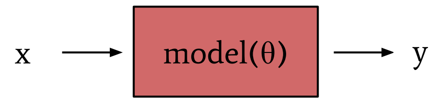
Modular Probabilistic Models
via Algebraic Effects

Minh Nguyen, Roly Perera, Meng Wang, Nicolas Wu

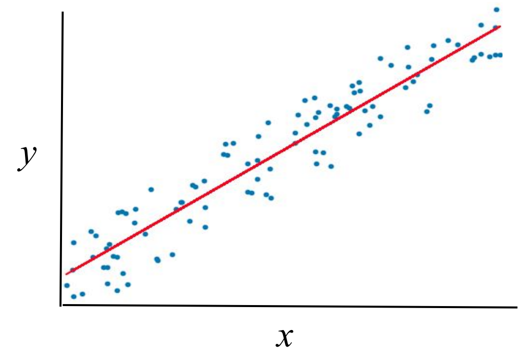
University of Bristol

Interacting with a probabilistic model

A *probabilistic model* is a set of relationships between certain random variables:



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



What might this look like in a PPL (probabilistic programming language)?

<u>A possible simulation</u>

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

<u>A possible inference</u>

```
inferLinRegr 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)
```

new model interactions
=
new model implementations

Interacting with a probabilistic model

Simulation in WebPPL

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

Inference in WebPPL

```
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)
}
```

Simulation in Anglican

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

Inference in Anglican

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

Motivation 1: Multimodal models

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

Wasabaye: A Haskell PPL for multimodal models

Example: Linear regression

Inputs $= x$
Parameters $= \mu, c, \sigma$
Outputs $= y$

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

What might this look like in Wasabaye?

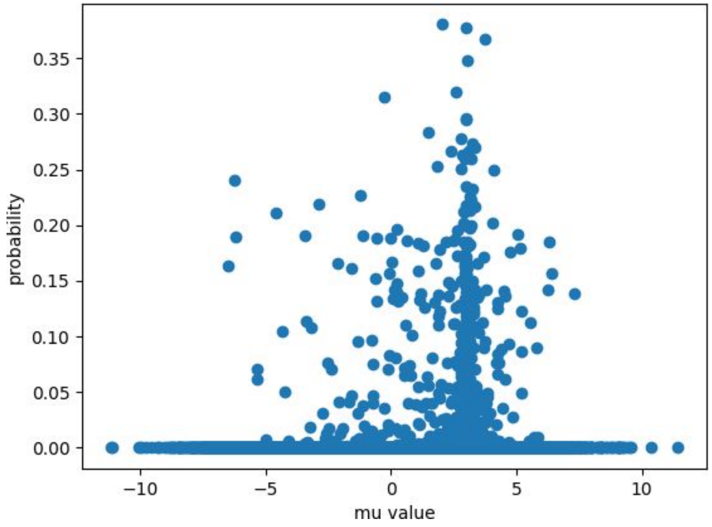
"model environment"

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

"observable variable"

Motivation 1: Multimodal models

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



Interacting with a multimodal model

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

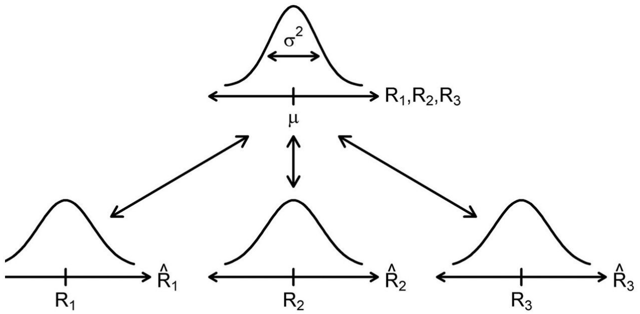
Motivation 2: First-class models

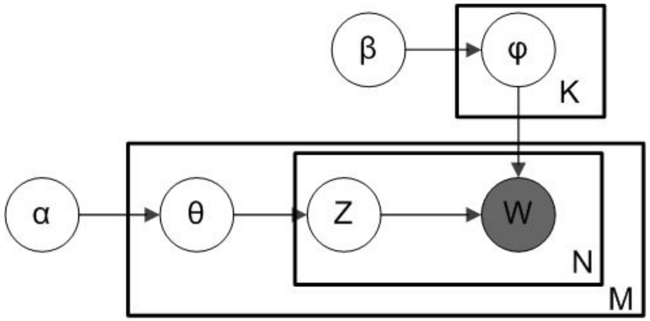
So, PPLs with multimodal models do already exist.

But models generally aren't first-class citizens.

