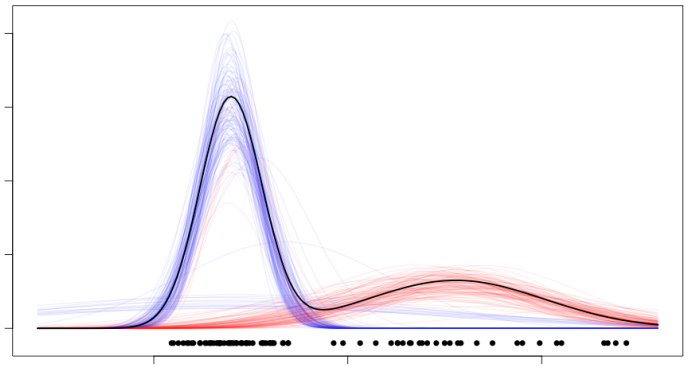
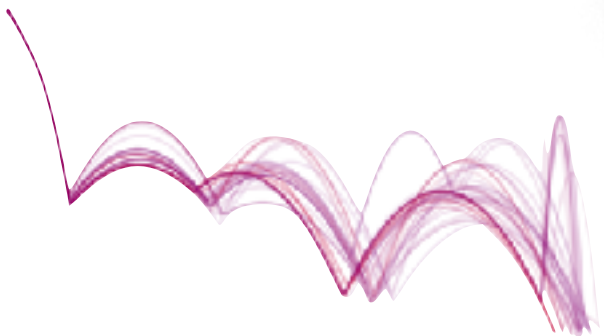
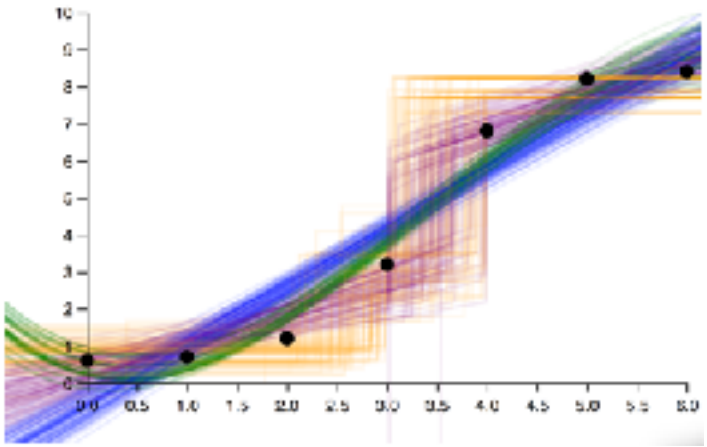
Some formal structures in probability

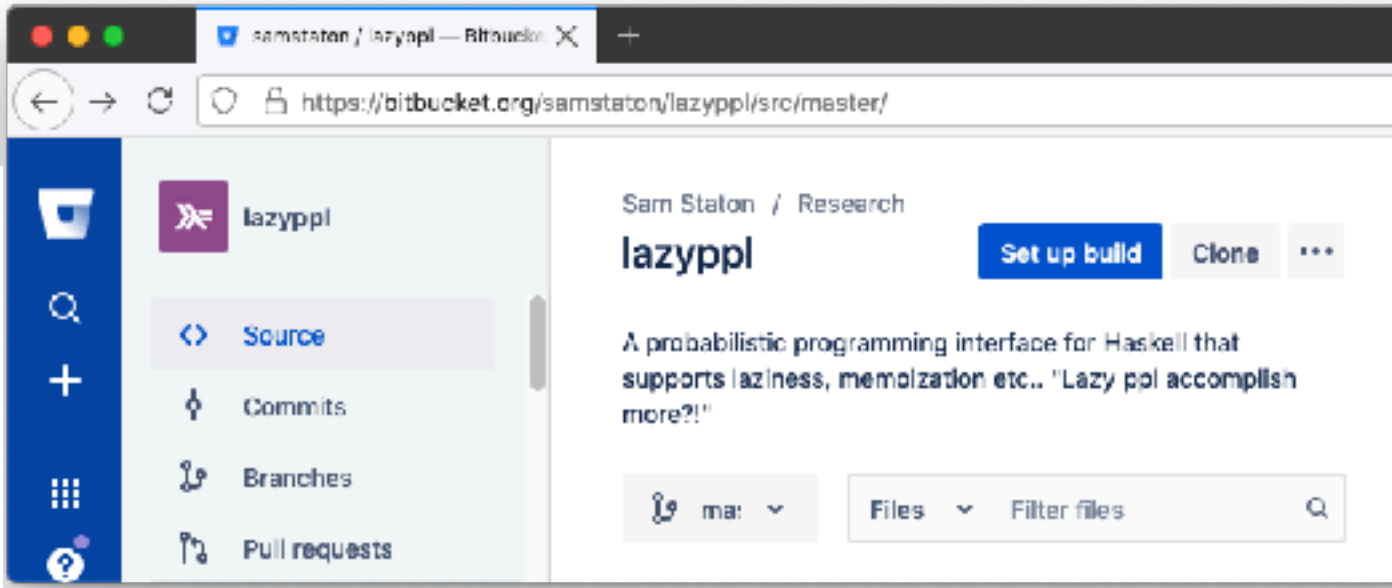
Sam Staton, Oxford

This talk









lazyppl

= lazy **p**robabilistic **p**rogramming **l**anguage

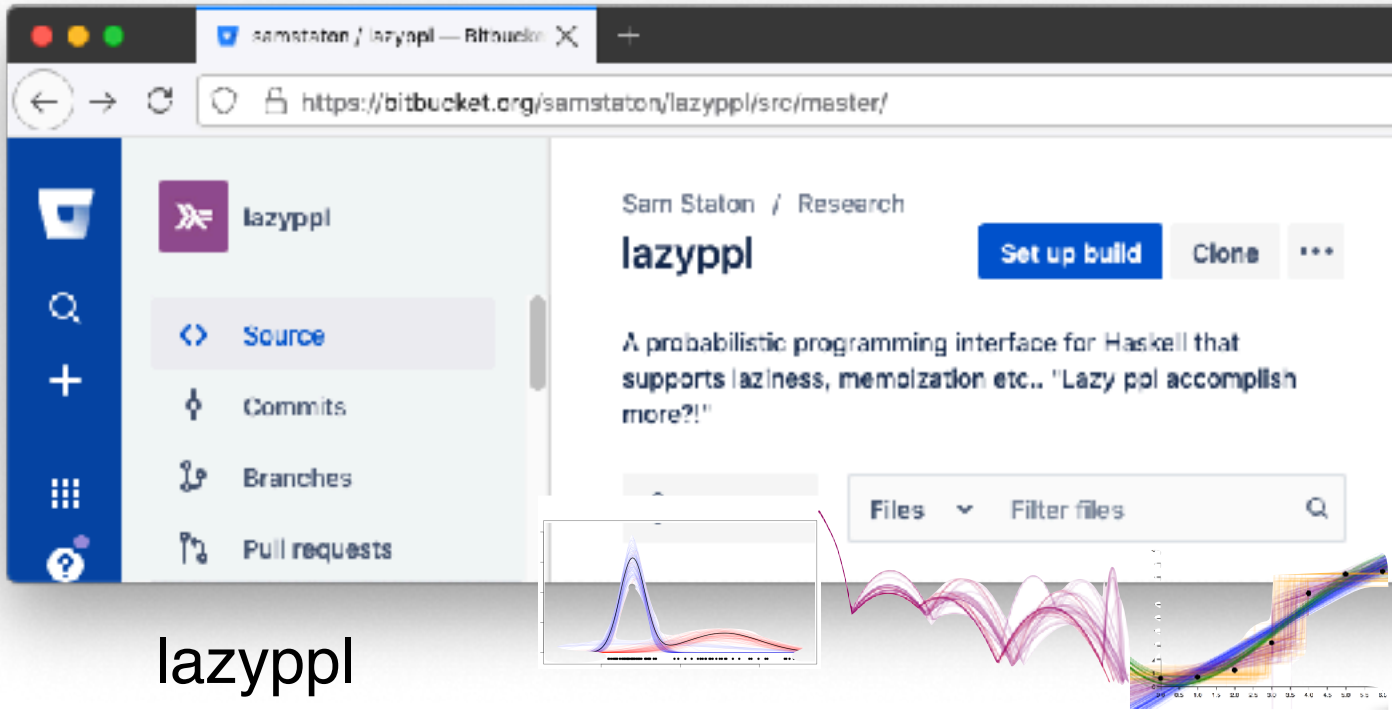
Builds on ideas from *Anglican, BayesDB, Birch, Church, Figaro, Hansei, WebPPL* and others. (But only ~25 lines of code.)

also Koller, McAllester, Pfeffer. AAAI 1997

Key ideas: types, higher order functions, laziness, stochastic memoization.

This talk

Open question:
What is a sound **semantic model** for lazyppl ?



lazyppl
= lazy **p**robabilistic **p**rogramming **l**anguage

Builds on ideas from *Anglican, BayesDB, Birch, Church, Figaro, Hansei, WebPPL* and others. (But only ~25 lines of code.)

also Koller, McAllester, Pfeffer. AAAI 1997

Key ideas: types, higher order functions, laziness, stochastic memoization.

Suggests new ***formal structures in probability.***

e.g. quasi-Borel spaces, Markov categories, nominal sets

Abstraction in traditional programming

High level e.g. higher-order functions
abstract types

Low level e.g. machine code,
Boolean circuits

Abstraction in traditional programming

High level e.g. higher-order functions
abstract types

Low level e.g. machine code,
Boolean circuits

	Engineering	Foundational
High level	✓	✓
Low level	✓	✓

Abstraction in *probabilistic* programming

High level e.g. infinite dimensional systems
higher-order functions
abstract types

Low level e.g. bets, frequencies, decisions
Monte Carlo simulation

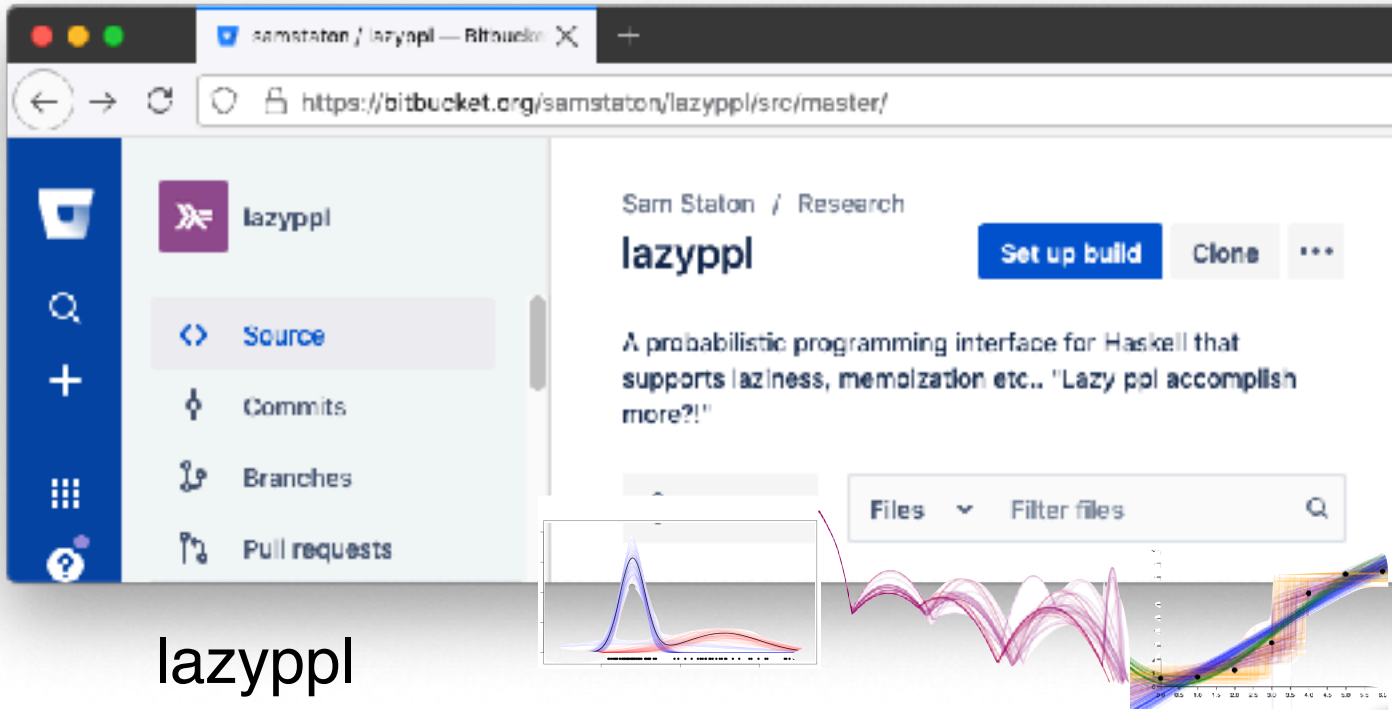
	ML / stats apps	Foundational
High level	✓	✓
Low level	✓	✓

This talk

	ML / stats apps	Foundational
High level	✓	✓
Low level	✓	✓

This talk

Open question:
What is a sound **semantic model** for lazyppl ?



lazyppl
= lazy **p**robabilistic **p**rogramming **l**anguage
Builds on ideas from *Anglican, BayesDB, Birch, Church, Figaro, Hansei, WebPPL* and others. (But only ~25 lines of code.)

Key ideas: types, higher order functions, laziness, stochastic memoization.

Suggests new ***formal structures in probability.***

e.g. quasi-Borel spaces, Markov categories, nominal sets

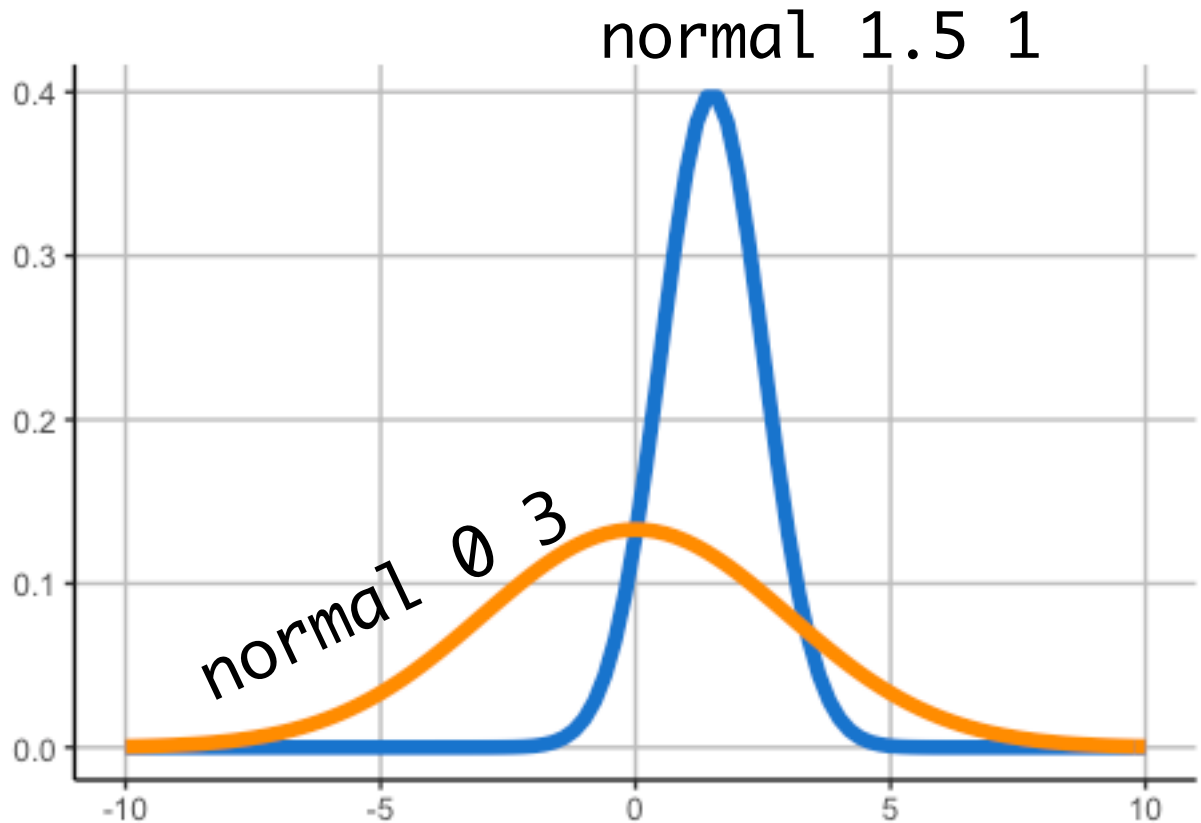
	ML / stats apps	Foundational
High level	✓	✓
Low level	✓	✓

Overview

1. **Examples of high level probabilistic programs**
2. First semantic models
3. Laziness and point processes
4. Stochastic memoization and ongoing work

Random linear functions

```
randlinear :: Prob (RealNum , RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     return (a,b)
```




```
type RealNum = Double
```

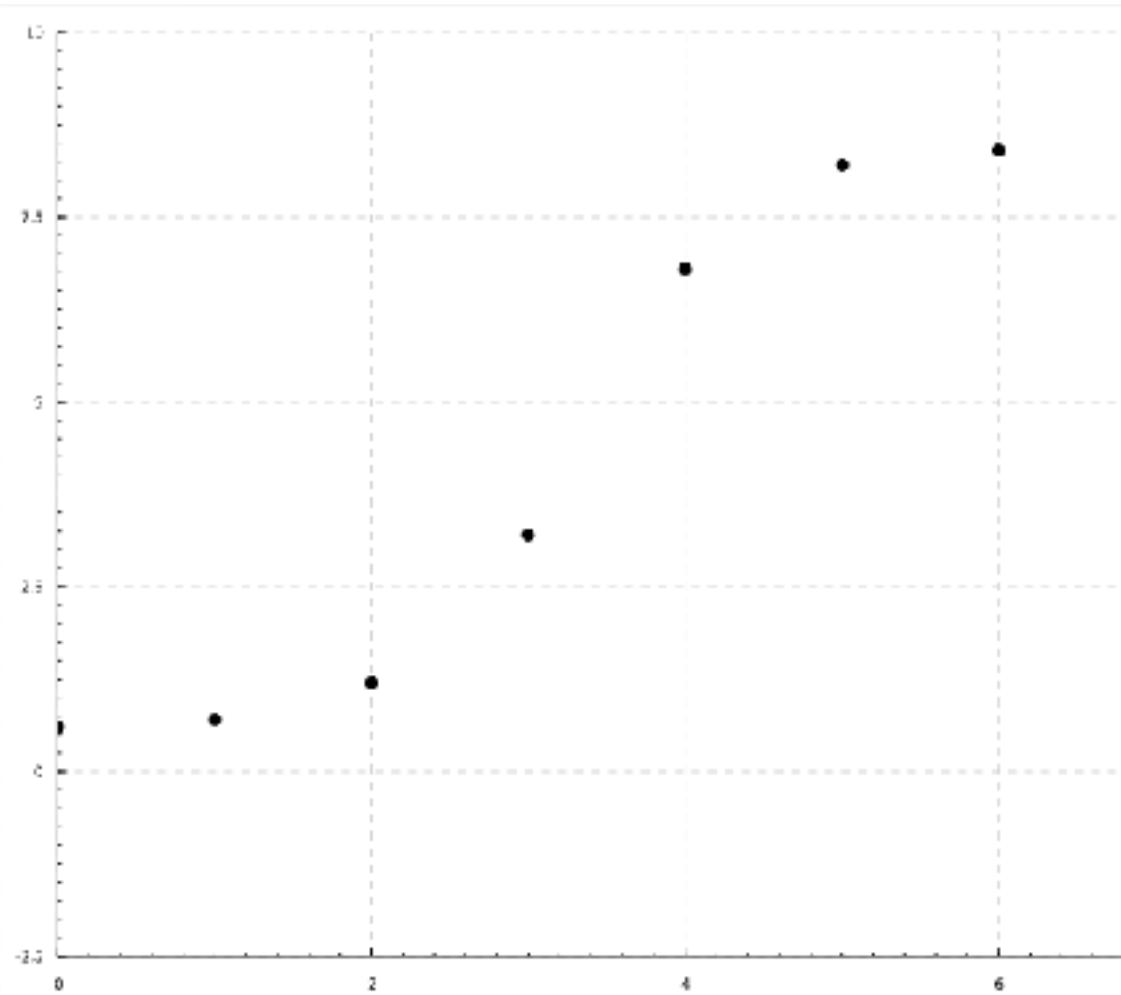
bitbucket.org/samstaton/lazyppl

Random linear functions

```
randlinear :: Prob (RealNum , RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     return (a,b)
```

We will use this for a regression problem:

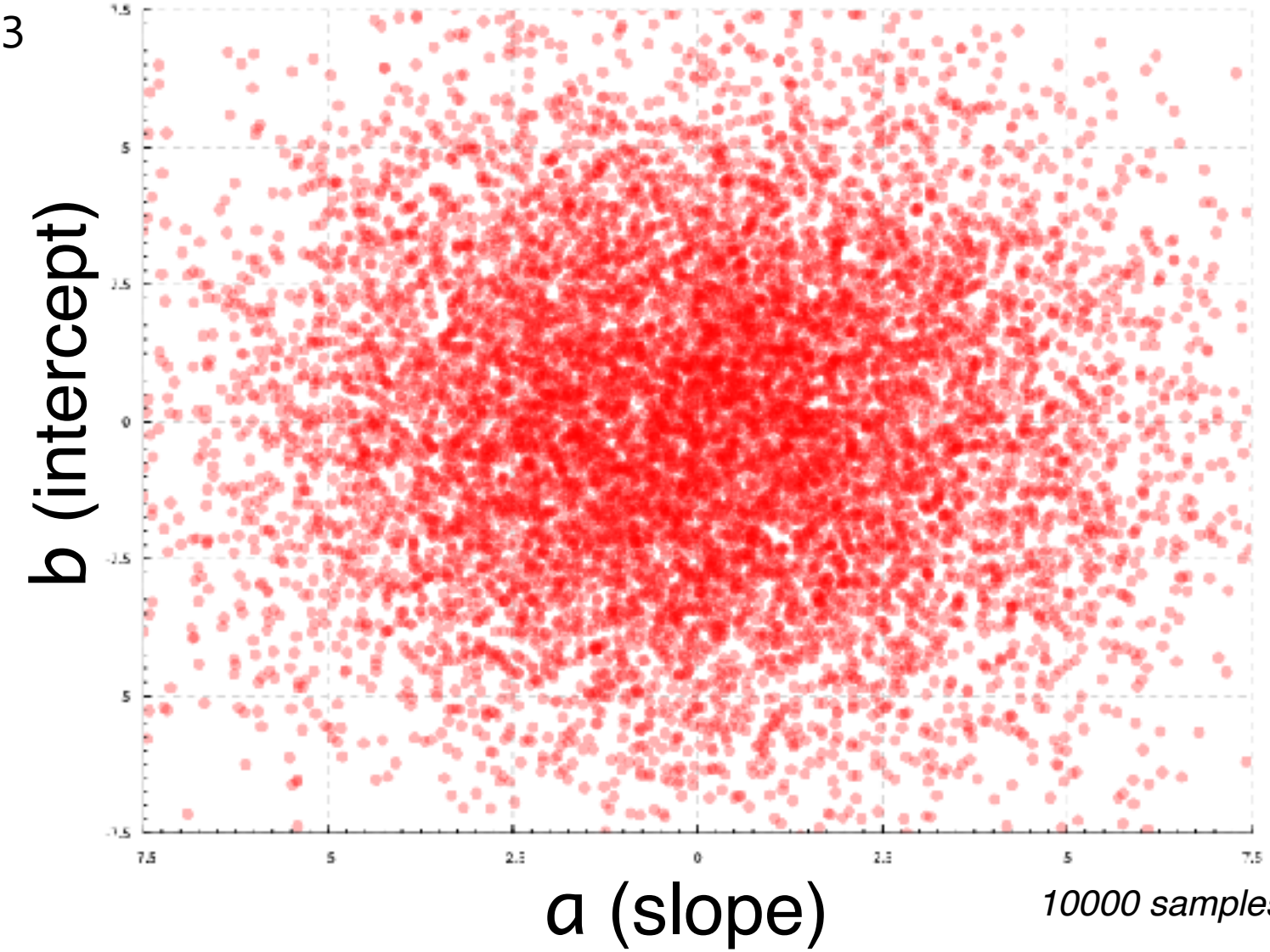
which function probably generated these points?




```
type RealNum = Double
```

