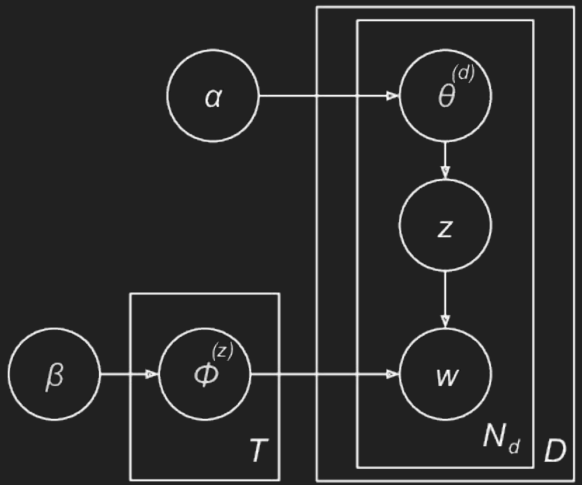
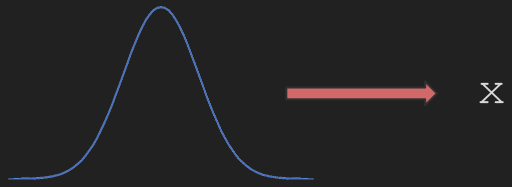
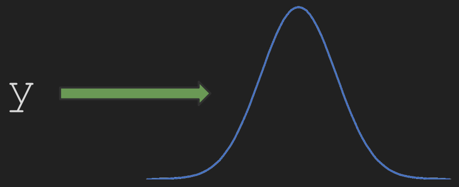
Embedding Probabilistic Models in Haskell!



In a probabilistic language, we have access to two additional primitive operations:

```
x <- sample(dist)          observe(dist, y)
```





Given these operations, we can capture the following notions:

Model

$\boxed{\text{Model}} = R(\theta, x, y)$

Simulation

$\theta, x \longrightarrow \boxed{\text{Model}} \longrightarrow y$

Inference

$y, x \longrightarrow \boxed{\text{Model}} \longrightarrow \theta$

We can categorize probabilistic programming languages (PPLs) into:

1. Query Based Languages
2. Model Based Languages

Query-Based PPLs

Probabilistic Query

```
def query(...):
  ...
  sample(dist)
  ...
  observe(dist, y)
  ...
```

Queries are functions where we can **explicitly** call `sample` and `observe`

Benefits of query-based PPLs:

They are flexible

They are modular - they can be combined and sometimes even composed

Disadvantage of query-based PPLs

They can only express **specific** interpretations of models

Query-Based PPLs

Query for Simulation (Monad Bayes)

```
linRegr :: MonadSample m
  => Double -> Double -> Double -> Double -> m Double
linRegr x μ c σ = do
 y <- normal (μ * x + c) σ
 return y
```

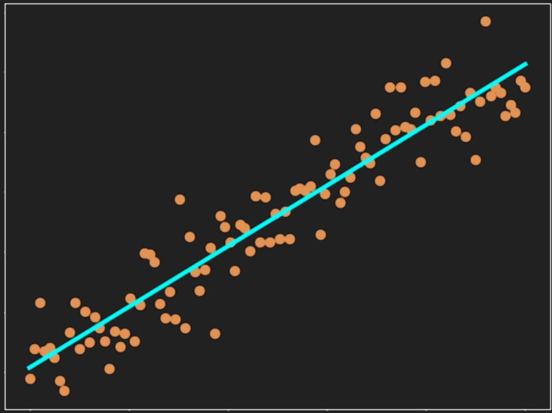
Query for Inference (Monad Bayes)

```
linRegr :: MonadSample m
 => Double -> Double -> m (Double, Double, Double)
linRegr x y = do
 μ <- normal 0 3
 c <- normal 0 2
 σ <- uniform 1 3
 observe $ normalPdf (μ* x + c) σ y
 return (μ , c , σ)
```

Problem:

1) We can’t capture a universal description of a model

2) We can’t simply evolve a model, we must evolve and maintain all queries that describe it.