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

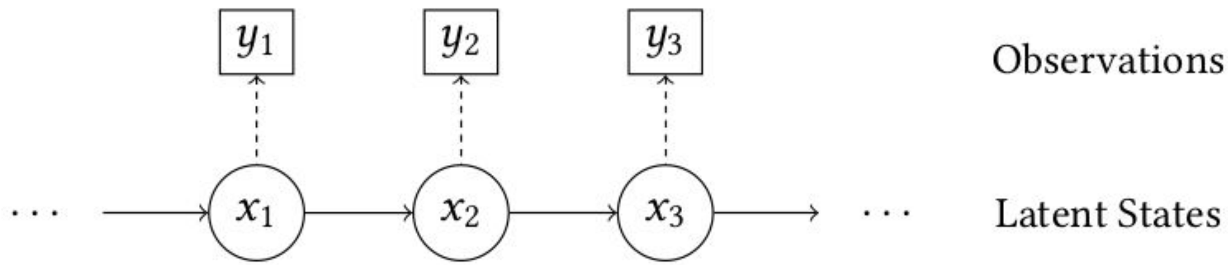
● Full support
◐ Partial support
○ No support





Compositional multimodal models

Hidden Markov Model (HMM)



We can decompose a HMM into two sub-models:

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

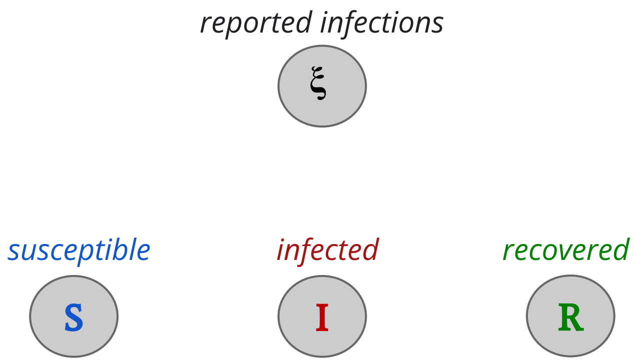
And then define a HMM as a higher-order model:

```
hmm :: TransModel env es x -> ObsModel env es x y -> Int -> x -> Model env es 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 es
b)
       -> (b -> Model env es
c)
       -> (a -> Model env es
c)
```

