Embedded Domain-Specific Languages for Bayesian Modeling and Inference

Thesis

Novel and useful domain-specific languages for solving statistical problems can be embedded in statically-typed, purely-functional programming languages.

Defence

First Claim

The Giry monad[1] is suitable for representing probabilistic semantics in an embedded functional language.

[1] Lawvere (1962), Giry (1981)

Its functorial, applicative, and monadic structure corresponds to image measure, product measure, and marginalization[2] respectively.

[2] Ramsey & Pfeffer (2002)

$\mathcal{P} : \mathbf{Meas} \to \mathbf{Meas}$	functor (image measure)
$\phi : \mathcal{P}(M) \otimes \mathcal{P}(N) \to \mathcal{P}(M \otimes N)$	applicative (product measure)
$\eta : M \to \mathcal{P}(M)$	applicative (Dirac measure)
$\gg\!=\! : \mathcal{P}(M) \to (M \to \mathcal{P}(N)) \to \mathcal{P}(N)$	monad (marginalize)

```haskell
instance Functor Measure where
  fmap g nu = Measure (\f ->
    integrate (f . g) nu)

instance Applicative Measure where
  pure x                    = Measure (\f -> f x)
  Measure g <*> Measure h = Measure (\f ->
    g (\k -> h (f . k))

instance Monad Measure where
  return x  = Measure (\f -> f x)
  rho >>= g = Measure (\f ->
    integrate (\m -> integrate f (g m)) rho)
```

image measure

```
inverseGamma :: Double -> Double -> Measure Double
inverseGamma a b = fmap recip (gamma a b)
```

product measure (convolution)

```haskell
chiSq2 :: Measure Double
chiSq2 = gsq + gsq where
  gsq = fmap (^ 2) (gaussian 0 1)
```

marginalization

```
betaBinomial :: Int -> Double -> Double -> Measure Int
betaBinomial n a b = do
  p <- beta a b
  binomial n p
```

The Giry monad is equivalent to the well-known continuation monad[3] when the return type is restricted to the real numbers.

[3] Wadler (1994)

giry

```haskell
newtype Measure a = Measure ((a -> Double) -> Double)

integrate :: (a -> Double) -> Measure a -> Double
integrate f (Measure nu) = nu f
```

continuation

```haskell
newtype Cont r a = Cont ((a -> r) -> r)

run :: (a -> r) -> Cont r a -> r
run f (Cont k) = k f
```

Second Claim

The probabilistic semantics of the Giry monad are preserved under an abstract or free[4] representation that faithfully represents probabilistic programs.

[4] See also: Scibior et al (2015)

$U : (G, \ldots) \mapsto G$	forgetful functor
$F : G \mapsto (G, \mu, \eta)$	free monad
$F \dashv U$	adjunction

abstract giry functor $(\mathcal{P} : \mathbf{Meas} \to \mathbf{Meas})$

```
data Prob r =
    Beta Double Double (Double -> r)
  | Bernoulli Double (Bool -> r)
  | Gaussian Double Double (Double -> r)
  | ...
```

probabilistic program $\left(F\mathcal{P} : \mathbf{Meas} \to \left(\mathcal{P}, \int, \delta\right)\right)$

```
type Program = Free Prob
```

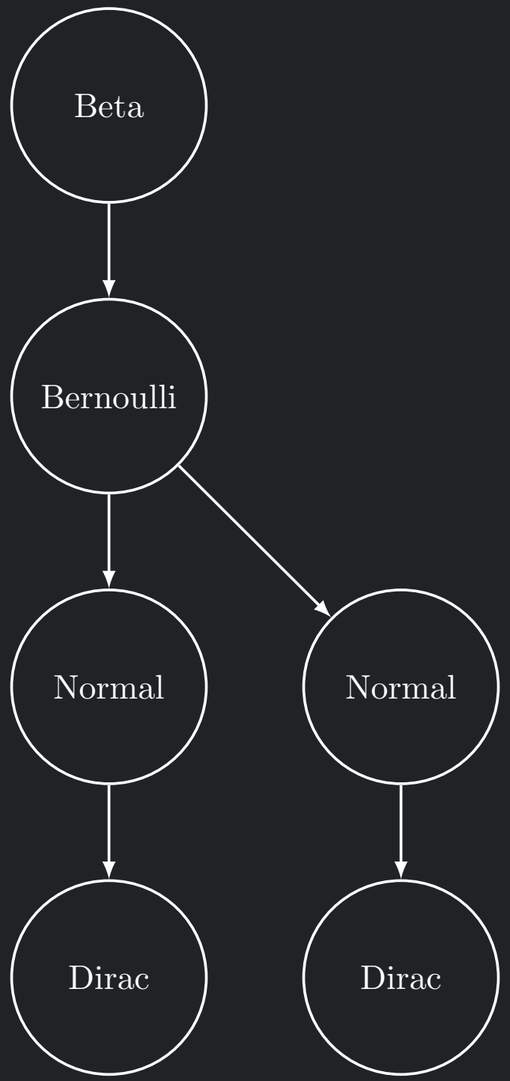
example: bayesian regression model

```
prior = do
  intercept <- gaussian 0 10
  slope     <- gaussian 0 10
  variance  <- uniform (0, 100)
  return (intercept, slope, variance)

likelihood observations (a, b, v) = do
  let model x = gaussian (a + b * x) (sqrt v)
  for observations model

predictive observations = do
  parameters <- prior
  likelihood observations parameters
```