Query-Based PPLs

Query for Simulation (WebPPL)

```
var linearRegr = function(mu, sigma, x) {
 var y = sample(Normal(mu * x, sigma))
 return y
}
var linearRegrModel = function () {
 linearRegr({mu = 0, sigma = 1, x = 4})
}
```

Query for Inference (WebPPL)

```
var linearRegr = function(mu, sigma, x, data_y) {
 observe(Normal(mu * x, sigma), data_y)
 return (mu, sigma)
}
var linearRegrModel = function () {
 linearRegr({mu = 0, sigma = 1, x = 4, data_y = 3})
}
var params = Infer({ model: linearRegrModel })
```

Query for Simulation (Anglican)

```
(defquery linear-regression [mu-prior, x]
 (let [mu    (sample mu-prior)
       sigma (sample sigma-prior)
       y (reduce + (map (* mu) x)) sigma)]
   {:output y}))
```

Query for Inference (Anglican)

```
(defquery linear-regression [mu-prior data_y x]
 (let [mu    (sample mu-prior)
       sigma (sample sigma-prior)
       predictive (fn [x] (normal (reduce + (map (* mu) x)) sigma))]
       (observe (predictive x) data_y)
   {:mu mu :sigma sigma :predictor (predictive x)}))
```

Model-Based PPLs

Probabilistic Model

```
@model function linRegr(μ, c, σ, x, y)
     μ ~ Normal(0, 3)
     c ~ Normal(0, 2)
     σ ~ Uniform(1, 3)
     y ~ Normal(μ * x + c, σ)
  end
```

- Models are a description of relationships between random variables
- We do *not* explicitly `sample` or `observe`

A Specification of Sampled vs Observed variables

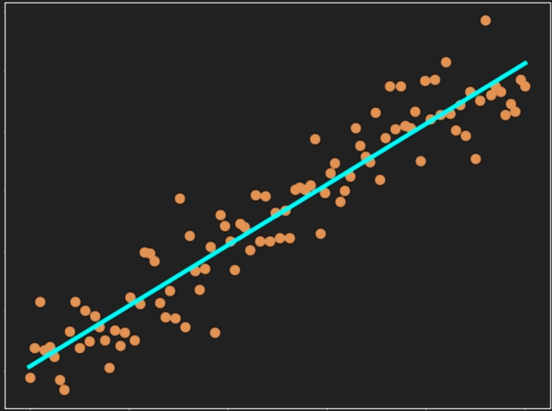
(~) is `observe` whenever observed data is provided for a random variable.
(~) is `sample` in all other cases.

Benefits of model-based PPLs:

We can interpret a model for simulation *and* inference

Disadvantage of query-based PPLs

Models are not first-class citizens



Composable, Modular, Probabilistic Models

We implement a probabilistic language in Haskell where:

- Models are interpretable for simulation *and* inference
- Models are first-class citizens - they can be combined and composed

To achieve this, we acknowledge the following ideals:

Ideal 1. Models should be syntactic descriptions of a data generative process.

a. Syntax for `sample` and `observe` should be unified.

Ideal 2. We need a clean mechanism of associating observed data to random variables.

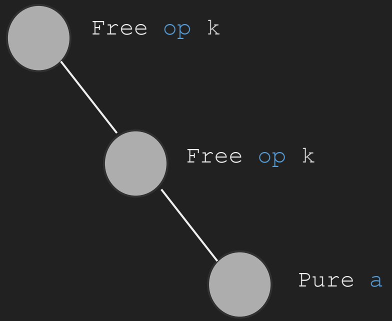
a. Observed data should only be provided when absolutely necessary
b. Observed data should not be passed as function arguments

Ideal 3. Simulation and inference should be higher-order functions which assign semantics to models.

Composable, Modular, Probabilistic Models

Ideal 1. **Models should be syntactic descriptions of a data generative process.**

a. Syntax for `sample` and `observe` should be unified.