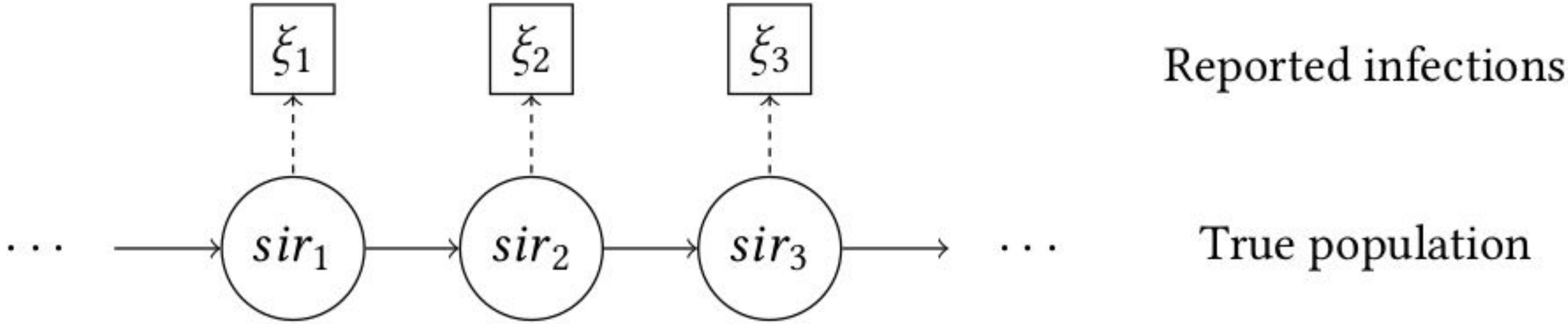
Modelling an epidemic: The SIR model

Setting: We assume a fixed population of **s**usceptible, **i**nfected, and **r**ecovered individuals.




```haskell
type ReportedInf = Int

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

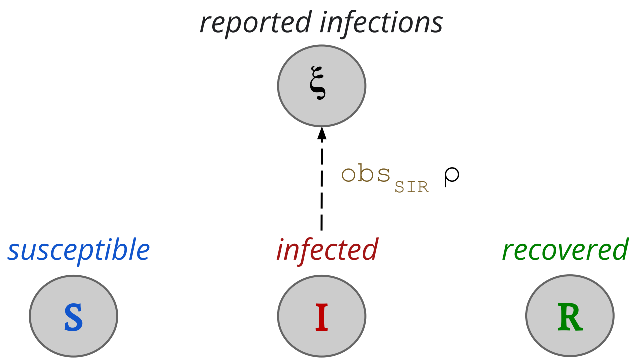
SIR Model: During an epidemic, how do these populations vary over time (days)?



Modelling an epidemic: The SIR model

SIR observation model

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

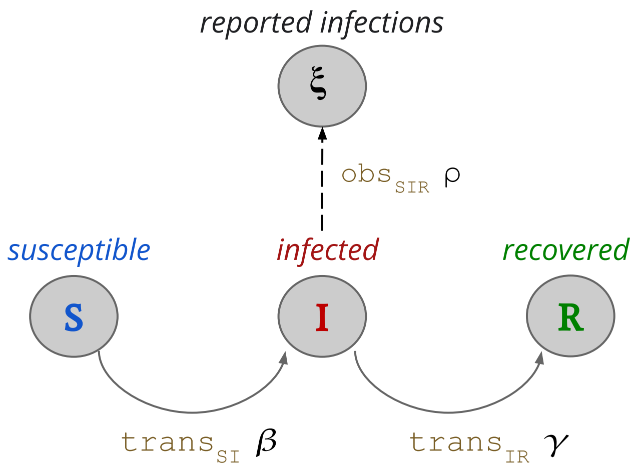



```
obsSIR :: Observable env "ξ" Int
       => Double -> ObsModel env es Population ReportedInf
obsSIR ρ (Population _ i _)  = poisson (ρ * i) #ξ
```

Modelling an epidemic: The SIR model

SIR transition model

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

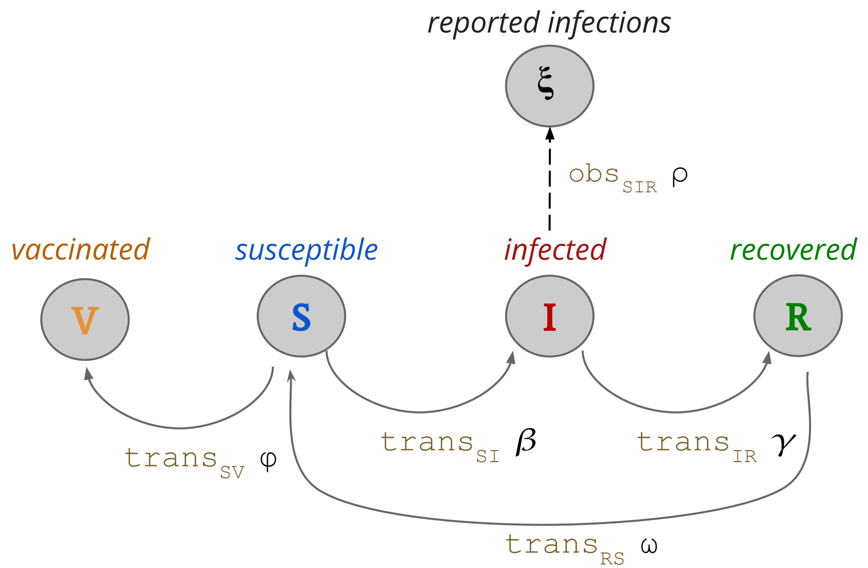


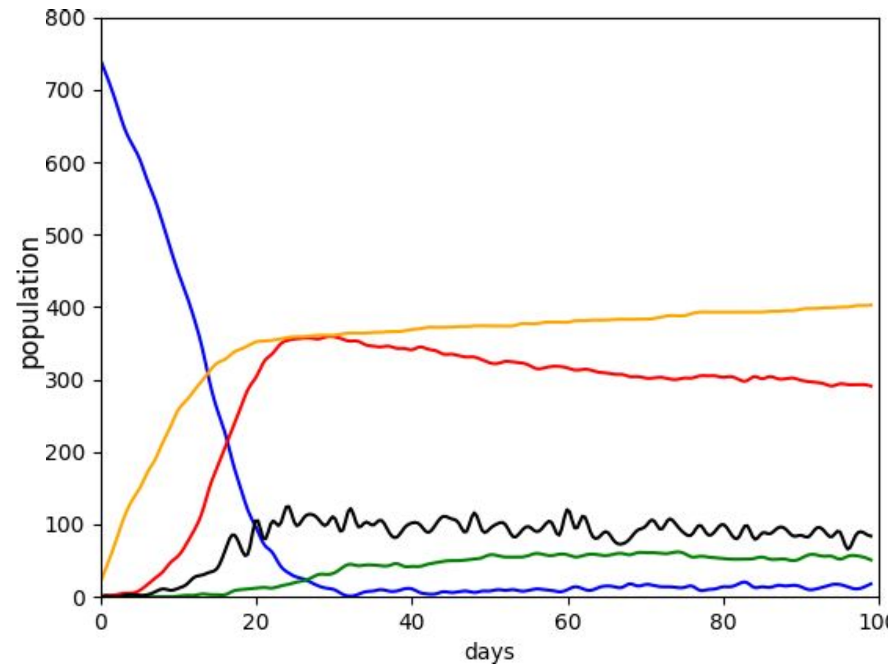