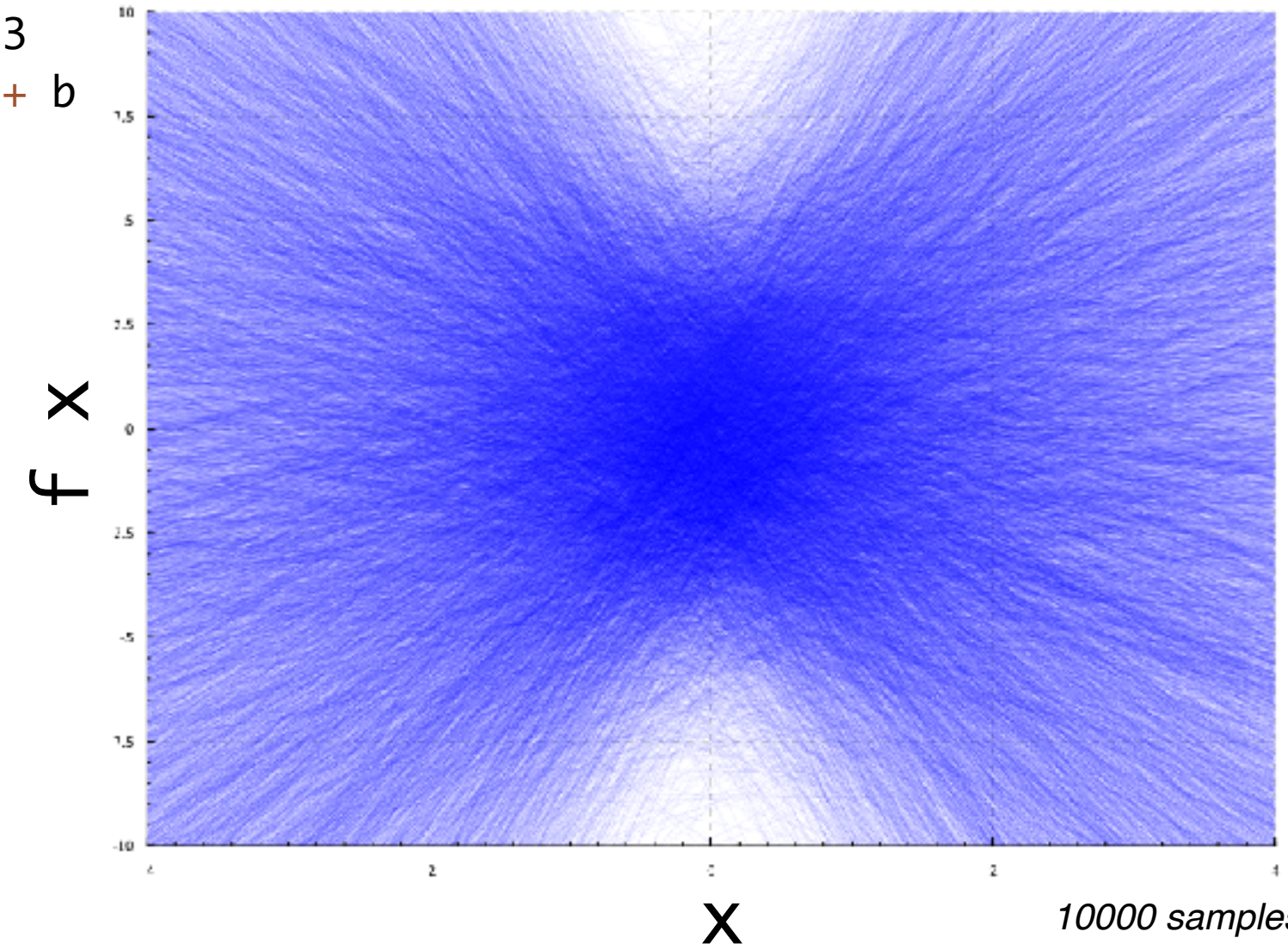
Random linear functions

```
randlinear :: Prob (RealNum , RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     return (a,b)
```



Random linear functions

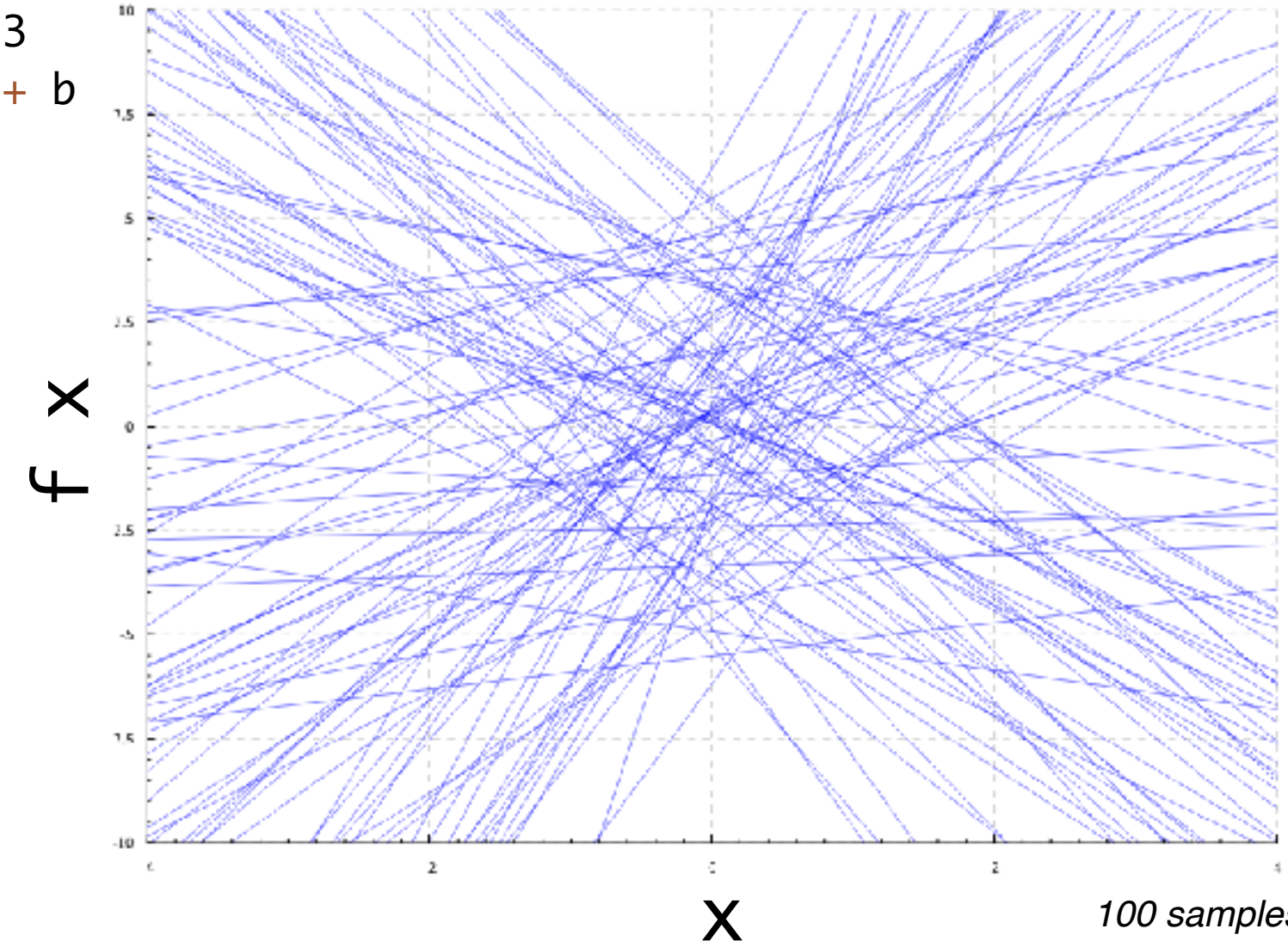
```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```



10000 samples

Random linear functions

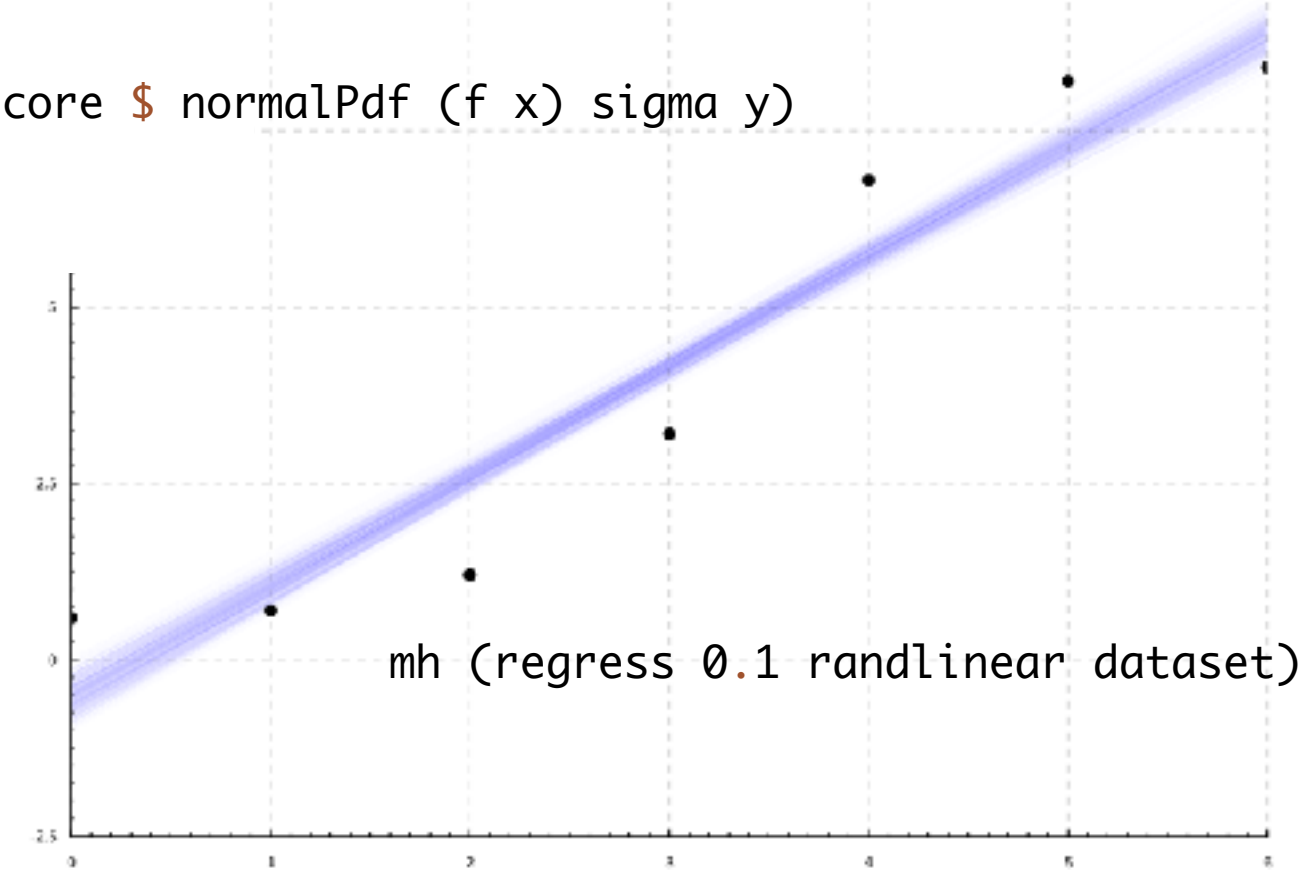
```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```



Bayesian regression

```
randlinear :: Prob (RealNum -> RealNum)
```

```
regress :: RealNum -> Prob (a -> RealNum) -> [(a,RealNum)] -> Meas (a -> RealNum)
regress sigma prior dataset =
  do f <- sample prior
     forM dataset (\(x,y) -> score $ normalPdf (f x) sigma y)
     return f
```



lazyppl *includes a type*

`Meas a`

of unnormalized measures and

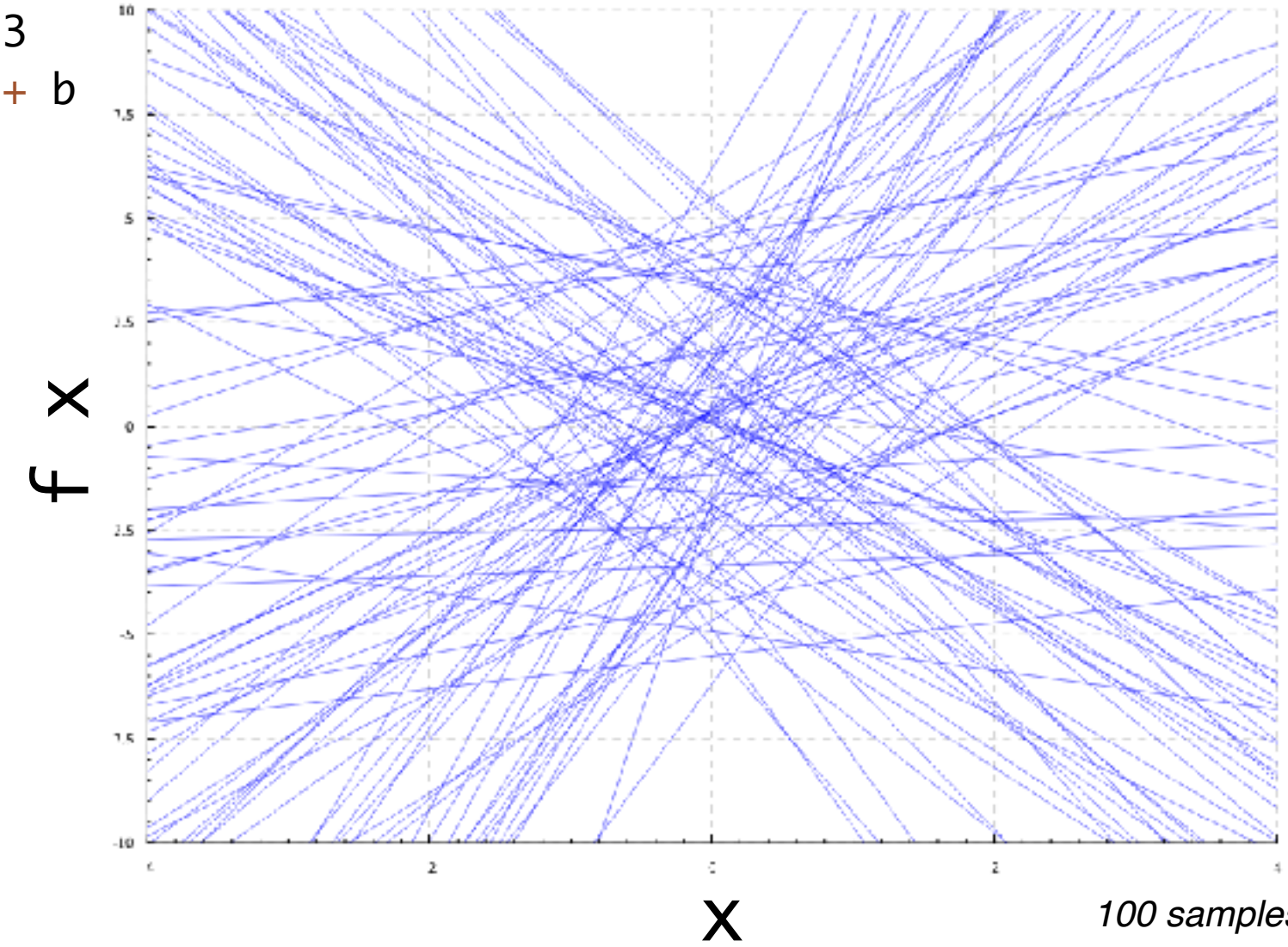
`mh`

a Metropolis-Hastings inference method.

bitbucket.org/samstaton/lazyppl

Random linear functions

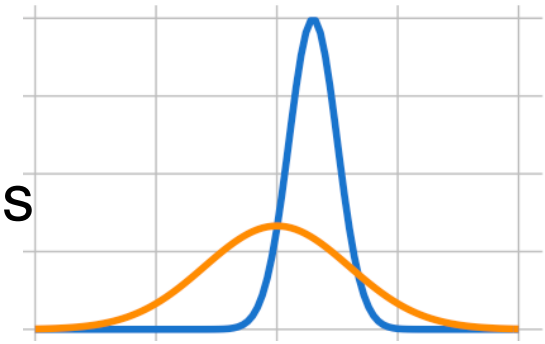
```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```



Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…



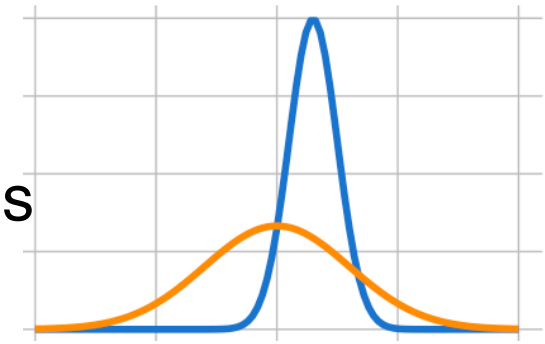
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)
- `Prob Bool` contains probability distributions like `bernoulli 0.5`

Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…

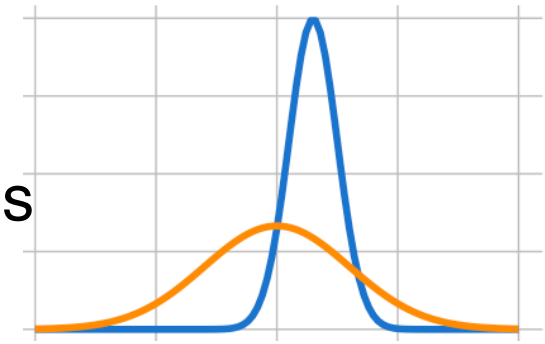
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)



Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…

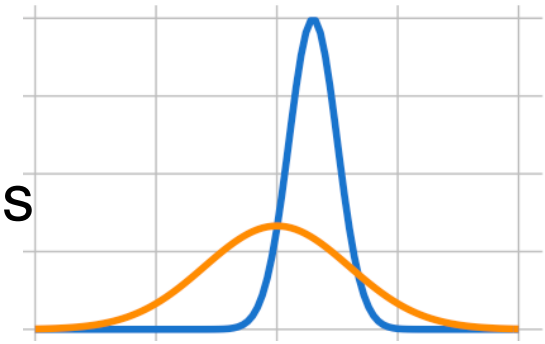


- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)
- `normal :: RealNum -> RealNum -> Prob RealNum` is a parameterized distribution
- `bernoulli :: RealNum -> Prob Bool` is a parameterized distribution too

Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…

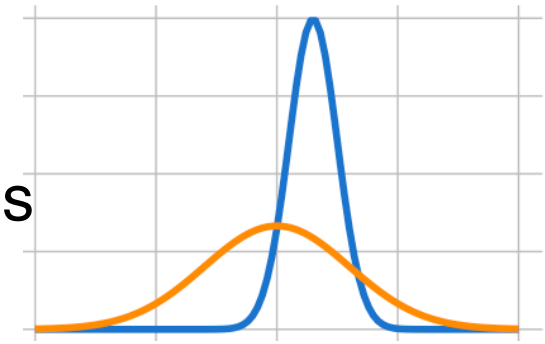


- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)

Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…



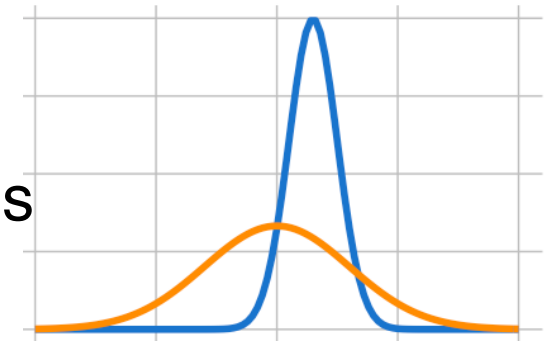
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)



Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
            :: RealNum -> RealNum
```

There's a type constructor `Prob` (a monad), and…



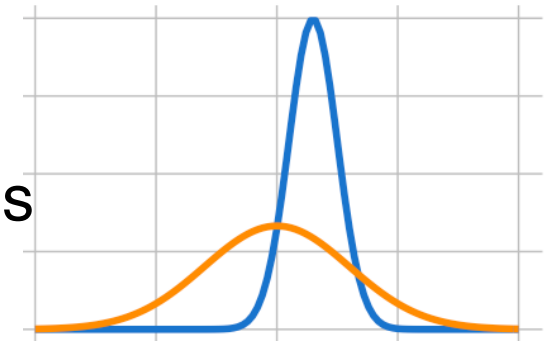
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)



Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
  :: Prob (RealNum -> RealNum)
```

There's a type constructor `Prob` (a monad), and…



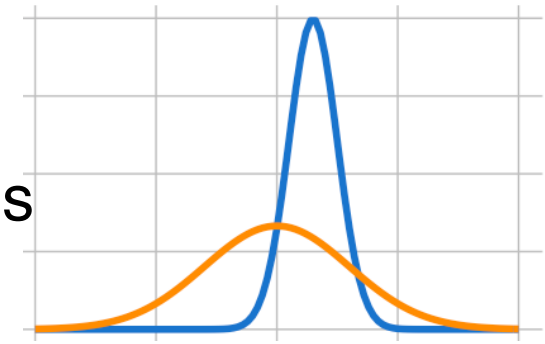
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)



Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…



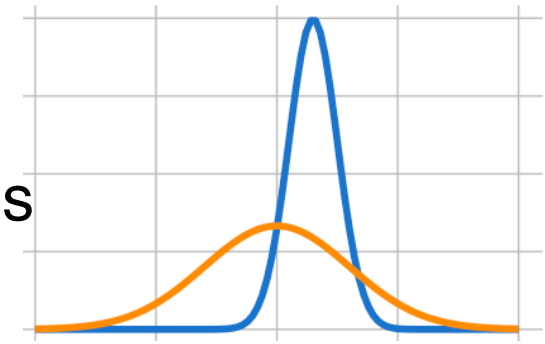
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)



Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…



- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)



- `Prob (Prob Bool)` contains random distributions, etc..

Random functions & program synthesis

```
data Expr = Var | Constt RealNum | Add Expr Expr | Mult Expr Expr
          | IfLess RealNum Expr Expr

eval :: Expr -> (RealNum -> RealNum)
eval Var x = x
eval (Constt r) _ = r
eval (Add e1 e2) x = (eval e1 x) + (eval e2 x)
eval (Mult e1 e2) x = (eval e1 x) * (eval e2 x)
eval (IfLess r e1 e2) x = if x < r then eval e1 x else eval e2 x
```

Random functions & program synthesis

```
data Expr = Var | Constt RealNum | Add Expr Expr | Mult Expr Expr
          | IfLess RealNum Expr Expr

eval :: Expr -> (RealNum -> RealNum)
eval Var x = x
eval (Constt r) _ = r
eval (Add e1 e2) x = (eval e1 x) + (eval e2 x)
eval (Mult e1 e2) x = (eval e1 x) * (eval e2 x)
eval (IfLess r e1 e2) x = if x < r then eval e1 x else eval e2 x


randexpr :: Prob Expr

randprog :: Prob (RealNum -> RealNum)
randprog = do e <- randexpr
              return (eval e)
```

Random functions & program synthesis

```
data Expr = ...

