Modular Probabilistic Models
via Algebraic Effects

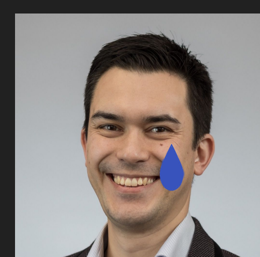
Minh Nguyen, Roly Perera, Meng Wang, Nicolas Wu

Modular Probabilistic Models

via Algebraic Effects

Minh Nguyen, Roly Perera, Meng Wang, Nicolas Wu



Probabilistic model: a set of relationships between random variables

$$x \longrightarrow \boxed{\text{model}(\theta)} \longrightarrow y$$

Probabilistic model: a set of relationships between random variables.

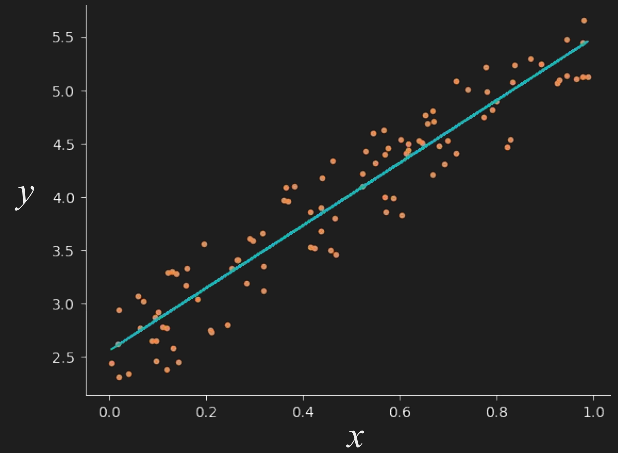
$$x \longrightarrow \boxed{\text{model}(\theta)} \longrightarrow y$$

Linear regression

input $\left\{ \lambda x. \right.$

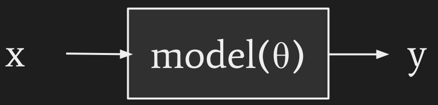
parameters $\left\{ \begin{array}{l} \mu \sim \text{Normal}(0, 3) \\ c \sim \text{Normal}(0, 2) \\ \sigma \sim \text{Uniform}(1, 3) \end{array} \right.$

output $\left\{ \; y \sim \text{Normal}(\mu * x + c, \sigma) \right.$



PPLs: Probabilistic models as programs

Probabilistic model: a set of relationships between random variables.

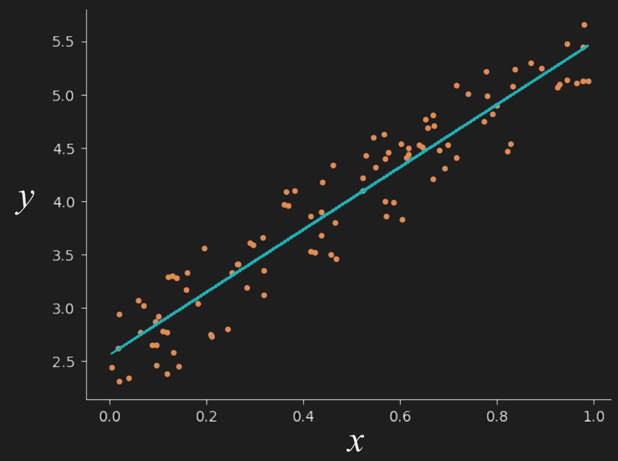


Linear regression

input $\{\ \lambda x.$

parameters $\left\{ \begin{array}{l} \mu \sim \mathrm{Normal}(0, 3) \\ c \sim \mathrm{Normal}(0, 2) \\ \sigma \sim \mathrm{Uniform}(1, 3) \end{array} \right.$

output $\{\ \mathrm{y} \sim \mathrm{Normal}(\mu * x + c, \sigma)$



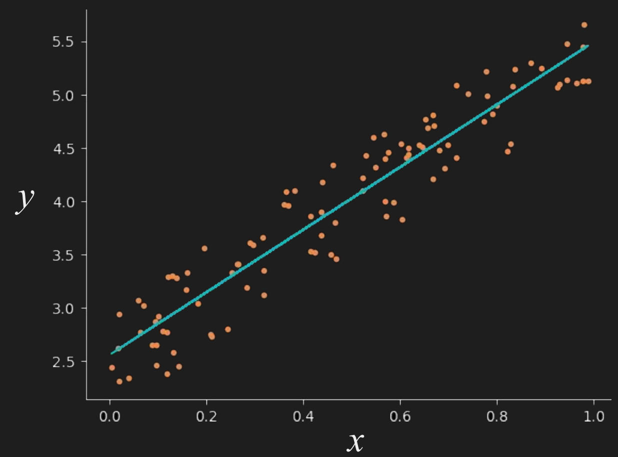
PPLs: Probabilistic models as programs

Probabilistic model: a set of relationships between random variables.

$$x \longrightarrow \boxed{\text{model}(\theta)} \longrightarrow y$$

Linear regression

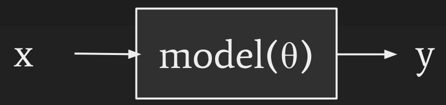
input	$\{\ \lambda x.$
parameters	$\mu \sim \text{Normal}(0, 3)$ $c \sim \text{Normal}(0, 2)$ $\sigma \sim \text{Uniform}(1, 3)$
output	$\{\ \text{y} \sim \text{Normal}(\mu * x + c, \sigma)$



What might linear regression look like as a probabilistic program?

PPLs: Probabilistic models as programs

Probabilistic model: a set of relationships between random variables.

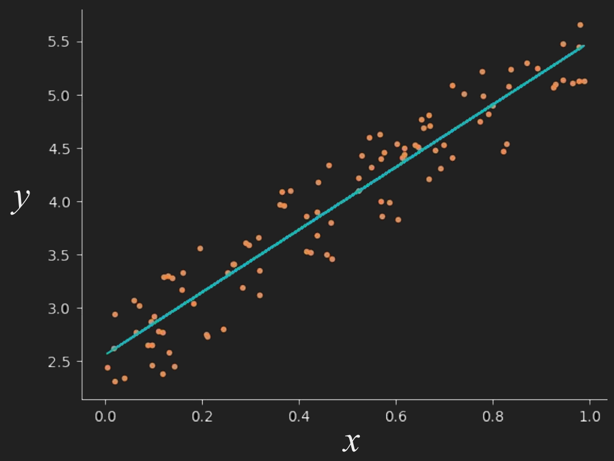


Linear regression

input $\{\ \lambda x.$

parameters $\left\{\begin{array}{l} \mu \sim \mathrm{Normal}(0, 3) \\ c \sim \mathrm{Normal}(0, 2) \\ \sigma \sim \mathrm{Uniform}(1, 3) \end{array}\right.$

output $\{\ \mathrm{y} \sim \mathrm{Normal}(\mu * x + c, \sigma)$



What might linear regression look like as a probabilistic program?

[Monad Bayes**]**

A possible simulation

```
linRegr x μ c σ = do
  y ← sample (normal (μ * x + c) σ)
  return y
```

PPLs: Probabilistic models as programs

Probabilistic model: a set of relationships between random variables.

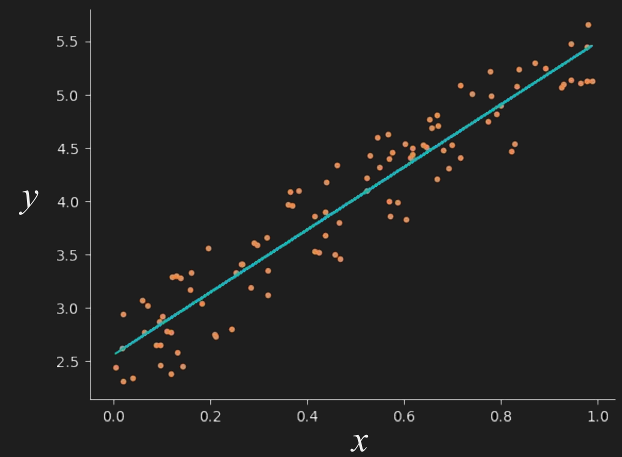
x → model(θ) → y

Linear regression

input { $\lambda x.$

parameters $\left\{ \begin{array}{l} \mu \sim \text{Normal}(0, 3) \\ c \sim \text{Normal}(0, 2) \\ \sigma \sim \text{Uniform}(1, 3) \end{array} \right.$

output { $\text{y} \sim \text{Normal}(\mu * x + c, \sigma)$



What might linear regression look like as a probabilistic program?

[Monad Bayes]

A possible simulation

```
linRegr x μ c σ = do
  y ← sample (normal (μ * x + c) σ)
  return y
```

A possible inference

```
linRegr x y = do
  μ  ← sample (normal 0 3)
  c  ← sample (normal 0 2)
  σ  ← sample (uniform 0 3)
  observe (normal (μ * x + c) σ) y
  return (μ, c, σ)
```

PPLs: Probabilistic models as programs

[WebPPL]

A possible simulation

```
var linRegr = function(x, mu, c, σ) {
 y = sample(Normal(mu * x + c, σ), y)
 return y
}
```

A possible inference

```
var linRegr = function(x, y) {
  mu     = sample(Normal(0, 3))
  c      = sample(Normal(0, 2))
  σ      = sample(Uniform(0, 2))
  observe(Normal(mu * x + c, σ), y)
  return (mu, c, σ)
}
```

[Anglican]

A possible simulation

```
(defquery linRegr [x, mu, σ, c]
(let [y (sample(normal (c + (* mu) x)) σ)]
 {:output y}))
```

A possible inference

```
(defquery linRegr [x mu-prior σ-prior c-prior y]
(let [mu    (sample mu-prior)
      c     (sample c-prior)
      σ     (sample σ-prior)
      pred (fn [x] (normal (c + (map (* mu) x)) σ))]
      (observe (predictive x) y)
{:mu mu :σ σ :predictor (pred x)}))
```

PPLs: Probabilistic models as programs

[WebPPL]

A possible simulation

```
var linRegr = function(x, mu, c, σ) {
 y = sample(Normal(mu * x + c, σ), y)
 return y
}
```

A possible inference

```
var linRegr = function(x, y) {
  mu    = sample(Normal(0, 3))
  c     = ~~sample(Normal(0, 2))~~ 0
  σ     = ~~sample(Uniform(0, 2))~~ 1
  observe(Normal(mu * x + c, σ), y)
  return (mu, c, σ)
}
```