example: gaussian mixture model

```
mixture a b = do
  prob  <- beta a b
  heads <- bernoulli prob
  if   heads
  then gaussian 0 2
  else gaussian 2 2
```

example: gaussian mixture model



This abstract probability monad has no probabilistic interpretation tied to it a priori; it is generic and can be ascribed specific interpretations (e.g. probability measure, random variable)[5] after the fact.

[5] Ramsey & Pfeffer (2002), Park et al (2008)

```haskell
measure :: Program a -> Measure a
measure = iterM $ \case
  Beta a b r      -> Measurable.beta a b >>= r
  Bernoulli p r   -> Measurable.bernoulli p >>= r
  Gaussian m s r -> Measurable.gaussian m s >>= r
  ...

rvar :: Program a -> RandomVariable a
rvar = iterM $ \case
  Beta a b r      -> MWC.beta a b >>= r
  Bernoulli p r   -> MWC.bernoulli p >>= r
  Gaussian m s r -> MWC.gaussian m s >>= r
  ...
```

The dual or cofree comonad of the free probability monad represents an execution trace[6] of a probabilistic program.

[6] Wingate et al (2011), Mansinghka et al (2014)

$$U : (G, \ldots) \mapsto G \qquad \text{forgetful functor}$$
$$F' : G \mapsto (G, \nu, \chi) \qquad \text{cofree comonad}$$
$$U \dashv F' \qquad \text{adjunction}$$

$$F \dashv U \dashv F'$$

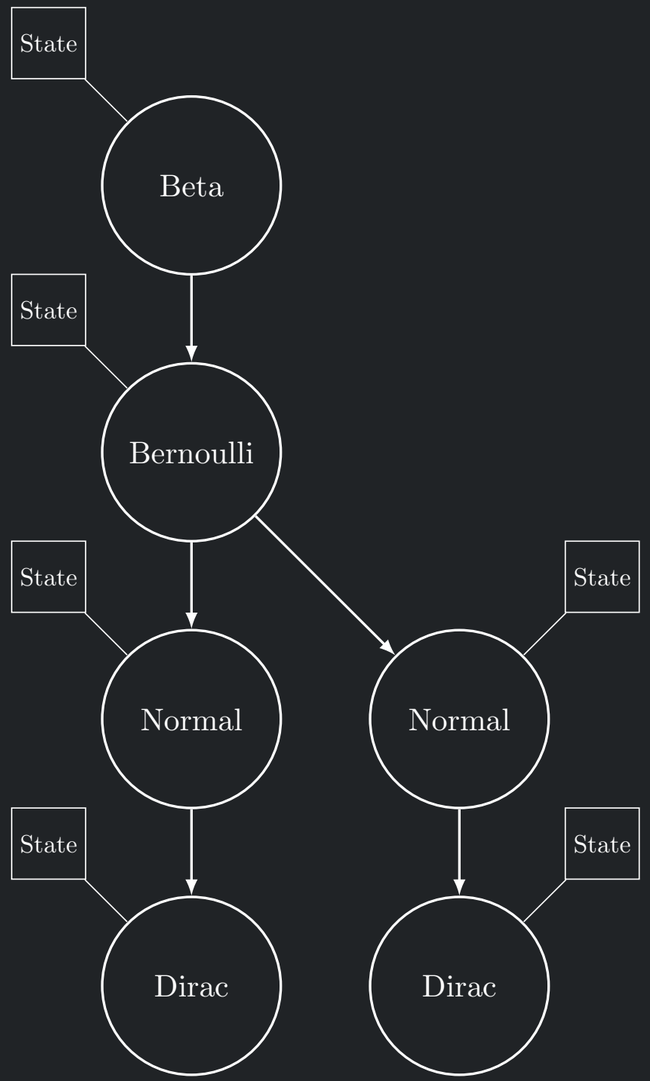
probabilistic node state

```haskell
data State = State {
    cost    :: Double
  , value   :: Dynamic
  , prng    :: Seed
  , history :: [Dynamic]
  }
```