eval :: Expr -> (RealNum -> RealNum)


randexpr :: Prob Expr

randprog :: Prob (RealNum -> RealNum)
randprog = do e <- randexpr
              return (eval e)
```

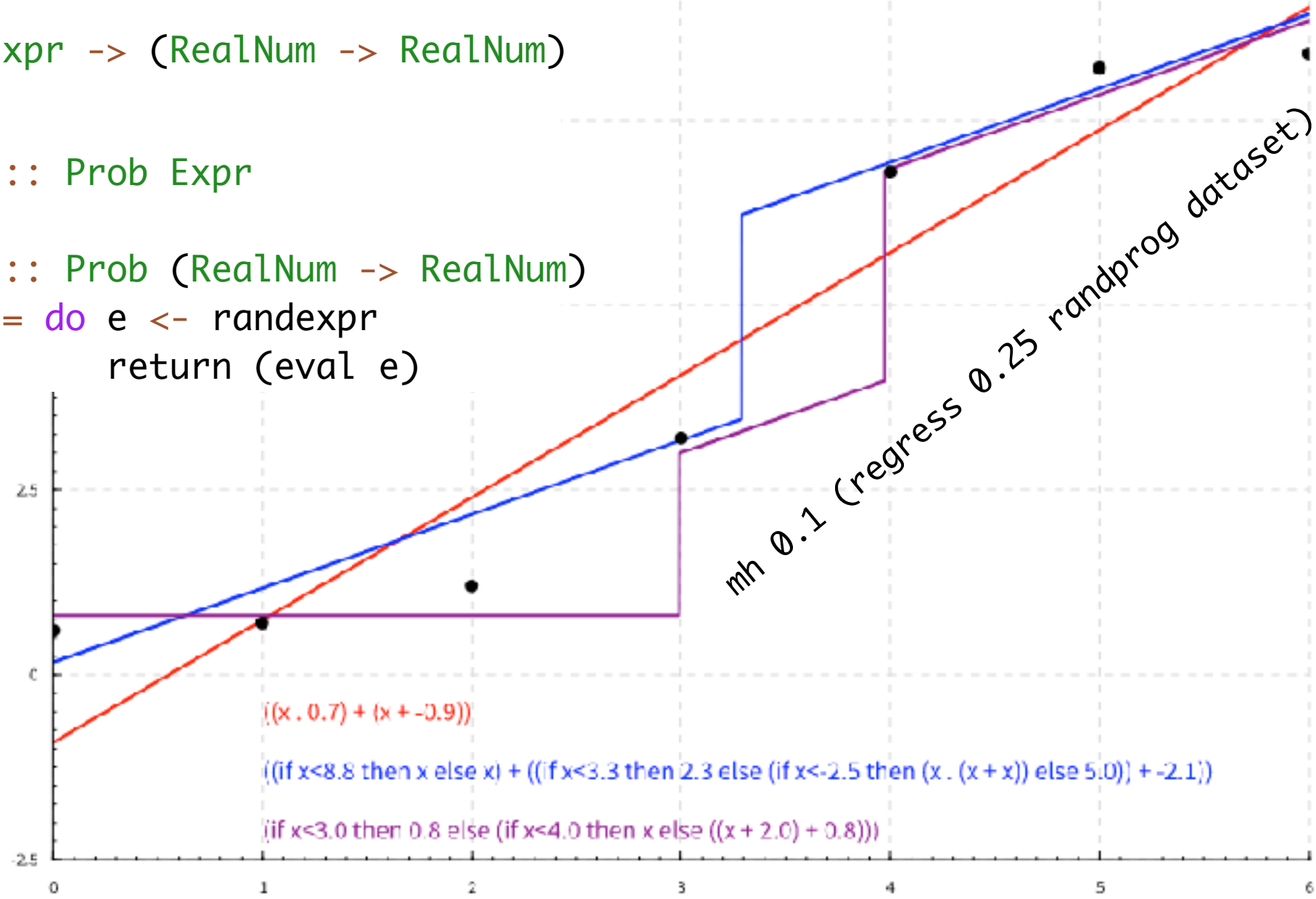
Random functions & program synthesis

```
data Expr = ...

eval :: Expr -> (RealNum -> RealNum)

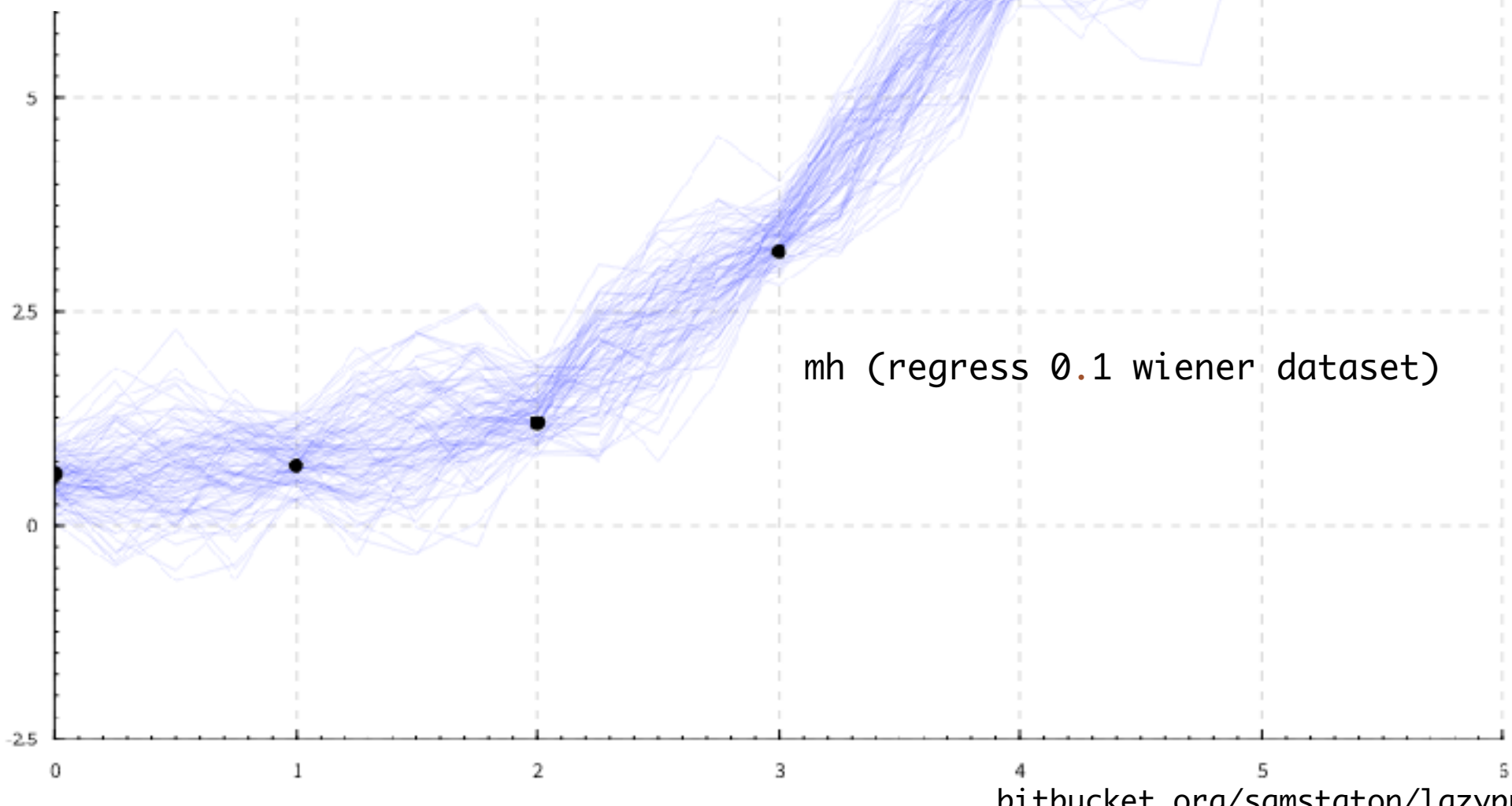

randexpr :: Prob Expr

randprog :: Prob (RealNum -> RealNum)
randprog = do e <- randexpr
              return (eval e)
```



Gaussian processes as random functions

```
wiener :: Prob (RealNum -> RealNum)
```




```
mh (regress 0.1 wiener dataset)
```

bitbucket.org/samstaton/lazyppl

Overview

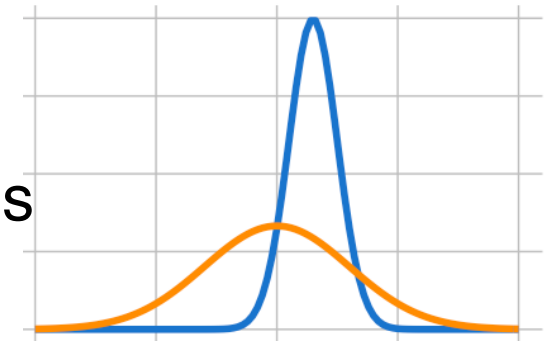
1. Examples of high level probabilistic programs
   `randlinear, randprog, wiener, regress`
2. **First semantic models**
3. Laziness and point processes
4. Stochastic memoization and ongoing work

A semantic model:

Quasi-Borel spaces

Heunen, Kammar, Staton, Yang, LICS 2017

There's a type constructor `Prob` (a monad), and…



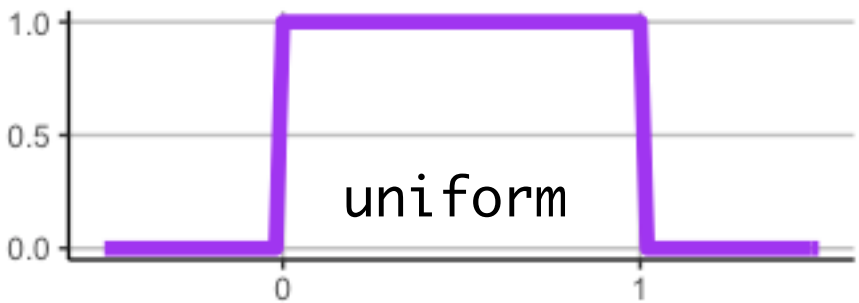
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)



- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)
- `Prob (Prob Bool)` contains random distributions, etc..

Randomization lemma

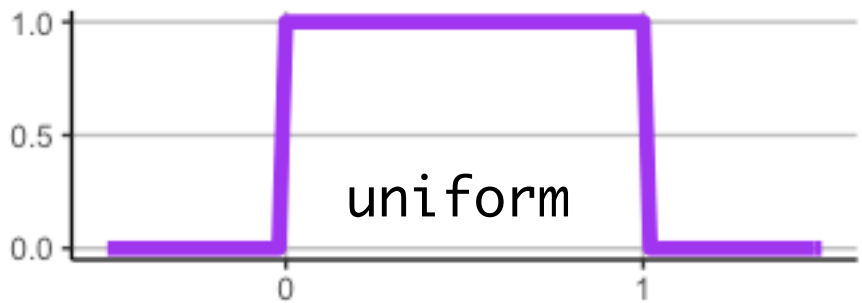
Lemma. One uniform distribution is sufficient to generate all probability measures*.



* with some condition on the underlying spaces, e.g. on standard Borel spaces

Randomization lemma

Lemma. One uniform distribution is sufficient to generate all probability measures*.



For any probability measure
`p :: Prob b` there exists
`α :: RealNum -> b`
such that

$$p \sim f(U(0,1))$$

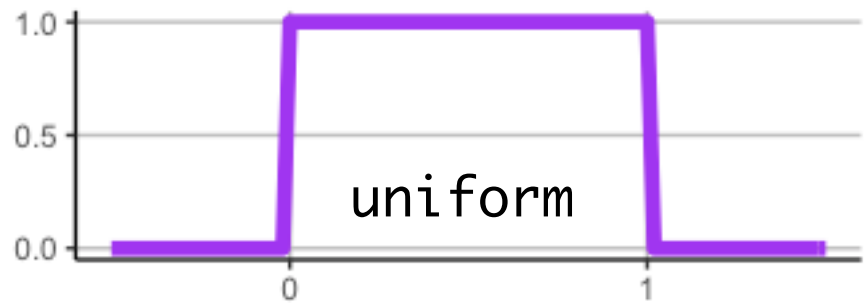
i.e.

```
p = do { r <- uniform ; return (α r) }
```

* with some condition on the underlying spaces, e.g. on standard Borel spaces

Randomization lemma

Lemma. One uniform distribution is sufficient to generate all probability measures*.



For any *parameterized* probability measure
`p :: a -> Prob b` there exists
`α :: a -> RealNum -> b`
such that

$$p(x) \sim f(x, U(0,1))$$

i.e.

```
p x = do { r <- uniform ; return (α x r) }
```

* with some condition on the underlying spaces, e.g. on standard Borel spaces

Randomization lemma

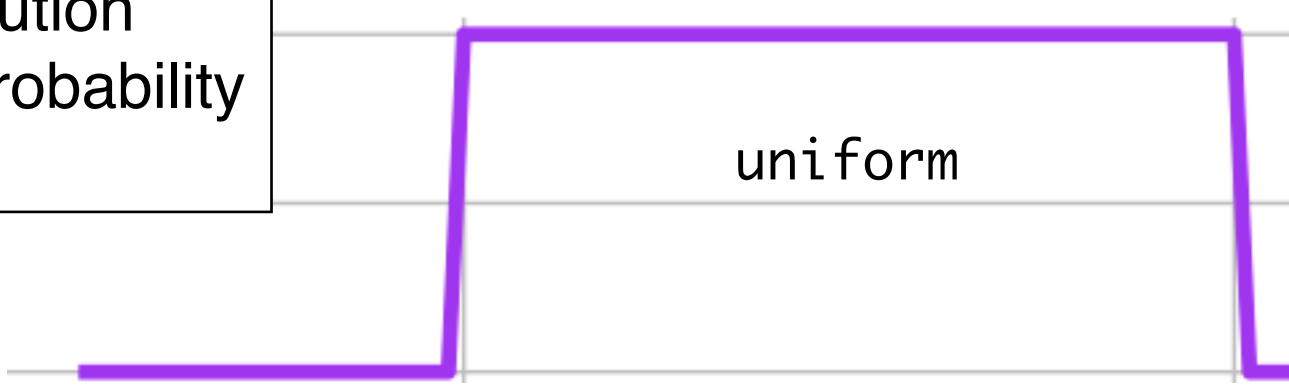
```
linear :: Prob (RealNum -> RealNum)
linear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

Lemma. One uniform distribution is sufficient to generate all probability measures*.

Randomization lemma

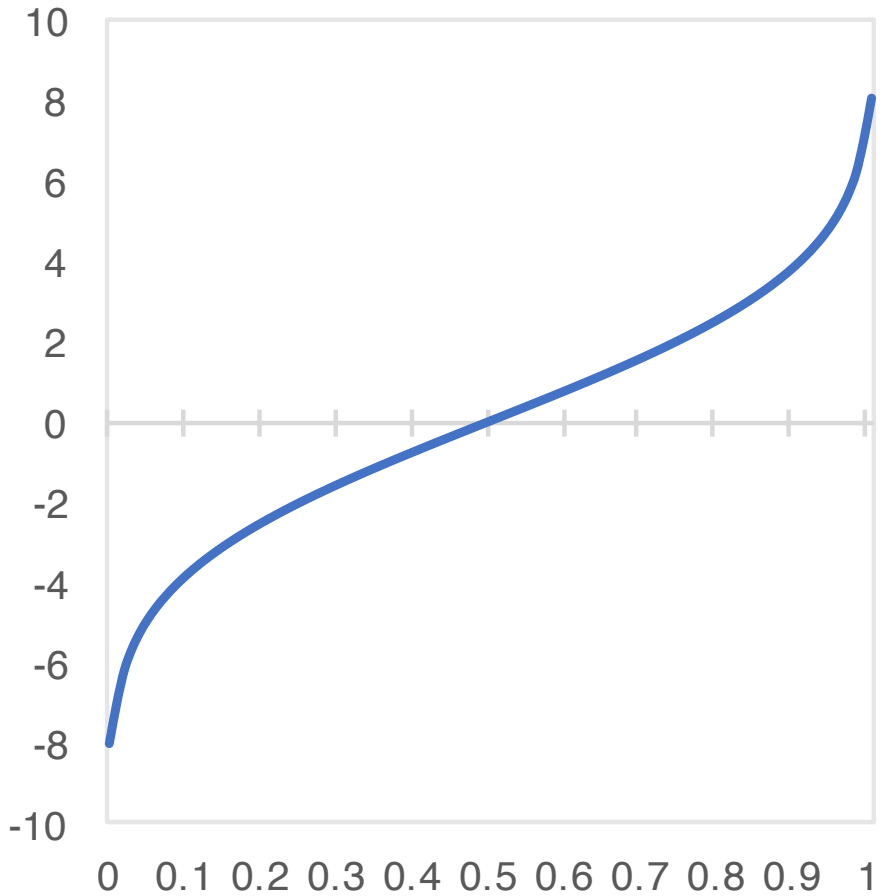
```
linear :: Prob (RealNum -> RealNum)
linear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

Lemma. One uniform distribution is sufficient to generate all probability measures*.

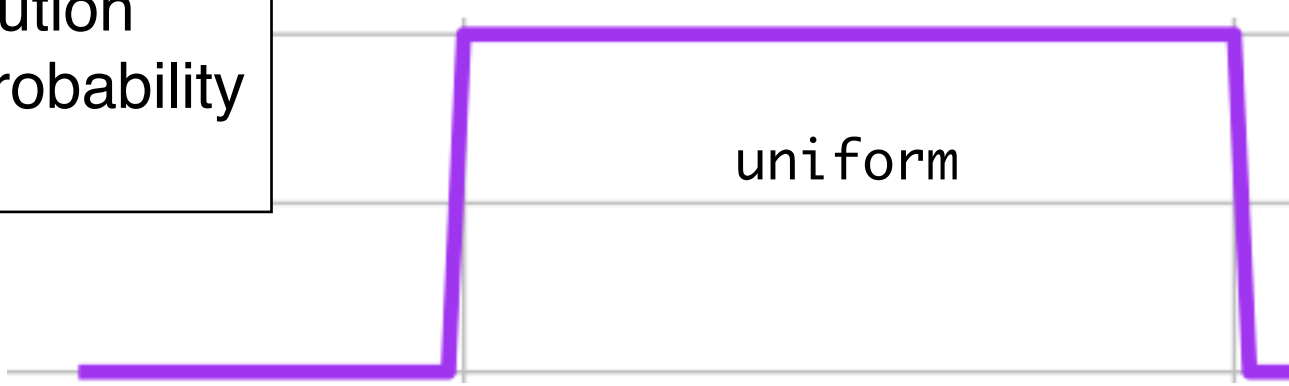


Randomization lemma

```
linear :: Prob (RealNum -> RealNum)
linear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

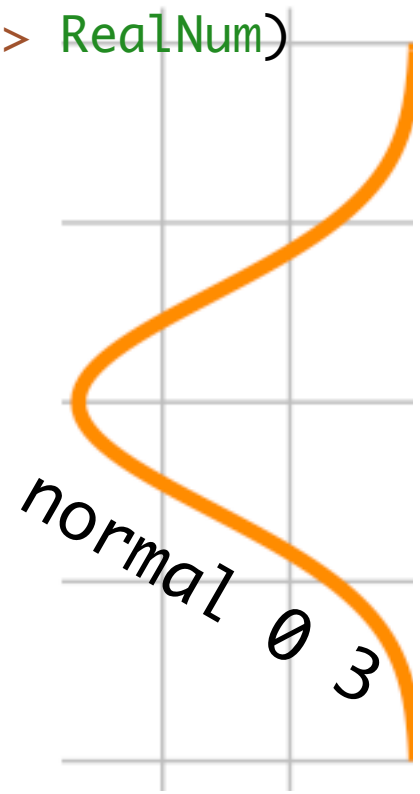


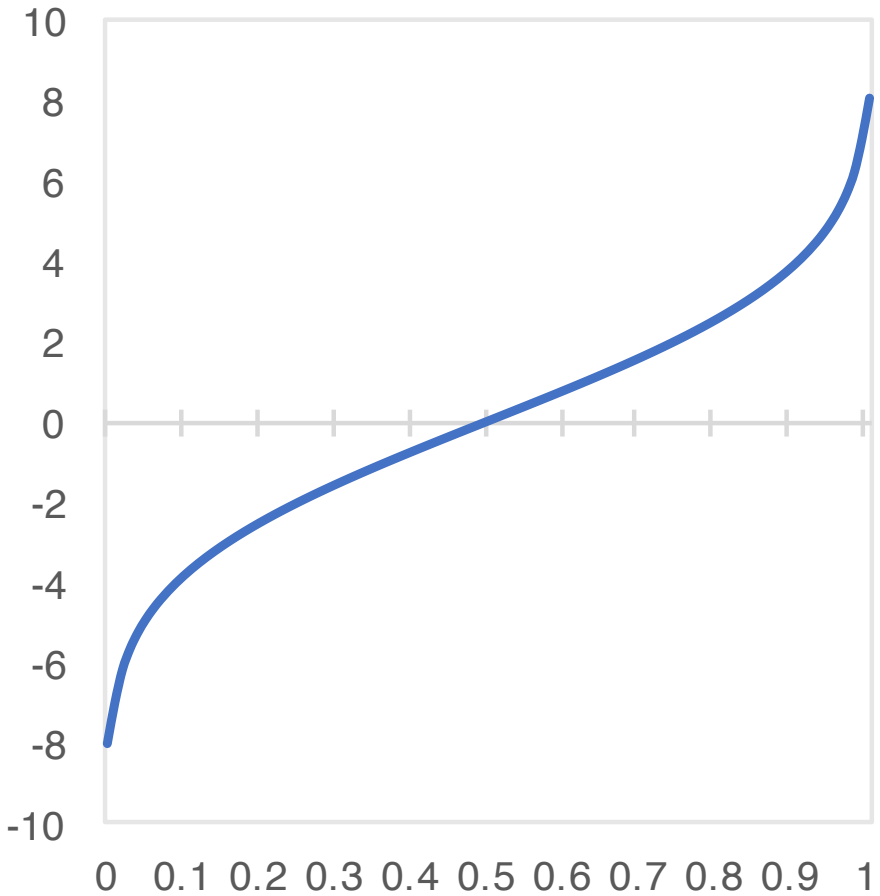
Lemma. One uniform distribution is sufficient to generate all probability measures*.



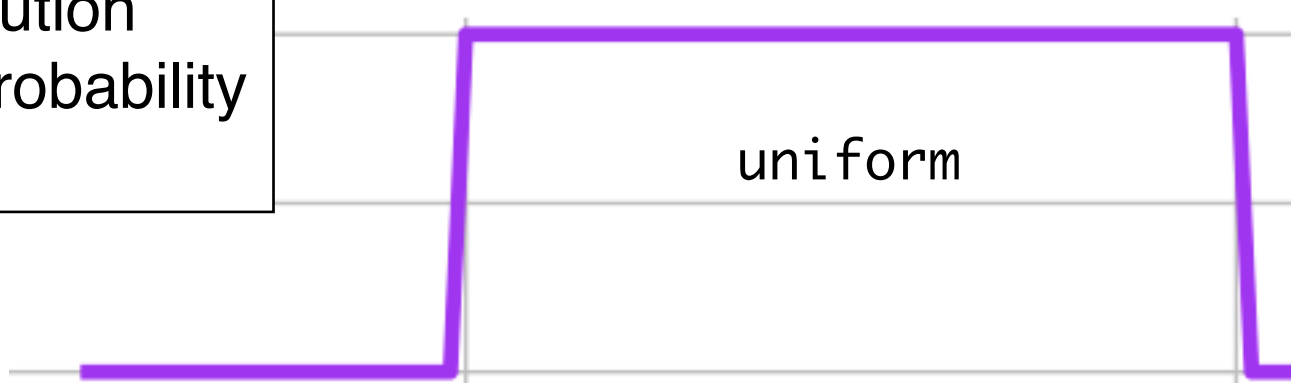
Randomization lemma

```
linear :: Prob (RealNum -> RealNum)
linear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```





Lemma. One uniform distribution is sufficient to generate all probability measures*.

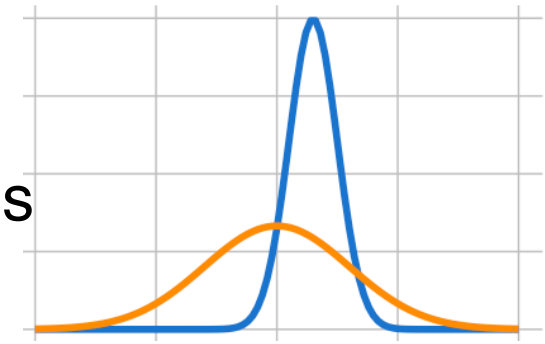


A semantic model:

Quasi-Borel spaces

Heunen, Kammar, Staton, Yang, LICS 2017

There's a type constructor `Prob` (a monad), and…



- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `normal 1.5 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)



- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)
- `Prob (Prob Bool)` contains random distributions, etc..

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Lemma. One uniform distribution is sufficient to generate all probability measures*.