[Anglican]

A possible simulation

```
(defquery linRegr [x, mu, σ, c]
(let [y (sample(normal (c + (* mu) x)) σ)]
 {:output y}))
```

A possible inference

```
(defquery linRegr [x mu-prior σ-prior c-prior y]
(let [mu    (sample mu-prior)
      c     (sample c-prior)
      σ     (sample σ-prior)
      pred (fn [x] (normal (c + (map (* mu) x)) σ))]
      (observe (predictive x) y)
{:mu mu :σ σ :predictor (pred x)}))
```

PPLs: Probabilistic models as programs

[WebPPL]

A possible simulation

```
var linRegr = function(x, mu, c, σ) {
 y = sample(Normal(mu * x + c, σ), y)
 return y
}
```

A possible inference

```
var linRegr = function(x, y) {
  mu    = sample(Normal(0, 3))
  c     = sample(Normal(0, 2)) 0
  σ     = sample(Uniform(0, 2)) 1
  observe(Normal(mu * x + c, σ), y)
  return (mu, c, σ)
}
```

[Anglican]

A possible simulation

```
(defquery linRegr [x, mu, σ, c]
(let [y (sample(normal (c + (* mu) x)) σ)]
 {:output y}))
```

A possible inference

```
(defquery linRegr [x mu-prior σ-prior c-prior y]
(let [mu    (sample mu-prior)
     c      (sample c-prior)
     σ      (sample σ-prior)
     pred (fn [x] (normal (c + (map (* mu) x)) σ))]
     (observe (predictive x) y)
{:mu mu :σ σ :predictor (pred x)}))
```

How about just one general-purpose model?

Motivation 1: Multimodal models

Multimodal model: a model whose random variables can be specialised to `sample` or `observe` modes

ProbFX: a Haskell PPL supporting multimodal models

Motivation 1: Multimodal models

Multimodal model: a model whose random variables can be specialised to `sample` or `observe` modes

ProbFX: a Haskell PPL supporting multimodal models

Linear regression

input $\{\ \lambda x.$

parameters $\left\{ \begin{array}{l} \mu \sim \mathrm{Normal}(0, 3) \\ c \sim \mathrm{Normal}(0, 2) \\ \sigma \sim \mathrm{Uniform}(1, 3) \end{array} \right.$

output $\{\ \mathrm{y} \sim \mathrm{Normal}(\mu * x + \mathrm{c}, \sigma)$

Linear regression in ProbFX

```
linRegr :: Observables env ["μ", "c", "σ", "y"] Double
        => [Double] -> Model env [Double]
linRegr xs = do
  μ  ← normal 0 3 #μ
  c  ← normal 0 2 #c
  σ  ← uniform 1 3 #σ
  ys ← mapM (λx → normal (μ * x + c) σ #y) xs
  return ys
```

Motivation 1: Multimodal models

Multimodal model: a model whose random variables can be specialised to `sample` or `observe` modes

ProbFX: a Haskell PPL supporting multimodal models

Linear regression

input $\{\ \lambda x.$

parameters $\left\{\begin{array}{l} \mu \sim \mathrm{Normal}(0, 3) \\ c \sim \mathrm{Normal}(0, 2) \\ \sigma \sim \mathrm{Uniform}(1, 3) \end{array}\right.$

output $\{\ \mathrm{y} \sim \mathrm{Normal}(\mu * x + \mathrm{c}, \sigma)$

Linear regression in ProbFX

```
linRegr :: Observables env ["μ", "c", "σ", "y"] Double
        => [Double] -> Model env [Double]
linRegr xs = do
  μ  ← normal 0 3 #μ
  c  ← normal 0 2 #c
  σ  ← uniform 1 3 #σ
  ys ← mapM (λx → normal (μ * x + c) σ #y) xs
  return ys
```

observable variable

Motivation 1: Multimodal models

Multimodal model: a model whose random variables can be specialised to `sample` or `observe` modes

ProbFX: a Haskell PPL supporting multimodal models

Linear regression

input $\{ \lambda x.$

parameters $\left\{ \begin{array}{l} \mu \sim \text{Normal}(0, 3) \\ c \sim \text{Normal}(0, 2) \\ \sigma \sim \text{Uniform}(1, 3) \end{array} \right.$

output $\{ \; y \sim \text{Normal}(\mu * x + c, \sigma)$

Linear regression in ProbFX

model environment

```
linRegr :: Observables env ["μ", "c", "σ", "y"] Double
        => [Double] -> Model env [Double]
linRegr xs = do
  μ  ← normal 0 3 #μ
  c  ← normal 0 2 #c
  σ  ← uniform 1 3 #σ
  ys ← mapM (λx → normal (μ * x + c) σ #y) xs
  return ys
```

observable variable

Motivation 1: Multimodal models

Linear regression in ProbFX

```
linRegr :: Observables env ["μ", "c", "σ", "y"] Double
        => [Double] -> Model env [Double]
linRegr xs = do
  μ  ← normal 0 3 #μ
  c  ← normal 0 2 #c
  σ  ← uniform 1 3 #σ
  ys ← mapM (λx → normal (μ * x + c) σ #y) xs
  return ys
```

Interacting with a ProbFX model

Motivation 1: Multimodal models

Linear regression in ProbFX

```
linRegr :: Observables env ["μ", "c", "σ", "y"] Double
        => [Double] -> Model env [Double]
linRegr xs = do
  μ  ← normal 0 3 #μ
  c  ← normal 0 2 #c
  σ  ← uniform 1 3 #σ
  ys ← mapM (λx → normal (μ * x + c) σ #y) xs
  return ys
```

Interacting with a ProbFX model

```
do -- | Simulation
   let xs     = [0 .. 100]
       env_in  = (#μ := [3]) • (#c := [0]) • (#σ := [1]) • (#y := [])
```

Motivation 1: Multimodal models

Linear regression in ProbFX

```
linRegr :: Observables env ["μ", "c", "σ", "y"] Double
        => [Double] -> Model env [Double]
linRegr xs = do
  μ  ← normal 0 3 #μ
  c  ← normal 0 2 #c
  σ  ← uniform 1 3 #σ
  ys ← mapM (λx → normal (μ * x + c) σ #y) xs
  return ys
```

Interacting with a ProbFX model

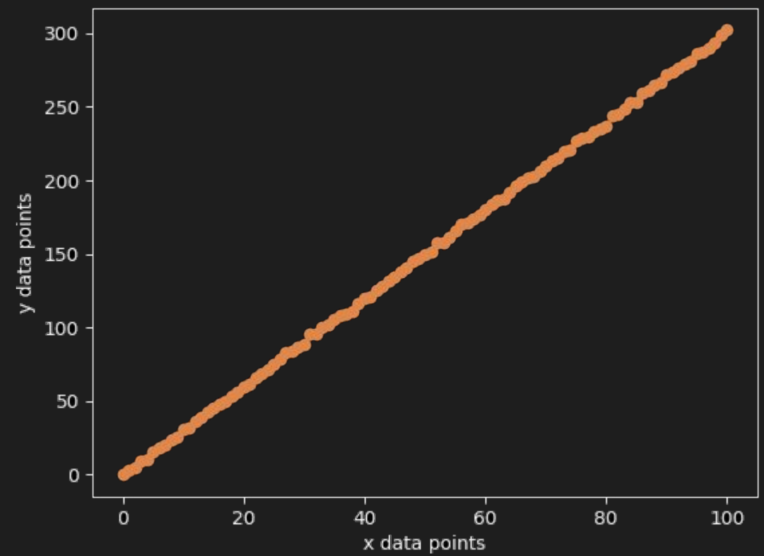
```
do -- | Simulation
   let xs     = [0 .. 100]
       env_in  = (#μ := [3]) • (#c := [0]) • (#σ := [1]) • (#y := [])
```

We **observe** μ,c,σ
We **sample** y

Motivation 1: Multimodal models

Linear regression in ProbFX

```
linRegr :: Observables env ["μ", "c", "σ", "y"] Double
        => [Double] -> Model env [Double]
linRegr xs = do
  μ  ← normal 0 3 #μ
  c  ← normal 0 2 #c
  σ  ← uniform 1 3 #σ
  ys ← mapM (λx → normal (μ * x + c) σ #y) xs
  return ys
```



Interacting with a ProbFX model

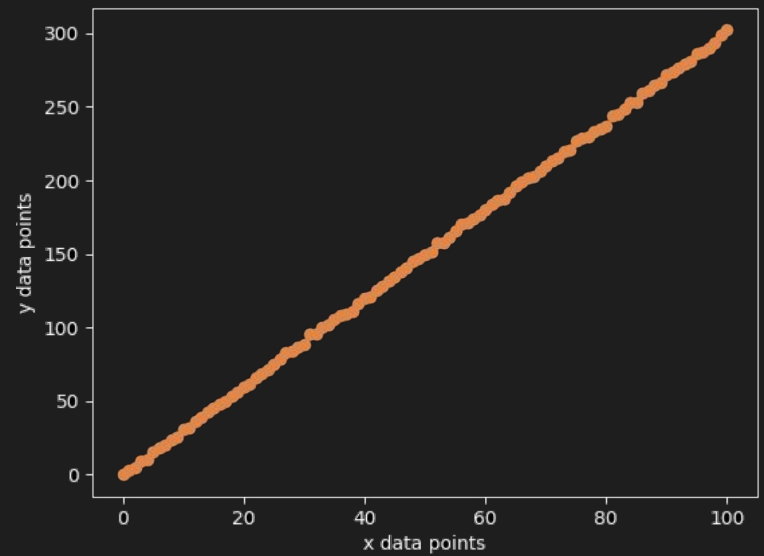
```
do -- | Simulation
   let xs      = [0 .. 100]
       env_in   = (#μ := [3]) • (#c := [0]) • (#σ := [1]) • (#y := [])
   ys :: [Double] ← simulate (linRegr xs) env_in
```

We **observe** μ,c,σ
We **sample** y

Motivation 1: Multimodal models

Linear regression in ProbFX

```
linRegr :: Observables env ["μ", "c", "σ", "y"] Double
        => [Double] -> Model env [Double]
linRegr xs = do
  μ  ← normal 0 3 #μ
  c  ← normal 0 2 #c
  σ  ← uniform 1 3 #σ
  ys ← mapM (λx → normal (μ * x + c) σ #y) xs
  return ys
```