execution trace $(F'\mathcal{P} : \mathbf{Meas} \to (\mathcal{P}, \partial, X))$

```haskell
type Trace a = Cofree (Prob a) State
```

execution trace



Execution traces can be perturbed using standard comonadic machinery in order to perform inference via e.g. single-site MCMC in trace space.

per-node proposal

```
proposal :: Trace a -> State
proposal = <abstract>
```

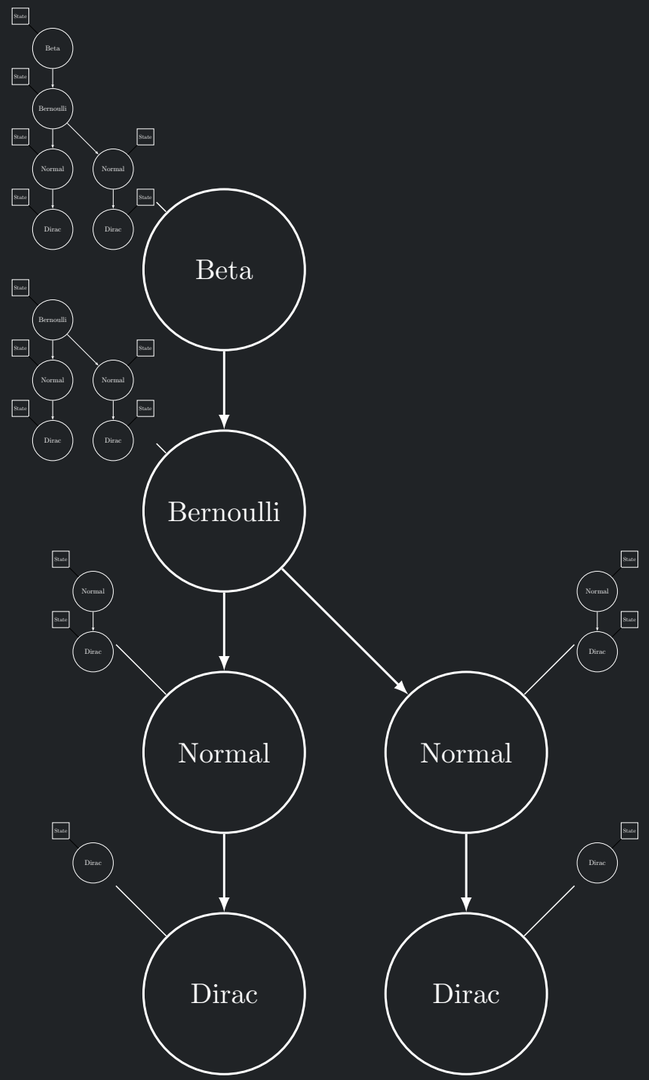
comonadic 'extend'

$$\ggg: \mathcal{P}(M) \to (\mathcal{P}(M) \to N) \to \mathcal{P}(N)$$

markov transition

```
transition :: Trace a -> Trace a
transition = extend proposal
```

comonadic duplication



The free applicative functor encodes probabilistic independence such that it can be observed and interpreted statically.

'iid' combinator

```
model :: Int -> Int -> Double -> Double -> Program [[Bool]]
model m n a b = do
  c <- fmap (+ 1) (beta a b)
  d <- fmap (+ 2) (beta a b)
  iid m $ do
    p <- beta c d
    iid n (bernoulli p)
```

Third Claim

The well-known state monad can be used to represent transition operators for constructing Markov chains.

target function

```
data Target a = Target {
    ltarget  :: Vector a -> Double
  , gltarget :: Maybe (Vector a -> Vector a)
  }
```

markov chain state

```
data Markov a = Markov {
    target   :: Target a
  , position :: Vector a
  }
```

transition operator

```
type Transition a = StateT (Markov a) Prob (Vector a)
```

A simple, shallowly-embedded language consisting of deterministic and random concatenation terms can be used to build compound transition operators.

```
transition ::= primitive <primitive>
             | concat transition transition
             | sample transition transition
```

implementation

```
concat :: Transition a -> Transition a -> Transition a
concat s t = s >> t

sample :: Double -> Transition a -> Transition a -> Transition a
sample p s t = do
  heads <- lift (bernoulli p)
  if   heads
  then s
  else t
```

These compound transitions are guaranteed by construction[7] to obey the Markov, ergodicity, and detailed balance properties required for MCMC.