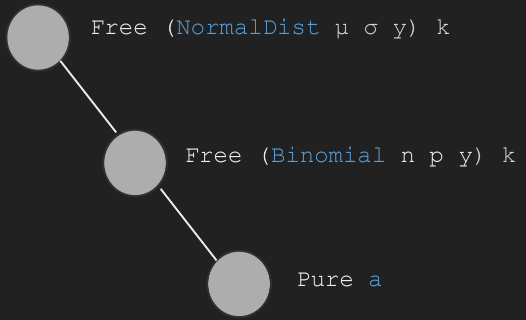


Solution 1. **Extensible Algebraic Effects**

Algebraic effects allow us to syntactically construct programs as trees where its nodes are shaped by some effectful operations.

Distribution Effects, for unifying the syntax of sampling and observing

```
data Dist a where
 NormalDist        :: Double -> Double     -> Maybe Double -> Dist Double
 BinomialDist      :: Int    -> Double     -> Maybe Int    -> Dist Int
```




```
newtype Model ts a = Model {
  runModel :: (Member Dist ts) => Freer ts a
}
```

Composable, Modular, Probabilistic Models

Ideal 2. **We need a clean mechanism of associating observed data to random variables.**

a. Observed data should only be provided when absolutely necessary
b. Observed data should not be passed as function arguments

Solution 2.

Extensible Environments

Extensible environments represent the observed variables of a model.

```
#μ @= [0.2] <: #σ @= [1.5] <: #y @= [] <: nil
```

We can specify the observed variables of a model via a type-class constraint:

```
Observable env "y" Double => ...
```

And then reference them inside a model:

```
  y <- normal 0 1 #y
```

Affine Reader Effects

We could use a Reader effect ...

But we can only sensibly call this once:

```
    y <- normal 0 1 #y
```

Instead we use an *Affine Reader effect,* which consumes read values.

```
newtype Model env ts a = Model {
   runModel :: (Member Dist ts,
                Member AffReader env ts) => Freer ts a }
```

Composable, Modular, Probabilistic Models

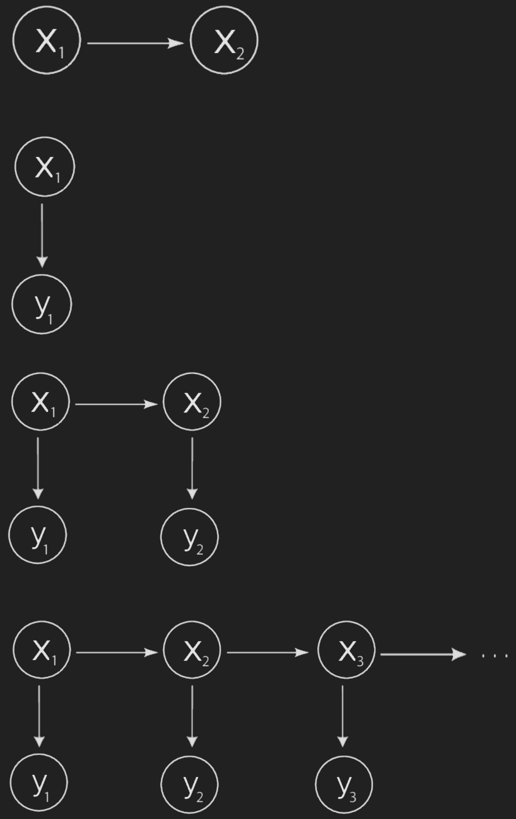
Example: Hidden Markov Model

```haskell
transitionModel :: Double -> Int -> Model env ts Int
transitionModel transition_p x_i = do
 dX <- bernoulli' transition_p
 let x_{i+1} = x_i + dX
 return x_{i+1}
```

```haskell
observationModel :: (Observable env "y" Int)
     => Double -> Int -> Model env ts Int
observationModel observation_p x = do
 binomial x observation_p #y
```

```haskell
hmm :: (Observable s "y" Int)
     => Double -> Double -> Int -> Model env ts Int
hmm transition_p observation_p x_i = do
 x_{i+1} <- transitionModel transition_p x_i
 y_{i+1} <- observationModel observation_p x_{i+1}
 return x_{i+1}
```