Interacting with a ProbFX model

```
do -- | Simulation
   let xs     = [0 .. 100]
       env_in  = (#μ := [3]) • (#c := [0]) • (#σ := [1]) • (#y := [])
   ys :: [Double] ← simulate (linRegr xs) env_in

   -- | Inference
   let env_in  = (#μ := [])  • (#c := [])  • (#σ := [])  • (#y := ys)
```

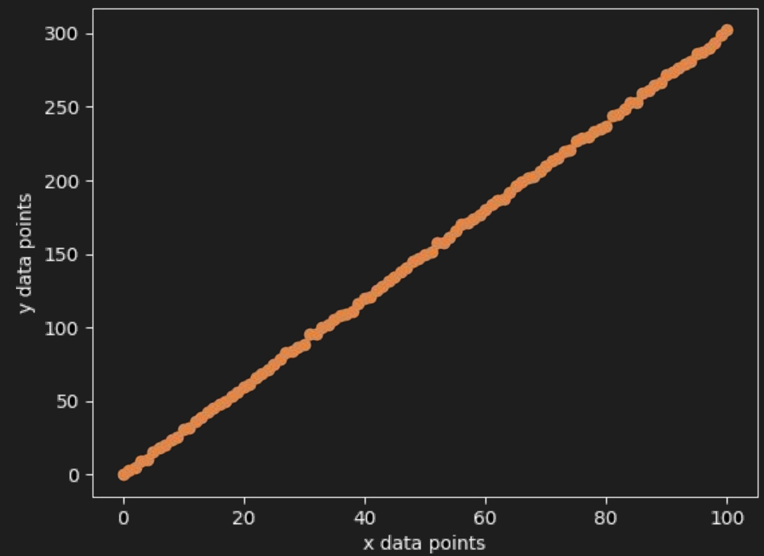
We **observe** μ, c, σ
We **sample** y

We **sample** μ, c, σ
We **observe** y

Motivation 1: Multimodal models

Linear regression in ProbFX

```
linRegr :: Observables env ["μ", "c", "σ", "y"] Double
        => [Double] -> Model env [Double]
linRegr xs = do
  μ  ← normal 0 3 #μ
  c  ← normal 0 2 #c
  σ  ← uniform 1 3 #σ
  ys ← mapM (λx → normal (μ * x + c) σ #y) xs
  return ys
```



Interacting with a ProbFX model

```
do -- | Simulation
   let xs     = [0 .. 100]
       env_in  = (#μ := [3]) • (#c := [0]) • (#σ := [1]) • (#y := [])
   ys :: [Double] ← simulate (linRegr xs) env_in

   -- | Inference
   let env_in  = (#μ := [])  • (#c := [])  • (#σ := [])  • (#y := ys)
   (env_outs, weights) ← lw 1000 (linRegr xs) env_in
```

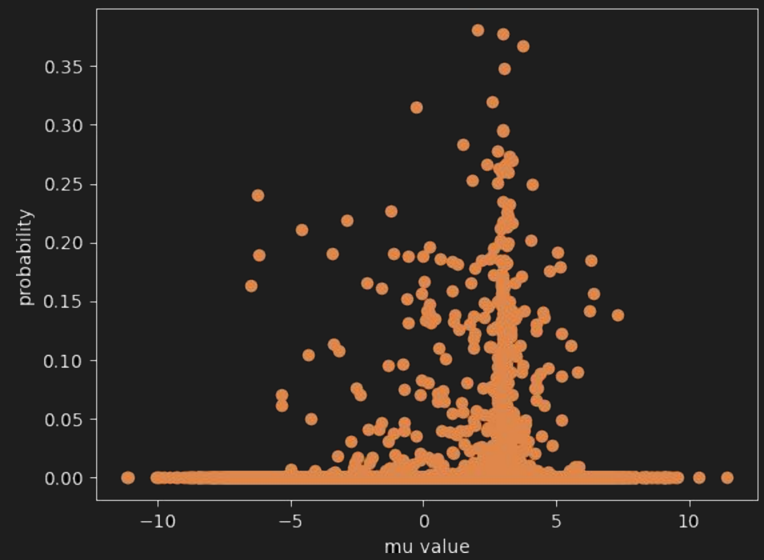
We **observe** μ, c, σ
We **sample** y

We **sample** μ, c, σ
We **observe** y

Motivation 1: Multimodal models

Linear regression in ProbFX

```
linRegr :: Observables env ["μ", "c", "σ", "y"] Double
        => [Double] -> Model env [Double]
linRegr xs = do
  μ  ← normal 0 3 #μ
  c  ← normal 0 2 #c
  σ  ← uniform 1 3 #σ
  ys ← mapM (λx → normal (μ * x + c) σ #y) xs
  return ys
```



Interacting with a ProbFX model

```
do -- | Simulation
   let xs      = [0 .. 100]
       env_in   = (#μ := [3]) • (#c := [0]) • (#σ := [1]) • (#y := [])
   ys :: [Double] ← simulate (linRegr xs) env_in

   -- | Inference
   let env_in   = (#μ := [])  • (#c := [])  • (#σ := [])  • (#y := ys)
   (env_outs, weights) ← lw 1000 (linRegr xs) env_in
   let μs = map (get #μ) env_outs
   return (μs, weights)
```

We **observe** μ, c, σ
We **sample** y

We **sample** μ, c, σ
We **observe** y

Motivation 2: Compositional models

Support for multimodal models already exists…

Supported model features	ProbFX	Gen	Turing	Stan	Pyro
Multimodal	●	●	●	●	●
Modular	●	●	●	○	●
Higher-order	●	◐	○	○	◐
Type-safe	●	○	○	●	○

● Full support
◐ Partial support
○ No support

Motivation 2: Compositional models

Support for multimodal models already exists…

Supported model features	ProbFX	Gen	Turing	Stan	Pyro
Multimodal	●	●	●	●	●
Modular	●	●	●	○	●
Higher-order	●	◐	○	○	◐
Type-safe	●	○	○	●	○

● Full support

◐ Partial support

○ No support

But models are generally not first-class citizens

[Turing]

```
@model function linRegr(x, mu, c, σ, y)
  y ~ Normal(mu * x + c, σ)
end
```

Motivation 2: Compositional models

Support for multimodal models already exists…

Supported model features	ProbFX	Gen	Turing	Stan	Pyro
Multimodal	●	●	●	●	●
Modular	●	●	●	○	●
Higher-order	●	◐	○	○	◐
Type-safe	●	○	○	●	○

● Full support
◐ Partial support
○ No support

But models are generally not first-class citizens or are not statically typed

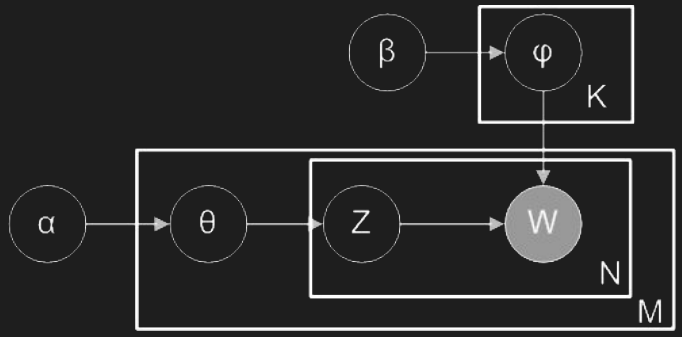
[Turing]

```
@model function linRegr(x, mu, c, σ, y)
  y ~ Normal(mu * x + c, σ)
end
```

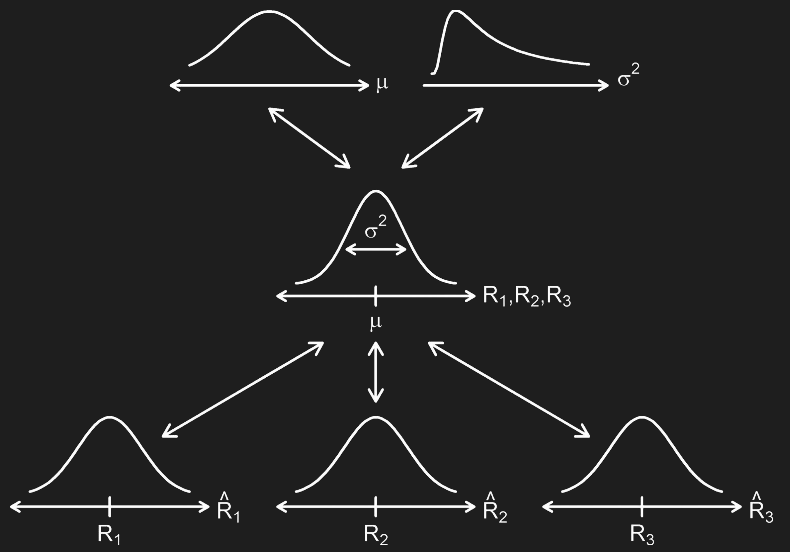
[Pyro]

```
def linRegr(x, mu, c, σ):
  pyro.sample("y", dist.Normal(mu * x + c, σ))
  return y

pyro.condition(linRegr, {"hotdog" = True}) ??
```

What about compositional, higher-order models?

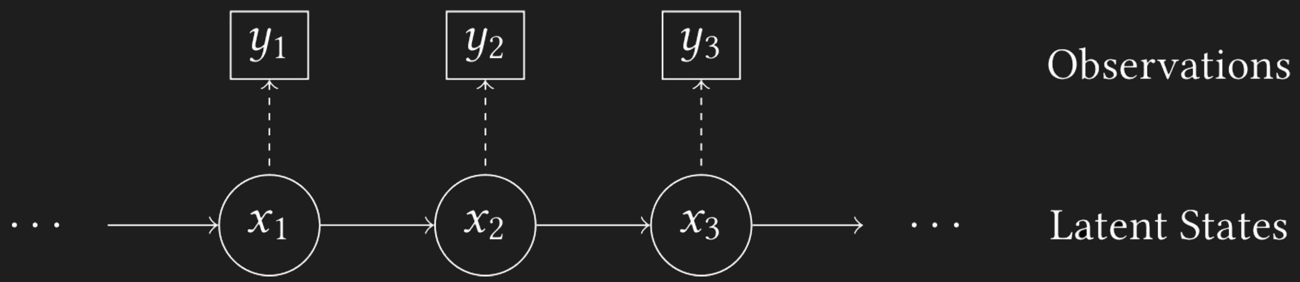


What about compositional, higher-order models?