```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y \qquad \mathbb{R} \xrightarrow{\alpha} X \xrightarrow{f} Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

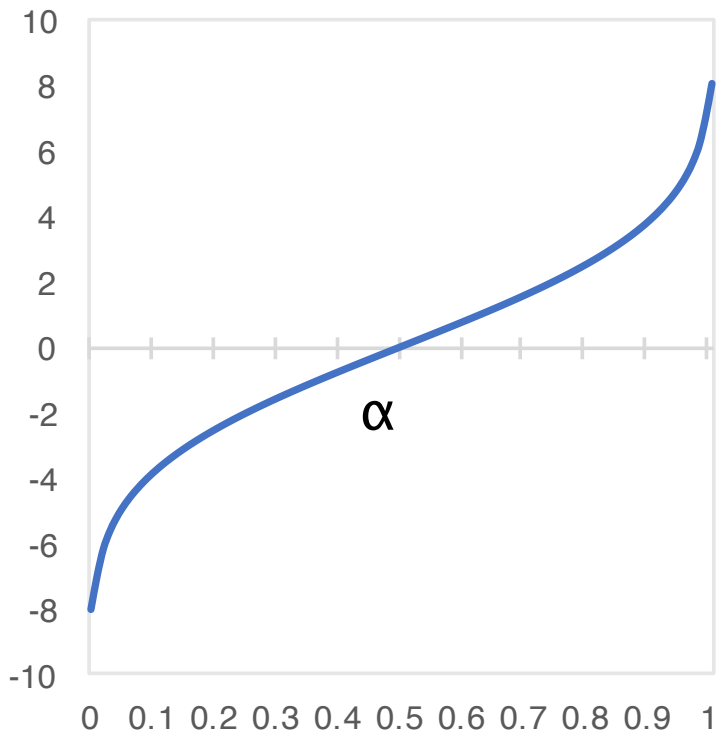
Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.




```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

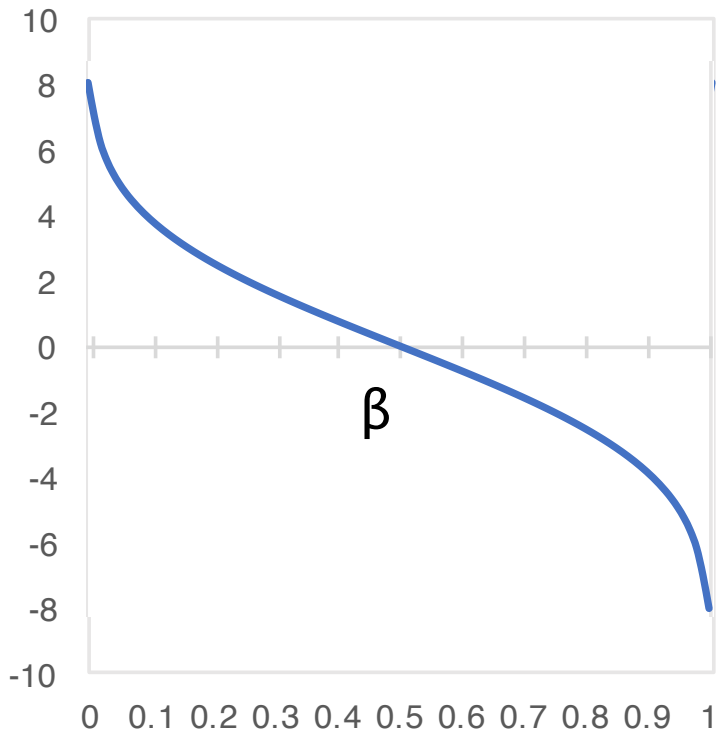
Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.




```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

```
do { r <- uniform ; return (α r) }
```

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

The qBs of reals $(\mathbb{R}, M_{\mathbb{R}})$ has $M_{\mathbb{R}} \subseteq [\mathbb{R} \to \mathbb{R}]$ as the Borel functions.

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

The qBs of reals $(\mathbb{R}, M_{\mathbb{R}})$ has $M_{\mathbb{R}} \subseteq [\mathbb{R} \to \mathbb{R}]$ as the Borel functions.

Two measures are equal, $\alpha \sim \beta$, if for all morphisms $k : X \to \mathbb{R}$,

$$\int k(\alpha(r))\,\mathrm{d}r = \int k(\beta(r))\,\mathrm{d}r.$$

$$\mathbb{R} \underset{\beta}{\overset{\alpha}{\rightrightarrows}} X \overset{\forall k}{\to} \mathbb{R}$$

```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

The qBs of reals $(\mathbb{R}, M_{\mathbb{R}})$ has $M_{\mathbb{R}} \subseteq [\mathbb{R} \to \mathbb{R}]$ as the Borel functions.

In other words,

$$Pr(X) \subseteq ((X \to \mathbb{R}) \to \mathbb{R})$$

$$Pr(X) = \left\{ \lambda k \,.\, \int k(\alpha(x)) \;\middle|\; \alpha \in M_X \right\}.$$

$$\mathbb{R} \xrightarrow{\alpha} X \xrightarrow{k} \mathbb{R}$$

```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Types : quasi-Borel spaces.

$$[\![\texttt{RealNum}]\!] = \mathbb{R}$$

$$[\![\texttt{Prob a}]\!] = Pr([\![\texttt{a}]\!])$$

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

The qBs of reals $(\mathbb{R}, M_{\mathbb{R}})$ has $M_{\mathbb{R}} \subseteq [\mathbb{R} \to \mathbb{R}]$ as the Borel functions.

In other words,

$$Pr(X) \subseteq ((X \to \mathbb{R}) \to \mathbb{R})$$

$$Pr(X) = \left\{ \lambda k \,.\, \int k(\alpha(x)) \;\middle|\; \alpha \in M_X \right\}.$$

$$\mathbb{R} \xrightarrow{\alpha} X \xrightarrow{k} \mathbb{R}$$

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Note: The lazyppl implementation is also based on randomization, but we use $\mathfrak{T} = \mathbb{R}^{(\omega^\omega)}$ so that $\mathfrak{T} \cong \mathbb{R} \times \mathfrak{T}^\omega$, but also $\mathfrak{T} \cong \mathbb{R}$.

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

The qBs of reals $(\mathbb{R}, M_\mathbb{R})$ has $M_\mathbb{R} \subseteq [\mathbb{R} \to \mathbb{R}]$ as the Borel functions.

In other words,

$$Pr(X) \subseteq ((X \to \mathbb{R}) \to \mathbb{R})$$

$$Pr(X) = \left\{ \lambda k \,.\, \int k(\alpha(x)) \;\middle|\; \alpha \in M_X \right\}.$$

$$\mathbb{R} \xrightarrow{\alpha} X \xrightarrow{k} \mathbb{R}$$

Desiderata for a theory of Prob

Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3          do b <- normal 0 3
     b <- normal 0 3             a <- normal 0 3
     let f x = a*x + b           let f x = a*x + b
     return f                    return f
```

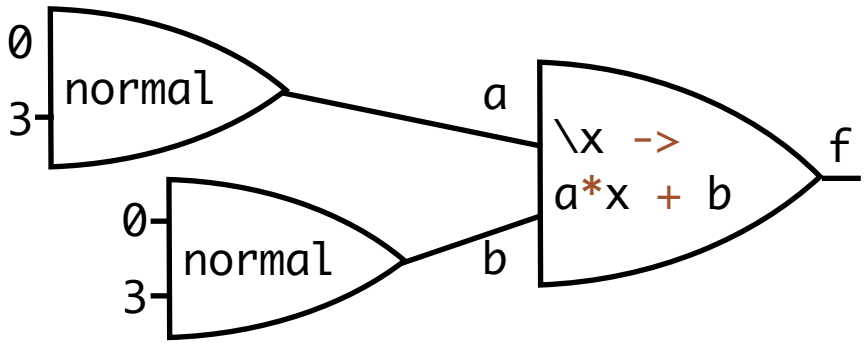
Dataflow property:

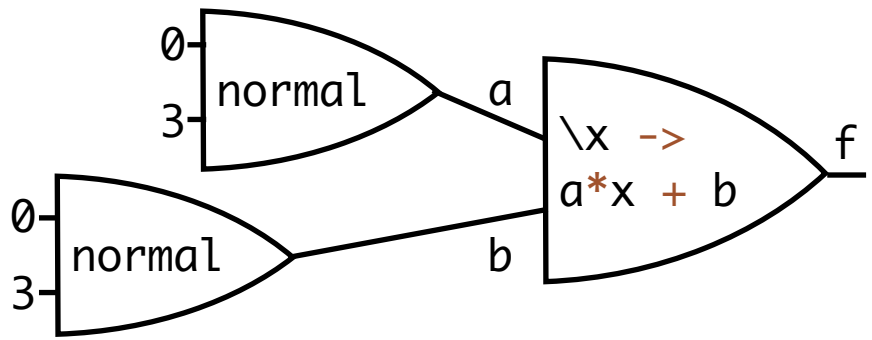
*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

```
  do b <- normal 0 3
     a <- normal 0 3
     let f x = a*x + b
     return f
```





Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

```
do b <- normal 0 3
   a <- normal 0 3
   let f x = a*x + b
   return f
```

```
do c <- normal 0 3
   b <- normal 0 3
   a <- normal 0 3
   let f x = a*x + b
   return f
```

Dataflow property:

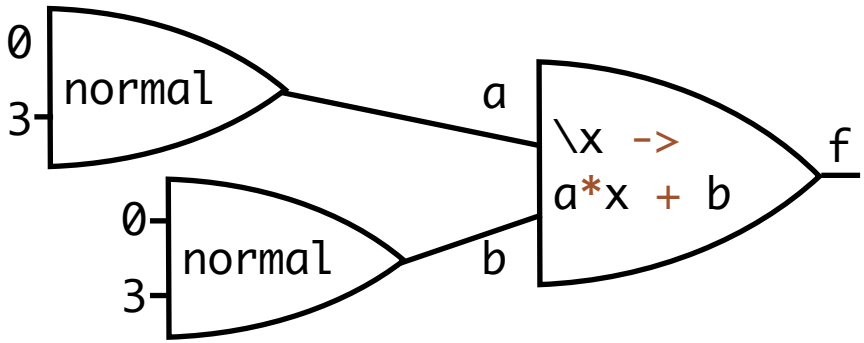
*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

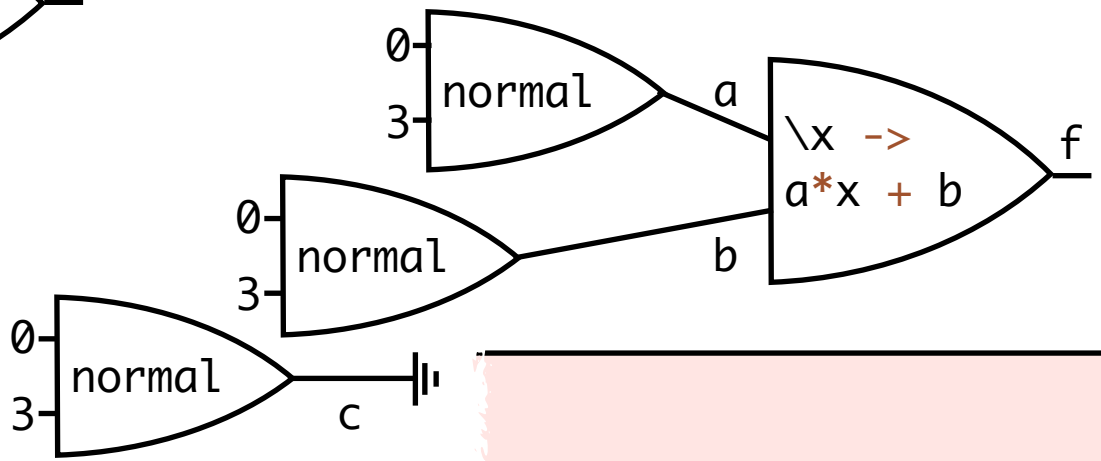
Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

```
do b <- normal 0 3
   a <- normal 0 3
   let f x = a*x + b
   return f
```

```
do c <- normal 0 3
   b <- normal 0 3
   a <- normal 0 3
   let f x = a*x + b
   return f
```





Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

```
do b <- normal 0 3
   a <- normal 0 3
   let f x = a*x + b
   return f
```

$$\int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}b\,\mathrm{d}a \qquad \int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}a\,\mathrm{d}b$$

Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Related to Fubini's theorem.

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

```
do b <- normal 0 3
   a <- normal 0 3
   let f x = a*x + b
   return f
```

```
do c <- normal 0 3
   b <- normal 0 3
   a <- normal 0 3
   let f x = a*x + b
   return f
```

$$\int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}b\,\mathrm{d}a \qquad \int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}a\,\mathrm{d}b$$

$$\int\int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}a\,\mathrm{d}b\,\mathrm{d}c$$

Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Related to Fubini's theorem.

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- norma
     b <- norma
     let f x =
     return f
```

```
do c <- normal 0 3
   normal 0 3
   normal 0 3
    x = a*x + b
   n f
```

Theorem. The quasi-Borel space model satisfies the dataflow property.

Heunen, Kammar, Staton, Yang, LICS 2017

$$\int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}b\,\mathrm{d}a \qquad \int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}a\,\mathrm{d}b$$

$$\int\int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}a\,\mathrm{d}b\,\mathrm{d}c$$

Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Related to Fubini's theorem.

Desiderata for a theory of Prob

Theorem. The quasi-Borel space model satisfies the dataflow property.

- The probability monad is commutative and affine. cf Kock TAC 2012
- The parameterized distributions form a monoidal category cf Fritz Adv Math 2020, Cho & Jacobs MSCS 2019, Stein & Staton LICS 2021

Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Related to Fubini's theorem.

Overview

1. Examples of high level probabilistic programs
   `randlinear, randprog, wiener, regress`
2. First semantic models
   quasi-Borel spaces, Markov categories
3. **Laziness and point processes**
4. Stochastic memoization and ongoing work

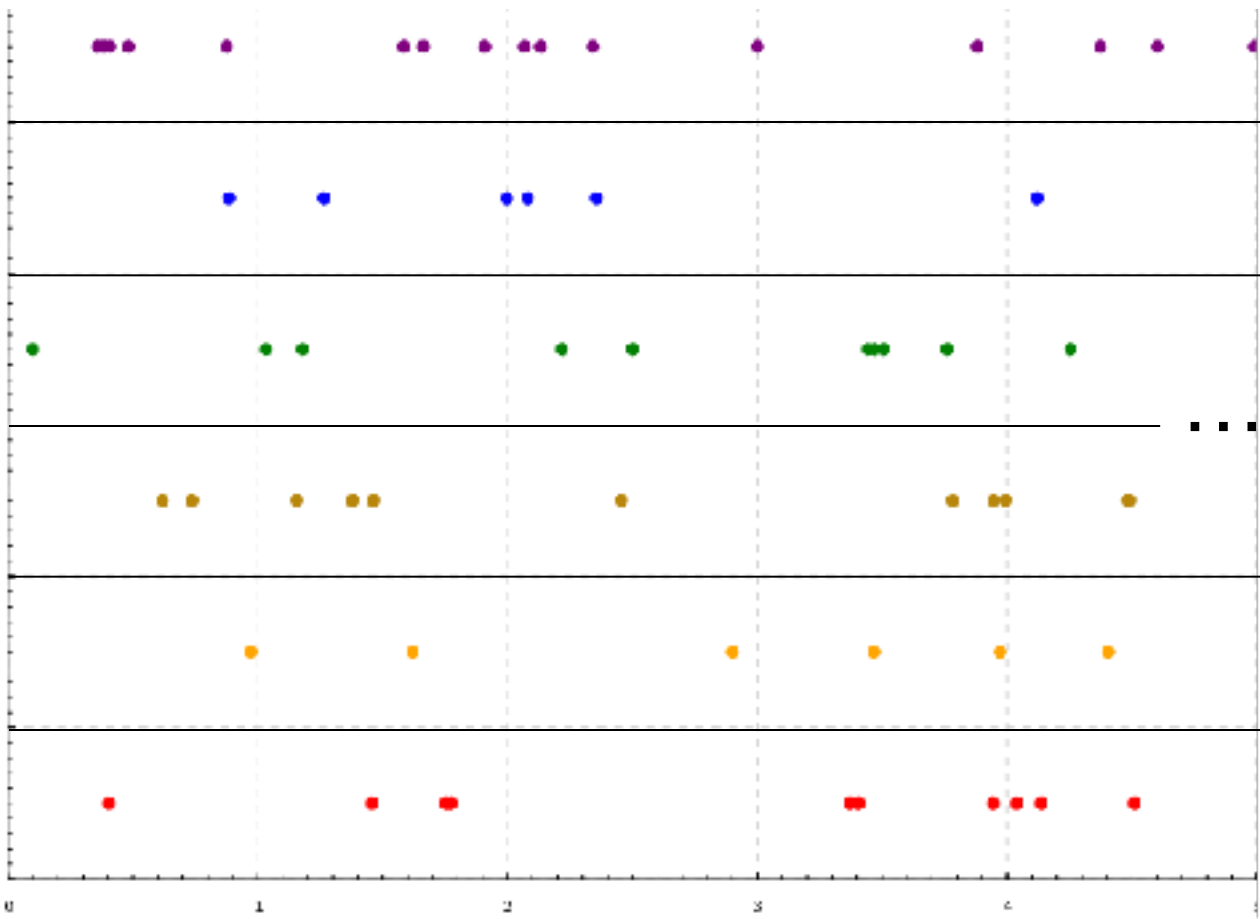
Point processes & laziness

Defn. A *point process* on `a` is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

Point processes & laziness

Defn. A *point process* on a is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

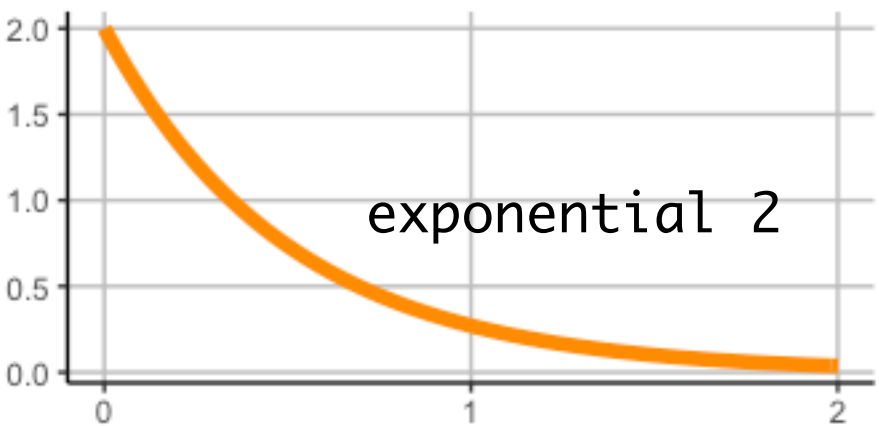
Example. A Poisson point process on $\mathbb{R}^+$:

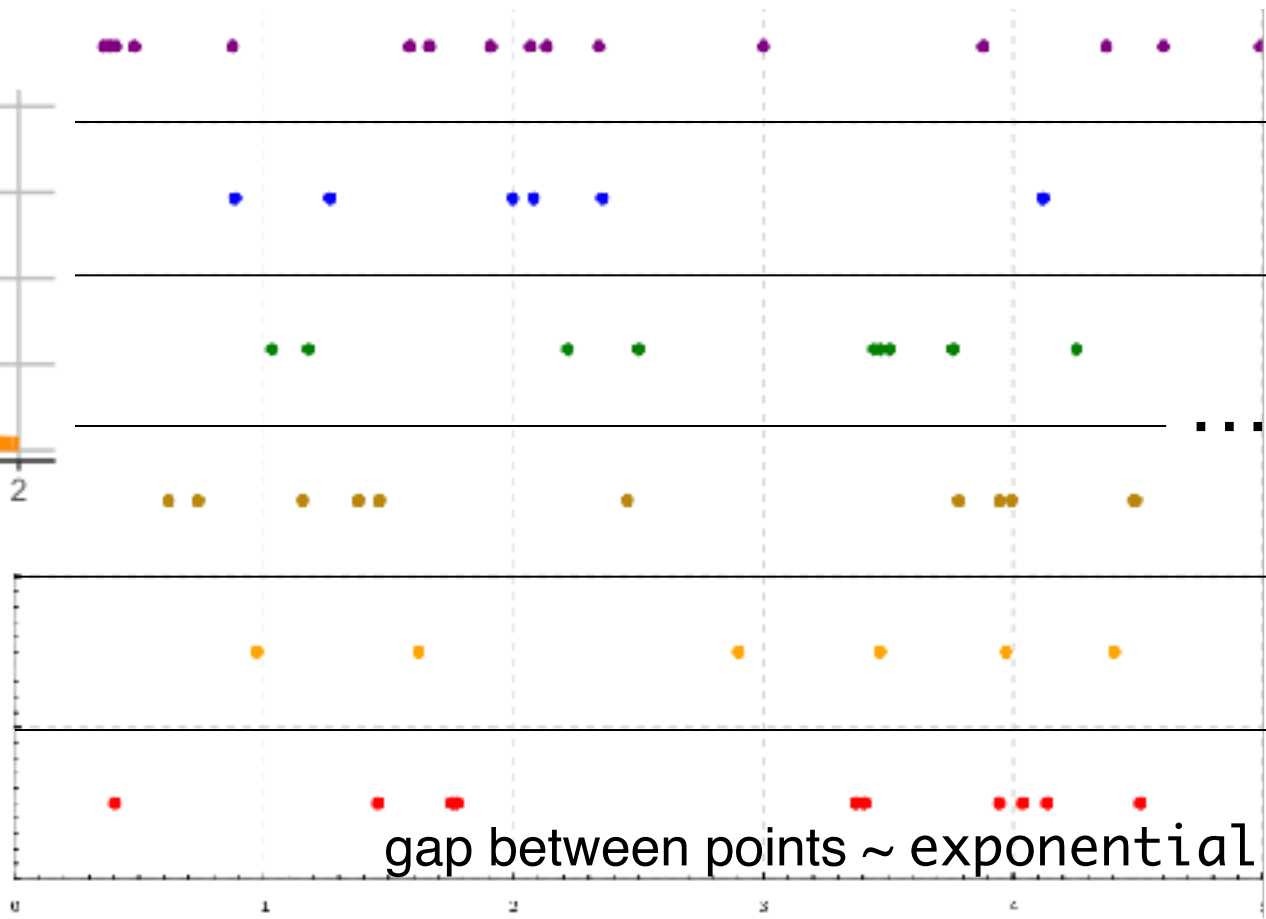


Point processes & laziness

Defn. A *point process* on `a` is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

Example. A Poisson point process on $\mathbb{R}^+$:





