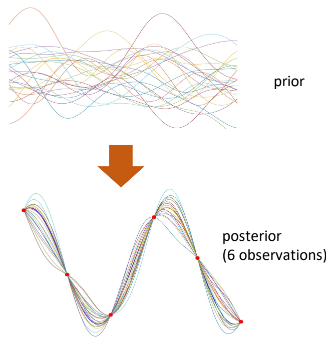
# Compositional Semantics for Probabilistic Programs with Exact Conditioning [LICS'21]

Dario Stein, Sam Staton University of Oxford @damast93

We study the introduction of an *exact conditioning operator* (`=:=`) to a probabilistic language, for example

```
ys = gaussian_process(n=100, kernel=rbf)
for (i,obs) in observations:
  ys[i] =:= obs
```

- Available in e.g. [Hakaru, Infer.NET]
- Contrasts with likelihood-based *scoring* [Stan, WebPPL]




## Advantages of Exact Conditioning

- Intuitive use
- Clean separation of model and observation (`score` statements would have to be interleaved with sampling in `gaussian_process`)
- Equational reasoning and possibly symbolic inference, e.g.

```
x = normal(0.0, 1.0)
y = normal(0.0, 1.0)
x =:= y
```

≈

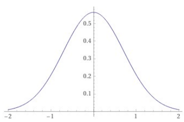
```
x = normal(0.0, 0.5)
y = x
# x = y holds exactly!
```

## Challenges: Semantics

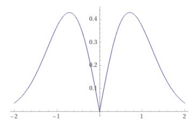
Probability-zero observations introduce many subtleties [Jules Jacobs'21]. **Borel's paradox** can be restated as „equivalent equations need not give equivalent conditions".

*Example:* If x, y are independent and normally distributed, we obtain the following posteriors for x after conditioning [Proschan&Presnell'98]

`x-y =:= 0`



`x/y =:= 1`



When are two conditions interchangeable? Can we still reorder independent lines of the program if they may invoke conditions?

➔ Compositional conditioning needs *semantics* to show consistency, and justify program transformations.

We will develop such semantics, and prove the following desirable properties of exact conditioning:

| | | | |
|---|---|---|---|
| Commutativity: | `a1 =:= a2; b1 =:= b2` | ≈ | `b1 =:= b2; a1 =:= a2` |
| Aggregation: | `a1 =:= a2; b1 =:= b2` | ≈ | `(a1,b1) =:= (a2,b2)` |
| Initialization: | `a = normal(); a =:= 0` | ≈ | `a = 0` |
| Substitution: | `a =:= b; return a` | ≈ | `a =:= b; return b` |

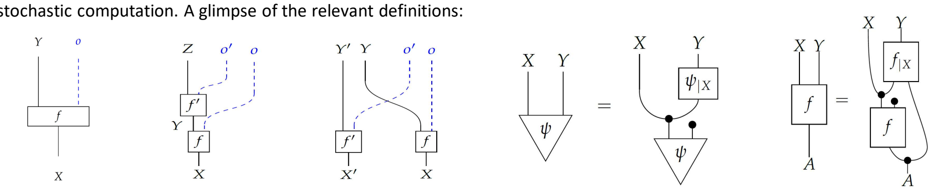
We develop a *structural theory of conditioning* based on program transformations (related to the *symbolic disintegration* of [Shan&Ramsey]). Conditioning is NOT about *densities*, *limiting procedures* or *measure theory* but only about *dataflow properties:*

An **inference problem** is a closed program of the form `let (y,k) = f() in k =:= obs; return y`

An **conditioning channel X → Y** is an open conditioning program of the form `let (y,k) = f(x) in k =:= obs; return y`

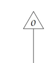
We identify two conditioning channels if they are **contextually equivalent** (compute the same posteriors in all contexts)

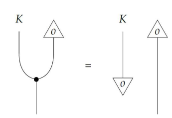
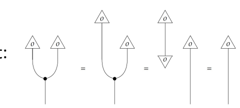
This has a very general formulation in Markov categories [Fritz, Cho-Jacobs], which are an abstract formalism for stochastic computation. A glimpse of the relevant definitions:



Conditioning channel — Sequential composition — Parallel composition — Def: Conditional distribution — Def: Parameterized conditional

**Theorem:** If C is a well-behaved Markov category, then conditioning channels modulo contextual equivalence compose in a well-defined way, forming a CD category Cond(C). The desirable properties hold in Cond(C).

*Bonus:* We obtain a graphical calculus for conditioning, where observations o become *effects* in Cond(C) 

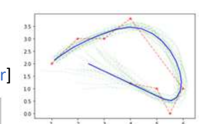
e.g. the graphical substitution law reads  Conditioning is idempotent: 

## Conclusion

The Cond construction provides *general and compositional semantics* for exact conditioning.
It has convenient formal properties and enables equational and graphical reasoning about conditioning programs.

- Assumption: we work in a Markov category with „well-behaved disintegrations" [Shan&Ramsey'17].
- Current examples: Discrete probability, and multivariate Gaussians (help develop more!)

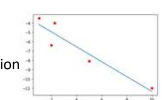
We have implemented *GaussianInfer*, a toy language for Gaussian probability + exact conditioning, based on conditioning channels. Implementation in Python & F# [github.com/damast93/GaussianInfer]



2d Kálmán filter

*Bonus:* An algebraic axiomatization of contextual equivalence for GaussianInfer is available.

Ridge regression



Using *algebraic effects* & abstract types: Exact conditioning (`x=:=y`) and boolean equality (`x==y`) have different formal status and cannot be confused, which helps clear up Borel's paradox.