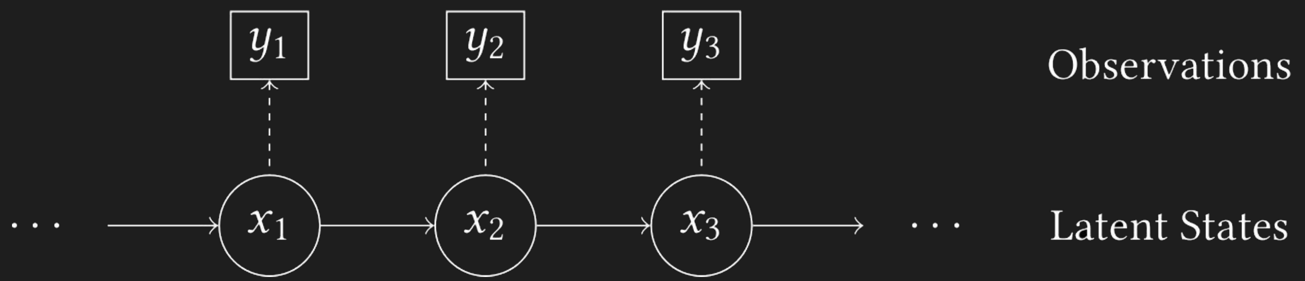
Motivation 2: Compositional models

Hidden Markov Model (HMM)



Motivation 2: Compositional models

Hidden Markov Model (HMM)

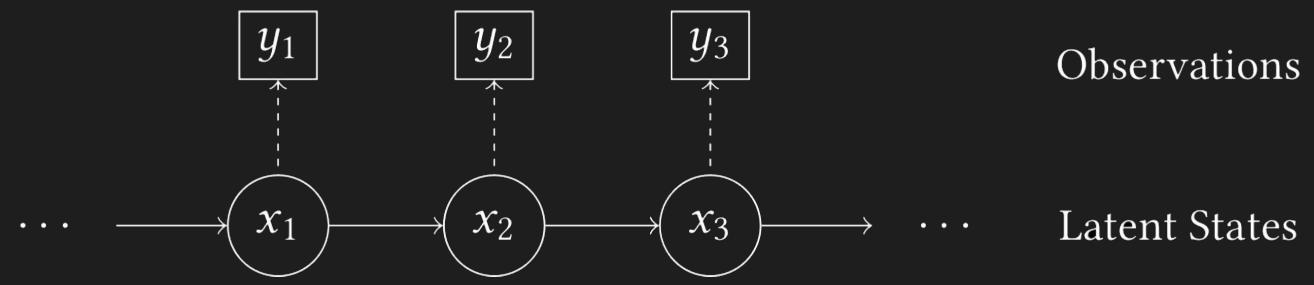


We can decompose this into two sub-models:

```
type TransModel env x   = x -> Model env x
type ObsModel   env x y = x -> Model env y
```

Motivation 2: Compositional models

Hidden Markov Model (HMM)



We can decompose this into two sub-models:

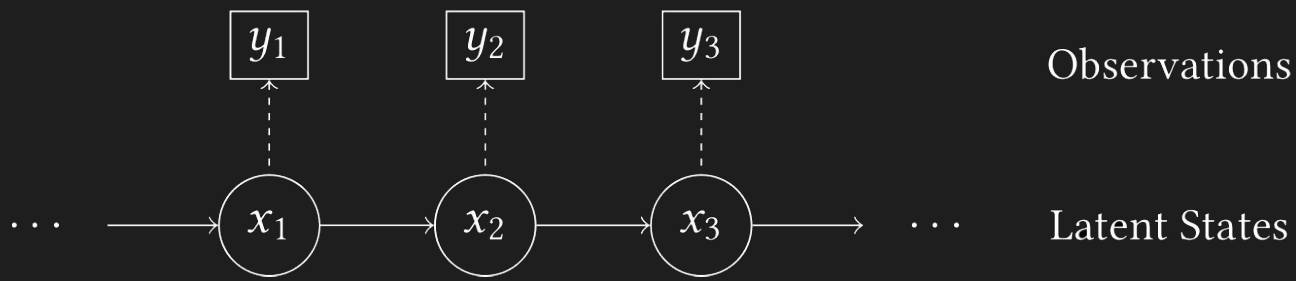
```
type TransModel env x   = x -> Model env x
type ObsModel   env x y = x -> Model env y
```

and then define a HMM as a higher-order model:

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
hmm transModel obsModel n x0 = do
```

Motivation 2: Compositional models

Hidden Markov Model (HMM)



We can decompose this into two sub-models:

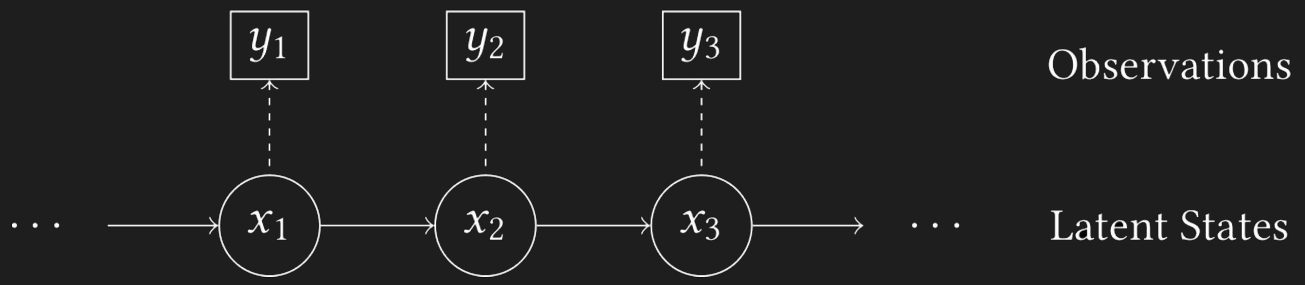
```
type TransModel env x   = x -> Model env x
type ObsModel   env x y = x -> Model env y
```

and then define a HMM as a higher-order model:

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
hmm transModel obsModel n x0 = do
  let hmmNode x = do x' <- transModel x
                     y' <- obsModel x'
                     return x'
```

Motivation 2: Compositional models

Hidden Markov Model (HMM)



We can decompose this into two sub-models:

```
type TransModel env x   = x -> Model env x
type ObsModel   env x y = x -> Model env y
```

and then define a HMM as a higher-order model:

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
hmm transModel obsModel n x0 = do
  let hmmNode x = do x' <- transModel x
                     y' <- obsModel x'
                     return x'
  foldl (>=>) return (replicate n hmmNode) x0
```

```
(>=>) :: (a -> Model env b)
      -> (b -> Model env c)
      -> (a -> Model env c)
```

Modelling an Epidemic: the SIR model

Modelling an Epidemic: the SIR model

Setting: We assume a fixed total population of **S**usceptible, **I**nfected, and **R**ecovered individuals.

Modelling an Epidemic: the SIR model

Setting: We assume a fixed total population of **S**usceptible, **I**nfected, and **R**ecovered individuals.

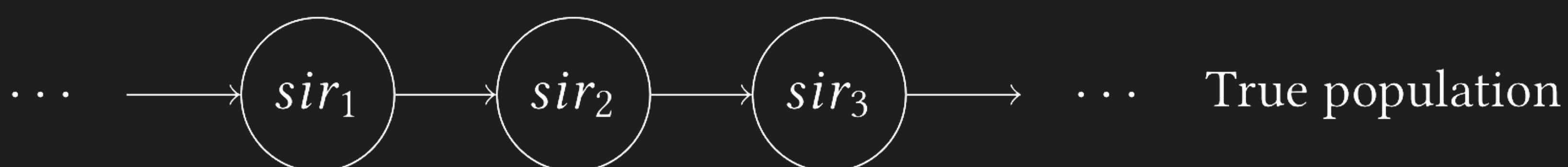
The SIR model: During an epidemic, how do these populations vary over time (days)?

Modelling an Epidemic: the SIR model

Setting: We assume a fixed total population of **S**usceptible, **I**nfected, and **R**ecovered individuals.


```
data Population  = Population { s :: Int,
                                i :: Int,
                                r :: Int }
```

The SIR model: During an epidemic, how do these populations vary over time (days)?

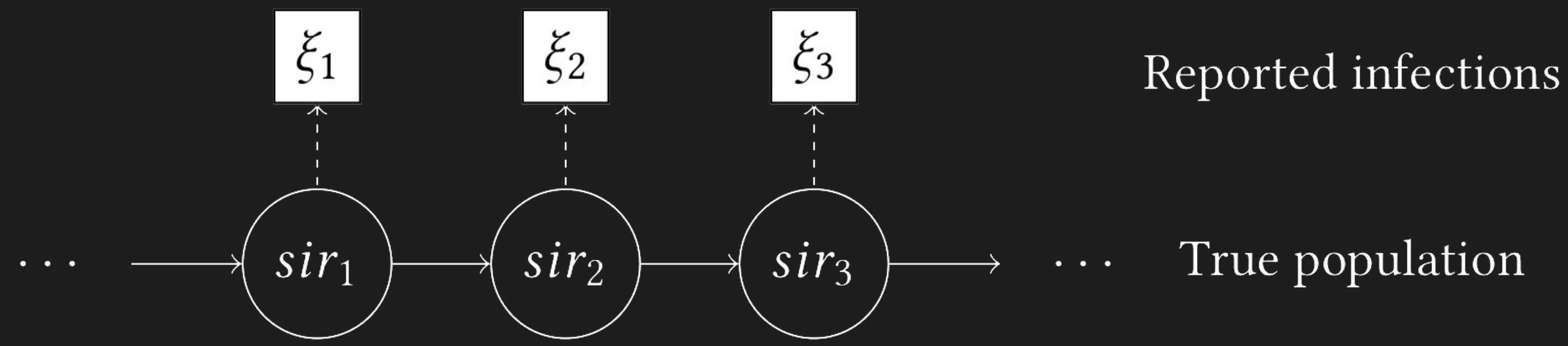


Modelling an Epidemic: the SIR model

Setting: We assume a fixed total population of **S**usceptible, **I**nfected, and **R**ecovered individuals.