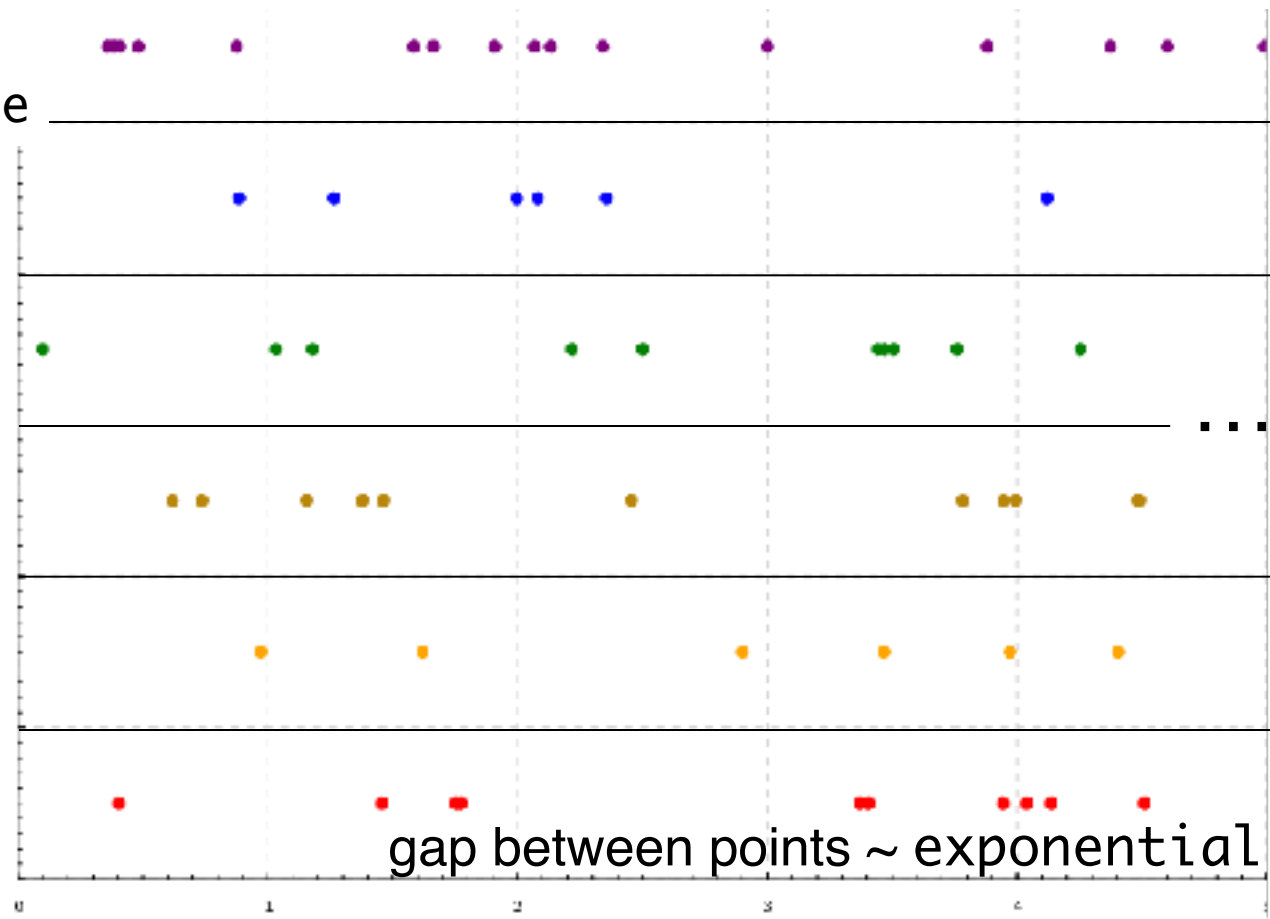
Point processes & laziness

Defn. A *point process* on a is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

Example. A Poisson point process on $\mathbb{R}^+$:

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```

6 samples:



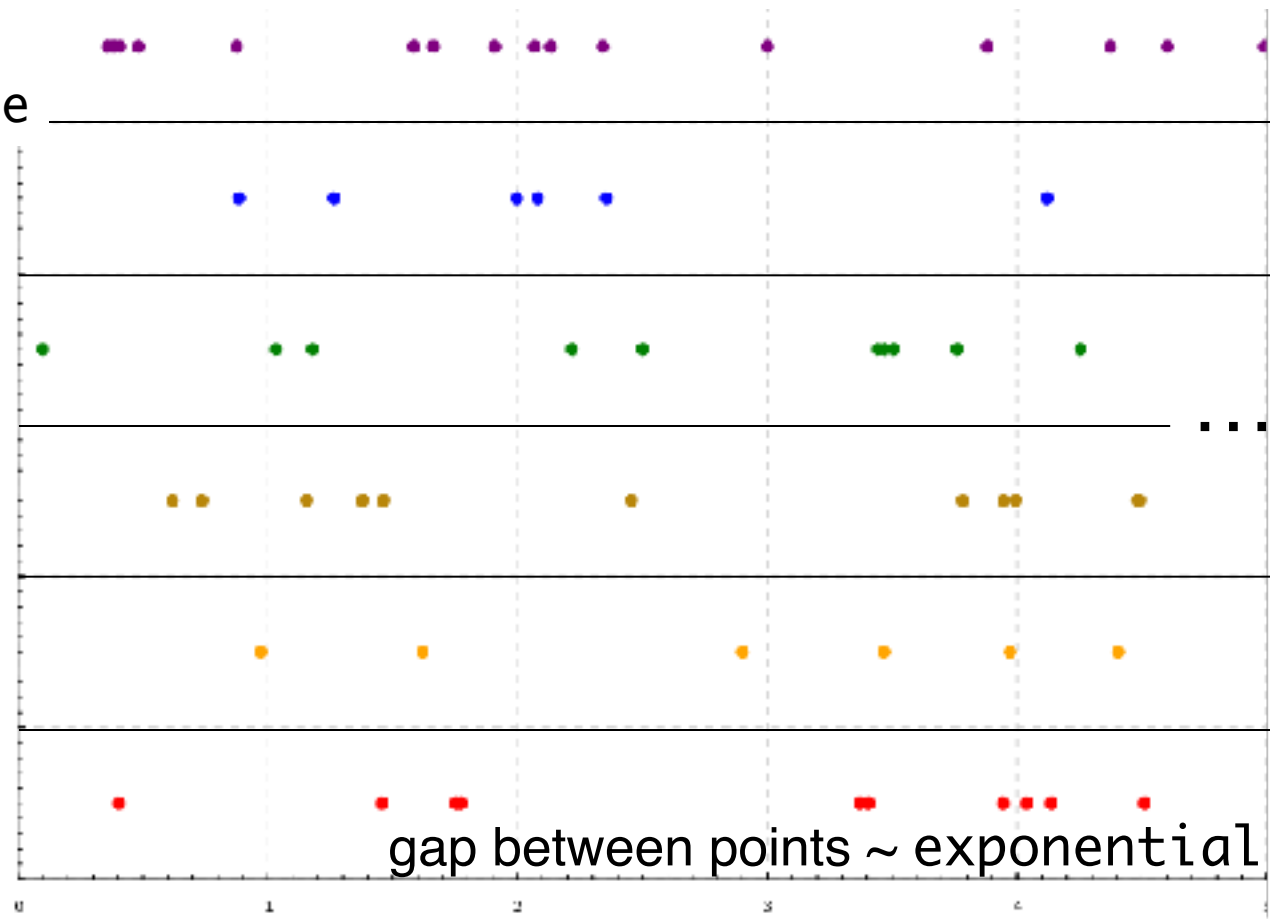
Point processes & laziness

Defn. A *point process* on a is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

Example. A Poisson point process on $\mathbb{R}^+$:

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```

6 samples:



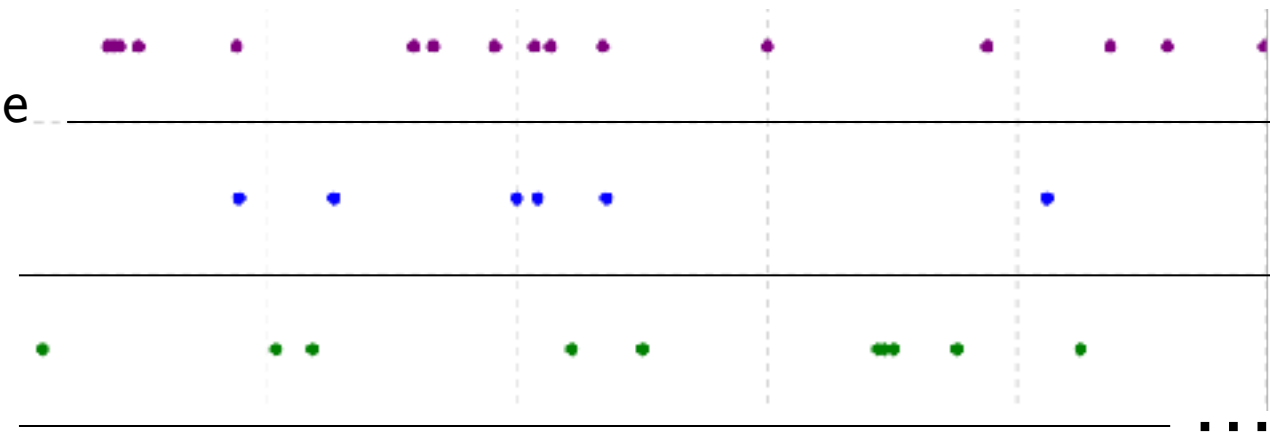
A distribution over
infinite bags,
but lazily evaluated.

Point processes & laziness

Defn. A *point process* on a is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

Example. A Poisson point process on $\mathbb{R}^+$:

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```



Point processes arise everywhere across stats in science and ML.

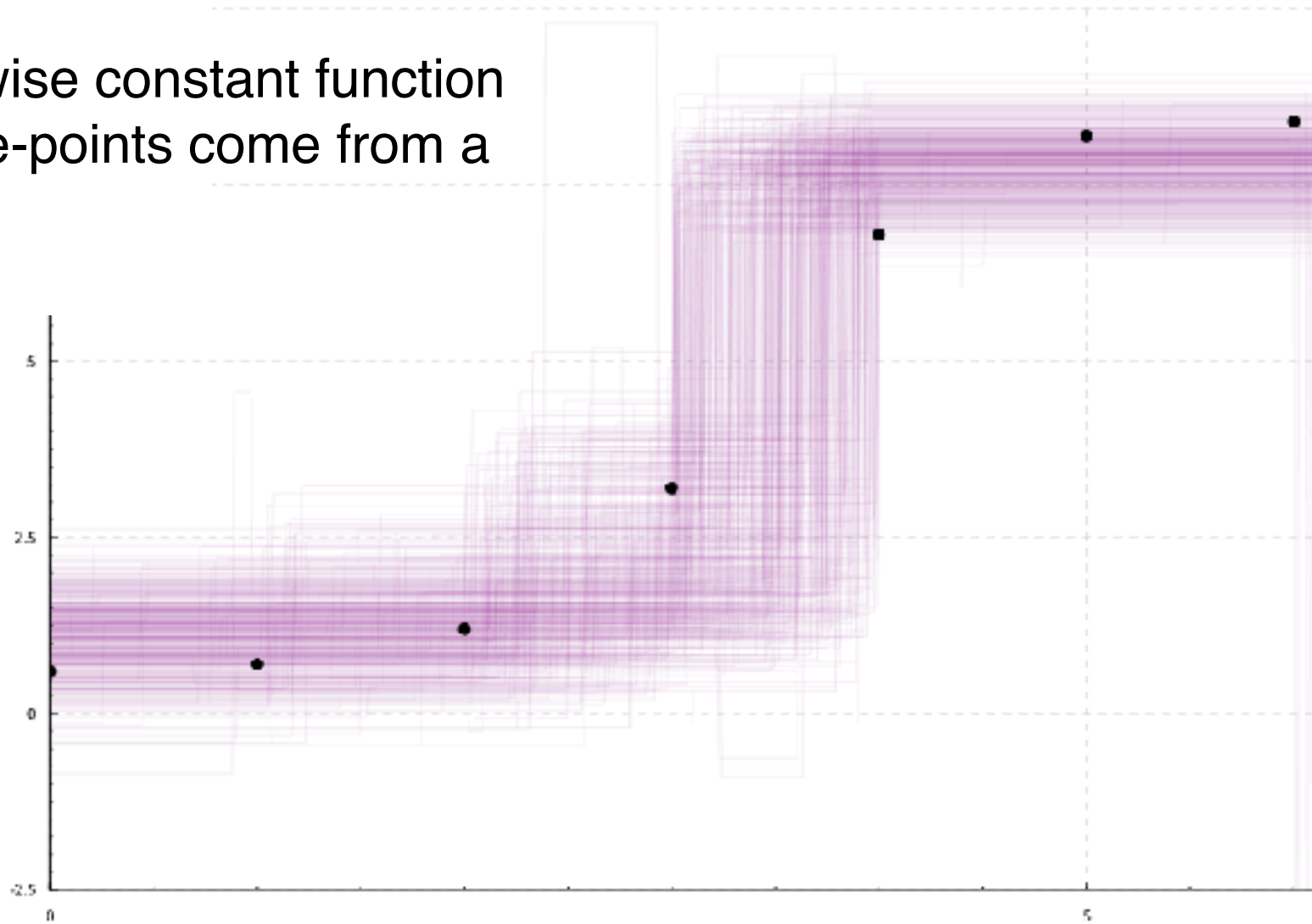
Also: point processes = generalized probabilistic databases.

Grohe, Lindner. PODS 2019.
Dash, Staton. ACT 2020.

Piecewise constant regression

Defn. A *point process* on `a` is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

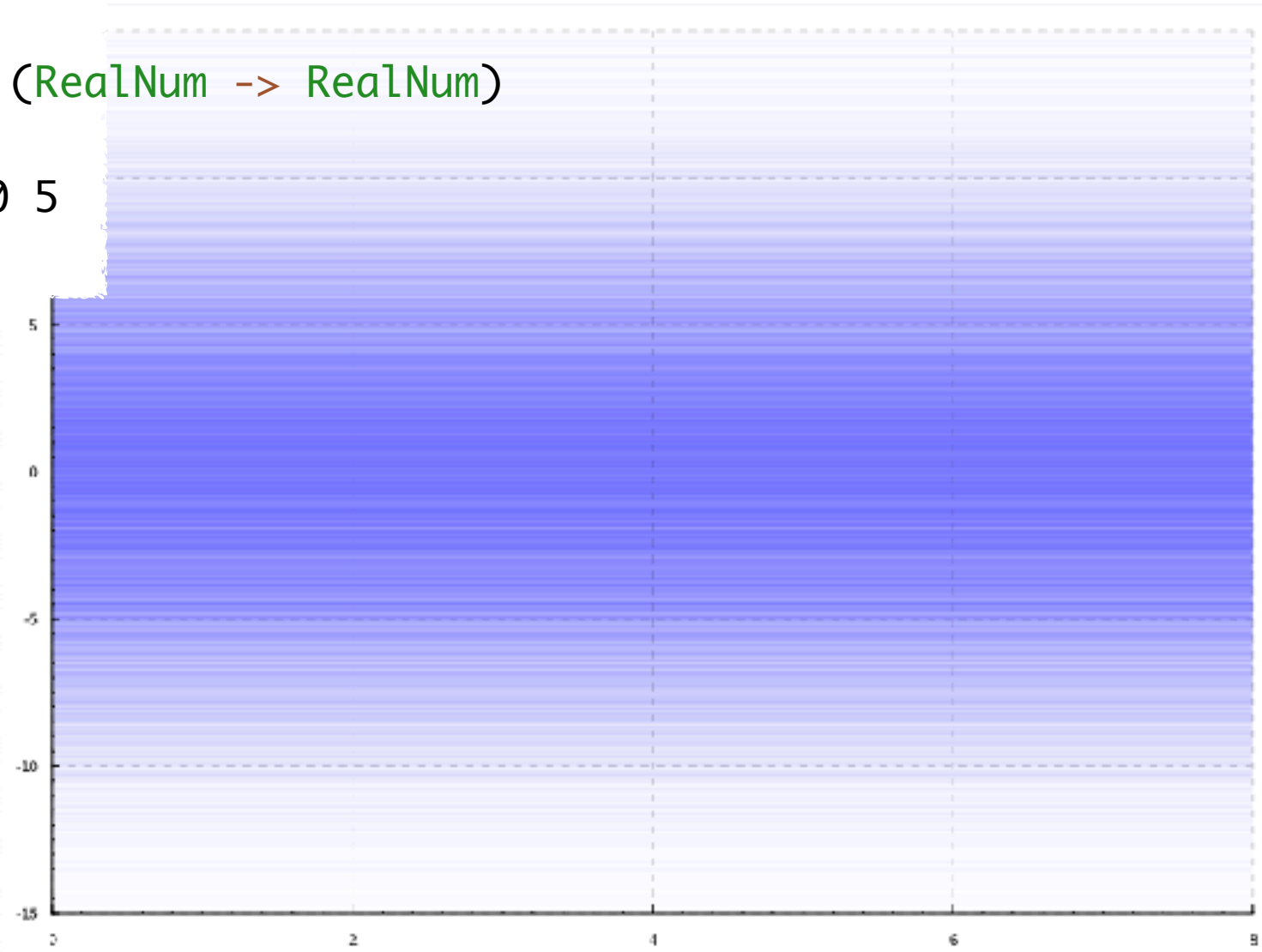
Idea: Fit a piecewise constant function where the change-points come from a point process.



Piecewise constant regression

Defn. A *point process* on a is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

```
randconst :: Prob (RealNum -> RealNum)
randconst =
  do a <- normal 0 5
     let f x = a
     return f
```



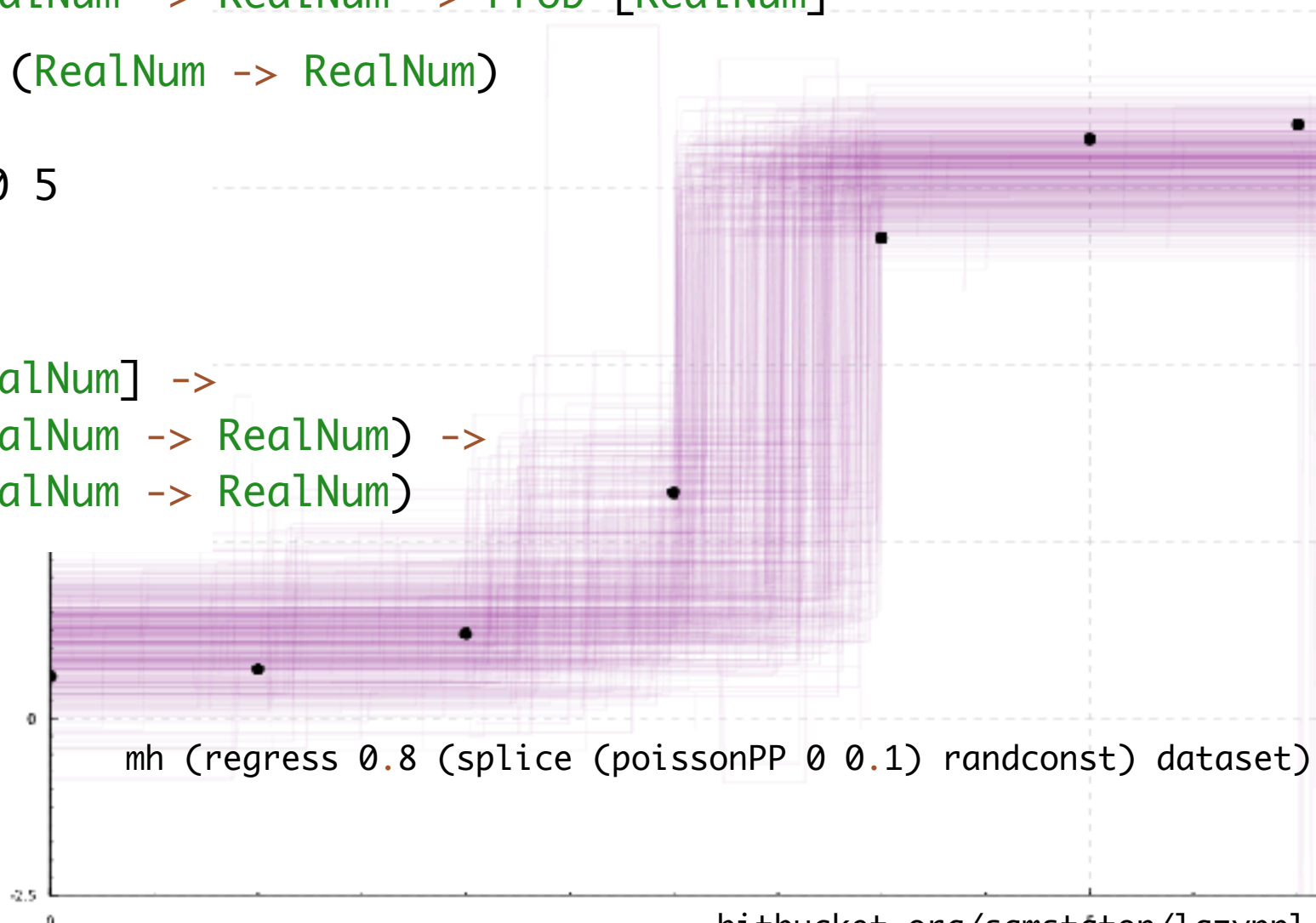
Piecewise constant regression

Defn. A *point process* on a is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

e.g. `poissonPP :: RealNum -> RealNum -> Prob [RealNum]`

```
randconst :: Prob (RealNum -> RealNum)
randconst =
  do a <- normal 0 5
     let f x = a
     return f

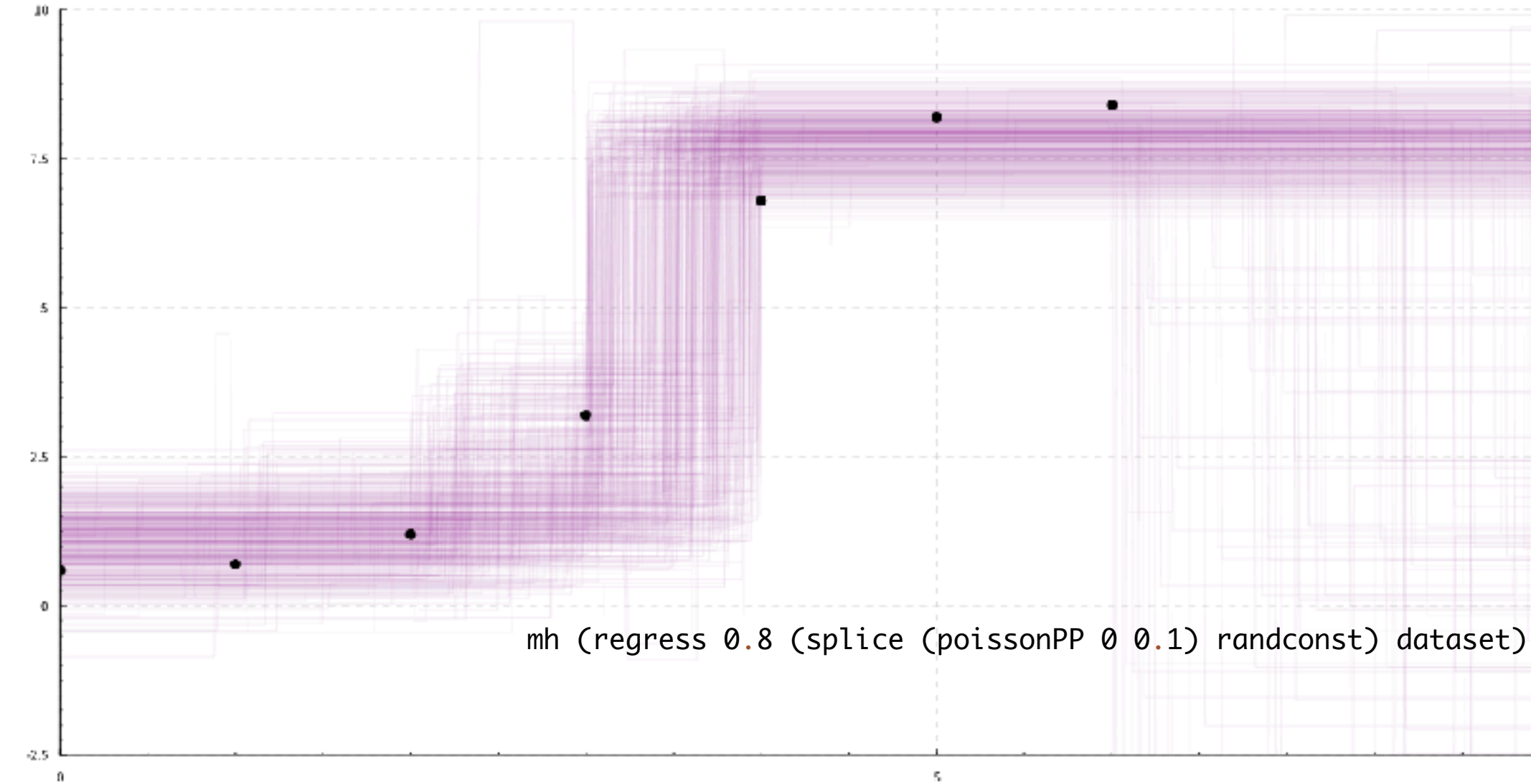
splice :: Prob [RealNum] ->
          Prob (RealNum -> RealNum) ->
          Prob (RealNum -> RealNum)
```