[7] Geyer (2005)

use a number of metropolis transitions

```
transition :: Transition Double
transition = do
  metropolis 0.5
  metropolis 1.0
  metropolis 2.0
```

use a variety of slice sampling transitions

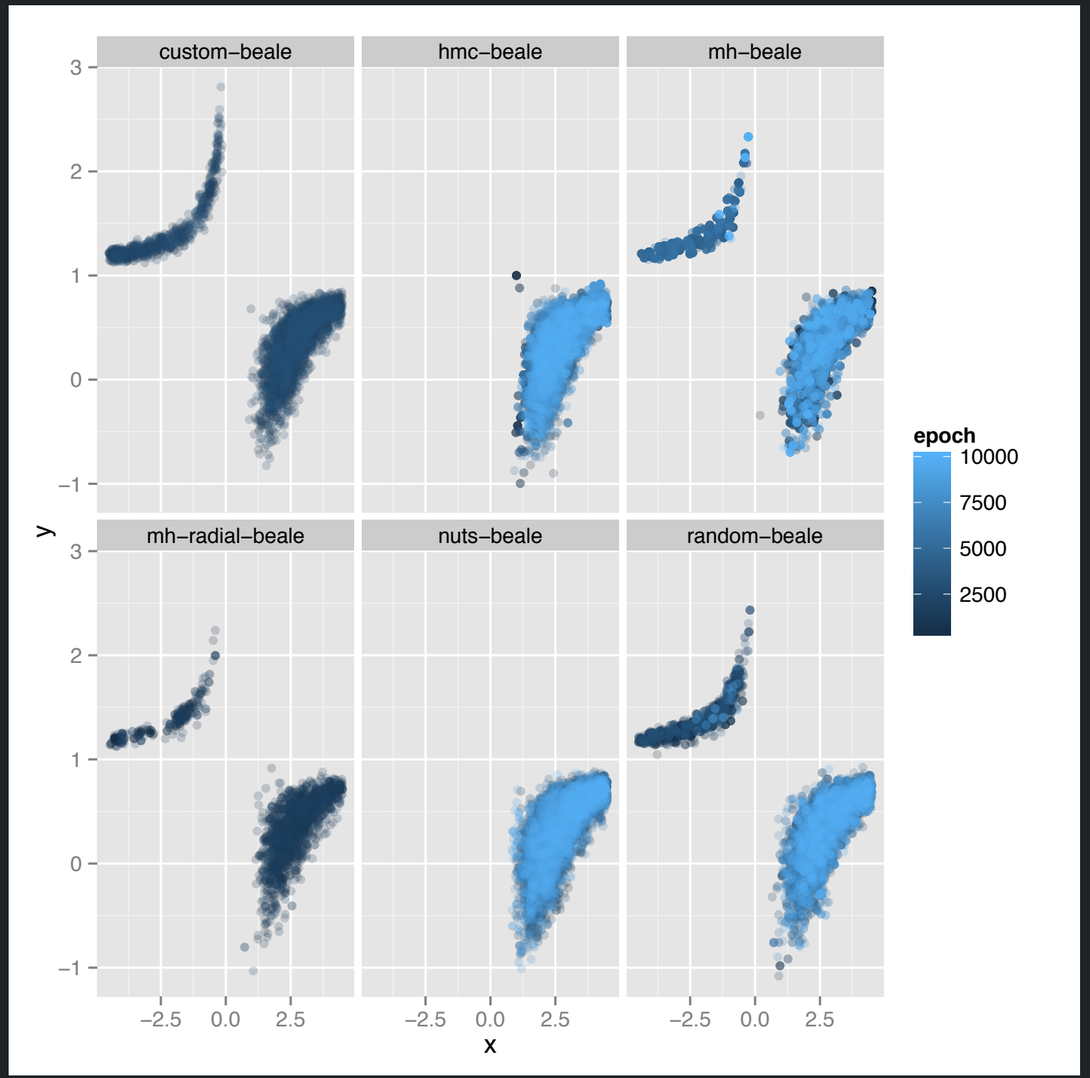
```
transition :: Transition Double
transition = do
  slice 0.5
  slice 1.0
  sample 0.5 (slice 0.4) (slice 10.0)
```

occasionally use gradient information

```
transition :: Transition Double
transition = do
  sample 0.8 (metropolis 3.0) (hamiltonian 0.05 20)
  slice 3.0
```

Compound transition operators can be more effective than simple transitions.

simulations



transition	ESS(x)	ESS(y)
mh-beale	12	47
mh-radial-beale	26	60
hmc-beale	489	663
nuts-beale	437	749
custom-beale	31	58
random-beale	21	64

The language can easily be extended to support techniques such as annealing.