```
type ReportedInf = Int

data Population  = Population { s :: Int,
                                i :: Int,
                                r :: Int }
```

The SIR model: During an epidemic, how do these populations vary over time (days)?



Modelling an Epidemic: the SIR model

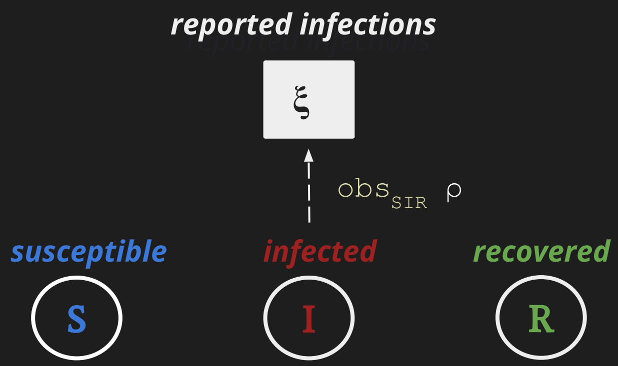
SIR observation model


```
type ReportedInf = Int

data Population  = Population { s :: Int,
                                i :: Int,
                                r :: Int }
```

```
type ObsModel env x y = x -> Model env y
```

Modelling an Epidemic: the SIR model

SIR observation model




```
type ReportedInf = Int
```

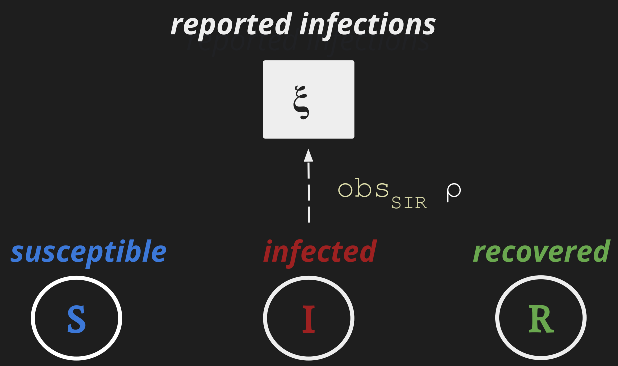
```
data Population  = Population { s :: Int,
                                i :: Int,
                                r :: Int }
```

```
type ObsModel env x y = x -> Model env y
```

```
obsSIR :: Observable env "ξ" Int => Double -> ObsModel env Population ReportedInf
obsSIR ρ (Population _ i _)  = poisson (ρ * i) #ξ
```

Modelling an Epidemic: the SIR model

SIR transition model



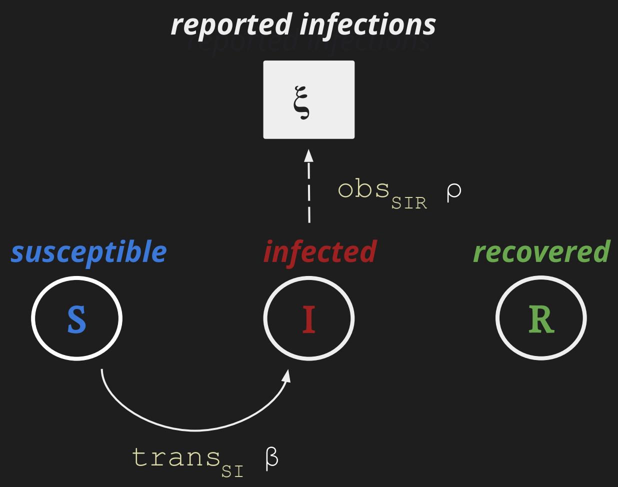

```
type ReportedInf = Int
```

```
data Population  = Population { s :: Int,
                                i :: Int,
                                r :: Int }
```

```
type TransModel env x = x -> Model env x
```

Modelling an Epidemic: the SIR model

SIR transition model




```
type ReportedInf = Int
```

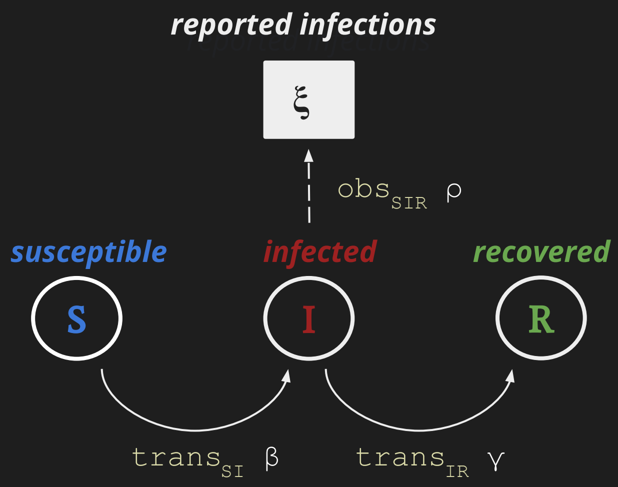
```
data Population  = Population { s :: Int,
                                 i :: Int,
                                 r :: Int }
```

```
type TransModel env x = x -> Model env x
```

```
trans_SI :: Double -> TransModel env Population
trans_SI β (Population s i r) = do
  δ_SI ← binomial' s (1.0 - exp (-β * i / s + i + r))
  return $ Population (s - δ_SI) (i + δ_SI) r
```

Modelling an Epidemic: the SIR model

SIR transition model



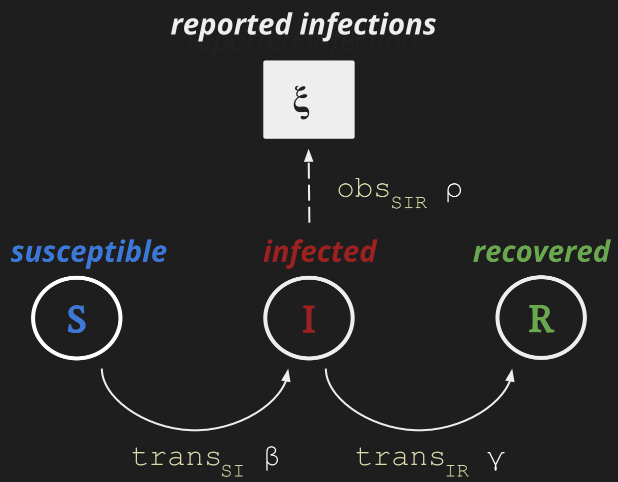

```
type ReportedInf = Int

data Population  = Population { s :: Int,
                                i :: Int,
                                r :: Int }
```

```
type TransModel env x = x -> Model env x
```

```
transSI :: Double -> TransModel env Population
transSI β (Population s i r) = do
  δSI ← binomial' s (1.0 - exp (-β * i / s + i + r))
  return $ Population (s - δSI) (i + δSI) r

transIR :: Double -> TransModel env Population
transIR γ (Population s i r) = do
  δIR ← binomial' i (1.0 - exp (-γ))
  return $ Population s (i - δIR) (r + δIR)
```

Modelling an Epidemic: the SIR model

SIR transition model




```
type ReportedInf = Int

data Population  = Population { s :: Int,
                                i :: Int,
                                r :: Int }
```

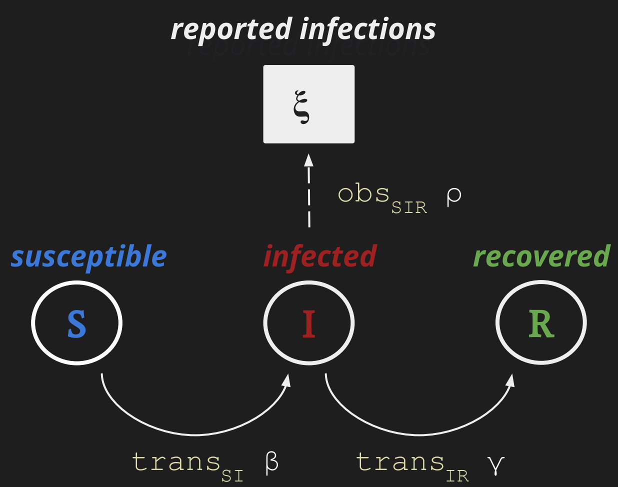
```
type TransModel env x = x -> Model env x
```

```
transSI :: Double -> TransModel env Population
transSI β (Population s i r) = do
  δSI ← binomial' s (1.0 - exp (-β * i / s + i + r))
  return $ Population (s - δSI) (i + δSI) r

transIR :: Double -> TransModel env Population
transIR γ (Population s i r) = do
  δIR ← binomial' i (1.0 - exp (-γ))
  return $ Population s (i - δIR) (r + δIR)
```

```
transSIR :: Double
         -> Double
         -> TransModel env Population
transSIR β γ = transSI β >=> transIR γ
```

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model

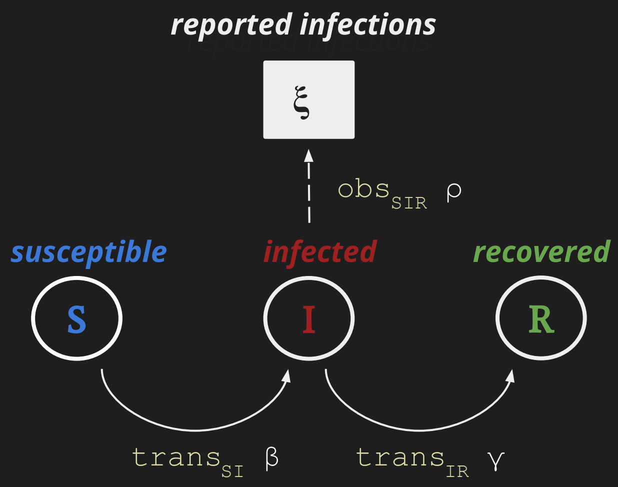



```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

```
obsSIR ρ (Population _ i _)
  = poisson (ρ * i) #ξ

transSIR β γ
  = transSI β >=> transIR γ
```