```
trans_SI :: Double -> TransModel env es Population
trans_SI β (Population s i r) = do
 δsi <- binomial' s (1 - exp (-β * i/(s + i + r)) )
 return $ Population (s - δsi) (i + δsi) r

trans_IR :: Double -> TransModel env es Population
trans_IR γ (Population s i r) = do
 δir <- binomial' i (1 - exp (-γ)
 return $ Population s (i - δir) (r + δir)
```

```
trans_SIR :: Double -> Double
          -> TransModel env es Population
trans_SIR β γ = trans_SI β >=> trans_IR γ
```

Modelling an epidemic: The SIR model

SIR as a Hidden Markov Model






```
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
```

```
let sir_0 = Population { s = 760, i = 1, r = 0 }
    n_days = 100
hmm (obs_sir ρ) (trans_SIR β γ ω φ) n_days sir_0
```

Models as Algebraic Effects

```
newtype Model env es a =
        Model { runModel :: (Member Dist es, Member (ObsReader env) es) => Prog es a }
```

Infrastructure:

A program containing *syntactic operations* `op` belonging some *effect* `E` in the *signature* `es`

`data Prog es a = Val a | Op op k` where `op : E ∈ es`

Effect 1: Primitive distributions

```
data Dist a where
 Normal    :: Double -> Double -> Maybe Double -> Dist Double
 Bernoulli :: Double -> Maybe Bool -> Dist Bool
```

optional observed value (annotating `Maybe Bool`)

Effect 2: Reading observable variables from a model environment `env`

```
data ObsReader env a where
 Ask :: Observable env x a  =>  ObsVar x -> ObsReader env (Maybe a)
```

`env` *assigns* `x` *a list of type* `[a]` (annotating `Observable env x a`)

type-level string (annotating `x`)

`#foo :: ObsVar "foo"`

Models as Algebraic Effects

Example: Desugaring models

```
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 :: (Observable env "p" Double,
             Observable env "y" Bool)
         => Model env es Bool
coinFlip = do
 maybe_p <- send (Ask #p)
 p       <- send (Uniform 0 1 maybe_p)
 maybe_y <- send (Ask #y)
 y       <- send (Bernoulli p maybe_y)
 return y
```

Model Environments

Model environments:

Extensible records from observable variables to lists of values

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

assigns `x` *list of type* `[a]`

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

```
loop n = do
 y <- normal 0 1 #y
 if n <= 0 then return () else loop (n-1)
```

Executing Models with Effect Handlers

An effect handler interprets an effect in `es`

```
handler :: Prog es a -> Prog es' b
```

Handler 1: Handling reading of observable variables:

```
handle_ObsReader :: Env env -> Prog (ObsReader env : es) a -> Prog es a
handle_ObsReader env (Op (Ask x) k) = do
   let maybe_v = getHead x env
       env'    = setTail x env
   in  handle_ObsReader env' (k maybe_v)
```

Handler 2: Handling primitive distributions:

```
handle_Dist     :: Prog (Dist : es) a -> Prog (Observe : Sample : es) a
handle_Dist (Op (Normal μ σ maybe_v) k) =
  case maybe_v of Just v  -> (handle_Dist . k) (send $ Observe (Normal ..) v)
                  Nothing -> (handle_Dist . k) (send $ Sample  (Normal ..)  )
```

```
data Observe a where
 Observe :: Dist a -> a -> Observe a
```

```
data Sample a where
 Sample  :: Dist a -> Sample a
```

Executing Models Compositionally

Interpreting multimodal models to samples and observes

```
handle_CORE :: Model env es a -> Prog (Observe : Sample : es') a
handle_CORE env = handle_ObsReader env ∘ handle_Dist ∘ runModel
```

```
coinFlip = do
 p <- uniform 0 1 #p
 y <- bernoulli p #y
 return y
```

handle_CORE →

(#p := [0.7]) • (#y := [])