bitbucket.org/samstaton/lazyppl

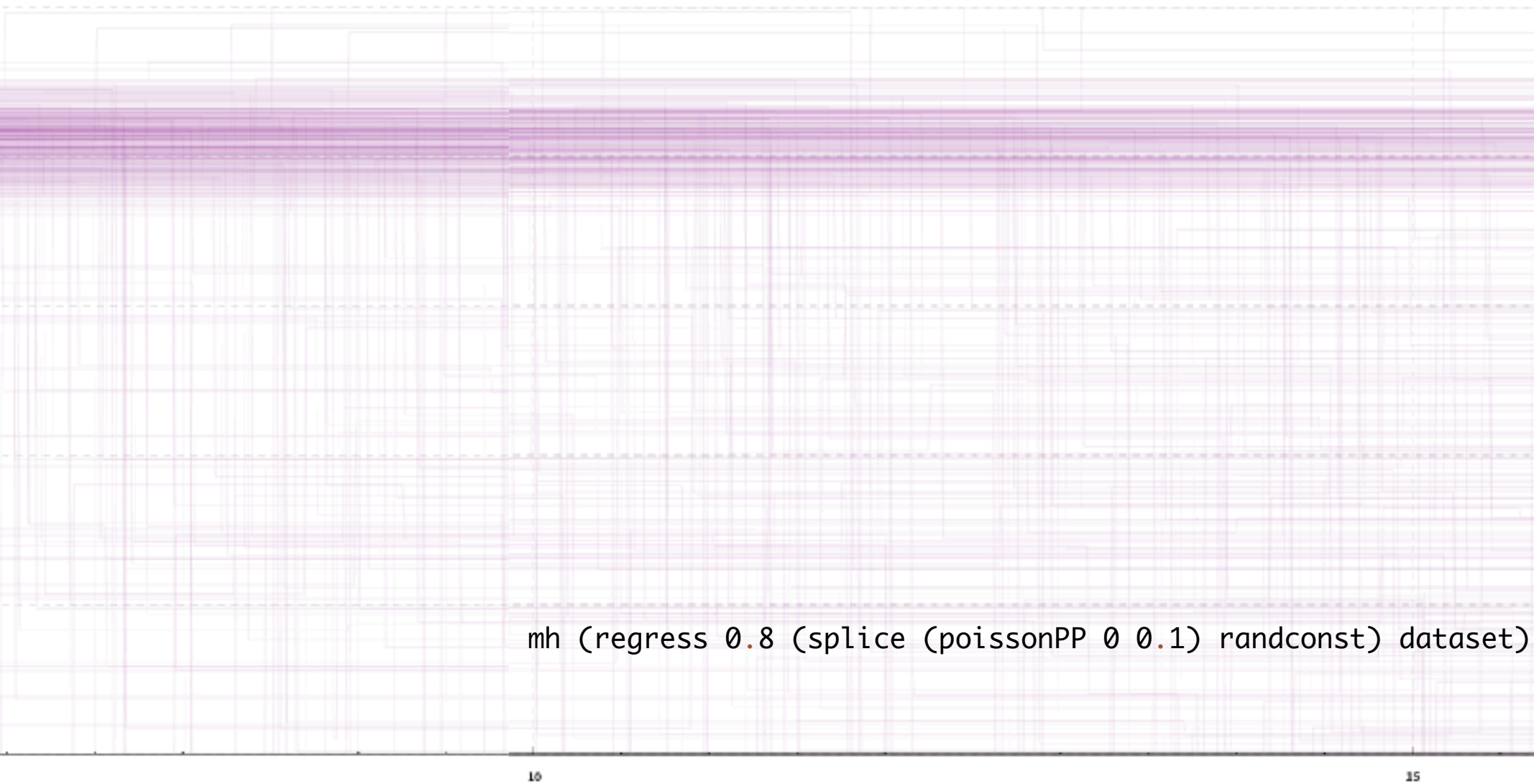
Piecewise constant regression

Haskell is lazy, so the simulation only looks at the viewport.
There is no infinite computation nor any explicit truncation.



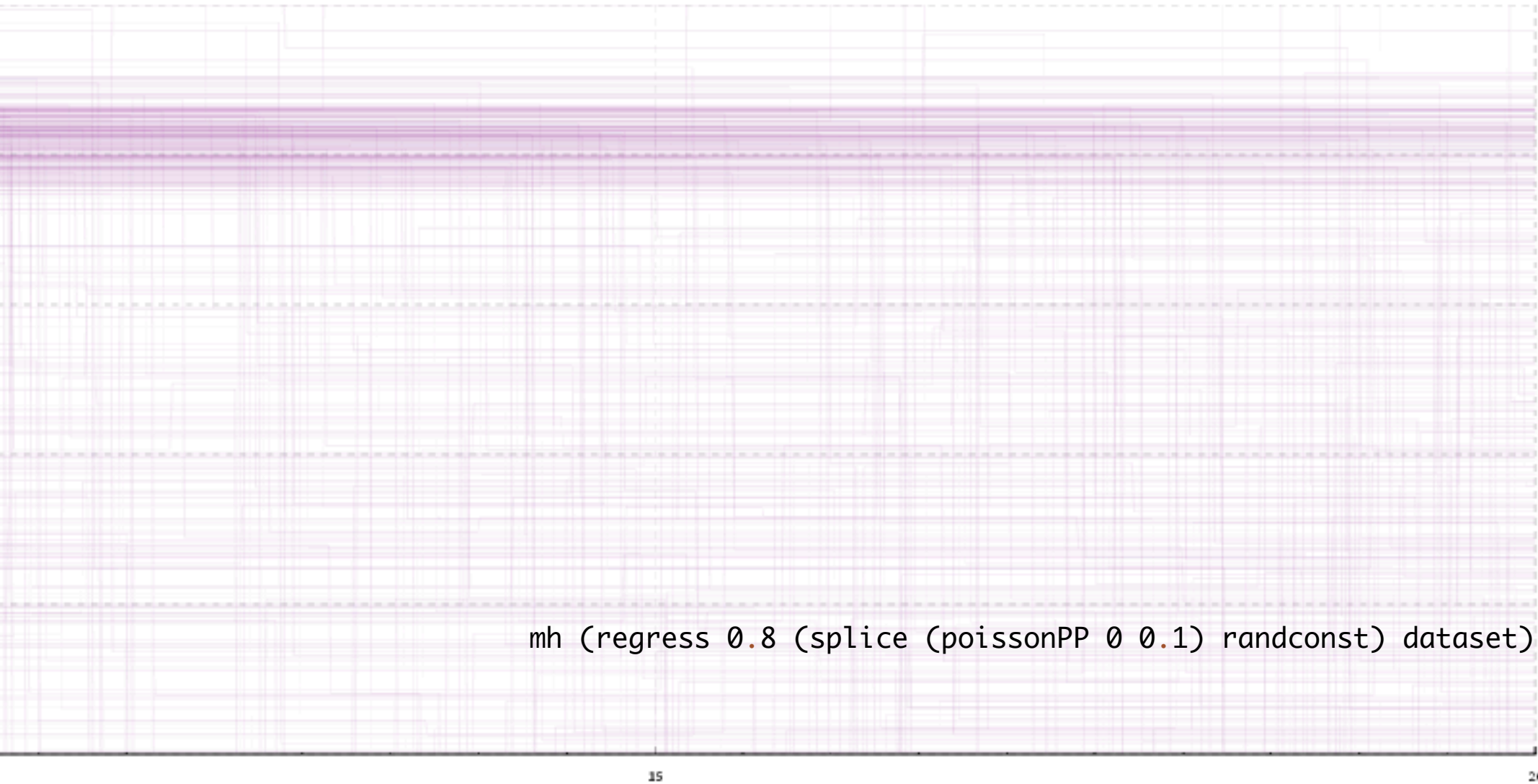
Piecewise constant regression

Haskell is lazy, so the simulation only looks at the viewport.
There is no infinite computation nor any explicit truncation.



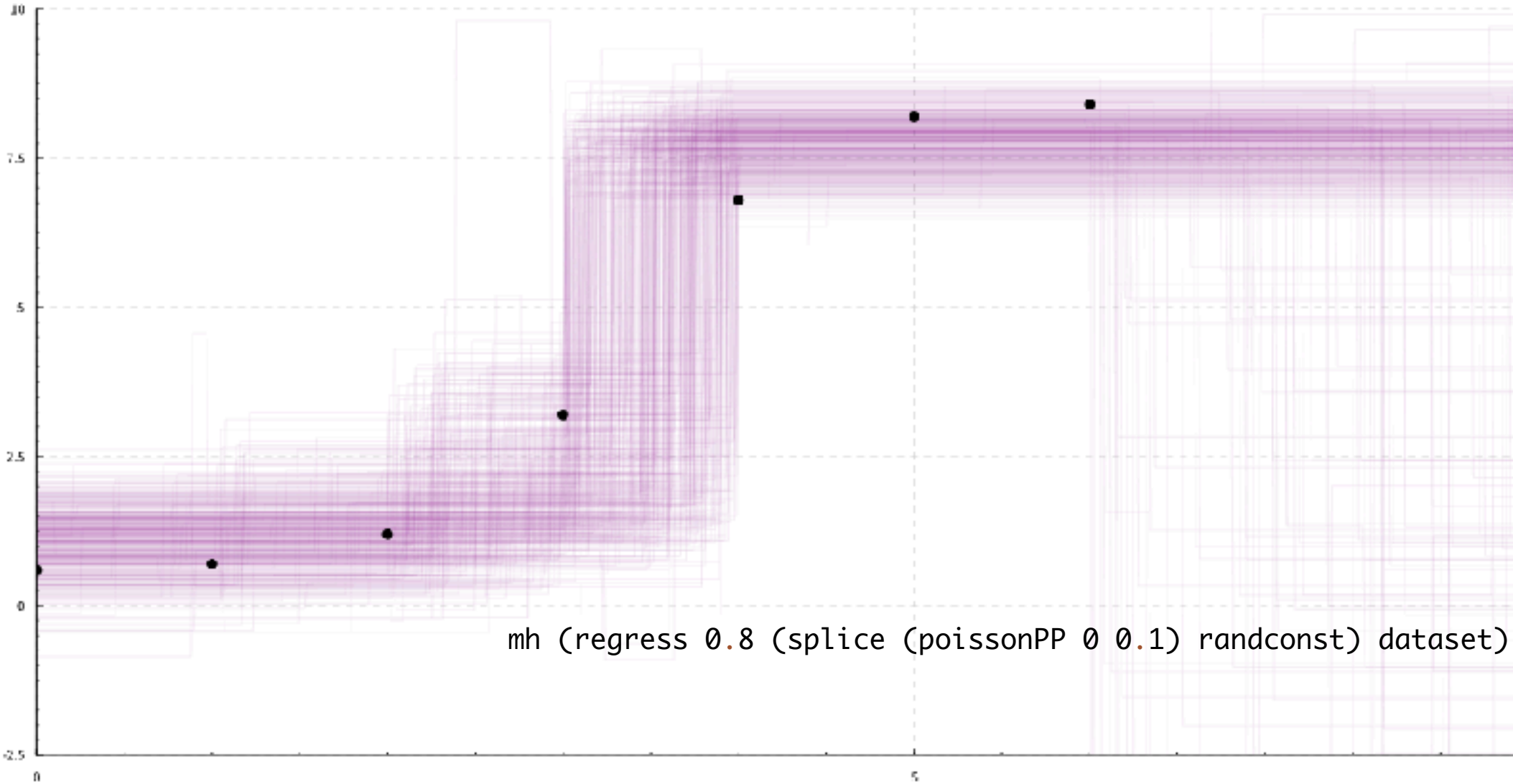
Piecewise constant regression

Haskell is lazy, so the simulation only looks at the viewport.
There is no infinite computation nor any explicit truncation.



Piecewise constant regression

Haskell is lazy, so the simulation only looks at the viewport.
There is no infinite computation nor any explicit truncation.



Overview

1. Examples of high level probabilistic programs
   `randlinear, randprog, wiener, regress`
2. First semantic models
   quasi-Borel spaces, Markov categories
3. Laziness and point processes
   Poisson pt processes via laziness, piecewise regression
4. **Stochastic memoization and ongoing work**

Memoization



Memo functions & machine learning
Brian Milchie, Nature 1968

Idea: a function
tables the results of prior calls
only computes for fresh arguments.

`square 57`

x	square x
57	3249



3249

Memoization



Memo functions & machine learning
Brian Milchie, Nature 1968

Idea: a function
tables the results of prior calls
only computes for fresh arguments.

`square 3`

x	square x
57	3249
3	9

 9

Memoization



Memo functions & machine learning
Brian Milchie, Nature 1968

Idea: a function
tables the results of prior calls
only computes for fresh arguments.

`square 57`

x	square x
57	3249
3	9

🤔 3249

Memoization



Memo functions & machine learning
Brian Milchie, Nature 1968

Idea: a function
tables the results of prior calls
only computes for fresh arguments.

`square 42`

x	square x
57	3249
3	9
42	1764

 1764

Memoization



Memo functions & machine learning
Brian Milchie, Nature 1968

Idea: a function
tables the results of prior calls
only computes for fresh arguments.

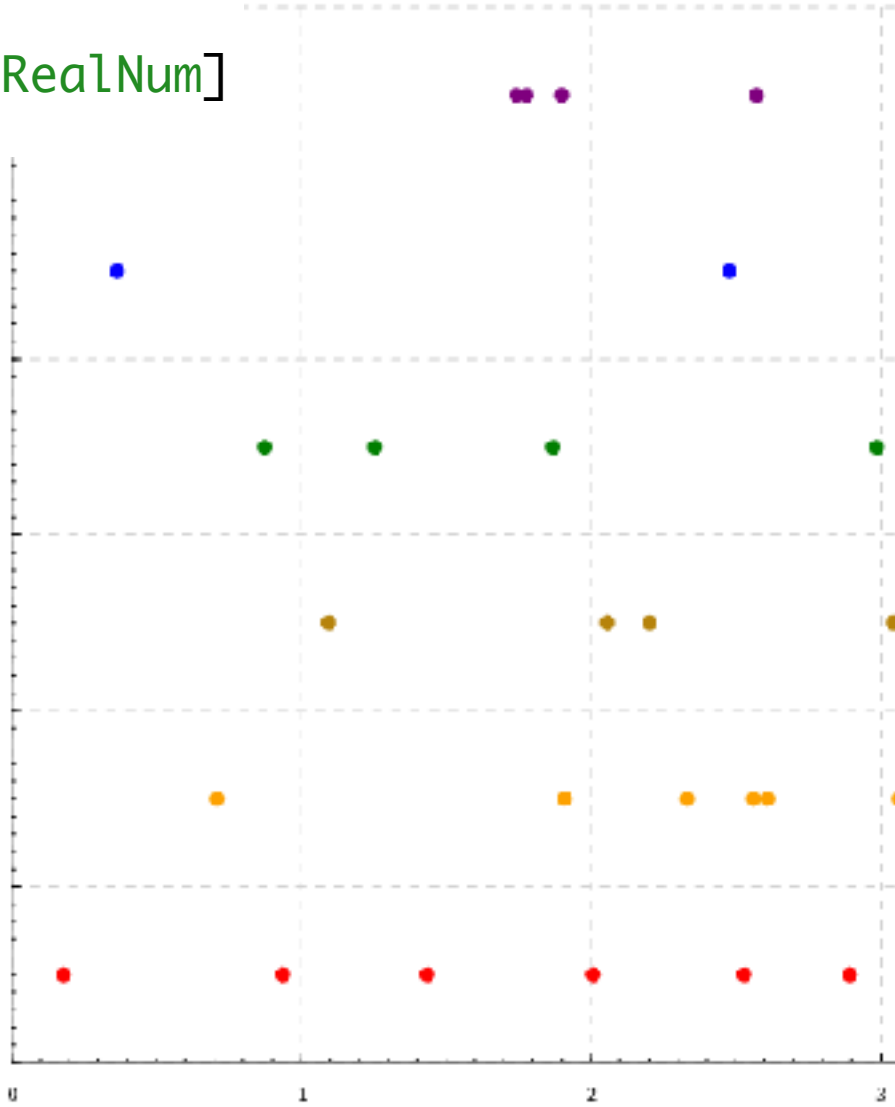
- A variation on laziness
- Over pure functions it's just a speed-up like 'rote-learning'.
- With functions involving probability, memoization *changes semantics.*
  A key way of building infinite dimensional structures.

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Wood, Archambeau, Gasthaus, James, Teh. ICML 2009.

Photo by Peter Mowforth, CCBYSA 3

Laziness to Memoization

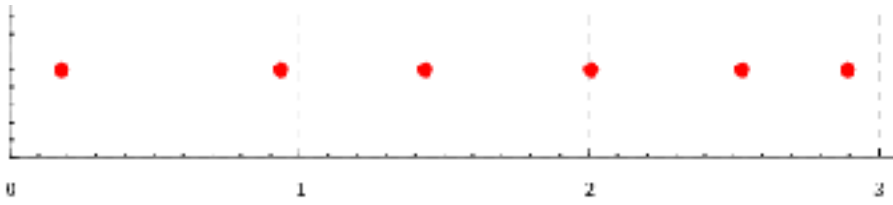
```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```



Laziness to Memoization

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps
```



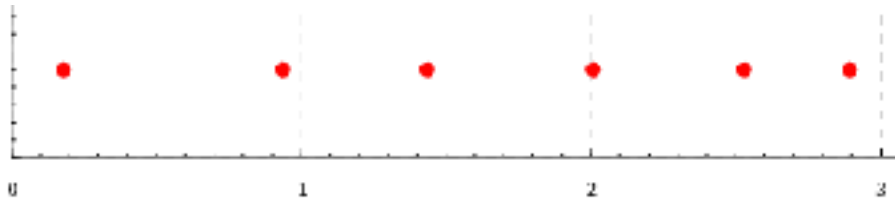

```
memlist :: [Prob a] -> Prob [a]
memlist (p : ps) = do { x <- p ; xs <- memlist ps ; return (x : xs) }
```

Laziness to Memoization

```haskell
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```

```haskell
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps
```

exp exp exp exp exp exp …



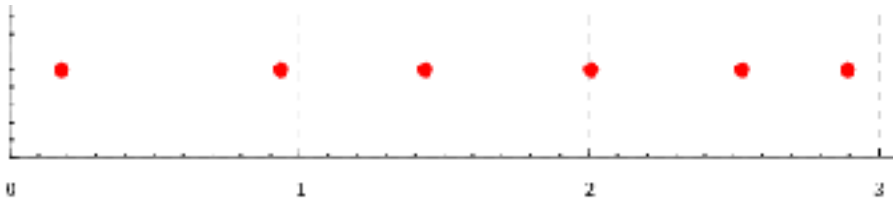

```haskell
memlist :: [Prob a] -> Prob [a]
memlist (p : ps) = do { x <- p ; xs <- memlist ps ; return (x : xs) }
```

Laziness to Memoization

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps
```

0.18 0.76 0.50 0.57 0.52 0.36 …



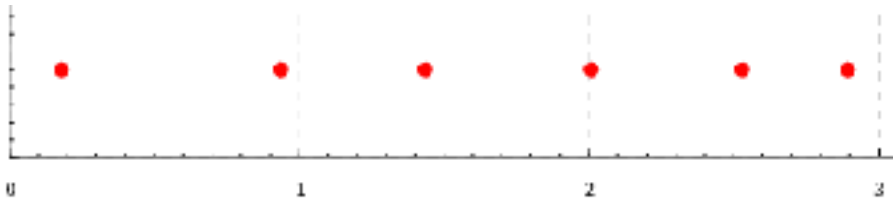

```
memlist :: [Prob a] -> Prob [a]
memlist (p : ps) = do { x <- p ; xs <- memlist ps ; return (x : xs) }
```

Laziness to Memoization

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps
```

```
0.18 0.76 0.50 0.57 0.52 0.36 …
0.18 0.94 1.44 2.01 2.53 2.89 …
```




```
memlist :: [Prob a] -> Prob [a]
memlist (p : ps) = do { x <- p ; xs <- memlist ps ; return (x : xs) }
```

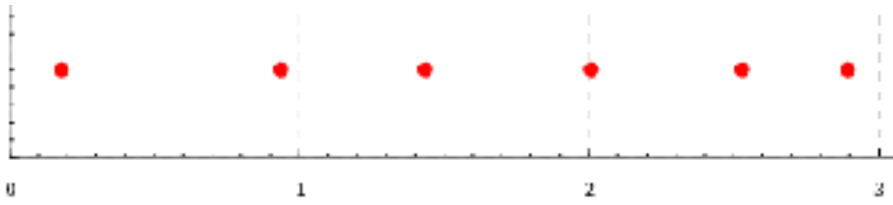
Laziness to Memoization

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```

Open problem.
Do quasi-Borel spaces support `memlist` over streams in general?
(ok for `a = RealNum, Bool` etc)

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps
```

```
0.18 0.76 0.50 0.57 0.52 0.36 …
0.18 0.94 1.44 2.01 2.53 2.89 …
```




```
memlist :: [Prob a] -> Prob [a]
memlist (p : ps) = do { x <- p ; xs <- memlist ps ; return (x : xs) }
```

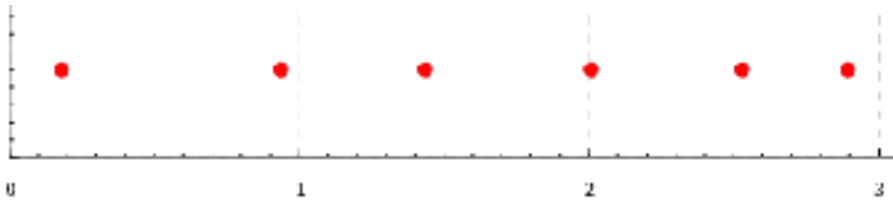
Laziness to Memoization

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps
```

0.18 0.76 0.50 0.57 0.52 0.36 …

0.18 0.94 1.44 2.01 2.53 2.89 …



Distributive law:
Dash & Staton. ACT 2020.
Jacobs. LICS 2021
Dahlqvist, Parlant, Silva. ICTAC 2018.
Keimel, Plotkin. 2016
Zwart, Marsden. LICS 2019.

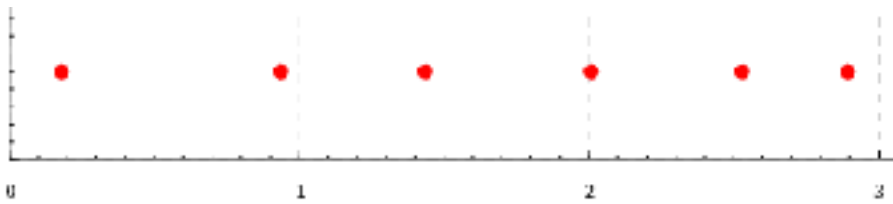
```
memlist :: [Prob a] -> Prob [a]
memlist (p : ps) = do { x <- p ; xs <- memlist ps ; return (x : xs) }
```

Laziness to Memoization

```haskell
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do step <- exponential rate
     let x = lower + step
     xs <- poissonPP x rate
     return (x : xs)
```

```haskell
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps
```

```
0.18 0.76 0.50 0.57 0.52 0.36 …
0.18 0.94 1.44 2.01 2.53 2.89 …
```




```haskell
memlist :: [Prob a] -> Prob [a]

memoize :: (Int -> Prob a) -> Prob (Int -> a)
memoize p = do { xs <- memlist (map p [0..]) ; return (xs !!) }
```

Stochastic Memoization

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Wood, Archambeau, Gasthaus, James, Teh. ICML 2009.

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps


memlist :: [Prob a] -> Prob [a]

memoize :: (Int -> Prob a) -> Prob (Int -> a)


memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
memoize p =
```

"Randomly pick from p x *for all* x*,*
and return that assignment as a function."

Stochastic Memoization

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Wood, Archambeau, Gasthaus, James, Teh. ICML 2009.