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model

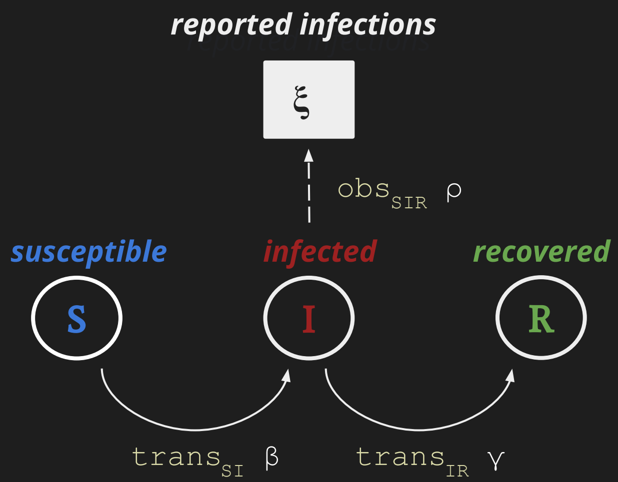



```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

$$\left.\begin{aligned}&\mathtt{obs}_{SIR}\ \rho\ \mathtt{(Population\ \_\ i\ \_)}\\&\quad = \mathtt{poisson}\ (\rho\ *\ \mathtt{i})\ \#\xi\\&\mathtt{trans}_{SIR}\ \beta\ \gamma\\&\quad = \mathtt{trans}_{SI}\ \beta\ \mathtt{>=>}\ \mathtt{trans}_{IR}\ \gamma\end{aligned}\right\}\quad \mathtt{sirModel} = \mathtt{hmm}\ (\mathtt{trans}_{SIR}\ \beta\ \gamma)\ (\mathtt{obs}_{SIR}\ \rho)$$

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model



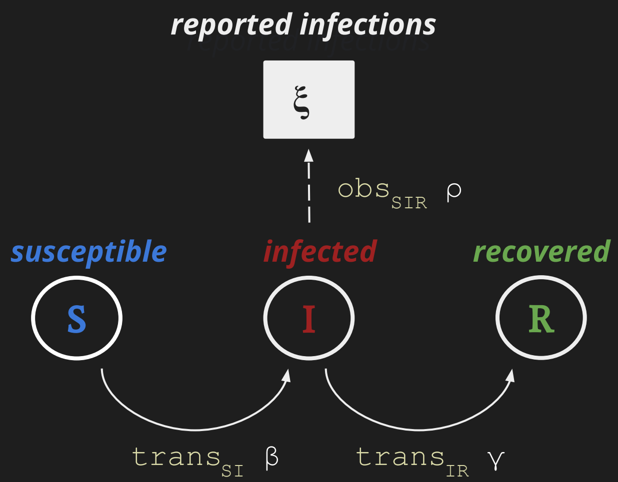

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

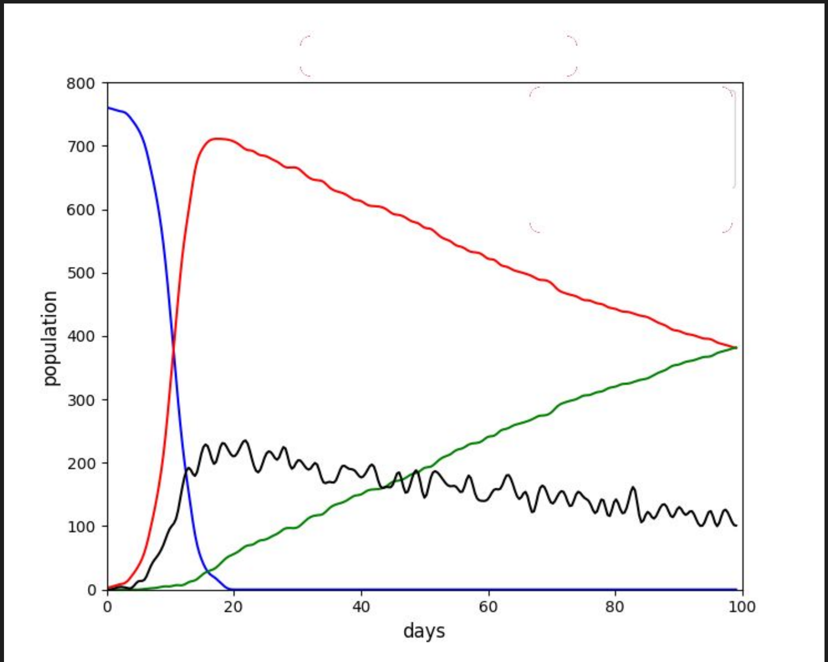
```
obs_SIR ρ (Population _ i _)
  = poisson (ρ * i) #ξ

trans_SIR β γ
  = trans_SI β >=> trans_IR γ
```

```
sirModel = hmm (trans_SIR β γ) (obs_SIR ρ)
                 100 (Population {s = 762, i = 1, r = 0})
```

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model





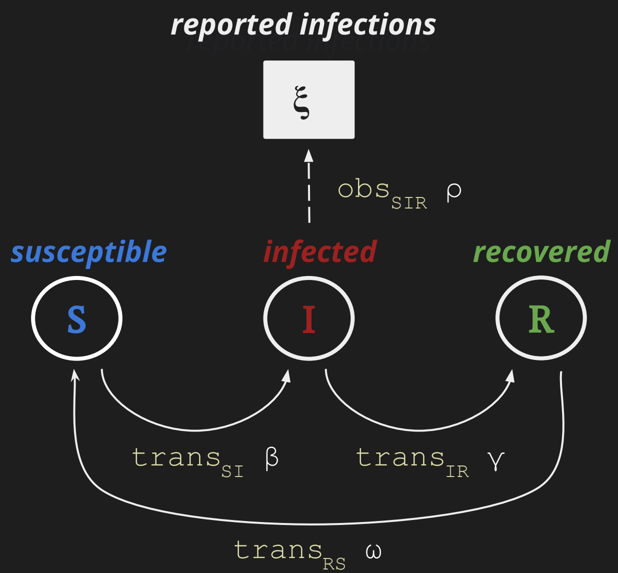

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

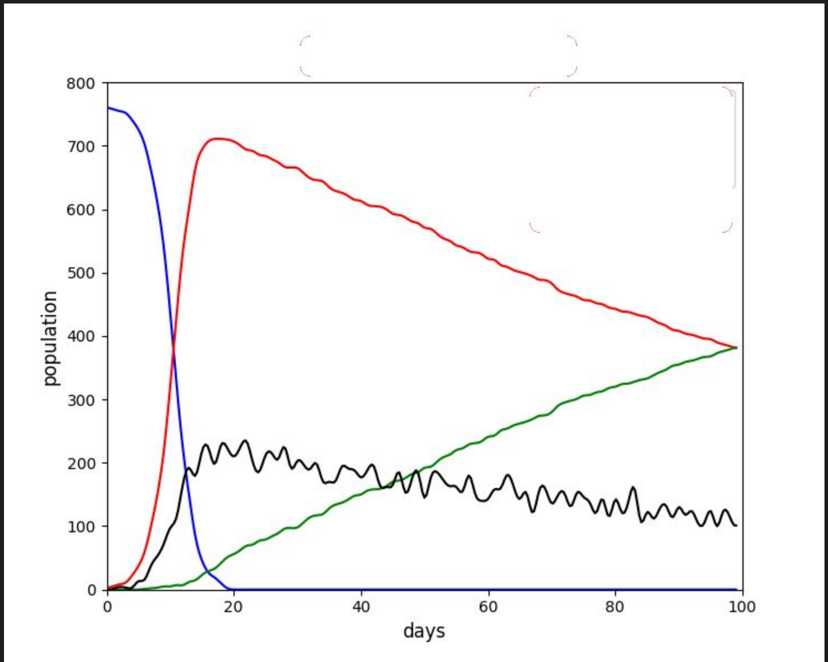
```
obsSIR ρ (Population _ i _)
  = poisson (ρ * i) #ξ

transSIR β γ
  = transSI β >=> transIR γ
```

```
sirModel = hmm (transSIR β γ) (obsSIR ρ)
               100 (Population {s = 762, i = 1, r = 0})
```

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model





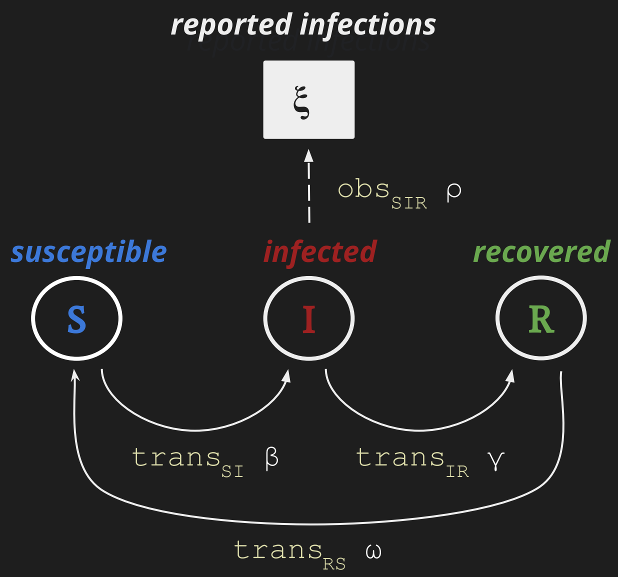

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

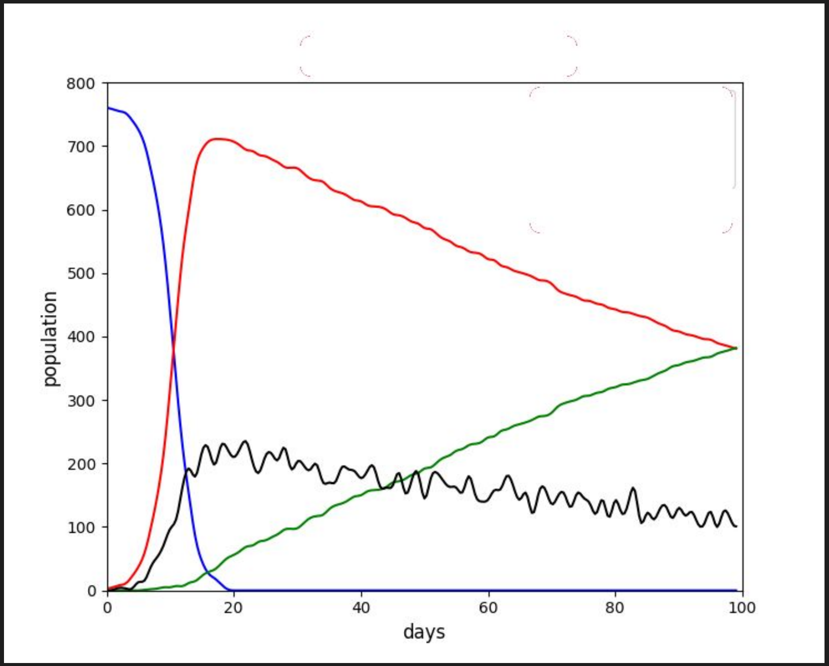
```
obs_SIR ρ (Population _ i _)
  = poisson (ρ * i) #ξ