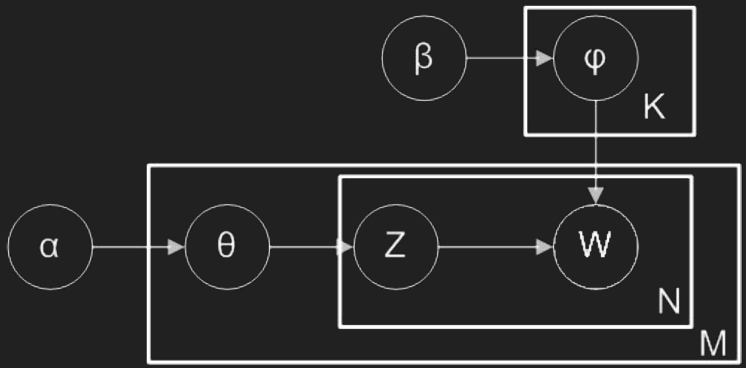
```haskell
hmmNSteps :: (Observable s "y" Int)
     => Double -> Double -> Int -> (Int -> Model env ts Int)
hmmNSteps transition_p observation_p n =
 foldl (>=>) return (replicate n (hmm transition_p observation_p))
```



Composable, Modular, Probabilistic Models

Example: Topic Model

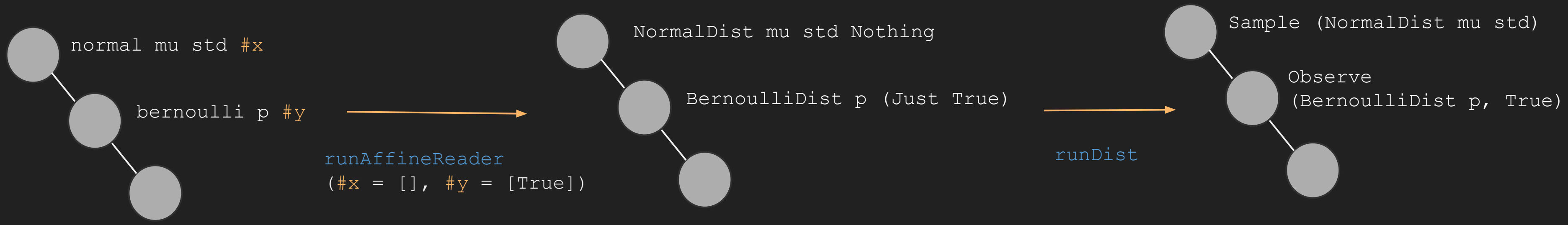
```
wordDist :: Observable env "w" String
 => [String] -> [Double] -> Model env ts String
wordDist vocab ps = categorical (zip vocab ps) #w

topicWordPrior :: Observable env "φ" [Double]
 => [String] -> Model env ts [Double]
topicWordPrior vocab
 = dirichlet (replicate (length vocab) 1) #φ

docTopicPrior :: Observable env "θ" [Double]
 => Int -> Model env ts [Double]
docTopicPrior n_topics = dirichlet (replicate n_topics 1) #θ


documentDist :: (Observables env '["φ", "θ"] [Double],
                 Observable  env "w" String)
 => [String] -> Int -> Int -> Model env ts [String]
documentDist vocab n_topics n_words = do
 topic_word_ps <- replicateM n_topics $ topicWordPrior vocab
 doc_topic_ps  <- docTopicPrior n_topics
 replicateM n_words (do  z <- discrete doc_topic_ps
                         let word_ps = topic_word_ps !! z
                         wordDist vocab word_ps)
```



Composable, Modular, Probabilistic Models

Ideal 3. **Simulation and inference should be higher-order functions which assign semantics to models.**

Solution 3. **Composable Program Transformations via Effect Handlers**



Theoretically, every algorithm could be implemented in terms of handlers for sample and observe.

Complex algorithms are better implemented via further program transformations.

We have used this approach to demonstrate:

- Simulation
- Likelihood Weighting
- Metropolis-Hastings