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps


memlist :: [Prob a] -> Prob [a]

memoize :: (Int -> Prob a) -> Prob (Int -> a)


memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
memoize p =
```

Return a random function that keeps a memo table:

Stochastic Memoization

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Wood, Archambeau, Gasthaus, James, Teh. ICML 2009.

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps


memlist :: [Prob a] -> Prob [a]

memoize :: (Int -> Prob a) -> Prob (Int -> a)


memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
memoize p =
```

Return a random function that keeps a memo table:

```
f 3
```

Stochastic Memoization

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Wood, Archambeau, Gasthaus, James, Teh. ICML 2009.

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps


memlist :: [Prob a] -> Prob [a]

memoize :: (Int -> Prob a) -> Prob (Int -> a)


memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
memoize p =
```

Return a random function that keeps a memo table:

```
f 3
```

```
p 3
```



Stochastic Memoization

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Wood, Archambeau, Gasthaus, James, Teh. ICML 2009.

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps


memlist :: [Prob a] -> Prob [a]

memoize :: (Int -> Prob a) -> Prob (Int -> a)


memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
memoize p =
```

Return a random function that keeps a memo table:

f 3

x	f x
3	23.6

p 3  23.6

Stochastic Memoization

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Wood, Archambeau, Gasthaus, James, Teh. ICML 2009.

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps


memlist :: [Prob a] -> Prob [a]

memoize :: (Int -> Prob a) -> Prob (Int -> a)


memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
memoize p =
```

Return a random function that keeps a memo table:

f 57

x	f x
3	23.6
57	17.5

p 57  17.5

Stochastic Memoization

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Wood, Archambeau, Gasthaus, James, Teh. ICML 2009.

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps


memlist :: [Prob a] -> Prob [a]

memoize :: (Int -> Prob a) -> Prob (Int -> a)


memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
memoize p =
```

Return a random function that keeps a memo table:

```
f 3
```

x	f x
3	23.6
57	17.5

🤔 23.6

Stochastic Memoization

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Wood, Archambeau, Gasthaus, James, Teh. ICML 2009.

```
poissonPP :: RealNum -> RealNum -> Prob [RealNum]
poissonPP lower rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) lower gaps


memlist :: [Prob a] -> Prob [a]

memoize :: (Int -> Prob a) -> Prob (Int -> a)


memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
memoize p =
```

Return a random function that keeps a memo table:

f 42

x	f x
3	23.6
57	17.5
42	9.4

p 42  9.4

Stochastic Memoization

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Wood, Archambeau, Gasthaus, James, Teh. ICML 2009.

```
poissonPPzero :: RealNum -> Prob [RealNum]
poissonPPzero rate =
  do gaps <- memlist $ repeat (exponential rate)
     return $ scanl (+) 0 gaps


memlist :: [Prob a] -> Prob [a]

memoize :: (Int -> Prob a) -> Prob (Int -> a)


memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
memoize f =  Prob $ do g <- get
                       let ( (Tree _ gs), g2) = splitTree g
                       put g2
                       return $ unsafePerformIO $ do
                                ref <- newIORef Data.Map.empty
                                return $ \x -> unsafePerformIO $ do
                                           m <- liftM (Data.Map.lookup x)
                                           . . .
```

Desiderata for a theory of Prob



Conjecture. Stochastic memoization still admits the dataflow property even though its implementation uses hidden state.

Dataflow property:

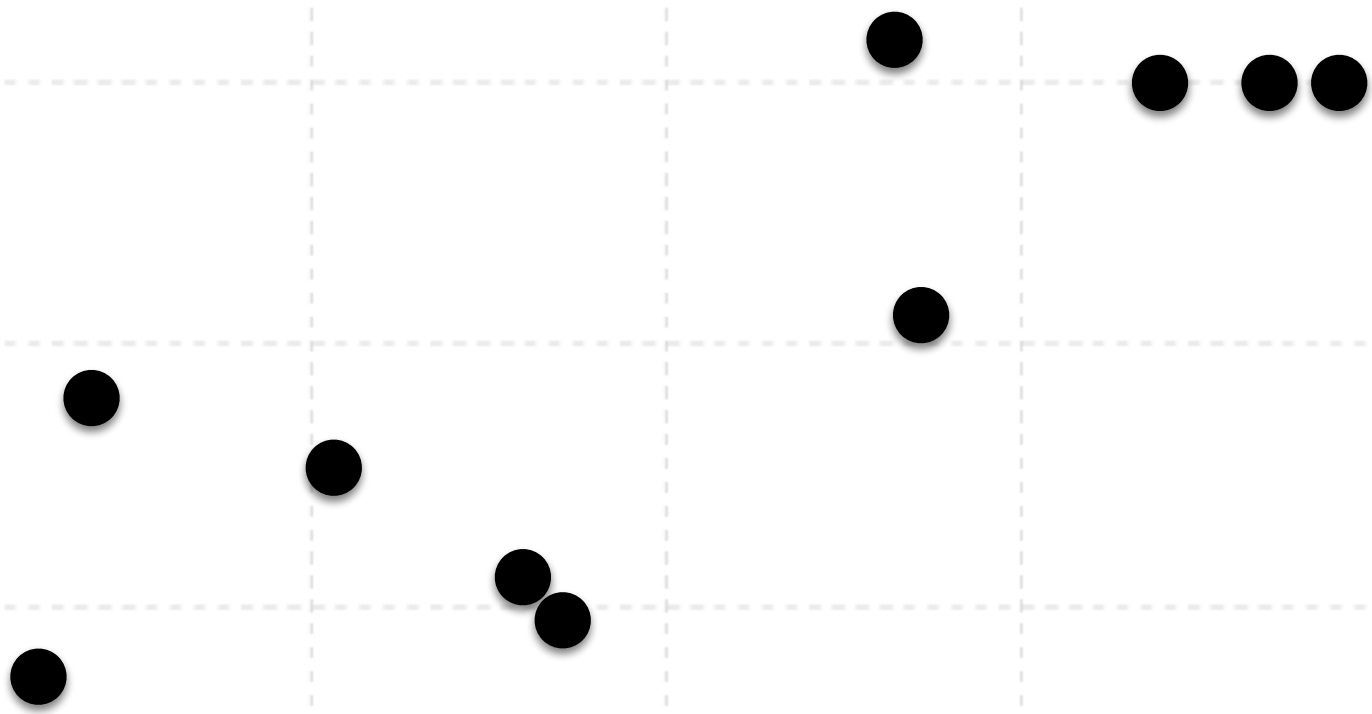
*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Related to Fubini's theorem.

Memoization and clustering

We want to cluster these points according to their position.

We don't know how many clusters.



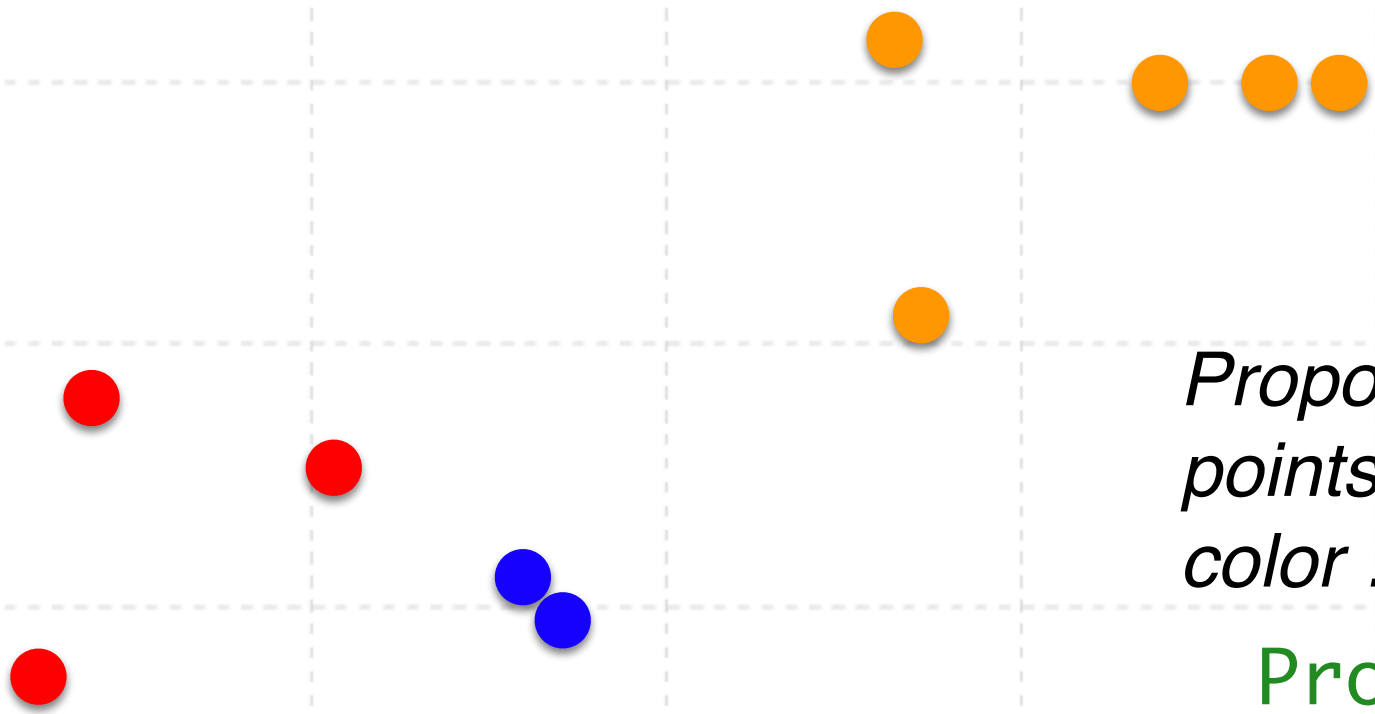
```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Memoization and clustering

We want to cluster these points according to their position.

We don't know how many clusters.



Proportion of points of each color :

```
Prob Color
```

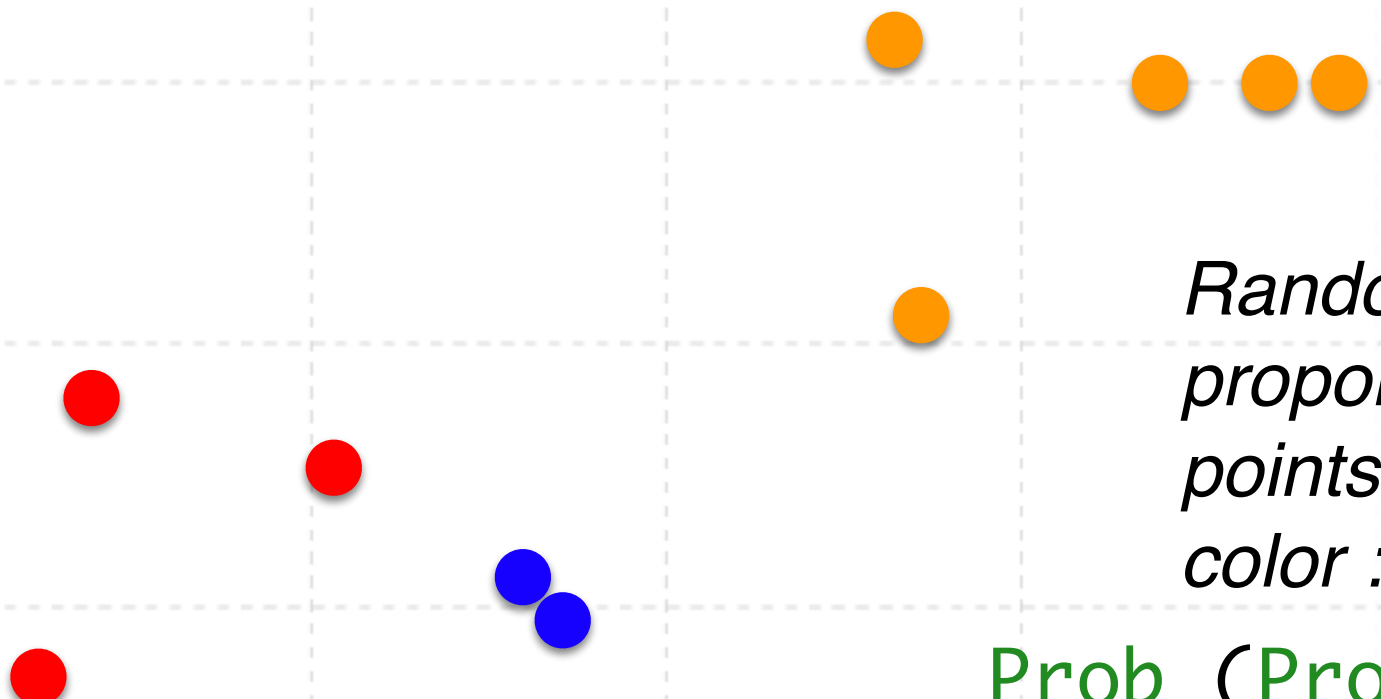
```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Memoization and clustering

We want to cluster these points according to their position.

We don't know how many clusters.



Random proportion of points of each color :

```
Prob (Prob Color)
```

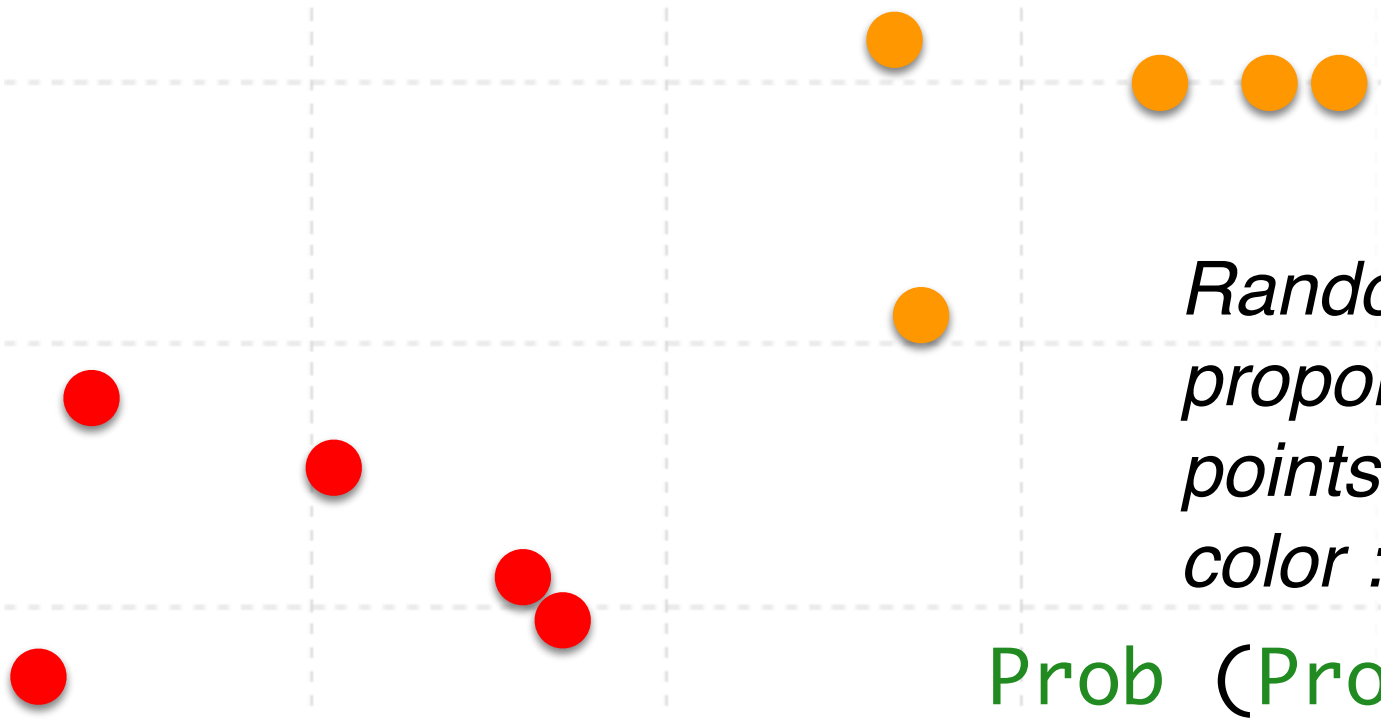
```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Memoization and clustering

We want to cluster these points according to their position.

We don't know how many clusters.



Random proportion of points of each color :

```
Prob (Prob Color)
```

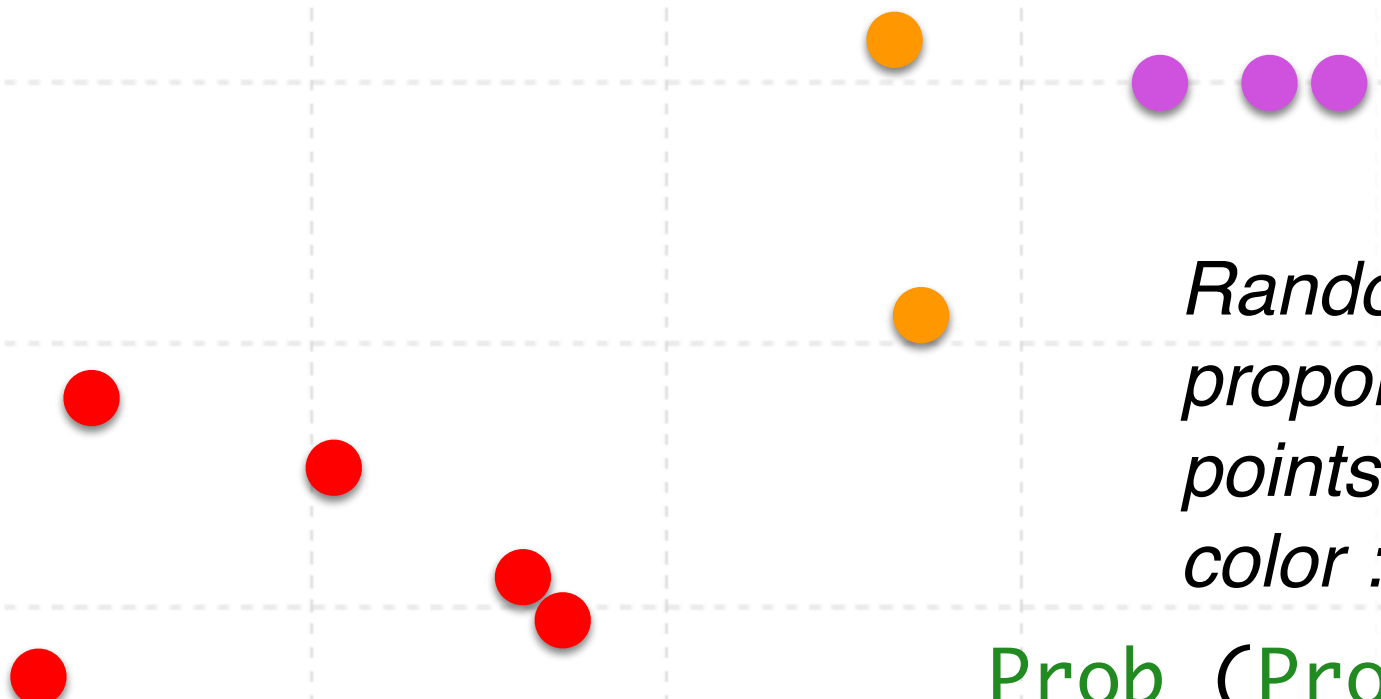
```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Memoization and clustering

We want to cluster these points according to their position.

We don't know how many clusters.



Random proportion of points of each color :

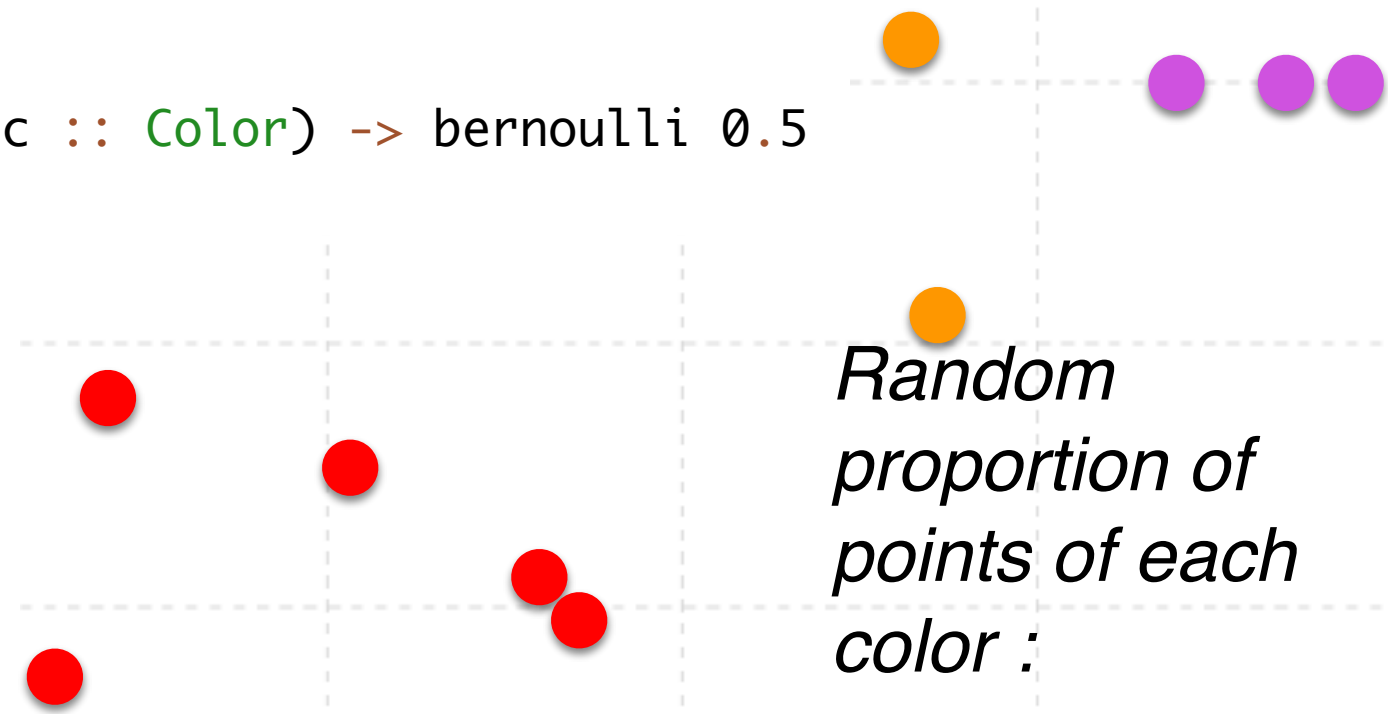
`Prob (Prob Color)`

```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Memoization and clustering

```
vote <- memoize $ \(c :: Color) -> bernoulli 0.5
```



Random proportion of points of each color :

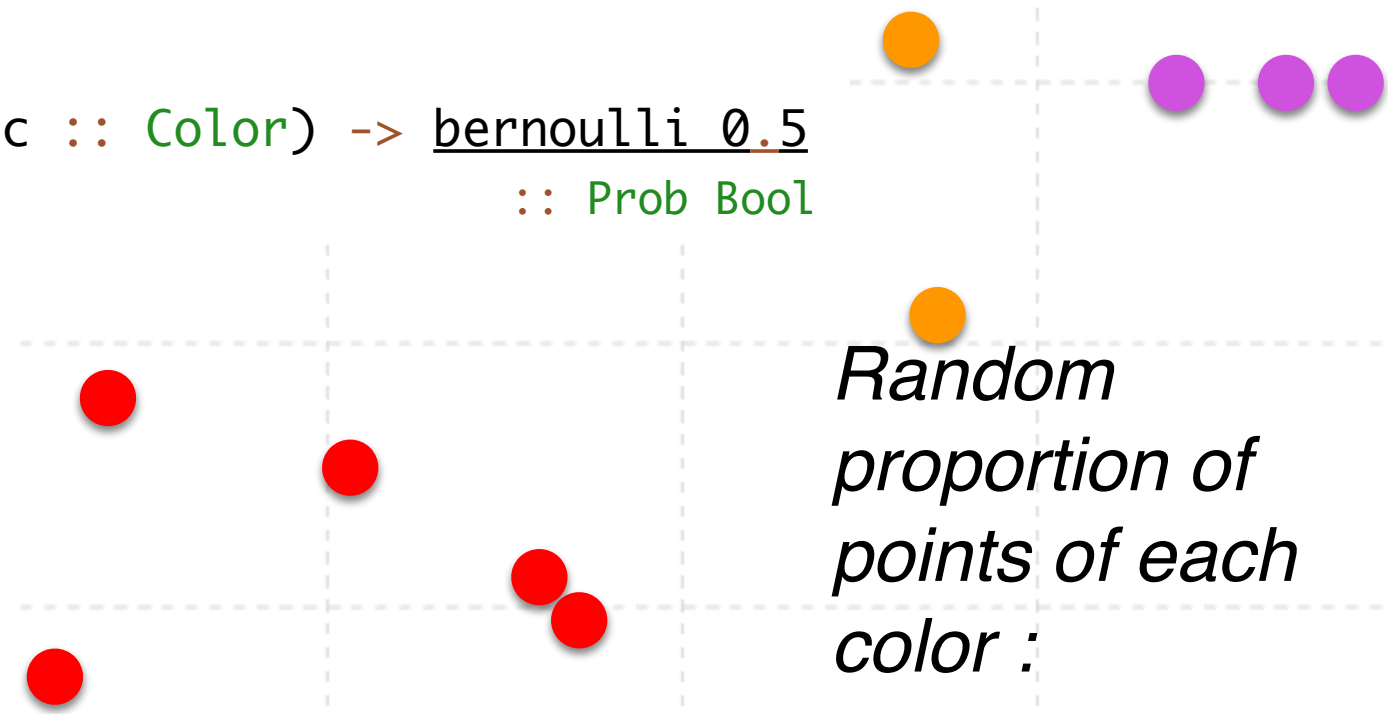
`Prob (Prob Color)`