trans_SIR β γ
  = trans_SI β >=> trans_IR γ
```

```
sirModel = hmm (trans_SIR β γ) (obs_SIR ρ)
               100 (Population {s = 762, i = 1, r = 0})
```

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model





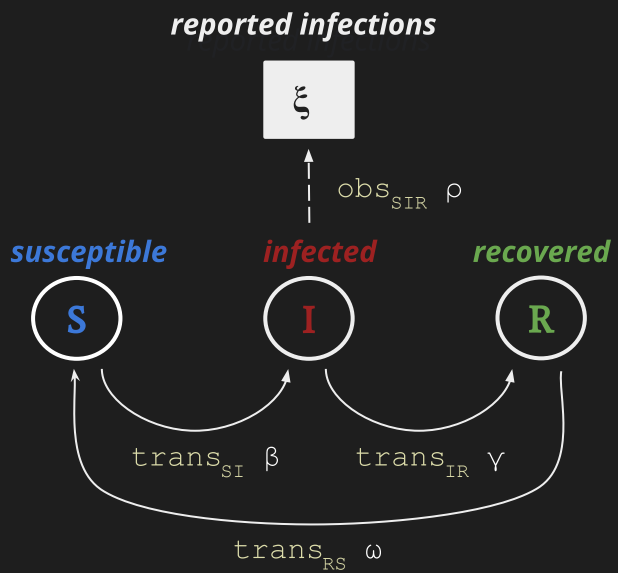

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

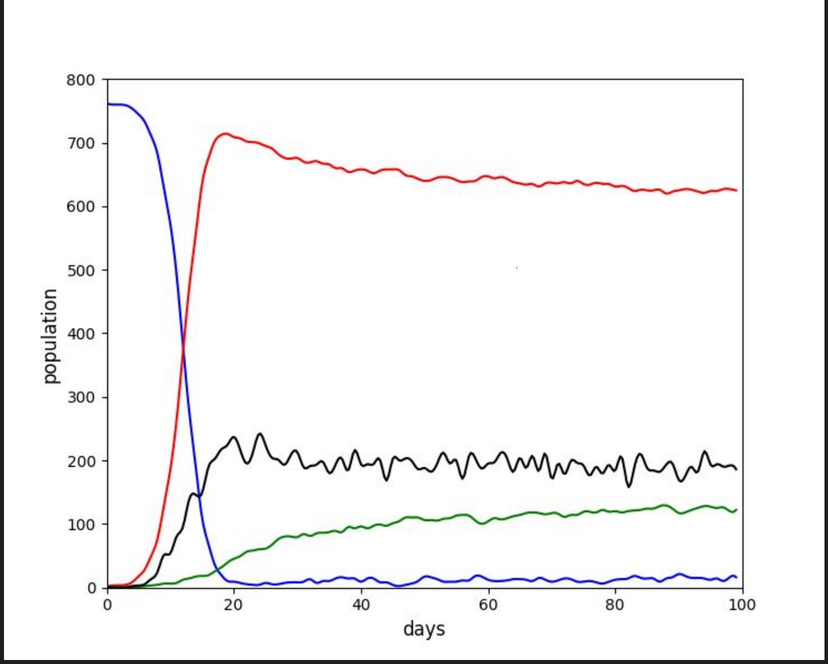
```
obs_SIR ρ (Population _ i _)
  = poisson (ρ * i) #ξ

trans_SIR β γ ω
  = trans_SI β >=> trans_IR γ
               >=> trans_RS ω
```

```
sirModel = hmm (trans_SIR β γ ω) (obs_SIR ρ)
               100 (Population {s = 762, i = 1, r = 0})
```

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model





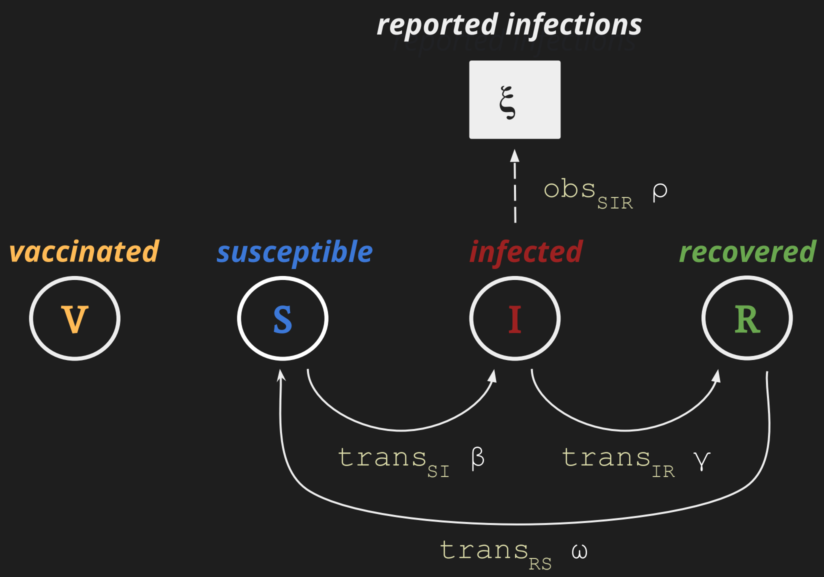

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

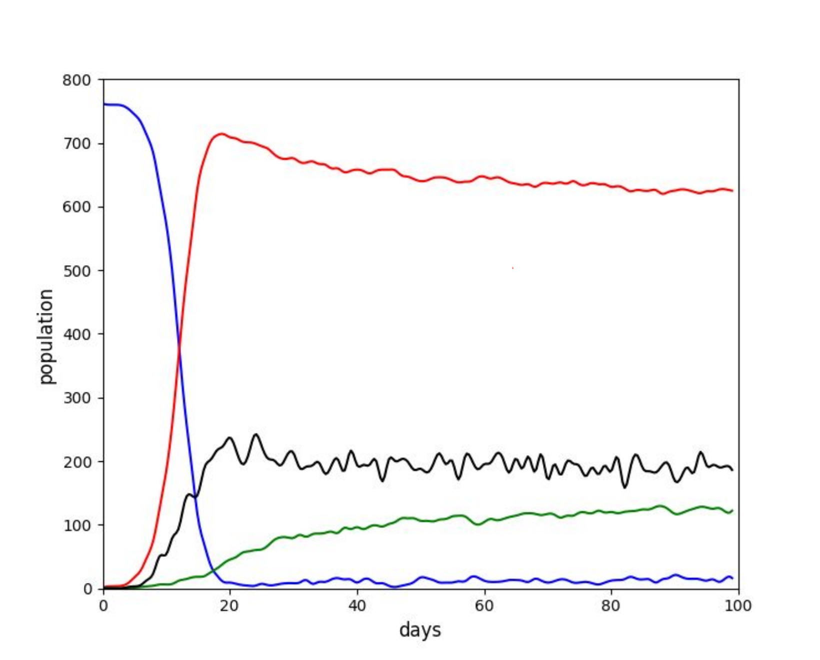
```
obs_SIR ρ (Population _ i _)
  = poisson (ρ * i) #ξ

trans_SIR β γ ω
  = trans_SI β >=> trans_IR γ
               >=> trans_RS ω
```

```
sirModel = hmm (trans_SIR β γ ω) (obs_SIR ρ)
               100 (Population {s = 762, i = 1, r = 0})
```

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model





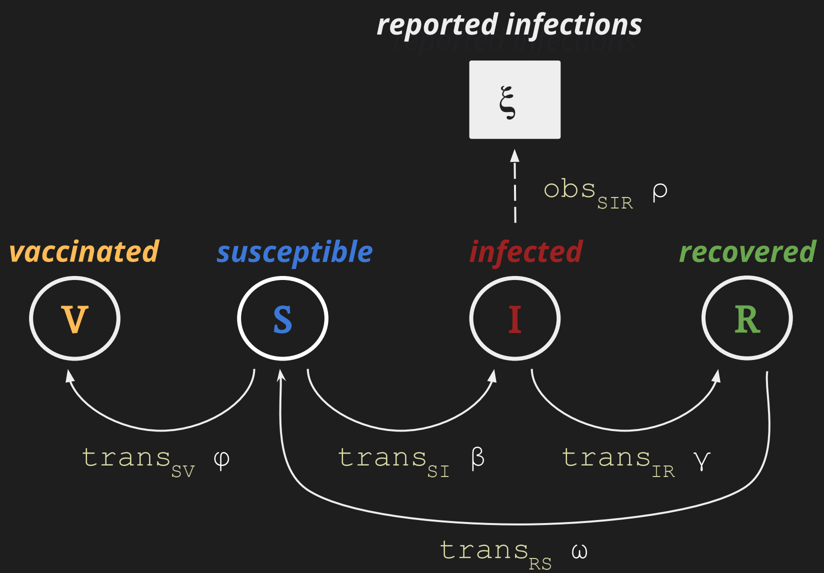

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

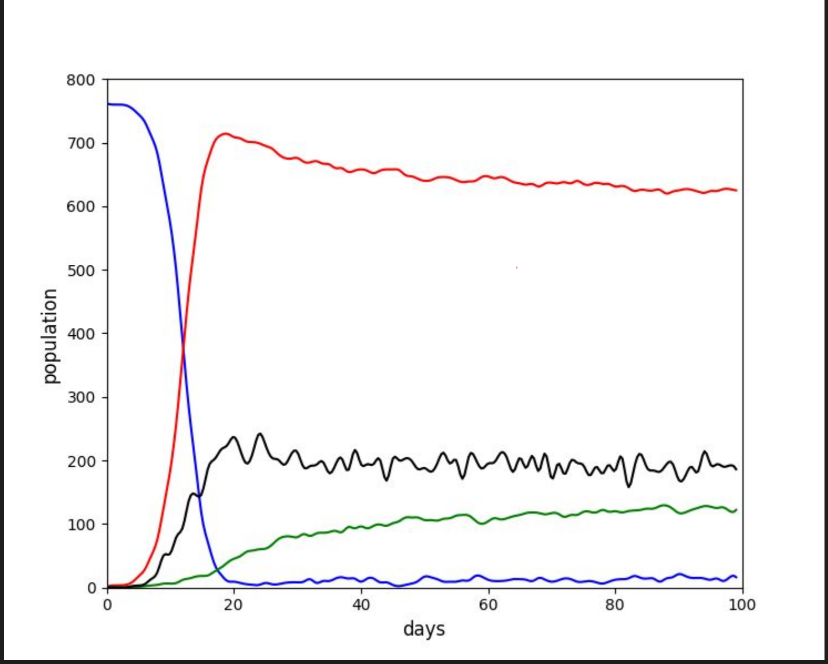
```
obsSIR ρ (Population _ i _ _)
  = poisson (ρ * i) #ξ

