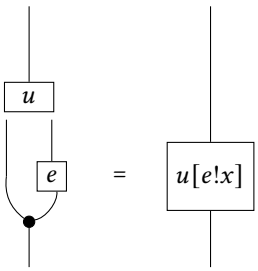


We argue by induction over the term structure of t.

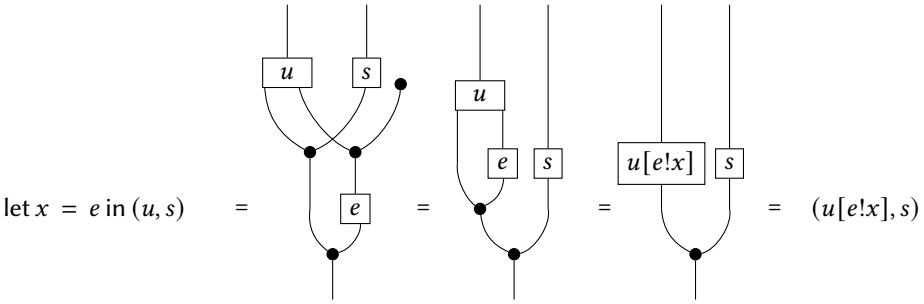
Variable. If $t = x$, then



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

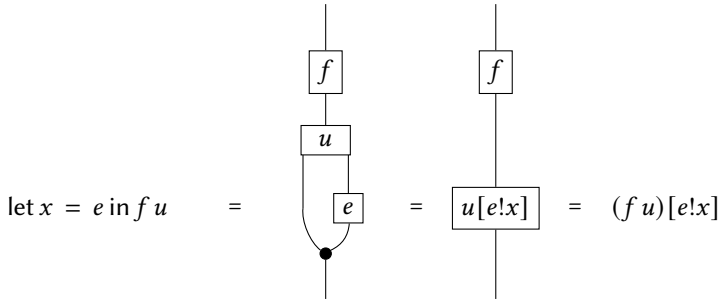


from which we derive by the comonoid laws and weakening of s



The case for (s, u) is symmetric.

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



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

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

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

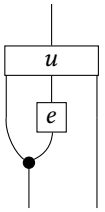
is a special case of (assoc) which was proved in (9).

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

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

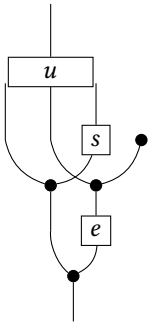
is a special case of (comm). The inductive hypothesis on u involves both weakening and exchange and reads

$u[e!x]$ =

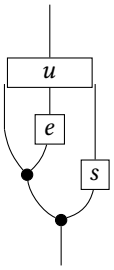


from which we derive

$\mathsf{let}\, x = e \,\mathsf{in}\, \mathsf{let}\, y = s \,\mathsf{in}\, u$ =



=



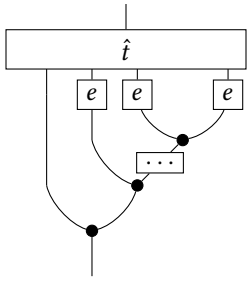
= $\mathsf{let}\, y = s \,\mathsf{in}\, u[e!x]$

This finishes the validation for linear substitution.

For (let.val), we carry out the semantic analogue of Proposition 3.15 and consider sequences of let-bindings

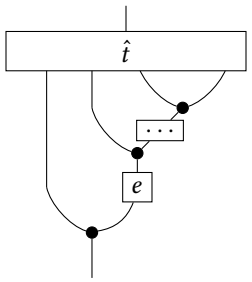
$$\mathsf{let}\, x_1 = e \,\mathsf{in}\, \cdots \mathsf{let}\, x_n = e \,\mathsf{in}\, \hat{t}$$

whose denotation is



Γ

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



Γ

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. □

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

$$
\begin{aligned}
& t[e/x] \\
\overset{(11)}{\equiv}\ & \mathsf{let}\, x_1 \;=\; e \,\mathsf{in}\, \mathsf{let}\, x_2 \;=\; e \,\mathsf{in}\, \hat{t} \\
\overset{(\mathrm{let.val}),(*.\beta)}{\equiv}\ & \mathsf{let}\, p \;=\; (e, e) \,\mathsf{in}\, \mathsf{let}\, x_1 \;=\; \pi_1\, p \,\mathsf{in}\, \mathsf{let}\, x_2 \;=\; \pi_2 p \,\mathsf{in}\, \hat{t} \\
\overset{(12)}{\equiv}\ & \mathsf{let}\, p \;=\; (\mathsf{let}\, x \;=\; e \,\mathsf{in}\, (x, x)) \,\mathsf{in}\, \mathsf{let}\, x_1 \;=\; \pi_1\, p \,\mathsf{in}\, \mathsf{let}\, x_2 \;=\; \pi_2 p \,\mathsf{in}\, \hat{t} \\
\overset{(\mathrm{assoc})}{\equiv}\ & \mathsf{let}\, x \;=\; e \,\mathsf{in}\, \mathsf{let}\, p \;=\; (x, x) \,\mathsf{in}\, \mathsf{let}\, x_1 \;=\; \pi_1\, p \,\mathsf{in}\, \mathsf{let}\, x_2 \;=\; \pi_2 p \,\mathsf{in}\, \hat{t} \\
\overset{(\mathrm{let.val}),(*.\beta)}{\equiv}\ & \mathsf{let}\, x \;=\; e \,\mathsf{in}\, \mathsf{let}\, x_1 \;=\; x \,\mathsf{in}\, \mathsf{let}\, x_2 \;=\; x \,\mathsf{in}\, \hat{t} \\
\overset{(11)}{\equiv}\ & \mathsf{let}\, x \;=\; e \,\mathsf{in}\, t
\end{aligned}
$$

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

$$\mathsf{let}\, x_1 \;=\; e \,\mathsf{in}\, \cdots \mathsf{let}\, x_n \;=\; e \,\mathsf{in}\, \ldots$$

can be replaced by

$$\mathsf{let}\, x \;=\; e \,\mathsf{in}\, \mathsf{let}\, x_1 \;=\; x \,\mathsf{in}\, \cdots \mathsf{let}\, x_n \;=\; x \,\mathsf{in}\, \ldots$$

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

$$
\begin{aligned}
& \mathsf{let}\, x \;=\; e \,\mathsf{in}\, t \\
\overset{(\mathrm{let.val})}{\equiv}\ & \mathsf{let}\, x \;=\; e \,\mathsf{in}\, \mathsf{let}\, y \;=\; () \,\mathsf{in}\, t \\
\overset{(\mathrm{assoc})}{\equiv}\ & \mathsf{let}\, y \;=\; (\mathsf{let}\, x \;=\; e \,\mathsf{in}\, ()) \,\mathsf{in}\, t \\
\overset{(13)}{\equiv}\ & \mathsf{let}\, y \;=\; () \,\mathsf{in}\, t \\
\overset{(\mathrm{let.val})}{\equiv}\ & t
\end{aligned}
$$

□