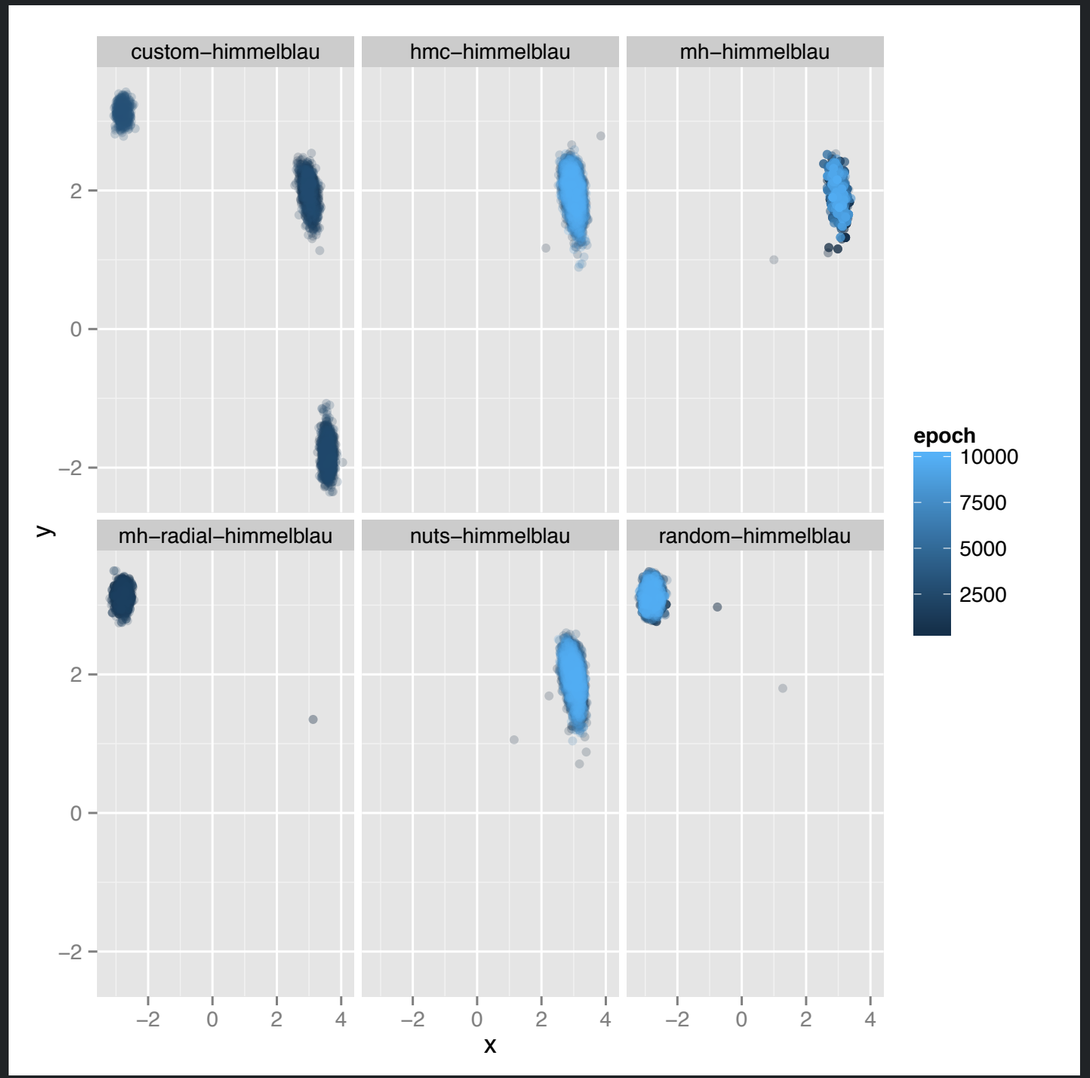
annealing transformer

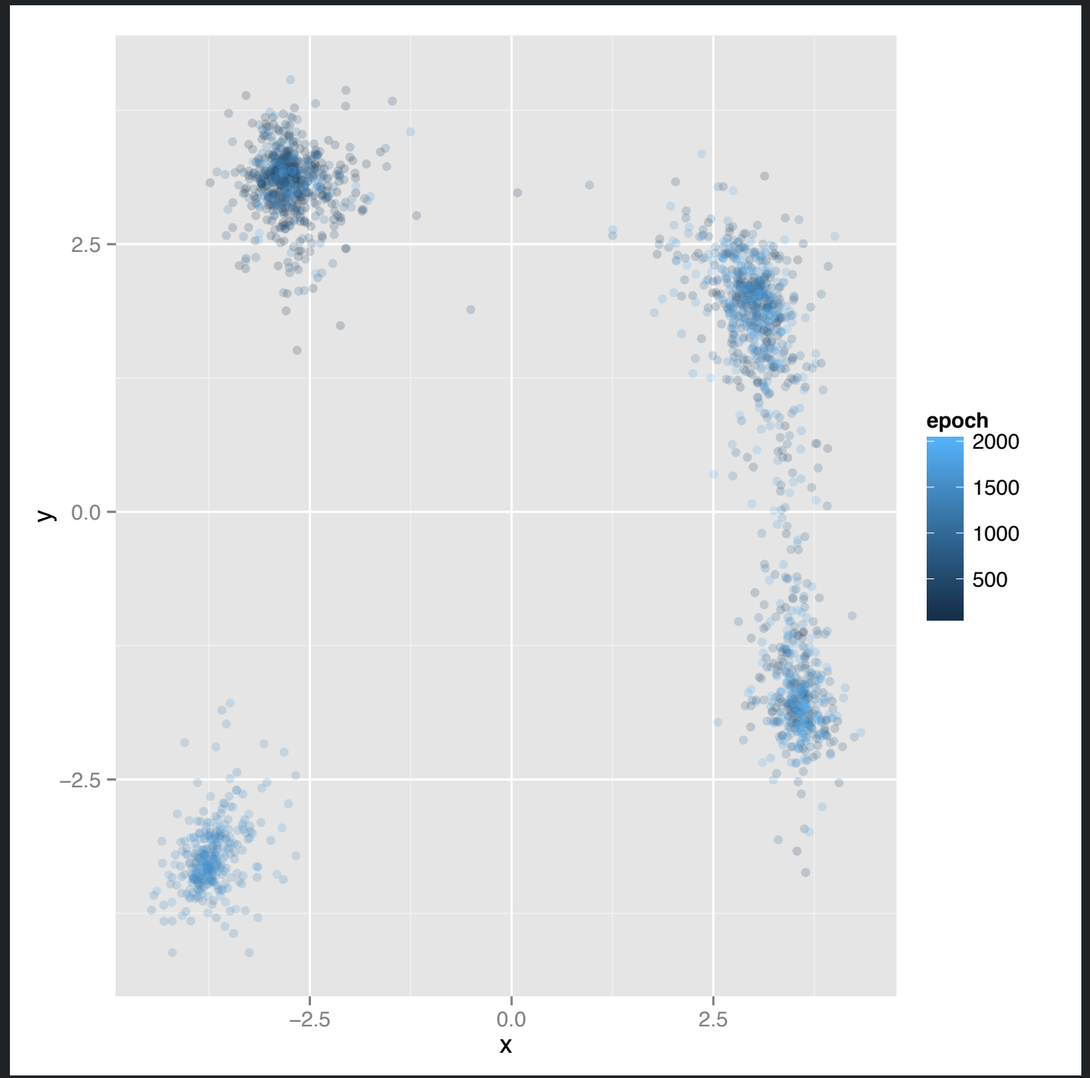
```
anneal :: Double -> Transition Double -> Transition Double
anneal invtemp = <abstract>
```

annealed transition

```
annealed :: Transition Double -> Transition Double
annealed transition = do
  anneal 0.70 transition
  anneal 0.05 transition
  anneal 0.05 transition
  anneal 0.70 transition
  transition
```

simulations





Conclusion

The Giry monad captures meaningful probabilistic semantics and allows one to construct an embedded language for probability measures.

Abstract free or cofree representations preserve the probabilistic semantics of the Giry monad and allow for rich interpretation.

The state monad allows one to construct an embedded language for Markov transitions that are useful in MCMC.

Thus: novel and useful domain-specific languages for solving statistical problems can be embedded in statically-typed, purely-functional programming languages.

QED