transSIR β γ ω
  = transSI β >=> transIR γ
              >=> transRS ω
```

```
sirModel = hmm (transSIR β γ ω) (obsSIR ρ)
                    100 (Population {s = 762, i = 1, r = 0})
```

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model





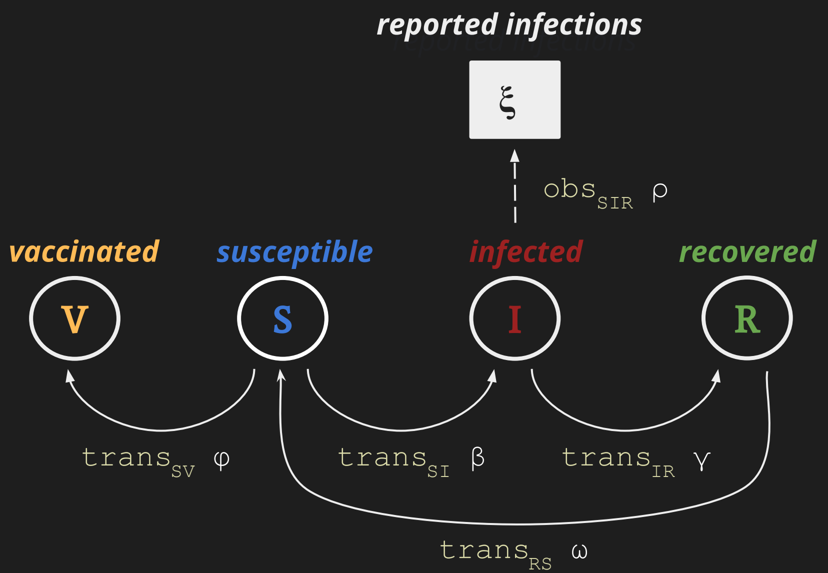

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

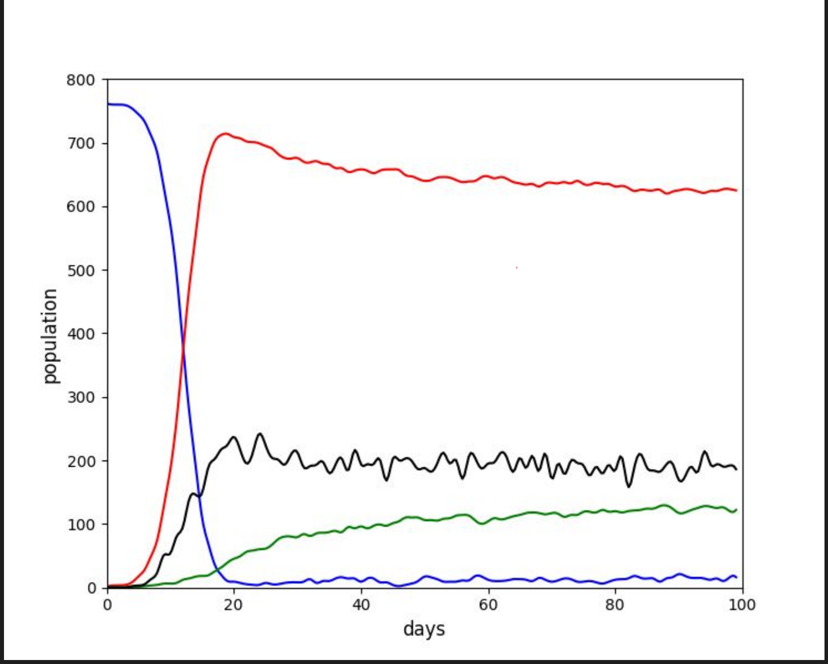
```
obs_SIR ρ (Population _ i _ _)
  = poisson (ρ * i) #ξ

trans_SIR β γ ω
  = trans_SI β >=> trans_IR γ
               >=> trans_RS ω
```

```
sirModel = hmm (trans_SIR β γ ω) (obs_SIR ρ)
                   100 (Population {s = 762, i = 1, r = 0})
```

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model





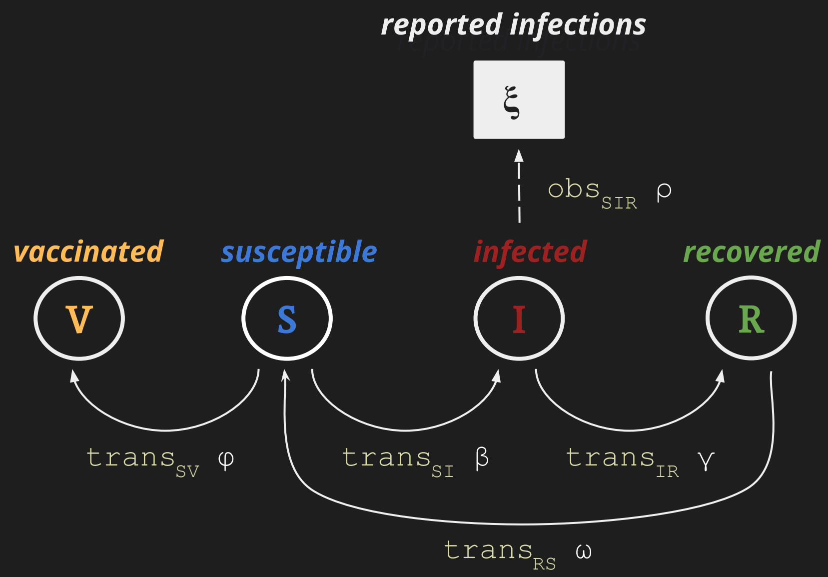

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

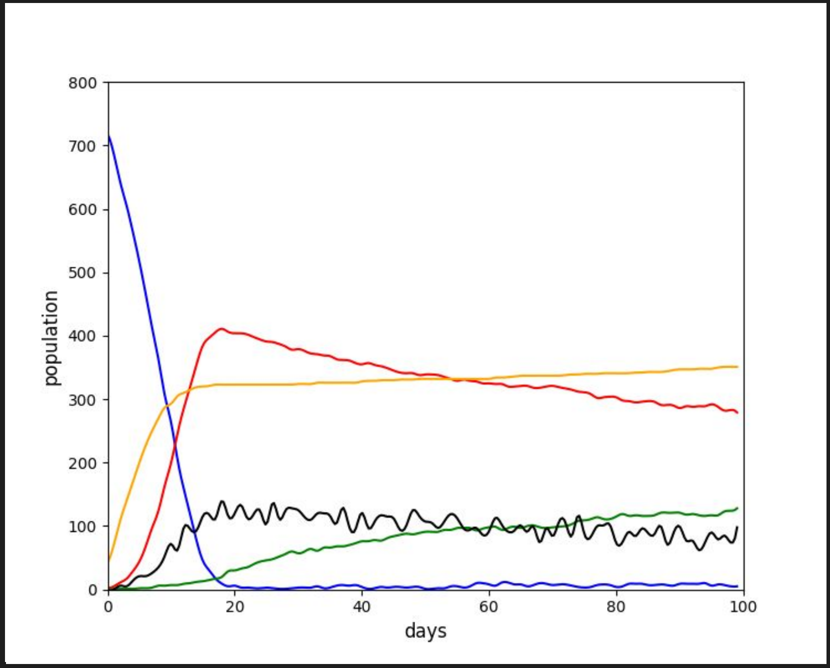
```
obs_SIR ρ (Population _ i _ _)
  = poisson (ρ * i) #ξ

trans_SIR β γ ω φ
  = trans_SI β >=> trans_IR γ
               >=> trans_RS ω
               >=> trans_SV φ
```

```
sirModel = hmm (trans_SIR β γ ω φ) (obs_SIR ρ)
                 100 (Population {s = 762, i = 1, r = 0, v = 0})
```

Modelling an Epidemic: the SIR model

SIR as a Hidden Markov Model





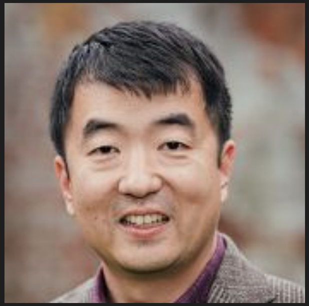

```
hmm :: TransModel env x -> ObsModel env x y -> Int -> x -> Model env x
```

```
obs_SIR ρ (Population _ i _ _)
  = poisson (ρ * i) #ξ

trans_SIR β γ ω φ
  = trans_SI β >=> trans_IR γ
               >=> trans_RS ω
               >=> trans_SV φ
```

```
sirModel = hmm (trans_SIR β γ ω φ) (obs_SIR ρ)
               100 (Population {s = 762, i = 1, r = 0, v = 0})
```

Minh Nguyen, Roly Perera, Meng Wang, Nicolas Wu






[github.com/min-nguyen/prob-fx]

Models

```
type Model env es a = (Member Dist es, Member (ObsReader env) es) => Prog es a
```

Primitive distributions

Reading observable variables

Effect signature

Smart constructors for primitive distributions

```
coinFlip :: (Observable env "p" Double,
             Observable env "y" Bool)
         => Model env es Bool
coinFlip = do
  p <- uniform 0 1 #p
  y <- bernoulli p #y
  return y
```

desugars to

```
coinFlip = do
  maybe_p <- call (Ask #p)
  p       <- call (Uniform 0 1 maybe_p)
  maybe_y <- call (Ask #y)
  y       <- call (Bernoulli p maybe_y)
  return y
```

handles to

```
(#p := [])
(#y := [True])
```

```
coinFlip = do
  p       <- call (Sample (Uniform 0 1))
  y       <- call (Observe (Bernoulli p) True)
  return y
```

Model environments

```
data Env env where
  ENil  :: Env '[]
  ECons :: [a] -> Env env -> Env ((x := a) : env)

data Assign x a = x := a

-- For example:
(#μ := [3.0]) • (#c := [0.0]) • (#σ := [1.0]) • (#y := []) • nil
  :: Env ["μ" := Double, "c" := Double , "σ" := Double, "y" := Double]
```

Observable variables

```
data ObsVar (x :: Symbol) where
 ObsVar :: KnownSymbol x => ObsVar x


-- For example:
#foo :: ObsVar "foo"
```