```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Memoization and clustering

```
vote <- memoize $ \(c :: Color) -> bernoulli 0.5
                                      :: Prob Bool
```



Random proportion of points of each color :

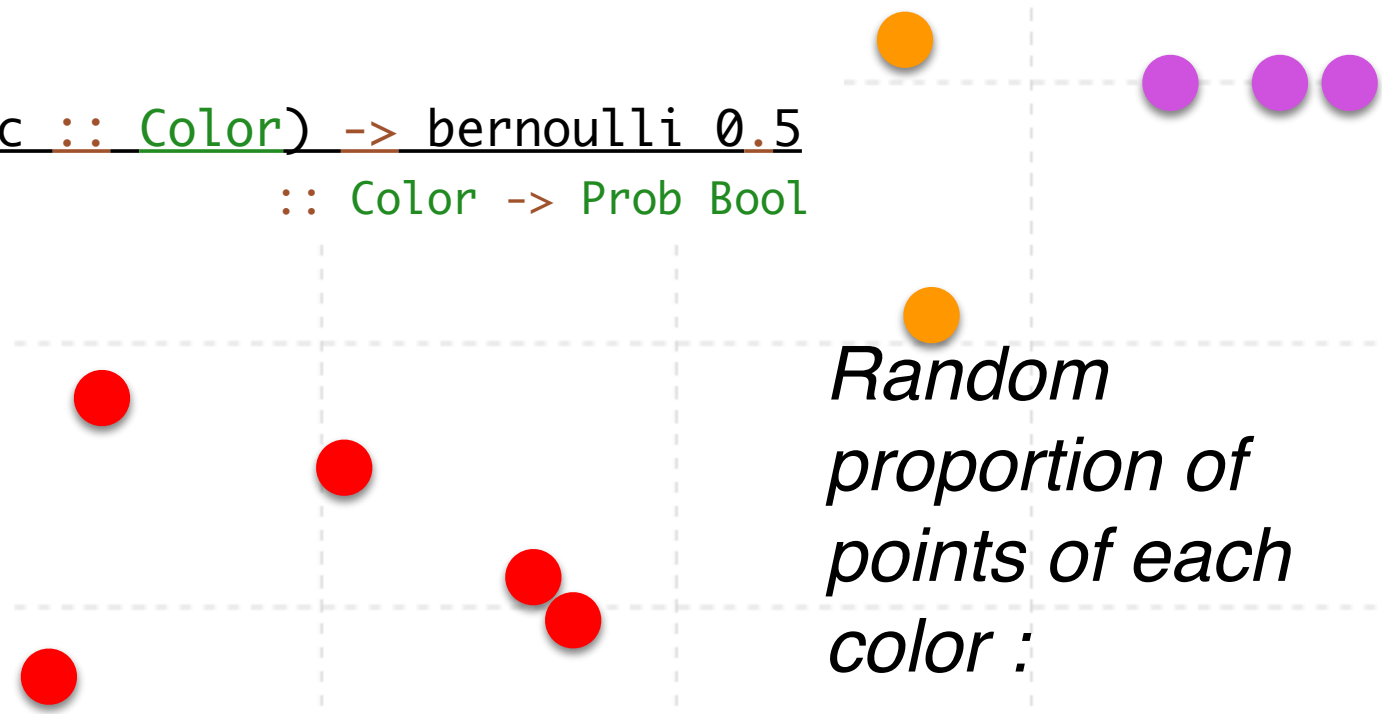
`Prob (Prob Color)`

```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Memoization and clustering

```
vote <- memoize $ \(c :: Color) -> bernoulli 0.5
                                  :: Color -> Prob Bool
```



Random proportion of points of each color :

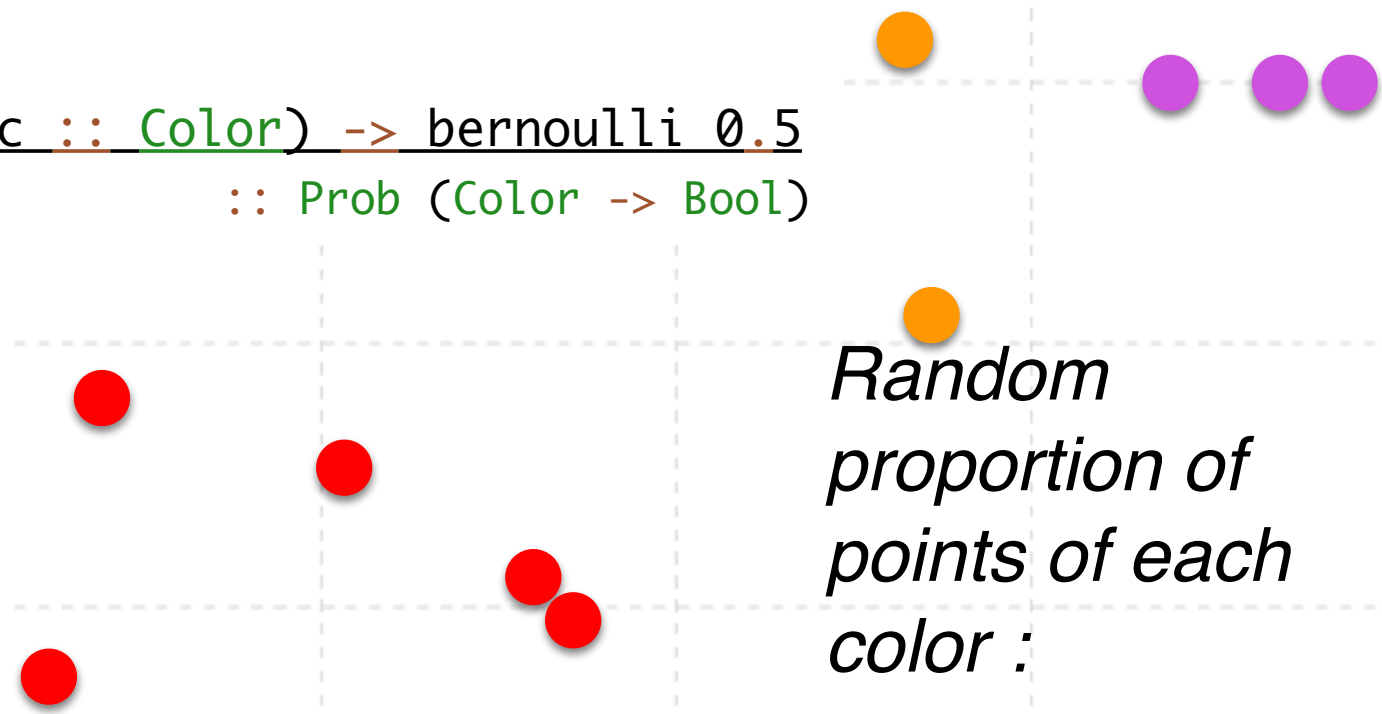
```
Prob (Prob Color)
```

```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Memoization and clustering

```
vote <- memoize $ \(c :: Color) -> bernoulli 0.5
                                  :: Prob (Color -> Bool)
```



Random proportion of points of each color :

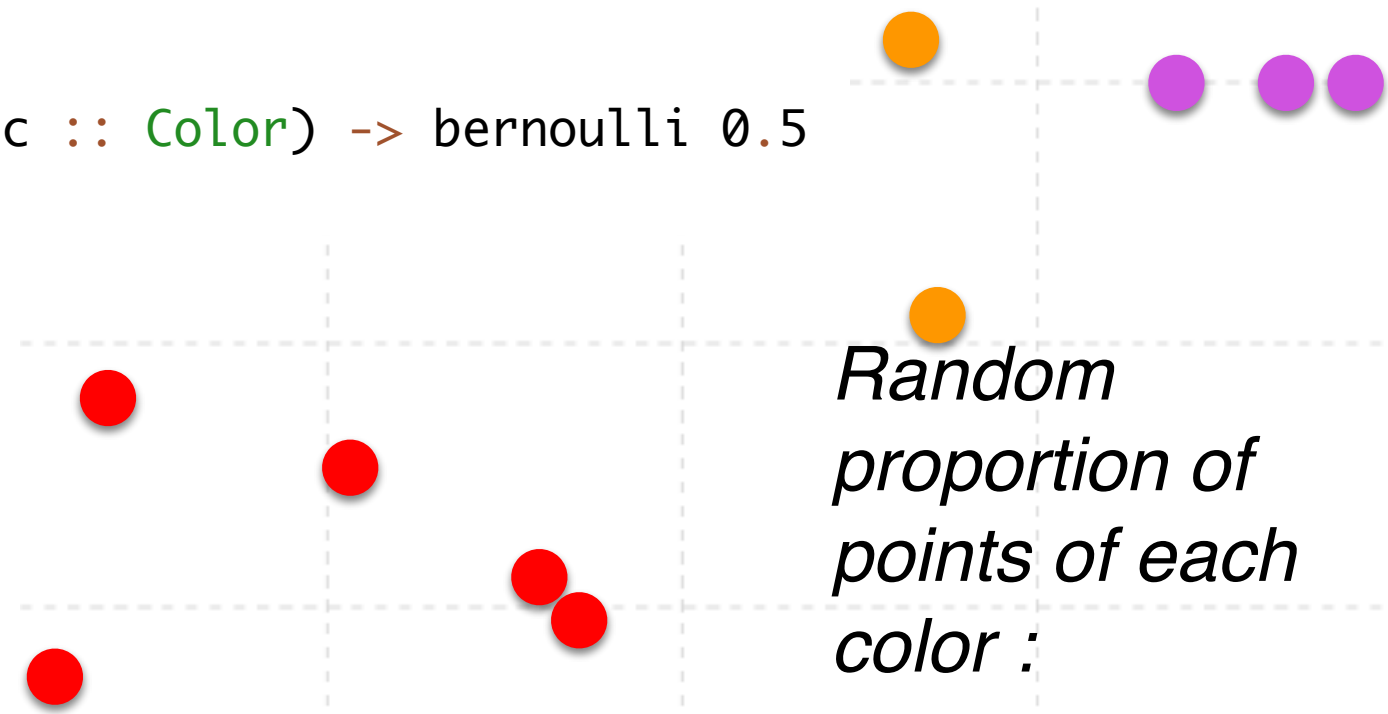
`Prob (Prob Color)`

```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Memoization and clustering

```
vote <- memoize $ \(c :: Color) -> bernoulli 0.5
:: Color -> Bool
```



Random proportion of points of each color :

`Prob (Prob Color)`

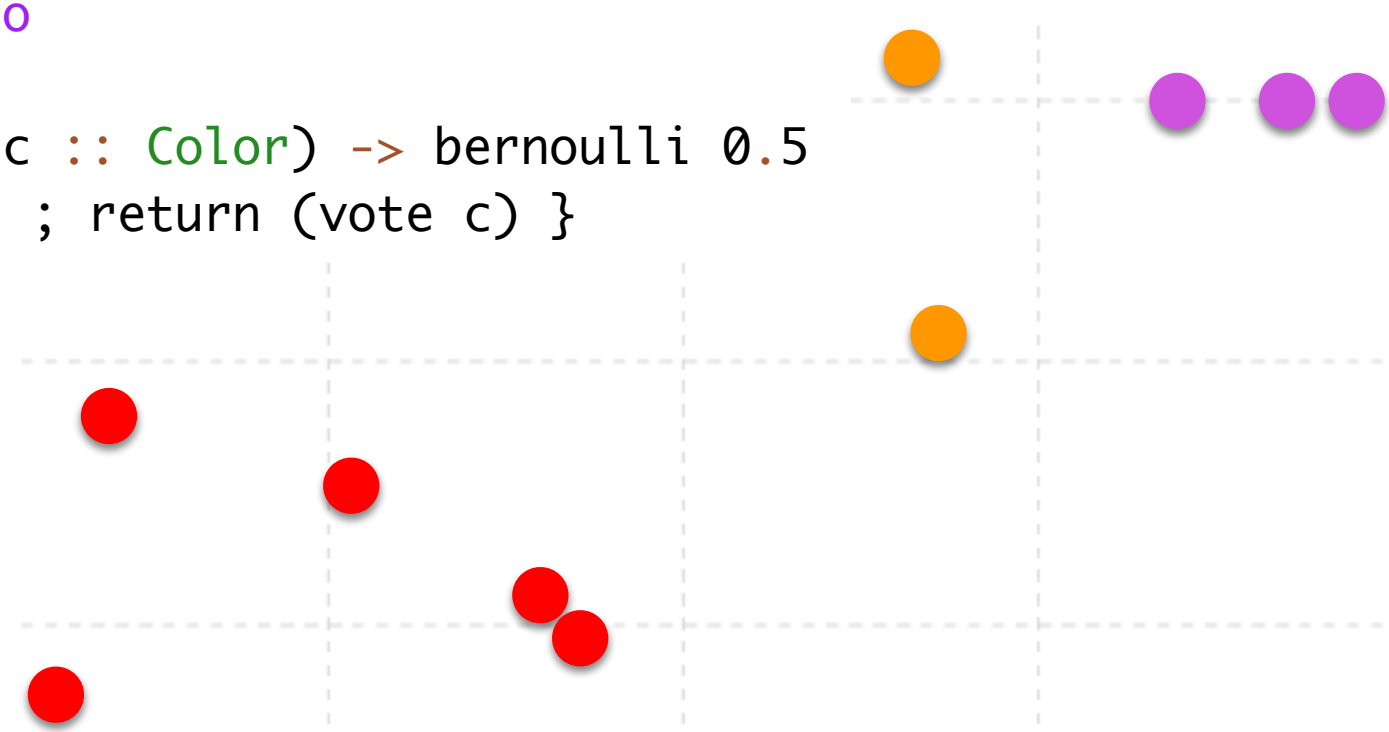
```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Memoization and clustering

```
dirichletPmix :: Prob (Prob Color)

electionPrediction :: Prob (Prob Bool)
electionPrediction = do
  p <- dirichletPmix
  vote <- memoize $ \(c :: Color) -> bernoulli 0.5
  return $ do { c <- p ; return (vote c) }
```



```
memoize :: Ord a => (a -> Prob b) -> Prob (a -> b)
```

e.g.
Roy, Mansinghka, Goodman, Tenenbaum, NPB 2008
Goodman, Mansinghka, Roy, Bonawitz, Tenenbaum. 2008.

Semantics of stoch. memoization

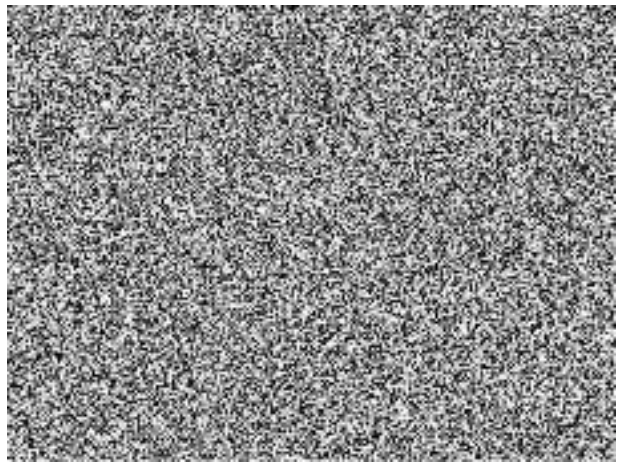
```
memoize :: (RealNum -> Prob Bool) -> Prob (RealNum -> Bool)
memoize $ \r -> bernoulli 0.5 :: Prob (RealNum -> Bool)
```

Theorem: No such random function in quasi-Borel spaces

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

So in qBs's, a probability measure on $[\![$ `RealNum -> Bool` $]\!]$ is a Borel function $\mathbb{R} \to \mathbb{R} \to \{0,1\}$ $(\mathrm{mod} \sim)$

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…



Semantics of stoch. memoization

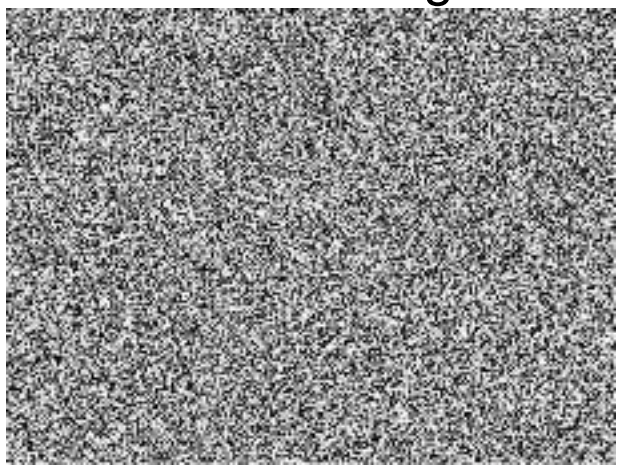
```
memoize :: (RealNum -> Prob Bool) -> Prob (RealNum -> Bool)
memoize $ \r -> bernoulli 0.5 :: Prob (RealNum -> Bool)
```

Theorem: No such random function in quasi-Borel spaces

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

So in qBs's, a probability measure on $[\![$ `RealNum -> Bool` $]\!]$ is a Borel function $\mathbb{R} \to \mathbb{R} \to \{0,1\}$ $(\mathrm{mod} \sim)$

We would need something like this:



But it needs to behave like this graphon:

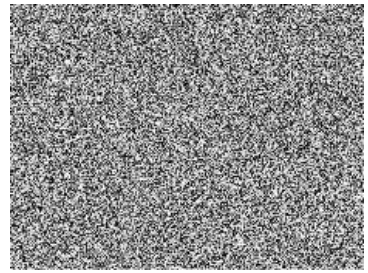


and these two graphons are not equivalent. $\mathbb{R} \times \mathbb{R} \to [0,1]$

Semantics of stoch. memoization

jww Younesse Kaddar
See also WIP by Hugo Paquet and also Alex Simpson.

```
memoize :: (Atom -> Prob Bool) -> Prob (Atom -> Bool)
```





- Idea: Let's use nominal sets! Gabbay, Pitts LICS 1999
- **Theorem.** Graphons *are* probability measures in nominal set theory. me + Ackerman, Freer, Roy, Yang. CPS 2020. https://www.youtube.com/watch?v=8Y0b7e4OIUQ
- Also, nominal sets are a great place for local memory effects!
- Use atoms with structure (bipartite graphs)

e.g. Bojanczyk, Klin, Lasota. LMCS 2014

Overview

1. Examples of high level probabilistic programs
   `randlinear, randprog, wiener, regress`
2. First semantic models
   quasi-Borel spaces, Markov categories
3. Laziness and point processes
   Poisson pt processes via laziness, piecewise regression
4. **Stochastic memoization and ongoing work**
   Typed memoization, towards nominal sets and graphons

Abstraction in *probabilistic* programming

High level e.g. infinite dimensional systems
higher-order functions
abstract types

Low level e.g. bets, frequencies, decisions
Monte Carlo simulation

	ML / stats apps	Foundational
High level	✓	✓
Low level	✓	✓

Final observations

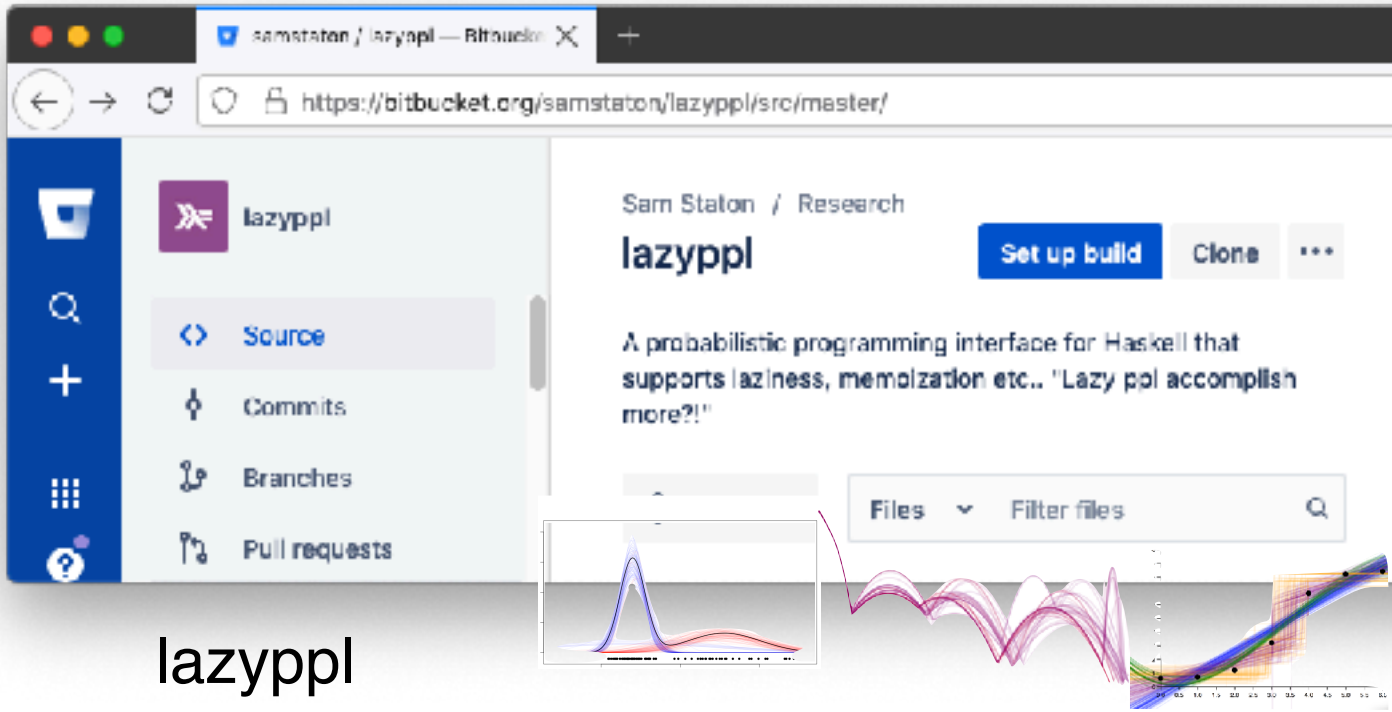
- High level structure is abstract
  "Real data scientists don't use lazyppl"?

- Non-termination?

	ML / stats apps	Foundational
High level	✓	✓
Low level	✓	✓

This talk

Open question:
What is a sound **semantic model** for lazyppl ?



lazyppl
= lazy **p**robabilistic **p**rogramming **l**anguage
Builds on ideas from *Anglican, BayesDB, Birch, Church, Figaro, Hansei, WebPPL* and others. (But only ~25 lines of code.)

Key ideas: types, higher order functions, laziness, stochastic memoization.

Suggests new ***formal structures in probability.***

e.g. quasi-Borel spaces, Markov categories, nominal sets

	ML / stats apps	Foundational
High level	✓	✓
Low level	✓	✓