```
coinFlip = do
 p <- send (Op (Observe (Uniform ..) 0.7
 y <- send (Op (Sample (Bernoulli ..))
 return y
```

Executing Models Compositionally

Interpreting multimodal models to samples and observes

```
handle_CORE :: Model env es a -> Prog (Observe : Sample : es') a
handle_CORE env = handle_ObsReader env ∘ handle_Dist ∘ runModel
```

Simulation

`handle_Sample` ∘	`handle_Observe` ∘	`traceSamples` ∘	`handle_CORE`
sample from the distributions	return any observed values	for each `Sample` operation: install then handle a `State` effect	interpret a model to `Sample` and `Observe` operations

```
traceSamples :: Member Sample es => Prog es a -> Prog (a, SampleTrace)
```

```
handle_Observe :: Prog (Observe : es) a -> Prog es a
handle_Observe (Op (Observe d y) k) = handle_Observe (k y)
```

```
handle_Sample :: Prog (Sample : []) a -> IO a
handle_Sample (Op (Sample d) k) =  IO.sample d >>= (handle_Sample ∘ k)
```

Executing Models Compositionally

Likelihood weighting

$\texttt{handle}_{\texttt{Sample}} \circ \texttt{handle}_{\texttt{ObserveLW}}\ 0 \circ \texttt{traceSamples} \circ \texttt{handle}_{\texttt{CORE}}$

($\texttt{handle}_{\texttt{ObserveLW}}$): Accumulate the log probabilities of observed values

```
handle_ObserveLW :: Double -> Prog (Observe : es) a -> Prog es (a, Double)
handle_ObserveLW p (Op (Observe d y) k) = handle_ObserveLW (p + logProb d y) (k y)
```

Metropolis-Hastings

$\texttt{handle}_{\texttt{SampleMH}}\ \alpha \circ \texttt{handle}_{\texttt{Observe}} \circ \texttt{traceLogProbs} \circ \texttt{traceSamples} \circ \texttt{handle}_{\texttt{CORE}}$

($\texttt{handle}_{\texttt{SampleMH}}\ \alpha$): selectively sample from chosen address

(`traceLogProbs`): for each `Observe + Sample` operation: install and handle a `State` effect

Executing Models Compositionally

What I’m up to

Formalising the language metatheory

So far
A lambda calculus based on algebraic effects,
extended with: random variables, primitive distributions, multimodal models.

Aims
Understanding what metatheory and properties we would like to show for this language
- (Denotational) semantics of probabilistic models under model environments:
  what underlying distributions do they denote?

Using row polymorphism to elegantly express effects and model environments

Exploring effect handlers for compositional inference

You can play with Wasabaye!

https://github.com/min-nguyen/wasabaye/

Example program

Ω = (μ : Double) · (σ : Double) · (c : Double) · (y : Double)

$\rho : \Omega$

ρ = (μ, [3]) · (σ, [1]) · (c, [3]) · (y, [])

$\rho' : \Omega$

ρ' = (μ, []) · (σ, []) · (c, []) · (y, [])

M : Double ! Dist_Ω

```
let linearRegression : Double ⟶ Double ! Dist_Ω
    linearRegression = model(x : Double).
                          (let μ ~ normal  (1, 2)         in
                           let σ ~ uniform (1, 3)         in
                           let c ~ normal  (0, 5)         in
                           let y ~ normal (μ * x + c, σ)  in
                           return y)
in  linearRegression 7
```

⇝*

M : Double ! Dist_Ω

```
let μ ← dist_normal   ((1, 2), Just 3)          in
let σ ← dist_uniform  ((1, 3), Just 1)          in
let c ← dist_normal   ((0, 5), Just 3)          in
let y ← dist_normal   ((μ * 7 + c, σ), Nothing) in
return y
```

N : Double ! Observe · Sample

```
with { return x              ↦ return x
     , dist_φ (A, Just y) k  ↦ let y' ← observe_φ (A, y)
                                in k y'
     , dist_φ (A, Nothing) k ↦ let y' ← sample_φ A
                                in k y' }
handle (linearRegression 7)
```

⇝*

N : Double ! Observe · Sample

```
let μ ← observe_normal   ((1, 2), 3)    in
let σ ← observe_uniform  ((1, 3), 1)    in
let c ← observe_normal   ((0, 5), 3)    in
let y ← sample_normal    (μ * 7 + c, σ) in
return y
```

Note: let x = V in M ⇝ $(\lambda x \longrightarrow M)\ V$

Note: the reduction shown above will only partially happen, as evaluation will get stuck on the first unhandled operation $\mathcal{E}[\text{op } V]$.